4 Methodology

In this chapter I will describe the process of converting the data from the *Paalgeld* portbooks to a machine readable encoded source transcript. The methods used to input this vast amount of data into the computer will be shown and I will discuss why other methods offered no practicable alternatives. I will explain the methods developed to accelerate the input-process while at the same time guaranteeing a greater reliability of the meta-source by applying self-learning methods for default generation.

Furthermore, I will explain the most common model for database design for highly structured data, the relational model. I will discuss the process of normalization and I shall explain some aspects of historical data, that make them ill fitted for this model. Next I will explain why I preferred to use a variant of the relational model, which allows data-storage in a not completely normalized form while keeping the tabular structure of the source.

Finally the technicalities of the record-linkage process will be discussed. The results of this record-linkage have been used in chapter three, to establish the meaning of the date mentioned in the *Paalgeld* portbooks.

4.2 The Meta source

The *Paalgeld* portbooks contain a treasure of data for the economic historian. In their original form however, they are of little use for further analysis. It would take a herculean effort to come to the simplest tabulations of the data by hand. From the very beginning of this project it was obvious that this task could only be tackled with the help of a computer. In a sense this only shifts the problem to another stage: the data have to be stored in a machine readable form before they can be analyzed. To get them stored, they have to be entered first and here comes the new herculean task: how to get the data of about 2800 ships each year, having an average of 2 cargo items each into the computer.

There are two ways to confront the problem of entering historical data. If we exclude iconographic and oral material, historical sources come to us in two forms : printed and hand-written. For printed sources there are two methods to convert them into machine-readable form: you can either enter the data by hand or you can scan the originals into the computer and have the image translated to text by "**O**ptical **C**haracter **R**ecognition" (OCR) software. Using the first method, the qualifications of the personnel for the job need not be very high. Still there would be the problem of possible misreads to deal with, which would result in an

incorrect digital copy of the original source, if left uncorrected. Therefore, an expert is needed to do the final corrections, which is a very time-consuming and boring job.

In spite of a rather naive optimism about the future developments OCR-software has not yet reached the standards that one would like at this stage in time.[1] Experience with scanning all sorts of historical data has taught that the task of "training the program" to read correctly is extremely time consuming. After this training-stage the speed improves, but the reliability of the results is not quite satisfactory. First of all the reliability will never be much more than around 98%, which still means that 2 out of every 100 characters will not be recognized by the program. Missing characters will be flagged by the program, so that the corrector can find them easily. A greater problem are misreads by the computer: these will not be flagged and will be very hard to find, if a spelling-checker cannot find them. This implies that the actual scanning can be done by secretarial staff but every line of the files produced by the OCR-programs must be thoroughly corrected by an expert. Van Horik has calculated that for printed matter OCR will reach a break-even point with keying in the data after about 1400 pages, assuming inspection and adjustment of the setting of the software and hardware after every 20 pages. This implies that it is not yet an economically sound alternative.[2] Mark Olsen and Alice Music McLean have shown that it is usually a political decision within an institute to use OCR, influenced by an unrealistic measuring of personnel costs. Compared to scanning with "state of the art" hardware and software, in most cases the keyboarding of texts in low-wage countries is of a higher quality and much cheaper.[3] Mark Olsen will only make an exception for printed numerical data, because much of the data verification can be done algortihmicly.[4]

If the "state of the art" OCR-programs provide no satisfactory solution for printed matter, scanning and recognizing hand-written sources, which include the bulk of historical sources, is even more problematic. Handwriting recognition is divided in on-line and off-line recognition. On-line recognition concerns the recognition of handwriting at the moment it is written, as in the case of pen-interfaces, off-line recognition deals with the recognition of script written on paper. In both cases there is a marked difference between the recognition of isolated characters or of cursive script, where the latter is much more complicated, because of its diversity. There is considerable interest from commercial parties in this problem. The main

---

[1] O. Boonstra a.o. (1990)

[2] R. van Horik (1993)

[3] M. Olsen and A.M. McLean (1993)

[4] M. Olsen (1993)

attention is for automatic processing of handwritten mail, checks, forms, and faxes. Considerable progress has been made--in some cases leading to practical application of recently developed techniques, but most of the research is still in a laboratory-phase. Some modern and indeed most medieval handwriting can be recognized with a 60-80% accuracy by some experimental OCR-programs. In cases of very restricted vocabularies interesting results have been gained with software based on neural-networks. [5] The rather stunning results in recognition of handwriting that are claimed for the software package Optiram cannot be tested since the software is not commercially available, nor have the algorithms been published on which it is based.[6] But these approaches still need the use of time consuming image-enhancement techniques, while the results still leave a lot to be desired. The human being on the other hand is able to recognize handwritten text with an almost complete precision if the context is known.[7]

If we focus on 17th and 18th century material, the results with OCR are so poor and would require so much pre-processing and post-editing, that it is no alternative to human reading and keyboarding. Handwritten sources from this period are actually quite difficult to read: it takes trained historians or archivists to read to these sources and even then differences of interpretation remain. It would be very difficult to formalize the rules for recognition of single characters or words, a prerequisite for successful automatic recognition. Hence, the only viable solution to reach my first objective of making a machine readable form of the complete portbooks was human reading and keyboarding of the text. Since a complete character-by-character transcription of the source would require too much time, methods had to be developed to accelerate the data collection. These will be discussed in the next paragraphs. But even after the development of techniques to speed up the input-process I had to reconsider my original goal to make an integral machine readable transcript of the portbooks, which would make it possible to compare the data for the "West-Indian" commerce with the other commerce on a year to year basis. After processing the data for the complete portbooks for the years 1742, 1771 - 1787 I realized that this task was not within the reach of a single person with a limited amount of time and limited funds. The data for the 18 years that have been entered cover 54537 ships coming to Amsterdam,[8] with descriptions of 104090 cargo items.

---

[5] A. Senior (1993), 59-65

[6] R. van Horik (1992)

[7] E.L. Helsper, L.R. Schomaker and H. Teulings (1993)

[8] 2467 ships in the West-Indian trade and 52060 in the rest.

This project provided valuable information about the portbooks themselves, but more important, it forced me to think of ways to cut this herculean task down to human proportions.

An obvious solution would have been to study a sample of the total population. For a number of reasons sampling techniques have not been used. First of all, the results of an earlier pilot-project carried out in 1982 with a group of students of the University of Amsterdam by Heeres indicated that the structure of the data can produce serious sampling-errors. The character of the data is so irregular--probably caused by the weather conditions, that several small but important contacts of Amsterdam did not appear in the test sample. Furthermore, the custom of sailing in convoys creates clusters in the data-set which can easily be missed in a random sample. In addition, a sample would not permit the same amount of detail as an integral approach. Secondly, to create a representative sample a good insight in the whole of the population is needed. This study being the first on this source, no such knowledge was available. Thirdly, even if a representative sample could have been obtained, searching for the randomly selected cases in the handwritten source would be almost as time consuming as entering all the data. Finally and most importantly, the complete transcription will be of much more value to other researchers than the sample. They will be able to use the data in ways which have not been envisaged before.

Thinking about the input-process of historical structured data became one of the goals of this study in itself. The historical analysis of the trans-Atlantic trade of Amsterdam, which will be presented in chapter 6, relies only on the 5543 cases that are mentioned in the second part of the portbooks concerning the Atlantic trade covering the period from 1771 to 1817. But important ideas about the input process were developed during the data capture from the parts of the portbooks that deal with the European trade. The analysis of these books will be presented in chapter 5 and will provide a frame of reference for chapter 6. Since this study has a dual focus on the content of history as well as on the methods for computer-aided historical research, the next paragraphs, mainly dealing with the solution of problems that emerged in the data capture for this study, are essential.[9]

4.4 Considerations in making machine-readable forms of historical data

In some traditions in the historical disciplines it is assumed to be not *comme il faut* to discuss methodology extensively. Especially in the Anglo-Saxon historical tradition the emphasis is on *story-telling* and not on the nitty-gritty details of how the research was done. A

[9] Some parts of the following paragraphs have been published before in : G.M. Welling (92,93-1, 93-2)

