Lecture 6 — Collect, Sets and Tables

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 15, 2011

Today:

- Collect, and example of Google map-reduce
- Sets
- Tables, and example of web searching

1 Collect

In many applications it is useful to collect all items that share a common key. For example we might want to collect students by course, documents by word, or sales by date. More specifically let's say we had a sequence of pairs each consisting of a student's name and a course they are taking, such as

```
D = <("jack sprat", "15-210"),
     ("jack sprat", "15-213"),
     ("mary contrary", "15-210"),
     ("mary contrary", "15-251"),
     ("mary contrary", "15-213"),
     ("peter piper", "15-150"),
     ("peter piper", "15-251"),
     ... >
```

and we want to collect all entries by course number so we have a list of everyone taking each course. Collecting values together based on a key is very common in processing databases, and in relational database languages such as SQL it is referred to as "Group by". More generally it has many applications and furthermore it is naturally parallel.

We will use the function `collect` for this purpose, and it is part of the sequence library. Its interface is:

$$\mathit{collect} : (\alpha \times \alpha \to \mathit{order}) \to (\alpha \times \beta)\ \mathit{seq} \to (\alpha \times \beta\ \mathit{seq})\ \mathit{seq}$$

The first argument is a function for comparing keys of type α, and must define a total order over the keys. The second argument is a sequence of key-value pairs. The *collect* function collects all values that share the same key together into a sequence. If we wanted to collect the entries of `D` given above by course number we could do the following:

```
fun swap(x,y) = (y,x)
val rosters = collect String.compare (map swap D)
```

This would give something like:

```
rosters = <("15-150", < "peter piper", ... >),
           ("15-210", < "jack sprat", "mary contrary", ... >)
           ("15-213", < "jack sprat", ... >)
           ("15-251", < "mary contrary", "peter piper", ...>)
           ... >
```

The *swap* is used to put the course number in the first position in the tuple. It is often the case that the key needs to be extracted before applying collect.

You might find `collect` useful in your current assignment.

Collect can be implemented by sorting the keys based on the given comparison function, and then partitioning the resulting sequence. In particular, the sort will move all the equal keys so they are adjacent. A partition function can then identify the positions where the keys change values, and pull out all pairs between each change. Doing this partitioning is relatively easy and very similar to the `fields` function described at the end of the last class. The dominant cost of *collect* is therefore the cost of the sort. Assuming the comparison has complexity bounded above by W_c work and S_c span then the costs of `collect` are $O(W_c n \log n)$ work and $O(S_c \log^2 n)$ span. It is also possible to implement a version of collect that runs in linear work using hashing. But hashing would require that a hash function is also supplied and would not return the keys in sorted order. Later in the lecture we discuss tables which also have a `collect` function. However tables are specialized to the key type and therefore neither a comparison nor a hash function need to be passed as arguments.

1.1 Using Collect in Map Reduce

Some of you have probably heard of the map-reduce paradigm first developed by Google for programming certain data intensive parallel tasks. It is now widely used within Google as well as by many others to process large data sets on large clusters of machines—sometimes up to tens of thousands of machines in large data centers. The map-reduce paradigm is often used to analyze various statistical data over very large collections of documents, or over large log files that track the activity at web sites. Outside Google the most widely used implementation is the Apache Hadoop implementation, which has a free license (you can install it at home). The paradigm is different from the `mapReduce` function you used in 15-150 which just involved a map then a reduce.[1] The map-reduce paradigm actually involves a map followed by a collect followed by a bunch of reduces, and therefore might be better called the map-collect-reduces.But we are stuck with the standard terminology here.

The map-reduce paradigm processes a collection of documents based on *mapF* and *reduceF* functions supplied by the user. The *mapF* function must take a document as input and generate a sequence of key-value pairs as output. This function is mapped over all the documents. All key-value pairs across all documents are then collected based on the key. Finally the *reduceF* function is applied to each of the keys along with its sequence of associated values to reduce to a single value.

In ML the types for map and reduce functions are the following:

$$\begin{aligned} \textit{mapF}: &\quad (\textit{document} \to (\textit{key} \times \alpha)\ \textit{seq}) \\ \textit{reduceF}: &\quad (\textit{key} \times (\alpha\ \textit{seq}) \to \beta) \end{aligned}$$

[1] However, there was a question on the 15-150 final about the map-reduce paradigm.

