A General Compression Scheme for Databases

Student: Adam Cannane
Supervisor: Justin Zobel

October 27, 1997

Abstract

Compression of databases not only achieves a reduction in storage space but can reduce overall retrieval times. Current schemes such as *gzip* and *compress* are impractical for the purposes of databases as they do not allow individual records to be retrieved. A recent compression scheme, SEQUITUR, allows quick decompression of any individual section of the database, however it uses extravagant amounts of main memory during compression. We propose an extension to the SEQUITUR algorithm that limits main memory usage during compression. We show that our new scheme achieves compression performance only slightly worse than SEQUITUR while allowing massive reductions in memory usage.

1 Introduction

Databases store large collections of related electronic data. The data is stored in the form of text, images, sound or binary data such as numbers. The major function of a database is to allow retrieval of records that match queries generated by users [4]. Often a database is shared among several concurrent users. Ideally, a database is compressed to reduce the storage cost.

There are many general-purpose compression schemes that will effectively compress the different types of data. The schemes generally use adaptive models that encode the data in a single pass — the *compress* utility is one such compression scheme [1]. Compression schemes can be special–purpose, that is, achieve better compression for a particular type of data. On the other hand, compression schemes can exist for special–purpose applications such as databases. Databases typically use compression schemes that have semi-static models, in order to allow independent decompression of records. Independent decompression means that any record from the database can be decompressed on its own without having to decompress the entire database. This is why adaptive compression schemes can not be used on databases. However, there is no general–purpose compression scheme for databases that can effectively compress all kinds of data types.

Nevill-Manning [11] proposed an algorithm, SEQUITUR, that identifies structure within a sequence of symbols by recognising repetition. The repetition found within a sequence is often a sequential collection of symbols or subsequence. SEQUITUR represents the structure it has identified as a hierarchy. The hierarchical structure removes the redundant subsequences and presents itself as potentially an effective compression technique for any type of data. The hierarchy is stored in the form of a context–free grammar, where the grammar is incrementally updated during processing to allow the structure to reflect the original at any point [14]. Unfortunately, the design of the SEQUITUR algorithm prohibits it from being directly applicable to databases, primarily due to the extravagant use of memory.

We propose two alternative extensions to the SEQUITUR algorithm that address aspects of the problem of unbounded memory, at the expense of reducing the effectiveness of the compression scheme. The first extension, *model–combining*, segments the original sequence into small sections. Each section permits SEQUITUR to generate a grammar given a limited amount of memory. The grammars for each section are then merged together, by identifying similarities between them, to produce a single model for compression. The second extension, referred to as *sequ–snapshot*, restricts the number of symbols over which the constraints of SEQUITUR can be applied. The restricted number of symbols that SEQUITUR is applied to prohibits it from identifying large subsequences and consequently has some effect on the compression result.

We put forward an approach that proves to be a good general–purpose scheme that will compress the database, irrespective of the format of the data type. Our results show that the reduction in compression performance is minimal compared to the reduction in memory resources.

This paper is organised as follows. In Section 2 we provide a background on data compression. Section 3 looks at choosing a model for the database. The SEQUITUR algorithm is explained in detail in Section 4. Section 5 presents the two approaches to bound memory size and in Section 6 we describe the test collection used for our experiments. Section 7 looks at the results from our experiments. Finally, Section 8 presents our conclusions and areas for possible future work.

2 Data compression

Data compression techniques seek to code redundant information to reduce overall storage and transmission costs. The transmission costs are improved by better using communication channel bandwidths. It has also been shown that improving data transfer, in exchange for decompression time spent by the CPU, can result in significant decreases in the overall time for retrieving data, by reducing the transfer costs from disk [22].

Irrespective of the method and approach used to compress a sequence of data, compression involves two different activities: modelling and coding [16]. The modelling process constructs a model for the data that represents distinct symbols. It also provides an estimate of the likelihood of a symbol to appear, its probability. The coding process produces a compressed representation of the data, mapping symbols to different codewords by using the model. A detailed discussion of compression is presented in Sections 2.2 and 2.3.

The removal of redundant information is generally achieved by one of two distinct approaches, *lossy* or *lossless* compression. The lossy approach reduces detail of the data stored, that is, some of the original data is lost. Lossy compression is often used for images and sound, where the original digital image or sound itself is an approximation; a small degradation in quality and information stored has little or no impact on the application. A lossless approach is required in other applications where loss of the original data is unacceptable.

Decompression of data using a lossless approach will result in an exact copy of the original. Text is a typical example that relies upon lossless compression. As we are proposing a universal scheme that caters for all data types, we restrict ourselves to a lossless compression scheme. The next section looks at the different methods that can be used to achieve lossless compression of data.

2.1 Approaches to Lossless compression

A lossless approach to compression can be effectively implemented using either a *symbol–wise* or *dictionary* method, where the difference is how the data is interpreted. The symbol–wise method, also known as the statistical method, considers the data as a series of individual symbols. Each unique symbol is assigned a probability that corresponds to how often it occurs in the data. The probabilities can also be used to predict upcoming symbols; we discuss estimation of probabilities in Section 2.2. In contrast, dictionary methods consider multiple symbols together as a collective group. For example, consider the sequence

the cat sat on the mat

The collective group of symbols that make up the subsequence "*the*" in the above sequence may be added as an entry to the dictionary. The dictionary entry is assigned a reference symbol and replaces all occurrences of the subsequence in the main sequence. The sequence now contains a dictionary entry and is represented as:

Sequence	Dictionary
Φ cat sat on Φ mat	$\Phi \Rightarrow$ the

Other potential dictionary entries exist in the above sequence. The group of symbols "*at*" is another possible dictionary entry. The symbol–wise and dictionary methods are different ways of modelling the input data.

2.2 Modelling

Modelling involves generating a representation of the distinct symbols in the data. The model also stores information on how often each symbol occurred throughout the data, which is represented as a probability. The probabilities for each unique symbol make up the probability distribution for the data.

Figure 1 illustrates a model's purpose in the compression process. A model is used by an encoder to construct a code for each symbol, based on

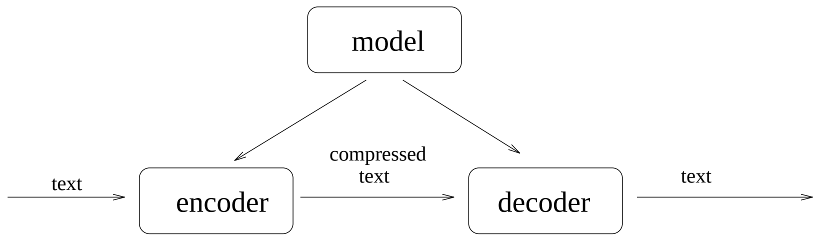

Figure 1: The compression process

the probabilities. The same model is then used by the decoder to reproduce the original data from the compressed symbol.

There are 3 main types of models: static models, that remain the same for each symbol of the sequence; adaptive or dynamic models, that alter the model after each symbol is inspected; and semi-static models, that change the model during encoding, but remain static during decoding.

