Record-Based Processing

... or: "How to do stuff to files made up of records".

Topic	***Folk & Zoellick***
• Old Hat & New Hat	
• Basic Record Operations	
• Searching	
◦ Direct Access	§ 4.2.4
◦ Sequential Search	§§ 4.2.2, 5.3.3
◦ Binary Search	§§ 5.3.2, 5.3.3
• Cosequential Processing	
◦ Matching	§ 7.1.1
◦ Merging	§ 7.1.2
• Batch Updating	
• Space Reclamation	§ 5.2

The Key

Key

some subset of fields

Unique Key

can be used to identify each record in file uniquely

Dataless Key

contains no information specific to the record

What's the point of avoiding real data in keys?

- real data (e.g. Last Name, First Name, etc.) is rarely guaranteed unique
- real data has a tendency to change

Order and Disorder

The kinds of operations that can be performed often depend on whether the records in a file are ordered or unordered.

If the records are ordered, the file is considered ***sorted***.
If the records are unordered, the file is considered ***unsorted***.

File *F* contains records $r_1 \ldots r_n$ having keys $k_1 \ldots k_n$.

F is sorted, increasing:

$$k_1 < k_2 < \ldots < k_{n-1} < k_n$$

F is sorted, non-decreasing:

$$k_1 \leq k_2 \leq \ldots \leq k_{n-1} \leq k_n$$

F is sorted, decreasing:

$$k_1 > k_2 > \ldots > k_{n-1} > k_n$$

F is sorted, non-increasing:

$$k_1 \geq k_2 \geq \ldots \geq k_{n-1} \geq k_n$$

What Sort of File?

Q: Are these files sorted? How?

5 4 3 2 1 0 -1	10 11 12 AA AB AC	1 12 12 121 123 2 22 241 34	a ab ab aba abc b bb bdc cd	2 20 -2 0005 01	charlie bravo alpha Foxtrot

A:

Big O

Complexity analysis allows us to compare the efficiency of very different algorithms. If an algorithm is doing some *thing* with *n* items, the "big O" notation tells us how the performance of the algorithm varies as *n* varies.

```
for(i = 0; i < n; i++)
   for(j = 0; j < n; j++)
      ch = getc(infil);
```

Let's say every read (`getc()`) takes 1 ms.

Q: If *n* is 4, how many milliseconds does the program spend reading?

A:

Q: What if we increase *n* to 5? ... to 6?

A:

Q: What is the "complexity" (in big O notation) of our program?

A:

Q: Is the number of reads a fair measure of complexity of this program?

A:

Basic Record Operations

Things to do to Records:

- *add* new records to a file
  - where should the record be added? (beginning? middle? end?)
  - if added in the middle or at the beginning, what happens to the keys of the rest of the records?
  - if added in the middle or at the beginning, is there room available

- *modify* existing records in a file
  - does the modification change the length of the record?
  - if the length changes, what happens to the keys of the rest of the records?
  - does the key change?

- *delete* an existing record from a file
  - do we erase the record or simply mark it as deleted?
  - if the record deleted is in the middle of the file, do we have to shuffle all the remaining records around?
  - do the keys of other records change?
  - what do we do with the space available after deleting a record?

If you can figure out how to *add* and *delete* properly, *modify* doesn't really pose any new problems.

Searching: Direct Access

Searching:

Looking for a particular record r_i in a file.

Direct Access:

Seeking directly to the beginning of r_i without having to read any other records in the file.

If we have fixed length records...

knowing the record number *i* of r_i allows us to seek directly to the beginning of the record (*i* is called the *relative record number* or *RRN*).

If we have variable length records...

we must know the *byte offset* of r_i to seek directly to the beginning of the record.

Q: What kind of *key* is *RRN*? What else?

A:

□

Hmmmm... if only there were a way to know the byte offset of every record in a file....

Searching: Sequential Search

RRN and byte offset are decent keys: they're unique, they're dataless. But what if we don't know the RRN (or byte offset)?

