NTNU
Innovation and Creativity

Duplicate Detection with PMC -- A Parallel Approach to Pattern Matching

Robert Leland

Master of Science in Computer Science
Submission date: August 2007
Supervisor: Arne Halaas, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Duplicate detection is a complex problem at the mercy of the users who enter data you intend to use. Not all data does what you want it to, and this project will approach the problem of doing duplicate detection on very dirty data, with the use of a specialized hardware for pattern matching.

Assignment given: 05. March 2007
Supervisor: Arne Halaas, IDI

Contents

Abstract

Fuzzy duplicate detection is an integral part of data cleansing. It consists of finding a set of duplicate records, correctly identifying the original or most representative record and removing the rest. The rate of Internet usage, and data availability and collectability is increasing so we get more and more access to data. A lot of this data is collected from, and entered by humans and this causes noise in the data from typing mistakes, spelling discrepancies, varying schemas, abbreviations, and more. Because of this data cleansing and approximate duplicate detection is now more important than ever. In fuzzy matching records are usually compared by measuring the edit distance between two records. This leads to problems with large data sets where there is a lot of record comparisons to be made so previous solutions have found ways to cut down on the amount of records to be compared. This is often done by creating a key which records are then sorted on with the intention of placing similar records near each other.

There are several downsides to this, for example you need to sort and search through potentially large amounts of data several times to catch duplicate data accurately. This project differs in that it presents an approach to the problem which takes advantage of a multiple instruction stream, multiple data stream (MIMD) architecture called a Pattern Matching Chip (PMC), which allows large amounts of parallel character comparisons. This will allow you to do fuzzy matching against the entire data set very quickly, removing the need for clustering and re-arranging of the data which can often lead to omitted duplicates (false negatives). The main point of this paper will be to test the viability of this approach for duplicate detection, examining the performance, potential and scalability of the approach.

Chapter 1

Introduction

"Where is the Life we have lost in living? Where is the wisdom we have lost in knowledge? Where is the knowledge we have lost in information?"

T.S. Eliot

Today more people than ever are interconnected through the Internet. People chat, shop, research, do business, educate themselves, get entertained, find new friends and even find old friends online. Even people who do not want to use computers or the Internet find themselves almost forced into using the Internet in a case of what is approaching social Darwinism, where people who do not use the Internet will lose out. This could already be said to be the case in the business world, where people often will browse online catalogs to decide what they want and where to go before they leave the house; if a store does not have an online presence it might not even be considered as an option. With such widespread adaptation, and integration of the Internet into our daily lives there is more data than ever available, and even more data just waiting to be collected – a process made all the easier by the level of Internet penetration.

Another area that has seen great improvement is that of data storage. Since magnetic storage first became popular we have seen an exponential increase hard disk areal density [2]. This has the interesting implication that there is very little gain in deleting old data. The size of old data compared to capacity of new storage devices coupled with the increase in amount of data available makes the gain from deleting old data almost negligible in many cases.

With these two factors in mind how can the amount of data available do anything but increase?

So assuming the volume of data being stored all over will increase, what conclusions can we draw from this? Is data like money to a business – the more you have of it the better off you are? Or is it more like water in that a flood of data would just be a big mess that you could drown in if you are not careful? What are actually the implications of an increase in volume of available data? To address the questions and to introduce the problem I will be approaching I will bring up an example.

1.1 The idea

Today one of the most popular ways to make a profit on the Internet is creating something of value to people and putting it online free for them to access. The idea is then that your service or content will be valuable and unique enough that it will attract many visitors, become a popular webpage and receive lots of hits. You can then leverage this traffic in several ways to make money, for example by selling advertising space on your page, by selling merchandise related to your site, by offering premium services and content for a membership fee, or even by getting bought up by a larger company. Your idea is to create a database of music played by your users, by storing information tags and filenames on their music files. The information will be collected by plugins usable with your userbase's media player of choice, which will submit information on the tracks they play, perhaps along with data like how often they play it, what times of day they usually play it, and other data you might think is statistically relevant. Your webpage will then display some statistic of the user's music habits, along with the statistics of his friends and perhaps some global statistics.

In the theoretical parallel universe where our example takes place no such service has yet been implemented in a satisfactory fashion, so after a lot of hard work your service is up and running and gaining in popularity. You now wish to expand on the services you are offering, amongst other things you want to generate some artist-related statistics. For example you have an idea of showing artist-popularity statistics, where an artist's popularity is determined by the number of times their song has been played. You wish to have a global measure of popularity, as well as options of seeing who is popular in what countries, and perhaps trending data that can show over a time period the rise and fall in popularity of a given artist.

You wish to have as broad an appeal as possible and as not everyone listens to only the top 10 artists you will have to scour your database and generate statistics for all the artists. There will be many artists who are popular only with a certain group of people, for example an artist only popular in one country will not seem so significant on a global scale, but might be very important for the people who live in that one country. It is therefore vital that you generate statistics on as many of the artists in your database as possible, to keep the global popularity of your site.

By now your service has become popular and has been up and running for a respectable amount of time. Your database therefore contains a very large number of songs, and to generate meaningful statistics on the popularity trends of artists you will have to search all of this data for each of the artists in your database. Since your database has playing statistics for every song submitted this is no small task, but it is a one-time task that will only have to be completed once for all the past records and then at periodic intervals with the new data you receive. In this case as long as you are able to go through new data faster than you receive it you will be able to tread water.

With some form of indexing like a hash table for storing total play counts for artists, allowing you to quickly look up an artist name, ARTIST-COUNT is an example of what kind of algorithm you will need for the task of traversing the database of songs, and summing up the total number of times a song by the artist has been played in a time period.

```
ARTIST-COUNT($D$)
1  for each song $s \in D$
2        do $count[artist[s]] \leftarrow count[artist[s]] + play\_count[s]$
```

However upon closer inspections of the results this produces, if you had not already expected something like this, you notice that there is a problem with your data: Some artists are appearing more than once, see Figure 1.1.

Artist Name
The Rolling Stones
Rolling Stones
Ludwig van Beethoven
Beethoven

Artist Name
Johann Sebastian Bach
Johan Sebastian Bach
Motorhead
Motörhead

Figure 1.1: An example of some artist names you might find.

1.2 The human factor

One of the problems with a lot of the data available on the internet is that it was created by humans. There is nothing directly wrong with humans, we are simply just a good source of noise.

For very many years we have lived without computers, and still today most people should be by far more accustomed to talking to humans than to computers. The human mind is very good at pattern recognition and using contextual information to help with understanding. Because of our accumulated knowledge, past experiences and the ability to apply that to our current context it is generally understood that if someone says “Bach” or “Mozart” or “Beethoven” they are referring to the same people as someone who were to say “Johann Sebastian Bach”, “Wolfgang Amadeus Mozart”, or “Ludwig van Beethoven”. This is something we could guess out of the context without having learned the association between the short and long version of their names – for example if someone were to read a text where someone had first used the full name, but later only use the surname, we would deduce that they are still talking about the same person. Or if we only hear the name “Beethoven”, but the only name containing “Beethoven” we know if is “Ludwig van Beethoven” we will naturally assume they are both the same person.

We also have our own preferences for how things should be done, and tend to stick by them as much as possible. Some people might prefer the writing “Bach, Johann Sebastian” to “Johann Sebastian Bach”, or prefer just always using the short form because they know everyone will understand who you are talking about anyway. Lastly we might not always know how something is spelled – usually we remember words as sounds not as how they look printed on paper, or we might never have seen the word written before, only heard it.

This boils down to a number of factors that influence how something is typed, and which can lead to inconsistencies in the way an artist’s name is written:

- A different understanding of how a word or name is spelled (for example U.S. English compared to British English).

- A typographical error.
- An omission of one or several words in the artist name, for example a leading article.
- Differences in the formatting of an artist's name, using for example initials plus lastname, or lastname, comma, first name, as opposed to first name followed by last name.
- Incorrect or lacking knowledge about the spelling of a name.
- Lack of easy keyboard access to certain letters, like letters diacritical marks.

For humans a number of these factors could have influenced the writing of a name, and the person is still likely to deduce it. For computers however just the presence of one of these complicate matters a great deal.

Moving back to our example it should be easy to see that we will have to make some adjustments to how we do things. The way it is now an artist might be unfairly split into two or more different ways of having their name written, and so a user could stumble upon any of these, for example one containing a typo and thus one that has a very low popularity, and get a misleading impression.

Up until now we could have relied on something as simple as a hash table to help us add together the play count for artists, however now that we have to consider that two strings which are not equal can refer to the same artist this is no longer enough. We now have to search through the database for each artist and find all possible fuzzy matches with that artist, then determine which of them is the most authoritative representation of the artist's name. This is the problem of duplicate detection in data cleansing. A new version of Artist-count might look something like:

```
Artist-count($D$)
1  for each artist $a \in artists[D]$
2      do for each song $s \in songs[D]$
3             do if Fuzzy-match($a$, $artist[s]$)
4                   then $count[a] \leftarrow count[a] + play\_count[s]$
5                        $alt\_names[a] \leftarrow alt\_names[a] + artist[s]$
6         $name[a] \leftarrow$ Chose-name($a$)
```

We can see that the function's time complexity has gone from being $O(n)$ to $O(n^2)$, and the function to be called $O(n^2)$ times is Fuzzy-match which can easily be a very costly function call if you want to allow for spelling mistakes, reshuffling of words in artist name, variations in spelling, and omission or inclusion of words. Making this as cheap as possible by either limiting the number of comparisons you have to make and/or optimizing the fuzzy matching algorithm to be as cheap as possible while mantaining an acceptable level of accuracy is the main problem in duplicate detection.

1.3 My approach

As mentioned above there are two things to consider for duplicate detection: Accuracy and speed. The measure of accuracy in duplicate detection depends on the number of false negatives (duplicates you did not classify as such) and false positives (non-duplicates which were classified as duplicates) you get as a result. False negatives are a result of your search for duplicates being too specific, and a high number of false positivies means your search was too general. The algorithm's speed is mainly affected by the number of records you compare, and how costly these comparisons are. Generally CPUs are not able to do duplicate detection on large databases within any reasonable time, so you normally have to cut down on the number of records you compare.

This leads to a problem: How to pick ahead of time which records to compare without knowing which records are duplicates or not? Often the compromise made here is that you compare those which are likely to be duplicates. But how do you chose which are likely? Your selection can not be too costly or the whole point is lost. The problem here is that there are so many variations of duplicates – a duplicate record could be the result of a typo anywhere in a word for example, or the result of the words not appearing in the order you expect. If you are unable to catch a certain type of inconsistency then you will get an increase in false negatives as a result. I intend to get around this problem by skipping it alltogether. Instead I will search through the entire dataset for matches. This will be possible, without spending a very long time, because of a piece of specialized pattern matching hardware called the Pattern Matching Chip (PMC).

The PMC is designed to do a large number of parallel character comparisons and can use a combination of these character combinations to match more advanced patterns. The combination of the ability to do such advanced pattern matching, geared towards fuzzy matches, the speed at which these matches can be done and the ease of scalability makes the PMC a good candidate for replacing the CPU as the most important piece of hardware for duplicate detection. I intend to show that the PMC can be used as an efficient and accurate solution to fuzzy matching, and that the PMC approach scales easily and well for large datasets.

The dataset which will be used will be the freedb dataset [4]. It is a dataset, not at all unlike the fictional dataset of our example, used for labelling CDs, and consists of user-inputted artist, album and song names. To get the amount of data necessary to properly test scalability a custom size dataset will also be built, by combining random names and introducing noise in the form of common typing mistakes.

Chapter 2

Duplicate detection

"Let everyone sweep in front of his own door, and the whole world will be clean."

Johann Wolfgang von Goethe

Let us step away from my approach and get back to our example where we can have a look at duplicate detection in more general terms. Research on detecting duplicate records in a database can be found from as far back as the 1950s, when one of the major applications has been merging medical records for one individual from separate databases [15], and later duplicate detection in business databases. This section will introduce duplicate detection in a step-by-step fashion, adding functionality on our earlier trivial duplicate detection algorithm as we the need becomes aparrent, as a way of properly introducing some of the challenges encountered in duplicate detection.

Getting back to our problem at hand we know that if we were to compare each record to all the other records in the dataset this would be prohibitively expensive for large datasets. It is therefore desirable to limit the number of comparisons we have to make; we need to isolate the number of records which are most likely to be duplicates. For this to be a time-saving measure it has to be kept very simple and as a result it is not going to be very accurate. It must strike a balance between being too specific, leading to false negatives, and being too general, leading to an unnecessary amount of comparisons. Ideally the filtering will result in a very low number of false negatives, while not letting the number of non-duplicate hits get too large. Note that there is also no point in making it so general that it will catch duplicates so dissimilar or mangled that they will not be identified as such by the FUZZY-MATCH algorithm.

2.1 Finding duplicates

The common approach to eliminating duplicate data where two records are identical – they are simply duplicates because they were added twice not because of inconsistencies – is to sort the data lexicographically. This guarantees that entirely identical records end up next to each other. Also records which are duplicates, but not identical, might end up very near each other if the are duplicates due to a typo which does not appear too early in the data. A typo

Insertion	hop → hope
Deletion	lady → lad
Substitution	ball → fall

Figure 2.1: The operations used in the Levenshtein algorithm.

in the first character would, for example, place the record very far from its duplicates. Sorting it lexicographically will also not help at all in catching duplicates which are duplicates due to the names appearing in different order, or adding or removing of names, but it is a start.

Sorting the data lexicographically is only half the job. Now that the records that might be duplicates are near one-another we need to find out whether they actually are duplicates. Since we have determined this approach will mostly only catch duplicates due to small inconsistencies in spelling we need an approach that can determine whether one word is likely to be a typo of another. The common way to measure the similarity of two strings is by edit distance, which is defined as the number of operations needed to transform one string to the other. A popular measure is the *Levenshtein distance* [1], where operations can be insertions, deletions or substitions as demonstrated in Figure 2.1.

A basic Levenshtein algorithm might look as follows:

```
LEVENSHTEIN($A, B$)
1  for $i \leftarrow 0$ to $length[A]$
2      do $d[i, 0] \leftarrow i$
3  for $j \leftarrow 0$ to $length[B]$
4      do $d[0, j] \leftarrow j$
5  for $i \leftarrow 1$ to $length[A]$
6      do for $j \leftarrow 1$ to $length[B]$
7             do if $A[i] = B[j]$
8                   then $cost \leftarrow 0$
9                   else $cost \leftarrow 1$
10                $d[i, j] \leftarrow$ MIN(
                          $d[i-1, j] + 1$,
                          $d[i, j-1] + 1$,
                          $d[i-1, j-1] + cost$
                          )
11 return $d[length[A], length[B]]$
```

LEVENSHTEIN returns the number of insert, deletion and substitution operations required to change string A into string B. Note that one transposition (*rob* to *orb*)operation counts as two operations in the Levenshtein algorithm. There is a variant of the Levenshtein Distance algorithm called the Damerau-Levenshtein distance algorithm, which will count transposition as only one operation. It is, however, slightly more complex without adding anything to the discussion of duplicate detection in general, so we will not be covering it here.

Using LEVENSHTEIN we might define FUZZY-MATCH simply as:

```
FUZZY-MATCH($A, B$)
1  if LEVENSHTEIN($A, B$) > THRESHOLD
2      then return FALSE
3      else  return TRUE
```

Note that the minimum edit distance between two strings, A and B, is the difference in size between A and B. So to further optimize the above you can first check whether the difference in size between A and B exceeds the threshold. This is likely to rule out some records without having to call LEVENSHTEIN. THRESHOLD is a suitable cut-off point for the acceptable Levenshtein distance between two duplicates. A good value for this will be lenient enough to allow the most common typographical errors, but strict enough to not match two different artists as duplicates. In our example we might want to use a value which depends on the length of the strings to be compared, as the length of the string will affect the number of inconsistencies which are likely to be present.

Using the Levenshtein distance, if you have attempted to check whether B is a duplicate of A, you need not check whether A is a duplicate of B, as the Levenshtein distance is symmetrical; B being a duplicate of A implies that A is a duplicate of B, and vice versa. Because of this, if you are starting on one end of the data and moving through it you need only check each record against records below it. How many records to check depends on how similar your (non-duplicate) records are, where two records in this case might be similar if the first few letters in each are the same. In cases where you have many similar records a duplicate would end up further away from the master record, particularly if the inconsistency appears early in the string, because of the lexicographical ordering. To reduce the number of false negatives in those cases you would have to increase the amount of records you check, however at the cost of speed.

2.2 Merging duplicates

After deciding which records are duplicates and which are not you have to store this information for later use to avoid having to re-do the detection for all the old records every time you want to update the popularity statistics with new data. The most straight forward way to do this is to give each artist an ID, and then give duplicate records the same ID, like in FIND-DUPLICATES. Note that while it would be possible to simply overwrite the artist name on a duplicate record with the correct name the fact that the name is a duplicate is valuable information. Not only would the duplicate record simply be re-created next time the track with those tags is played again, but, while the space of strings which might qualify as duplicates is large, a large percentage of actual duplicates will come from only a small percentage of these strings. This is partly due to the mechanics of our example – the information is gathered from tags which might for example have been applied to an entire album and thus a typo in artist name would propagate through all tracks on the album. And partly due to the nature of the mistakes: A lot of misspelled words will be mis-spelled for a reason, for example due to a user using his native language's spelling on a non-native name.

```
FIND-DUPLICATES($D$)
1  LEXICOGRAPHICAL-SORT($D$)
2  for $i \leftarrow 0$ to $size[D]$
3      do if $id[D[i]] = \text{NULL}$
4            then $id[D[i]] \leftarrow i$
5          for $j \leftarrow 1$ to WINDOW_SIZE
6              do if FUZZY-MATCH($D[i], D[i+j]$)
7                    do if $id[D[i+j]] = \text{NULL}$
8                          then $id[D[i+j]] \leftarrow id[D[i]]$
9                          else $id[D[i]] \leftarrow id[D[i+j]]$
```

You can see here on line 4 that a new id will only be given to record which has no id – a record would have been previously given an id if it was a duplicate of a previous record. Not only is this necessary for the algorithm to actually function, but it means records classified as a duplicate of a duplicate will be given the id of the earliest occurrence of that artist name. Combined with line 7, which will ensure that if the record being matched against has already been classified as a dupe earlier the current record gets its id, we have something which in most cases is the transitive closure. This is useful because it effectively extends the window size only in cases where there are a lot of duplicates (something you might see for very popular artists with complicated names) without knowing anything about which records have a lot of duplicates beforehand and without having to increase our WINDOW_SIZE for all records just for the few records with a high number of duplicates. The only case we would not have the transitive closer here is if we have a case where we have, for example 4 records in the order A, B, C, D, and A and C is first classified as duplicates, then B and D, then C and D. Which would leave C marked as a duplicate of D and B, and no longer of A. A would end up being listed as having no duplicates.

This is unlikely to occur very often, but you can get around the problem by storing a table of links between IDs where a link is formed if two records which both have separate IDs are classified as duplicates. Once the FIND-DUPLICATES is done you can then go through the table with links and give all groups of records which are linked a common id.