Static models make assumptions about data that has not yet been seen; they are built by considering the average probability distribution derived from previously gathered statistics on similar data. To understand this concept, consider ordinary text: In English, the letter "e" is the most commonly occurring symbol. Therefore, the highest probability from the distribution would be assigned to it, whereas the letters "x" and "z" are less common and would have a smaller probability.

Static models do not always result in efficient compression. Compression becomes inefficient when the model provides a probability distribution that is a poor approximation of each symbol's frequency in the data. A static model performs poorly when the data being compressed is dissimilar to the data used to first produce the model.

Adaptive models address the problem of poor probability distributions for data by altering the distribution as each new symbol of the data is inspected. Adaptive models only require a single pass over the data, creating a new model for each symbol observed. Initially, before any of the data has been inspected, adaptive models begin with a general probability distribution, similar to one that would be used by a static model; the benefits of an adaptive model start to evolve as more data is inspected. Adaptive models achieve benefits by recalculating the probability distribution after each symbol is inspected, taking advantage of the local properties of the data.

Since the probability distribution for an adaptive model alters after each symbol, all preceding symbols need to be decoded to determine the model for any symbol in the sequence. A limitation on adaptive models is that in order to reconstruct the original sequence, the encoded symbols must be decoded from the beginning.

A compromise approach to modelling data is to use a semi-static model. A semi-static model requires two passes over the data. An initial pass gathers the statistics necessary to build the model, where in dictionary-based methods, the dictionary entries are formed. Symbol–wise methods use this initial pass to accumulate symbol probabilities. The second pass of the data is used to encode the symbols according to the model created during the first pass.

The characteristics of semi-static modelling combine together the benefits achieved by adaptive and static models. A semi-static model makes good use of specific properties of the data while at the same time remains static during decoding to allow independent decompression. The disadvantage of a semi–static model is that two passes of the data are required and the model parameters need to be stored with the compressed data [22]. The Huffman compression algorithm is a typical example of this. The next section discusses coding, that is, using a model to produce a compressed representation of the data.

2.3 Coding

Coding is the second activity of the compression, a process that involves the mapping of symbols to different codewords to achieve a reduction in the size of the input. A codeword generally consists of a collection of bits that form a representation for a symbol. Each symbol has a unique codeword that is determined from the probability distribution supplied by the model. The transformation of the original data into a sequence of codewords forms a bit-stream, and is also a part of the coding process.

A symbol that is common throughout the data, one having a high probability, is represented by fewer bits than a rare symbol. In some cases, the length of a codeword will be larger than the length of the initial symbol, because the length of a codeword is directly related to the symbols probability.

The length of the codeword in bits for a symbol is called the information content. If the ith symbol has a probability p_i, then the information content is $-\log_2 p_i$ bits. The average cost per symbol of storing the information over the entire alphabet of n symbols is the *entropy* [17] and is given by

$$\sum_{i=1}^{n} -p_i \cdot \log_2 p_i$$

Entropy is measured in bits per symbol and represents the best possible compression that can be achieved using a particular model [9]. The information content, that is, the length of the codeword for a symbol, may be a decimal value since the information content is calculated using the function $-\log_2 p_i$. For example, assigning the three letters "abc" an equal probability of 1/3 will result in an entropy = 1.58 bits per symbol. A static code for each letter must consist of at least two bits — either 00,01,10 or 11. Static codes can not represent symbols in fractional bits. Huffman coding is a coding technique that provides the best possible compression for a code that cannot represent fractions of a bit [1]. In many applications, Huffman coding achieves compression performance near the entropy, but performs poorly when the probabilities are highly skew — the probability p_i is such that $p_i \gg 0.5$.

Arithmetic coding [21], is another coding technique that represents a string of symbols as a single probability and effectively represents individual symbols in fractional numbers of bits. In general, arithmetic coding is the preferred coding technique, especially for high performance adaptive models. It is however, typically slower than Huffman coding [20].

3 Choice of model

Although the primary aim of compression is often to achieve a significant space saving [8], it is necessary at the same time for the scheme to be efficient for practical purposes. A database requires a compression technique that will allow efficient retrieval of the compressed data, as well as produce a saving in storage costs. A scheme that achieves optimal compression, yet is unreasonably slow during decompression, is not practical.

Section 2.2 described the three types of models that can be used for decompression — static, adaptive and semi-static. Benefits of each of these methods and their inefficiencies were briefly explained, however the application of a model to databases introduces factors that contribute to the resulting choice of model.

Modern compression techniques are generally adaptive, to avoid the transmission of the model, and so the data can be compressed in a single pass. Databases are often divided into multiple records, documents or easily segmented parts. It is necessary that these divisions can be decompressed independently. In general, adaptive techniques are not effective for database applications, since they code data as a function of both the preceding symbols and the initial probability distribution [23]. Similarly, an adaptive code, such as arithmetic coding, is not practical for database compression as it is too slow [2]. An adaptive code would have to encode each record individually to maintain independent decompressibility, there by restricting the compression gains. To allow atomic decompression, as well as quick retrieval, the only presently feasible alternative is to have a single model for the entire database.

A semi-static model tends to be better suited to data than a static model [1] and thus, is generally the reason why semi-static models are used for databases, rather than static models. Semi-static models provide a single model for the entire database that allows fast decompression at any point in the compressed data. Previously, we mentioned that semi-static models required two passes over the data. This does not disadvantage the compression technique for static databases, since effort can be spent on compression, providing retrieval remains fast. The only disadvantage with a semi-static model is that it needs to be stored with the database.

Previously, there did not exist a good general–purpose semi-static modelling technique. A recent compression scheme, SEQUITUR [15], that uses a semi-static model and dictionary approach was presented by Nevill-Manning. The SEQUITUR algorithm presents itself as a general-purpose compression scheme for databases, since it achieves good compression and is a semi–static modelling technique.

4 SEQUITUR

SEQUITUR is an algorithm that forms a grammar from a sequence, identifying repeated phrases in the sequence. It has been proven that the detection of repeated phrases performs well as a compression scheme [13]. The two main reasons SEQUITUR is a good scheme for databases are; first, it uses semi-static modelling so individual parts of the database can be decoded independently; and, second, it achieves good compression for large data collections.

Compression is achieved by removing the repetitions, where the repetitions consist of small subsequences from the original sequence. The subsequences often are made up of even smaller repetitions. The smaller repetitions within a subsequence form a hierarchy, with the smallest repetition possible, two consecutive symbols, at the base of the hierarchy. The hierarchical structure built by repetitive phrases is represented by SEQUITUR in the form of a context–free grammar. The following sections give a concise description of the SEQUITUR algorithm, including a brief explanation of context–free grammars.

Sequence	Grammar	Sequence	Grammar
cannanecannane	$S \rightarrow \gamma\ \gamma$	cannane	$S \rightarrow c\ \Lambda\ n\ \Lambda\ e$
	$\lambda \rightarrow a\ n$		$\Lambda \rightarrow a\ \Lambda \mid n$
	$\gamma \rightarrow c\ \lambda\ n\ \lambda\ e$		

Figure 2: Examples of sequences and their associated context-free grammars.