For example, say we want to find the record for *John Gruden* in this file:

```
Olausson, Fredrik, ana, D, 72, 198
Pushor, Jamie, ana, D, 75, 218
Haller, Kevin, ana, D, 74, 195
  ...
Matteau, Stephane, was, L, 76, 220
Friesen, Jeff, was, L, 72, 200
Rathje, Mike, was, D, 77, 230
```

We have no choice but to read through all 1,294 records in the file, record-by-record, comparing the first field of each record to *Gruden* and the second field to *John*.

Q: How many records will we have to read to find Gruden (best case, worst case, average case)?

A:

Q: What if Gruden is not in the file?

A:

Q: What is the complexity (big O) of this search?

A:

Searching: Sequential Search (cont.)

But what if that file were *sorted* on the first and second fields?

```
Aalto, Antti, ana, C, 74, 195
Abid, Ramzi, col, L, 74, 195
Abrahamsson, Elias, bos, D, 75, 240
   .
   .
   .
Zubrus, Dainius, phi, R, 75, 220
Zultek, Matt, la, L, 75, 218
Zyuzin, Andrei, sjo, D, 73, 200
```

Q: What is the complexity of sequential search through the sorted file?

A:

Q: What if Gruden is not in *this* file?

A:

□

But we can do better...

Since the file is sorted, we can look at any record in the file and know immediately if what we're looking for comes before the record or after the record. By looking at the middle record in the file, we can cut the search space in half with just one read!

Searching: Binary Search

```
set low to 0; set high to 1,294

while high ≥ low
   set middle to (high + low) / 2
   read record middle

   if key(middle) = key2find
      return(middle)!
   else if key(middle) < key2find
      set low to middle + 1
   else
      set high to middle - 1
endwhile

return(BUST)
```

Q: What is the complexity of binary search through the sorted file?

A:

Q: What if Gruden is not in the file?

A:

Q: What if we want to add a record to the file?

A:

Cosequential Processing

Sometimes we have two files (or more) that share some common information. Often we want to find records in the two files that have the same key value.

For example, let's say we have two files: one lists the NHL Eastern Conference leading Ottawa Senators, and the other lists some Scandinavians who have never been in my kitchen.

- *How could we process the records in these two files to find out which Scandinavian Senators have never been in my kitchen?*

- *In fact, none of the Senators (Scandinavian or otherwise) have ever been in my kitchen. How could we process the records in these two files to generate a single complete ordered list?*

```
Alfredsson
Arvedson
Berg
Bonk
Dackell
Gardiner
Gruden
Hossa
Johansson
Kravchuk
Laukkanen
Martins
McEachern
Murray
Phillips
Pitlick
Prospal
Redden
Rhodes
Salo
Sarault
Traverse
Tugnutt
VanAllen
Yashin
York
```

```
Ahonen
Alfredsson
Arvedson
Björklund
Dackell
Doberhof
Glad
Johansson
Jurvanen
Konestabo
Laukkanen
Osnes
Puukko
Salo
Trygg
Uggla
```

Matching

Matching is the coordinated processing of multiple files to find only records that have the same key in all files (*intersection*).

Here's how it works for two files:

```
open both files F_A F_B

get k_A from F_A
get k_B from F_B

while records remain in F_A and F_B
   k_A < k_B: get next k_A from F_A
   k_A > k_B: get next k_B from F_B
   k_A = k_B: output k_A
              get next k_A from F_A
              get next k_B from F_B
endwhile

close files
```

Q: What happens if there is an error in the ordering of a file?

A:

Merging

Merging is the coordinated processing of multiple files to find the *union* of records in all files (no duplicates).

Here's how it works for two files:

```
open both files F_A F_B

get k_A from F_A
get k_B from F_B

while records remain in F_A or F_B
   k_A < k_B: output k_A
              get next k_A from F_A
   k_A > k_B: output k_B
              get next k_B from F_B
   k_A = k_B: output k_A
              get next k_A from F_A
              get next k_B from F_B
endwhile

close files

/* Note: when getting the next k_X from file F_X,
   if F_X is at end of file, set k_X to some
   "out-of-range" high value.                    */
```

Q: Wow, isn't this almost the same algorithm as the Matching algorithm?

Batch Updating

When there are frequent changes (additions, modifications, deletions) to records in a data file, and instantaneous updating is not required, it is often more practical to "save up" the changes in a separate file (called a *transaction* file) until there are enough to warrant updating the data file.

