Cleaning Customer Data

Jack Smodey
CS 395
Spring 2006

Table of Contents

Intro to the Department of Natural Resources

The Alaska Department of Natural Resources' role as a state department is to manage all state-owned land, water, and resources, excluding fish and game, for the state of Alaska. The state owns 65 million acres of tidelands, shorelands, and submerged lands as well as 34,000 miles of coastline. DNR also manages the State's fresh water resources, which accounts to about 40% of the entire Nation's fresh water. The department is divided into seven divisions, which include the Division of Agriculture, Forestry, Geological and Geophysical Surveys (DGGS), Mining, Land & Water, Oil & Gas, Park and Outdoor Recreation, and Support Services.

A sub-department of Support Services, the Land Resource Information Section (LRIS) maintains the department's land records repository and maintains DNR's computer system and network services. Subunits include the Computer Information Center, Status Graphics Unit, Geographics Information System, and the Business Programming unit (BPU). Projects that are managed by BPU include software integration, Web application development, database development, and more.

Project Overview

During the interview process, manager of the BPU John Casey, my immediate supervisor informed me that I would be hired to do an analysis and cleaning of the customer database. This cleaning would include writing software that would compile a list of duplicate customer data which would then be used as input for another piece of software that would combine all occurrences of duplicate data. The software that would do the combining would be authored by BJ McJimsey, an analyst programmer level IV. I would be working in a Microsoft Window's environment using Oracle Workstation and its various tools to complete my project.

Research

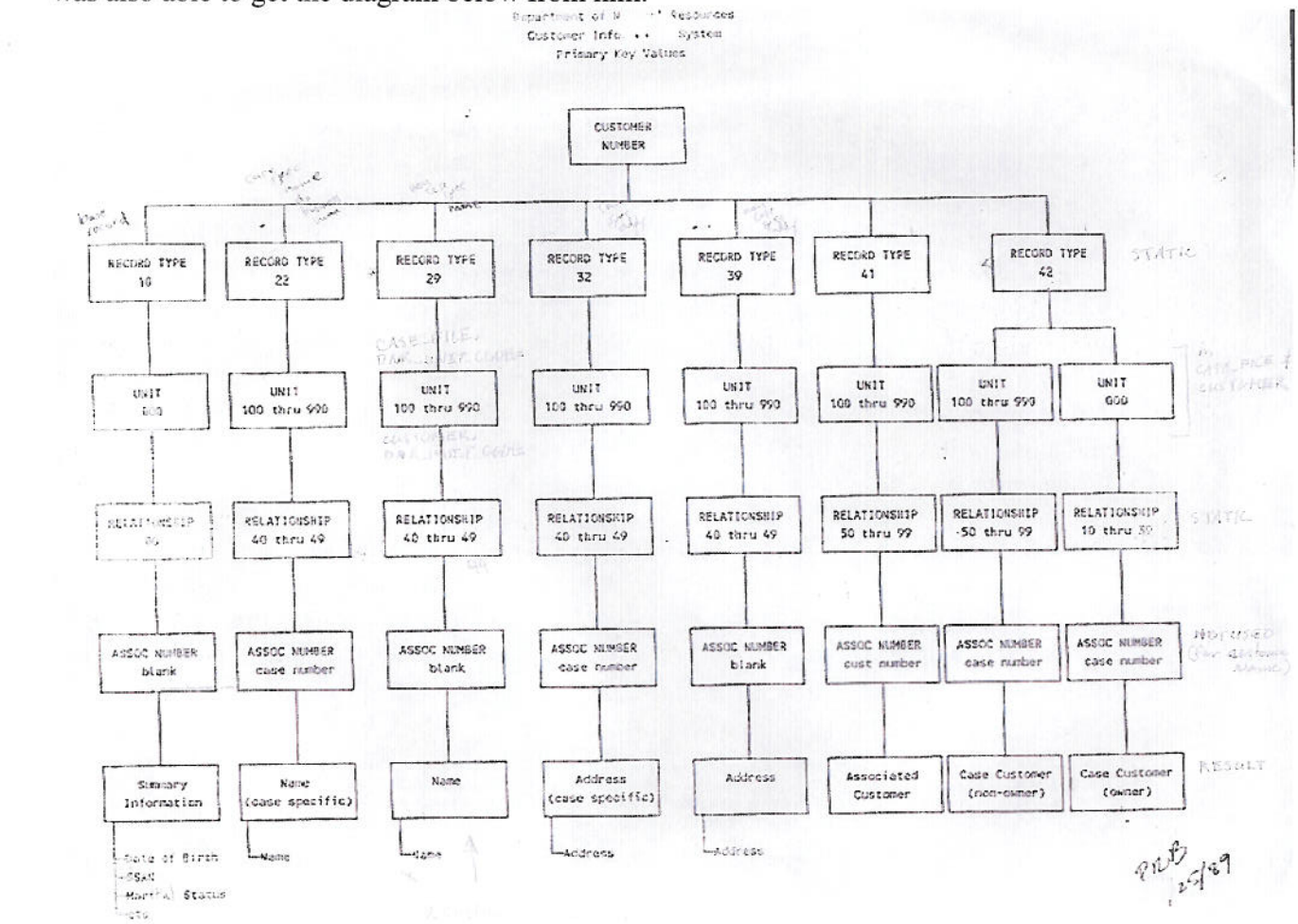
Student interns are extremely valuable to the LRIS, and usually make great candidates for full-time hire. I was told that in the BPU, it usually takes three months for a new hire programmer to learn and understand the systems that they maintain and develop. After this period, if a programmer is still in the dark, they can usually expect to be terminated so that department funds are not wasted. The process is then repeated until a suitable employee is found. A student intern as a new hire is desirable due to the fact that they are paid much less than a normal hire, and are generally enthusiastic to learn outside of a classroom setting.