In most implementations of map-reduce the document is a string (just the contents of a file) and the key is also a string. Typically the α and β types are limited to certain types. Also, in most implementations both the *mapF* and *reduceF* functions are sequential functions. Parallelism comes about since the mapF function is mapped over the documents in parallel, and the reduceF function is mapped over the keys with associated values in parallel.

In ML map reduce can be implemented as follows

```
fun mapCollectReduce mapF reduceF S =
  let
    val pairs = flatten (map mapF S)
    val groups = collect String.compare pairs
  in
    map reduceF groups
  end
```

The function `flatten` simply flattens a nested sequence into a flat sequence, e.g.:

```
  flatten < < a, b, c>, < d, e> >
= < a, b, c, d, e >
```

We now consider an example application of the paradigm. Suppose we have a collection of documents, and we want to know how often every word appears across all documents. This can be done with the following *mapF* and *reduceF* functions.

```
fun mapF D = map (fn w => (w,1)) (tokens spaceF D)
fun reduceF(w,s) = (w, reduce op+ 0 s)
```

Now we can apply `mapCollectReduce` to generate a `countWords` function, and apply this to an example case.

```
val countWords = mapCollectReduce mapF reduceF

countWords < "this is a document",
             "this is is another document",
             "a last document" >

= < ("a", 2), ("another", 1), ("document" 3), ("is", 3),
    ("last", 1), ("this", 2) >
```

2 Sets

Sets play an important role in mathematics and often needed in the implementation of various algorithms. It is therefore useful to have an abstract data type that supports operations on sets. Indeed most programming languages either support sets directly (e.g., python) or have libraries that support them (e.g., in the C STL library and Java collections framework). Such languages sometimes have more than one implementation

of sets. Java, for example, has sets bases on hash tables and balanced trees. We should note, however, that the set interface in different libraries and languages differ in subtle ways. Consequently, when using one of these interfaces you should always read the documentation carefully.

We now define an abstract data type for sets. The definition follows the mathematical definition of sets from set theory and is purely functional. In particular, when updating a set (e.g. with insert or delete) it returns a new set rather than modifying the old set.

Definition 2.1. For a universe of elements $\mathbb{U}$ (e.g. the integers or strings), the *Set* abstract data type is a type $\mathbb{S}$ representing the powerset of $\mathbb{U}$ (i.e., all subsets of $\mathbb{U}$) along with the following functions:

`empty`	:	$\mathbb{S}$	=	$\emptyset$
`size`(S)	:	$\mathbb{S} \to \mathbb{Z}^*$	=	$\lvert S \rvert$
`singleton`(e)	:	$\mathbb{U} \to \mathbb{S}$	=	$\{e\}$
`filter`(f, S)	:	$((\mathbb{U} \to \mathbb{B}) \times \mathbb{S}) \to \mathbb{S}$	=	$\{s \in S \mid f(s)\}$
`find`(S, e)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{B}$	=	$\lvert\{s \in S \mid s = e\}\rvert = 1$
`insert`(S, e)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S}$	=	$S \cup \{e\}$
`delete`(S, e)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S}$	=	$S \setminus \{e\}$
`intersection`(S_1, S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \cap S_2$
`union`(S_1, S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \cup S_2$
`difference`(S_1, S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \setminus S_2$

where $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ and $\mathbb{Z}^*$ are the non-negative integers.

This definition is written to be generic and not specific to SML. In the SML Set library we supply, the type $\mathbb{S}$ is called `set` and the type $\mathbb{U}$ is called `key`, the arguments are not necessarily in the same order, and some of the functions are curried. For example the interface for find is `find` : `set` $\to$ `key` $\to$ `set`. Please refer to the documents for details. In the pseudocode we will give in class and in the notes we will use standard set notation as in the right hand column of the table above.