4.1 Context–free grammars

SEQUITUR constructs a hierarchical structure to maintain references to repetitive phrases it has detected. The hierarchical structure that it builds for a sequence is represented as a context–free grammar. Context–free grammars represent structure by having the following characteristics [18]:

- Context–free grammars consist of a finite set of grammar rules.
- Each rule has a left-hand and right-hand side separated by an arrow $\rightarrow$.
- The left-hand side of a rule consists of only one symbol, a non-terminal.
- The right-hand side of a rule must be a string containing at least one symbol, either a non-terminal, or a terminal, or a combination of both.
- Nested rules are non-terminals within non-terminals that form the hierarchy. For example, A$\rightarrow$bB, where B is a non–terminal, is a nested rule.
- Terminals are a set of indecomposable symbols that the original sequence is constructed from.
- Non-terminals are similar in behaviour to variables, they represent a defined string of terminals.

There exists a single non-terminal within the rule set that is the origin at which all strings of the language of the grammar begin. The non-terminal is often denoted by an S symbol and for the rest of this paper will be referred to as rule S, the starting rule. Figure 2 shows two examples of context–free grammars. The Λ, λ and γ symbols represent non–terminals, while the terminals are the letters from the set $\{$a,c,e,n$\}$. The context–free grammar that SEQUITUR produces requires certain constraints to be applied, so that only the original sequence is derived from rule S.

4.2 Grammar constraints

The context–free grammar that SEQUITUR constructs has two constraints applied. The first constraint is that it remain unique, only being able to reproduce a single string, that is, the original sequence. Consider the two context–free grammars given in Figure 2. Both of the grammars are context–free, however, the grammar on the right can produce an infinite number of strings. The following examples represent some of the strings that can be produced from the grammar:

cannne   cnnane   caanne   cannane

A grammar that adheres to the constraint, causing rule S to remain unique,is deterministic, and therefore requires no decisions to be made regarding the replacement of non–terminals with symbols during decoding.

Recursive rules contain a non–terminal on the right-hand side of the rule that is the same as the non–terminal on the left-hand side. These rules must be prohibited to ensure a deterministic grammar. The context–free grammar on the right in Figure 2 contains a recursive rule and is therefore non-deterministic. A decision needs to be made as to whether the non-terminal Λ, is replaced by the terminal n or the sequence aΛ, causing the rule to be repeatedly applied.

SEQUITUR is a dictionary-based scheme, with each dictionary entry corresponding to a rule from the grammar. The dictionary is adaptive during the first pass over the data, because the model alters as SEQUITUR incrementally inspects each symbol. SEQUITUR updates the model as if the last symbol inspected could be the end of the data sequence. This allows a sequence of any length to be independently decompressed at any stage. Another advantage is that it identifies structure within the data, which is ideal for databases, as most are structured.

SEQUITUR uses a *digram* approach, where it considers consecutive symbols as a single entity. For example, the sequence “abc” consists of two digrams: “ab” and “bc”. SEQUITUR differs from conventional digram compression schemes by enforcing two simple constraints:

1. Digram uniqueness: No single digram may appear in the grammar more than once.

2. Rule utility: A rule of the grammar must be referred to at least twice.

Digram uniqueness identifies repetition of two symbols by creating a dictionary entry that is referenced by both occurrences of the digram. Rule utility allows repetitions that are larger than two symbols, by eliminating superfluous rules [11]. The full affect of these constraints on the grammar is explained in Sections 4.3 and 4.4.

4.3 Digram Uniqueness

Digram uniqueness checks for duplicate digrams in a sequence and removes them by creating new rules of the grammar. To ensure that no two digrams are the same, a check of the last digram is required after each new symbol is appended to rule S. The new digram, consisting of the last symbol added and its predecessor, is checked against all of the other digrams to ensure uniqueness. If the digram is not unique, that is, there is repetition, then the digram uniqueness constraint is violated and a new rule of the grammar is created. The new right–hand side of the rule will consist only of the new digram that caused the violation. A non–terminal is then created to reference the new rule and it is used to replace the duplicate digrams.

The algorithm appends observed symbols to the end of rule S, illustrated in Figure 3 for the sequence *cannanecann*. The first column indicates each step in the process. The second column represents how much of the sequence has been inspected. The third column is the grammar for the sequence seen so far. The fourth column identifies the constraint, if any, that has been violated, including the rule or digram responsible for the violation. The final column is a list of the digrams observed so far.

A violation of digram uniqueness occurs when the last n is appended to rule S in step 6 of Figure 3, as the duplicate digram an exists. A new rule referenced by γ is created, with the right side of the rule consisting only of the digram an. The non–terminal γ replaces both occurrences of an in rule S. The insertion of γ into rule S results in three new digrams. The digrams, $c\gamma$, γn and $n\gamma$ replace the digrams ca, nn and na. The resulting grammar and digram list is shown at step 7.

A violation of the digram uniqueness constraint does not necessarily result in the creation of a new rule. If the digram happens to match exactly to the right–hand side of a rule then the digram at the end of rule S is simply replaced by the non–terminal heading the rule, and the frequency of the rule is updated. This is shown at step 11 in Figure 3 where appending n to rule S has created the duplicate digram an again. Since an exactly matches the contents of rule γ, the occurrence of an in rule S is replaced by the non–terminal γ. Only one digram is removed from the list, ca, but it is replaced by the new digram $c\gamma$.

The addition of $c\gamma$ to the digram list causes another digram uniqueness

step	string so far	resulting grammar	constraints violated	digram list
1	c	S → c		
2	ca	S → ca		ca
3	can	S → can		ca,an
4	cann	S → cann		ca,an,nn
5	canna	S → canna		ca,an,nn,na
6	cannan	S → cannan	digram uniqueness duplicate “an”	ca,an,nn,na
7		S → cγnγ γ → an		cγ,γn,an,nγ
8	cannane	S → cγnγe γ → an		cγ,γn,an,nγ γe
9	cannanec	S → cγnγec γ → an		cγ,γn,an,nγ γe,ec
10	cannaneca	S → cγnγeca γ → an		cγ,γn,an,nγ γe,ec,ca
11	cannanecan	S → cγnγecγ γ → an	digram uniqueness duplicate “an”	cγ,γn,an,nγ γe,ec
12		S → ωnγeω γ → an ω → cγ	digram uniqueness duplicate “cγ”	ωn,nγ,γe,eω an,cγ
13	cannanecann	S → $\Gamma\gamma$eΓ γ → an ω → cγ Γ → ωn	digram uniqueness duplicate “ωn”	$\Gamma\gamma$,γe,eΓ,an cγ,ωn
14		S → $\Gamma\gamma$eΓ γ → an Γ → cγn	rule utility Rule “ω”	$\Gamma\gamma$,γe,eΓ,an cγ,γn

Figure 3: An example of applying SEQUITUR to the sequence *cannanecannane*

Sequence	Grammar Option 1	Grammar Option 2
aaa	S → ω a	S → a ω
	ω → a a	ω → aa

Figure 4: Two options to choose from for the sequence *aaa*

violation. In Figure 3, step 12 illustrates the new rule ω being created, with the digram $c\gamma$ being the only contents. Both occurrences of the digram are replaced by the non–terminal heading the new rule. The digrams, ωn and $e\omega$, replace the digrams, γn and ec, in the digram list.