On my first day on the job, I spent much of my time being introduced to my co-workers. Naturally most of them were curious to what my project entailed. Apparently, my project had been on the to-do list for some time and had become affectionately

referred to as "The Briar Patch." Unshaken, with my elbows pointed outward and with a straight back, I marched to my cubicle to start my work.

I later found out that I was intentionally given limited information on the data I was working with to see if I had the ability to ask my co-workers for leads to find documentation. The "Customer Database" itself is not built in a multi-table relational model like I practiced on in school. Instead, it is one large table organized through a series of columns that hold numerical values which identify what the rest of a given row contains. Not realizing this at first, I was shocked to see that out of 777,295 rows, only 120,522 (15.5%) had a name field that was not null. After a week of trying to figure out the layout of this massive table, I was forced to ask around for documentation and advice. Jason Kettel, a former BP intern and graduate of UAA, understood my frustration and became an invaluable part in helping me find the information I needed.

Robert Clement, Supervisor of Geographic Information Systems and long time employee in LRIS, had been one of the founding fathers of the table I was working with. Jason and I sat down with him one morning and discussed the problem in depth. Mr. Clement was able to give some insight to why they developed the customer system in that fashion. In older systems, it was a much more efficient design than the relational model. I was also able to get the diagram below from him.



This diagram illustrates a trail of column values that ultimately lead to different parts of customer information. This information wasn't all obtained in one sitting, but ultimately the basic breakdown of the table is as follows:

Customer Number

Customer number refers to the unique key that identifies a given customer.

Record Type

Record type simply indicates what type of information we're dealing with in a given row

record_type = 10 → Base record. Contains SSN and DOB (94,485 rows)
record_type = 22 → Case specific name (1395 rows)
record_type = 29 → Non-case specific name (119127 rows)
record_type = 32 → Case specific address (882 rows)
record_type = 39 → Non-case specific address (117,721 rows)
record_type = 41 → Special customer-customer relation (281 rows)
record_type = 42 → Case file (443,404 rows)

Unit

"Unit" or "dnr_unit_code" as it appears in the table, refers to what division of DNR that information is related to. This allows for different divisions to create and secure their own customer information. The Unit codes correspond as follows:

dnr_unit_code = 100 → Title Administration
dnr_unit_code = 150 → Mental Health Trust Land Unit
dnr_unit_code = 200 → Land Management
dnr_unit_code = 250 → Land Records Info Section
…
…
dnr_unit_code = 800 → Water
dnr_unit_code = 900 → DGGS

Relationship

Relationship explains what type of record we're dealing with in regards to record type. For instance: In a row marked record type 29 (non case specific name) can be a relationship_code = 49 (default name/address) or a relationship_code = 45 (alternate name/address). These values are not too important to the project since I was told to use only default values.