Note that the interface does not contain a `map` function. A `map` function does not make sense in the context of a set, or at least if we interpret map to take in a collection, apply some function to each element and return a collection of the same structure. Consider a function that always returns zero. Mapping this over a set would return a bunch of zeros, which would then be collapsed into a set of size one. Therefore such a map would reduce the set of arbitrary size to a singleton, which doesn't match the map paradigm.

In addition to the semantic interface, we need a cost model. The most common efficient ways to implement sets are either using hashing or balanced trees. These have various tradeoffs in cost, but dealing with hash tables in a functional setting where data needs to be persistent is somewhat complicated. Therefore here we will specify a cost model based on a balanced-tree implementation. For now, we won't describe the implementation in detail, but will later in the course. Roughly speaking, however, the idea is to use a comparison function to keep the elements in sorted order in a balanced tree. Since this requires comparisons inside the various set operations, the cost of these comparisons affects the work and span. In the table below we assume that the work and span of a comparison on the elements is bounded by C_w and C_s respectively.

	Work	*Span*
`size`(S) `singleton`(e)	$O(1)$	$O(1)$
`filter`(f, S)	$O\Big(\sum_{e \in S} W(f(e))\Big)$	$O\Big(\log \lvert S\rvert + \max_{e \in S} S(f(e))\Big)$
`find`(S, e) `insert`(S, e) `delete`(S, e)	$O(C_w \log \lvert S\rvert)$	$O(C_s \log \lvert S\rvert)$
`intersection`(S_1, S_2) `union`(S_1, S_2) `difference`(S_1, S_2)	$O\big(C_w m \log(\frac{n+m}{m})\big)$	$O\big(C_s \log(n+m)\big)$

where $n = \max(|S_1|, |S_2|)$ and $m = \min(|S_1|, |S_2|)$. The work for `intersection`, `union`, and `difference` might seem a bit funky, but the bounds turn out to be both the theoretical upper and lower bounds for any comparison-based implementation of sets. We will get to this later, but for now you should observe that in the special case that the two input lengths are within a constant, the work is simply $O(n)$. This bound corresponds to the cost of merging two approximately equal length sequences, which is effectively what these operations have to do. You should also observe that in the case that one of the sets is a singleton, then the work is $O(\log n)$.

On inspection, the three functions `intersection`, `union`, and `difference` have a certain symmetry with the functions `find`, `insert`, and `delete`, respectively. In particular `intersection` can be viewed as a version of find where we are searching for multiple elements instead of one. Similarly `union` can be viewed as a version of `insert` that inserts multiple elements, and `difference` as a version of `delete` that deletes multiple elements. In fact it is easy to implement `find`, `insert`, and `delete` in terms of the others.

$$\begin{aligned} \texttt{find}(S, e) &= \texttt{size}(\texttt{intersection}(S, \texttt{singleton}(e))) = 1 \\ \texttt{insert}(S, e) &= \texttt{union}(S, \texttt{singleton}(e)) \\ \texttt{delete}(S, e) &= \texttt{difference}(S, \texttt{singleton}(e)) \end{aligned}$$

Since `intersection`, `union`, and `difference` can operate on multiple elements they are well suited for parallelism, while `find`, `insert`, and `delete` have no parallelism. Consequently, in designing parallel algorithms it is good to think about how to use `intersection`, `union`, and `difference` instead of `find`, `insert`, and `delete` if possible. For example, one way to convert a sequence to a set would be to insert the elements one by one, which can be coded as

```
fun fromSeq(S) = Seq.iter (fn (S',e) => Set.insert e S') Set.empty S
```

However, the above is sequential. To do it in parallel we could instead do

```
fun fromSeq(S) = Seq.reduce Set.union Set.empty (Seq.map Set.singleton S)
```

Exercise 1. *What is the work and span of the first version of* `fromSeq`*.*

Exercise 2. *Show that on a sequence of length n the second version of* `fromSeq` *does $O(C_w n \log n)$ work and $O(\log^2 n)$ span.*

3 Tables

A table is an abstract data type for storing data associated with keys. They are similar to sets, but along with each element (key) we store some data. The table ADT supplies operations for finding the value associated with a key, for inserting new key-value pairs, and for deleting keys and their associated value. Tables are also called dictionaries, associative arrays, maps, mappings, or, in set theory, functions. For the purpose of parallelism the interface we will discuss also supplies "parallel" operations that allow the user to insert multiple key-value pairs, to delete multiple keys, and to find the values associated with multiple keys.