The final violation of digram uniqueness in Figure 3 occurs in step 13. Appending n to rule S causes the digram ωn to be duplicated. The new rule Γ is created to remove the duplicate digram and the old digrams, $n\gamma$ and $e\omega$, are replaced by $\Gamma\gamma$ and $e\Gamma$.

There is only one other case where duplicate digrams do not give rise to new rules, that is, when they overlap. The sequence *aaa* contains a duplicate digram *aa*, but a new rule is not created because the same *a* is part of both digrams.

The new rule to create is unclear and requires that SEQUITUR choose. Figure 4 illustrates the two possible options to choose from when considering creating a rule for the sequence *aaa*.

A simple count of symbols shows that both representations require an extra non–terminal even though the three *a*s still remain. This situation causes the grammar to expand rather than compress, ultimately going against the aims that the algorithm set out to achieve. It is for this reason only that overlapping digrams remain the same.

4.4 Rule Utility

Rule utility ensures that rules are used more than once allowing larger rules to be created. Every rule in the grammar has a counter associated with it to record the number of times it has been referenced throughout the hierarchy. This counter is updated whenever the non-terminal heading the rule is added or removed from the grammar.

If a non-terminal is removed from the grammar the rule frequency reduces, whereas if one is added, then the frequency increases. After checking for digram uniqueness, once a symbol has been added to rule S, the rule frequencies are checked. Any rule that has a frequency count less than two is removed because it violates the second constraint. The contents of the rule to be removed replace the single non-terminal that referred to it. Until

```
1.  Append each new symbol inspected to rule S.

2.  If(duplicate digram) then
        if(original digram is complete rule) then
            replace occurrences with non–terminal heading the rule
        else
            create a new rule and replace occurrences
            with non-terminal heading the new rule

3.  if(rule count $< 1$) then
            remove rule, replacing the non–terminal with
            the contents of the rule
```

Figure 5: The SEQUITUR algorithm

this time all of the rules consisted only of a digram, it is the removal of rules that create the large phrases of repetition.

Step 14 in Figure 3 demonstrates the removal of one of the non–terminals. The creation of rule Γ, in step 13, removed all references to the ω rule except one, which is found in the contents of rule Γ. This violates the rule utility constraint, as the rule ω is not referred to at least twice and is therefore removed from the grammar. The contents of the rule to be removed, in this example $c\gamma$, replace the only occurrence of ω. The result of enforcing rule utility is that the contents of Γ now become $c\gamma n$.

Since rule S refers to the initial sequence, it is not part of the dictionary and therefore the model is the rules of the grammar, excluding rule S. Rule S contains the dictionary references and therefore is encoded during the coding phase of compression.

4.5 Implementation of SEQUITUR

The SEQUITUR algorithm is practical and operates by ensuring that the two constraints, digram uniqueness and rule utility, are enforced. The simple algorithm is shown in Figure 5.

Nevill-Manning [10] describes one implementation of the SEQUITUR algorithm, including the data structures used to store and manipulate the grammar during creation. We mentioned briefly in Section 1 why SEQUITUR is not practical for database compression, since it uses significant in–memory

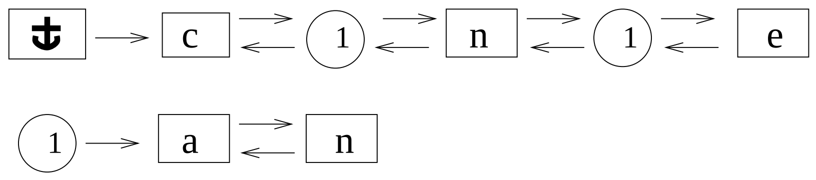

Figure 6: Linked list with the anchor character identifying rule S

resources; SEQUITUR uses large amounts of main memory to ensure independent decodeability of the data. It is therefore necessary to store the entire grammar, including the whole of rule S, in memory whilst new symbols are appended. The amount of memory SEQUITUR requires is directly related to the size of the input data. In the context of databases, the amount of main memory required is typically five to seven times the size of the database, making it impractical to compress.

SEQUITUR allocates space for a new symbol in memory before it appends the symbol to rule S. Figure 6 shows one implementation that stores the grammar as doubly-linked lists. A doubly-linked list provides an easy way to maintain the rules, since the length of them is constantly changing all the time. This is the reason why doubly-linked lists are chosen as the data structure for SEQUITUR instead of ordinary linked lists. Doubly-linked lists also allow easy reference to digrams that include both the current and previous symbols, and each rule in the grammar has its own list. An overhead is required by the linked list to maintain a reference to the first symbol, which is shown in Figure 6 by the circles. The anchor character in the figure represents the overhead required by rule S.

Additional memory, other than memory for the rules, is required by SEQUITUR for overheads such as space to store each unique digram; to assist in checking for digram uniqueness. An effective implementation that stores these digram references is a hash-table. Each slot in the hash–table has space to store a single pointer to the digram. This provides an easy way of searching for duplicate digrams by hashing on the digrams contents to a slot. If a collision occurs when hashing to the slot, one of two events could occur. If the digram in the slot is a duplicate then the algorithm signifies that one of the constraints has been violated and replaces the digram with the appropriate rule. The alternative event that may occur is that the slot is currently occupied by another digram.

These types of collisions are overcome by implementing some method that will rehash the digram to an empty slot. Nevill–Manning chose to implement this method as a double–hashing routine that incrementally jumps a certain number of slots until an empty one is found. Our experiments found that this approach can cause delays in the processing time, especially when the slot usage is high. Alternative rehashing schemes such as slot-chaining or linear hashing may remove the delays, but we have not experimented with this. The major factor limiting SEQUITUR from use as a database compression scheme, on general-purpose hardware, is the significant use of main memory. We propose modifications to bound the size of main memory used.

5 Bounding model size

In this Section, we address the limitations of SEQUITUR, as a compression scheme for databases by proposing two new approaches As highlighted in the previous section, the major modification required to SEQUITUR is to bound the amount of main memory it uses when building the grammar.

The first approach, *modelcombining*, divides the database into parts large enough to be used by SEQUITUR without exceeding a threshold of main memory. A model for each division is constructed and combined together with the other models to produce a single model that the encoder uses to encode the database. The second approach, *sequ–snapshot*, restricts the amount of rule S that can be retained in main memory. The section of rule S that has been seen by SEQUITUR that is no longer stored in memory becomes static and cannot be altered to reflect any changes in the grammar resulting from the creation of new rules. We describe the two novel approaches in detail.

5.1 Model Combining

One approach that we propose to address SEQUITUR'S memory limitations is model combining. Model combining addresses SEQUITUR'S memory problem by reducing the size of the input data. The model combining approach involves two seperate activities: model creation and model merging.

Model creation is the division of input into separate segments, so that a model can be built for each segment. The other activity, model merging, applies an algorithm that merges two rule sets to create a single model. A final pass over the entire input data is required after model merging to encode the data with the new model, maintaining independent decodeability. Figure 7 shows how both activities are used in the model combining approach.