Associated Number

Associated Number usually refers to a case file of some sort. A case file can be anything from a signed document to a deed or even a map. For record type 41, the Assoc Number field will contain a customer number, which signifies some sort of relationship between the original customer number and the new one. Record types 10, 29, and 39 contain null values for this field. Assoc Number is found as associated_number in the Customer table.

After finding this information, it was time to do a few checks on the integrity of the data. If record type 10 was the base record, then there should be no repeat primary key (customer_number) values. Running a simple group by query revealed no duplicates. The next step was to check for orphan customer numbers in the other record types. These would be customer numbers from other record types that didn't have a base record with the same number. To my reassurance, the query also reported 0 results.

With these results and others, I attended a short meeting with John and BJ to go over the best ways to find duplicates. Record type 22,32,29, and 39 all have customer names and their respective addresses, however record type 22 and 32 are special cases where a specific customer name/address is used in relation to a case file. Record type 29 and 39 is where I should be looking. It was also noted that record type 10 contained social security numbers and dates of birth, as well as a field "customer_type" which distinguished types of customers in the following way:

customer_type = 1 → individual
customer_type = 2 → company
customer_type = 3 → government
customer_type = 4 → pseudo

Starting out, I would only be working with individuals and later I could move on to company and government customer types.

Building on Relations

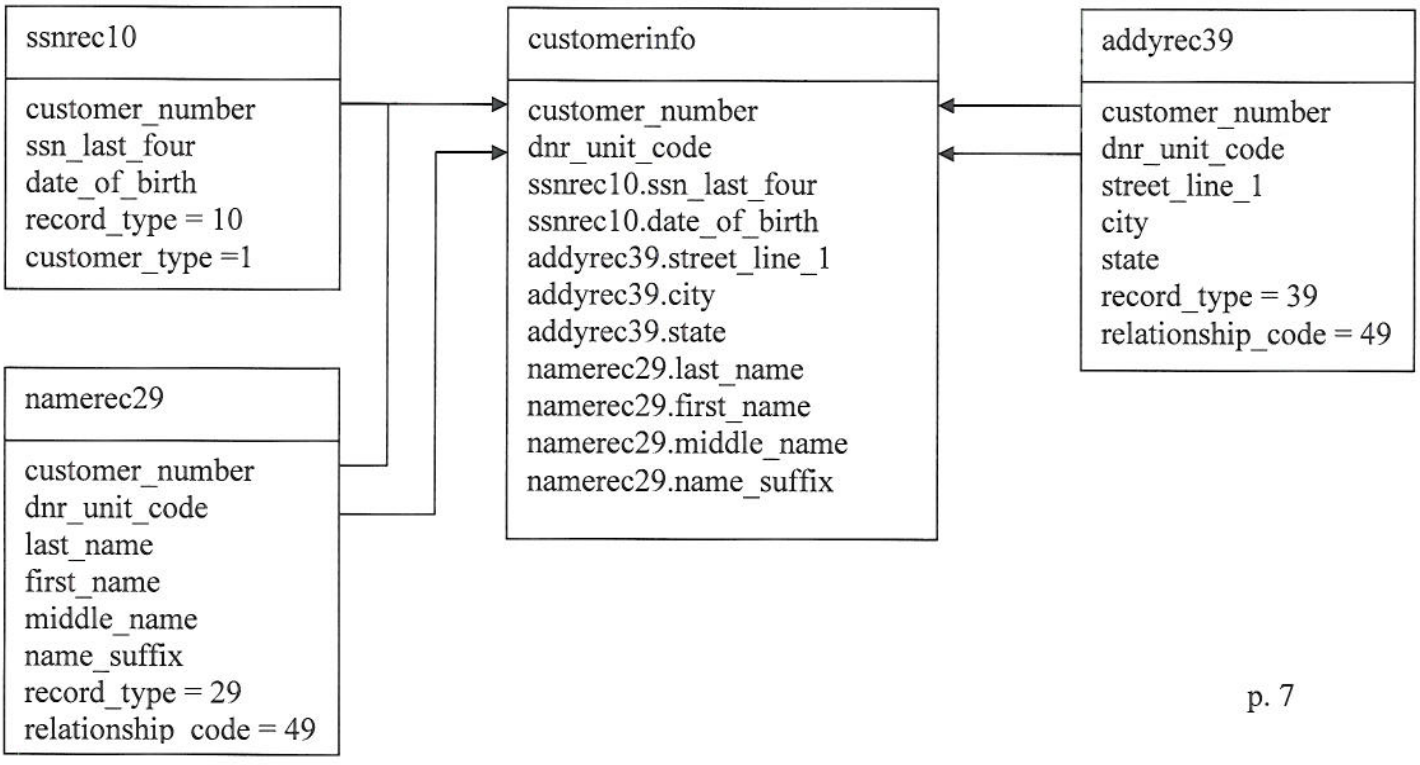
In the early weeks of developing a solution, I had a problem getting passed the design of the database. I would try and write single monolithic queries that attacked only one of the record types at a time by using nested "group by" expressions. These would usually end up in incorrect and inconsistent data. The fact was, I wasn't including all the information I should have been. I finally realized that I needed to reorganize the data into something I could understand better and work with. I needed to build my own pseudo relational database. I managed to build the following views:

ssnrec10
customer_number ssn_last_four date_of_birth record_type = 10 customer_type =1

addyrec39
customer_number dnr_unit_code street_line_1 city state record_type = 39 relationship_code = 49

namerec29
customer_number dnr_unit_code last_name first_name middle_name name_suffix record_type = 29 relationship_code = 49

This gave me a good starting point. I created a view for each record type that I would be dealing with and treated them as though they were their own tables. I then created a third view "customerinfo" that consisted of a three-way join between ssnrec10, addyrec39, and namerec29.



Now I had a "table" that I could work with that had all the information I needed for finding duplicates. You may note that namerec29 and addyrec39 besides being related through customer_number, are also related with dnr_unit_code. The reason for this is that even though each customer is only supposed to have one customer_number, different divisions within DNR can secure their own customer information by entering the data under their own unit code. This means that we would be excluding customer records that do not have an address associated to them. However, if we didn't use this constraint, we would end up with the Cartesian product of names and addresses spanning all units for a given customer_number. It would have been interesting to see what type of duplicates I could have hit with the different name/address combinations, but I wanted my results to be as accurate as possible.

The next step was to take two instances of customerinfo, and find matches where customer_number is not equal, but other fields are. My first instinct was to try and match on all fields, which only resulted 70 rows out of the 117,721. I tried different combinations of field matching resulting in varying amounts of candidate duplicates, but the result set would always come out too strict or too lenient. Obviously it was more important to get all the duplicates I could than to have a small set with high accuracy, but that would mean the secondary program would have to re-examine every match to see if it really was a duplicate.

I had a meeting with John and BJ once again, and I let them in on the situation. We all agreed that writing a program that would detect false positives as well as process the real duplicates would not be the best option. The only other way to process the results would be by hand. John would take care of finding someone to process the duplicates once I had solid results.

Increasing Results

Examining the records row by row, ordered in different fashions revealed some reasons why I wasn't harvesting as many duplicates as I had hoped. I jotted down problems that prevented my query from catching duplicates and came up with three main issues: missing data, abbreviations, and misspellings.

Missing Data

While perusing through the customerinfo view, I started to notice a common theme. There seemed to be many social security numbers (ssn_last_four) and date's of birth (date_of_birth) that were showing up as 0 or null. Writing a small sql query revealed that out of 94,485 base customer records, only 26,200 (27%) had a valid ssn_last_four, and 22,775 (24%) had a valid date_of_birth. Other fields that had a high number of null values were middle_name and name_suffix. The easiest way to fix this problem was to incorporate wild cards values for the null and 0 valued entries. Here's what the query looked like after I incorporated these changes:

```
select a.customer_number, a.last_name, a.first_name, a.middle_name, a.name_suffix,
a.date_of_birth, a.ssn_last_four, a.phone_number, a.street_line_1, a.city
from indcustomerinfo a, indcustomerinfo b
        where a.customer_number != b.customer_number
        and a.last_name = b.last_name
        and a.first_name = b.first_name
        and a.street_line_1 = b.street_line_1
        and (a.middle_name = b.middle_name)
            or (b.middle_name is null)
            or (a.middle_name is null))
        and ((a.date_of_birth = b.date_of_birth)
            or (a.date_of_birth = 0)
            or (b.date_of_birth = 0))
        and ((a.ssn_last_four = b.ssn_last_four)
            or (a.ssn_last_four = 0)
            or (b.ssn_last_four = 0)
            or (a.ssn_last_four is null)
            or (b.ssn_last_four is null))
        and ((a.name_suffix = b.name_suffix)
            or (a.name_suffix is null)
            or (b.name_suffix is null))
group by a.last_name, a.first_name, a.street_line_1, a.middle_name, a.name_suffix, a.city,
a.customer_number, a.ssn_last_four, a.date_of_birth, a.phone_number
```