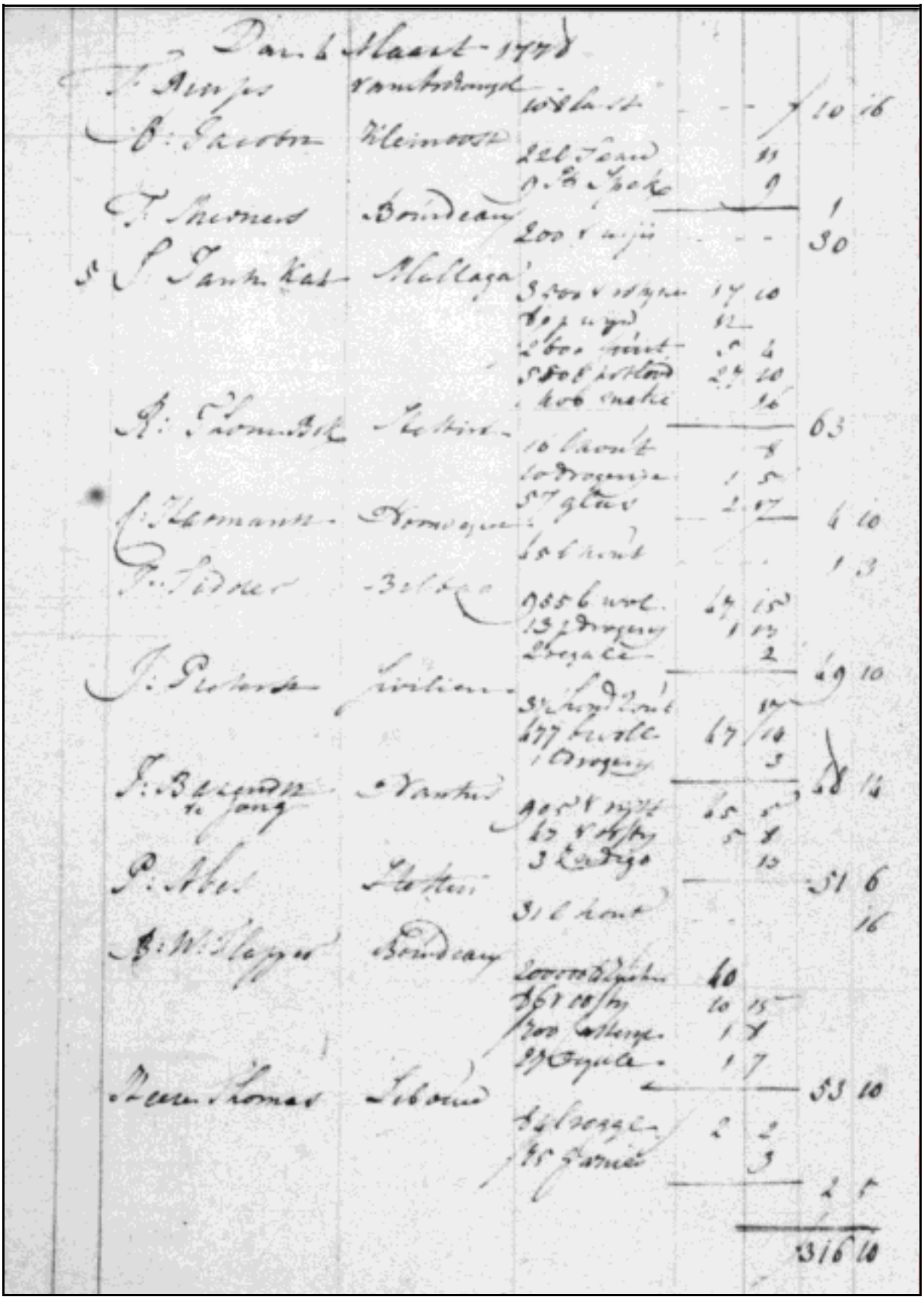

Figure 1 Sample page from the *Paalgeld* portbooks, March 4, 1778

recent study[10] takes this approach to the limit by not even providing accurate bibliographical references. The replicability of the analysis, which is a *conditio sine qua non* for all sciences, is hindered by the often implicit steps of reasoning in the narrative. Many historians regard their trade more as an art than as a science. In the 1960's Cliometricians tried to introduce econometric methodology to the historical discipline. Their influence on mainstream history has been overwhelmingly negative: many historians decided that if that was the new form of history, they would rather have nothing to do with it. When Fogel and Engerman decided to publish a special volume about the evidence and methods used in their research for "Time on the Cross", many historians considered it as an absolute low point of historical practice.[11]

Thus it is not surprising that the introduction of the computer to historical research has made very little impact. First of all, the historians who were in the vanguard of computer users came from a group of historians that were already on the periphery of historical research themselves: mainly doing quantitative research and only interested in the number-crunching capacities of the computer. In an almost endless stream of *"the historian and the computer"* publications they advocated adjusting the practice of history to ways which were thought to facilitate computer analysis. Their limited view of possible computer application to history led them to advocate reducing historical research to the form of statistics applied to historical problems. The methods used in social science were suggested as the only correct way for historical research. First you define your question, then you search the data that can be analyzed to come to the answer of the question, and finally you test you hypothesis.

This *problem-oriented* approach to history required an adjustment of the historian, who was used to a more or less intuitive approach, using a sort of *snow ball method*, where the reading of one book led to the study of other literature or sources mentioned in that book, and so on. As long as you start with the right books this approach can be very productive, but it has also given history a bit of the reputation of a discipline where books are written only about other books. Even if quantitative methods have made little impact, the problem-driven approach is emulated frequently. Posing the question, selecting the evidence, analyzing and presenting the results is a framework which can be recognized in almost every historical study, even if sometimes one might get the impression that many studies have been forced into this straightjacket afterwards.

The introduction of the computer to historical research has also had an effect in the opposite direction to a more *source oriented* approach that I have mentioned before. The

[10] J. de Vries and A. van der Woude (1995)

[11] R.W. Fogel and S. Engerman (1974)

storage facilities of the computer and its capability of processing vast amounts of data provide the possibility to store sources in their entirety in a digital form and then analyze the data until some form of picture emerged in the historian's mind. From that phase onwards there is little difference with the *problem-oriented* approach: once a picture has formed or a problem is formulated, the process of selecting other evidence and analyzing that to test the validity of the question is not much different from traditional research. This source-oriented approach has quite often been mistaken for a neo-positivist reverence for the sources as "historical facts", but since E.H.Carr's *What is history*[12] every historian should know that historical facts are created by the interpretation of history by historians.

In this study the *Paalgeld* portbooks have not only been the main source, but also the inspiration for further research. The temptation to get an almost perfect grip on this source by computer manipulation has been irresistible. This has resulted in a apparently unexpected outcome: the focus of this study has been narrowed and not broadened after the completion of part of the input process. The resources were insufficient to complete the project as originally planned, and during the process the insight was gained that a closer investigation would lead to too many questions, at least more than can be answered within a lifetime. Harvey's circular model of historical research adapted the form of an inward spiral: every round led to a narrowing of the question in order to restrict the problem to a size of human scale.

4.4.2 Data modeling

Information science suggests that one make a model of the data before one starts entering them. However, this requires a full overview of all the peculiarities of the data, which the historian does not have before he starts to input the data from a source. If all is well he has had a good look at the source, but will probably not have seen all the irregularities that surface during the very close examining that takes place during the actual input-process. Data modeling is an activity which can be successfully undertaken only if all the irregularities of a source can be overseen.

In computer science a number of data models are known, but the relational model in a variety of forms has become the most popular for use in large data sets, especially for those that are too large to fit into the computer's main memory at the same time.[13] It is based on decomposition of the data into tables where the rows or *tuples* are the records and the columns

[12] E.H. Carr (1961)

[13] For a formal explanation of the relational model see: J. G. Brookshear (1991), 332-341. See also: C.J. Date (1983)

contain the attributes. To avoid redundancy, which might lead to problems in maintaining data-integrity, tables are split up until each table consists of a key-attribute, uniquely defining each tuple, and a number of attributes depending on this key. Tables can be linked to each other if they key-field of one also exists in another. This seems rather simple, but a complete implementation of this model can lead to a very complicated data-model, in which every table on its own gives very little information and handling multiple tables can be tricky.

One of the prerequisites of the relational-model is that a key-attribute can never be null or missing. For the historian who has to deal with the problem of missing data, or conflicting data for the same attribute the rigidity of the relational model can be a serious drawback. Manfred Thaller has argued that the historical sciences need a special model, which can deal with the complexities of historical data. He developed a database management system for the public domain, called Kleio (Κλειω), which should meet these needs.[14] Kleio has sophisticated facilities to deal with most of the problems of historical data management that commercial database management software do not tackle, but it has some serious drawbacks. Thaller claims that the substandard quality of the user-interface is acceptable because the system allows the trained user to do things that would not be possible with other systems. However, the learning curve for this system is so steep that only very few research groups in Europe do use this system. A further problem with the system is its black-box character. Thaller has not produced a formal description of the data-model on which Kleio is based, which makes it impossible to discuss whether this system is a theoretically correct solution of the problems.[15]

Discussing data models for historical sources, Schijvenaars discerns two types of data-models: source-oriented, and topic-oriented methods.[16] He subdivides the source-oriented models into document-oriented methods, which try to conserve the structure of the original document on which the database is based as closely as possible, and data-oriented models that try to record all the data of the source, but do not try to mimic the original structure in the digital transcript. Topic-oriented methods only capture from a source those pieces of information that are needed for the predefined research purposes and may even be coded at the input stage. His critique of the quality of these models is based on hindsight: once a source has been entered it is possible to create a model which will permit a minimum of redundancy.

---

[14] M. Thaller (1989)

[15] See also: J. E. Everett (1995)

[16] T. Schijvenaars (1994), 511-512

At first glance the *Paalgeld* portbooks seem to have a structure that would fit nicely in the tabular structure of the relational data model. But historians have never been very happy with the "flatness" of the relational model, which was not designed to deal with the complexities of historical data. The historian has no control over the data that he has to work with; he has to make do with the data that are left to us by our ancestors. The point of departure of the relational model is that you first build your model and then start collecting your data, while the historian usually has to deal with the opposite situation.

Following the rules laid down by Codd[17] I have made a complete relational model for the data in the *Paalgeld portbooks*, which I will discuss now. Before we can make a model for our project, we have to understand what information the *Paalgeld* portbooks consists of. It may seem that this has been discussed in chapter 3, but now I follow a different approach: I focus now not on the quality of the data, but on its the structure. The portbooks have an almost tabular form. All the pages on which the ships and their cargo items are mentioned have a structure of a header, in which a date is mentioned, and a sequence of rows and columns. The columns have no descriptor, but from the data in the column it is simple to define one. The structure of the part on the European trade differs from that of the part on the West Indian trade. We will describe the structure of the part on the European trade first, because it is more complicated.

The first column has a date indicator, the second a name of a person, the third a geographical name. The fourth column can have more rows for one ship: this is the column where the cargo description is given, and one ship can carry a variety of goods. For each type of goods a quantity in a given measure is given in this column. The fifth and sixth column are used to write down the amount of money to be paid for each part of the cargo: in the fifth column we find the amount of guilders and in the sixth the amount of stuivers. Columns seven and eight are used for a summation of the amounts to be paid for all the goods on one ship. If a ship only carried one good, the amount to be paid was put directly in these columns. At the bottom of each page we find a summation of the all the entries of that page.