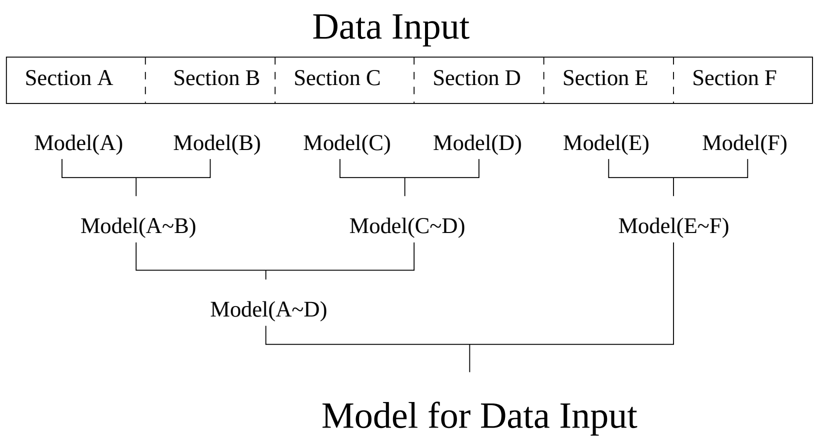

Figure 7: The Model Combining Process

5.1.1 Model creation

Model creation is the first activity of the model combining approach that limits main memory usage. It involves reducing the size of the input data, since the size of the input data is directly related to the upper bound of main memory used by SEQUITUR.

Reduction in the size of the data is achieved by dividing it into separate segments. The segments may be a collection of records, a single or multiple disk block, or input data of a specific size. The single restriction that must apply is that the size of the segment is less than the threshold that will cause SEQUITUR to exceed its main memory limit. Consider a main memory limit of 10 Mb. Assuming that the upper bound of SEQUITUR'S memory usage is 7 times the size of the data, then the size of the maximum input data that can be used is approximately 1.4 Mb. For a 5 Mb file, the smallest number of divisions that can be obtained is 4, with each segment having less than 1.4 Mb of data.

After the data is divided into appropriate segments, each division is passed as input to SEQUITUR. A segment from the original data is considered independent of the other segments as it is inspected by SEQUITUR. A unique grammar for the section is produced and stored so that, once all models have been built, the model merging process can reference the grammar. It is not necessary to store rule S with the rest of the grammar at this point, as it will no longer be appropriate once the new grammar is built by merging all models. Model merging commences once a model has been built for each division of the original input data.

5.1.2 Model merging

Model merging is the second activity of the model combining process that involves merging two existing grammars together to form a new one. An algorithm is necessary to identify similar phrase structure between two rule sets, in order to merge them together.

We propose a new algorithm, *sequ–merge*, that merges two rule sets together to produce a new grammar. Our proposed algorithm identifies similar phrase structure between two rule sets by applying SEQUITUR'S constraints to both the rules sets.

Every rule is considered as a seperate sequence so that a digram is not formed between the last symbol of one rule and the first symbol of the next rule. The disadvantage of not considering rules seperately is that rules can be created for digrams that do not exist. For example, if we were to consider rule set 1 in Figure 8, the digrams that should be checked are : "bc","cd", "ef" and "an". When the rules are sequentially inspected by SEQUITUR, the additional digrams, that do not exist, are added to the list. The extra digrams that do not exist that would be added to the digram list are: "de" and "fa".

A grammar consisting of new rules, referencing duplicate digrams across the two rule sets, is built as each rule is inspected by SEQUITUR. Figure 8 shows how rule sets are merged together to produce a new grammar. The digrams within each rule are added to the digram list. The first three steps add the digrams of rule 1,2 and 3 to the digram list without any duplicates. This is expected since the grammar should already contain no duplicate digrams. When the digram from rule 4 is appended it causes the digram uniqueness constraint to be violated. The new rule, 6, is created. Likewise, when the digram from rule 5 is added, a duplicate digram is detected and a new rule, 7, is created. The resulting combined grammar contains both rule 6 and 7.

The process of model merging continues until all of the initial rule sets have been combined with each other. When more than two rule sets exist the model merging process needs to be applied until a final model is reached; in a similar way as in Figure 7. Once a final model is reached, all of the grammar rules are loaded into memory, and the original piece of data is input to SEQUITUR. The new alteration to SEQUITUR is that no new rules can be created or deleted, so the original data is transformed into a compressed sequence that has the final model as its rule set.

Rule Set 1	Rule Set 2	Action	Digram List	New grammar
1 →bcd 2 →ef 3 →an	4 →ef 5 →bc	Append Rule 1	bc,cd	
		Append Rule 2	bc,cd,ef	
		Append rule 3	bc,cd,ef,an	
		Append rule 4	bc,cd,ef,an	6 →ef
		Append rule 5	bc,cd,ef,an	6 →ef 7 →bc

Figure 8: An example of merging two rule sets to create a new grammar

5.2 Sequ–Snapshot

We propose a second novel, successful approach, that improves SEQUITUR'S memory usage, *sequ–snapshot*. The sequ-snapshot algorithm addresses the memory problem of SEQUITUR by only applying the grammar constraints to a defined buffer of input. The buffer provides a window for rule S where grammar changes can be updated. Sections of rule S that are outside the buffer remain static and as a result force some underused rules to remain as part of the grammar. The grammar produced by sequ-snapshot is similar to the grammar produced by SEQUITUR, however parts of rule S remaining static alter the constraints that SEQUITUR uses to produce the grammar.

A description of the new sequ–snapshot algorithm is presented in Figure 9.

Initially, the algorithm begins, as in SEQUITUR, by appending each new symbol to rule S. While the buffer is not yet full, the amended constraints of sequ–snapshot do not alter the grammar. The amended constraints affect the grammar when a new symbol appended to rule S exceeds the buffer limit. The buffer moves sequentially through the data, one symbol at a time, in a single direction to include the new appended symbols. Symbols that are outside the buffer remain static and cannot be altered to reflect any new changes that occur within the buffer. Moreover, some alterations to the grammar are prohibited because of the static symbols.

The static symbols outside the buffer cause the alterations to the constraints of SEQUITUR. For example, digram uniqueness is affected by static symbols when the duplicate digram is outside the buffer. The creation of new rules still applies, however, the compression benefits are not fully achieved as the non-terminal heading the new rule can not replace the static digram.

```
1.  Append each new symbol inspected to rule S.

2.  If the number of symbols in rule S exceed the size of
    the window then slide the window to the right by one symbol

3.  If(duplicate digram) then
        if(original digram is complete rule) then
            replace occurrences with non–terminal heading the rule
        else
            create a new rule
            if(original occurrence is within window)
                replace all occurrences with non-terminal heading
                the new rule
            else
                only replace the last digram with the non-terminal
                heading the new rule

4.  if(rule count < 1) then
            if(non–terminal referencing rule in window)
                remove rule, replacing the non–terminal with
                the contents of the rule
```

Figure 9: The SEQU-SNAPSHOT algorithm

In Figure 10, the underlined symbols illustrate the size of the window buffer. It shows a duplicate digram that forces a new rule to be created. The occurrence of the digram *ca* in the buffer is the only position where the digram is replaced by the non-terminal γ. The occurrence of *ca* at the beginning of rule S is not altered to incorporate the new rule.

