Software Change Document
Augment the Fragment Clear Range

Summary
This project is to augment the fragment store to hold a modified clear range. Currently, the fragment store holds one clear range (integer start & integer stop) per fragment. Until now, the clear range has been immutable. Now, two subsystems (Overlapper and Scaffolder) want to calculate better clear ranges and pass on the improvements to downstream components.

Project resolution is given at end of document.

Project History
9/14/01 – Start requirements gathering (Jason)
9/19/01 – Start document (Jason)
10/01/01 – 1st Review (Jason, Clark[2], Granger, Ian, Mike, Randall, Zhongwu).
10/4/01 – 2nd Review (Jason, Clark[2], Granger, Karin, Mike, Randall, Art). Changes in purple.
10/09/01 – Management decision: finish this fast to accommodate upcoming human run.
10/10/01 – Decide on implementation, adding fields to existing file.
10/11/01 – Broadcast issues to Assembly Team, get comments.
10/15/01 – Regression test fails during CGW on Chrom21.
10/17/01 – Problem ascribed to external cause. Regression tests pass.
10/19/01 – Code release. See last appendix.

Project Background
A fragment's clear range represents the maximum contiguous span of sequence that the genome assembler should trust. Sequence outside the clear range may contain sequencing artifacts such as low-quality base calls or contaminant or vector. Pre-Assembly (Anna Glodek's group) marks clear ranges using heuristic algorithms.

Recent research by Mike Flanigan indicated that Scaffolder could reliably extend some clear ranges. That project found many fragments whose 3' ends extended into small gaps in an assembly, and whose masked sequence (beyond the clear range end) aligned significantly. Now, there is management interest in "closing gaps" by extending clear ranges in this way.

It has always been anticipated that the genome assembler might output modified clear ranges. Therefore, there is already some code to support this. But clear ranges have never actually been modified previously. Managers of post-assembly processes need to be notified of the impending change [Zhongwu was notified 10/1.]

For instance, Terminator has code to handle modified clear ranges. It converts Scaffolder's IFG message to AFG messages (Scaffolder unconditionally creates one IFG per fragment). Terminator is coded to copy the IFG clear range to the AFG, unless a (-1) is in the IFG clear range, and in that case, to look up the (presumably modified) clear range from the frag store instead. In practice, Scaffolder always writes (-1) to the IFG, and Terminator always copies the clear range from the frag store to the AFG.

Project Requirements
1. Subsystems must be able to modify a fragment's clear range.

2. Retain the original clear range, and each revision to it. For immediate future, we must store original, OVL, and CGW clear ranges. For longer term, we might need clear ranges for CNS and one other. Thus, store 4 additional clear ranges.
    a. Overlapper (OVL)
    b. Scaffolder (CGW)
    c. Consensus (CNS)
    d. One other subsystem, as yet undefined, from anywhere in the pipeline.
3. Subsystems must be able to read the most current clear range per fragment. Most current is currently defined as CGW>>OVL>>original. However, the definition could change if other clear ranges get defined. For example, CGW>>CNS>>OVL>>original.
4. Subsystems must be able to specify which clear range they want, and get it.
5. Allow subsystems to write modified clear ranges on or before Oct 19, 2001.
6. Post-assembly processes need notification of each clear range modification.
    a. Terminator (TER) needs to recognize modified clear ranges.
    b. For those fragments whose clear range changes, Terminator must write an augmented fragment message (AFG) in each assembly's ASM file.
7. Permit assembly runs involving {stop, analyze, fix, backup, rerun}.
    a. Permit reset frag store to previous clear range. [Thanks Ian!]
    b. Permit historical analysis (what range existed when?) [Thanks Clark!]
    c. Permit test runs (retry Unitigger without destroying frag store).
    d. Permit test runs simultaneous with production runs.
8. Do not significantly increase assembler resource consumption (CPU time, RAM).
9. Do not break existing programs or utilities.
10. Until now, FragStore was read-only for every subsystem except PopulateFragStore. If FragStore gets multiple writers, then we must retain the ability to multi-process. For instance, we can now run parallel Scaffolders with different options against the same FragStore.

