

# **Changes to P4 Integrate in 2004.2**

**Jeff Anton**

**Senior Software Developer**

**Perforce Software, Inc.**

## **Abstract**

The 2004.2 release of the Perforce Server saw significant changes to the operation of the p4 integrate command. This paper covers Perforce's motivation for the changes, how the command operated previous to these changes, how the changes affect behavior, and what operations will likely affect performance. This will involve some detail regarding how indirect integration credit is computed and how we find indirect integration base revision files.

## Summary of changes

Indirect Integration performance improvement  
Indirect Integration default (and only supported operating mode)  
Indirect Integration does not imply baseless integrations allowed  
Common Ancestor Base determination

## Motivations

Rename  
Indirect crediting – better  
Baseless integration issues

## Old Operation (Direct integrations)

The *p4 integrate* command pairs source files with target files and schedules target files for integration. Prior to 2004.2, the default form of integration was the *direct* form. With direct integration, two filenames in the depot are found via a branch view map. If both files in source-target pair exist, a search of previous direct integrations between the two filenames is performed. If all the revisions of the source had already been *credited* into the target (that is, found to be previously integrated into the target), no integration would be scheduled. (Scheduling an integrate means that the user must use *p4 resolve* before submitting a change list.) If there were not enough credited integrations, a *resolve* between the source and target would be scheduled which would bring changes from the source to the target. The specific revisions to be used as the source and base would be computed from the credits and the source file revision range specified. Those rules are:

- Source revision would be the newest revision in the source revision range (except for previously ignored integrations)
- Base revision would be the revision of the source just prior to the oldest non-credited revision in the source revision range.
- Integrations would be considered ‘baseless’ if the source revision range started with an *add* or *branch* revision and the base revision would be before that *add* or *branch* revision.

(I’m ignoring the delete integration and new branch cases which are different.)

Each scheduled integration which is resolved and then submitted becomes a permanent integration record searched for in later integration commands.

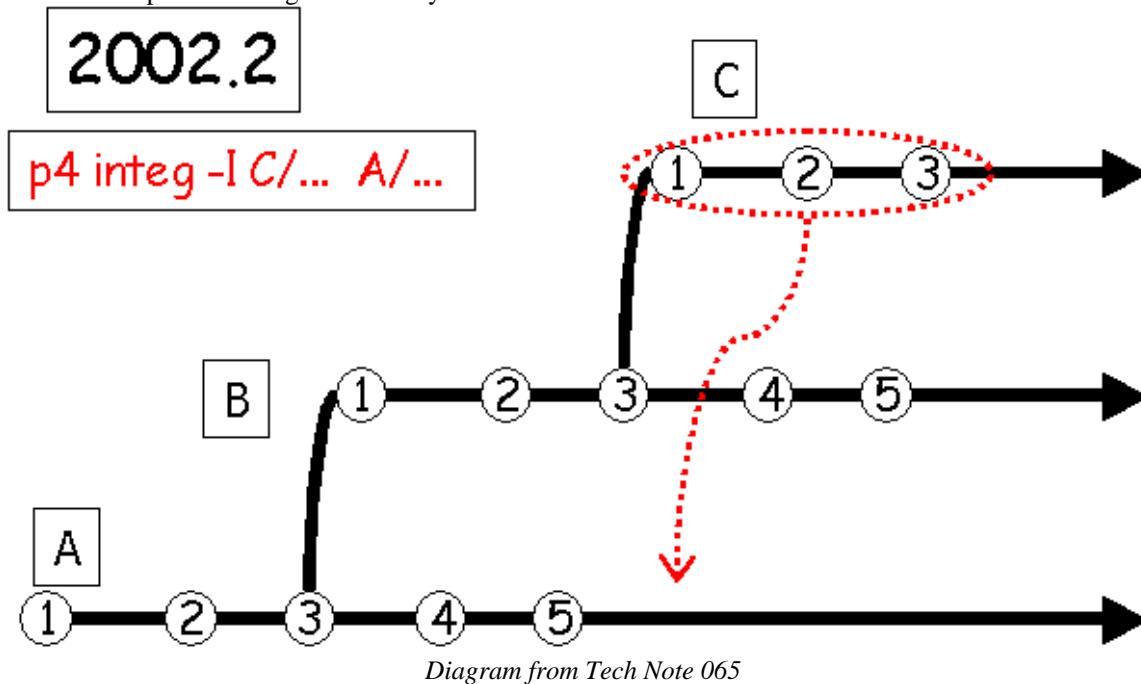
## Cherry Picking vs. Cumulative Integrations

A common integration strategy we call ‘*Cherry Picking*’ is to integrate only a specific change from one code line to another. This strategy involves integrating files with the source having a ‘@*change,@change*’ qualifier (or other narrow change range) and allows many integrations between indirectly related code lines due to the ‘baseless’ rule requiring the source start an *add* or *branch* revision and that is not likely with a narrow range unless a file was added with that change and the target already had a file of that name. (A kind of conflict.) But if the target file is truly not related to the source, the *resolve* of the change will probably not be helpful as the base revision is not really a common ancestor of the target which is a requirement for most file merge systems to work sanely. The idea of ‘*Cherry Picking*’ is to narrowly integrate specific functionality between code lines. This does not need an established relationship between source and target to work, but if there are multiple code lines being managed, duplicate integrations might occur if changes are blindly integrated between code lines. Cumulative integrations however check that two code lines are up to date with all of each others changes and can operate with clearly defined common ancestor base revisions.