The static symbols also affect the "rule utility" constraint. To enforce rule utility, under-used rules need to be removed from the grammar, however, when the reference to the rule the result of replacing only one duplicate digram, as in the above example, the rule must remain in the grammar. Removal of the rule from the grammar would cause the scheme to lose compression performance by not identifying duplication of digrams that are separated by a distance greater than the span of the window. As mentioned previously, the constraints of SEQUITUR remain the same for the symbols

string so far	resulting grammar	constraints violated	digram list
c	$S \rightarrow \underline{c}$		
ca	$S \rightarrow \underline{ca}$		ca
can	$S \rightarrow \underline{can}$		ca,an
cann	$S \rightarrow c\underline{ann}$		ca,an,nn
canna	$S \rightarrow ca\underline{nna}$		ca,an,nn,na
cannae	$S \rightarrow can\underline{nae}$		ca,an,nn,na,ae
cannaec	$S \rightarrow cann\underline{aec}$		ca,an,nn,na,ae,ec
cannaeca	$S \rightarrow canna\underline{eca}$	digram uniqueness	ca,an,nn
		digram "ca"	na,ae,ec
	$S \rightarrow canna\underline{e\gamma}$		eγ,ca,an,nn
	$\gamma \rightarrow ca$	digram "ca"	na,ne

Figure 10: The sequ–snapshot buffer

within the buffer, however when considering the entire sequence being coded, the two constraints have slight amendments. The amended constraints for the context–free grammar of SEQU–SNAPSHOT are:

1. Digram uniqueness: No single digram may appear in the grammar more than once. If the initial digram exists outside the window then it is not replaced by the non-terminal symbol. Only the second occurrence of the digram is replaced by the non-terminal.

2. Rule utility: A rule of the grammar must be referred to at least twice, unless the only non–terminal referencing the rule to be removed exists outside the window, in which case, it must remain in the grammar.

6 Test data

The test collection that we used to compare compression performance of the new approaches, model combining and sequ–snapshot, consist of a set of databases having different data–types. Two collections having the same data–type, but of different size, were also part of the test collection.

The collection consists of two text files, trec1 and trec2; two numeric data files, HRL100 and HRL101; a file containing the vocabulary for a text file, voc; and two files containing genetic material, referred to as gbmam, and hew. We chose this collection in order to vary the data-types so we could test for compression as a general-purpose scheme, rather than a special–purpose scheme for a particular data type.

The two text files, trec1 and trec2, were taken from the TREC data [7]. The TREC data consists of collections of text documents. Trec1 is a 28.6 Mb segment extracted from a collection belonging to TREC. It contains 7640 full documents, and another partial document to make up 28.6 Mb of data. Similarly, trec2 is a 2.9 Mb segment of text extracted from a collection to the collection trec1 was extracted from. The trec2 file contains 964 full documents, and one partial document to complete the 2.9 Mb of data.

The second two files from the collection, HRL100 and HRL101, contain meteorological data. The data is collections of weather readings taken at different time intervals throughout the day, over a given period. The readings are stored as numbers and are highly repetitive. A small collection of the weather data, HRL100, is 2.1 Mb in size. The other file, HRL101, is somewhat larger, being 34.1 Mb in size.

The vocabulary data file, voc, is taken from the vocabulary generated for a trec data file. The file contains 5268678 vocabulary words. The total size of voc is 20.2 Mb.

The last two files that make up the test collection contain genetic sequences taken from the nucleotide database, GenBank [3]. The first file, hew, contains amino–acids that are often referred to as Protein Identification Resource or PIR [5]. The data is well-classified amino–acid sequences with known relationships and redundancy. Hew contains the first 4.77 Mb of the amino–acids from the PIR database. The second file gbmam, is a 4.77 Mb segment of GenBank. It contains the mammalian, non primate, section of DNA.

7 Results

7.1 Compression using SEQUITUR

Initially we investigated the compression performance of SEQUITUR on the test collection to determine how effective SEQUITUR works as a compression scheme. We compared the compression results from SEQUITUR with the results from four other compressions schemes.

The four compression schemes that we used to compare SEQUITURS compression performance against include two well-known schemes *gzip*, and *compress* [1]. The other two compression schemes are different forms of Huffman coding. *Huffword* [1] is a word-based Huffman coding scheme whereas *pack* [20] is a character-based Huffman coding scheme.

Schemes	HRL100 kBytes	HRL101 kBytes	hew kBytes	gbmam kBytes	trec1 kBytes	trec2 kBytes	voc kBytes
original	2170	34959	4883	4883	29297	2930	20648
SEQUITUR	184	3486	1447	2327	10348	1329	9979
gzip	170	3656	1204	2119	9728	1101	7864
compress	192	3908	1313	2315	12025	1277	9101
huffword	434	7551	3511	3511	9346	1046	10138
pack	697	11999	1370	1370	18494	1811	11915

Table 1: Compressed file sizes for different compression schemes

The results for the five compression schemes are shown in Table 1. The measurements are in terms of the size of the compressed file. Note that the results shown for the compressed files using SEQUITUR have been estimated by calculating the entropy for the main rule S and adding it with the entropy calculated for the right hand side of all the rules. A further 4 bytes per rule have been added to allow for exit codes; exit codes distinguish a rule from an encoded symbol. Our estimation for the compressed file size is quite generous, in regards to the space allocated for each rule. A Huffman coding technique, mentioned in Section 2.3, would provide a compressed file close to this value.

Results show that overall, SEQUITURS compression performance is better than *pack*. This result is expected since pack is a character-based Huffman coding scheme. SEQUITUR also outperforms $huffword$, except for both of the text files. $Huffword$ is a compression scheme designed specifically for text data which is why it achieves the best compression on text but performs poorly on the other files.

SEQUITUR achieves similar compression to *compress* for most files in the test collection, and even betters all of the schemes for the large HRL101 weather file. We attribute this result to the highly repetitive nature of the weather files. We note also that SEQUITUR performs best, in comparison to the other compression schemes, when the size of the data is large, greater than 20 Mb. We attribute this to better utilisation of the extra space set aside for exit codes. Of all of the compression schemes, *gzip* performs the best for the entire test collection, however, since *gzip* uses adaptive modelling it does not allow individual decompression of records and can not be used as a scheme for databases.

Step	Action	Rule added	Digram List	Duplicate digrams	New grammar
1	Add 1st Rule set	1 →abc	ab,bc		
2	Add 1st Rule, 2nd set	2 →ab	ab,bc	ab	1 → ab
3	Add 2nd Rule, 2nd set	3 →bc	ab,bc	bc	1 → ab 2 → bc
4	Add 2nd Rule	1 →ab	ab,bc		
5	Add 1st Rule, 1st set	1 →abc	ab,bc	ab	1 →ab
6		1 → 1c	ab,bc,1c		1 →ab

Figure 11: An example of two rule sets producing two grammars

7.2 Model Combining

We review implementation details of model combining to highlight design problems with the algorithm. Initial results from our model combining approach show that although memory usage was kept under the predefined limit, the compression performance deteriorated quite badly. The model combining approach caused large sub-sequences found in one rule set to be lost. Also, the resulting grammar produced from model combining depended upon which rule set was inspected first. This enabled two totally different grammars to be derived from the same two rule sets.