As with sets, tables are very useful in many applications. Most languages have tables either built in (e.g. dictionaries in python), or have libraries to support them (e.g. map in the C STL library and the Java collections framework). We note that the interfaces for these languages and libraries have common features but typically differ in some important ways, so be warned. Most do not support the "parallel" operations we discuss. Here we will define tables mathematically in terms of set theory before committing to a particular language.

Formally, a table is set of key-value pairs where each key appears only once in the set. Such sets are called *functions* in set theory since they map each key to a single value. We will avoid this terminology so that we don't confuse it with functions in a programming language. However, note that the `(find T)` in the interface is precisely the "function" defined by the table `T`. In fact it is a *partial function* since the table might not contain all keys and therefore the function might not be defined on all inputs. Here is the definition of a table.

Definition 3.1. For a universe of keys $\mathbb{K}$, and a universe of values $\mathbb{V}$, the *Table* abstract data type is a type $\mathbb{T}$ representing the powerset of $\mathbb{K} \times \mathbb{V}$ restricted so that each key appears at most once (i.e., any set of key-value pairs where a key appears just once) along with the following functions:

$$
\begin{array}{lcll}
\texttt{empty} & : & \mathbb{T} & = \ \emptyset \\
\texttt{size}(T) & : & \mathbb{T} \to \mathbb{Z}^* & = \ |T| \\
\texttt{singleton}(k,v) & : & \mathbb{K} \times \mathbb{V} \to \mathbb{T} & = \ \{(k,v)\} \\
\texttt{filter}(f,T) & : & ((\mathbb{V} \to \mathbb{B}) \times \mathbb{T}) \to \mathbb{T} & = \ \{(k,v) \in T | f(v)\} \\
\texttt{map}(f,T) & : & ((\mathbb{V} \to \mathbb{V}) \times \mathbb{T}) \to \mathbb{T} & = \ \{(k,f(v)) | ((k,v) \in T)\} \\
\texttt{insert}(f,T,(k,v)) & : & (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T} \times (\mathbb{K} \times \mathbb{V}) \to \mathbb{T} & = \\
 & & \quad \forall k \in \mathbb{K}, \begin{cases} (k, f(v,v')) & (k,v') \in T \\ (k,v) & (k,v') \notin T \end{cases} & \\
\texttt{delete}(T,k)) & : & \mathbb{T} \times \mathbb{K} \to \mathbb{T} & = \ \{(k',v) \in T | k \neq k'\} \\
\texttt{find}(T,k) & : & \mathbb{T} \times \mathbb{K} \to (\mathbb{V} \cup \bot) & = \ \begin{cases} v & (k,v) \in T \\ \bot & otherwise \end{cases} \\
\texttt{merge}(f,T_1,T_2) & : & (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T} \times \mathbb{T} \to \mathbb{T} & = \\
 & & \quad \forall k \in \mathbb{K}, \begin{cases} (k, f(v_1,v_2)) & (k,v_1) \in T_1 \wedge (k,v_2) \in T_2 \\ (k,v_1) & (k,v_1) \in T_1 \\ (k,v_2) & (k,v_2) \in T_2 \end{cases} & \\
\texttt{extract}(T,S) & : & \mathbb{T} \times \mathbb{S} \to \mathbb{T} & = \ \{(k,v) \in T | k \in S\} \\
\texttt{erase}(T,S) & : & \mathbb{T} \times \mathbb{S} \to \mathbb{T} & = \ \{(k,v) \in T | k \notin S\}
\end{array}
$$

where S is the powerset of K (i.e., any set of keys), $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ and $\mathbb{Z}^*$ are the non-negative integers.

Distinct from sets, the `find` function does not return a Boolean, but instead it returns the value associated with the key k. As it may not find the key in the table, its result may be bottom ($\perp$). For this reason, in the Table library, the interface for find is `find : 'a table` $\rightarrow$ `key` $\rightarrow$ `'a option`, where `'a` is the type of the values.