## Old indirect flag –I and baseless cases – Tech Note 065

The ‘-I’ flag which enabled indirect integration also enabled baseless integration cases because of a case it did not handle well. That was described in Tech Note 065. The problem here was that the best base is not on the source code line and the system could not handle or find base revisions not on the source code line. Because there were valid indirect integrations which did not have a base on the source line ‘-I’ implied to

allow baseless cases to get this case to work. However allowing all baseless cases would allow integration of two unrelated files to proceed in baseless fashion without warning. This needed to be changed to allow related integrations in which the base was not on the source line while disallowing integrations between files with no previous integration history.



### Rename

With Perforce, renaming files is performed by branching to the new name and deleting the old name. Direct integration could not handle this because crediting could not be properly computed and base revisions would not be on the source file name. It seemed clear that to tackle the problems with rename we would need to move beyond direct integration. Although we had '-I' indirect integration it was too slow to use all the time and did not find the common ancestor base revision. This became the single best reason to make substantial changes to integrate. **To handle renamed revisions we needed to have indirect integration work all the time, be fast, and find the 'right' base.**

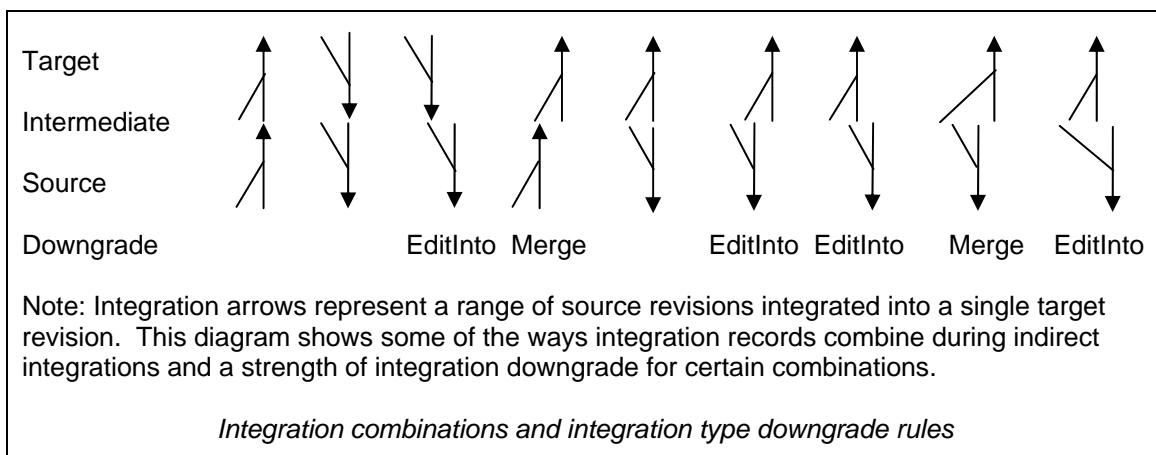
### Speeding up indirect integrations

The old indirect integration method was clearly too slow, internally it followed every integration record and every integration record which overlapped with those etc. By generating every possible indirect integration record the volume of seeking of the db.integed table as well of information and sorting made the first implementation a problem. Further, a bug in the indirect integration crediting logic was found and the fix involved more integration record combinations and was going to almost double the run time, clearly we had to improve performance. Two approaches to improve this were apparent. First, it was clear that when we are combining integration records, once a path reaches between the source and target files we would not need to continue searching further as that would have to not reach back to another path productively. With our test data set in which an indirect integration took 23 minutes with the old code we got the time down to 5 minutes which is significant but not enough. We were still generating all the indirect integration records we could before applying them. The next step was to stop the expansion of indirect integration paths as soon as we know that no integration will be needed. To do this the system was changed to a breadth-first search in which the each level of integration records (direct records) are considered and if all the credits needed to show that no integration is needed then we would end the credit search early. This was a significant improvement because it meant that if the old direct integration would discover that an integration was not needed, then the new indirect system would find that just as fast as the old direct system. Any slower performance would be due to indirect searching. That indirect searching continued in breadth-first fashion with each new depth explored having the chance to stop indirect searching. Once this

system was coded our test data set integrated in 5 seconds. The old direct case took 3 seconds but scheduled some unneeded resolves which would, with the time for automerge to determine that no work is needed and the submit of a no-change integration, easily make the old system more costly than the new indirect crediting.