Figure 11 shows how the order the rule sets are inspected, effects the resulting grammar. Since a grammar produced by SEQUITUR can not contain any duplicate digrams, the first rule set will not contain any duplicate digrams. Therefore, no new rules will be formed and, as seen in step 1 of Figure 11, the digrams from the entire grammar are inserted into the digram list. In step 2, the digrams from the first rule of rule set 2 are checked for uniqueness. A duplicate digram *ab* is detected and a new rule created. The digrams from the last rule of rule set 2 are checked for digram uniqueness. Once again there is a duplicate digram *bc* detected and a new rule created. The resulting grammar produced by model combining consists of two rules. However, if the second rule set is inspected first as in step 4 of Figure 11, the resulting new grammar is different. Step 5 shows that when the first digram, *ab*, from the rule is checked for digram uniqueness, a new rule is created. The new digram, 1*c*, is added to the digram list in step 6 and we are left with a new grammar containing only one rule. This clearly indicates, as mentioned previously, that two different grammars can be produced from the same two rule sets, depending upon the order of inspection.

Schemes	HRL100 (%)	HRL101 (%)	hew (%)	gbmam (%)	trec1 (%)	trec2 (%)	voc (%)
original	100	100	100	100	100	100	100
SEQUITUR	8.5	10.0	29.6	24.2	35.3	34.4	48.3
sequ–snaphot (100, 000)	9.0	10.1	36.6	31.2	40.1	42.9	52.3

Table 2: Compression performance of sequ-snapshot compared to SEQUITUR

It is evident from our preliminary results that using the model combining approach we can only produce a grammar that has rules with a single digram as the contents. The formation of larger rules is restricted since they can only develop when a rule violates the rule utility constraint. As can be seen in step 6 of Figure 11, the rule utility constraint is relaxed, and since the rules are not maintained, new rules only ever have a single reference.

Our early results show that in order to maintain an acceptable compression level, a much more complicated algorithm to merge rule sets is required. We have left the research development of an effective algorithm to merge rule sets for further research.

7.3 Sequ-snapshot

In order to measure the compression performance of sequ–snapshot, we compare the compression performance against SEQUITUR. Table 2 shows the results obtained by comparing our new scheme, sequ–snapshot,with a buffer size of $100,000$ bytes, against SEQUITUR. Compression performance is presented as a percentage of the size of the original file.

Results show the sequ–snapshot algorithm, with a buffer of $100,000$, achieves compression almost equivalent to that of sequitur. On average, sequ–snapshot's compression is only 4.5% worse than SEQUITUR.

Figure 12 illustrates the impact on compression performance of varying the buffer size for the file, hew. The sequ–snapshot algorithm is equivalent to SEQUITUR when the buffer size is greater than the size of the input data; the right-most values on the graph. The graph shows the entropy achieved for the model and rule S, represented by Stringlen, as well as the total which is simply the addition of both.

The reduction of the grammar constraints to a defined buffer space has little impact on the compression performance of sequ–snapshot compared to SEQUITUR. We attribute this behaviour to the constraint used by sequ–

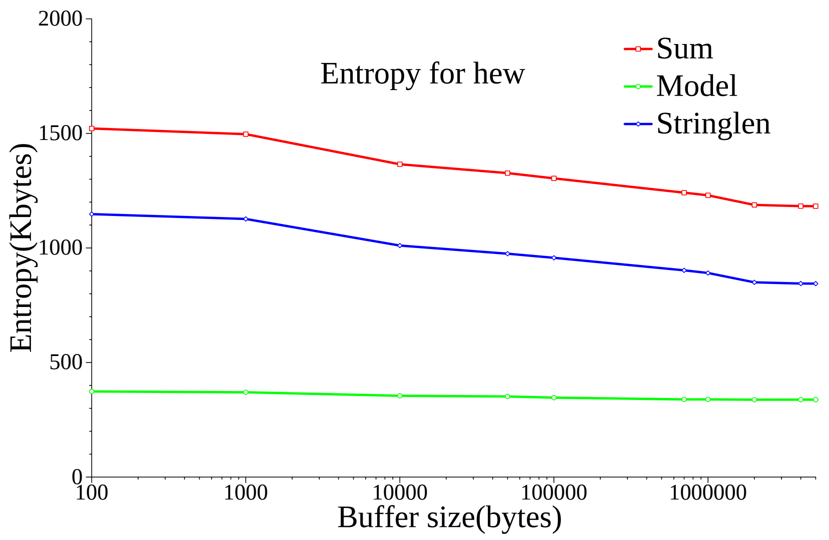

Figure 12: Compression for sequ–snapshot for various buffer sizes

snapshot, that allows a new rule to be created when there is only one reference to it. Sequ-snapshot creates a new rule for the duplicate digram but only replaces the digram in the buffer with the non–terminal heading the rule. However, the resulting compression performance is affected by all the single-referenced rules in the grammar.

The most striking result for the entire test collection is that in the worst case, for a slot-size 50,000 times greater, the compression performance of SEQUITUR is only 8.5% better than sequ-snapshot. That is, reducing slot size to 0.002% of the file size has only a limited impact on compression performance. The reason why there is not a significant reduction in compression performance is that sequ–snapshot still identifies repetitive subsequences, even when they are separated by a length greater than the size of the buffer. The only restriction is that sequ–snapshot can not identify subsequences whose total length exceeds the buffer. Overall, we conclude that sequ-snapshot provides similar compression to SEQUITUR, whilst achieving a massive reduction in memory usage.

7.4 Model size during runtime

Results for the memory size used by the model during runtime, for various file sizes of trec2, are presented in Figure 13.

The results suggest that the model increases almost linearly as the file size increases. This poses a small problem to our effort of achieving good compression with a limited amount of memory. As the graph illustrates,

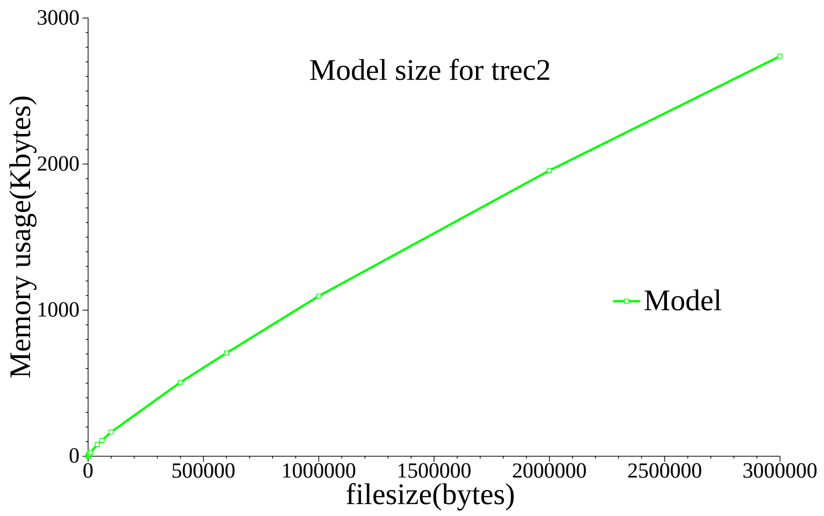

Figure 13: Increase in model size for various sizes of trec2

the model continues to grow even though we are limiting the size of the buffer for the main rule S. ultimately consuming more memory resources as more data is inspected. A large database would require a significant amount of memory to store the model. These results indicate that although sequ–snapshot reduces memory usage compared to SEQUITUR, it does not bound the amount of memory it requires. Further research is needed to limit not only the memory used to store rule S but also to limit the size of the model.