Abbreviations

A big problem that I noticed in the street_line_1 field was that half of the addresses would use abbreviations and the other half would not. After spending some time contemplating how I could fix this, I figured the best way would be to abbreviate everything and create that as a standard. Going through the list wrote down the following commonly used address abbreviations:

SUITE → STE
PLACE → PL
BOULEVARD → BLVD
NORTH → N
SOUTH → S
EAST → E
WEST → W
HIGHWAY → HWY
CIRCLE → CIR
AVENUE → AVE
LOOP → LP
STREET → ST
DRIVE → DR
ROAD → RD
POBOX → BOX

Using this list I created a chain of "replace()" functions in addyrec39 (See Appendix A). I debated whether I should do the changes while comparing the two fields, leaving them unchanged in the final report, but standardizing them in the beginning was faster and gave the field a cleaner look.

Another abbreviation problem that needed taking care of was with the middle_name field. Most records only had a single character representing the middle initial, but some would have a single character with a period and a few would have the full name. The simplest solution was to only match on the first character.

```
and ((substr(a.middle_name,1,1) = substr(b.middle_name,1,1)
      or (b.middle_name is null)
      or (a.middle_name is null))
```

Misspellings

The most common situation between two records that should be marked as duplicates is a misspelling. To solve this problem I spent some time researching fuzzy matching techniques that are used in search engines and spell checkers. I came up with three algorithms that could help me match on slightly misspelled words.

- Levenshtein Distance (edit distance)
- Soundex

Levenshtein Distance

Levenshtein Distance is the measure of similarity between two strings. The distance is the number of deletions, insertions, or substitutions required to transform a source string to a target string. Here are a few examples:

Lev("kungfoo", "foo") = 4, since it takes 4 deletions to change "kungfoo" to "foo"

Lev("cungfoo","kungfoo") = 1, since only 1 substitution is needed.

The algorithm was developed by Vladimir Levenshtein in 1965. Levenshtein Distance is also commonly known as "Edit Distance."

While researching this algorithm, I came across an Oracle PL/SQL implementation created by Barbara Boehmer (See Appendix D). After looking through the algorithm and testing it multiple times, I was convinced it worked correctly and implemented it as a function in my schema.

The Levenshtein algorithm proved to be most beneficial in street_line_1. For some address records, the only difference between one and the other would be an extra word or group of numbers. Therefore, by taking the difference in length between the two

strings that were being tested, we would have a constraint that we could use in our expression.

```
and lev(a.street_line_1, b.street_line_1)<
abs(length(a.street_line_1) - length(b.street_line_1))
```

I also came to the conclusion that the bigger the two address strings are, the more misspellings are likely to occur. If a fraction of the length of the smallest string were added, it would account for these misspellings.

```
and lev(a.street_line_1, b.street_line_1)<
abs(length(a.street_line_1) - length(b.street_line_1))
+ least(length(a.street_line_1),length(b.street_line_1))/8
```

Soundex

The soundex algorithm takes a string consisting of only letters and returns a 4 character code that is a phonetic representation of that string. There are many variations of this algorithm, the first being patented in 1918 by Robert Russell and Margaret Odell.

Oracle 9i includes a version of soundex, which works in the following way:

1. Retain the first letter of the input string and remove all occurrences of the letters a,e,i,o,u,h,w,y

2. Assign numbers to the remaining letters (after the first) as follows:

   b, f, p, v = 1
   c, g, j, k, q, s, x, z = 2
   d, t = 3
   l = 4
   m, n = 5
   r = 6

3. If two or more letters with the same number were adjacent in the original string (before step 1), or adjacent except for any intervening h or w, then omit all but the first.