Test Plan

(Test 1) Verify that code changes didn't break programs. Perform a 'cvs update' at the start of development. Record the time (so the baseline code could be checked out again later). Perform all the baseline runs (run the assembler and any utilities) before coding. Insulate this code from simultaneous changes by others; since this is a short cycle, just avoid 'cvs update' during the development. After regression tests pass, perform the next 'cvs update'. If other people's changes were merged in, regression test again before any 'cvs commit'.

Regression test defined. Use the assembler.pl script. Run assembler to completion with before & after codebase. Expect the <project>.asm file to have no significant differences (other than date, pathname, etc.) Repeat the regression test for each of these data sets.

- a0005
- Chrom 22

Include these utilities.

- populateFragmentStore
- partitionFragmentStore_
- Which others?

(Test 2) Verify that code changes were effective, that OVL and CGW could write new clear ranges. Just run a test program that pokes in new values and reads them back. Verify that a value poked in and then retrieved, but also that no other fields (in ReadStruct, for example) were inadvertently altered.

(Test 3) Verify that the Post-Assembly team can handle the new values. How?

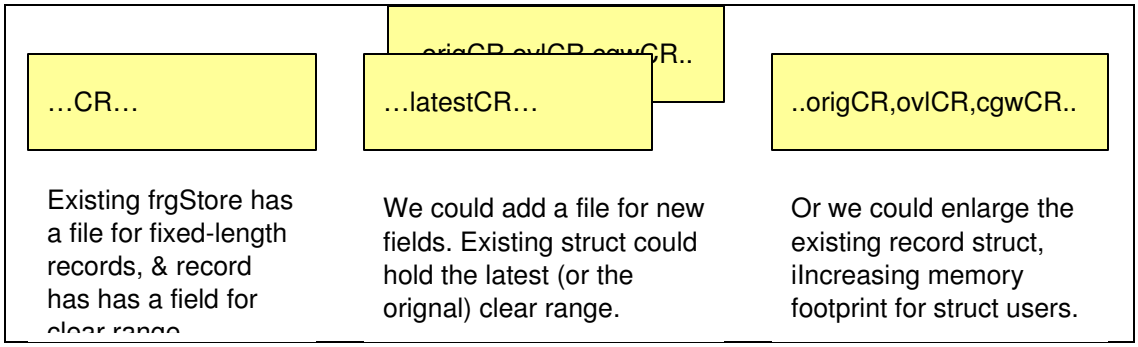
Fragment Store Background

The clear range is stored in the Fragment Store. The store is a directory, named <project>.frgStore, containing a fixed number of binary, record-oriented files. Access to the store is through an API in the C language. The API exposes about a dozen iterator, accessor, and mutator functions. The API does not expose the underlying FragRecord struct. It appears that no client code violates the API by accessing the struct directly (no code imports the header AS_PER_fragStore_private.h). This API, by Saul Kravitz et al., has withstood the test of time! Thus, the programming ramifications of this change request could be nearly self-contained in the API itself.

The API has a utility (loadFragStore() and loadFragStorePartial()) that copies all or part of a fragment store from disk into memory. Several modules (OVL, CGB, CGW) appear to use this utility. Thus, before enlarging the FragRecord struct, consider memory consumption. The marginal gain would be small: compare an extra 16 bytes to {the 40 bytes in the exiting ShortFragRecord plus ~1000 bytes for the sequence and source}. Clark says most systems load into memory only the ShortFragRecord, going to disk for the sequence, quality, and source. Thus, the marginal gain is significant (almost half). However, Karin says it is trivial to make the partitions smaller.

The API has utilities to partition and index the frag store. These utility programs must continue to work under the new scheme.

The figure below shows some ways the fragment store could be altered. All the proposals that



follow employ one of these options.