The West-Indian part has a more simple structure. The pages have the same sort of layout: they have a header and are also divided in columns. But since the description of the cargo is not given, every entry just takes up one row. For most years the date is not given for every entry, but there is one additional part of information, the name of the ship. The name of the ship can be found in the column before the name of the ship master.

To fit these data into a relational model we must understand the difference between the rows and columns of the portbook on the one hand (see figure 1), and the rows and columns of

[17] E.F. Codd (1972)

the relational model on the other. We will use the entry for B.W.Flapper--the penultimate case of the scanned page in figure 1. B.W. Flapper is registered as coming from Bourdeaux on the fifth of March with four cargo items, 200.000 lb Zuiker, 86 v. Coffy, ƒ700 Carstengen, and ƒ75 Garnier, for which he paid resp. 40 guilders, 10 guilders and 15 stuivers, 1 guilder and 8 stuivers, and one guilder and 7 stuivers, totaling 53 guilders and 10 stuivers, as an example. Although the date-column is in fact empty, I assume that the same date is meant as for the entries above this one. What will be stored in our database is not the data concerning the amount of money a certain ship master paid on a given day, but the data from an entry into the portbooks. Each row or tuple in the database will refer to one single registration of the amount of money paid for the cargo of a ship. Each entry must be distinguished from the others: it must have a unique identifier. Since the name of the ship is not given in this case the combination of date of registration, name and origin of the ship master would be used to identify this entry in the register. Furthermore all columns in the relational model may only contain "atomic information", which means information that has no further internal structure and hence cannot be split up any further. Here the name-column of the portbooks present a problem. The name *B.W.Flapper* still has a structure of family name, and initials. In the model a separate column must be created for the family name and the initials or first names. One could even argue that the *B* refers to a first name and the *W* to a patronym, which would mean that both initials have a different meaning and should be stored in separate columns,[18] which would mean that the name column from the portbooks should be split into three columns in the relational model. Another prerequisite of the relational model is that all columns in a table should be either part of the identifier or attributes of that identifier only. The name of the ship master, the date and the place of origin are all part of the identifier, but the description of the cargo items presents another problem. Here is an example of a repeating structure: a number of cargo items is described and for each information is given. There is a one-to-many relation between the ship and the cargo items. In the relational model these relations are stored in separate tables, which should be linked by the identifier of the parent record, in this case the combination of date, name and origin. Since this would require repeating these data many times a trick is used: we substitute the complex identifier by a unique symbolic identifier. In this case I gave it the number 17780095, indicating that it was the 95th entry in the portbook of 1778. Now I can identify the cargo items of a given ship by its number. The reason to store these repeating structures in a separate table is rather simple: if the data were stored in one table and the structure of that table had to be predefined, then that structure would have to be able to store the most extreme cases. A ship might carry some twenty different cargo items, and the

[18] In the database I have made for these data there is only one column for first names or initials.

structure would have to be prepared for such cases. This implies that I will make very inefficient use of computer storage, since the number of bytes used for every record will be the same. Ships carrying only one cargo will require just the same amount of storage-space as the more extreme cases.

Column 4 of the portbook contains the measure, the quantity and the name of a cargo item: these must be put into separate columns since columns should only contain "atomic information". The columns where the amounts of guilders and stuivers paid for each cargo item are given present no real problem: they contain atomic information and can be stored in columns of their own in the model. However, one could argue that there is no need to store this information, since I could always reconstruct it if I had the tariff of the levy for all goods. In the *Observantie van den Ontfang van 't Paal-Gelt* I find a levy of 4 stuivers for 1000 Lb of sugar, so the levy for 200.000 lb should be 40 guilders. If I stored the tariff, then I would not have to store the actual amount paid, since it could be computed. The amount to be paid is *process data*, which normally are not stored, but computed when needed. Dealing with historical data however, it is quite normal to store process data for a number of reasons. First of all I want to store the data as they are in the source: if the clerk who took down these data made a mistake in his calculations, what should I do? Put his mistake in our database, or let the computer make the correct calculation? Since I want the data-transcript to be as close as possible to the original source, I must store these process data in the data-base. The second reason is that quite often the measure used in the portbooks is not found in the *Observantie*. Shipper Flapper has 86 v *coffy* on board, i.e. 86 vats of *coffy*: however, the only measure mentioned for *coffy* in the *Observantie* is the bale. Storing only the amount and the measure would create serious calculation-problems: the ratios of all measures used to each other should be known, which they are not. The last two columns of the portbooks offer the same problem: for ships with more than one cargo item we find here a summation of the levy on all cargo items, which by nature are process data. After checking the register thoroughly I

entity: portbook_entry

portbook_entry:	ship	=	n : 1	
portbook_entry:	date	=	n : 1	*
portbook_entry:	shipname	=	n : 1	**
portbook_entry:	shipmaster	=	n : 1	
portbook_entry:	harborname	=	n : 1	
portbook_entry:	cargo-item	=	n : m	*

entity: cargo_description *

cargo_item	:	name	=	n : 1
cargo_item	:	quantity	=	n : 1
cargo_item	:	measure	=	n : 1
cargo_item	:	levy	=	n : 1

* not in West-Indian trade
** not in European trade

Figure 2 Relations in the *Paalgeld* portbooks

have decided not to store the summations, since I have not found a single mistake in them for the data of 1778, the year of the pilot-project. This is a choice, which may have been wrong.

Now we have a model to store the data of these entries in the portbooks. I have distinguished two entities: entry of ship arrival on the one hand and entry of cargo items of that ship on the other. There is a one-to-many relation between the ship table and the cargo table, meaning that every ship can have zero or more records in the cargo table. This implies that two tables are needed to describe these two entities: one for the data that are attributes of portbook-entry concerning a ship and one for the data that are attributes of the cargo. Now I can establish the relations between the data entities. As the central entity I view the entry into the portbooks. It is immediately clear that there must be a different model of associations for the first part of the portbooks and the second part, because of the essential difference in information. The associations presented in figure 2 should fit both parts however.

This model of the associations between the data is rather straightforward. The process of breaking down the data into relations in which every relation contains occurrences of one entity and its attributes is called *normalization,* which is a prerequisite for storing data in a relational database. The great advantages of this model are its simplicity and its efficient use of computer storage facilities. There is a set of prescribed steps to do a normalization of a data-set, but as has been shown above the same result can be reached by simply applying the basic rules of the relational model.

However, a complete normalization of the data can create more problems than it solves for the historian. One clear example is the n:1 (many-to-one) relation of the portbook_entry to ship master. Many entries in the portbooks may refer to the same ship master, indicating that one ship master can have made many voyages to Amsterdam. But historical data are not as simple as that: how can I know in the input phase if one mention of a captain with a given name refers to the same captain with that name mentioned before? To consider them to be the same person is a historical interpretation which should be postponed until after the input-process is finished to avoid mistakes that cannot be undone. The relational model, however, requires cutting down redundancy to an absolute minimum to avoid possible forms of inconsistency. This would imply that two apparently identical entities should only be stored once. But at the time of input the historian has no real way of knowing if two apparently identical forms of an entity do indeed refer to the same object. After completing the whole input phase he might be able to make this decision. This would imply that a complete normalization could only take place after finishing the input process.[19]

[19] One could try solve the problem by labeling such fields "spelling of ....", which would permit seemingly redundant forms. The Relational Model has no built-in facilities to deal with "nearness of spelling".

The great benefit of the relational model is the checks is allows against inconsistency and the ease of data-manipulation. For each column of a table conditions can be defined: input should be of a certain data-type, of which ranges can be defined. Templates can be used to define the required form of the input. Validations can be programmed to check if the input conforms to predefined conditions. By describing the domain of all possible values for a column, faulty input can be recognized and rejected. Because the number of bytes each record takes is known, the database management system can calculate the beginning byte of each record and storage and retrieval are extremely fast.

A serious drawback is the complexity of data-manipulation of the relational model. Single tables only rarely offer all the information needed. The user must know in which table what information is placed to formulate correct queries, which means that the user must have the picture of the model of his database in his head. In some cases this can get very complicated. Even using a query language like SQL, which comes close to natural language, formulating correct queries for a complicated database can be cumbersome. Most modern database management systems try to solve this problem by offering a possibility *to query by example*: the user fills in required values in the columns of an image of the database structure, which the database management system translates to SQL. This approach solves the complexity of querying, but cannot solve the other problem of relational systems: since in a complicated database a great number of tables will be used at the same time the amount of disk-access is enormous. This makes the performance of relational database management systems dependent on the performance of the disk-operating system.

portbook_entry :	idno, folio, date of registration, second date, total levy, remarks
LinkTable Ship :	Shipno, idno
ShipInfo :	Shipno, shipname, size, crew
Shipmaster :	Familyname, firstname, idno
Harborcode :	Harborcode, idno
Harbors :	Harborname, Harborcode, soundtollcode
Cargo :	Idno, gmwcode, quantity, measure, levy
Produce :	Produce, gmwcode, soundtollcode
Levy :	Gmwcode, standardmeasure, amount

Figure 3 Normalized form for the *Paalgeld* portbook data

In figure 3 the complete normalization to a relational model is presented, in which the use of code-lists is also incorporated, a technique which allows us to use one code for multiple forms of the same name. Since in this model all forms of redundancy have been eliminated, it could only be implemented after finishing the input. However in this case the cure is worse

that the disease. Converting the data from a unnormalized form into this model, the historian will have to make far-reaching decisions, which cannot be easily corrected. Another and maybe even more important problem is that single tables of this model have little or no meaning on their own: only a view incorporating the data from several tables will reproduce the information that is gathered from a single entry in the portbooks. This requires that the historian should have more than average knowledge of database handling techniques.

