# CMPS340 File Processing
# The Development of an Algorithm for the Sequential File Update Problem via the Method of Successive Generalization

Robert McCloskey

November 30, 2012

## 1 Background

A typical use of a data file is to describe the state, at some point in time, of each item in a collection of "items of interest". For example, each record in an "accounts receivable" file might contain a client identifier together with a number representing the amount of money owed by that client. As the states of the items of interest change, so should the contents of the file describing these states, in order that the file remain "current". For example, if a client makes a payment, the corresponding record in the accounts receivable file should be modified accordingly.

Depending upon the purpose served by such a data file, it may or may not be necessary to update it in *real-time* (i.e., as each change-of-state occurs among the items of interest). In order to be useful, some files *must* contain data that is accurate "up-to-the-minute" (or even "up-to-the-second"). A good example is a file holding airline reservation data, such as would be used by a travel agent to reserve seats for customers. (If a reservation is made for a seat on some flight, it had better not be possible to reserve the same seat for someone else two seconds later.) An example of a file whose contents need not be updated in real-time is one in which is stored "hours worked so far this week by each employee" and that is processed, at the end of the week, in order to generate pay checks. (Clearly, it would not be necessary to update each working employee's record on a minute-by-minute, or even hourly, basis.)

In general, it is less costly to update a file periodically than to update it in real-time because, to achieve acceptable performance, the latter approach usually requires the use of more sophisticated file structures (e.g., an index or a hash table).[1] Thus, whenever it is suitable, we prefer to use the periodic update approach. Typically, this entails recording the state-changing events, which are encoded as so-called *transactions*, in a separate file and then, after some period of time has elapsed, applying the accumulated batch of transactions to the so-called

[1]Files that are accessed and modified simultaneously by numerous processes, such as the airline reservation file mentioned above, require even more sophisticated means to prevent data from becoming corrupted.

*master* file (i.e., the file containing the state information), thereby producing a new, up-to-date master file. The problem of applying a batch of transactions to a master file is often referred to as the Sequential File Update (SFU) problem; our goal is to develop an algorithmic solution. It is important for students of computing to learn about SFU not only because it is interesting from an algorithmic point of view, but also because SFU is ubiquitous in the realm of real-world data processing. Indeed, many common batch-oriented applications —including accounts payable/receivable, inventory control, and others— are naturally framed as particular instances of SFU.

## 2 Description of the Sequential File Update Problem

The inputs are two files: a *master file*, whose contents describe the states of some collection of items of interest at some time $t$, and a *transaction file*, in which has been recorded a sequence of transactions, each of which corresponds to a change-of-state in one of the items of interest, occurring between time $t$ and some later time $t + \delta$. The output is a new master file, which is the result of applying to the (old) master file the transactions appearing in the transaction file. The new master file thereby describes the states of the items of interest at time $t + \delta$.

In the standard formulation of the problem, each transaction is classified as being of one of three kinds: **add**, **change**, or **delete**. An **add** transaction describes a record that is to be inserted into the master file; a **delete** transaction identifies a record that is to be removed from the master file; a **change** transaction identifies a record in the master file and describes a modification that is to be made to it (e.g., "change value of `HOURLY-WAGE` field to $13.50" or "subtract 10 from `QUANTITY-ON-HAND` field"). We assume that the master file has a *key* field[2] by which its records can be uniquely identified and that this is also its *ordering* field[3].

A transaction record, then, contains an indication of which kind of transaction it describes (i.e., **add**, **change**, or **delete**), a value for the key field of the master file[4] to identify the master record to which it pertains, and any other data necessary to fully describe its intended effect.

Consider an accounts receivable master file in which each record contains a customer ID (the key field), a balance (indicating the amount owed by the customer), and possibly other fields that are irrelevant to the current discussion. See Figure 1. For convenience, in our example we (unrealistically) allow first names to serve as customer ID's and we express all monetary amounts in whole numbers. Suppose that there are four kinds of state-changing events:

(i) A new account is opened.

[2]A field, or a combination of fields, qualifies as a key of a file if no two records in the file are allowed to have identical contents in that (those) field(s).

[3]A field qualifies as the *ordering* field of a file if each record in the file, except for the first, contains in this field a value greater than or equal to (under some suitable definition of "greater than") the value in the previous record.

[4]Following relational database terminolgy, we refer to this as the transaction record's foreign key.

(ii) An existing account is closed.

(iii) A customer incurs debt (e.g., by making a purchase).

(iv) A customer decreases debt (e.g., by making a payment).

Clearly, (i) and (ii) can be modeled by **add** and **delete** transactions, respectively, and both (iii) and (iv) can be modeled by **change** transactions. As illustrated in Figure 1, each transaction contains a `Kind` field indicating its type, as well as a `Cust-ID` field to identify the master record to which it pertains. No other information is required by **add** or **delete** transactions. (Assume that when a new account is opened, its balance is zero.) In **change** transactions, there is a third field, `Amount`, which indicates the amount to be added to the client's balance. For example, the first two transactions record an increase of \$34 in Helen's balance and a decrease of \$27 in Cathy's, respectively.

| Cust-ID | Balance | other fields |
|---|---|---|
| Alan | 40 | ... |
| Beth | 27 | ... |
| Cathy | 0 | ... |
| Emily | 74 | ... |
| Frank | -12 | ... |
| Helen | 77 | ... |
| Iggy | 0 | ... |
| Jack | 14 | ... |
| Kim | -54 | ... |

Accounts Receivable
Master File

| Kind | Cust-ID | Amount |
|---|---|---|
| Change | Helen | 34 |
| Change | Cathy | -27 |
| Add | George | |
| Change | Emily | -40 |
| Change | George | 83 |
| Delete | Emily | |
| Change | George | -19 |
| Change | George | 32 |
| Change | Jack | -14 |
| Delete | George | |
| Delete | James | |
| Change | George | 20 |
| Add | Emily | |

Accounts Receivable
Transaction File

Figure 1: Accounts Receivable Master File and Transaction File

Notice that a given master record may be the target of zero, one, or more transactions. If an **add** transaction specifies that a new master record is to be created, then it makes sense for subsequent transactions to pertain to that new record. (E.g., see the *George* transactions.) If a **delete** transaction specifies that a master record is to be removed, then it makes sense for a subsequent **add** transaction to cause a new master record with the same key value to be created. (E.g., see the *Emily* transactions.)

It is also possible for a transaction to be *invalid*, meaning that there is no sensible way to carry it out. There are essentially two cases: *adding a record with the same key as an existing record* and *changing or deleting a non-existent record.* To fully comprehend these, we need a precise understanding of what it means to say that a master record (with some particular key) exists. Unfortunately, it is not as simple as to say "A record with that key appears in the (old) master file". Rather, we should say that a master record (with a particular key) exists, relative to the *current* transaction, if either

(i) it exists in the (old) master file and no (valid) **delete** transaction pertaining to that record precedes the current transaction, or

(ii) a (valid) **add** transaction creating it occurs prior to the current transaction, and, between that transaction and the current one, there is no (valid) **delete** transaction removing it.

Notice that, unlike the master file, the records in the transaction file are not ordered with respect to the `Cust-ID` field. Indeed, it is much more likely that the order in which the transactions appear in the file matches the order in which they were "posted" (i.e., written into the file), which probably corresponds, at least roughly, to the order in which the corresponding real-life events took place. In general, the order in which transactions are applied to master records has a profound effect upon the resulting master file. However, due to the fact that applying a transaction pertaining to one master record has no effect upon any other master record, this observation holds only with respect to transactions with the same foreign key. That is, if, for example, six transactions pertain to *George* and three pertain to *Emily*, it makes no difference in what order we apply them, provided that the six *George* transactions are applied in the correct order, relative to one another, and similarly for the three *Emily* transactions. Naturally, this generalizes to the case of there being transactions pertaining to three or more distinct master records.

The conclusion we draw is that we may apply the transactions in any order we like, provided that within each group of transactions pertaining to the same key, we apply them in the proper order. (To keep things simple, we will assume here that the proper order in which to apply transactions with the same key corresponds to the order in which they appear in the transaction file.) Given that the records in the master file are ordered by the key field, this suggests that, as a first step in solving SFU, we should sort the transaction records —using a **stable** sorting algorithm[5]— using the key field as the sort key. This keeps transactions with the same key in the proper relative order, while at the same time reducing the subsequent task of applying transactions to the master file to a particular case of an algorithmic pattern/template that has been called *cosequential processing*. Indeed, it should not surprise the reader to learn that, with the files ordered in this way, the task of applying transactions to master records to produce new master records is much like that of merging (or finding the intersection of) two ordered lists. In particular, this means that the job can be completed using a single sweep over each of the two input files. In what follows, then, we assume that the transaction file has already been sorted.

[5]A sorting algorithm is said to be stable if two records with the same sort key are guaranteed to end up in the same positions, relative to one another, as they were in the original file. In the current context, this means that two transaction records pertaining to the same master record will remain in the same relative order as they were initially.

## 3 Development of an Algorithm

In developing an algorithm, we shall employ a method that reasonably could be called "development by successive generalization".[6] That is, we start by developing an algorithmic solution to a very restricted version of the problem. Then we lift some of the restrictions (thereby generalizing the problem) and modify the algorithm so that it solves the more general version of the problem. We repeat this step until we have arrived at a solution to the original (general version of the) problem.

### 3.1 Admitting only Valid Change Transactions

Let us begin by imposing the following restrictions upon the transaction file:

**(a)** Every transaction is a **change**.

**(b)** Every transaction is valid.

We could say, equivalently, that the transaction file contains only **change** transactions, all of which pertain to master records appearing in the original master file. Keeping in mind that each master record may be the target of any number of transactions, we propose the following high-level algorithmic solution:

```
FOR each master record MR
   FOR each transaction record TR pertaining to MR
      change MR in accord with TR;
   ROF;
   write MR to new master file;
ROF;
```

Each iteration of the outer loop handles a single master record. During each such iteration, all transactions pertaining to the current master record are applied to it and the resulting master record is written to the new master file.

Translating this program to somewhat less abstract pseudocode, we get the program in Figure 2. A few explanatory remarks are in order. First, the program is written under the assumption that the master file ends with a *sentinel* record in which the `Key` field equals `HIGH_VAL`, which is simply some value that is "greater than" any value that may occur in the `Key` field of a normal (i.e., non-sentinel) record.[7] Also, in order to keep us mindful of the fact that we

[6]A book by Dromey (*Program Derivation: The Development of Programs From Specifications*, Addison-Wesley, 1989) propounds this approach, although Dromey does not use the term *successive generalization*.

[7]The use of a *sentinel* value at the end of a file (or list) is a commonly-used technique that often admits a slightly simpler algorithmic solution than would be possible otherwise. Note that the presence of the sentinel record may be simulated, rather than real; for example, in the present context, we could design the `get()` operation so that, if all the records in the file have already been read in, it returns a record containing `HIGH_VAL` in its `Key` field.

intend to modify the program to handle **add** and **delete** transactions, we process the current transaction via a `if` statement in which the code segments that handle the cases of **add** and **delete** transactions are replaced by stubs. Aside from these points, the comments provided in the program should suffice to make it easily comprehensible by the reader.

```
TR := TF.get();    //read 1st record of transaction file, TF, into TR
MR := MF.get();    //read 1st record of master file, MF, into MR

//loop invariant: MR.key() <= TR.fKey()
while MR.key() != HIGH_VAL
{
   //loop invariant: MR.key() <= TR.fKey()
   while MR.key() = TR.fKey()  //process all transactions pertaining to MR

      if TR.isAdd()
         { }                 //Add transactions not yet supported
      else if TR.isChange()
         { TR.change(MR); }  //change MR in accord with TR
      else if TR.isDelete()
         { }                 //Delete transactions not yet supported
      else
         { }                 //should never happen

      TR := TF.get();        //read next transaction record into TR

   }

   //assert: MR.key() < TR.fKey() (thus, no more transactions pertain to MR)
   NMF.put(MR);                 //write MR into new master file
   MR := MF.get();              //read next (old) master record into MR
}
```

Figure 2: Program Allowing Only Valid Change Transactions

## 3.2 Admitting Valid Delete Transactions

Let us now modify the program in Figure 2 so that it is capable of handling not only valid **change** transactions but also valid **delete** transactions. (In the absence of **add** transactions, for all transactions to be valid requires, as before, that each one pertains to a record that exists in the old master file and, in addition, that any **delete** transaction must be the last among those pertaining to a particular master record.) Perhaps the most obvious way to process a delete transaction is to discard the current master record simply by fetching the next one. That is, within the **if** statement in Figure 2, change the line intended to handle **delete** transactions to `MF.get();`

The resulting program is correct. However, we choose to abandon this approach because it departs —in a subtle yet significant way— from the basic logical structure of the high-level algorithm initially proposed. Specifically, it fails to satisfy this condition:

*Each iteration of the (outer) loop performs whatever processing is required by one, and only one, master record.*

Why should we insist (or even prefer, for that matter) that our algorithm satisfies such a condition? *Answer:* Because it reflects a simple, coherent logical structure, and if an algorithm's logical structure is simple, it tends to be —relative to an equivalent algorithm having a more complicated logical structure— not only more easily understood but also more easily modified. (Understandability and modifiability are two important qualities of algorithms!) In the present context, modifiability is especially relevant because our intent is to continue modifying the algorithm to make it solve successively more general (i.e., less restrictive) versions of SFU until, ultimately, we arrive at a solution to the unrestricted form of SFU. As an exercise, the reader is invited to take the abandoned program and to attempt to transform it into a solution for the general form of SFU.

In order to ensure that our algorithm remains faithful to the condition discussed above, let us agree that the only time a record from the old master file will be fetched is as the last step of the outer loop. How, then, shall we handle a **delete** transaction? *Answer*: We invent a way to indicate that the buffer holding the current master record, `MR`, is *logically empty*, and we implement the delete operation so that its effect is to cause `MR` to become (logically) empty. One natural way to accomplish this is to introduce a boolean variable, which we call `MR_Empty`, whose value indicates whether or not `MR` is (logically) empty. Carrying out a delete transaction then amounts to setting `MR_Empty` to **true**.

In order to ensure that the value of `MR_Empty` accurately reflects the state of `MR`, we must remember to set it to **false** each time a master record is read into `MR`. We must also make sure, after having applied all pertinent transactions to the current master record, that the contents of `MR` are written if and only if `MR` is not logically empty. We arrive at the program in Figure 3.

## 3.3 Admitting Some Invalid Transactions

Having introduced the boolean variable `MR_Empty`, it becomes quite easy to handle a **change** or **delete** transaction that is invalid due to its pertaining to a master record that was deleted by a previous transaction. Thus, as our next generalization, let us allow for the existence of invalid transactions in this restricted class.

In order to prevent the presentation of the algorithm from becoming cluttered, we introduce a subprogram `Apply_Transaction` and isolate inside it the **if** statement that carries out the application of a single transaction. The main program is thus what appears in Figure 4. Figure 5 contains the initial version of `Apply_Transaction`.

```
TR := TF.get();      //read 1st record of TF into TR
MR := MF.get();      //read 1st record of MF into MR
MR_Empty := false;

//loop invariant: MR.key() <= TR.fKey()
while MR.key() != HIGH_VAL
{
   while MR.key() = TR.fKey()  //process all transactions pertaining to MR
   {
      if TR.isAdd()
         { }                   //Add transactions not yet supported
      else if TR.isChange()
         { TR.change(MR); }    //change MR in accord with TR
      else if TR.isDelete()
         { MR_Empty := true; }   //mark MR as empty
      else
         { }                   //should never happen

      TR := TF.get();           //read next transaction record into TR
   }

   //assert: MR.key() < TR.fKey() (thus, no more transactions pertain to MR)
   if !MR_Empty
      { NMF.put(MR); }   //write MR into new master file

   MR := MF.get();       //read next (old) master record into MR
   MR_Empty := false;
}
```

Figure 3: Program Allowing Valid Delete Transactions

```
TR := TF.get();      //read 1st record of TF into TR
MR := MF.get();      //read 1st record of MF into MR
MR_Empty := false;

//loop invariant: MR.key() <= TR.fKey()
while MR.key() != HIGH_VAL
{
   while MR.key() = TR.fKey()    //process all transactions pertaining to MR
      Apply_Transaction();       //apply current transaction
      TR := TF.get();            //read next transaction record into TR
   }

   //assert: MR.key() < TR.fKey() (thus, no more transactions pertain to MR)
   if !MR_Empty
      { NMF.put(MR); }         //write MR into new master file

   MR := MF.get();       //read next (old) master record into MR
   MR_Empty := false;
od;
```

Figure 4: Program Allowing Some Invalid Transactions

```
Apply_Transaction:
   if TR.isAdd()
      { }                  //Add transactions not yet supported
   else if TR.isChange() {
      if MR_Empty  { ERROR: Attempt to change nonexistent record }
      else { TR.change(MR); }   //change MR in accord with TR
   }
   else if TR.isDelete() {
      if MR.isEmpty() { ERROR: Attempt to delete nonexistent record }
      else { MR_Empty := true; }   //mark MR as empty
   }
   else
      { }   //impossible
```

Figure 5: Initial Version of `Apply_Transaction`

## 3.4 Admitting Add Transactions

The next generalization is to allow **add** transactions. We have left them until last because they are the most difficult to handle.

For the moment, let's allow only those having a key equal to that of some record in the (old) master file.