Proposal #4 (advocated by 10/1 review team)

Increase the size of the ShortFragmentRecord struct so that it contains

- unsigned int-16 : original clear range start & end (as before)
- unsigned int-16 : ovl clear range start & end
- unsigned int-16 : cgw clear range start & end
- unsigned int-16 : cns clear range start & end
- unsigned int-16 : other clear range start & end
- 4 bits: which clear ranges were modified

Thus, all clear ranges will co-locate on disk (in the <project>.frg file in the frgStore directory). Unfortunately, all three will co-locate in memory after a call loadFragmentStore().

- Advantage: Minimal impact on utilities such as partitionFragStore [thanks Clark!]
- Advantage: No additional files or maintenance concerns.
- Advantage: A backup of the <project>.frg file would suffice for assembler reruns.
- Disadvantage: Record size grows. But we assert that the (2 bytes/int) * (2 ints/range) * (4 ranges per frag) = 16 bytes per fragment increase will not whack any program.

Modify the API such that client code explicitly sets which clear range it wants, such as OVL. Make it valid to choose "the latest one." For the meaning of latest, use the same ordering proposed in the next section. Do not offer default behavior -- force clients to make a valid choice, and assert otherwise. Preferably, clients specify their choice once, perhaps in the ReadStruct, instead of on every getClearRange_ReadStruct(). [Thanks Ian!]

- Disadvantage: Incompatible with existing API. All frag store client code must be modified. If we add a parameter to new_ReadStruct(), then 70 lines of client code must change. The abort behavior should help find all the client code that needs fixing.
- Advantage: Reruns are not special cases. Unitigger & Scaffolder would simply ask for the OVL clear range (and write the CGW clear range) as always.
- Disadvantage: Scaffolder reruns must reinitialize the CGW clear ranges, or overwrite the CGW clear range on EVERY fragment. Otherwise, writes from previous runs will persist.

Insure that every value is initialized and always remains valid. Inside the API, automatically propagate new clear ranges up. Define some ordering in code, for example, ORIGINAL=100, OVL=200, CGW=300, CNS=400, OTHER=500. (The ordering could change between runs.) Then, whenever a low-order clear range is written, propagate it to the high-order ranges. For the ordering shown, OVL clear ranges would overwrite existing OVL, CGW, CNS and OTHER values. (Note we never store an invalid range in any column, as other proposals do.)

Provide a utility for restarting an assembler subsystem. For instance, if Scaffolder were to run a 2nd time, the utility would copy the previous (OVL) clear ranges over the CGW (and automatically all subsequent) ranges.

Initialize the disk store by copying the original clear range into the OVL and CGW columns. Thus, a call to getClearRange(fragIID, OVL) would return the original value if Overlapper had never updated this fragment. Use 2 of the 5 "spare" bit-flags in ShortFragRecord to indicate whether OVL and CGW did modify the clear ranges (or just test for inequality of the clear ranges).

- Advantage: Access time is optimal, avoiding if…then logic in the accessor.
- Disadvantage: The OVL and CGW values must be initialized when the fragment store is created, and reinitialized before reruns. This is a maintenance concern.

Ideally, multiple iterators could exist in one program. A future utility program could open the frag store twice, each time with a different clear range selected. Hopefully, the ReadStruct infrastructure does not use static variables.

The multiple writer problem shall be avoided procedurally. Basically, when processes are run in parallel with the main run, they must provide themselves with a copy of the FragStore. To save disk space, the copy can be partial -- the large read-only files as virtual copies (hard links), and the small read-write files as true copies.

Proposal #5 (suggested on 10/4, but not loved)

Add a file to the FragStore that only contains modified clear ranges. It is a write-once, or append-only, file. it contains the fields {frag IID, ordering, new clear range}. The ordering would distinguish OVL from CGW, but it could also distinguish CGW from CGW-parallel-test. All aspects of the API and the ReadStruct remain the same as Proposal #4. Internally, when populating the ReadStruct, the API should use in-memory random access to the clear range file to see if a modified clear range exists.

- Advantage: Very extensible.
- Assumes: Average number of clear range fixes per run is small. The proposal saves space if the assumption holds, but wastes space otherwise.