Aberson has shown that it is of major importance that the historian has a very close relationship with his data.[20] Dependency on external experts to manipulate his own data will only make the historian feel like a guest in his own house. The experts will probably confront him with graphical representations of his data set, which will make the historian wonder if there should have been an electro-technical component in his training in order to understand these graphs. What may even be worse is that he will just accept what the experts offer him, without really understanding what is done. If the analysis a historian advances is partly based on database-handling, he will have to understand that, because he and only he is responsible for his conclusions. Lack of understanding might lead to serious misinterpretations. The opportunity to approach the data set from whichever angle the researcher desires is essential: this manipulability of the data provides the stimulus to develop ideas about the data set and to discern the underlying historical structures.

Finally, the original problem - a data model which can be used in the input phase - is still not solved. Of course there is always the minimum solution: entering the data in a flat-file format. The flat-file model for the *Paalgeld* portbooks would store all data for one entry in one record. This would require a record-structure which would allow one to enter the most complicated case. Many historians would be prone adopt this solution, which can be concluded from the astonishing popularity of data management software like Reflex, which is essentially a flat-file database management system.[21]Flat-file databases do have a number of serious problems: they have no facilities to deal with repeating structures within a record. In the *Paalgeld* portbooks we find this problem in the association between ship and the various parts of the cargo. First of all to set up the database I would have to consider a "worst case" scenario: how many parts can a cargo have at maximum? In the record-structure fields would have to be created to allow the input of this case as well as that of a ship with only one lading. Since I cannot know this worst case before the start of the input, the database would have to be restructured frequently to allow for the latest worst case. To save disk space flat-file systems

[20] D. Aberson (1994)

[21] For the popularity of Reflex (a Borland product) see the first three volumes of History and Computing (1989-1991)

usually store the data by putting field-separators between fields, a sequence of two separators indicating an empty field. This way of storing the data will only allow sequential access to the data, which means that to retrieve a record 100 from the database, the database management system will have to start from the first record and look for the 99th record separator, to find the begin of the record and for the 100th record separator to find the end of the record. Then it would have to look inside the record for the field separators to present all the data of that record. This may not take so much time in small databases, but the performance will go down steeply in larger databases. Related to this is the problem of updating flat-file databases: if an update changes the length of a record with only one byte, the whole file will have to be reorganized, which can be a time-consuming affair in the case of large databases.

A major problem of flat-file systems is consistency checking: in the relational model it is very simple to define conditions for each column of a table and to design checks to see that only data fitting these conditions can be entered in the input phase. The column with the amounts of stuivers in the records for the cargo of an entry of the *Paalgeld* portbooks should have the condition that no number below zero or above 19 can entered. Since the data for the cargo are stored in a separate table in the relational model, this condition only needs to be defined once. In a flat-file system this condition would have to be defined for the maximum number of cargo items, which would create great opportunities for inconsistencies. Manipulating the data in a flat file is not impossible, but it would be rather difficult to find ways to do statistical calculations on all cargo items of all ships. In the relational model a program would consequently pick the same field from the table with the cargo items. In a flat-file system a program would have to be written to go through all the fields, where cargo-specifications may be found, to count the number of cargo items in that record, and to do calculations on these fields. The amount of extra programming needed is an open invitation to errors.

Storing the data in a spread-sheet program may seem very attractive alternative, because of the built-in statistical facilities of these programs. However spreadsheet programs have no adequate solutions for storing one-to-many relations for projects of this size, although modern three dimensional spreadsheets can deal with this problem on a smaller scale. Essentially spreadsheets are flat-file systems and are not suited for projects of this scale.

A final and very drastic step would be to enter all data simply as marked-up text. This would of course generate the same problems as using a flat-file system would. In spite of the development of very flexible mark-up systems like SGML, this cannot be considered to be a real solution. In order to get any information from the data set it would have to be tagged, which would require a very efficient SGML-editor to keep the required input labor to an acceptable minimum. But the definition of all tags requires a prior understanding of the text to

be tagged, which leaves us with the same problem: to set up the system properly complete knowledge of the source is a prerequisite, which is exactly what is missing at the start of a project. Next to that, one would dearly miss the range checking and validation facilities that a normal database management system offer. Even if it would be possible to find a way to store the informational structure in a mark-up system, it would still have the same drawbacks for manipulation as a flat-file approach, of which it is a sub-form.

If there is no real solution for the problem of the complexity of database handling, can it be reduced to a minimum? The approach chosen for this project is the development of a data-model which closely resembles the method of LOGIC.[22] LOGIC is not an alternative model for databases, it is a variation on the relational model that tolerates more redundancy than the relational model in order to avoid the splitting up of relations into tables that have no more meaning on their own. In simple cases LOGIC-databases will look just like relational databases. In complex cases there will be more difference. Each LOGIC-table will provide complete information on an entity. Some of this information may be redundantly stored in other tables as well. Keeping the redundant information consistent is crucial to the approach. This method requires you to formulate normalized natural language sentences about your data set. A normalized sentence can only have one verb and each sentence should be independent, meaning that it makes sense on its own. All complete normalized sentences translate into tables. These sentences take the form of *NOUN verb [NOUN]……[preposition NOUN] etc.*. This forces us to create more than one sentence for repeating groups. In this case the repeating group are the parts of the cargo with their measure, quantity and levy.

This rather simple method resembles the relational model in its tabular structure, but does not have the absolute ban on redundancy. Theoretically this must be regarded as a setback, because the possibilities for logical inconsistencies cannot be avoided. On a practical level the model has great advantages for historical research: each table contains complete information and allows the researcher to experiment with single tables, which have a meaning on their own. This improves the historian's grip on the data and at the same time makes him less dependent on external experts for designing a data model. In sharp contrast to most other methods this method does not require one to understand a complicated set of graphical symbols: it expresses itself in natural language. In 1989 Professor Nijssen stated that it was about time to give control over database design back to the end-users, who are the real experts on the content of the database. He also pleaded for the use of natural language as a tool for

[22] LOGIC Language Oriented Graphical Interface computing, see: B. Young (1994)

database design, instead of the complicated graphical languages that have been developed to represent data models.[23] The simplicity of LOGIC seems to be an answer to his question.

The data-model of the first part of the portbooks is expressed in two normalized sentences:

1. *a ship with a captain arrives at a date from a port with cargo*
2. *for each cargo-item of that ship in a quantity with a measure a levy is paid*

This model is then translated to two tables: one for the ship-arrival and one for the cargo items of that ship. These tables are linked by the identification number that is given to the ship's arrival. The result of this model apparently involves redundant storing of data, but as long as there is no absolute certainty that two identical names for a captain refer really to the same person, there is no real redundancy.[24] After the interpretation of the data one might decide to attempt a full normalization, but in the input phase this is not desired in historical research, so this method is very well suited for the input-phase of a project and might even be used throughout the whole project.

The data-model of the second part of the portbooks is even more simple:

1. *A ship with a captain from a harbor pays an amount of Paalgeld (at a certain date)*

The result looks very much like a flat table.

For input-purposes the LOGIC-model is ideal because all interpretations can be postponed to a later stage. On the other hand it allows for simple restructuring, which may be necessary during the input-process if new information pops up, which must also be recorded. Most cases only require adding another noun to the normalized sentence, which implies adding a column to that table. Early interpretation of the data would create the problem of the n:m relation between ships and captains: this model avoids these irreversible interpretations that are required by the relational model.

A comparison of the LOGIC model of this data set with the model depicted in figure 2 can explain my preference for this simplicity: it can do without entity-relation diagrams or bubble charts, or any other form of diagrams, which quite often obscure more than they explain for untrained readers, and it gives the historian full control over his data. So many authors have stressed the necessity that historians should adapt their research methods to the computer, but if the computer is to be accepted as a serious research tool it will have to be adapted to the historian.

---

[23] R. Keijzer (1989)

[24] In a fully normalized relational model at least one extra table would be needed for the ship master, since the family name and initials or first names only refer to him, they should be placed in a separate table.

4.4.4 The input-process

As mentioned before the great bottle-neck in projects of this scale is the input-process. Surprisingly, the bulk of attention of computer science in respect to database handling is focused on data models, querying, and data-management: there's very little attention to the input-process and some of the better handbooks do not even mention it as a problem.[25] The input-process is regarded as a tedious job, but not as really problematic. Range checking and validations are supposed to suffice to guarantee the quality of the data. I suggest that the input-process deserves more attention since it is the very foundation of all further analysis: if the input is wrong, who would care for the analysis?

First of all the decision should be made what should be included. In the introduction I have explained why I decided not to use sampling techniques and to refrain from coding the data in the input-phase, but to make a complete digital transcription of the source. The implication is that all data of the source should be entered into the database. This project started in 1983 with a program that forced us to copy the source letter by letter.[26] This program that ran on a main-frame had no facilities for validation of the input, it had no range or type checking, and last but not least, it had an awful user-interface. The amount of time it took to enter the data for one year and the poor quality of the input made it necessary to reconsider this method of input.

The question arose as to whether it is possible to make a literal transcription of the source without keying in every letter. If the computer could be programmed to imitate some human capacities, there might be a answer. To be able to program the computer to do this, it is necessary to analyze the activities of a human being in the input-process. If a human being were to transcribe the *Paalgeld* portbooks, he would learn during the process of transcription. He would start to see that the same sort of data can always be found in the same places, so if he finds something illegible on the spot where there is usually the name of a harbor, he will think of a harbor-name that resembles the illegible entity as close as possible and quite often this will be a correct interpretation. Next to that he will need fewer and fewer letters to recognize a complete word: after having seen *Koppenhagen* a hundred times, seeing *Kopp.* will usually be enough to identify the entity. Of course this is a tricky approach: because of this tendency to recognize words before having read them completely variations at the end of a word might be overlooked.

---

[25] J.D. Ullman (1988)

[26] In 1984 the project started with a program developed by the Alpha-Informatica department of the University of Amsterdam.