Data File

```
Alfredsson, Daniel, R, 71, 194
Bonk, Radek, C, 75, 224
Hossa, Marian, L, 73, 194
Murray, Chris, R, 74, 215
Neckar, Stan, D, 73, 212
Phillips, Chris, D, 74, 218
Yashin, Alexei, C, 75, 228
```

Transaction File

```
98/11/27: del, Neckar, Stan
98/11/27: add, Berg, Bill, L, 73, 205
99/01/01: mod, Murray, Chris, ?, ?, 222
99/02/27: add, Barker, Ken, D, 70, ?
```

We want to use the *transaction file* to update the *data file*. But if we leave the transaction file in its current order, we'll have to do a search through the data file for every record in the transaction file.

There must be a better way...

Batch Updating (cont.)

If we sort the transaction file on the same key as the data file, we can use *cosequential processing* to update the data file in one loop!

Transaction File

```
99/02/27: add, Barker, Ken, D, 70, ?
98/11/27: add, Berg, Bill, L, 73, 205
99/01/01: mod, Murray, Chris, ?, ?, 222
98/11/27: del, Neckar, Stan
```

And let's repeat the data file just for fun.

Data File

```
Alfredsson, Daniel, R, 71, 194
Bonk, Radek, C, 75, 224
Hossa, Marian, L, 73, 194
Murray, Chris, R, 74, 215
Neckar, Stan, D, 73, 212
Phillips, Chris, D, 74, 218
Yashin, Alexei, C, 75, 228
```

Batch Updating Algorithm

The algorithm for updating might look like this:

```
open both files F_D F_T

get k_D from F_D
get k_T from F_T

while records remain in F_D or F_T
   k_D < k_T: output r_D, get next k_D from F_D
   k_D > k_T: if action = add
                 output r_T, get next k_T from F_T
              else error
              endif
   k_D = k_T: if action = mod
                 output r_D with new field values
              else if action = del
                 do nothing
              else error
              endif

              get next k_D from F_D
              get next k_T from F_T
endwhile

close files

/* Again: when getting the next k_X from file F_X,
   if F_X is at end of file, set k_X to some
   "out-of-range" high value (like 'ZZZZZZ'). */
```

Not Batch Updating

The cosequential processing algorithms do exhaustive sequential reading of multiple files. For example, in the batch updating scenario, if there are *N* records in the data file and *M* records in the transaction file, the algorithm is O(M + N) (or O(max(M, N))).

What if *instantaneous* updating is required? That is, every time a transaction takes place, the data file must be updated immediately.

Q: What is the complexity of instantaneous updating of a data file with *N* records with *M* transactions?

A:

□

The updating we've seen writes a new file with the updated information. That's kind of a waste if we're only changing one record! Can't we just make the changes directly to the data file?

Modifying a File Directly

Suppose we have to make some changes to the following file and we don't care about maintaining order in the file.

```
  1  Olausson, Fredrik, ana, D, 72, 198
  2  Pushor, Jamie, ana, D, 75, 218
  3  Haller, Kevin, ana, D, 74, 195
  4  Trnka, Pavel, ana, D, 75, 200
 ...
n-2  Matteau, Stephane, was, L, 76, 220
n-1  Friesen, Jeff, was, L, 72, 200
  n  Rathje, Mike, was, D, 77, 230
```

Let's say David Oliver gets called up from the minors. We need to add the record

```
Oliver, David, ott, L, 72, 190
```

```
  1  Olausson, Fredrik, ana, D, 72, 198
  2  Pushor, Jamie, ana, D, 75, 218
  3  Haller, Kevin, ana, D, 74, 195
  4  Trnka, Pavel, ana, D, 75, 200
  ...
n-2  Matteau, Stephane, was, L, 76, 220
n-1  Friesen, Jeff, was, L, 72, 200
  n  Rathje, Mike, was, D, 77, 230
n+1  Oliver, David, ott, L, 72, 190
```

Adding a record to our file was pretty simple. What about deleting a record?