Proposal #3 (voted down on 10/1, changes infrastructure too radically)

Always update the existing file, overwriting the old values. Thus, the main file would contain the most up to date clear ranges at any given time. Create a "backup" file in the fragment store directory to hold just previous clear ranges. The Overlapper would generate the backup file with these 3 columns:

- Frag IID
- original clear range (as calculated by Pre-Assembly)
- OVL clear range (in case CGW later overwrites OVL's changes in the main file)

To support the Terminator, use a bit on FragRecords (there are 5 "spare" bits) to indicate which records were modified. To support restarts, write a new executable that restores frag store clear ranges record-by-record from the backup.

- Assumes: At least one of the 5 "spare" bits in the FragRecord struct is available to us.
- Assumes: Augmented fragment message (AFG) does not include the original clear range.
- Assumes: Few programs (if any) need the original clear range after OVL clear ranges exist.

- Assumes: Few programs (if any) need the OVL clear ranges after CGW clear ranges exist.
- Advantage: Struct remains the same. Minimum impact on existing code.
- Disadvantage: Overhead for the Overlapper to generate backup file.

Proposal #2 (disliked, double I/O on every record access)

Add files to the fragment store to hold modified clear ranges. An OVL file and a CGW file would contain their respective modified clear ranges. To support restarts, just erase the appropriate OVL or CGW file.

- Assumes: Backward compatibility is important [it is not, says Granger].
- Advantage: Allows compatibility with old stores.
- Disadvantage: Increased I/O (2 files) for all readers of the fragment store.
- Disadvantage: Increased I/O (2 files) for programs that write modified clear ranges.
- Disadvantage: Non-trivial to make Terminator detect modified clear ranges.

This proposal got more respect at the 10/4 review. This design is the most extensible because to add a field you add a file. Also, it does not require enlarging ReadStruct, or therefore, the FragStore memory footprint. It was pointed out that the FragStore's super efficient bulk fread() would only have to become 2 fread()s, since each ReadStruct could be constructed from 2 in-memory arrays. Off-line, Saul backed this proposal strongly because it involves the least amount of code change. Also, Saul feels the I/O penalty would be small.

Proposal #1 (similar to 4, but API remains same; voted down on 10/1)

Add 4 new fields to the fragment struct. A fifth field, representing which clear range is the most recent, can be optimized away: each clear range would be initialized with an invalid value such as (end<start).

- unsigned int-16 : ovl clear range start & end
- unsigned int-16 : cgw clear range start & end

The API would deliver the first valid clear range in the ordering {cgw, ovl, original}. Embed the new code in #ifdefs so old stores can be read with a compiler switch. To support restarts, write a new executable that reinitializes the OVL and CGW modified clear ranges. (Better than modifying the API to optionally deliver the previous values.)

- Assumes: Struct modifications will be hidden by the frag store API.
- Disadvantage: Increases memory consumption for programs that load the entire store.
- Disadvantage: Makes new code incompatible with old fragment stores.
- Advantage: Co-locates related information (all clear ranges per frag).
- Advantage: Trivial to make Terminator detect modified clear ranges.

Appendix 1: FragRecord struct

Defined in AS_PER_fragStore_private.h

```
typedef struct {
  uint   deleted:1;
  uint   readType:8;
  uint   hasQuality:1;
  uint   numScreenMatches:16;
  uint   hasModifiedClearRegion:1;
  uint   spare1:5;
  VLSTRING_SIZE_T clearRegionStart;  // Elsewhere defined as uint16
  VLSTRING_SIZE_T clearRegionEnd;
  uint64 accID;             // UID
  uint32 readIndex;         // IID
  uint32 localIndex;
  uint64 sequenceOffset;
  uint64 sourceOffset;
  time_t entryTime;    // 32-bit
}ShortFragRecord;    // Total size = 320 bits = 40 bytes

typedef struct {
  ShortFragRecord frag;
  uint flags;
  uint64 localeID;
  uint32 localePosStart;
  uint32 localePosEnd;
  char source[MAX_SOURCE_BUFFER_LENGTH];
  char sequence[MAX_SEQUENCE_LENGTH];
  char quality[MAX_SEQUENCE_LENGTH];
  IntScreenMatch matches[MAX_SCREEN_MATCH];
}FragRecord;
```

Appendix 2: How loadFragStore() works

The following code is repeated for each of the files in the fragment store: frg (the fixed length headers), seq (the nucleotide sequences), and src (the “source” or comments).