### Indirect integration combining rules

Indirect integration records are formed internally when the 'from' range overlaps with the 'to' range and other merge conditions involving the kind of integration allow a combination. There are combination rules which set how we type an indirect integration. If both arrows are *copy from* and their end positions line up perfectly, we produce a new indirect copy which identifies that we know indirectly that two revisions are identical. If the end ranges do not line up, we downgrade to either a *merge from* or an *edit from* depending on which end range occurred first. When the second arrow's end range is later it generates a merge because the target will have the changes from the source, but also have some merged in changes from the intermediate branch. The following diagram shows some of the combinations of integration records and the corresponding downgrade for that case.



Strength of integration types (from Strong to Weak):

(When different integration types are combined indirectly the weaker type survives)

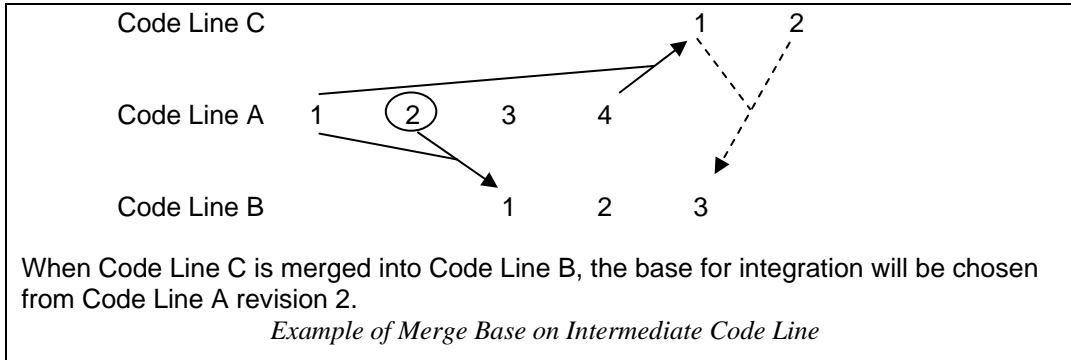
- Copy, Branch
- Merge, Ignore
- Edit into, Edit from, Add Into

For example a copy integration combined with a merge integration will be treated as a merge integration. This is important because a copy integration is synchronizing and establishes that all past changes are contained in the target, a merge integration however only credits a specific range into the target.

### Base finding

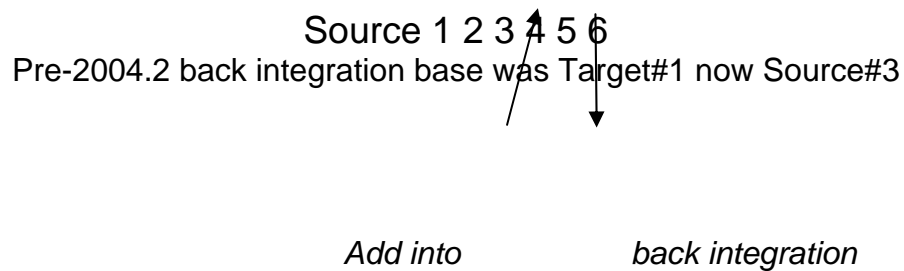
Because base finding is now more complicated, all commands which would know what file revision would be the base of an integration report that base file revision when the *-o* flag is given. (i.e. *p4 integrate*, *p4 resolve*, and *p4 resolved*)

When there is an indirect integration credit, the combination of integration records indicates a possible base revision. While the rules are complicated, the base revision will likely be on the source file. The base revision will come from the last integration credit before the oldest non-credited revision in the source revision range.



### Dirty Branching – Special Base case

A small but significant new behavior occurs with ‘dirty branching.’ This occurs when a file is branched via the *p4 integrate* command, but then that newly created file is re-opened for edit via *p4 edit*. This creates an *add into* integration record when the file is submitted. In this case an integration back into the source of that branch will now use a base on the original source – now the target. Even the undocumented ‘-1’ direct integration support will now choose a base on the target line in this case.



### *Dirty branch base change*

### Potential Problems

#### Cherry Picking costs

Frequent cherry picked integration will produce many more integrate records than cumulative integrations. Those extra records will have to be examined on subsequent integration commands. (This assumes that cumulative integrations between branches are not happening automatically or on every change.)

### **Expensive Integrations (Copyright or Name Change example)**

Because the new indirect integration system uses breadth first search, when a resolve is found necessary the integrate command will have done an exhaustive search though all integration records which can be found indirectly which might connect the two files being integrated. This produces the following problem case. Say there is a main source code line and every file is opened for edit and changed to update the copyright year or change a company name and then all that is submitted. Now as each branch which integrates from that mainline will cause an exhaustive search will have to search every integration record indirectly. This can be very slow because the indirect breadth first search can never terminate early. The undocumented '-1' flag to force direct integration and 'cherry picking' may be desirable in this case, but would once the change is merged and submitted, it would not be needed.