4. Return the first four bytes padded with 0.

Examples:

Anderson → A536
Andorsen → A536

Rubin → R150
Robert → R163

This version of Soundex is only useful for English words and names, however there exist other versions that are suitable for other languages.

Soundex was applied to the first_name and last_name fields.

```
and soundex(a.last_name) =soundex(b.last_name)

and soundex(a.first_name) = soundex(b.first_name)
```

After applying these changes, my result list jumped from a pitiful 70 to a hefty 2470 customer_numbers out of the 77,610 unique customer_numbers for individuals in customer. I wasn't sure if this was enough. I held another meeting with John who was more than happy with the results and quickly scheduled me to meet Kathy Dugan, a manager in the Public Information Center. The PIC would be in charge of processing my results once I had formatted them to their specifications.

Formatting

I visited Kathy many times and emailed her even more. It took a bit of correspondence to figure out what exactly the PIC needed to get the job done. The final list of requirements was the following:

- Display the associated numbers that are related to a given customer_number
- Mark associated numbers that are owned by given customer_number
- Mark whether a customer_number has been loaded into the Land Administration System (LAS).

Since an associated number in record_type 42 is just a reference to a case file, I will be using the two terms interchangeably.

Displaying the associated_numbers for a given customer_number was an easy enough task, however there is a many to many relationship between a customer_number and their case files since different departments can create their own customer data. That meant that when the records from record_type 42 are joined with the duplicate results, We would get a product of the set of associated_numbers and the set of customer data for a given customer_number. This problem was solved through sequencing.

I created a new view in the schema for the list of duplicates and named it dupelist. Within it, the list was partitioned by customer_number and ranked over the rest of the information. This would number the occurances of the same customer_number.

```
RANK() OVER (
 PARTITION BY a.customer_number
 ORDER BY a.last_name,
       a.first_name,
       a.middle_name,
       a.name_suffix,
       a.date_of_birth,
       a.ssn_last_four,
       a.phone_number,
       a.street_line_1,
       a.city) AS SeqNumber
```

See attached spreadsheet inddupelistsample.xls for sample results of dupelist after sequencing.

With this sequencing in place, I could then display the associated_numbers for only the first occurrence of a customer_number saving a lot of space in the final report.

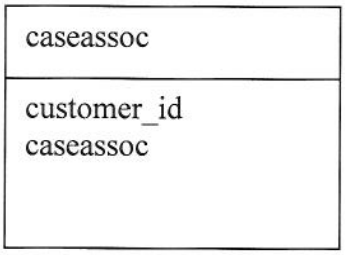
To bring in the associated_numbers I decided to create a new view called preformat.

preformat
customer_number
last_name
first_name
middle_name
name_suffix
date_of_birth
ssn_last_four
phone_number
street_line_1
city
seqnumber
seqnum
assnum
owner
inlas

Within preformat dupelist is joined with all the records in record_type 42, which is where all the associated_numbers in the customer table are located. They are only displayed where seqnumber = 1, which is the first occurance of a customer_number.

See attached spreadsheet indpreformatsample.xls for sample results of preformat.

The last two requirements given to me by the PIC required an entirely different table. If a customer owns a case_file, then the associated_number will also be loaded into the LAS system under the same customer_number. When an associated_number is loaded into LAS the customer_number and associated number is found in the case_file table. This table stores associated number slightly differently than customer does. While customer stores the whole number in the field associated_number, table case_file has it split into two fields file_type and file_number. I created another view to concatenate these two fields so it could be used in preformat.



I apologize for the poor naming. By this time, the deadline to submit the report to the PIC was coming soon, so some things were overlooked.

The field caseassoc would match up with associated_number in customer and customer_id would match with customer_number. By performing an outer join between the dupelist after the join with record_type 42 and the table caseassoc, I could check the caseassoc field for an entry. If it was null, then we know that the associated_number was never loaded into LAS. By using a decode() function, the caseassoc field was changed into "owner" which would display a '*' if the field held and associated_number , and a null if it didn't. Again, this value is only displayed if the seqnumber = 1.