- loadFragStore() in AS_PER_fragStore.c
- calls loadFragStorePartial()
- calls loadStorePartial()in AS_PER_genericStore.c
  - calls computeOffset(), block size =
  - (lastElem-firstElem+1) * elemSize
- calls copyStore()
- calls readBufFromStore()
- calls SafeRead() in AS_PER_SafeIO.c
- calls fread()
  - bulk read of entire block

Appendix 3: The Fragment Store API

Access to the store is defined in AS_PER_fragStore.h. Here are some functions

- createFragStore()
- createPartionedFragStore()
- createIndexedFragStore()
- openFragStore()
- loadFragStore()

- appendFragStore()
- setFragStore()
- openFragStream()
- nextFragStream()
- getFirstElemFragStore()

Fragment records are always returned in a struct called ReadStruct. Clients (unwittingly) malloc the struct, then pass its pointer to every accessor function. The API hides the definition of ReadStruct, exposing only accessor functions in AS_PER_ReadStruct.h. Here are some accessors

- getClearRegion_ReadStruct(structPtr, intPtr, intPtr)
- getHasModifiedClearRegion_ReadStruct(structPtr, intPtr)

(Looks like that last function was never implemented.)

Appendix 4: API clients

Subsystems where grep finds openFragStore

- ASG
- CGB
- CGW
- CNS
- CVT
- ORA
- OVL
- PER
- REZ
- SDB
- TER
- URT

Subsystems where grep finds loadFragStore

- CGB
- CGW
- CNS
- OVL
- PER

Appendix 5: Resolution

Many implementations were considered, but management selected the easiest to implement, in order to accommodate a rescheduled human assembly. The easiest implementation was to modify the structure of the db.frg file, thus increasing the memory demand on programs that load frag stores into memory.

The implementation was finished quickly, in about 2 days. We did not commit to cvs right away.

Testing on Pyro then Human Chrom 21 took 2 days originally. Investigation of test failure took another 3 days. The assembler crashed during Scaffolder on Chrom21. This is a large data set, and each test requires 4 hours. We last time investigating the problem, mostly because we assumed we needed to debug the running code. It turned out that simply updating my code to cvs cured the problem. At the beginning, we had tested on Pyro but not Chrom21, and we hypothesize that the crash codes pre-existed our work, and were fixed by others while we worked.

Some problems were discovered during the implementation. Those were addressed in the release notes, which follow.

Appendix 6: Release notes

RELEASE NOTES

- **New fragStores have extra fields.** Starting today, you can build fragStores with multiple clear ranges. You get extra fields ONLY IF you build your fragStore after compiling with FRAGSTORE_VERSION=5. Note the extra fields increase storage requirements, and memory requirements for programs that load fragStores to RAM with loadFragStore(). In particular, the fixed-length record file **db.frg** grows from 40 to 64 bytes per record in VERSION 5.
- **The fragStore API changed a tiny bit.** I added some functions, and added one parameter to two existing functions. I changed all the client code as needed. Here are particulars.
  - setClearRegion_ReadStruct() now takes a flag, such as READSTRUCT_ORIGINAL. The flags have this effect.
    - READSTRUCT_ORIGINAL overwrites ORIGINAL, OVL, CNS and CGW clear range.
    - READSTRUCT_OVL overwrites OVL, CNS and CGW clear range.
    - READSTRUCT_CNS overwrites CNS and CGW clear range.
    - READSTRUCT_CGW overwrites CGW only clear range.
  - getClearRegion_ReadStruct() now takes a flag. All the flags cited above are valId, plus READSTRUCT_LATEST, which is the same as READSTRUCT_CGW.
  - hasModifiedClearRegion() is removed from the API. No code was using it.
  - hasOVLClearRegion() is new.
  - hasCNSClearRegion() is new.
  - hasCGWClearRegion() is new.
  - The dumpFrag utility displays all the new fields if compiled with FRAGSTORE_VERSION=5.
- **A new `make` flag controls code/store compatibility.** In cds/AS/src/AS_PER/**Makefile** you will find a new flag called FRAGSTORE_VERSION. Going forward, set this to 5. For backward compatibility with old stores, just leave this at 4.
- **CVS tag.** I created cvs tags AS_ASSEMBLER_BEFORE_FRAGSTORE_VERSION_5 and AS_ASSEMBLER_AFTER_FRAGSTORE_VERSION_5.