A human would probably also develop forms of shorthand to represent long words that appear very frequently in a text: it becomes rather tedious to have to write *St. Petersburg* a few thousand times so one would develop some sort of code, which could easily be converted back to the original.

Furthermore, a human would quickly find regularities in a source even if he did not know the rules. He would see that the cargo for ships coming from Archangel was never specified, he would see that ships in ballast always paid the same levy, and he would see that ships coming from Norway usually carried wood. If he had learned the rules by heart, he would know what levy had to be paid for 14 lasts of barley, or he would know that if the levy was 7 stuivers and the cargo was barley, that the quantity had to be 14 last. If this form of human-behavior could be mechanized, it would be possible to accelerate the input process considerably. Translated to the language of information science: at the core of the problem are type and range checking, and default-generation and validation.

Most database management systems offer simple forms of checks on input, mainly type checking, range checking and validations. These checks should guarantee a minimum quality of the input, but they still leave much room for faulty input. Type-checking will guarantee that input will be of a predefined type: if the input is not of that type, an error message will be generated and new input will be prompted. Type checking will not allow character data to be entered into a numeric field, but the opposite is completely acceptable. A date-entry in the form "12-01-1789" would be acceptable as numeric input, where it would immediately be evaluated as -1778. It would also be acceptable as character input, since almost everything is acceptable as character input, and it would of course be acceptable as date input, if this date format were acceptable to the system. If a system has a special type for temporal data, such as the date-type of most database management systems, impossible dates like February 30 will not be accepted. Since the benefits of type-checking will only work if the right data type is chosen when the data-structure was defined, this choice is crucial. Type-checking is a safeguard against very crude mistakes in the input, but cannot filter out all possible input mistakes.

Range checking allows the user to define an upper and a lower limit in between which all entries of a given type are considered valid. A simple examples can be constructed using human ages: these must be positive numbers and cannot be over 140. So, if we would have a system where the date of birth of a person would have to be entered, we would set the lower limit of the range on the system-date, being today, and the upper limit on today plus 140 years, if we wanted to be absolutely on the safe side. Usually the upper limit would be put a little lower here. Since the *Paalgeld* portbooks are available for a limited period, it would be logical to limit date-entries to that period. The problem here is that within the whole period 1742 - 1836 there is a period for which the portbooks have not survived, namely the period 1743-

1770. To avoid dates from that period range checking is not enough, we also need a validation of the input against predefined rules. Most systems offer extensive facilities to define these valid-states, but if the domain of valid states is irregular it can be difficult to formulate exact rules.

Defaults are values that are either suggested by the computer which must be accepted by the user before they will be used, or they are the values that a computer will use if no other value is explicitly indicated. A simple example is the DIR command in DOS: this command will give you the content of the directory that you are working in at that time if you do not provide any parameters. If you do provide them, the system will use those. Another example is the DATE command: this will suggest that the system-date is the correct date, but as soon as the user starts to type in something else the suggestion will disappear.

The possibilities for default-generation are dependent on the type of data for which defaults must be generated. We discern three types of data: nominal data, ordinal data, and interval or ratio-data. Nominal data are divided in to categories only by their names, which are mutually exclusive. The names of the categories are arbitrary: they have no order. Names are a good example. Ordinal data are a bit like nominal data, only there relations between the various categories and the categories do have an order, but the exact interval between items is not known. Social categories are an example of this data-type: there is an order between the various categories, but the interval is arbitrarily chosen. The third group of interval or ratio data have fixed units, which have an absolute order. The only difference between ratio data and interval data is that ratio data have a conceivable zero-point. For historical research one can safely treat them as being the same.[27] Some data are hard to categorize: geographical names can be treated as nominal data, but one could also say that they are interval-data, since they are names for a coordinate somewhere on the globe, and hence are in a very well defined relation to each other.

There are two types of default generation: static, if the system always suggests the same value for input, and dynamic, when the system generates a different default for each new situation. The options for default generation for nominal data are limited, unless there is some regularity in the data. If there is a clear vision what the total input will be, the expected most frequent value, the mode, would be the best choice as default, since that value would be correct more than others. This mode could even be computed dynamically from the input so far. However, this would demand some statistical analysis after each input, which might slow down performance to unacceptable levels. However, the problem is, that the mode will be incorrect in most cases, and a default that is incorrect in most cases will start to cause irritation

[27] R. Floud (1973), 12

by the user. A way to work around this problem is to offer the user a choice of a series of values close to the mode in a menu like form, the pick-list.

In many cases static default generation will not produce a value which the user would accept even with non-nominal data. If there is a population of data in which the mode is not very clear, or if there is a multi-modal distribution, picking one value as default will not work. Usually one would not suggest a static default for data that have a high standard deviation from the mean of the population. If there is a clear correlation between two items that have to be entered, using a static default is not sensible. Considering the fact that most Dutch men have larger feet than Dutch women, it would be inefficient to suggest the same default value for both groups if we were setting up a database with shoe-sizes. This implies, that data should be entered in a prescribed order. An input form which asked for the input of shoe-size before the input of sex would make it impossible to use information about sex for generating a useful default.

In historical input projects quite often there is no prior knowledge of the expected frequency distribution of the data. Sometime upper and lower limits may be known, but more often they are not. In these cases default generation can only be done dynamically, based on a constant statistical analysis of the input so far. A simple brute force approach will not produce satisfying results. In these cases an input program will have to be written in which rules about the input will have to formulated and an inference mechanism to reason based on these rules: it will resemble a rule-based expert system very closely. It will need a knowledge-base in which the statistical facts about the data will have to be stored. It will have rule-base in which the rules about the data will be stored, including a description of correlations between items, and even range and type checks, and previous validations. It will use input as the database on which it will work to acquire new knowledge about the data. It must be able to use the rules in an intelligent way. It should even be able to formulate new rules during the process, and it should be able to reason with uncertainties. This implies a lot of programming in preparation for the input-process and will only be feasible in large scale projects, though the expected quality improvement might be an enticement for small scale projects.

The final check we have at the input stage is validation. Most database management systems allow the user to formulate special rules which the input should meet, usually in the form of a user-defined function to be tested after each input. The validation-functions can be as complex as the user wants and hence can slow down the system considerably.

In the next paragraphs I will discuss the solutions for default generation and data validation, that have been developed in this project, which were not all available in commercial off-the-shelf software at the time of the input. Modern database management systems offer some of this functionality to the user capable of some programming.

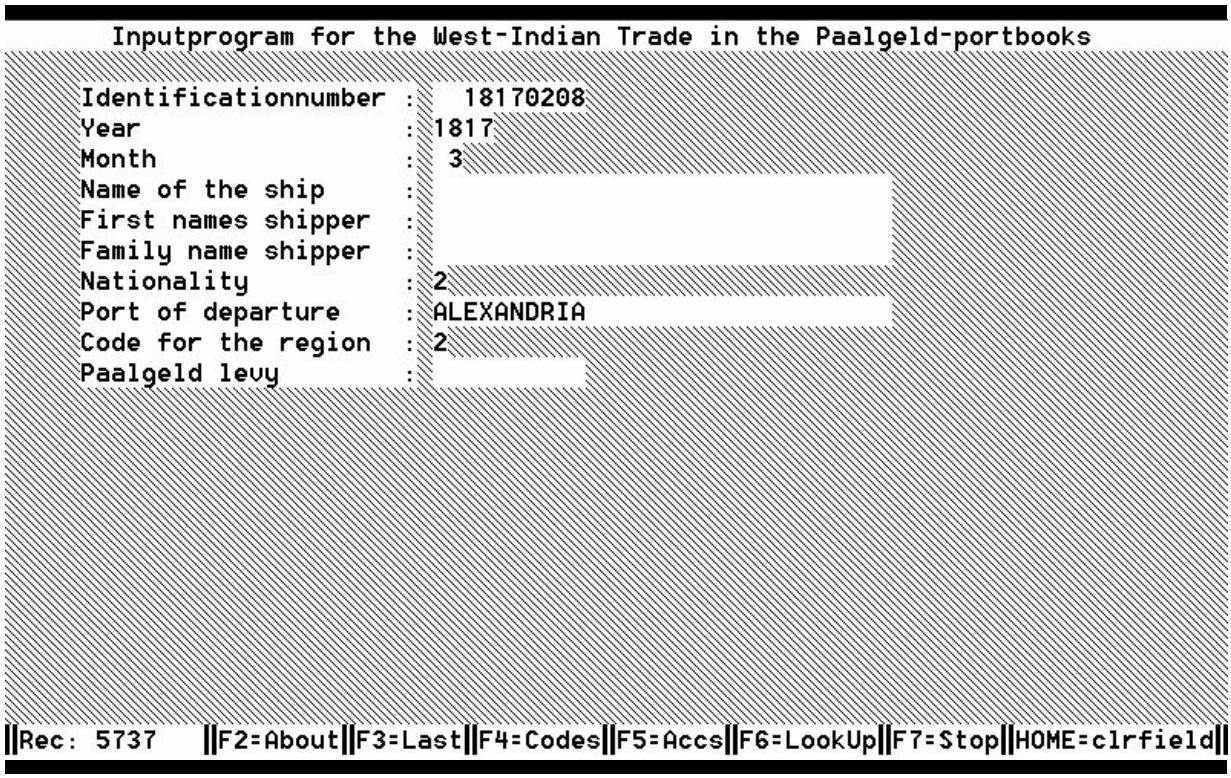

Figure 4 Screendump from the input program for the West Indian trade showing an example of Limited Carry over