Note that the `insert` function takes a function $f : (\mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V})$ as an argument. The purpose of f is to specify what to do if the key being inserted already exists in the table; f is applied to the two values. This function might simply return either its first or second argument, or it can be used, for example, to add the new value to the old one. The `merge` takes a similar function since it also has to consider the case that an element appears in both tables.

Technically a table is a set of pairs and can therefore be written as

$$\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\},$$

as long as the keys are distinct. However, to better identify when tables are being used, we will use the notation $k \mapsto v$ in pseudocode to indicate we are using a table, where the key k maps to the value v. For example we will use:

$$\{(k_1 \mapsto v_1), (k_2 \mapsto v_2), \ldots, (k_n \mapsto v_n)\}.$$

The set notation

$$\{(k \mapsto f(v) : (k \mapsto v) \in T\}$$

is equivalent to $\texttt{maps}(f, T)$ and

$$\{(k \mapsto v) \in T | f(v)\}$$

is equivalent to $\texttt{filter}(f, T)$.

The costs of the table operations are very similar to sets.

	Work	*Span*
$\texttt{size}(T)$ $\texttt{singleton}(k, v)$	$O(1)$	$O(1)$
$\texttt{filter}(f, T)$	$O\Big(\sum_{(k,v)\in T} W(f(v))\Big)$	$O\Big(\log \lvert T\rvert + \max_{(k,v)\in T} S(f(v))\Big)$
$\texttt{map}(f, T)$	$O\Big(\sum_{(k,v)\in T} W(f(v))\Big)$	$O\Big(\max_{(k,v)\in T} S(f(v))\Big)$
$\texttt{find}(S, k)$ $\texttt{insert}(T, (k, v))$ $\texttt{delete}(T, k)$	$O(C_w \log \lvert T\rvert)$	$O(C_s \log \lvert T\rvert)$
$\texttt{extract}(T_1, T_2)$ $\texttt{merge}(T_1, T_2)$ $\texttt{erase}(T_1, T_2)$	$O\big(C_w m \log(\frac{n+m}{m})\big)$	$O\big(C_s \log(n+m)\big)$

where $n = \max(|T_1|, |T_2|)$ and $m = \min(|T_1|, |T_2|)$.

As with sets there is a symmetry between the three operations `extract`, `merge`, and `erase`, and the three operations `find`, `insert`, and `delete`, respectively, where the prior three are effectively "parallel" versions of the earlier three.

We note that, in the SML Table library we supply, the functions are polymorphic (accept any type) over the values but not the keys. In particular the signature starts as:

```
signature TABLE =
sig
  type 'a table
  type 'a t = 'a table
  structure Key : EQKEY
  type key = Key.t
  structure Seq : SEQUENCE
  type a seq = 'a Seq.seq
  type set = unit table
  ...
  val find : 'a table -> key -> 'a option
  ...
```

The `'a` in `'a table` refers to the type of the value. The key type is fixed to be `key`. Therefore there are separate table structures for different keys (e.g. `IntTable`, `StringTable`). The reason to do this is because all the operations depend on the key type since keys need to be compared for a tree implementation, or hashed for a hash table implementation. Also note that the signature defines `set` to be a `unit table`. Indeed a set is just a special case of a table where there are no values.

In the SML Table library we supply a `collect` operation that is analogous to the collect described at the beginning of the class. It takes a sequence S of key-value pairs and generates a table mapping every key that appears in S to all the values that were associated with it in S. It is equivalent to using a sequence collect followed by a `Table.fromSeq`. Alternatively it can be implemented as

```
1  fun collect(S) =
2  let
3    val S' = ⟨ {k ↦ ⟨ v ⟩} : (k, v) ∈ S ⟩
4  in
5    Seq.reduce (Table.merge Seq.append) {}  S'
6  end
```

Exercise 3. *Figure out what this code does.*

4 Building and searching an index

[This material will be covered in lecture 7]

Here we consider an application of sets and tables. In particular, the goal is to generate an index of the sort that Google or Bing create so that a user can make word queries and find the documents in which those words occur. We will consider logical queries on words involving *and*, *or*, and *andnot*. For example a query might look like

"CMU" *and* "fun" *and* ("courses" *or* "clubs")

and it would return a list of web pages that match the query (*i.e.*, contain the words "CMU", "fun" and either "courses" or "clubs"). This list would include the 15-210 home page, of course.