ARTIST-COUNT can then be updated to use the IDs from FIND-DUPLICATES instead of artist names:

```
ARTIST-COUNT($D$)
1  FIND-DUPLICATES($D$)
2  for each song $s \in D$
3      do $count[id[s]] \leftarrow count[id[s]] + play\_count[s]$
```

What you have after running this is IDs each representing one or more artist names which are duplicates of one-another, and a rating of how popular each ID is. The next step you would want to take is find the best representative of the artist's name – the string most likely to be the correct representation of the name. In our example this is not very difficult. We can assume that the artist name which occurs the most often in songs is the correct one, and even if it is not the most popular spelling of an artist's name would be the one you would want to use on your statistics because this is the one likely to be most familiar among your users. Then it is merely a matter of, for each id, counting the number of occurrences of each unique variation of the artist's name, and picking the one which occurs most frequently.

This task will of course not always be this trivial. If you for example had only a database of artist names, not of a combination of artist names and song titles, then you would likely not store the same string more than once, and thus each unique occurrence of an artist's name would occur only once. There would be a number of approaches to selecting a representative name in this case, and generally the fewer duplicates there are the less certain you can be which record is the best representative (on the other hand with fewer records you are more likely to pick the correct one by pure chance). For example:

Pick the record closest to all other records For each record measure the Damerau-Levenshtein distance between it and all other records in a group and pick the record with the lowest average Levenshtein distance to the other records in the group.

Construct a mean sample First find the mean size for the artist name, and then, for each position in the artist name, check the letter at the given position in all records and pick the letter that occurs most often. In the case of duplicates bigger or smaller than the mean size you could ignore any excess inserted letters or insert a dummy letter in place of a deleted letter (relative to a record of mean size).

These suggestions work because a large percentage of human spelling mistakes can be expressed as one of the four operations in the Damerau-Levenshtein algorithm mentioned earlier [3], meaning if two duplicates are separated by a distance of more than one, e.g. two, it is likely the intermediate step is the correct record – which would have a distance of one to each of the two records.

The suggestions of course rely heavily on the assumption that the only duplicate records found are those created by simple spelling mistakes. Once you begin looking for other types of duplicates these will be insufficient. One approach that might be attempted is a generalization of the first example where you have a measure of how likely it is that two records are duplicates, based on the number of transformations it would have to go through where for example each operation in the Levenshtein algorithm is one transformation, and swapping names around is another operation, deleting and adding words are also each one operation. In other words you use a combination of Levenshtein on a word and name level. Then pick the record which has the highest average likelyhood of being a duplicate when compared with the other records in the group.

2.3 Dealing with new data

The last challenge is the arrival of new data. As time goes on and your database gets new data added to it you will want to generate updated statistics. For this purpose it is important to save which records are duplicates so for new records you can check if the artist name belongs to any of the duplicates known from earlier runs. If the artist has not been previously entered in the database, assuming we are still using the method of sorting the data lexicographically, we insert the artist in its correct position in a sorted database with the older artists, and then check it against WINDOW_SIZE records above and below it as those are the records it would have been compared to if it had been part of the data in the last search. If it does not match with any records above or below it is given its own id and the search continues with the next fresh record.

Depending on how often you upgrade statistics and what order you would normally want your data in, you could make a time-memory trade-off by storing two copies of the database – keeping one of them sorted lexicographically on artist name and one of them in your preferred order to avoid having to sort it every time, which might be especially useful for larger datasets where the amount of time spent sorting can be very noticeable, particularly if it is not possible to keep all the data in memory and you have to resort to swapping.

This demonstrates the basic theory behind general duplicate detection, and the problems you come across trying to solve it. Since the main focus has been an introduction to the problems faced in duplicate detection and a simple example of an algorithm one could use it has several shortcomings. Amongst others the ability to only catch duplicates resulting from a small number of spelling variations none of which occur early in the name, and several algorithms where corners can be cut and optimizations made to increase efficiency.

Chapter 3

The Pattern Matching Chip

"The jack-of-all-trades seldom is good at any. Concentrate all of your efforts on one definite chief aim."

Napoleon Hill

The Pattern Matching Chip (PMC) (Figure 3.1) is a piece of specialized search hardware developed by Interagon AS [7]. It has been designed with a focus on complex pattern matching in large bases, the ability to run parallel searches, and scalability. The philosophy of the PMC is that if the number of PMCs you are using is not enough for your purposes, you simply add more, and it scales trivially and without extra overhead. A PMC contains 1024 processing elements (PEs), which are designed to do single character comparisons, and 16 of these PMCs are stored on one PCI card along with 128 MB of local SDRAM (with support for larger SDRAM chips) per PMC chip (total of 2 GB). Each PMC is capable of performing $1.024 * 10^{11}$ character comparisons per second, and allow searching for up to 64 patterns in parallel with a throughput of 100 MB per second [5]. Additionally one PCI card with 16 PMC chips consumes only 25 watts, so as one node can hold up to 6 PCI cards (96 PMCs), a cluster of 100 nodes can achieve a theoretical bandwidth of 1 TB/s requiring only 35 kilowatt. The performance of PMCs have been shown to adhere closely to the theoretically max value in [5].

The main strength of the PMC is in its raw searching power and the cheap cost of doing complex pattern matching. This makes it suitable for searching through large amounts of data where indexing is not viable, for example because the nature of the data or the searches makes indexing hard or pointless or the data will be searched only a very limited amount of times so creating indexes would be more costly than the searching itself, for example in monitoring a stream.

3.1 Searching

A search is performed by loading data into the SDRAM on the PCI card, configuring the PMCs with what character comparisons the PEs are supposed to do and the patterns of matches to look for, and finally the data is loaded and distributed amongst the PEs, using a binary distribution tree. Each PE does

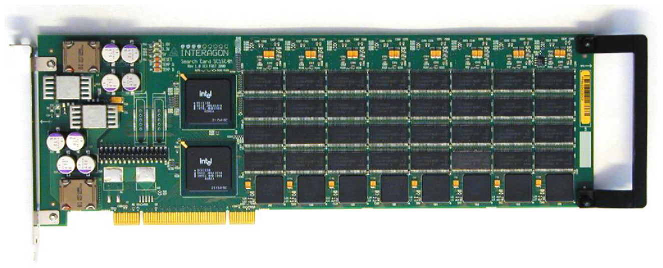


Figure 3.1: PMCs housed on a PCI card.

single character comparisons as the data goes through it and the results of the character comparisons are combined and compared with the patterns loaded through the configuration file. If a match is found the location in the dataset is reported back to the host machine.

The part of the PMC responsible for reporting hits back is the *hit manager*. It can report results as an offset in bytes from the start of the data source, or as a document number the hit occurred in. The documents are determined by the *document manager*. The document manager scans the data for a document separator, which can be up to 4 characters long, and counts up the documents. The PMC can be set to only report one hit per document when doing a search, so if you are searching a database you would normally split it up so each record is a document, then the PMC would report the record number where the hit was found. Additionally there is a byte remapping table in the chip, through which all data gets passed before it gets passed to the document manager. The table will then re-map each character in the data according to the table. This allows you to, for example, map all upper- to lower-case letters (or vice versa) if you are not interested in case sensitivity.

When doing searches the design of the PMC lends itself to two distinctive cases: Static config and static data searches. In a static config search you know ahead of time how many queries you will be searching for, and what they are, but the size of the data source is unknown. The query source is exhausted and the PMC is configured with the queries. Then data is read from the source, loaded into the PMC and a search is run. The hits are reported back and once the search is done, if the data source is not yet empty more data is read and loaded into the PMC, while the configuration stays. An example of this type of search being put to use would be searching in a stream where the PMC listens in on the stream and constantly checks for matches.

A static data search is when the size of the data and its contents is known beforehand, but the number of queries is unknown. For this search the data source is first exhausted and data is loaded into PMC memory. Queries are then read from the query source until no further queries are available (either because the source is blocking or it is empty), and the PMC is configured with the read queries and data is distributed to the PEs. Once the search is done new queries are read and the search is done again, without the overhead of having to load the data back into PMC memory each time. An example of this type of search might be using the PMC to interface with a database. A program could run which would accept input from users in form of queries. Meanwhile the PMC would be looping the search only stopping to read new queries between

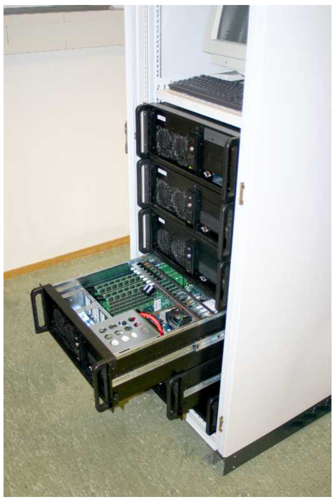


Figure 3.2: A node containing the PCI cards.

each search. This would be useful in a scenario where you are working against a database where indexing is hard or impractical.

If you have too many queries to run, or too much data, you can simply add more PMCs to speed up the process. For example if you are using one PCI card (16 PMCs) to search a dataset, but you get a batch of new data causing your dataset to double in size, if you add one more PCI card you can spread the data across more PMCs and run the queries in parallel on both PCI cards, in the same time it used to take to do a search on a dataset half the size on one PCI card. Likewise if your data fits on one PCI card but you have a very large volume of queries you can use more PCI cards, load one copy of the dataset to each card and distribute the queries across all the cards.

3.2 The Interagon Query Language

The Interagon Query Language (IQL) [6] is a regular expression-like language used to expresses the pattern matching available through the PMC. In the cost of the, queries available the power of the PMC can be demonstrated as a lot of matching operations, which would be complex operations on a CPU are very simple on the PMC requiring a low number of PEs. This is due to the way the PMC searches, with PEs doing parallel character comparisons and combining the hits (or absence of hits) to form complex patterns which would require dynamic programming on a CPU.

The simplest example of IQL would be a string matching search, expressed:

```
‘‘Eric Clapton’’
```

The quotes denote that the entire string is one query – if not space will be treated as a query delimiter. The string is of length 12, however because of the architecture the smallest number of PEs that can be reserved is 16, and so the pattern would require 16 PEs – a string match for a string of length 17 would require 32 PEs, as anything more than 16 would require you to go one level further up the tree, doubling the number of PEs reserved. This is hardly putting the PMC to good use, however.

A number of mistakes are caused by orthographic gemination, particularly so in artist names where whether phonetic gemination is reproduced in spelling or not is left in the hands of the artist. The `+` operator can be used, like in regular expressions, for positive closure. For example you might replace the last consonant of a gemination with this operator, to allow for matching one, two or even three or more repetitions of the consonant:

```
``Basement Ja''x+
```

Note that the ending of the quotes here was necessary because the `+` operator would be interpreted only as the character "+" if it were inside quotation marks. This has no effect on the query since it is merely there to clarify that the whitespace is a part of the query; ``` ``ja''x ``` and `jax` and ``` ``jax'' ``` will match the same records and require the same number of PEs (even if they were not all rounded up to requiring 16 PEs each). As an added benefit this positive closure operator does not increase the amount of resources used. This is handled by the reading of matches from the PEs, not by the matching at PE level, so you get that extra matching for free; a string of length 16 with the operator affixed to it would still only require 16 PEs.

Another mistake could be mixing up spelling like -ize and -ise, so you might look for words ending in said suffixes and generalize them with the character set operator:

```
Spirituali[zs]ed
```

This would match both "Spiritualised" and "Spiritualized". As one might expect matching both -ize and -ise requires one more PE than matching only one of the two, as one PE has to look for 'z' and the other for 's'.

There are many such operators, very similar to what is found in regular expressions, and they can all be found in the reference guide[6]. What these basic operators have in common is that they are all generally cheap, costing little or no extra PEs and so you can attach them should you so desire and have little or no effect on the resources required. However, fuzzy matching is very general in nature, and chances are you would want to use more general matches than those presented above.

The *character-displacement* operator allows you to specify how large a displacement (to the right) you want to allow each character relative to its position in the query. This can be very useful as it allows arbitrary insertions in the middle of the pattern you are trying to match, one of the basic typographical errors, and still matches the pattern.

```
{Queen :  c = 1}
```

The syntax used here is a bit different than earlier. The curly brackets are used to group the pattern so as to know what the operators applies to. The colon marks the end of the pattern and the beginning of the operators. In this case the operator is `c`, character-displacement, which has the value of equal to one. This means that the pattern "Queen" is to be matched, allowing individual character-displacements 1 position to the right of their original position. In other words the above pattern will match "Queen" and "Quieen" and "Qwueen", as in the latter two some of the characters have simply been displaced one position to the right by the inserting of a new letter. Since matching for character displacement is a matter of allowing the data to shift it does not require extra PEs for low values of `c` (< 3), and for the purpose of duplicate detection this is often sufficient since large values of `c` tends to make the query very general very quickly.

A second operator which is very interesting for the purposes of duplicate detection is the *pattern N-of-M* operator. This allows you to specify that only N of M specified patterns need be matched, or you can even specify that only N of M letters in the pattern need match:

```
{‘‘Led Zeppelin’’ :  p >= 10 }
```

Again here the quotation marks mean nothing more than that the white-space is to be treated as part of the pattern and not a delimiter. As the length of the pattern is 12 characters saying that `p` is to be greater than or equal to 10 (10 or more characters need to match) 2 of the characters need not match. So "Lep Zeppelin" would match, as would "Led Zeddelin". Again this is an operator which is free of cost, as the design of just looking to the PEs for matches allow you to declare a match when only part of the PEs have flagged a match, as opposed to all.

There are also a number of operators which work on patterns. There are the common logical operators: `and`, `or`, and `not`. These all match on a document level, so for example

```
Elvis and Presley
```

would return a match for any documents where both the pattern "Elvis" and the pattern "Presley" occurs. This would include for example "Elvis Aaron Presley" and "Presley, Elvis", so you can see that this could be very useful in duplicate detection, and again this comes at no extra cost. The cost in PEs of `A and B`, where `A` and `B` are patterns, is the cost of pattern `A` plus the cost of pattern `B`.

Additionally there is the `near` operator. It works like `and` except that it allows you to specify how far apart the ends of two given patterns should be to allow a match. For example

```
Elvis near(12) Presley
```

would match "Elvis A. Presley", but not "Elvis Aaron Presley", and also, like `and`, comes at no extra cost.

Lastly I would like to mention an operator similar to the previous pattern N-of-M operator, simply called the N-of-M operator. This operator returns a match if a certain number of sub-expressions are matched, allowing you to do, amongst other things, n-gram based searches:

```
{Pr, ri, in, nc, ce :  n >= 4}
```

Which would match any document where four of the five 2-grams occur. This operator, like the pattern N-of-M operator, does not add any overhead costs. Note though that n-gram matching itself requires more character comparisons than string matching, so there PEs required will be decided by the total sum of characters from all the n-grams. Thus using this operator for the sake of n-gram matching would increase the number of PEs required compared to string matching, but as a result of the operator itself.

A summary of the examples provided is available in Figure 3.3.

	Query	**Matches**	**Cost**
Positive closure	“Basement Ja”x+	Basement Jaxx	none
Character set	Spirituali[sz]ed	Spiritualized	none
C-displacement	{Queen:c=1}	Quieen	none c<3
Pat-nofm	{“Led Zeppelin”:p>=10}	Led Zeddelin	none
Logical and	Elvis and Presley	Presley, Elvis	none
Near	Elvis near(12) Presley	Elvis A. Presley	none
N-of-M	{Pr, ri, in, nc, ce:n>=4}	Princ	none

Figure 3.3: Some examples of IQL queries, their behavior and their cost in addition to the number of characters in the pattern.

3.3 Search speed

So the PMC is fast, it can do a large number of character comparisons per second, it can do queries in parallel, it scales easily, and it can match very complex patterns without taking much of a penalty. But what are the implications of this?

When loading the data into the PMCs you can split the load evenly amongst all the PMCs. Given that you have M PMCs available to you with s_{mem} MB of memory each, s_{data} MB of data, and assuming that $M * s_{mem} >= s_{data}$ then each PMC will have

$$D = \frac{s_{data}}{M} \tag{3.1}$$

MB of data stored on it. One search through the data will then take

$$t_s = \frac{D}{\theta_{max}} \tag{3.2}$$

seconds, where θ_{max} is the bandwidth of one PMC (100 MB/s).

The same configuration will have to be distributed amongst all the PMCs, since each PMC holds a chunk of the data to be searched on, so the number of queries per search, assuming the average number of PEs required by a query is q_{pe}, will be

$$N = \frac{n_{pe}}{q_{pe}} \tag{3.3}$$

where n_{pe} is the number of PEs per PMC (1024).

Given then that you have n_q queries to get through, the number of searches that will have to be run is

$$n_s = \frac{n_q}{N} \tag{3.4}$$

and the total time taken for the search will be

$$t_T = n_s * t_s \tag{3.5}$$

While one search is running the configuration for the next search is uploaded, given that the time to upload a configuration is t_c the above calculations assume that $t_s \geq t_c$. t_c can be expressed as

$$t_c = \frac{M * C}{S} \tag{3.6}$$

where C is the size of the configuration that has to be loaded, and S the PCI bandwidth. A node with 96 PMCs and PCI bandwidth of 50 MB/s will have a t_c of 30 ms, so D has to be larger than or equal to 3 MB[5] to take advantage of the PMC speed. If your dataset is not large enough for this you can load it more than once in parallel and run one set of separate queries on each of the loaded copies.

Consider a dataset 2 GB in size. Let us say the dataset is one containing artist and song names, from our example. The average artist name might be about 14 characters long, and the average song name 22 characters long (data based on freedb.org dataset[4]). This leaves us with an average record length of 36 characters, and so

$$\frac{2 * 2^{30}}{36} = 59652324$$

records in our file assuming one byte characters (rounded to the nearest whole record).

In our example we are interested in matching based on the artist, and assuming we use only operators which do not add much or anything in terms of cost the PEs required for a record with an average sized artist name would be 16. However it would be incorrect to assume this means the average number of PEs required by a query is 16; those artists requiring more than 16 characters would lead to a doubling, and all below the average would not pull down the average number of PEs. Using the freedb.org dataset again the average number of PEs required per query, assuming an additional of 2 PEs needed per query by operators, is 22.4.

In a node with 6 PCI cards we would have $6 * 16 = 96$ PMCs, so from equation 3.1 each PMC would store

$$\frac{2 * 2^{10}}{96} = 21.33$$

MB of data, meaning each search will take

$$\frac{21.33}{100} = 0.21$$

seconds, from equation 3.2.

Since the dataset is only loaded once all our PMCs need a copy of each configuration, so for each search we can fit 1024 PEs worth of queries in, which is

$$\frac{1024}{22.4} = 45.71$$

queries (equation 3.3) on average.

Assuming we have done any pre-processing we might have wished to do and that the 2 GB of data we have is the remaining data where all artist names are unique, the time taken for the search, from equation 3.4 and 3.5, would then be

$$0.21 * \frac{59652324}{45.71} = 274054$$

seconds. Or in other words 3.17 days to do almost 60 million fuzzy matches, each against another 60 million of records in a dataset of 2 GB. If a search will take too long you can simply add more PMCs. The time will decrease linearly as you add PMCs – doubling the number of PMCs halves the amount of time needed.

Chapter 4

Previous work

"I say with Didacus Stella, a dwarf standing on the shoulders of a giant may see farther than a giant himself."

Robert Burton

As mentioned earlier, the traditional way to detect identical duplicates is to sort the data in some fashion, and for each record compare it with its neighbors until a record is found which is not an identical duplicate [8]. As seen you can expand on this for approximate duplicate detection. It is done to get clustering of likely duplicates, for then to do a more accurate neighborhood search for each record [10, 11]. The key to be sorted on is usually picked that gives the greatest likelyhood for duplicates to appear near one another. The pairwise fuzzy matching usually relies heavily on string matching algorithms, a problem that has been around for a long time and for which there has been done a lot of work [17, 18, 19].

When chosing which neighbors on which to do pairwise comparison a window can be used [12], of a fixed window size W. For each record number i, if $i < W$, then i is compared with the records $[0, i)$, and if not then it is compared with records $[i - W, i)$. Meaning the total number of comparisons to be performed for a database with N records is $O(NW)$. While this increases linearly for a constant W, a larger N often means you need a larger W or suffer a loss of accuracy – the more records you have the more likely it is that other records end up between duplicate records after sorting.

Since just one search with one pass will not normally be able to catch a very large percentage of duplicates it is possible to use several passes of the data for more accurate duplicate detection [10, 9]. If you, for example, were to do one pass with a window size W and compare it with doing two different passes with a window size of $\frac{W}{2}$ the latter would usually be more accurate without having to do any more comparisons (outside of the extra cost of a second sort).

[12] expands further on this by taking the transitive closure of the record links from several passes. This mostly increases accuracy by grouping together more duplicates than what you wouldd normally achieve without closure – if B is found to be a duplicate of A, and C is found to be a duplicate of B, then, after transitive closure, C will be classified as a duplicate of A.

In [13, 16] they introduce a general algorithm, based on Smith-Waterman, which can be used for very general, domain-independent approximate matching.

They also use an algorithm determined to reduce the number of comparisons that have to be made. It works by keeping a given number (4 in the paper) of the last few discovered sets of duplicates in memory, and for each record checking if it belongs to either of the sets. This is done by comparing the new record with all members of each set and stopping when a match is found, or moving on to the next set if they encounter a *bad miss* – where the fuzzy match comparison scores very low. Lastly they also present an algorithm for doing transitive closure incrementally.

Doing duplicate detection alone is not always enough, often results can be improved by scrubbing the data before doing duplicate detection; [14] takes a look at the possible ways to scrub data to improve accuracy. This includes correcting common typographical mistakes and fixing schema inconsistencies, relying on external data and knowledge of the data to be scrubbed to correct mistakes. They also apply field-weighting to their duplicate detection, where the similarity of some fields have an increased impact on deciding whether two records are duplicates.

Transitive closure is not always a good thing. In some cases two sets of duplicates might not be very far apart, and so by chance they might each have one record with a typographical error that brings them just close enough for them to be classified as duplicates, and then transitive closure will merge the groups, giving inaccurate results. A way around this is discussed in [20] where they use an algorithm based on the certainty of the two groups being duplicates and compare this value to a threshold to determine if they should be merged.

A lot of earlier duplicate detection algorithms have relied on being able to sort on a field that has similar properties to a primary key field, a field that is distinguishing for that one record. If instead no field itself is unique or distinguishing, and the only way to determine if two records are duplicates is to look at a number of fields, one way around this could be using the K-way sorting method from [21]. For this method the sorting is done on a combination of fields, instead of just one. Often there will be several different combinations of the same or varying fields, and one pass will be done for each combination.

There has also been a number of frameworks for duplicate detection created, for example IntelliClean [20], which is a rule based approach with four sets of rules:

Duplicate identification Rules which say how to determine what records should be classified as duplicates.

Merging/Purging Which specify what to do with duplicate records when they are found.

Updating Rules for when you want to update the data with an integrity constraint or to fill in missing values.

Alerts These can be set to notify the user if a given situation occurs, allowing for the user to manually intervene in the cleaning process.

Chapter 5

My Approach

"Computers have enabled people to make more mistakes faster than almost any invention in history, with the possible exception of tequila and hand guns."

Mitch Ratcliffe

There are two main concerns with duplicate detection: Resources and accuracy. Under resource concerns we find factors like:

- Efficiency of the algorithm itself – how long time it will take for the algorithm to finish running.
- Manpower spent:
  - Manual interaction with the duplicate detection algorithm.
  - Studying the dataset to learn its traits like where mistakes are most likely to appear, what kind of mistakes are common, and what fields can be thought of as representative of a record.
  - Writing rules and algorithms specific to the type of data to best clean it.
- Computer power needed.

And for accuracy we are mainly concerned with:

- False positives.
- False negatives.
- Correctly identifying which record is the best representative of a set of duplicates.

Usually we strive for a healthy balance amongst these:

- Increasing accuracy in an automated algorithm would increase the time taken to complete the duplicate detection.
- Decreasing time without a decrease in accuracy might require an increase in computer power.
- Increasing accuracy without an increase in algorithm cost would likely require more human intervention of some sort.

Record	**Key 1**	**Key 2**	**Key 3**
Pink Floyd	pifl	pkfd	nkyd
Pimk Floid	pifl	pkfd	mkid
Punk Floy	pufl	pkfy	nkoy
Oink Fkloyd	oifl	okfd	nkyd

Figure 5.1: An example of how 3 passes with 3 keys could catch typing mistakes. Key 1 is the two first letters of each word, key 2 is the first and last letter of each word, and key 3 is the last two letters of each word.

5.1 Problems and Challenges

Usually the challenge in improving accuracy is not in improving the pair-wise fuzzy matching algorithm. While there is something to be said for improving the accuracy of the fuzzy match without making it prohibitively expensive, it is fairly limited by the clustering algorithm. This is because the clustering algorithm is what records will be compared in the first place, and for sorting this is usually limited to records which are very similar in terms of some measure, as is necessary to keep the duplicate detection from taking unreasonably long time. As such there is rarely any point in improving the accuracy of your fuzzy matching duplicates to match records your clustering is unable to recognize.

If you wish for your clustering algorithm to catch a larger variation of duplicates this generally requires more passes and more sorting. Sorting in itself, compared to duplicate detection, might not be that costly, but for very large datasets it could be, especially if you have to sort several times. Additionally there are some cases which can prove hard to properly cluster.

Consider our previous example with the music database. It might be easy to do data cleansing in a database where each field is properly labled and all data is inputted in the correct fields, but in our example we are looking for songs featuring the same artist; finding duplicates that are caused by simple typographical errors is the smallest of problems.

If you are clustering by sorting based on some key generated by a hasing algorithm you might generate one from the first two letters of each word in the artist field. This would ensure that records where the first few letters in each word are similar will end up near each other. If the typo is not in the first two letters this will be fine. To catch those extra ones where you were unlucky might do a second pass that sorts on a key based on another selection of letters. This should catch most duplicates caused purely by typographical inconsistencies, see figure 5.1. On a side-note the larger the dataset is the more letters you will need to pick – if you pick too few too many records will have the same key and your window size will have to be too large. However picking more letters means you need to do more passes for the same accuracy since there is a greater chance of an error propagating to the key. Also note that the way you pick letters is important, if you decide to pick “the 4^{th}, 5^{th} and 6^{th} letter from the left” then an insertion or deletion error early in the word will make all letters picked from that word wrong.

As you can see it does require a few passes, but eventually you should have gotten most duplicates caused by typographical errors. You will however have problems if you have insertions or deletions of words. This could particularly

Record	Key 1	Key 2	Key 3	Key 4
The Rolling Stones	throst	tergss	henges	tsorht
The Rollig Stpnes	throst	tergss	heiges	tsorht
Rolling Stones	rost	rgss	nges	tsor
The Beatles	thbe	tebs	hees	ebht
Beatles	be	bs	es	eb

Figure 5.2: The effect of missing out a word on the keys. Notice how Key 1, 2 and 3 from figure 5.1 can not be used, but one could perhaps use Key 2, Key 3 and Key 4 instead, as long as the window size is increased.

Record	Key 1	Key 2	Key 3	Key 4	Key 5
Nat King Cole	nakico	ntkgce	atngle	ocikan	nale
NatKingCole	na	ne	le	an	nale

Figure 5.3: Key 5 is an example of a key that could be used to cluster adjacent duplicates where spaces have been omitted.

be the case with leaving out a leading article like “the”. While it is possible for a human to see the keys could be somewhat related, a clustering algorithm that relies on sorting on the key will have problems. If a word is missing from the beginning or the end though one approach could be to have one key pick letters in reverse order – starting with the last word instead of the first. You will still need to increase your window size to catch this, especially in large databases, since the keys will not be similar, but the first few letters of the keys will be similar at least; see figure 5.2. Note that unless you wish to double the amount of passes you make you will have to accept some inaccuracies in the case where there is both a typographical error and a missing word, in the case where the error affects the letters used in the key.

There is one type of inconsistency which could be classified as a typographical error, but which I chose to put in its own class because of the frequency of this particular type of error: Missing spaces. A sample taken from the freedb.org dataset suggests that somewhere around 0.5% to 0.7% of records are records without spaces which should have had spaces. I am uncertain as to the cause, but it is a good example of a type of error which introduces more complexity in picking keys. One way you might make a key is by picking the letters depending on how many characters (ignoring any spaces) they are away from the beginning or end of the artist field. This probably works best with letters appearing really near the beginning or the end of the field since letters further in are more likely to be affected by insertion and deletion operations as the letters are picked with regard to their relative difference from the beginning or end, an example in figure 5.3.

Another source of duplicates is related to how people write names, as it is common to write both `<firstname> <surname>` and `<surname> <firstname>`, for this case I have not heard of, nor am I able to come up with, a similar type of key which would cluster both variations near one another for an arbitrary artist name. The best solution I can come up with is to make use of commutative operators, such as addition or multiplication. You could, for example, assign each letter a value from 1 to 26, as long as you are working with only the English alphabet and case sensitivity is not necessary, and then add letters at a given position in each word together, and then sort based on a set containing these values, for example “Bob Dylan” would give you the set $2 + 4, 15 + 25$.

Obviously there are more than two letters which could form the majority of the sums you could come up with, so you would require a large window size. Another approach which might work better is assigning each letter a prime number, and multiplying them together to form the set of products to sort on. In an English alphabet, case insensitive scenario these values would grow larger than sums, with the max product for a record with 3 words being $101^3 = 1030301$, which seems managable, but this could end up being a much larger value if you need to be able to distinguish more than 26 separate characters.

This key would also require an increase in window size, as even though only one set of letters can form a given number the order could be arbitrary (which is, after all, the whole point), and so you would generally match more records which are dissimilar.

Finally the last problem, one that might not be so common in some datasets, but which might occur in more “messy” datasets. It is just a generalization of an earlier problem: What if instead of just having one word added or removed there could be an arbitrary number of additional words in a record? This is frequent in music where albums or tracks could have more than one artist, for example one artist featuring another. In this case it is also hard to generate a key that will work, since you do not know how many words might be added or removed, and not if it is before or after, or both. Also this is not so much an issue in all cases of duplicate detection – if you are simply detecting duplicates for the sake of purging excess records you might want to leave it as a separate artist, but if you are genuinely interested in finding out which records have what artists then you would want it to match as a duplicate. Also note in such a case it would give problems when doing transitive closure. You could get around this by making a few comparisons between sets before taking the transitive closure, and not joining two groups unless most of their records are relatively similar.

Fixing some of these problems can be done, however in a lot of cases this means a larger number of passes and larger window sizes. There are already a lot of unnecessary and costly pair-wise fuzzy matches being made due to the relatively low percentage of duplicates, and doing more passes and increasing window size will only cause this to grow.

Of course there are datasets which do not exhibit all of the above problems, or maybe just one or two of them, and there are datasets without any duplicates at all. There are datasets where all the data is neatly ordered and labeled, and there are datasets with primary-key values like a social security numbers where detecting duplicates involve little more than looking at the social security number. This is not the kind of data this project intends to look at.

I will be using the PMC, taking advantage of its specialized nature and fast pattern matching capabilities, and prove that it can still do efficient pattern matching. I will be running two tests; one to check what the PMC is capable of in terms of accuracy on a dirty dataset, and one to test the scalability of the PMC. For this I will be using two different datasets.

5.2 Datasets

5.2.1 Accuracy

For accuracy testing I will be using the freedb.org dataset [4]. It is a dataset consisting of album, and track information for CDs where anyone can store the information for their CD and it will be stored together with a disc ID calculated from the CD. This disc ID can then be used to retrieve this data from the database. This is used with great effect in software CD players which connect to the database in order to retrieve the titles and artists for all the songs on a CD when an unknown CD is inserted.

My task will be, in the spirit of our example, finding what albums an artist appears on. However the dataset is not annotated, and it is quite large and I do not have the resources to annotate the entire dataset by hand. As such I only look at the most popular artists, and I only use the titles and artists for CDs, to make the number of records that need to be annotated more manageable. The way the dataset was annotated was by first running several very general search on the dataset with the PMC, modifying the queries by hand to what I believe should catch all occurrences of an artist name, and then some. The hits were then annotated "by hand" – a computer program did the actual annotation, I was just prompted, for each of the records, on whether or not they were duplicates.

There are of course several shortcomings with this kind of approach. For one using the PMC to both annotate and do duplicate detection might introduce some bias. I did however make the queries used in annotating more general in the hope of catching even the most obscure duplicate. There is also the problem of records that are duplicates because of an incorrectly entered artist name. The user might have entered the wrong name for the artist, or entered data in the wrong fields (for example the artist name in the album field). I chose to look away from these cases because the first one would be almost impossible to discover, and the second one would greatly increase the number of records I would have to annotate manually without adding much, if any, relevant records.

There are also a number of records I am not sure how to handle. Some CDs might for example have names like "Various Artists - The Hits of Elvis Presley" and I will be uncertain whether they mean there are various artists singing the hits of Elvis Presley, or whether the Various Artists is just a mistake (as it is might be a common default option). Other albums may be things like guitar lessons or instrumentals where, where the artist name still appears in the artists field, and it is uncertain whether the artist actually plays the songs on the CD. In these cases (at least in the cases of instrumentals and guitar lessons) I have chosen to tag the records as not a duplicate. While I could just have removed records I was unsure about to avoid the problem, I do not wish to take on the attitude that if some records are "difficult" to clean that they should be ignored. This is the reality of dirty data: There will be ambiguity and uncertainty and mistakes. That is the reason I am using this dataset.

With this in mind I shall, instead of just showing off numbers, give some examples of what kind of records can be matched with the PMC, and those of you who know more about data cleaning than me can make an educated guess about how much I am leaving out.

5.2.2 Scalability

For scalability I will be using a customly generated dataset. This dataset will be generated from a list of approximately 16000 names by picking 3 names from the list at random and combining them to form a record. After forming a record there is a given probability that duplicates of that record will be created, the number of duplicates to be created being given by the formula:

$$1 + (N - 1) * r^{10} \tag{5.1}$$

where N is the max number of duplicates desired, and r is a random number in the interval $[0, 1)$. In my algorithm the max duplicates per record is set to 50, leading to approximately 20% of the records in the dataset being duplicates of another record. For each duplicate of a record the number of errors is decided by the same formula as in 5.1, but with N as the max number of errors per record (set to 3 in my algorithm). For each error a random operation is chosen between the four basic operations from [3]: Insertion, deletion, substitution and transposition. This operation is then applied at a random position in the record.

The amount of duplicates is probably larger than what might be considered normal, also the duplicates themselves are very mechanical. However my intention on this dataset is not to test duplicate detection itself, but rather the scalability of the algorithm. The duplicates are just there so that I can a) have some confirmation that the algorithm is behaving as expected and that it is detecting duplicates like it should, and b) be sure that that if the amount of duplicates detected has an impact on the speed then the overestimation of the number of duplicates present should cover this.

5.3 Accuracy Testing

For testing accuracy I will be using three files:

- A flat file with all the album names and the album artist from the freedb database.
- A list of the artist names which I have annotated which will be used to make the queries.
- A flat file containing the annotations

The program will read in the arist names, use the PMC to search against the database and when it is done compare the results to the annotations.

The data file is a flat file where each line is a record and the fields are separated by tabs:

```
<artist name> <album name> <album genre> <main db address>
```

For the purposes of this testing however we will mostly be concerned with the first field, the album name is only relevant for annotation, and the last two will be ignored.

There are two things I will be testing, mainly

- The accuracy of the PMC

- What can be done to filter out some of the false positives.

The queries are generated based on a set of rules I have defined – they could easily be specifically designed for our current dataset, but part of the idea is to prove that relatively general queries can produce accurate results, which is important knowledge as it can let you do duplicate detection on data you know little or nothing about.

Based on my testing there seem to be two main concepts which have had the most success in general duplicate detection with the PMC: A combination of the character-displacement and pattern n-of-m operators

{ `string :  c =` x`, p >=` $s - y$ }

for a string of size s and n-gram matching

{ $\texttt{tok}_1$, $\texttt{tok}_2$, ..., $\texttt{tok}_m$ `:  n >=` $m - x$ }

The character-displacement and pattern n-of-m pattern alone will generally catch typographical errors. While it is not edit-distance centric, having $x = 1$ and $y = 1$ for a string of size s will catch all typographical errors with a Damerau-Levenshtein distance of one, plus some cases where the distance is two. Increasing x and/or y will make the query less specific and catch more duplicates but also increase the chance of false positives. Especially increasing x, which increases the character-displacement permitted, will very quickly lead to a lot of false positives as it essentially allows for reshuffling of letters in the string. Combined with the pattern n-of-m operator which will not require all letters in the string to be present this means that you can easily end up with entirely new words that only have a few letters in common with your starting string. How high values of x and y that are permittable without total degeneration of accuracy varies with string length.

N-grams are used extensively in natural language processing and are a common way to do fuzzy matching when correcting OCR input, and has also been applied to spell checking, amongst other areas. Essentially it lets you search for words based on the token occuring in the words. You can specify that only a certain number of the tokens need to match before you consider a word a match, which allows for some mistakes in the word. You also have the ability to survey your dataset and create an n-gram frequency map and not including n-grams which are very common, since matching a common n-gram does not increase the probability that you have an actual match by much. N-grams have successfully been applied in data cleaning before in [24, 25].

Specifically the first pattern I will be testing is a very simple version of the first pattern:

{ `''record'' :  c = 1, p >=` $s - x$ } `before \t`

where the record has a size s, and $x = 3$ for $s > 12$, 2 for $12 \geq s > 6$, 1 for $6 \geq s > 3$ and 0 for $s \leq 3$, as this has functioned fairly well for catching inconsistencies due to spelling mistakes. The purpose of the `before \t` is so that the pattern will only match the first field of the record, as the fields are separated by tabs.

Secondly I will be testing a more complex variant, which is created by splitting the record up into separate words and, for each word which is not considered optional ("the", "and", "an", and "a"), applying the simple pattern from above, minus the `before` operator. Each of the formed patterns are then joined with the `and` operator, and `before \t` is appended. “Bruce Springsteen”, for example, would become:

```
{Bruce :  c=1,p>=3} and {Springsteen :  c=1,p>=9} before \
```

This point here is that, firstly, optional words are left out, which are words likely to be left out in names when used. Secondly the amount of permittable deviance from the actual record is split up more evenly across the words, preventing inaccuracies that are caused by permitting all of the allowed omissions to occur in one place. For example “Johnny Cash” could cause a lot of false positives if the artist only needs to have “Johnny” as a first name, and perhaps one of the letters in “Cash” as part of his last name, due to the allowed number of omissions being counted for the entire record. Thirdly the application of the `and` operator says that these patterns all need to match the same document (record), but they do not need to be near one another, or in the order specified. The only thing required is that they all match once in the record and before the first tab. This allows matching both `<firstname> <surname>` and `<surname> <firstname>`, and cases of there being added words, and even the cases where there is no space character separating the parts of the name.

Lastly I will be testing a bigram pattern, which will consist of all bigrams in an artist name, for example for “Grateful Dead”:

```
{ Gr, ra, at, ..., ea, ad :  n >= 8 }
```

where 80% of the bigrams need to match.

While these searches are being run on the PMC the CPU is sitting unused. To remedy this I will also be exploring a few options for filtering the results provided by the PMC, to increase the accuracy of the search. The four options for this I will explore are:

Let the CPU idle and trust the PMC A trivial approach where you just take everyting the PMC tells you for granted, to avoid having to do extra work.

Plain Levenshtein Calculate the Levenshtein distance between the artist being search for and the artist name matched. Allow a Levenshtein distance equal to 20% of the size of the artist name. As you might imagine if all you are going to do is test the Levenshtein distance there is little point in all the complex pattern matching. Nonetheless this will be included to allow for comparisons.

Word-by-word Levenshtein For each non-optional word in the artist name being searched for take the Levenshtein distance between it and each of the words in the artist name of the matched record. Take the average of the smallest distance found for each word in the search and accept it if this average is below a threshold. A very expensive approach, and could easily take longer than the PMC searches, causing a bottleneck on the CPU side. A waste of PMC power. It should however be fairly accurate, and might prove to be a good measure for the accuracy of other, less expensive algorithms.

Size difference-based accept For all matches within a certain size range (I used 80% to 125% of the record being searched for) blindly accept the matches from the PMC. For all matches above this range demand that each non-optional word in the name of artist being serached for be present for it to be accepted as a match. This will save a lot of time on not testing a large portion of the returned matches, and on just doing simple string searching for the few outliers. The results could easily be compared to the word-by-word Levenshtein: A greater amount of false positives suggests that blindly accepting all records in the range 80% to 125% is too lenient, and a greater amount of false negatives would suggest that requiring exact matches for larger records is too strict.

A step-by-step walk through of the steps taken is as follows:

1. PMC reads the data from the database into PMC memory.
2. List of artists get read and configurations for the PMC are created from the artist names.
3. Configurations get read into the PMC.
4. When PEs are full the data is distributed to the PEs and hits are reported back.
5. Hits are passed in batches to a result handler, which might do post-processing of the results.
6. Steps 3-5 are repeated until there are no configurations left.
7. Results are checked against annotation file.
8. Results are printed.

The results include which records were marked as duplicates for which artist, which of these were actually correct, which records were false negatives, which were false positives, and if any records were matched that did not get annotated these are included as well as "unknowns" so that they can be annotated and their validity as duplicates checked by hand. Also a summary is included which sums up how many of hits, false negatives/positives and unknowns were found for each artist, in what way this changed from the last search, and a total sum of hits, false negatives/positives and unknowns.

5.4 Scalability Testing

Scalability is tested with the randomly generated dataset of names. This is because it allows me to easily generate arbitrarily large, annotated data, and I will be able to make sure the percentage of duplicate records is significant. The program for scalability testing will be doing simple duplicate detection, and purging of duplicate records, on the dataset passed to it:

1. PMC reads the generated data into memory.
2. An initial batch of names are read from the data and queries and configurations are generated from them.

3. The configurations are loaded into PMC and the search begins.
4. Results are passed to a result handler which stores the hits.
5. A constant stream of queries is generated from the file so that the next configuration can be loaded into the PMC while the search is running, and so that the PMC can be ready for another search immediately after the first one is done.
6. This goes on until the source of the queries (the data file) is exhausted.
7. The results are compared with the annotations and results are printed.

The purpose of storing and checking the hits is mainly meant as a way of confirming that the duplicate detection is functioning as intended. With this dataset the duplicates are quite easy to find, most of them being as little as an edit-distance of 1 or 2 away from the original, and it is easy to produce enough diversity in the data that very few non-duplicate records will be similar. Thus if there are a large number of false positives or false negatives it will be apparent that something is not right. Likewise if there is a low number of false positives and negatives then everything is as expected and the time taken to complete the search is a good indication of the time it would take to do duplicate detection on data of this size.

The search query I will be using is

$$\{\ \texttt{string :}\ \ \texttt{c = 1, p >=}\ s-2\ \}$$

where the string is the record being matched against, and s is the size of the string. As mentioned earlier this is a good way to catch duplicates with typographical errors, which is what our generated data set consists of. Remembering that most queries add little if any requirement on the number of PEs needed this will still be similar in resource requirements to other more complex queries.

Chapter 6

Results

"However beautiful the strategy, you should occasionally look at the results."

Winston Churchill

6.1 Accuracy Testing

The following data contains the results for the PMC accuracy on the dataset. Keep in mind that the dataset was annotated by hand and as such the overall false negative percentage will be higher than what is displayed here if the actual truth about the data was known. The numbers are a total of the hits and misses for all the 17 artists searched for. More elaborate statistics with per-artist numbers are provided in the appendix.

Any record annotated as a non-duplicate which was included as a duplicate qualifies as a *false positive*, while records annotated as duplicates which were not included in the matches qualify as *false negatives.* The *hits* are the number of duplicates correctly identified as such. The total number of records annotated as duplicates amounts to 7530, and the percentages shown in the table below are percentages of this value. This means the percentage of hits and of false negatives is the actual percentage of duplicates which was classified as such, while the percentage of false positives can be considered more of a signal-to-noise ratio.

Note that the records in the freedb dataset contain a lot of trivial matches – where the artist name is identical to the one being searched for. In fact this accounts for 80% of the matches found when searching. To better show the capabilities of the PMC these trivial matches have been subtracted from the number of hits in the results displayed (the appendix still shows the full numbers). So each hit counted below are records that are not identical (case ignored) to the artist name being searched for.

The value which was settled on for the percentage of bi-grams that need match for it to be a match was 80%. This value was 'trained' from our dataset, however, so it is by no means necessarily the optimal value for all datasets. The values of hits, false positives and false negatives, for the varying percentage of bi-grams needed can be seen in figure 6.1.3

6.1.1 Simple pat-nofm and c-disp pattern

	Hits	False positives	False negatives
TrustPMC	4125	1173	3405
Levenshtein	362	115	7168
Word Levenshtein	4098	603	3432
Ranged Accept	4044	446	3486
TrustPMC	55%	16%	45%
Levenshtein	5%	2%	95%
Word Levenshtein	54%	8%	46%
Ranged Accept	54%	6%	46%

6.1.2 More complex, compound pattern

	Hits	False positives	False negatives
TrustPMC	7304	1159	226
Levenshtein	1536	54	5994
Word Levenshtein	7201	371	329
88 **Ranged Accept**	7126	182	404
TrustPMC	97%	15%	3%
Levenshtein	20%	1%	80%
Word Levenshtein	96%	5%	4%
Ranged Accept	95%	2%	5%

6.1.3 Bi-gram pattern, 80% bi-gram match required

	Hits	False positives	False negatives
TrustPMC	7451	1153	83
Levenshtein	1647	88	5887
Word Levenshtein	7346	541	188
Ranged Accept	7341	296	220
TrustPMC	99%	15%	1%
Levenshtein	22%	1%	78%
Word Levenshtein	98%	7%	2%
Ranged Accept	97%	4%	3%

6.1.4 Sample hits

The following hits are samples taken from the results, album names omitted because they are not relevant. This is by no means close to all the hits, but rather a representative collection of the types of duplicates the PMC is capable of matching. As this is a showcasing of the PMC the results to be presented have not been passed through any of the CPU-based filtering algorithms. The query used for gathering these was the bi-gram query.

Elvis Presley

```
christmaspresley elvis
elis presley
elvil presley
elvis presey
elvisaaronpresleytupelomswiththejordanairesspringfieldmo
```

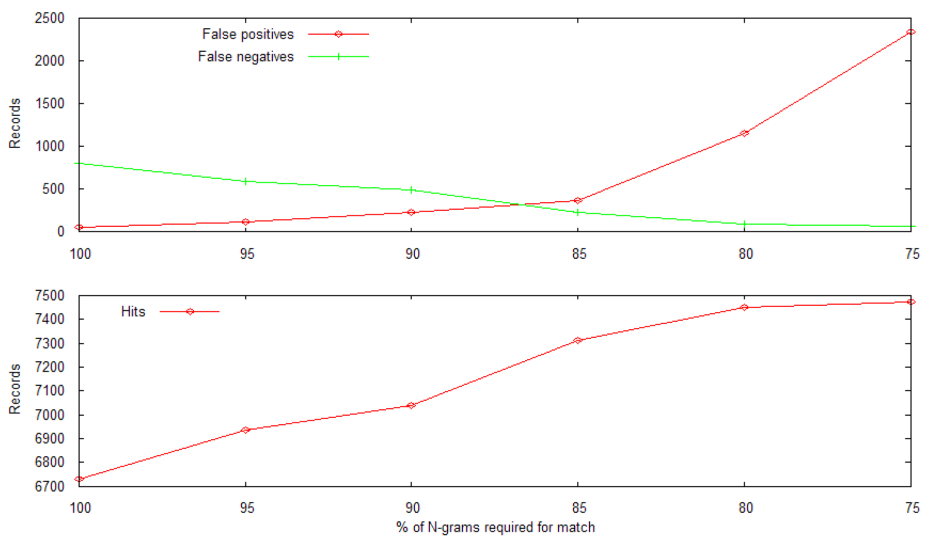


Figure 6.1: The varying accuracy as the number of bi-grams needed varies.

```
frank sinatra and elvis presley
presley elvis various
```

Pink Floyd

```
altmusicpinkfloyd
oink floyd
orb vs pink floyd remixes
pink fcloyd
syd barrettpink floyd
```

Bruce Springsteen

```
jon bonjovi/southside johnny/bruce springsteen/max weinberg 7
brice springsteen and the e street band
bruce spriggsteen
springsteenbruce
```

Wolfgang Amadeus Mozart

```
25  symphony no29mozart wolfgang armadeus 17561791
anima eterna mozart wolfgang amadeus 17561791
sir colin daviswolfgang amadeus mozart 17561791
wolfgangamadeusmozartorchestrafilarmonicaitalianacondalessandr
```

6.1.5 False Positives

One of the problems with duplicate matching on popular artist names is that there are many who pick names that are similar, as a pop-culture reference. The following records, all false positives when matching for “Led Zeppelin”, stand as a good example of this:

```
chicken shed zeppelin
dread zeppelin
fred zeppelin
great zeppelin
lets eppelin
mad zeppelin
```

Other cases of false positives are of a less interesting kind, where the artist's name appears in the artist list, without them actually appearing on the CD, such as:

```
in the style of elvis presley
intrumental memories of elvis presley
a tribute to pink floyd
blues explorations into the music of pink floyd
syd barrett architectural abdabs  pre pink floyd
the piano tribute to pink floyd
tributo a pink floyd
```

6.1.6 False Negatives

False negatives are often caused by a record having more mistakes than is permitted, especially so in short records which are generally not permitted a lot of mistakes as that would make the search way too general. The following are false negatives in searching for "Bob Marley":

```
bob marl
bob marlay
bob marly
```

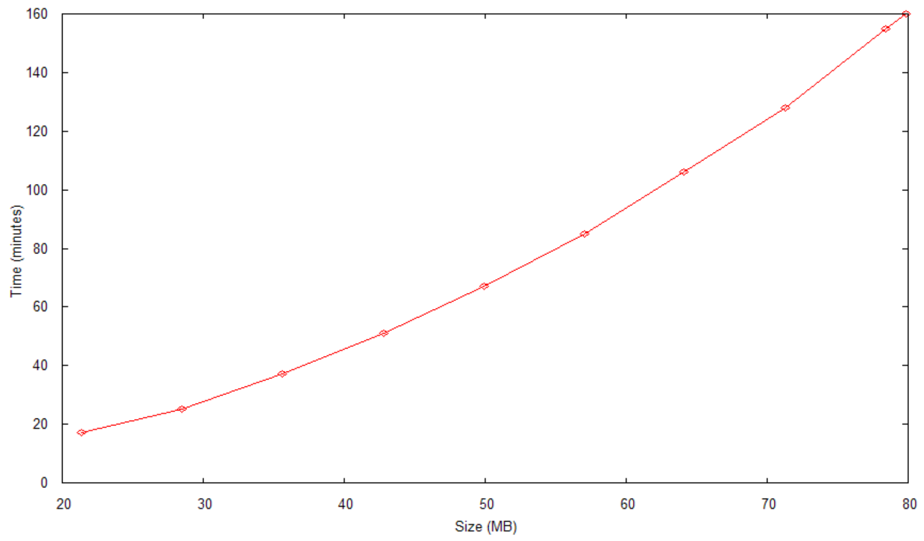
In all of the above cases 2 bi-grams are invalidated which is enough for it to not match in the case of such short artist names.

False negatives are also caused by the typographical error transposition, as the current search was done with the use of bi-grams; one transposition error will alter 3 bigrams, which might be more of a discrepancy than what is permitted in shorter record names. Such as in:

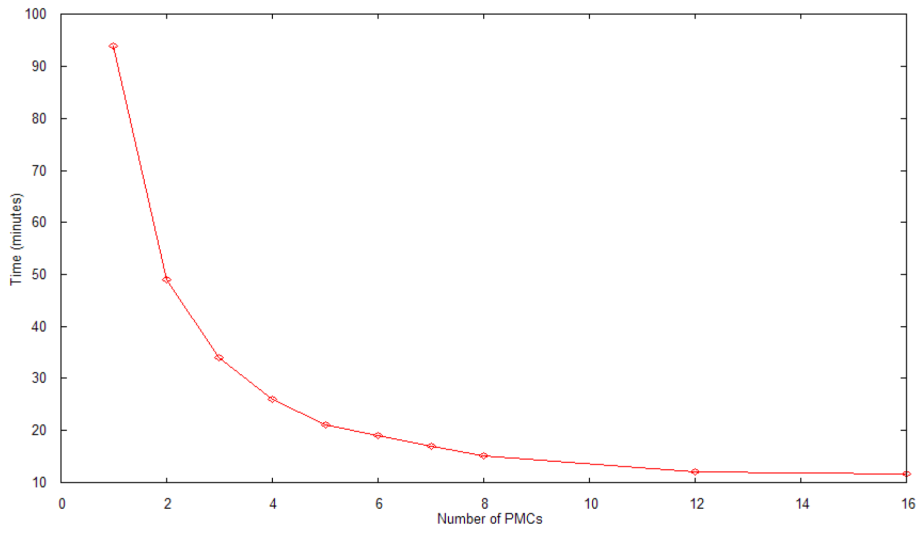
```
elivs presley
elvis presely
```

6.2 Scalability Testing

Scalability is always an important consideration. The graph below shows the increase in time (here measured in minutes) as the size of the dataset increases. Consider that as the dataset increases so does the number of records (at least in our test dataset). As such an increase in file size of the data set will not only lead to more data having to be searched through, but more data having to be cleansed. The graph below shows the effect of this increase on the time. This is for a system with 16 PMCs – one PCI card.



Another factor of scalability is the effect of adding more hardware to our system. Below is a graph for the decrease in time taken as the number of PMCs in use is increased. The size of the dataset used is 16 MB. Note that for each time the number of PMCs are doubled, the time taken is halved.



Chapter 7

Discussion

"If you torture data sufficiently, it will confess to almost anything."

Fred Menger

7.1 The nature of our data

As can be seen from the results in the previous chapter the accuracy varies greatly depending on the approach taken. The first simple PMC query did not match very well. It is however a query which has worked very well on the generated dataset, so it must have some merit. Knowing how the generated dataset is built up, with only simple typographical errors, and rarely very many of them in one record, we might realize that this simple query is very good at detecting simple typographical errors. With that in mind there is one thing in particular we can infer from our results, and that is that real world data is not always as pretty. To be fair 55% of the total duplicates did get caught by this query, however that leaves 45% which it seems can not be detected as easily by simply looking for spelling mistakes.

Also Levenshtein distance on its own between two records does not seem the way to go. However from the fact that the Levenshtein algorithm discarded most of the hits caught by even the simple pattern, in retrospect this would suggest that it might have been tuned to be a bit too specific, with a too small threshold. However I believe that, keeping in mind the performance across the board on all the queries, no amount of increasing the threshold would have made a plain Levenshtein function the proper way to go. Again it would seem that there is more to data cleaning than edit distance, and that there is something in this dataset which is complicating matters.

One filtering which did have some success, however, was the word-by-word Levenshtein. When you consider the dramatic improvement from the plain Levenshtein to the word-by-word approach, it would seem to suggest that the difference between the two is a significant one. The word-by-word approach picks each word of the artist name being searched for, and essentially searches for something which looks like that word in the match, as opposed to trying to match one whole chunk of artist name with another chunk of artist name. As it takes the Levenshtein distance between it and every word in the match to find the word which is closest it is going to be very expensive and impractical, but it does have some value. Because of its success it can tell us another thing about the dataset, which might explain why edit distance alone is not enough: A large amount of duplicates are duplicates because they either do not have the words in the same order, or because there are additional in the duplicate record not present in the original.

The fact that there are certainly problems where edit distance is a fair solution seems to suggest that there is, perhaps not unsurprisingly, more than one type of dirty data. There is the orderly dirty data, where predictions could easily be made about the data going to be entered ahead of time, such as the length or syntax of the data allowing for data validation on entry, and splitting of the records into a number of relevant fields. And then there is the more wild and chaotic dirty data, where you can not really make any assumptions about the data you will receive ahead of time. As such you can at best make suggestions to users on what information should go in what fields, and hope that they listen and that they all interpret your request as you intended it. This second wild type is where you would find that data often does not appear in the order you expect it to, and often you either find data you did not expect to appear there, or you find that data you expected to appear did not.

The implication of these traits in wild datasets is that, not only will edit distance be a bad measure, but clustering will be a very difficult job since, as discussed earlier, it is hard to cluster for duplicates when the words in the duplicate might appear in arbitrary order, with an arbitrary number of additional words in the same field.

Looking at the performance of our two more lenient patterns it would seem they had much greater success. One thing they both have in common is that they do not expect the records they are searching through to be same as the record they are searching for, only with a few slight adjustments. Instead what they do is expect to find traces of the artist name not only *inside* the artist fields of the records they are looking through, but also not in any particular order. The success of both of these algorithms lends further weight to our earlier assumptions about the more disorderly nature of the freedb dataset.

7.2 The accuracy of the PMC

Looking at the two more successful IQL queries it should be apparent that the PMC is capable of being quite accurate, allthough we should not forget that the dataset has been annotated by hand. As such, both due to the size and the wild nature of the data, it will not be entirely accurately annotated and the presence of false negatives is probably underestimated. However it can also be compared to the performance of the simpler query which is more aimed towards basic edit distance and which operates under the belief that the data should appear in the correct order and exactly where you expect it to be. This query would suffer from the same underestimation of false negatives, so comparing them to each other would be fair ground. And making this comparison it would be easy to see the merit of the more complex pattern matching.

The PMC has a second benefit to its complex pattern matching, and that is a result of the speed at which it can do this complex pattern matching. This speed allows the PMC to not only search for complex patterns, but to search for the exact complex pattern in the entire dataset, eliminating the need for clustering. This will greatly reduce the number of false negatives, especially in wild data such as our test data where clustering would be a very complex task.

As you can see though these general queries come at a price: A high number of false positives. Of course it might be slightly inaccurate to say it is only due to the fact that they are so general, as our earlier more specific query has a higher number of false positives, however there is no escaping that if you wish to catch as many duplicates as possible, you will end up with more false positives. There are, however, ways to limit the impact of this.

Taking the bi-grams as an example, consider the graph in figure 6.1.3. Doing testing to determine which percentage of bi-grams should have to match I looked at how that percentage affected the number of hits, false positives and false negatives. The more general it gets the more duplicates you are likely to find – meaning less false negatives and a greater amount of hits, however you also get a larger amount of false positives. On the graphs it would seem that 85% is sort of around the cutting point where the number of false positives starts increasing more dramatically. However the number of hits at this point is not satisfactory, and it is not until 80% we notice that the number of hits you can gain by making the query more general is flatting out. This is only natural as at this is the point where we have reached about 99% hits.

At this point we have a respectable rate of hits and false negatives, but we have to do something with the number of false positives. This is the point at which we can start utilizing our CPU, which has been neglected up until this point. We need a kind of filter which will clear out some of our false positives, without having too large an impact on our hits. It also needs to be efficient, as we would not want it to become a bottleneck and limit the speed at which the PMC can search. The pure Levenshtein approach fails spectacularly at the first point here, and the word-by-word Levenshtein would probably be rather expensive, in addition to mis-classifying those cases where people have decided the space button is overrated as it splits the match up into words for Levenshtein measuring.

Consider again the nature of our queries: They look for pieces of the artist name inside the record against which it is matching. This is especially true for bi-grams. With this in mind you can imagine that the longer the record is the greater the chance you will find something which looks like your artist name, this probably being a large part of your false positives. Because of this I implemented an algorithm which compares the size of the match with that of the pattern to be matched against, and if the sizes are close enough to each other the PMC will be trusted fully in its judgement, and if not each non-optional word in the pattern being searched for will be looked for in the match. This should block out false positives which have occurred simply because of their size. There will of course be long records where the artist name is present, but with a typographical error, leading to it being discarded. However this is a compromise between accuracy and time. It would be rather speedy compared to our alternatives and as such we can accept it if its only slightly inaccurate.

In fact the numbers prove that this method, allthough it is very simple and not at all as subtle as the word-by-word Levenshtein, is just about as accurate as the more expensive approach, and does remove a large percentage of false positives, at the cost of some false negatives.

There is one thing which need be considered of course: Each dataset has separate needs. Depending on the dataset you might not mind a large number of false positives and as such you can do very general queries, or perhaps the absence of false positives is very important and as such you are willing to sacrifice hits to avoid false positives. Clearly one size does not fit all, but I believe that it would be easy to make queries more general or more specific, especially in the case of the bi-gram pattern, as shown in figure 6.1.3.

7.3 Scalability

As you can see from the graph in the previous chapter it should be apparent that the PMC duplicate detection falls under $O(n * n)$ where n is the size of your dataset. This is not at all hard to imagine either, as increasing the size of your dataset both gives you one more record to match against, and one more record to search with. There are shortcuts that can be made, but it is hard to escape the $O(n * n)$ nature – even relying on clustering and window sizes is essentially $O(n * n)$ as an increase in data size would mean that your window size, W, would have to increase eventually if you wished to mantain accuracy. With the size of W increasing with the size of n the increase in time, allthough increasing slowly like with that of the PMC, will still be $O(n * n)$.

Being, in a lot of cases (especially if your data does not cluster easily) stuck with $O(n*n)$ you just have to make the best of it. It is becoming more and more popular to have multiple processors, with parallel processing. Programming intelligently to take advantage of parallel processing is not a simple task. Taking advantage of the parallel searching in PMCs however, is. The PMC is designed to run in parallel and as such there is an ease of splitting up data, particularly across PCI cards in one node, but even over a large number of nodes. Each node can hold one piece of the data, and the queries are then just distributed to each node (or each node generates the queries from the same source), and run a search on their part of the data and report back. This is the way it would work on a node level, and this is the way it works all the way down to PMC level with PMCs having their own memory for storing their part of the file.

The other graph on scalability clearly shows the increase in speed which can be gained; a doubling in number of PMCs used halves the time taken to search. Combine this with the ease of scaling and the fact that PMCs are relatively cheap to run, in terms of power consumption, and a large cluster of nodes with 96 PMCs each seems like a good way to keep your data sparkling clean.

7.4 Improvements

There will always be more work that can be done, and this is no exception. I have tried to present as general an approach as possible to show that it is possible to do accurate duplicate detection with the PMC without knowing too much about the data available. However I think in any problem where you apply the PMC you should be able to further improve upon your results by making your queries more specific to the dataset, or perhaps by adding additional queries to the ones already in use. A lot can be done with a rules-based approach where you might say for scenario A use this query, for scenario B use that query, and for scenario C use both.

I also think there is a lot more that can be done with bi-gram fuzzy matching.

The bi-grams I have done are relatively costly, and I am certain that with some research there would be ways to reduce the number of actual bi-grams needed to be included. One might even consider counting up all the bi-grams in the data being searched, and omitting from your pattern a certain % of the most common tokens. As it stands all the bi-gram tokens present are searched for, and this is probably redundant in a lot of cases.

It could also be that there are corners to be cut in the amount of queries being done. For example if you had the scenario of B matching as a duplicate of A implying A would match as a duplicate of B, then you could cut down to half the amount of queries as you would not need to test both B vs A and A vs B. This is not as simple when B can contain A but not the other way around, and as such B will match if searching for A, but A will not match when searching for B. However one could design queries with this particular shortcut in mind – being able to potentially halve the amount of searches that need to be run would halve the time taken.

There are also potentially interesting things which could be done with taking advantage of the properties of different queries. You could essentially construct, for example, two queries for a given record. Since the PMC will tell you which query the matches are from you could design your queries in such a way that which query the matches were returned from tells you something particular about the property of the matches – maybe one thing if its returned from one query, another if its returned from the other, and a third thing if it is matched by both. I only have vague ideas for this so far, but it seems like it could be interesting.

Chapter 8

Conclusion

"Information is not knowledge."

Albert Einstein

I believe that in the grand scheme of things this has only been a short introduction to what the PMC is capable of.

Starting this project my goal was to prove that PMC had an application in duplicate detection. I have not had much experience with data cleaning before this. As such I started out not entirely certain what kind of problems you might encounter, and I am still not certain what problems there might be, and which of them actually have reasonable solutions.

However a lot of traditional pattern matching seems to rely on clustering, and/or complex knowledge of the data being cleaned. One of which I believe I have demonstrated has several pitfalls if your data is not very predictable or orderly, and the second of which requires a lot of resources. One of the main strengths of the PMC is in its ability to do complex pattern matching against the entire dataset, and thus bypassing the need for clustering techniques and their weaknesses. Not having to cluster also makes it easier to do a very general search since you really only need to know what you are looking for, you do not need to know much about the inherent traits of the data. As long as you know what you are looking for the speed of the PMC allows it to search through the entire dataset to find it, without the kind of prior knowledge that might be necessary to take "shortcuts" in the data.

he power of the pattern matching available to you through PMC has also been shown in the accuracy which is possible and the type of records it is capable of matching. It is very beneficial to be able to do such a specific search in such a short period of time – the PMC can even be used as a way of clustering data before going over with a fine-toothed comb. In real world datasets most records do not have duplicates, and as such when you are, for every record, checking W records in its neighborhood you are doing a lot of unnecessary comparisons. Granted the PMC does a much larger number of comparisons, but it does them a lot faster. The PMC could thus also be thought of as a way of doing clustering – the complexity of the queries allow you to form them such that only the records which are really likely to be duplicates are returned, no others, and additionally *all* are caught, not only those which you manage to find a way to cluster. This would be an improvement over most clustering algorithms today.

So it would seem that, firstly, not all data is neat and orderly. Secondly data which is not neat and orderly does enjoy being clustered. And thirdly data which does not like to be clustered will be hard to do duplicate detection on if you do not have the luxury of the time it would take you to match each record against every other. The PMC seems especially adept at providing a solution in these cases because of the speed with which it does pattern matching which actually lets you search the entire dataset. I have also proved that you can achieve some measure of accuracy with the PMC, and that it can be further augmented by using the CPU in parallel with the PMC.

Still I believe I might only have scratched the surface.

Bibliography

[1] V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, Soviet Physics Doklady 10, 1966: 707?710

[2] D. A. Thompson, and J. S. Best, *The future of magnetic data storage technology* IBM Journal of Research and Development, Volume 44, Issue 3, pp. 311-321 (2000)

[3] F.J. Damerau, *A technique for computer detection and correction of spelling errors*, Communications of the ACM, 1964

[4] freedb.org, *http://www.freedb.org/en/download__database.10.html*, [last accessed July 25th, 2007]

[5] O. R. Birkeland, O. Snøve Jr., A. Halaas, M. Nedland, P. Sætrum, *The Petacomp Machine – A MIMD Cluster for Parallel Pattern-mining*, Cluster Computing, 2006 IEEE International Conference on, 2006

[6] Interagon AS, *The Interagon Query Language – A User's Guide*, April 5, 2005

[7] A. Halaas, B. Svingen, M. Nedland, P. Sætrum, O. Snøve Jr., O. R. Birkeland, *A Recursive MISD Architecture for Pattern Matching* IEEE Transactions on Very Large Scale Integration (VLSI) systems, Vol. 12, No. 7, July 2004

[8] D. Bitton and D. J. DeWitt, *Duplicate record elimination in large data files*, ACM Transactions on Database Systems, 8(2):255-65, 1983.

[9] B. Kilss, and W. Alvey, editors, *Record linkage techniques, 1985: Proceedings of the Workshop on Exact Matching Methodologies*, Arlington, Virginia, 1985 Internal Revenue Service, Statistics of Income Division

[10] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James, *Automatic linkage of vital records*, Science, 130:954-959, October 1959. Reprinted in [9]

[11] C. A. Giles, A. A. Brooks, T. Doszkocs, and D. J. Hummel, *An experiment in computer-assisted duplicate checking*, In proceedings of the ASIS Annual Meeting, page 108, 1967

[12] M. Hernández and S. Stolfo, *Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem*, In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 127-138, May 1995

[13] A. E. Monge and C. Elkan, *An Efficient Domain-Independent Algorithm for Detecting Approximately Duplicate Database Records*, Research Issues on Data Mining and Knowledge Discovery, 1997

[14] M. L. Lee and T. W. Ling and H. J. Lu and Y. T. Ko, *Cleansing Data for Mining and Warehousing*, Database and Expert Systems Applications, p751-760, 1999

[15] H. B. Newcombe, *Handbook of record linkage: methods for health and statistical studies, administration, and business* Oxford University Press, 1988

[16] A. E. Monge *Matching Algorithms within a Duplicate Detection System* IEEE Data Engineering Bulletin, 2000

[17] Z. Galil and R. Giancarlo, *Data structures and algorithms for approximate string matching*, Journal of Complexity, vol. 4, pp. 33-72, 1988

[18] W. I. Chang and J. Lampe *Theoretical and empirical comparisons of approximate string matching algorithms*, in CPM: 3rd Symposium on Combinatorial Pattern Matching, pp. 175-84, 1992

[19] M.-W. Du and S. C. Chang, *Approach to designing very fast approximate string matching algorithms* IEEE Transactions on Knowledge and Data Engineering, vol. 6, pp. 620-633, Aug. 1994

[20] M. L. Lee, T. W. Ling and W. L. Low *IntelliClean : A Knowledge-Based Intelligent Data Cleaner* Proceedings of ACM SIGKDD, pp. 290-294, 2000

[21] A. Feekin, Z. Chen, *Duplicate Detection Using K-way Sorting Method* Proceedings of ACM SAC, March 2000, pp. 323-327

[22] R. Ananthakrishna, S. Chaudhuri, V. Ganti *Eliminating Fuzzy Duplicates in Data Warehouses*, Proceedings of the 28^{th} VLDB Conference, Hong Kong, China, 2002

[23] V. Raman, J. M. Hellerstein, *Potter's Wheel: An Interactive Framework for Data Transformation and Cleaning* Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

[24] S. Chaudhuri, K. Ganjam, V. Ganti, R. Motwani, *Robust and Efficient Fuzzy Match for Online Data Cleaning* Proceedings of ACM SIGMOD June 2003, pp. 313-324

[25] L. Gravano, P. G. Ipeirotis, N. Koudas, D. Srivastava, *Text Joins for Data Cleansing and Integration in an RDBMS* Proceedings of ICDE' 2003 pp. 729-731

Appendices

Appendix A

Artists used in the searching

```
0 elvis presley
2 pink floyd
4 grateful dead
5 bruce springsteen
7 led zeppelin
8 metallica
9 wolfgang amadeus mozart
11 depeche mode
14 johann sebastian bach
15 the rolling stones
17 ludwig van beethoven
21 louis armstrong
22 eric clapton
26 duke ellington
27 jimi hendrix
28 bob marley
30 johnny cash
```

Appendix B

Detailed results from searching

Number in brackets correspond to the inrease/decrease from the trivial approach (no filtering by CPU), except for in the case of the trivial searches, where the numbers in the brackets should be ignored.

H is hits, F+ false positives, F- false negatives and UK unknowns.

```
Simple Fuzzy, Trivial
 0  H 3591 (0)    F+ 6 (1)    F- 97 (0)    UK 0 (-1)
2  H 2661 (0)    F+ 45 (5)    F- 8 (0)    UK 0 (-5)
4  H 2517 (0)    F+ 5 (4)    F- 1 (0)    UK 0 (-4)
5  H 3499 (0)    F+ 1 (0)    F- 81 (0)    UK 0 (0)
7  H 2164 (0)    F+ 51 (4)    F- 68 (0)    UK 0 (-4)
8  H 2045 (0)    F+ 527 (0)    F- 11 (0)    UK 0 (0)
9  H 2192 (0)    F+ 0 (0)    F- 564 (0)    UK 0 (0)
11  H 1891 (0)    F+ 2 (0)    F- 11 (0)    UK 0 (0)
14  H 1854 (0)    F+ 0 (0)    F- 600 (0)    UK 0 (0)
15  H 1574 (0)    F+ 5 (0)    F- 1083 (0)    UK 0 (0)
17  H 1788 (0)    F+ 0 (0)    F- 540 (0)    UK 0 (0)
21  H 1597 (0)    F+ 4 (4)    F- 102 (0)    UK 0 (-4)
22  H 1500 (0)    F+ 17 (0)    F- 91 (0)    UK 0 (0)
26  H 1324 (0)    F+ 18 (18)    F- 42 (0)    UK 0 (-18)
27  H 1319 (0)    F+ 26 (0)    F- 40 (0)    UK 0 (0)
28  H 1808 (0)    F+ 37 (6)    F- 24 (0)    UK 0 (-6)
30  H 1090 (0)    F+ 429 (378)    F- 42 (0)    UK 0 (-378)

TOTAL H 34414    F+ 1173    F- 3405    UK 0
4125



Simple Fuzzy, Plain Lev,
0  H 3534 (-57)    F+ 0 (-6)    F- 154 (57)    UK 0 (0)
2  H 2605 (-56)    F+ 30 (-15)    F- 64 (56)    UK 0 (0)
```

```
4  H 2356 (-161)    F+ 1 (-4)    F- 162 (161)    UK 0 (0)
5  H 2293 (-1206)    F+ 0 (-1)    F- 1287 (1206)    UK 0 (0)
7  H 2139 (-25)    F+ 4 (-47)    F- 93 (25)    UK 0 (0)
8  H 2016 (-29)    F+ 23 (-504)    F- 40 (29)    UK 0 (0)
9  H 2030 (-162)    F+ 0 (0)    F- 726 (162)    UK 0 (0)
11  H 1880 (-11)    F+ 0 (-2)    F- 22 (11)    UK 0 (0)
14  H 1672 (-182)    F+ 0 (0)    F- 782 (182)    UK 0 (0)
15  H 1549 (-25)    F+ 1 (-4)    F- 1108 (25)    UK 0 (0)
17  H 1591 (-197)    F+ 0 (0)    F- 737 (197)    UK 0 (0)
21  H 1398 (-199)    F+ 3 (-1)    F- 301 (199)    UK 0 (0)
22  H 1319 (-181)    F+ 0 (-17)    F- 272 (181)    UK 0 (0)
26  H 1090 (-234)    F+ 3 (-15)    F- 276 (234)    UK 0 (0)
27  H 1091 (-228)    F+ 0 (-26)    F- 268 (228)    UK 0 (0)
28  H 1059 (-749)    F+ 13 (-24)    F- 773 (749)    UK 0 (0)
30  H 1029 (-61)    F+ 37 (-392)    F- 103 (61)    UK 0 (0)

TOTAL H 30651    F+ 115    F- 7168    UK 0
362


Simple Fuzzy, WordLev
0  H 3591 (0)    F+ 3 (-3)    F- 97 (0)    UK 0 (0)
2  H 2661 (0)    F+ 45 (0)    F- 8 (0)    UK 0 (0)
4  H 2517 (0)    F+ 3 (-2)    F- 1 (0)    UK 0 (0)
5  H 3496 (-3)    F+ 1 (0)    F- 84 (3)    UK 0 (0)
7  H 2164 (0)    F+ 51 (0)    F- 68 (0)    UK 0 (0)
8  H 2043 (-2)    F+ 37 (-490)    F- 13 (2)    UK 0 (0)
9  H 2192 (0)    F+ 0 (0)    F- 564 (0)    UK 0 (0)
11  H 1890 (-1)    F+ 0 (-2)    F- 12 (1)    UK 0 (0)
14  H 1854 (0)    F+ 0 (0)    F- 600 (0)    UK 0 (0)
15  H 1574 (0)    F+ 5 (0)    F- 1083 (0)    UK 0 (0)
17  H 1787 (-1)    F+ 0 (0)    F- 541 (1)    UK 0 (0)
21  H 1590 (-7)    F+ 4 (0)    F- 109 (7)    UK 0 (0)
22  H 1497 (-3)    F+ 17 (0)    F- 94 (3)    UK 0 (0)
26  H 1322 (-2)    F+ 15 (-3)    F- 44 (2)    UK 0 (0)
27  H 1317 (-2)    F+ 26 (0)    F- 42 (2)    UK 0 (0)
28  H 1804 (-4)    F+ 36 (-1)    F- 28 (4)    UK 0 (0)
30  H 1088 (-2)    F+ 360 (-69)    F- 44 (2)    UK 0 (0)

TOTAL H 34387    F+ 603    F- 3432    UK 0
4098


Simple Fuzzy, Ranged Accept
0  H 3590 (-1)    F+ 2 (-4)    F- 98 (1)    UK 0 (0)
2  H 2661 (0)    F+ 40 (-5)    F- 8 (0)    UK 0 (0)
4  H 2511 (-6)    F+ 1 (-4)    F- 7 (6)    UK 0 (0)
5  H 3494 (-5)    F+ 1 (0)    F- 86 (5)    UK 0 (0)
7  H 2161 (-3)    F+ 50 (-1)    F- 71 (3)    UK 0 (0)
8  H 2044 (-1)    F+ 30 (-497)    F- 12 (1)    UK 0 (0)
```

```
9  H 2186 (-6)    F+ 0 (0)    F- 570 (6)    UK 0 (0)
11  H 1891 (0)    F+ 0 (-2)    F- 11 (0)    UK 0 (0)
14  H 1840 (-14)    F+ 0 (0)    F- 614 (14)    UK 0 (0)
15  H 1574 (0)    F+ 3 (-2)    F- 1083 (0)    UK 0 (0)
17  H 1771 (-17)    F+ 0 (0)    F- 557 (17)    UK 0 (0)
21  H 1583 (-14)    F+ 3 (-1)    F- 116 (14)    UK 0 (0)
22  H 1497 (-3)    F+ 15 (-2)    F- 94 (3)    UK 0 (0)
26  H 1321 (-3)    F+ 8 (-10)    F- 45 (3)    UK 0 (0)
27  H 1317 (-2)    F+ 26 (0)    F- 42 (2)    UK 0 (0)
28  H 1807 (-1)    F+ 19 (-18)    F- 25 (1)    UK 0 (0)
30  H 1085 (-5)    F+ 248 (-181)    F- 47 (5)    UK 0 (0)

TOTAL H 34333    F+ 446    F- 3486    UK 0
4044


Complex Fuzzy, Trivial
0  H 3677 (86)    F+ 10 (4)    F- 11 (-86)    UK 0 (0)
2  H 2664 (3)    F+ 25 (-20)    F- 5 (-3)    UK 0 (0)
4  H 2517 (0)    F+ 3 (-2)    F- 1 (0)    UK 0 (0)
5  H 3562 (63)    F+ 1 (0)    F- 18 (-63)    UK 0 (0)
7  H 2195 (31)    F+ 15 (-36)    F- 37 (-31)    UK 0 (0)
8  H 2045 (0)    F+ 527 (0)    F- 11 (0)    UK 0 (0)
9  H 2753 (561)    F+ 1 (1)    F- 3 (-561)    UK 0 (0)
11  H 1902 (11)    F+ 219 (217)    F- 0 (-11)    UK 0 (0)
14  H 2445 (591)    F+ 0 (0)    F- 9 (-591)    UK 0 (0)
15  H 2640 (1066)    F+ 42 (37)    F- 17 (-1066)    UK 0 (0)
17  H 2234 (446)    F+ 0 (0)    F- 94 (-446)    UK 0 (0)
21  H 1687 (90)    F+ 0 (-4)    F- 12 (-90)    UK 0 (0)
22  H 1590 (90)    F+ 359 (342)    F- 1 (-90)    UK 0 (0)
26  H 1363 (39)    F+ 9 (-9)    F- 3 (-39)    UK 0 (0)
27  H 1358 (39)    F+ 48 (22)    F- 1 (-39)    UK 0 (0)
28  H 1830 (22)    F+ 4 (-33)    F- 2 (-22)    UK 0 (0)
30  H 1131 (41)    F+ 296 (-133)    F- 1 (-41)    UK 0 (0)

TOTAL H 37593    F+ 1559    F- 226    UK 0
7304


Complex Fuzzy, Plain Lev
0  H 3538 (-139)    F+ 0 (-10)    F- 150 (139)    UK 0 (0)
2  H 2605 (-59)    F+ 11 (-14)    F- 64 (59)    UK 0 (0)
4  H 2356 (-161)    F+ 1 (-2)    F- 162 (161)    UK 0 (0)
5  H 2293 (-1269)    F+ 0 (-1)    F- 1287 (1269)    UK 0 (0)
7  H 2172 (-23)    F+ 0 (-15)    F- 60 (23)    UK 0 (0)
8  H 2016 (-29)    F+ 23 (-504)    F- 40 (29)    UK 0 (0)
9  H 2080 (-673)    F+ 0 (-1)    F- 676 (673)    UK 0 (0)
11  H 1891 (-11)    F+ 0 (-219)    F- 11 (11)    UK 0 (0)
14  H 1736 (-709)    F+ 0 (0)    F- 718 (709)    UK 0 (0)
15  H 2533 (-107)    F+ 0 (-42)    F- 124 (107)    UK 0 (0)
```

```
17  H 1550 (-684)    F+ 0 (0)    F- 778 (684)    UK 0 (0)
21  H 1447 (-240)    F+ 0 (0)    F- 252 (240)    UK 0 (0)
22  H 1319 (-271)    F+ 0 (-359)    F- 272 (271)    UK 0 (0)
26  H 1094 (-269)    F+ 0 (-9)    F- 272 (269)    UK 0 (0)
27  H 1094 (-264)    F+ 0 (-48)    F- 265 (264)    UK 0 (0)
28  H 1060 (-770)    F+ 0 (-4)    F- 772 (770)    UK 0 (0)
30  H 1041 (-90)    F+ 19 (-277)    F- 91 (90)    UK 0 (0)

TOTAL H 31825    F+ 54    F- 5994    UK 0
1536


Complex Fuzzy, Word Lev
0  H 3673 (-4)    F+ 3 (-7)    F- 15 (4)    UK 0 (0)
2  H 2662 (-2)    F+ 22 (-3)    F- 7 (2)    UK 0 (0)
4  H 2517 (0)    F+ 2 (-1)    F- 1 (0)    UK 0 (0)
5  H 3558 (-4)    F+ 1 (0)    F- 22 (4)    UK 0 (0)
7  H 2161 (-34)    F+ 2 (-13)    F- 71 (34)    UK 0 (0)
8  H 2043 (-2)    F+ 37 (-490)    F- 13 (2)    UK 0 (0)
9  H 2739 (-14)    F+ 1 (0)    F- 17 (14)    UK 0 (0)
11  H 1901 (-1)    F+ 7 (-212)    F- 1 (1)    UK 0 (0)
14  H 2438 (-7)    F+ 0 (0)    F- 16 (7)    UK 0 (0)
15  H 2636 (-4)    F+ 18 (-24)    F- 21 (4)    UK 0 (0)
17  H 2228 (-6)    F+ 0 (0)    F- 100 (6)    UK 0 (0)
21  H 1677 (-10)    F+ 0 (0)    F- 22 (10)    UK 0 (0)
22  H 1587 (-3)    F+ 73 (-286)    F- 4 (3)    UK 0 (0)
26  H 1360 (-3)    F+ 0 (-9)    F- 6 (3)    UK 0 (0)
27  H 1356 (-2)    F+ 26 (-22)    F- 3 (2)    UK 0 (0)
28  H 1825 (-5)    F+ 4 (0)    F- 7 (5)    UK 0 (0)
30  H 1129 (-2)    F+ 175 (-121)    F- 3 (2)    UK 0 (0)

TOTAL H 37490    F+ 371    F- 329    UK 0
7201



Complex Fuzzy, Ranged Accept
0  H 3675 (-2)    F+ 2 (-8)    F- 13 (2)    UK 0 (0)
2  H 2664 (0)    F+ 19 (-6)    F- 5 (0)    UK 0 (0)
4  H 2511 (-6)    F+ 1 (-2)    F- 7 (6)    UK 0 (0)
5  H 3556 (-6)    F+ 1 (0)    F- 24 (6)    UK 0 (0)
7  H 2160 (-35)    F+ 2 (-13)    F- 72 (35)    UK 0 (0)
8  H 2044 (-1)    F+ 30 (-497)    F- 12 (1)    UK 0 (0)
9  H 2723 (-30)    F+ 1 (0)    F- 33 (30)    UK 0 (0)
11  H 1902 (0)    F+ 0 (-219)    F- 0 (0)    UK 0 (0)
14  H 2408 (-37)    F+ 0 (0)    F- 46 (37)    UK 0 (0)
15  H 2633 (-7)    F+ 7 (-35)    F- 24 (7)    UK 0 (0)
17  H 2218 (-16)    F+ 0 (0)    F- 110 (16)    UK 0 (0)
21  H 1668 (-19)    F+ 0 (0)    F- 31 (19)    UK 0 (0)
22  H 1587 (-3)    F+ 51 (-308)    F- 4 (3)    UK 0 (0)
```

```
26  H 1357 (-6)    F+ 0 (-9)    F- 9 (6)    UK 0 (0)
27  H 1355 (-3)    F+ 26 (-22)    F- 4 (3)    UK 0 (0)
28  H 1828 (-2)    F+ 3 (-1)    F- 4 (2)    UK 0 (0)
30  H 1126 (-5)    F+ 39 (-257)    F- 6 (5)    UK 0 (0)

TOTAL H 37415    F+ 182    F- 404    UK 0
7126



NGram (1.0), Trivial
0  H 3657 (67)    F+ 2 (0)    F- 31 (-67)    UK 0 (0)
2  H 2649 (-12)    F+ 8 (-32)    F- 20 (12)    UK 0 (0)
4  H 2493 (-18)    F+ 1 (0)    F- 25 (18)    UK 0 (0)
5  H 3557 (63)    F+ 1 (0)    F- 23 (-63)    UK 0 (0)
7  H 2186 (25)    F+ 2 (-48)    F- 46 (-25)    UK 0 (0)
8  H 2037 (-7)    F+ 8 (-22)    F- 19 (7)    UK 0 (0)
9  H 2649 (463)    F+ 1 (1)    F- 107 (-463)    UK 0 (0)
11  H 1886 (-5)    F+ 1 (1)    F- 16 (5)    UK 0 (0)
14  H 2322 (482)    F+ 0 (0)    F- 132 (-482)    UK 0 (0)
15  H 2625 (1051)    F+ 8 (5)    F- 32 (-1051)    UK 0 (0)
17  H 2147 (376)    F+ 0 (0)    F- 181 (-376)    UK 0 (0)
21  H 1611 (28)    F+ 0 (-3)    F- 88 (-28)    UK 0 (0)
22  H 1580 (83)    F+ 2 (-13)    F- 11 (-83)    UK 0 (0)
26  H 1351 (30)    F+ 0 (-8)    F- 15 (-30)    UK 0 (0)
27  H 1338 (21)    F+ 2 (-24)    F- 21 (-21)    UK 0 (0)
28  H 1827 (20)    F+ 4 (-15)    F- 5 (-20)    UK 0 (0)
30  H 1106 (21)    F+ 15 (-233)    F- 26 (-21)    UK 0 (0)

TOTAL H 37021    F+ 55    F- 798    UK 0
6732


NGram (0.95), Trivial
0  H 3664 (7)    F+ 2 (0)    F- 24 (-7)    UK 0 (0)
2  H 2649 (0)    F+ 8 (0)    F- 20 (0)    UK 0 (0)
4  H 2494 (1)    F+ 1 (0)    F- 24 (-1)    UK 0 (0)
5  H 3561 (4)    F+ 1 (0)    F- 19 (-4)    UK 0 (0)
7  H 2186 (0)    F+ 2 (0)    F- 46 (0)    UK 0 (0)
8  H 2037 (0)    F+ 8 (0)    F- 19 (0)    UK 0 (0)
9  H 2649 (0)    F+ 1 (0)    F- 107 (0)    UK 0 (0)
11  H 1886 (0)    F+ 1 (0)    F- 16 (0)    UK 0 (0)
14  H 2428 (106)    F+ 2 (2)    F- 26 (-106)    UK 0 (0)
15  H 2653 (28)    F+ 50 (42)    F- 4 (-28)    UK 0 (0)
17  H 2190 (43)    F+ 0 (0)    F- 138 (-43)    UK 0 (0)
21  H 1621 (10)    F+ 8 (8)    F- 78 (-10)    UK 0 (0)
22  H 1580 (0)    F+ 2 (0)    F- 11 (0)    UK 0 (0)
26  H 1356 (5)    F+ 0 (0)    F- 10 (-5)    UK 0 (0)
27  H 1338 (0)    F+ 2 (0)    F- 21 (0)    UK 0 (0)
28  H 1827 (0)    F+ 4 (0)    F- 5 (0)    UK 0 (0)
```

```
30  H 1106 (0)    F+ 15 (0)    F- 26 (0)    UK 0 (0)

TOTAL H 37225    F+ 107    F- 594    UK 0
6936



NGram (0.9), Trivial
0  H 3664 (7)    F+ 2 (0)    F- 24 (-7)    UK 0 (0)
2  H 2661 (12)    F+ 14 (6)    F- 8 (-12)    UK 0 (0)
4  H 2494 (1)    F+ 1 (0)    F- 24 (-1)    UK 0 (0)
5  H 3561 (4)    F+ 1 (0)    F- 19 (-4)    UK 0 (0)
7  H 2191 (5)    F+ 8 (6)    F- 41 (-5)    UK 0 (0)
8  H 2051 (14)    F+ 37 (29)    F- 5 (-14)    UK 0 (0)
9  H 2649 (0)    F+ 1 (0)    F- 107 (0)    UK 0 (0)
11  H 1896 (10)    F+ 4 (3)    F- 6 (-10)    UK 0 (0)
14  H 2449 (127)    F+ 5 (5)    F- 7 (-125)    UK 0 (0)
15  H 2653 (28)    F+ 50 (42)    F- 4 (-28)    UK 0 (0)
17  H 2190 (43)    F+ 0 (0)    F- 138 (-43)    UK 0 (0)
21  H 1621 (10)    F+ 8 (8)    F- 78 (-10)    UK 0 (0)
22  H 1587 (7)    F+ 21 (19)    F- 4 (-7)    UK 0 (0)
26  H 1356 (5)    F+ 0 (0)    F- 10 (-5)    UK 0 (0)
27  H 1358 (20)    F+ 2 (0)    F- 1 (-20)    UK 0 (0)
28  H 1829 (2)    F+ 23 (19)    F- 3 (-2)    UK 0 (0)
30  H 1120 (14)    F+ 54 (39)    F- 12 (-14)    UK 0 (0)

TOTAL H 37330    F+ 231    F- 491    UK 0
7041



NGram (0.85), Trivial
0  H 3682 (25)    F+ 5 (3)    F- 6 (-25)    UK 0 (0)
2  H 2661 (12)    F+ 14 (6)    F- 8 (-12)    UK 0 (0)
4  H 2495 (2)    F+ 4 (3)    F- 23 (-2)    UK 0 (0)
5  H 3576 (19)    F+ 1 (0)    F- 4 (-19)    UK 0 (0)
7  H 2191 (5)    F+ 8 (6)    F- 41 (-5)    UK 0 (0)
8  H 2051 (14)    F+ 37 (29)    F- 5 (-14)    UK 0 (0)
9  H 2681 (32)    F+ 1 (0)    F- 75 (-32)    UK 0 (0)
11  H 1896 (10)    F+ 4 (3)    F- 6 (-10)    UK 0 (0)
14  H 2449 (127)    F+ 5 (5)    F- 7 (-125)    UK 0 (0)
15  H 2657 (32)    F+ 146 (138)    F- 0 (-32)    UK 0 (0)
17  H 2314 (167)    F+ 0 (0)    F- 14 (-167)    UK 0 (0)
21  H 1686 (75)    F+ 31 (31)    F- 13 (-75)    UK 0 (0)
22  H 1587 (7)    F+ 21 (19)    F- 4 (-7)    UK 0 (0)
26  H 1366 (15)    F+ 9 (9)    F- 0 (-15)    UK 0 (0)
27  H 1358 (20)    F+ 2 (0)    F- 1 (-20)    UK 0 (0)
28  H 1829 (2)    F+ 23 (19)    F- 3 (-2)    UK 0 (0)
30  H 1120 (14)    F+ 54 (39)    F- 12 (-14)    UK 0 (0)

TOTAL H 37599    F+ 365    F- 222    UK 0
```

```
7310



NGram (0.80), Trivial
0  H 3682 (25)    F+ 5 (3)    F- 6 (-25)    UK 0 (0)
2  H 2661 (12)    F+ 14 (6)    F- 8 (-12)    UK 0 (0)
4  H 2495 (2)    F+ 4 (3)    F- 23 (-2)    UK 0 (0)
5  H 3580 (23)    F+ 2 (1)    F- 0 (-23)    UK 0 (0)
7  H 2232 (46)    F+ 90 (88)    F- 2 (-44)    UK 0 (0)
8  H 2055 (18)    F+ 275 (267)    F- 1 (-18)    UK 0 (0)
9  H 2747 (98)    F+ 1 (0)    F- 9 (-98)    UK 0 (0)
11  H 1901 (15)    F+ 37 (36)    F- 1 (-15)    UK 0 (0)
14  H 2451 (129)    F+ 53 (53)    F- 5 (-127)    UK 0 (0)
15  H 2657 (32)    F+ 146 (138)    F- 0 (-32)    UK 0 (0)
17  H 2317 (170)    F+ 1 (1)    F- 11 (-170)    UK 0 (0)
21  H 1686 (75)    F+ 31 (31)    F- 13 (-75)    UK 0 (0)
22  H 1591 (11)    F+ 96 (94)    F- 0 (-11)    UK 0 (0)
26  H 1366 (15)    F+ 9 (9)    F- 0 (-15)    UK 0 (0)
27  H 1359 (21)    F+ 31 (29)    F- 0 (-21)    UK 0 (0)
28  H 1829 (2)    F+ 23 (19)    F- 3 (-2)    UK 0 (0)
30  H 1131 (25)    F+ 335 (320)    F- 1 (-25)    UK 0 (0)

TOTAL H 37740    F+ 1153    F- 83    UK 0
7451



NGram (0.80), Plain Lev
0  H 3543 (-139)    F+ 0 (-5)    F- 145 (139)    UK 0 (0)
2  H 2602 (-59)    F+ 6 (-8)    F- 67 (59)    UK 0 (0)
4  H 2340 (-155)    F+ 1 (-3)    F- 178 (155)    UK 0 (0)
5  H 2308 (-1272)    F+ 0 (-2)    F- 1272 (1272)    UK 0 (0)
7  H 2205 (-27)    F+ 4 (-86)    F- 29 (27)    UK 0 (0)
8  H 2026 (-29)    F+ 23 (-252)    F- 30 (29)    UK 0 (0)
9  H 2076 (-671)    F+ 0 (-1)    F- 680 (671)    UK 0 (0)
11  H 1890 (-11)    F+ 1 (-36)    F- 12 (11)    UK 0 (0)
14  H 1740 (-711)    F+ 19 (-34)    F- 716 (711)    UK 0 (0)
15  H 2539 (-118)    F+ 1 (-145)    F- 118 (118)    UK 0 (0)
17  H 1607 (-710)    F+ 0 (-1)    F- 721 (710)    UK 0 (0)
21  H 1448 (-238)    F+ 0 (-31)    F- 251 (238)    UK 0 (0)
22  H 1320 (-271)    F+ 0 (-96)    F- 271 (271)    UK 0 (0)
26  H 1097 (-269)    F+ 0 (-9)    F- 269 (269)    UK 0 (0)
27  H 1095 (-264)    F+ 0 (-31)    F- 264 (264)    UK 0 (0)
28  H 1059 (-770)    F+ 0 (-23)    F- 773 (770)    UK 0 (0)
30  H 1041 (-90)    F+ 33 (-302)    F- 91 (90)    UK 0 (0)

TOTAL H 31936    F+ 88    F- 5887    UK 0
1647
```

```
NGram (0.80), Word Lev
0  H 3678 (-4)    F+ 2 (-3)    F- 10 (4)    UK 0 (0)
2  H 2659 (-2)    F+ 14 (0)    F- 10 (2)    UK 0 (0)
4  H 2495 (0)    F+ 2 (-2)    F- 23 (0)    UK 0 (0)
5  H 3576 (-4)    F+ 1 (-1)    F- 4 (4)    UK 0 (0)
7  H 2196 (-36)    F+ 55 (-35)    F- 38 (36)    UK 0 (0)
8  H 2053 (-2)    F+ 38 (-237)    F- 3 (2)    UK 0 (0)
9  H 2733 (-14)    F+ 1 (0)    F- 23 (14)    UK 0 (0)
11  H 1900 (-1)    F+ 4 (-33)    F- 2 (1)    UK 0 (0)
14  H 2444 (-7)    F+ 48 (-5)    F- 12 (7)    UK 0 (0)
15  H 2653 (-4)    F+ 12 (-134)    F- 4 (4)    UK 0 (0)
17  H 2311 (-6)    F+ 1 (0)    F- 17 (6)    UK 0 (0)
21  H 1676 (-10)    F+ 0 (-31)    F- 23 (10)    UK 0 (0)
22  H 1588 (-3)    F+ 26 (-70)    F- 3 (3)    UK 0 (0)
26  H 1363 (-3)    F+ 7 (-2)    F- 3 (3)    UK 0 (0)
27  H 1357 (-2)    F+ 27 (-4)    F- 2 (2)    UK 0 (0)
28  H 1824 (-5)    F+ 18 (-5)    F- 8 (5)    UK 0 (0)
30  H 1129 (-2)    F+ 285 (-50)    F- 3 (2)    UK 0 (0)

TOTAL H 37635    F+ 541    F- 188    UK 0


NGram (0.80), ranged accept
0  H 3681 (1)    F+ 2 (0)    F- 7 (-1)    UK 0 (0)
2  H 2661 (0)    F+ 14 (0)    F- 8 (0)    UK 0 (0)
4  H 2495 (0)    F+ 1 (0)    F- 23 (0)    UK 0 (0)
5  H 3572 (1)    F+ 1 (0)    F- 8 (-1)    UK 0 (0)
7  H 2227 (34)    F+ 51 (0)    F- 7 (-34)    UK 0 (0)
8  H 2054 (0)    F+ 41 (11)    F- 2 (0)    UK 0 (0)
9  H 2731 (12)    F+ 1 (0)    F- 25 (-12)    UK 0 (0)
11  H 1901 (0)    F+ 1 (0)    F- 1 (0)    UK 0 (0)
14  H 2420 (7)    F+ 41 (0)    F- 36 (-7)    UK 0 (0)
15  H 2640 (1)    F+ 7 (0)    F- 17 (-1)    UK 0 (0)
17  H 2290 (4)    F+ 0 (0)    F- 38 (-4)    UK 0 (0)
21  H 1672 (3)    F+ 0 (0)    F- 27 (-3)    UK 0 (0)
22  H 1588 (0)    F+ 21 (0)    F- 3 (0)    UK 0 (0)
26  H 1361 (1)    F+ 0 (0)    F- 5 (-1)    UK 0 (0)
27  H 1356 (0)    F+ 26 (0)    F- 3 (0)    UK 0 (0)
28  H 1828 (1)    F+ 5 (0)    F- 4 (-1)    UK 0 (0)
30  H 1126 (0)    F+ 84 (17)    F- 6 (0)    UK 0 (0)

TOTAL H 37603    F+ 296    F- 220    UK 0
```