4.4.4.2 Rule-Based Input

If there are interdependencies between some fields in a data set, the rules for these interdependencies should be formulated, stored and used for validation or default generation. For example: we should never find a cargo-specification in the first part of *Paalgeld* portbooks of a ship coming from London, but only the levy paid, because those ships were exempt from the obligation to specify the cargo. So the fields: CARGO, NUMBER_OF_ENTITIES and MEASURE can be marked 'blank' by the system itself as default. But more often than not historical sources are inconsistent: sometimes a cargo-specification *is* given for a ship from London. So the program should provide defaults which the historian may overrule. The input-program can only suggest defaults, the historian decides. Historical practice creates some problems for rule-based input: quite often the structure of the source is only discovered during the input process and this structure may not be static: it can change within a period. As described above, some sort of expert-system input program would be needed, to deal dynamically with changing conditions. To deal with changing domains of possible entries it would have to keep statistics on input so far and suggest defaults based on analysis of these statistics, or based on other rules. In one version of the input-program this was implemented,

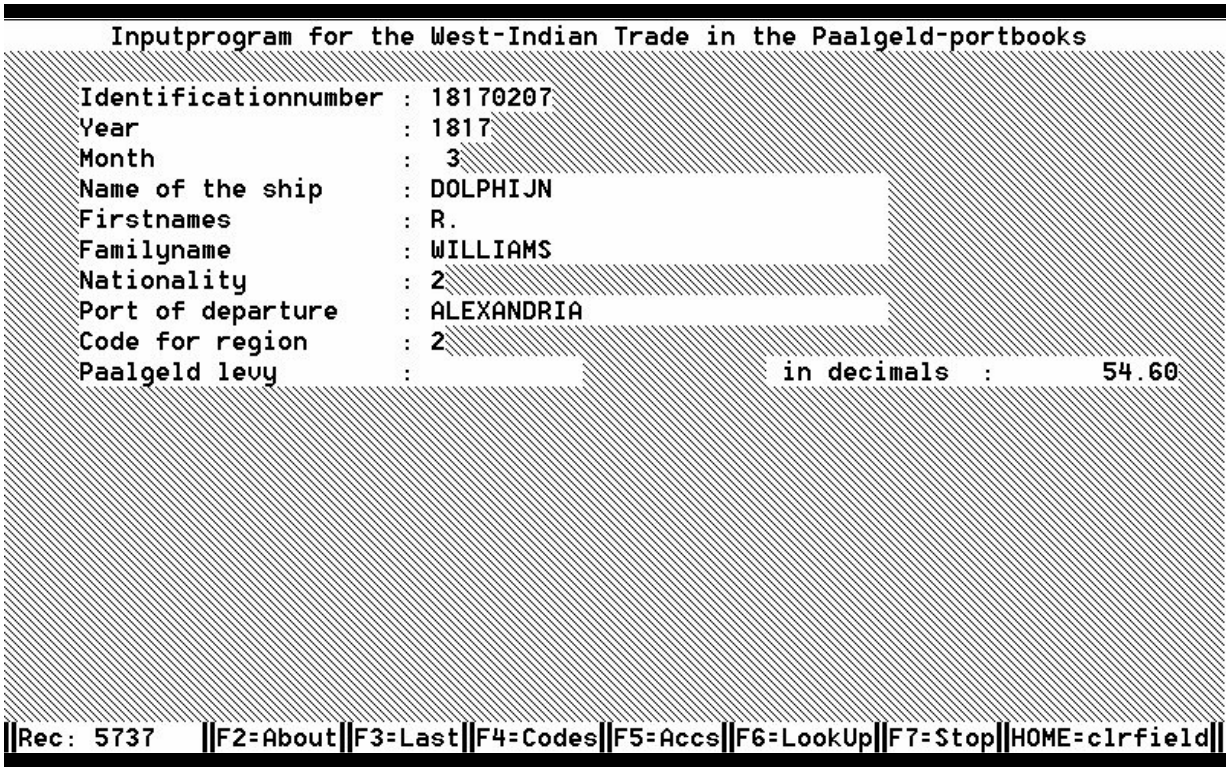

Figure 5 Screendump from the input program for the West-Indian trade showing an example of a recall of the last entry

but the required computations had such a serious negative influence on the performance of the program on the machine that I used that I decided to remove this feature.[28]

A simple example of rule-based default-generation can be explained with figure 4, which shows the screen that the input program for the portbooks concerning the West-Indian trade presents when a new entry is started. Here a *limited carry over* is used, which means that some of the values of the last input are carried over to this record. The identification number is generated automatically by the system. The year and month will probably be the same for a long series of input, so these items are carried over from the last record. The name of the ship and of the ship master are never the same in two successive cases, so these items should never be carried over. The practice of sailing in convoys of ships produces a high probability that two or more successive cases will come from the same port of departure, so this item is carried over. However, if a keystroke other than the ENTER-key is entered in this input-field, the suggested default will disappear.

The code for the region of the port of departure is generated automatically by the system from a table in which the code is given for every port. This information is an interpretation of

[28] Being an extremely low-budget project, almost all data-entry was done on a 4.77Mhz. 8088 PC.

the data in the portbooks, but since it can be filtered out very simply, there can be no objection to this procedure. Both port of departure and the code are stored, so recoding is always possible afterwards.

This input-screen shows another item -the nationality of the ship master, for which the carry over of the last input is also suggested as default. This piece of information is not found in the portbooks, but is an interpretation of the historian. If the name of the ship and of the ship master are clearly Dutch, a code 1 was entered. If it was very clear that both the name of the ship master and of the ship were not Dutch, a code 2 was entered. In all other cases a code 3 had to be entered. This information was needed for later analysis about the share of Dutch ship masters in the trade. This code could have been given algorithmicly after the input-process, but to define the rules for this would be so complicated that it was decided to input this added information at the input stage, when it would take very little time to make the decision. The disadvantage of this approach is, that no clear rules have been defined for the choice: it was a sophisticated though rather subjective choice.

Some database management systems offer the possibility of complete carry over of the last input. It is not advisable to use this facility, since it confuses the data-typist. It produces multiple cases of the same input when the data-typist thinks that the computer has not accepted the completed input and he will enter the same values again. On the other hand it can be very useful to be able to look at the last case, in order to avoid forgetting entries or duplicating them. In the input program this facility was taken care of by a module which could be called by hitting the F3 key of which an example is given in figure 5. On the computer screen the last entry would be put in a completely different color than the actual input.

4.4.4.4 Thesaurus-Based Input combined with an incremental search algorithm

All the conditions which the data to be entered must meet should be put in the data-dictionary. For each item the ranges, valid states and possible checks must be stored. For every field in the database that has possible repetitions in its content or a limited domain of possibilities, these must be described in an authority lists or thesaurus. If the domain cannot be defined exactly in advance the thesaurus or authority list could be empty at the beginning of the project. However, if not all input is possible and the data have a repetitive character it is advisable to generate the authority list dynamically from previous input. In figure 6 a screen

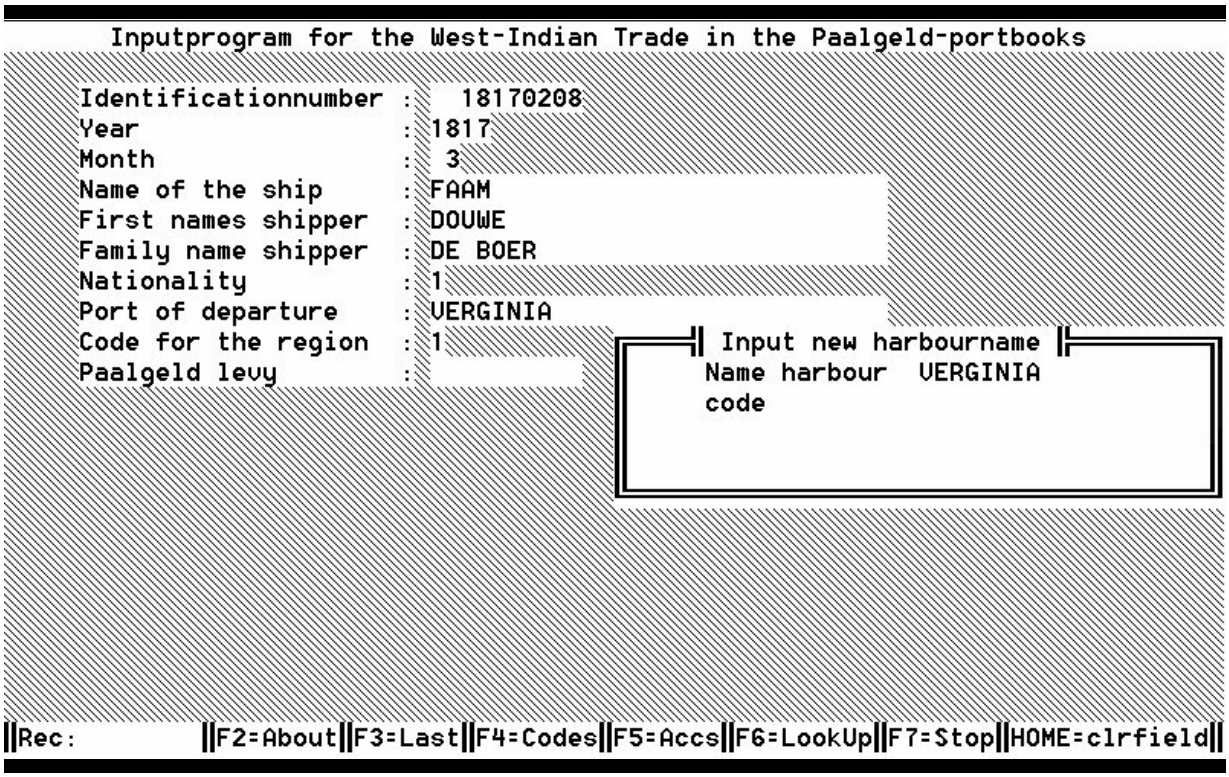

Figure 6 Screendump from the input program for the West-Indian trade showing that data must be in the authority list to be accepted

from the input program is presented in which the name Verginia is entered in the field for port of departure. The program searches the thesaurus for this value and since it is not found there, requires the data-typist to add it to the list first and optionally also reserves a code for it. This will assure that the thesaurus holds all possible values for the port of departure. The list that will grow during the input-process can be used for dynamic default-generation also. To do so the input program should evaluate each keystroke immediately and look for the closest match to the string typed in the authority list and generate this value as default, which should be accepted by hitting the ENTER-key. This so-called *command completion* or *incremental-searching* can speed up data-entry considerably if the items in the list do not resemble each other too much. To find *Christiansted* or *Christiansund* eleven characters should be typed in before the right form would be found. In such cases it can be helpful to add a facility to navigate through the authority list using a mouse or the cursor-keys. Usually entries are found within four key-strokes. In the input-program I made, I used the incremental search on all the input fields for names and places. It is obvious that the benefits of this approach increase with the size of the input-project. In the beginning the thesaurus will be empty and must be filled, and this will require some extra operations. But after entering about three hundred records the