FAQ

Q: How do I determine the version of an old fragStore?
A: Try the dumpFrag utility. If the store's version does not equal the code's version, the utility will report both version numbers in its run-time error.
Q: Can I switch fragStore versions at run-time?
A: Nope. If you compile to VER 4 and run on VER 5, or vice versa, you will get a run-time error.
Q: Can I convert an old fragStore?
A: Nope. But you can build a new fragStore from the project.frg file.
Q: How do I set fragStore version in code?
A: Set the FRAGSTORE_VERSION flag in the AS_PER Makefile. I commit the Makefile set to VERSION 4. See the comments in that Makefile.
Q: How do I set clear ranges in code?
A: To update clear ranges on disk, call openFragStore() (not loadFragStore(), which copies to RAM). Then call setClearRegion_ReadStruct() function, using one of the new flags like READSTRUCT_OVL defined in AS_PER_ReadStruct.h. Then call setFragStore() to write that record to disk. This set function uses random access -- fseek() then fwrite(). See demo code in AS_PER/testClearRange.c. Since this demo carelessly modifies a fragStore, I didn't add it to the Makefile.

Programs that modify clear ranges should allow the assembly pipeline to be stopped and restarted on a given fragStore. Thus programmers should think whether they should use the READSTRUCT_LATEST flag -- the latest clear range may have been set earlier by a downsteam component. Here is a suggestion. Overlapper should use get(ORIGINAL) and set(OVL). Consensus should use get(OVL) and set(CNS). Scaffolder should use get(CNS) and set(CGW).
Q: How can I make new set-clear-range-code compatible with VER 4 fragStores?
A: Compatibility is handled for you by the API. Suppose you add the function call setClearRegion_ReadStruct(OVL). When you compile and run with VER 4, the ReadStruct API will ignore the OVL flag and use the only clear range that a version 4 fragStore has -- the original clear range.

If for some reason you absolutely must write code for VER 5 only, then #include AS_PER_ReadStruct.h and use this compiler directive (see AS_PER_fragStore_private.h for an example):

```
       #if  FRAGSTORE_VERSION  >=
VERSION_OF_FRAGSTORE_WITH_MODIFIED_CLEARRANGES
```

Q: Can I set clear ranges in parallel on a partitioned fragStore?
A: Nope. The fragStore API has never supported this. The fragStore has no mechanism to support (or prevent) multiple simultaneous writers. And it can't update partitioned fragStores (some functions would assert, others would corrupt.). Don't try setFragStore()!

I propose we live with this limitation and build a work-around. When Overlapper and Consensus run on LSF, it is better for them to receive read-only fragStore partitions. Action item?: invent some other mechanism whereby LSF-partitioned programs pass clear range modifications to an accumulator that performs serialized updates to the original fragStore.
Q: Can I set clear ranges in RAM?
A: If you do, you won't be able to save those changes to disk. The fragStore API never supported that.

The function loadFragStore() copies your fragStore to RAM. Subsequent update functions, such as setClearRegion_ReadStruct(), write to the RAM version. Functions that might bulk copy RAM back to disk, like storeFragStore(), probably haven't worked for a long time. Action item?: fix this.

Note you can create *NEW* records in memory and *APPEND* them to a (non-partitioned) disk store. See PopulateFragStore for an example.