Storage Compaction

Let’s say Kevin Haller retires at the ripe old age of 28. Instead of writing a brand new file with Haller’s record deleted, we can just mark the record with a special mark that means: "this record is deleted". Any software using the file will just skip over records marked as deleted.

```
  1   Olausson, Fredrik, ana, D, 72, 198
  2   Pushor, Jamie, ana, D, 75, 218
  3 ~ Haller, Kevin, ana, D, 74, 195
  4   Trnka, Pavel, ana, D, 75, 200
  ...
n-2   Matteau, Stephane, was, L, 76, 220
n-1   Friesen, Jeff, was, L, 72, 200
  n   Rathje, Mike, was, D, 77, 230
n+1   Oliver, David, ott, L, 72, 190
```

Every once in a while, we can run a special *compactor* program that writes out a new file omitting all the deleted records (this is known as *space reclamation* by *storage compaction*).

Deleting records in this manner is simple and fast. But there’s another advantage as well...

Q: What’s the other advantage?

A:

Q: What’s the *other* other advantage?

A:

Dynamic Reclamation

The problem with reusing the space left by deleted records is that we have to search sequentially through the file for the available spots.

A better solution would be to use a data structure (let's say, a *stack*) to keep track of which records are deleted.

We store the RRN of the deleted record at the top of the stack in a special record in the file. The file below has no deleted records.

```
      0
  1     Olausson, Fredrik, ana, D, 72, 198
  2     Pushor, Jamie, ana, D, 75, 218
  3     Haller, Kevin, ana, D, 74, 195
  4     Trnka, Pavel, ana, D, 75, 200
  5     Barker, Ken, ana, D, 70, ???
  ...
n-2     Matteau, Stephane, was, L, 76, 220
n-1     Friesen, Jeff, was, L, 72, 200
  n     Rathje, Mike, was, D, 77, 230
n+1     Oliver, David, ott, L, 72, 190
```

Dynamic Reclamation (cont.)

If we then delete the record with RRN 5, we mark the record as deleted, change the special "next in stack" field to 0, and change the special "top of stack" record to 5.

```
      5
  1     Olausson, Fredrik, ana, D, 72, 198
  2     Pushor, Jamie, ana, D, 75, 218
  3     Haller, Kevin, ana, D, 74, 195
  4     Trnka, Pavel, ana, D, 75, 200
  5 ~ 0 Barker, Ken, ana, D, 70, ???
  ...
n-2     Matteau, Stephane, was, L, 76, 220
n-1     Friesen, Jeff, was, L, 72, 200
  n     Rathje, Mike, was, D, 77, 230
n+1     Oliver, David, ott, L, 72, 190
```

Dynamic Reclamation (cont.)

Then if we delete record 3, we mark the record as deleted, change the "next in stack" field to the value of the current "top of stack", and change the "top of stack" to 3.

```
      3
  1     Olausson, Fredrik, ana, D, 72, 198
  2     Pushor, Jamie, ana, D, 75, 218
  3 ~ 5 Haller, Kevin, ana, D, 74, 195
  4     Trnka, Pavel, ana, D, 75, 200
  5 ~ 0 Barker, Ken, ana, D, 70, ???
  ...
n-2     Matteau, Stephane, was, L, 76, 220
n-1     Friesen, Jeff, was, L, 72, 200
  n     Rathje, Mike, was, D, 77, 230
n+1     Oliver, David, ott, L, 72, 190
```

Dynamic Reclamation: The Reclaiming Part

The next time we add a record, we can reclaim deleted space simply by jumping directly to the deleted record at the top of the stack:

```
      5
  1     Olausson, Fredrik, ana, D, 72, 198
  2     Pushor, Jamie, ana, D, 75, 218
  3     Leclerc, Mike, ana, L, 73, 205
  4     Trnka, Pavel, ana, D, 75, 200
  5 ~ 0 Barker, Ken, ana, D, 70, ???
  ...
n-2     Matteau, Stephane, was, L, 76, 220
n-1     Friesen, Jeff, was, L, 72, 200
  n     Rathje, Mike, was, D, 77, 230
n+1     Oliver, David, ott, L, 72, 190
```

Food for Thought

What if the records are variable length?

Q: How do we refer to available records in the stack?

A:

Q: What if the space left by the deleted record is not big enough for the record we want to add?

A: