

# The Development of an Algorithmic Solution to the Sequential File Update Problem

Robert McCloskey

October 29, 2007

## 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).<sup>1</sup> 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 *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

---

<sup>1</sup>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.

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*<sup>2</sup> by which its records are uniquely identified and ordered (in ascending order, under some suitable definition of “greater than”).

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 of the master file<sup>3</sup> 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), 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.
- (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).

---

<sup>2</sup>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).

<sup>3</sup>Following the terminology of relational database theory, we refer to this as the transaction’s *foreign key*.

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	Kind	Cust-ID	Amount
Alan	40	...	Change	Helen	34
Beth	27	...	Change	Cathy	-27
Cathy	0	...	Add	George	
Emily	74	...	Change	Emily	-40
Frank	-12	...	Change	George	83
Helen	77	...	Delete	Emily	
Iggy	0	...	Change	George	-19
Jack	14	...	Change	George	32
Kim	-54	...	Change	Jack	-14
			Delete	George	
			Delete	James	
			Change	George	20
			Add	Emily	

Accounts Receivable  
Master File

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 apply it. 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 master record with a different key, 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 master record, we apply them in the proper order. (To keep things simple, we will assume here that the proper order in which to apply transactions in the same group 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<sup>4</sup>—using the foreign key field as the sort key.<sup>5</sup> This places transactions with the same foreign 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 what has been called (e.g., by Folk, Zoellick, and Riccardi in their *File Structures* text) *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.

### 3 Development of an Algorithm

As mentioned above, the task of applying the (sorted) transactions to the master file is similar to computing the intersection (or union, or difference, etc.) of two ordered sequences, such

---

<sup>4</sup>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.

<sup>5</sup>If each transaction record contains a *time stamp* field, we can use it as the secondary sort key and not worry about whether the sorting algorithm is stable.

as occurs in the classic *MergeSort* algorithm. Thus, let us explore the possibility of obtaining a solution for SFU by using, as a foundation, the algorithmic template for combining (i.e., merging, intersecting, etc.) two ordered sequences. Particularizing the template so that it produces a file *h* containing the union of the items contained in the files *f* and *g*<sup>6</sup>, we get Figure 2.

```

h.open(output);           // open h for output
f.open(input);  g.open(input); // open f and g for input
x := f.get();  y := g.get();  // read first items from f and g

while ( x != eofSentinel || y != eofSentinel )
{
  if ( x < y )
    { h.put(x);  x := f.get(); }

  else if ( x > y )
    { h.put(y);  y := g.get(); }

  else // x = y
    { h.put(x);  x := f.get();  y := g.get(); }
}

```

Figure 2: Computing the “union” of two ordered lists

We are assuming that `eofSentinel` is a value larger than all others and that `get()` returns it when the sequence to which it is applied is exhausted. In order to obtain algorithms for other operations, slight adjustments must be made. For example, for intersection, we would remove the first two invocations of `put`. For others, we might omit one of the two invocations of `get` in the  $x = y$  case.

Let  $f_{\text{past}}$  and  $g_{\text{past}}$  refer to the portions of *f* and *g*, respectively, that have already been processed (i.e., that precede *x* and *y*, respectively). The key to why the above is correct is that the following is an invariant of the loop (i.e., is true immediately before and after each iteration):

- (1) *x* is greater than or equal to all items in  $g_{\text{past}}$ ,
- (2) *y* is greater than or equal to all items in  $f_{\text{past}}$ ,
- (3) The current contents of *h* is the union of  $f_{\text{past}}$  and  $g_{\text{past}}$ , and
- (4) If  $x = y$ , then the number of occurrences of *x* in  $f_{\text{past}}$  equals the number of occurrences of *x* in  $g_{\text{past}}$ .

We assume this to be an invariant of the loop and leave it as an exercise for the reader to verify. Upon termination of the loop, we have  $f = f_{\text{past}}$  and  $g = g_{\text{past}}$ . From the fact that (3) holds at the end of each iteration (including the last one), it follows that, upon termination of the loop, *h* is the union of *f* and *g*, as desired.

Now let’s try to adapt the algorithm to the SFU problem. As a first step, we rename the

---

<sup>6</sup>More precisely, for each value *z* occurring in either *f* or *g*, the output file *h* will end up containing *k* copies of *z*, where *k* is the maximum among the number of occurrences of *z* in *f* and *g*. For example, if the value 363 occurs in *f* thrice and in *g* once, it will occur in *h* thrice.

variables. Rather than using `f`, `g`, and `h` to refer to the three files, we shall use `mastFile`, `transFile`, and `newMastFile`, respectively. In place of using `x` and `y` to refer to the current records from the two input files, we shall use `mastRec` and `transRec`, respectively. Also, we assume that `key()` is a method that, when applied to a master record, yields **ordering** and **key** information. That is, no two master records yield the same value when `key()` is applied, and each master record yields a greater value than its predecessor. When `fKey()` is applied to a transaction record, it yields a value identifying the master record to which the transaction pertains (i.e., the transaction's foreign key).

Translating the algorithm above, line for line, and numbering those lines that seem suspicious, we get Figure 3.

```

newMastFile.open(output);
mastFile.open(input);  transFile.open(input);
mastRec := mastFile.get();  transRec := transFile.get();

while ( mastRec.key() != eofSentinel  ||  transRec.fKey() != eofSentinel )
{
  if ( mastRec.key() < transRec.fKey() ) {
    newMastFile.put(mastRec);    // (1)
    mastRec := mastFile.get();
  }
  else if ( mastRec.key() > transRec.fKey() ) {
    newMastFile.put(transRec);  // (2)
    transRec := transFile.get();
  }
  else { // mastRec.key() = transRec.fKey()
    newMastFile.put(mastRec);   // (3)
    mastRec := mastFile.get();  // (4)
    transRec := transFile.get();
  }
}
mastFile.close();  transFile.close();  newMastFile.close();

```

Figure 3: Preliminary algorithm for SFU

Line (2) doesn't even make sense, as a transaction record would not be of the correct form to write into the new master file. As for (3) and (4), they don't seem appropriate, because what we would expect here is for the current transaction record, `transRec`, to be applied to the current master record, `mastRec`, possibly changing the latter (in the case of a *change* transaction). In this case, we want neither to write nor read a master record, because the next loop iteration might call for another transaction to be applied to the same master record, updating it further. Line (1) seems right, however, because at this point all transactions pertaining to the current master record have been processed already.

It seems that we have a pretty good handle on things, at least under the assumption that all transactions are of the *change* variety. We propose the program in Figure 4:

We have assumed that the `change()` method, applied to a master record, changes the contents of that master record in accord with the transaction record passed to it as a parameter. (For such an invocation to make sense, the transaction record must be of the *change* variety.)

```

newMastFile.open(output);
mastFile.open(input);  transFile.open(input);
mastRec := mastFile.get();  transRec := transFile.get();

while ( mastRec.key() != eofSentinel  ||  transRec.fKey() != eofSentinel )
{
  if ( mastRec.key() < transRec.fKey() ) {
    newMastFile.put(mastRec);    // No more transactions apply to
    mastRec := mastFile.get();   // current master rec, so write it
  }                               // and fetch next one.

  else if ( mastRec.key() > transRec.fKey() ) {
    print "invalid change transaction: no corresponding master record";
    transRec := transFile.get();
  }

  else { // mastRec.key() = transRec.fKey()
    mastRec.change(transRec); // change mastRec in accord with transRec
    transRec := transFile.get();
  }
}
mastFile.close();  transFile.close();  newMastFile.close();

```

Figure 4: SFU with only Change transactions

Feeling comfortable with handling *change* transactions, let's now extend the algorithm, leaving blanks where appropriate, to allow transactions of any kind. See Figure 5.

We have assumed the existence of boolean methods `isAdd()`, `isChange()`, and `isDelete()`, that, when applied to a transaction record, can be used to identify which kind of transaction it represents.

A *change* transaction is invalid when `mastRec.key() > transRec.fKey()`; for similar reasons, a *delete* transaction is invalid in that case, too. This allows us to fill in (2). Somewhat similarly, an *add* transaction is invalid when `mastRec.key() = transRec.fKey()`, because in this case there already must be a master record with the indicated key. Hence, we know how to complete (3).

Regarding (4), one way to carry out a deletion is to advance to the next master record. This sounds almost too simple to work, but indeed it does.

Somewhat more difficult is the task of filling in (1). The situation here is that the current transaction record, `transRec`, indicates that a master record with a smaller key than the current master record, held in `mastRec`, is to be inserted into the new master file. The obvious solution is to form the new master record and to write it into the new master file. But this is incorrect, as there may be one or more subsequent transactions that pertain to the newly-added record, and all these must be applied before we write the new record.

The next obvious solution is to form the new master record, place it into `mastRec`, and allow subsequent iterations of the loop to handle any transactions that may apply to the new record. But this is incorrect, because the "original" contents of `mastRec` are lost! For example, suppose

```

newMastFile.open(output);
mastFile.open(input); transFile.open(input);
mastRec := mastFile.get(); transRec := transFile.get();

while ( mastRec.key() != eofSentinel || transRec.fKey() != eofSentinel )
{
  if ( mastRec.key() < transRec.fKey() ) {
    newMastFile.put(mastRec);    // No more transactions apply to
    mastRec := mastFile.get();  // current master rec, so write it
  }                               // and fetch next one.

  else if ( mastRec.key() > transRec.fKey() ) {
    if ( transRec.isAdd() )
      { (1) }
    else if ( transRec.isChange() )
      { print "invalid change transaction: no corresponding master record"; }
    else // transRec.isDelete()
      { (2) }

    transRec := transFile.get();
  }

  else { // mastRec.key() = transRec.fKey()
    if ( transRec.isAdd() )
      { (3) }
    else if ( transRec.isChange() )
      { mastRec.change(transRec); }
    else // transRec.isDelete()
      { (4) }

    transRec := transFile.get();
  }
}
mastFile.close(); transFile.close(); newMastFile.close();

```

Figure 5: Incorporating other kinds of transactions

that consecutive records of the master file have as keys *Ernie*, *Gina*, and *Helen* and that consecutive transaction records pertain to *Ernie* and *Frodo* (the hobbit), the latter being an *add*. At some point during execution, the current master record will be that of *Gina* and the current transaction will be (the first) one pertaining to *Frodo*. If, at this point, we overwrite `mastRec` with the new *Frodo* record, the contents of the *Gina* master record will have become irretrievable!

How can we overcome this? The answer is that we invent a mechanism by which to *unget* a master record! That is, we insist that the master file should have a method, `unGet()`, that, when applied, leaves it in a state such that the next time `get()` is applied, the same record as was returned by the previous application of `get()` is returned once more! To implement this method is really quite simple, as all that is needed is a one-record buffer (for storing the “ungotten” record, if any) and a boolean variable indicating whether or not the buffer is (logically) empty.

Assuming the existence of a method `formMasterRec()` that, when applied to an *add* transaction, yields the master record that it says is to be added, we obtain the final version of the program, which is in Figure 6.

```

newMastFile.open(output);
mastFile.open(input);  transFile.open(input);
mastRec := mastFile.get();  transRec := transFile.get();

while ( mastRec.key() != eofSentinel  ||  transRec.fKey() != eofSentinel )
{
  if ( mastRec.key() < transRec.fKey() ) {
    newMastFile.put(mastRec);    // No more transactions apply to
    mastRec := mastFile.get();  // current master rec, so write it
  }
                                // and fetch next one.

  else if ( mastRec.key() > transRec.fKey() ) {
    if ( transRec.isAdd() ) {
      mastFile.unGet();
      mastRec := transRec.formMasterRec();
    }
    else if ( transRec.isChange() ) {
      print "invalid change transaction: no corresponding master record";
    }
    else { // transRec.isDelete()
      print "invalid delete transaction: no corresponding master record";
    }
    transRec := transFile.get();
  }

  else { // mastRec.key() = transRec.fKey()
    if ( transRec.isAdd() )
      { print "invalid add transaction: duplicate key"; }
    else if ( transRec.isChange() )
      { mastRec.change(transRec); }
    else { // transRec.isDelete()
      { mastRec := mastFile.get(); }

      transRec := transFile.get();
    }
  }
}
mastFile.close();  transFile.close();  newMastFile.close();

```

Figure 6: Final Algorithm for SFU