We also note that the size of main memory required to store the grammar used for decoding the compressed data will be much smaller than the memory required to generate the grammar. This is due to the memory saved by not having to store the grammar as doubly-linked lists.

8 Conclusions

We have shown that SEQUITUR compresses data almost as well as the well-known compression schemes. SEQUITUR consistently achieved good compression with respect to the other schemes, verifying that it is capable of general–purpose compression. Since SEQUITUR is a semi–static modelling technique it is a good compression scheme for databases. Another benefit of SEQUITUR for database compression is that the model is easily extended, so new rules can be added to the grammar at any stage.

However, the design of SEQUITUR consumes excessive in-memory resources making it impractical for large database applications. We have proposed a semi–static modelling approach, an extension to SEQUITUR, that has been proven to be practical for database applications. Our results have shown that, on average, sequ–snapshot achieves compression within 6% of the compression achieved by SEQUITUR, whilst using considerably less memory.

It was our original intention to bound the model size to ensure efficient use of memory. Our results show that although we do restrict memory usage, in comparison to SEQUITUR, the model size continues to grow unbounded. Extremely large databases would causes sequ–snapshot to become impractical as a general–purpose compression scheme for databases.

Several extensions to our algorithm are possible to achieve even greater compression performance. First, we believe that a second "clean–up" pass over the entire data will further reduce the string size by replacing digrams with non–terminals. After the completion of the first pass of sequ–snapshot, the resulting grammar can consist of rules with only a single reference. These rules are created when one of the digrams from a duplicate digram occurs outside of sequ-snapshot's buffer. A new rule is created but the digram is not replaced by the non-terminal. A second pass over rule S, keeping the grammar fixed, should identify the duplicate digrams and replace them with the correct non-terminals heading the single rules.

Second, to calculate the compressed file size for our results we simply used the entropy for the symbols in the grammar. Nevill–Manning [12] describes an encoding method on the dictionary that improves the compression result for SEQUITUR. The method that compresses the grammar involves front coding the grammar followed by using PPM. Front coding [6] involves sorting the grammar. Whenever the prefix of a rule matches the prefix of a preceeding rule, the prefix is replaced by its length. An example of front coding for the word "Adam" would be 3m if it were preceeded by "Ada", since the prefix is similar over 3 symbols. PPM [19] is a compression method that uses arithmetic coding. Applying these improvements to sequ–snapshot could improve the compression result.

Last, we believe further work can amend the sequ–snapshot algorithm to stop the growth of the model when it has reached the memory threshold. At this point the grammar can no longer change and the process continues, compressing the remainder of the data with the static grammar.

We have investigated the advantages of a semi–static modelling technique for compression of databases. We believe that the algorithm used by SEQUITUR does not actually require two separate passes of the data. Rather it uses one and a bit passes [9] to compress the data, since it does not reach the end and then return to the beginning again.

However, we believe that future work investigating modelling techniques that are neither semi–static nor adaptive, but somewhere in between, may result in an effective general–purpose compression scheme. A typical scheme may incorporate model combining, so a general model can be built as well as adaptively encode extra rules within the compressed data that are specific that section of the database.

9 Acknowledgements

I would like to thank Justin Zobel for his supervision and guidance throughout the year, and especially for his ideas and constructive criticisms. In addition, I would also like to thank Hugh Williams for the considerable amount of time he spent proof–reading this paper. Thank you also to Craig Nevill–Manning for his generous offer to supply the sequitur code.
To my friends, I thank them for their support throughout my Honours year, in particular Fadia Hatem.

References

[1] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[2] T.C. Bell, A. Moffat, C.G. Nevill-Manning, I.H. Witten, and J. Zobel. Data compression in full-text retrieval systems. 44(9):508–531, October 1993.

[3] D. Benson, D.J. Lipman, and J. Ostell. GenBank. *Nucleic Acids Research*, 21(13):2963–2965, 1993.

[4] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, Redwood City, 1994.

[5] D. George, W. Barker, H. Mewes, F. Pfeiffer, and A. Tsugita. The PIR-international protein sequence database. *Nucleic Acids Research*, 24:17–20, 1996.

[6] D. Gottlieb, S.A. Lehot, P.G.H., and H.S. Rabinowitz. A classification of compression methods and their usefulness for a large data processing center. *Proc. National Computer Conference*, pages 453–458, 1975.

[7] D.K. Harman. National institute of standards and technology special publication. *Proc. Text Retrieval Conference (TREC)*, 1992.

[8] D.A. Lelewer and D.S. Hirschberg. Data compression. *Computing Surveys*, 19(3):261–296, September 1987.

[9] A. Moffat, J. Zobel, and N. Sharman. Text compression for dynamic document databases. *IEEE Transactions on Knowledge and Data Engineering*, 9:1–38, November 1995.

[10] C.G. Nevill-Manning. *Inferring sequential structure.* PhD thesis, University of Waikato, May 1996.

[11] C.G. Nevill-Manning and I.H. Witten. Detecting sequential structure. *Proc. Workshop on Programming by Demonstration*, July 1995.

[12] C.G. Nevill-Manning and I.H. Witten. Compression and explanation using hierarchical grammars. *Computer Journal*, 1997.

[13] C.G. Nevill-Manning and I.H. Witten. Linear–time, incremental hierarchy inference for compression. *Proc. Data Compression Conference*, 1997.

[14] C.G. Nevill-Manning, I.H. Witten, and Jr. D.R. Olsen. Compressing semi–structured text using hierarchical phrase identifications. *Proc. IEEE Data Compression Conference*, pages 63–72, 1996.

[15] C.G. Nevill-Manning, I.H. Witten, and D. L. Maulsby. Compression by induction of hierarchical grammars. *Proc. IEEE Data Compression Conference*, pages 244–253, 1994.

[16] J. Rissanen and G.G. Langdon. Universal modeling and coding. *IEEE Transactions on Information Theory*, IT-27(1):12–23, January 1981.

[17] C.E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423,623–656, 1948.

[18] T.A. Sudkamp. *Languages and Machines : An Introduction to the Theory of Computer Science.* Addison–Wesley Publishing Company, Inc, Reading, Massachusetts, 1994.

[19] W.J. Teahen and J.G. Cleary. The entropy of english using ppm-based models. *Proc. Data Compression Conference*, pages 52–63, 1996.

[20] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and indexing documents and images.* Van Nostrand Reinhold, New York, 1994.

[21] I.H. Witten, R. Neal, and J.G. Cleary. Arithmetic coding for data compression. *Communications of the Association for Computing Machinery*, 30(6):520–540b, 1987.

[22] J. Zobel and A. Moffat. Adding compression to a full–text retrieval system. *Software–Practice and Experience*, 25:891–903, August 1995.

[23] J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. 25(8):891–903, 1995.