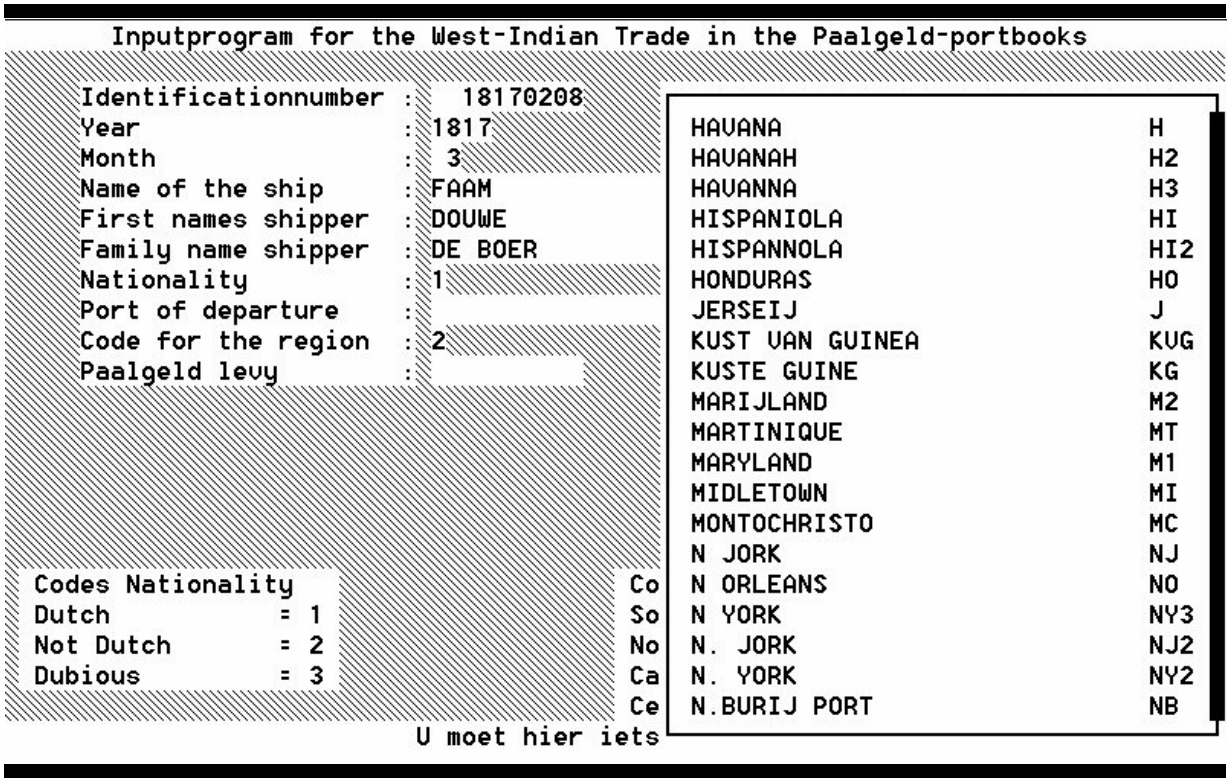

Figure 7 Screendump from the input program for the West-Indian trade showing a pick-list and the codes for coded-input

list will have most of the forms and from then on the default-generation will reduce the amount of necessary keystrokes increasingly.

4.4.4.6 Code-Based Input

The methods described in the last paragraph helped to produce a reliable transcript of the original source while reducing the number of keys-strokes necessary. However, using command completion combined with pick-lists still required more data-typing than was desired. Elaborating on the idea that humans would use some sort of short-hand codes to write down the most frequent forms they would encounter, I put a feature in the input-program, which I have called *code-based input*. For every field that had repetitive characteristics of form a code-book was created, in this case only for the names of the ports of origin.[29] These codes, preferably with a mnemonic value, were no longer than 3 or 4 characters and in the beginning were used only for longer character strings, since the proportional gain in speed related to the effort necessary to learn these codes was not high enough for short strings. These codes are used as a form of

[29]In the input-program for the European trade I used this feature also for the cargoes of the ships.

shorthand allowing the historian to type them in instead of long strings and thus saving keystrokes while the program immediately looks up the code in a table and replaces it by its complete form. Spelling-variations can be kept intact by using a letter-code combined with a digit: in figure 6 the codes for some of the spelling variations for New York can be seen. There were many forms for this place, and since the transcript should be as close as possible to the original source, all variations were recorded. So there is NJ for *N Jork* and NY3 for *N York*. These input-codes have no further purpose than accelerating the input process. As soon as the code is typed in, the program substitutes the full form for the code. Since coding is prone to mistakes the full form that is attached to the code should always be shown so that the historian can immediately see if a wrong code has been picked and correct it if necessary.

4.4.6 What are the advantages of these methods?

What do I gain from these methods? Quite obviously that depends on a number of factors. First of all it depends on the length of the strings that would have been entered if these methods were not used. If the average entry does not exceed four letters, these methods will not improve input-speed dramatically. But the speed-improvement by the use of code-based and thesaurus-based input can be quantified.

I have calculated the speed-improvement for those input items on which the described techniques were used, excluding only the field where measures had to be typed in. For most measures abbreviations were used, which were seldom longer than four characters, which means that there was very little to gain on this input. To calculate the input-acceleration a frequency distribution of the length in characters of these items was made (see table 1). Since on average all input was completed within four key-strokes, all lengths smaller or equal to four were treated as one frequency: on this group no gain was made. For all other lengths, the frequency was multiplied by the number of key-strokes gained. All lengths greater than 20 were treated as being 20. From this frequency-distribution I can give an estimate of the total number of keystrokes that would have been needed if no speed-up methods had been used: the sum of the Multiplication of all lengths with their frequencies, being 1.827.415. The computed reduction of keystrokes needed was 697.647, being 38%! These estimates are conservative: for quite a lot of input-items the reduction reached with coded-input was much higher: choosing very short codes for the most frequent occurrences can be very productive.

Assuming that a fast data typist, who also has to read this extremely difficult eighteenth century handwriting, can key in one character every four seconds on average reading time included, the improvement in time can be computed: 697.647 X 4 seconds = 2.790.588

word	Cargoes		Places		First Names		Family Names		Ship names	
length	**freq.**	**gain**	**freq.**	**gain**	**freq.**	**gain**	**freq.**	**gain**	**freq.**	**gain**
<=4	31939	0	2982	0	39054	0	6515	0	294	0
5	24187	24187	2885	2885	6475	6475	9069	9069	416	416
6	19895	21591	7224	14448	5703	11406	16102	32204	505	1010
7	7197	21591	6599	19797	2614	7842	13000	39000	523	1569
8	11775	47100	2963	11852	1775	7100	7701	30804	648	2592
9	2607	13035	8694	43470	817	4085	3173	15865	487	2435
10	4046	24276	2622	15732	491	2946	1359	8154	445	2670
11	478	3346	19740	138180	295	2065	437	3059	373	2611
12	658	5264	1093	8744	171	1368	147	1176	251	2008
13	109	981	2070	18630	104	936	47	423	249	2241
14	32	320	562	5620	56	560	40	400	279	2790
15	366	4026	30	330	31	341	3	33	232	2552
16	13	156	2	24	7	84	3	36	219	2628
17	9	117	2	26	6	78	1	13	161	2093
18	467	6538	1	14	3	42	2	28	110	1540
19	300	4500			1	15			110	1650
>=20	12	192	134	2144			4	64	241	3856
total		195419		281896		45343		140328		34661
total	**gain**		697647							

Table 1 Frequency distribution of input-items and the gain on needed key-strokes per frequency

seconds, or just over 775 hours.[30] Assuming that a data-typist would be able to do this strenuous work for eight hours a day, this means that this approach reduced the time needed for input with about 97 days! Of course this calculation is not correct, since the methods were developed during the input-process, so not all input has benefitted from the algorithms used in the final version of the input-program. Furthermore, the time needed for coding the various versions of the input-programs was not deducted. But since this project was completed, the algorithms developed here have been used in a number of other research projects, making it possible to do large-scale data entry projects with limited time and funding. I estimate that I saved about a year on the input process.

[30] Compared to normal typing 4 seconds for every keystroke sounds very slow, but if reading time is included this is realistic and maybe even optimistic. Deciphering eighteenth century handwriting can take several minutes for one record. The actual amount of keystrokes is not a good indicator of the complexity of the reading process. The input program had a timing procedure which showed incredible variance of input speed.