In this context, it is clear that an **add** transaction is invalid unless the last valid transaction preceding it caused the master record with the same key to be deleted, thereby causing `MR_Empty` to become **true**. (Otherwise, we would be attempting to add a master record having the same key as a master record that already exists.) This leads us to augment our initial version of `Apply_Transaction` to arrive at the one in Figure 6. The result of the invocation `TR.add()`, where `TR` is an add transaction, is assumed to be the master record that `TR` specifies is to be added.

```
Apply_Transaction:
   if TR.isAdd() {
      if MR_Empty { MR := TR.add();  MR_Empty := false; }
      else  { ERROR: adding a record with same key as existing record }
   }
   else if TR.isChange() {
      if MR_Empty { ERROR: Attempt to change nonexistent record; }
      else { TR.change(MR); }    //change MR in accord with TR
   }
   else if TR.isDelete() {
      if MR_Empty { ERROR: Attempt to delete nonexistent record }
      else { MR_Empty := true; }    //mark MR as empty
   }
   else {
      ERROR: Invalid transaction type
   }
```

Figure 6: Second Version of Apply_Transaction

## 3.5 Achieving Full Generality

The last step in our development is to eliminate the restriction that every transaction record must specify the key of a master record appearing in the (old) master file. That is, we wish to allow transactions pertaining to records that do not occur in the old master file. Such transactions are needed in order to add —and subsequently to change or delete— records with keys that do not exist in the old master file.

The most striking consequence of relaxing this restriction is that the condition `MR.key()` $\leq$ `TR.fKey()` is no longer an invariant of the outer loop. To demonstrate this, suppose that the transaction file contains records with keys

..., *Emily*, *Frank*, ...

and that the (old) master file contains records with keys

..., *Emily*, *George*, ...

Then it will necessarily happen that the *George* record will be read into `MR` as the last step of the (outer) loop iteration during which the *Emily* master record was processed. At that time, `TR` will contain the transaction pertaining to *Frank*. Thus, as the next iteration (of the outer loop) begins, we have `MR.key()` = *George* and `TR.fKey()` = *Frank*, which means that `MR.key()` $>$ `TR.fKey()`.

Continuing our example, suppose that the *Frank* transaction in `TR` is an **add**, and let $f$ be the master record that it says is to be added. Because one or more of the transactions that follow may also pertain to Frank, it would be premature to write $f$ into the new master file. Nor would it be correct to place $f$ into `MR`, read the next transaction, and then let execution flow to the inner loop, where any remaining *Frank* transactions would be applied. This approach is flawed because, by filling `MR` with the contents of the new *Frank* master record, we will have lost the contents of the old *George* master record. Indeed, one might say that the *George* record never should have been read in, due to its key being larger than that of the current transaction. But this puts us in a quandry, as there is no way to determine, without reading the next master record, whether its key is larger than that of the current transaction. That is, it is impossible to determine whether it is appropriate to read the next master record until after we have done so!

To overcome this, we invent a mechanism by which to "undo" the effects of the most recent invocation of `MF`'s `get()`. We call this operation, naturally, `unGet()`. To be more precise, if a record $r$ is returned by `MF.get()` and then we invoke `MF.unGet()`, the next invocation of `MF.get()` returns $r$ once again. (We leave it as an exercise for the reader to implement `unGet()` to meet this specification. It is not difficult.)

Assuming such a mechanism is in place, we insert an **if** statement at the beginning of the body of the outer loop. Its purpose is to determine the smaller among the key of the current master record and the foreign key of the current transaction, to assign that value to a new variable, `lesserKey`, and to "unget" the current master record if its key is strictly larger. Following that is the nested loop, the purpose of which is to apply all transactions having foreign key equal to `lesserKey`.

Because `MR.key()` $\leq$ `TR.fKey()` is no longer an invariant of the outer loop, we must, in addition to the modifications just described, weaken the outer loop's guard to allow for the possibility that all records in the old master file have been processed but that more transactions remain. We assume, as we did with the master file, that the last record in the transaction file is a sentinel.

Thus, we arrive at the final version of the main program, which is in Figure 7. This algorithm is but one variation of the classic solution for SFU, which is often referred to as the **Balanced Line** algorithm. In his seminal monograph, *A Discipline of Programming* (Prentice-Hall, 1976), E.W. Dijkstra attributes this algorithm to his colleague Wim Feijen.

```
TR := TF.get();      //read 1st record of TF into TR
MR := MF.get();      //read 1st record of MF into MR
MR_Empty := false;

while MR.key() != HIGH_VAL  ||  TR.fKey() != HIGH_VAL
{
   if MR.key() <= TR.fKey()
      { lesserKey := MR.key(); }
   else {
      lesserKey := TR.fKey();
      MF.unGet();  MR_Empty := true;
   }

   while TR.fKey() = lesserKey {  //process transactions with fKey = lesserKey
      Apply_Transaction();        //apply current transaction
      TR := TF.get();             //read next transaction record into TR
   }

   //assert: lesserKey < TR.fKey()
   if !MR_Empty
      { NMF.put(MR); }            //write MR into new master file

   MR := MF.get();            //read next (old) master record into MR
   MR_Empty := false;
}
```

Figure 7: Program Solving General SFU Problem