Appendix C

Source code

C.1 Source for duplicate detection

Listing C.1: main.cpp

```
#incdlude <iostream>
#include <pthread.h>
#include <vector>
#include <fstream>
#include "PmcHandler.h"
#include "ResultHandler.h"
#include "CheckResults.h"

using namespace std;

void * startpmc(void * arg)
{
  ((PmcHandler*)arg)->Start();
  return 0;
}

void MakeIndex(char * data_file_name, vector<size_t>& index)
{
  ifstream data_file(data_file_name);
  string s;
  do index.push_back(data_file.tellg());
    while (getline(data_file,s));
  data_file.close();
}
int main(int argc, char ** argv)
{
  if (argc != 5)
  {
    cout << "Usage :" << argv[0]
         << " <queries> <data> <answers> <result>\n";
    return 1;
  }

  cout << "Making index... ";
  vector<size_t> index;
  MakeIndex(argv[2], index);
  cout << "Done.\n";

  ResultHandler result_handler(argv[2], argv[1], index);
  PmcHandler pmc_handler(argv[2], argv[1], result_handler);

  pthread_t thread;
  pthread_create(&thread,NULL, startpmc,(void*)&pmc_handler);aaaaaa
  result_handler.Search();

  result_handler.SaveResults(argv[4]);
  cout << "Checking results...\n";
  ifstream fin(argv[1]);
  size_t i = 0;
  string s;
  while (getline(fin,s)) ++i;
  CheckResults result_checker(argv[2],argv[3],i,index);
  result_checker.Check(result_handler.records());
  result_checker.SaveScores();
  return 0;
}
```

Listing C.2: BuildQuery.h

```
#ifndef _BUILDQUERY_H
#define _BUILDQUERY_H
#include <string>

/**
 * The BuildQuery namespace contains tools for building IQL queries
 */
namespace BuildQuery
{
  bool Optional(std::string);
  std::string Build(std::string);
  std::string BuildFuzzySimple(std::string);
  std::string BuildFuzzyComplex(std::string);
  std::string BuildNGram(std::string);
};

#endif
```

Listing C.3: BuildQuery.cpp

```
#include <sstream>
#include <vector>
#include <pmcapi/exception/PmcApiException.h>
#include "BuildQuery.h"

#define BIGRAM 2

using namespace std;

bool NoCaseCompare(string a,string b)
{
  if (a.size() != b.size()) return false;
  for (size_t i = 0; i < a.size(); ++i)
    if (tolower(a[i]) != tolower(b[i])) return false;
  return true;
}

string WrapWord(string word)
{
  size_t p = word.size();
  if (p > 6) p -= 2;
  else if (p > 3) p -= 1;
  ostringstream ss;
  ss << '{' << word << ":c=1,p>=" << p << '}';
  return ss.str();
}

vector<string> NGrams(vector<string> words,int n)
{
  vector<string> ngrams;
  for (size_t i = 0; i < words.size(); ++i)
    for (int j = 0; j <= (int)words[i].size()-n; ++j)
      ngrams.push_back(words[i].substr(j,n));
  return ngrams;
}

/**
 * Returns true if the word is not a significant part of an artist's
 * name.
 * \param word The word to be checked
 * \return True if the word is not significant
 */
bool BuildQuery::Optional(string word)
{
  return NoCaseCompare(word,"the") ||
         NoCaseCompare(word,"and") ||
         NoCaseCompare(word,"an")  ||
         NoCaseCompare(word,"a");
}

/**
 * Constructs an IQL query. With p(n-of-m) >= length-2 for
 * length > 6, p >= length-1 for length > 3, and p >= length
 * for all else. Also c (character latency) is set to 1.
 * "Optional" words (see above) are removed, and all other words
 * are split up in seperate patterns and joined with 'and'
 * \param s The string which is to be searched for
 * \return The finished IQL query.
 */
string BuildQuery::Build(string s)
{
  return BuildFuzzyComplex(s);
}

/**
 * The following are the functions for constructing the various IQL
 * queries. More details on these patterns can be found in Section 5
 * of my project.
 */
string BuildQuery::BuildFuzzySimple(string s)
{
  size_t p = s.size();
  if (p > 12) p -= 3;
  else if (p > 6) p -= 2;
  else if (p > 3) p -= 1;
  ostringstream ss;
  ss << "{\"" << s << "\":c=1,p>=" << p << "} before \\t";
  return ss.str();
}
string BuildQuery::BuildFuzzyComplex(string s)
{
  string t;
  vector<string> v;
  istringstream strm(s);
  /* Build a vector of all non-optional words */
  while (getline(strm,t,' '))
    if (!t.empty() && !Optional(t))
      v.push_back(t);
  if (v.empty())
    throw PmcApi::IqlError("Unable to create IQL query from: " + s);
  t = WrapWord(v[0]);
  for (size_t i = 1; i < v.size(); ++i)
    t += " and " + WrapWord(v[i]);
  t += " before \\t";
  return t;
}

string BuildQuery::BuildNGram(string s)
{
  string t;
  vector<string> v;
  istringstream strm(s);
  while (getline(strm,t,' '))
    if (!t.empty() && !Optional(t))
      v.push_back(t);
  vector<string> ngrams = NGrams(v,BIGRAM);
  if (v.empty())
    throw PmcApi::IqlError("Unable to create IQL query from: " + s);
  ostringstream ss;
  ss << '{' << ngrams[0];
  for (size_t i = 1; i < ngrams.size() && i < 16; ++i)
    ss << ',' << ngrams[i];
  ss << ":n>=" << min(16,(int)(0.5 + ngrams.size()*0.80))
       << "} before \\t";
  return ss.str();
}
```

Listing C.4: CheckResults.h

```
#ifndef _CHECKRESULTS_H
#define _CHECKRESULTS_H
#include <vector>
#include <string>
#include <iostream>
#include "Types.h"

/**
 * The CheckResults class is responsible for comparing the results
 * obtained against the annotate dataset and outputting the results
 * of the comparison.
 */
class CheckResults
{
  struct Answer
  {
    std::vector<size_t> yes;
    std::vector<size_t> no;
  };
  struct Score
  {
    std::vector<size_t> hit;
    std::vector<size_t> false_pos;
    std::vector<size_t> false_neg;
    std::vector<size_t> unknown;
  };

  public:
    CheckResults(char*,char*,size_t ,std::vector<size_t >&);
    void SaveScores();
    void Check(std::vector<Record>&);
  private:
    std::vector<Answer> answers_;
    std::vector<Score> scores_;
    char * data_file_name_;
    std::vector<size_t>& index_;

    void ReadAnswers(char*);
    bool Hit(size_t ,size_t );
    bool Miss(size_t ,size_t );
    void GetPreviousScores(std::vector< std::vector<size_t> >&);
    void WriteScores();
    std::string GetLine(std::ifstream&,size_t );
};

#endif
```

Listing C.5: CheckResults.cpp

```
#include <sstream>
#include <fstream>
#include "CheckResults.h"

using namespace std;

string toBase64(int d)
{
  int r = d % 64;
  char c;
  if (r < 26) c = 'A' + r;
  else if (r < 52) c = 'a' + (r - 26);
  else if (r < 62) c = '0' + (r - 52);
  else if (r == 62) c = '+';
  else c = '/';
  if (d - r) return toBase64((d-r)/64) + string(1,c);
  else return string(1,c);
}

const char * LAST_RESULTS_ = "lastresults";
const char * UNKNOWNS_ = "unknowns";
const char * RESULT_LOG_ = "results.log";

/**
 * The constructor reads in the annotation data.
 * \param data_file_name The name of the file containing the
 *              searched data
 * \param answer_file_name The name of the file containing the
 *                annotations
 * \param artist_size The number of artists to be searched for
 * \param index The index file belonging to the searched data file
 */
CheckResults::CheckResults(char * data_file_name,
                char * answer_file_name,
                      size_t artists_size,
                vector<size_t>& index) :
  answers_(artists_size),
  scores_(artists_size),
  data_file_name_(data_file_name),
  index_(index)
{
  ReadAnswers(answer_file_name);
}

/* This could be done more efficiently by just writing directly to
 * answers_ -- just do base64 to base10 and write to the equivalent
 * answer_ entry.
 */
void CheckResults::ReadAnswers(char * answer_file_name)
{
  vector< vector<string> > yes; /* vector of records matching */
  vector< vector<string> > no; /* vector of nonmatching records */
  ifstream answer_file(answer_file_name);
  string s;
  while (getline(answer_file,s))
  {
    yes.push_back(vector<string >());
    no.push_back(vector<string >());
    istringstream strm(s);
    while (getline(strm,s,'|'))
    {
      if (!s.empty() && s[0] != '-')
        yes.back().push_back(s);
      else if (!s.empty())
        no.back().push_back(s.substr(1));
    }
  }
  answer_file.close();

  /* For each artist find for what lines it's been annotated
   * store store in the answer_ vector.
   */
  for (size_t i = 0; i < answers_.size(); ++i)
  {
    string id = toBase64(i);
    for (size_t j = 0; j < yes.size(); ++j)
    {
      for (size_t k = 0; k < yes[j].size(); ++k)
        if (id == yes[j][k])
          answers_[i].yes.push_back(j);
      for (size_t k = 0; k < no[j].size(); ++k)
        if (id == no[j][k])
          answers_[i].no.push_back(j);
    }

  }
}

/**
 * Checks a given set of records against the annotations and
 * stores the results locally.
 * \param records The records to be checked
 */
void CheckResults::Check(vector<Record>& records)
{
  if (records.size() != answers_.size())
    cout << "Sizes don't match up... Continuing anyway.\n";

  for (size_t i = 0; i < records.size(); ++ i)
  {
    for (size_t j = 0; j < records[i].neighbours.size(); ++j)
    {
      if (Hit(i,records[i].neighbours[j]))
        scores_[i].hit.push_back(records[i].neighbours[j]);
      else if (Miss(i,records[i].neighbours[j]))
        scores_[i].false_pos.push_back(records[i].neighbours[j]);
      else
        scores_[i].unknown.push_back(records[i].neighbours[j]);
    }
    /*Find all lines annotated as hits but not found by search*/
    for (size_t j = 0; j < answers_[i].yes.size(); ++j)
      if (find(records[i].neighbours.begin(),
          records[i].neighbours.end(),
          answers_[i].yes[j]) == records[i].neighbours.end())
        scores_[i].false_neg.push_back(answers_[i].yes[j]);
  }
}


bool CheckResults::Hit(size_t query_id, size_t record)
{
  for (size_t i = 0; i < answers_[query_id].yes.size(); ++i)
    if (record == answers_[query_id].yes[i])
      return true;
  return false;
}

bool CheckResults::Miss(size_t query_id, size_t record)
{
  for (size_t i = 0; i < answers_[query_id].no.size(); ++i)
    if (record == answers_[query_id].no[i])
      return true;
  return false;
}

string CheckResults::GetLine(ifstream& file, size_t line)
{
  file.seekg(index_[line]);
  string s;
  getline(file,s);
  return s;
}

void
CheckResults::GetPreviousScores
  (vector< vector<size_t> >& prev_scores)
{
  size_t score;
  ifstream last_results(LAST_RESULTS_);
  int count = 0;
  while (last_results >> score)
  {
    if (count % 4 == 0) prev_scores.push_back(vector<size_t >());
    prev_scores.back().push_back(score);
    ++count;
  }
  last_results.close();
}

void CheckResults::WriteScores()
{
  ofstream results(LAST_RESULTS_);
  for (size_t i = 0; i < scores_.size(); ++i)
    results  << scores_[i].hit.size() << ' '
       << scores_[i].false_pos.size() << ' '
       << scores_[i].false_neg.size() << ' '
       << scores_[i].unknown.size() << '\n';
  results.close();

  ofstream log_file(RESULT_LOG_);
  ifstream data(data_file_name_);

  for (size_t i = 0; i < scores_.size(); ++i)
  {
    log_file << "HITS\n";
    for (size_t j = 0; j < scores_[i].hit.size(); ++j)
      log_file << GetLine(data,scores_[i].hit[j]) << '\n';
    log_file << "\nFALSE POSITIVES\n";
    for (size_t j = 0; j < scores_[i].false_pos.size(); ++j)
      log_file << GetLine(data,scores_[i].false_pos[j]) << '\n';
    log_file << "\nFALSE NEGATIVES\n";
    for (size_t j = 0; j < scores_[i].false_neg.size(); ++j)
      log_file << GetLine(data,scores_[i].false_neg[j]) << '\n';
    log_file << "\nUNKNOWN\n";
    for (size_t j = 0; j < scores_[i].unknown.size(); ++j)
      log_file << GetLine(data,scores_[i].unknown[j]) << '\n';
    log_file << "\n\n\n";
  }
  data.close();
  log_file.close();

  ofstream unknown_file(UNKNOWNS_);
  for (size_t i = 0; i < scores_.size(); ++i)
  {
    unknown_file << (size_t)-1 << '\n';
    for (size_t j = 0; j < scores_[i].unknown.size(); ++j)
      unknown_file << scores_[i].unknown[j]  << '\n';
  }

}
/**
 * Saves the scores obtained with Check() to file. Filenames are:
 * results.log - the actual records; hits, false positives &
 *              negatives, and unknowns
 * unknowns - the line numbers for the unknown records (lines that
 *              need to be annotated)
 * lastresults - summary of the most recently obtained results, to
 *                be used for comparison
 * in the next run.
 */
void CheckResults::SaveScores()
{
  vector< vector<size_t> > prev_scores;
  GetPreviousScores(prev_scores);
  bool diff = prev_scores.size() == scores_.size();

  vector<size_t> totals(4,0);
  for (size_t i = 0; i < scores_.size(); ++i)
    if (scores_[i].hit.size())
    {
      cout << i << "  "
```

```
          << "Hits: " << scores_[i].hit.size() << ' ';
        if (diff)
      {
       cout << "("
           << (int)(scores_[i].hit.size()-prev_scores[i][0]) << ") ";
      }
        cout << " False pos: " << scores_[i].false_pos.size() << ' ';
        if  (diff)
      {
       cout << "("
           << (int)(scores_[i].false_pos.size() - prev_scores[i][1])
       << ") ";
      }
        cout << " False neg: " << scores_[i].false_neg.size() << ' ';
        if (diff)
      {
       cout <<  "("
           << (int)(scores_[i].false_neg.size() - prev_scores[i][2])
       << ") ";
      }
        cout << " Unknown: " << scores_[i].unknown.size() << ' ';
        if (diff)
      {
       cout << "("
           << (int)(scores_[i].unknown.size() - prev_scores[i][3])
       << ")";
      }
        cout << '\n';
        totals[0] += scores_[i].hit.size();
        totals[1] += scores_[i].false_pos.size();
        totals[2] += scores_[i].false_neg.size();
        totals[3] += scores_[i].unknown.size();
      }
  cout << "\nTOTAL Hits: " << totals[0]
       << "    False pos: " << totals[1]
     << "    False neg: " << totals[2]
       << "    Unknown: " << totals[3] << '\n';
  WriteScores();
}
```

Listing C.6: PmcHandler.h

```
#ifndef _PMCHANDLER_H
#define _PMCHANDLER_H
#include <vector>
#include <pmcapi/PmcApi.h>
#include "Types.h"
#include "ResultHandler.h"

/**
 * The PmcHandler class interfaces with the PmcApi, runs the
 * search and reports the results to the ResultHandler.
 */
struct PmcHandler
{
  typedef PmcApi::CompleteReportMode ReportMode;
  typedef PmcApi::FirstDocHitSelectMode HitSelectMode;
  typedef PmcApi::DocumentResultBuffer ResultBufferType;
  typedef ResultBufferType::iterator ResultIter;
  typedef size_t QueryId;

  PmcHandler(char*,char*,ResultHandler&);
  void Start();
  void Process(const ResultBufferType&,
               const PmcApi::Content&, QueryId);
  void SearchCompleted(const PmcApi::Content&, QueryId);
  void Error(const PmcApi::PmcApiException&,
             const PmcApi::Content&, QueryId);

  private:
  ResultHandler& result_handler_;
  char * data_file_;
  char * query_file_;
  std::vector<Query *> queries_;
  std::vector<Query *> temp_;
};

#endif
```