Is acceleration of the input the only thing gained by these methods? Maybe more important than the whole increase of speed is the increased quality of the digital transcript of the source, compared to normal input. Since almost all data are entered via thesaurus-based input mistakes are rare and those that are found can be corrected systematically. Another advantage of these sort of input-programs, is that they help the historian to decipher the sometimes difficult eighteenth century handwriting. The incremental-search will suggest sensible values for items of which only the first characters can be deciphered. In other cases looking through the pick-lists can trigger the recognition of a form that made no sense before. Maybe the best result of this approach is, that it reduces the irritation of the researcher, who otherwise would have to do even more boring repetitious work at the keyboard, slowly losing the enthusiasm he once had for his project.

4.4.8 From meta-source to encoded source transcript

The digital transcript of the source has great advantages over the original: it can be manipulated, searched and restructured at will. But the digital transcript has all the peculiarities of the original and is not fit for statistical analysis: it must be transformed to an encoded source transcript through an historical interpretation of the data. Different forms of the same entity have to be standardized to a singular form or code. The categories needed for further analysis had to be made up and the data had to be coded so that they can be recognized as belonging to a certain category. For this coding I have relied on the coding systems used by Johansen for the Soundtollregisters and Knoppers for the *Galjootsgeldregisters*.[31] They used a three digit and three letter code for harbor names: the first three digits indicate the region where this harbor was located, the three letters were a sort of mnemonic code to distinguish several places in the same area. Since harbors on the Western Hemisphere only appear very rarely in the sources they used, I had to add codes for these ports to the system. For cargo items they used a three digit code: the variety of goods that were imported to Amsterdam however, was greater than 1000 different sorts. To keep the same categorization Johansen and Knoppers used, I changed the code to a 4 digit code, adding a 0 to all their codes. This created enough room for all the required new codes. Since the input stage had produced the thesauri for all the fields, it was rather simple to write programs to do the coding of the data automatically. When I discovered at a later stage that the chosen coding for port names was not very well suited to make the categories I wanted, it was a simple thing to let the computer recode all 57603 entries for port names.

[31] J. Knoppers (1976), H.C. Johansen (1983), 16

The product of this second phase was the encoded source transcript, which could be used for statistical analysis. Since coding is by nature a one way step it is essential to save the original form of the digital source transcript, so that if the analysis produced unexpected results, it could still be checked if these were the result of inaccurate coding. The original digital transcript of the source is also the best form to be archived for later use by other researchers. Coding is based on historical interpretations and each historian should have the chance to work with the uncoded material.

4.4.10 Cross-validation by nominal record linkage

In order to determine the quality of the data in the *Paalgeld* portbooks it was necessary to check the data with information gathered from other sources. This might also provide an answer to the question about the real nature of the date mentioned in the portbooks. To do this the information from various sources must be combined which can only be done if they share a common item. All the sources that were used for this project have one item in common: the name of the ship master. So the solution was to take one case from the *Paalgeld* portbooks, and then look for the name of the ship master of that case in the other sources. However, this is not as simple as it seems. First of all, there is no standardization of names: forms of the same name can differ depending on how the clerk who took the name down interpreted what he heard. If the same clerk did the work on the sources used, which was probably the case with the *Buitenvuurgeld* and the *Paalgeld*, there is a fair chance that the spelling of names maybe more or less the same. But to identify forms of the same name when they have been taken down by different clerks is very difficult. Some names have spelling-variations, other names are completely different. In one case a patronym may have been used, in other cases a toponym, and in other cases a real family name, and sometimes the clerk translated names. This problem of nominal record linkage has been discussed in a large number of publications.[32]

In this project nominal record linkage only played a minor role for a number of reasons. First of all, the results of tests showed that the process would not be a straightforward as I had hoped: even the link between the registers of the Buitenvuurgeld and those of the Paalgeld, which I expected to be simple was complicated. To solve the problem I wrote a program that took the name from the *Paalgeld* portbooks as input and searched that string in the *Buitenvuurgeld*. All the hits found, were presented and a confirmation was asked before the

[32] A query in the 2.0 version of A historical Computing Bibliography listed 48 titles on this subject. (This bibliography is a project of the Association for History and Computing, and the Humanities Computing Centre, of Queen Mary and Westfield, London. An on-line version is available on the Web-site of the AHC. (http://www.let.rug.nl/~ahc/).

data from the two records were really joined. If no perfect matching hits were found, a sound-alike algorithm, Soundex, was used to search for forms that closely resembled the search item. If this would produce no results, the researcher could type in a form of the name, which might not be found by an algorithmic approach, like changes between a family name and a toponym. The result was a almost perfect match between the portbooks of the *Buitenvuurgeld* and the *Paalgeld*, which was expected since both registers were produced by the same agency. However, the effort and time this approach demanded in relation to the marginal importance of the problem for this study, made me decide to keep the linking restricted to perfect matches. The very small number of links that could be established this way are no real problem, since the record linkage was done to confirm the results of the search for seasonal patterns in the data of the portbooks, which have been described earlier. I decided that if the analysis of the data of the joins that were made with the small amount of successfully linked records would confirm these findings, I would not invest more time in trying to establish more links between the available sources.

If the data matched a simple hypothesis, the arrival of a ship would be first mentioned in the *Zeetijdingen in de Amsterdamsche Courant*, indicating that the ship had arrived before Texel or in Het Vlie. The trip from there to Amsterdam might take a week to ten days, assuming a very slow scenario. When the ship was in the port of Amsterdam the ship master would be available to make new contracts, so in this period a *Charteparthij* might be expected. Before leaving the port for a new voyage, the *Monsterrol* had to be given to the office of the *Waterschout*, probably just before departure. In the period from around the date of the *chertheparthij* until the date of the *Monsterrol* a ship would be in port and the ship master would have to pay the duties in that time. If the date in the *Paalgeld* portbooks falls in this period, it can be assumed that this date is the actual day of payment of the levy, or the day of registration of that payment. In any case this would mean that the ship was actually in the harbor at that time. This would imply that the date could be used to uncover seasonal patterns in the trade.

I have processed data for the year 1778 for the four sources mentioned above. The only piece of information that all these sources have in common is the name of the ship master. Connecting the data from these sources proved to be extremely difficult. A one-to-one linkage of two sources still produced acceptable results, if only perfect matches were accepted. In the Zeetijdingen in de Amsterdamsche Courant 2566 arrivals of ships are mentioned. Of these 411 could be automatically linked with cases from the *Paalgeld* portbooks. The average time between the date of the message in the paper and the date mentioned in the portbooks is 29 days. A total number of 740 contracts were traced in the archives for 1778, of which 177 could be linked to the portbooks. The average time between the dates of the contracts and the

portbooks is 43 days. Of the 699 muster-roles that were found for 1778 only 156 could be linked to the portbooks. The average time between the dates of the muster-roles and the portbooks is 3 days. A linkage on a perfect match of the ship master was successful for 101 cases, but confirmed the results of the one-to-one linkage. The information gathered from the newspaper and from the muster-roles confirms our expectations: the message of the arrival appeared about four weeks before the date in the portbooks. The date from the muster-roles comes about three days after the date in the portbooks. From this can concluded that the date in the *Paalgeld* portbooks probably is the date of registration of the payment of *Paalgeld.* Although according to the rules a ship master had to pay *Paalgeld* before unloading his cargo, it seems to have been common practice to pay this impost only just before leaving the port again. But since the interval between the messages in the newspaper and the dates in the muster-roles is about 32 days, I may safely assume that the date in the portbooks provides an indication of the actual arrival date of the ship. The dates found in the ship-contracts, which on average are outside the period that the ship should be in port according to the other data, can only be explained by the fact that quite often it was not the ship master himself, but someone acting on his behalf, who actually signed the contract. Still, this sequence of a contract just before the ship's arrival is mentioned in the newspaper, four weeks later the payment of *Paalgeld* and three days later a muster-role for a new voyage of the ship, indicates a very efficient organization. These results of this record-linkage process and the seasonal patterns described earlier allow us to assume that the dates mentioned in the *Paalgeld* portbooks are about two to three weeks after the actual arrival in port.

Though this conclusion is based on a linkage process that was only successful for less than 3% of the total number of cases for that year, I saw no reason to invest more time in trying to improve this score. To achieve a greater success-ratio in the linkage process an amount of time should have been invested which would not have been in relation to the importance of the problem. Furthermore, I doubt if a greater number of successful links would provide different results. The successful links on perfect matches are more or less randomly spread over the total population and I assume that the results are representative for the whole population. [33]

The 101 cases for which the data from all sources could be combined confirm the assumptions made about the quality of the *Paalgeld* portbooks: the names of the ports of origin that are mentioned in some sources usually match. In some cases another port is mentioned, but this usually is a port from the same region. Only in one case Norway is mentioned in one source

[33] The program for semi-automated record linkage was later perfected and used in two other research projects: see G.M. Welling (1995)

and Riga in the other: however this maybe a case in which a perfect match on the name of the ship master created a link between the data of two consecutive voyages.

4.6 Software used

During the years of this project the data-set migrated from system to system and was treated with consecutive generations of software. I started the project on the mini computer of the Arts Faculty of the University of Amsterdam in 1984 and finally the data set ended up on my private Pentium 200 PC and the Unix system of the Arts Faculty of the University of Groningen. The first versions of the input program were written for me Pascal, but after migrating the project to PC's I switched to Turbo Basic and later rewrote and improved the programs in various versions of Clipper. The database was treated with all versions of dBase and FoxBase that came out since 1984, and finally ended up in Ms-Access. The analysis was done mainly using the built in facilities of these database management systems and if needed I wrote small applications. In some cases SPSS-PC was used and even spreadsheets. The choice was always simple: I used the software that was available, easy to understand, and could do the job. The choice was always made to make the computer my slave and not vice-versa.

To avoid problems of getting funding for the publication of the extensive appendices, in which all the results of this research are presented-even those which are not discussed in this study-, I have created an index page on the World Wide Web on the Internet, from which one can access all the results and data sets. The URL is:

http://www.let.rug.nl/~welling/paalgeld/appendix.html.