These kinds of searchable indexes predate the web by many years. The Lexis system, for searching law documents, for example, has been around since the early '70s. Now searchable indexes are an integral part of most mailers and many operating systems. By default Google supports queries with *and* and *adjacent to* but with an advanced search you can search with *or* as well as *andnot*.

Let's imagine we want to support the following interface

```
signature INDEX = sig
  type word = string
  type docId = string
  type 'a seq
  type index
  type docList

  val makeIndex : (docId * string) seq -> index
  val find : index -> word -> docList
  val And : docList * docList -> docList
  val AndNot : docList * docList -> docList
  val Or : docList * docList -> docList
  val size : docList -> int
  val toSeq : docList -> docId seq
end
```

The input to `makeIndex` is a sequence of pairs each consisting of a document identifier (e.g. the URL) and the contents of the document. So for example we might want to index recent tweets, that might include:

$T = \langle$ ("jack", "chess club was fun"),
("mary", "I had a fun time in 210 class today"),
("nick", "food at the cafeteria sucks"),
("sue", "In 217 class today I had fun reading my email"),
("peter", "I had fun at nick's party"),
("john", "tidliwinks club was no fun, but more fun than 218"),
$\ldots\rangle$

We can make an index from these tweets:

```
1  val f = find (makeIndex(T))
```

In addition to making the index, this partially applies `find` on it. We can then use this index for various queries, for example:

```
1  toSeq(And(f "fun", Or(f "class", f "club"))
2     ⇒ ⟨"jack", "mary", "sue", "john"⟩

3  size(f "fun")
4     ⇒ 5
```

We can implement this interface very easily using sets and tables. The `makeIndex` function can be implemented as follows.

```
1  fun makeIndex(docs) =
2  let
3     fun tagWords(name, str) = ⟨ (w, name) : w ∈ tokens(str) ⟩
4     val Pairs = flatten ⟨ tagWords(d) : d ∈ docs ⟩
5  in
6     {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Table.collect(Pairs)}
7  end
```

Assuming that all tokens are constant length, the cost of `makeIndex` is dominated by the collect, which is basically a sort. It therefore has $O(n \log n)$ work and $O(\log^2 n)$ span assuming the words have constant length. The rest of the interface can be implemented as

```
1  fun find(T, v) = Table.find(T, v)
2  fun And(s1, s2) = s1 ∩ s2
3  fun Or(s1, s2) = s1 ∪ s2
4  fun AndNot(s1, s2) = s1 \ s2
5  fun size(s) = |s|
6  fun toSeq(s) = Set.toSeq(s)
```

Note that if we do a `size(f "red")` the cost is only $O(\log n)$ work and span. It just involves a search and then a length.

If we do `And(f "fun", Or(f "courses", f "classes"))` the worst case cost is

- work = $O($`size(f "fun")` + `size(f "courses")` + `size(f "classes")`$)$
- span = $O(\log |index|)$

5 SML Code

5.1 Indexes

```
functor TableIndex(Table : TABLE where type Key.t = string) : INDEX =
struct

structure Seq = Table.Seq
structure Set = Table.Set

type word = string
type docId = string
type 'a seq = 'a Seq.seq
type docList = Table.set
type index = docList Table.table
```

```
fun makeIndex docs =
let
  fun toWords str = Seq.tokens (fn c => not (Char.isAlphaNum c)) str

  fun tagWords(docId,str) = Seq.map (fn t => (t, docId)) (toWords str)

  (* generate all word-documentid pairs *)
  val allPairs = Seq.flatten (Seq.map tagWords docs)

  (* collect them by word *)
  val wordTable = Table.collect allPairs

in
  (* convert the sequence of documents for each word into a set
     which removes duplicates*)
  Table.map Set.fromSeq wordTable
end

fun find Idx w =
   case (Table.find Idx w) of
      NONE => Set.empty
    | SOME(s) => s

val And = Set.intersection
val AndNot = Set.difference
val Or = Set.union
val size = Set.size
val toSeq = Set.toSeq

end
```