The presence of seqnum in preformat may be unsettling to people since it could easily be confused with seqnumber that is mentioned above. The column seqnum is basically just another sequencing within the view that counts the associated_numbers for a given customer_number. The field "inlas" uses this seqnum and seqnumber to display whether that customer is present in case_file. It checks to see if there is a customer_id in case_file that matches customer_number. If there is, a "YES" is displayed and a "NO" if otherwise. This information is only displayed when seqnumber = 1 and seqnum = 1.

Once preformat was finished, it was time to switch from using sql to sql+. There are many commands that can be used in sql+ for generating reports that cannot be used in a normal sql scratchpad. In sql+ I could change the size of the columns so the output

could fit on an 8.5x11 piece of paper landscape oriented. After that I used seqnum and seqnumber so that the customer data is only displayed for the first row of the associated_number list, or rows where it is not the first occurance of a customer_number.

See Appendix A for the SQL+ query and document indsql+smallsample.doc for sample output.

Government Facilities and Companies

After finishing the duplicate report for individuals, I tried the same series of queries on companies and government facilities by setting the customer_type field to either 2 or 3. The result lists were small and inaccurate so I had to tweak the queries to work for these customer types. I made two copies of all the views I had created for individuals and adjusted them to work with companies and government facilities

The first thing that I noticed was that when a company or government record is entered, the name of the organization spans across all the name fields. This includes last_name, first_name, middle_name, and name_suffix. In both versions of namerec29, these fields were all concatenated into one for ease of use. Using Soundex() on such large names does not work very well since Soundex only creates a phonetic representation for the first few occurrences of a consonant. Instead of Soundex, the Levenshtein distance formula was used along with a chain of replace() functions that attempt to standardize the two fields being compared. The following are the standardizing rules used for the set of company queries and the set of government queries:

Companies
NATIONAL → NTL
ALASKA → AK
REGIONAL → RGNL
INSURANCE → INS
DEVELOPMENT → DEV
AND → &
COMPANY → CO
INCORPORATED → INC
CORPORATION → CORP

Government
ADFG → DFG
DFG → AK DEPARTMENT OF FISH &GAME
FINANCIAL → FIN
ADMINISTRATION → ADMIN
AND → &
DISTRICT → DIS
ALASKA → AK
DIVISION → DIV
SERVICE → SVC

For the street_line_1 field in both sets of queries, I reverted back to the original straight across matching because the duplicates need some way to be ordered by so the duplicates are found next to each other. With the individual queries, ordering was done by the Soundex of the last name because Soundex produces a value that can be ordered. Since the names of these record types are usually very large, more dupes were produced by implementing the fuzzy matching on names and ordering by address.

Conclusion

Here are the results of the duplicate finding queries:

Individual = 2,470 out of 77,610 unique customer numbers (3.2%)
Company = 566 out of 7,563 unique customer numbers (7.5%)
Government = 164 out of 759 unique customer numbers (22%)

Total = 3,200 out of 85,932 unique customer numbers

The remaining 8,553 customer numbers are located in customer_type = 4, which holds place markers for state use. I was assured that these were clean and secure and that I didn't need to bother with them.

The results shown above do not represent a number of actual duplicates, only candidate duplicates. The fuzzy matching algorithms were utilized in the design of the queries since an actual person is going to process these results by hand. This person will be able to discern if a set of candidates in the list are actually duplicates. The final report for individuals complete with associated numbers is 390 pages long with the same format as you see in document indsql+smallsample. There is still a debate of what to do with the Company and Government customer types. It has been suggested that the PIC might make an effort to standardize names themselves by hand and then run the queries against the data. Either way they are happy with the results I have shown them.

Overall, my internship at the Department of Natural Resources was a very rewarding experience. By the end of the semester, I really had a solid grasp on how data is stored and how different systems operate at DNR. I also learned many things about Oracle systems and SQL in general. The biggest lesson learned through this experience learning how to ask people for information or help on a problem. My coworkers were extremely friendly and helpful and always willing to lend a hand even if they were from a different unit. My supervisor was very supportive through the entire project and very enthusiastic about my progress. Within the short time I was able to work there, I gained a raise in pay and was offered a position over the summer. I accepted.