Listing C.7: PmcHandler.cpp

```
#include <flip/Flip.h>
#include <pmcapi/PmcApi.h>
#include <iostream>
#include <fstream>
#include "PmcHandler.h"
#include "UndupeConfigurationSource.h"

using namespace std;

/**
 * Constructs the PmcHandler struct.
 * \param data_file The name of the file with the data to be
 *                  searched
 * \param query_file The name of the file containing the data from
 *                   which to construct queries
 * \param result_handler The ResultHandler to which the results
                         should be written
 */
PmcHandler::PmcHandler(char * data_file,
               char * query_file,
               ResultHandler& result_handler) :
  result_handler_(result_handler),
  data_file_(data_file),
  query_file_(query_file)
{
}

/**
 * Stores the results of the queries locally.
 */
void PmcHandler::Process(const ResultBufferType& results,
               const PmcApi::Content&,
               QueryId qid)
{
  if (qid >= queries_.size())
    for (size_t i = queries_.size(); i <= qid; ++i)
      queries_.push_back(new Query(i));

  for (ResultIter it = results.begin(); it != results.end(); ++it)
    queries_[qid]->matches.push_back(it->GetDocumentNo());
}
/**
 * Writes completed queries to the ResultHandler at given intervals
 */
void PmcHandler::SearchCompleted(const PmcApi::Content&,
                    QueryId qid)
{
  temp_.push_back(queries_[qid]);
  /* This is often called several times for each query id so
   * make sure it doesn't write to something the ResultHandler
   * already has used and/or freed.*/
  queries_[qid] = new Query(qid);
  if (temp_.size() >= 20000)
  {
    cout << "PMC: Writing results to buffer...\n";
    result_handler_.WriteResults(temp_);
  }
}

/**
 * Report error to cerr.
 */
void PmcHandler::Error(const PmcApi::PmcApiException& e,
               const PmcApi::Content& content,
               QueryId id)
{
  cerr << "An error occured when saerching for query no. "
       << id << " in address range "
       << content.GetAddressOffset() << " . "
       << content.GetAddressOffset() + content.GetSize()
       << ": " << e.what() << '\n';
}
```

```
/**
 * Start the search. Returns when the search is completed.
 */
void PmcHandler::Start()
{
  try
  {
    auto_ptr<PmcApi::PmcArray> pmcs(Flip::OpenAllPmcs());
    PmcApi::FileDocumentSource
    docSource(data_file_, PmcApi::DocumentManager('\n'));
    PmcApi::UndupeConfigurationSource<PmcHandler>
    configSource(query_file_,*this);
    PmcApi::StaticData::Search(*pmcs, configSource,
                               docSource, PmcApi::Throughput());
  }
  catch (exception& e)
  {
    cerr << e.what() << endl;
  }
  for (size_t i = 0; i < queries_.size(); ++i)
    delete queries_[i];
  result_handler_.WriteResults(temp_);
  result_handler_.Done();
  cout << "PMC: Done with queries.\n";
}
```

Listing C.8: ResultHandler.h

```
#ifndef _RESULTHANDLER_H
#define _RESULTHANDLER_H
#include <vector>
#include <fstream>
#include <string>
#include <set>
#include <pthread.h>
#include "Types.h"

/**
 * The ResultHandler class takes results from the PmcHandler and
 * does secondary checks on the data in an attempt to filter out
 * any false positives not caught by the PMC. It is meant to be run
 * in a seperate thread and will block until it receives data from
 * PMC.
 */
class ResultHandler
{
  public:
    ResultHandler(char*,char*,std::vector<size_t>&);
    void Search();
    void Done();
    void WriteResults(std::vector<Query*>&);
    void SaveResults(char*);
    /**
     * Access the results of the search.
     * \return A vector with the results.
     */
    std::vector<Record>& records() { return links_; }
  private:
    std::vector<size_t>& index_;
    std::vector<Record> links_;
    volatile bool done_;
    std::ifstream data_file_;
    std::vector<Query*> results_;
    std::vector<std::string> artists_;

    pthread_mutex_t result_mutex_;
    pthread_cond_t result_write_;

    void ReadResults();
    void Evaluate(Query*);
    bool PlainLevenshtein(std::string,std::string);
    bool WordLevenshtein(std::string,std::string);
    bool RangedAccept(std::string,std::string);
    bool Dupe(std::string,std::string);
    void MakeLinks(std::vector<Query*>&);
    size_t Levenshtein(std::string,std::string);
};
#endif
```

Listing C.9: ResultHandler.cpp

```
//#include <sstream>
#include <iostream>
#include <sstream>
#include "ResultHandler.h"
#include "BuildQuery.h"

using namespace std;


vector<string> Tokenize(string s)
{
  string t;
  vector<string> v;
  istringstream strm(s);
  while (getline(strm,t,' ')) v.push_back(t);
  return v;
}

string GetField(string s,int field)
{
  string t;
  istringstream strm(s);
  for (int i = 0; i <= field; ++i) getline(strm,t,'\t');
  return t;
}

/**
 * The constructor prepares the ResultHandler class, but it will not
 * start reading results until Searc() is called.
 */
ResultHandler::ResultHandler(char * data_file_name,
                char * config_file_name,
                              vector<size_t>& index) :
  index_(index),
  done_(false),
  data_file_(data_file_name)
{
  pthread_mutex_init(&result_mutex_,NULL);
  pthread_cond_init(&result_write_,NULL);

  ifstream config_file(config_file_name);
  string s;
  while (getline(config_file,s)) artists_.push_back(s);
}

/**
 * Starts the result handling process. The ResultHandler will wait
 * for results, process these, and repeat until the PmcHandler
 * signals it's done.
 */
void ResultHandler::Search()
{
  while (!done_) ReadResults();
}

/**
 * Called by the PmcHandler when it is done outputting results.
 */
void ResultHandler::Done()
{
  done_ = true;
   /* For freeing stuck ResultHandlers */
  pthread_mutex_lock(&result_mutex_);
  pthread_cond_signal(&result_write_);
  pthread_mutex_unlock(&result_mutex_);
}

/**
 * Write results to the ResultHandler.
 * \param results A vector with pointers to the results. Note that
 *    the vector will be empty after this function is called, and
 *    all the pointers passed WILL BE FREED by the ResultHandler.
 */
```

```
void ResultHandler::WriteResults(vector<Query *>& results)
{
  pthread_mutex_lock(&result_mutex_);
    results_.insert(results_.end(), results.begin(), results.end());
    pthread_cond_signal(&result_write_);
  pthread_mutex_unlock(&result_mutex_);
  results.clear();
}
void ResultHandler::ReadResults()
{
  cout << "RH : Waiting for results...\n";
  vector<Query *> results;
  pthread_mutex_lock(&result_mutex_);
    if (results_.empty())
      pthread_cond_wait(&result_write_,&result_mutex_);
    results.swap(results_);
  pthread_mutex_unlock(&result_mutex_);
  cout << "RH : Read " << results.size() << " results.\n";

  for (size_t i = 0; i != results.size(); ++i) Evaluate(results[i]);
  cout << "RH : Evaluted results.\n";
  MakeLinks(results);
  cout << "RH : Made links.\n";
  for (size_t i = 0; i < results.size(); ++i) delete results[i];
  cout << "RH : Done processing results.\n";
}

void ResultHandler::Evaluate(Query * result)
{
  vector<size_t >::iterator it = result->matches.begin();
  while (it != result->matches.end())
  {
    data_file_.seekg(index_[*it]);
    string match;
    getline(data_file_,match);
    if (Dupe(artists_[result->query_id],match)) ++it;
    else it = result->matches.erase(it);
  }
}

bool ResultHandler::PlainLevenshtein(string a, string b)
{
  if (a == b) return true;
  double threshold = 0.2 * (double)a.size();
  return Levenshtein(a,b) <= (unsigned int)(threshold + 0.5);
}

bool ResultHandler::WordLevenshtein(string a, string b)
{
  if (a == b) return true;
  vector<string> a_words,b_words;
  string t;
  istringstream strma(a);
  while (getline(strma,t,' '))
    if (!BuildQuery::Optional(t)) a_words.push_back(t);
  istringstream strmb(b);
  while (getline(strmb,t,' '))
    b_words.push_back(t);

  vector<size_t> min_distances(a_words.size(),(size_t)-1);
  size_t sum = 0;

  for (size_t i = 0; i < a_words.size(); sum += min_distances[i++])
    for (size_t j = 0; j < b_words.size(); ++j)
    {
      if (a == b)
      {
        min_distances[i] = 0;
        break;
      }
      else
      {
        size_t dist = Levenshtein(a_words[i],b_words[j]);
        min_distances[i] = min(dist,min_distances[i]);
      }
    }

  return sum/a_words.size() <= 2;
}

bool ResultHandler::RangedAccept(string a, string b)
{
  double size_diff = b.size() / (double)a.size();
  if (size_diff < 0.8 || size_diff > 1.25)
  {
    bool match = true;
    string t;
    istringstream strm(a);
    while (getline(strm,t,' '))
      if (!BuildQuery::Optional(t))
        match *= b.find(t) != b.npos;
    return match;
  }
  else
  {
    return true;
  }
}

bool ResultHandler::Dupe(string a, string b)
{
  /* Get the first field, the artist field, of the record */
  b = GetField(b,0);
  return RangedAccept(a,b);
}

/*
 * Calculate the Levenshtein distance between string a, and b.
 */
size_t ResultHandler::Levenshtein(string a, string b)
{
  size_t width = a.size() + 1;
  size_t height = b.size() + 1;
  size_t * d = new size_t[width*height];

  for (size_t i = 0; i < width; ++i)
    d[i] = i;
  for (size_t j = 1; j < height; ++j)
    d[j*width] = j;

        unsigned int cost;
  for (size_t i = 1; i < width; ++i)
    for (size_t j = 1; j < height; ++j)
    {
      if (tolower(a[i-1]) == tolower(b[j-1])) cost = 0;
      else cost = 1;
      d[j*width + i] = min(
            d[j*width + i - 1] + 1, //deletion
            min(
              d[(j-1)*width + i] + 1, //insertion
              d[(j-1)*width + i - 1] + cost //substitution
               )
          );
    }
  size_t r = d[width*(height-1) + width-1];
  delete[] d;
  return r;
}

void ResultHandler::MakeLinks(vector<Query *>& results)
{
  for (size_t i = 0; i < results.size(); ++i)
    for (size_t j = 0; j < results[i]->matches.size(); ++j)
    {
      if (links_.size() <= results[i]->query_id)
        links_.resize(results[i]->query_id + 1);
      links_[results[i]->query_id].neighbours.push_back(
                  results[i]->matches[j]
```

```
                    );
      links_[results[i]->query_id].certainty.push_back(1);
    }
}
/**
 * Save results to a given file name. The format of the saved file
 * will bethe artist name searched for followed by all the records
 * on which the artist has been found to play.
 * \param filename The file where the data should be saved.
 */
void ResultHandler::SaveResults(char * filename)
{
  cout << "RH : Writing results to " << filename << '\n';
  ofstream result_file(filename);
  for (size_t i = 0; i < links_.size(); ++i)
  {
    result_file << artists_[i];
    string s;
    for (size_t j = 0; j < links_[i].neighbours.size(); ++j)
    {
      data_file_.seekg(index_[links_[i].neighbours[j]]);
      getline(data_file_,s);
      result_file << s << '\n';
    }
    result_file << '\n';
  }
  result_file.close();
}
```

Listing C.10: Types.h

```
#ifndef _MYTYPES_H
#define _MYTYPES_H

#include <string>
#include <vector>

struct Query
{
  Query(size_t qid) : query_id(qid) {}
  size_t query_id;
  std::vector<size_t> matches;
};

struct Record
{
  Record() : dupe(false) {}
  bool dupe;
  std::vector<size_t> neighbours;
  std::vector<double> certainty;
};

#endif
```

Listing C.11: UndupeConfigurationSource.h

```
/**
 *****************************************************************
 * Loosely based on code which is copyright of Interagon AS.
 *****************************************************************/

#ifndef PMCAPI_UNDUPECONFIGURATIONSOURCE_H_
#define PMCAPI_UNDUPECONFIGURATIONSOURCE_H_

#include <string>
#include <fstream>
#include <pmcapi/api/ConfigurationSource.h>
#include <pmcapi/config/ResultProcessorHolder.h>
#include <pmcapi/sources/FastPmcPslConfigurationSource.h>
#include <fastpmc/iql/Iql.h>
#include <fastpmc/iql_parser/IqlParser.h>
#include <fastpmc/mapper/Mapper.h>
#include <fastpmc/psl/PSL_Node.h>
#include <pmcapi/config/Configuration.h>
#include <pmcapi/exception/PmcApiException.h>
#include "BuildQuery.h"

namespace PmcApi {

/**
 * Configuration source creating configurations from a set of IQL
 * queries. The source combines the queries to keep the number of
 * required configurations at a minimum.
 */
template <class ResultProc>
class UndupeConfigurationSource : public ConfigurationSource {
  FastPmcPslConfigurationSource pslConfigSource_;

  ResultProc& result_processor_;
  std::ifstream source_file_;
  size_t query_id_;

  public:
    UndupeConfigurationSource(const char * source_file_name,
                 ResultProc& result_processor)
      : result_processor_(result_processor),
        source_file_(source_file_name),
        query_id_(0)
      {
    }

    virtual std::auto_ptr<Configuration> GetConfiguration();
    virtual bool WillBlock() const
    {
      return pslConfigSource_.WillBlock() || !source_file_;
    }
    virtual bool EndOfSource() const
    {
      return pslConfigSource_.EndOfSource() && !source_file_;
    }

  private:
    static std::auto_ptr<FastPMC::PSL_Node>
      CreatePsl(const std::string& iql);
    void LoadQueries(size_t n);
};
}

template <class ResultProc>
void
PmcApi::UndupeConfigurationSource<ResultProc>::
LoadQueries(size_t n)
{
  std::string artist;
  while (n-- && std::getline(source_file_, artist))
  {
    /* # can be used to comment out artists not to be included
     * without messing with the artist ids used during annotation */
    if (artist[0] != '#')
```

```
      pslConfigSource_.AddPsl(CreatePsl(BuildQuery::Build(artist)),
                    result_processor_, query_id_);
    std::cout << BuildQuery::Build(artist) << '\n';
    ++query_id_;
  }
}

template <class ResultProc>
std::auto_ptr<PmcApi::Configuration>
PmcApi::UndupeConfigurationSource<ResultProc>::GetConfiguration()
{
  if (source_file_ && pslConfigSource_.GetPslCount() < 51200)
    LoadQueries(51200);
  return pslConfigSource_.GetConfiguration();
}

template <class ResultProc>
std::auto_ptr<FastPMC::PSL_Node>
PmcApi::UndupeConfigurationSource<ResultProc>::
CreatePsl(const std::string& iql)
{
  try
  {
    return FastPMC::Mapper::Map(*FastPMC::IqlParser::Parse(iql));
  }
  catch (std::exception& e)
  {
    throw PmcApi::IqlError(
                "Invalid IQL query \"" + iql + "\". " + e.what()
                );
  }
}

#endif
```

C.2 Source used to generate data

Listing C.12: mkdata.cpp

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <sstream>
#include <math.h>
#include <stdlib.h>
#include <time.h>

using namespace std;

static double DUP_RATE = 0.05;
static int MAX_DUP_PER_REC = 50;
static int MAX_ERR_PER_REC = 3;

static int NUM_OF_FILES = 1;
static int LINES_PER_FILE = 720000;

static int NAMES_PER_LINE = 3;

vector<string> names;
int dupeid = 0;

string toString(int i)
{
  stringstream ss;
  ss << i;
  return ss.str();
}

double myRand(double low, double high)
{
  return rand() / (RAND_MAX + 1.0) * (high-low) + low;
}

string toBase64(int d)
{
  int r = d % 64;
  char c;
  if (r < 26) c = 'A' + r;
  else if (r < 52) c = 'a' + (r - 26);
  else if (r < 62) c = '0' + (r - 52);
  else if (r == 62) c = '+';
  else c = '/';
  if (d - r) return toBase64((d-r)/64) + string(1,c);
  else return string(1,c);
}

string getid(bool inc = false)
{
  if (inc) return toBase64(dupeid++);
  else return toBase64(dupeid);
}

void loadnames(char * namefile)
{
  ifstream fin(namefile);
  string buff;
  while (getline(fin,buff)) names.push_back(buff);
  fin.close();
}

string getline()
{
  string name = names[(int)myRand(0,names.size())];
  for (int i = 1; i < NAMES_PER_LINE; ++i)
    name += " " + names[(int)myRand(0,names.size())];
  return name;
```

```
}

string mkdupe(string source)
{
  int errors = (1+(MAX_ERR_PER_REC)*pow(myRand(0,1),10.0));
  for (int i = 0; i < errors; ++i)
  {
    int pos = (int)myRand(1,source.size());
    int type = (int)myRand(0,4);
    switch (type)
    {
      case 0:  if (!pos) swap(source[pos],source[pos+1]);
        else swap(source[pos],source[pos-1]);
        break;
      case 1: source[pos] = (char)myRand('a','z'+1);
        break;
      case 2: source.insert(pos,1,(char)myRand('a','z'+1));
        break;
      case 3: source.erase(pos,1);
    }

  }
  return source;
}

void makefile(int n)
{
  cout << "Making data " << n << '\n';
  vector<string> lines;
  vector<string> dupelog;
  size_t dupecount = 0;;
  while (lines.size() < LINES_PER_FILE)
  {
    lines.push_back(getline());
    dupelog.push_back("");
    if (myRand(0,1) < DUP_RATE)
    {
      string line = lines.back();
      dupelog.back() = getid(true);
      int dupes = (1+(MAX_DUP_PER_REC-1)*pow(myRand(0,1),10.0));
      for (int i = 0; i < dupes &&
          lines.size() < LINES_PER_FILE; ++i)
      {
        lines.push_back(mkdupe(line));
        dupelog.push_back(getid() + " *");
        dupecount++;
      }
    }
  }
  cout << 100.0 * (double)dupecount/LINES_PER_FILE << "% dupes.\n";
  cout << "Writing file " << n << '\n';
  ofstream flines(string("data/" + toString(n)).c_str());
  ofstream fdupes(string("data/" + toString(n) + ".dupes").c_str());
  while (!lines.empty())
  {
    int k = (int)myRand(0,lines.size());
    flines << lines[k] << '\n';
    fdupes << dupelog[k] << '\n';
    lines[k] = lines.back();
    dupelog[k] = dupelog.back();
    lines.pop_back();
    dupelog.pop_back();
  }
  flines.close();
  fdupes.close();
}

void mergefiles(char * outfile)
{
  cout << "Merging files.\n";
  vector<ifstream *> files;
  vector<ifstream *> dupefiles;
  for (int i = 0; i < NUM_OF_FILES; ++i)
  {
    files.push_back(
      new ifstream(
        string("data/" + toString(i)).c_str()
        ));
    dupefiles.push_back(
      new ifstream(
        string("data/" + toString(i) + ".dupes").c_str()
        ));
  }

  ofstream flines(outfile);
  string dupefile = outfile;
  dupefile += ".dupes";
  ofstream fdupes(dupefile.c_str());
  while (!files.empty())
  {
    string buff;
    int r = (int)myRand(0,files.size());
    if (getline(*files[r],buff))
    {
      flines << buff << '\n';
      getline(*dupefiles[r],buff);
      fdupes << buff << '\n';
    }
    else
    {
      files[r]->close();
      dupefiles[r]->close();
      delete files[r];
      delete dupefiles[r];
      files[r] = files.back();
      dupefiles[r] = dupefiles.back();
      files.pop_back();
      dupefiles.pop_back();
    }
  }
  flines.close();
  fdupes.close();
}

int main(int argc, char ** argv)
{
  if (argc != 3)
  {
    cout << "Use: " << argv[0] << " <namefile> <outfile>\n";
    return -1;
  }
  srand(time(0));
  cout << "Loading names.\n";
  loadnames(argv[1]);
  cout << "Loaded " << names.size() << " names.\n";

  for (int i = 0; i < NUM_OF_FILES; ++i) makefile(i);
  mergefiles(argv[2]);
  cout << "Done.\n";
}
```