ChatGPT

# Rediseño no destructivo del sistema de evaluación para File Nodes en Blender

## Enfoques no destructivos en addons complejos de Blender

En las herramientas nodales de Blender (tanto nativas como addons), es fundamental **evitar modificar directamente los datos originales** para mantener un flujo de trabajo no destructivo. La práctica recomendada es trabajar con *copias* o referencias temporales de los datablocks, de forma que los cambios puedan aplicarse y revertirse sin corromper el estado base. Por ejemplo, addons de modelado procedimental como *Sorcar* enfatizan la posibilidad de **ajustar parámetros en cualquier etapa sin perder la malla original**, permitiendo variar el resultado de forma ilimitada y sin destrucción de datos. Esto típicamente se logra manteniendo la malla original intacta y aplicando las operaciones sobre una versión duplicada o de vista previa. Del mismo modo, en *Animation Nodes* (otro addon nodal popular), las transformaciones de objetos se calculan en cada frame sin "quemar" esos valores en el objeto original, asegurando que al desactivar el nodo, el objeto retiene su estado inicial. En general, **guardar el estado original** (ya sea duplicándolo o registrando sus propiedades) antes de aplicar cambios, y luego operar sobre una copia, es una técnica fundamental para lograr no destructividad en addons avanzados.

Cuando un addon necesita *manipular temporalmente datablocks* (escenas, objetos, etc.), suele haber dos estrategias: (1) **Aplicar cambios y luego revertirlos** (como hace actualmente File Nodes guardando y restaurando propiedades), o (2) **Usar datos alternativos (copias o instancias)** para visualizar las modificaciones sin tocar el original. La segunda estrategia tiende a ser más robusta a escala, ya que reduce el riesgo de que una restauración falle y evita contaminar el original durante la evaluación. Algunos desarrolladores crean objetos "fantasma" o colecciones temporales que sirven de *proxy* para los originales: por ejemplo, duplicar un objeto y aplicar los nodos sobre el duplicado mientras el original se oculta. Así, el duplicado modificado muestra el resultado en pantalla, y al terminar se puede descartar sin consecuencias. Esta aproximación minimiza la necesidad de guardar cada propiedad individual, simplificando la reversión (basta con eliminar o desvincular el duplicado). En resumen, la **manipulación no destructiva** en addons complejos suele apoyarse en copiar datos (o usar instancias enlazadas) y limitar las escrituras directas sobre los ID originales, empleando estos últimos solo como *entrada* y no como lienzo de pintura.

## Consistencia y rendimiento con múltiples modificadores apilados

Cuando se encadenan múltiples modificadores que afectan a los mismos datos, mantener la consistencia y la eficiencia se vuelve desafiante. En el diseño actual de File Nodes, todos los modificadores se recalculan secuencialmente desde el estado original en cada evaluación, restaurando primero la escena base y aplicando luego cada gráfico uno por uno. Esto garantiza determinismo (el resultado siempre es reproducible dado el mismo input) a costa de recalcularlo *todo* en cada cambio. Una mejora importante aquí es aprovechar **evaluaciones incrementales**: idealmente, recalcular solo los nodos o mods afectados por un cambio, en lugar de recomputar la pila completa siempre. Blender internamente ya funciona así – su *Dependency Graph* invalida y actualiza únicamente las partes del árbol dependientes de la propiedad modificada, dejando intacto lo que no cambió. Adoptar un enfoque similar en el addon aumentaría la escalabilidad: por ejemplo, si varios modificadores actúan sobre

distintos aspectos (uno enlaza objetos y otro cambia el world de la escena), podrían evaluarse parcialmente sin rehacer todo el trabajo.

Otro aspecto crítico es manejar correctamente cuando **varios modificadores tocan el mismo dato**. Debemos definir claramente cómo se compone el resultado: normalmente, en una pila tipo *stack*, el modificador inferior establece una base y los superiores la alteran sucesivamente. Para lograr esto de manera consistente, conviene que cada modificador opere *sobre la salida del anterior*, no sobre el original global. En la implementación actual, esto se logra porque tras aplicar el mod1, sus cambios permanecen en la escena para que mod2 los vea. Sin embargo, el enfoque de duplicados temporales requeriría encadenar también esas copias: es decir, generar un estado modificado intermedio que sirva de entrada al siguiente. Un mecanismo posible es mantener una **estructura de datos caché** por modificador: después de calcular un mod, guardar su *resultado* (p. ej., referencias a los objetos/propiedades ya modificados) de modo que si otro mod subsecuente afecta a los mismos objetos, se parta de la versión actualizada. Este caché evitaría tener que recalcular desde cero toda la secuencia ante cada cambio menor. Además, permitiría *desactivar* un modificador intermedio y recomponer rápidamente el estado combinando los resultados cacheados de los demás.

En cuanto al rendimiento, es crucial evitar operaciones costosas repetitivas. Por ejemplo, el nodo **Read Blend File** actualmente vuelve a cargar librerías .blend en cada evaluación. Esto puede optimizarse implementando un **cache de librerías**: una vez linkados los datos externos, reutilizar esos datablocks en ejecuciones posteriores (mientras no cambie la filepath) en lugar de volver a cargarlos. Blender no elimina automáticamente los datos linkados aunque se desvinculen de la escena, por lo que se podrían mantener en memoria y simplemente relinkear objetos ya importados. Del mismo modo, si un nodo ya calculó una lista de objetos filtrados, almacenar ese resultado evitaría recalcularlo si los inputs no cambiaron. Estas estrategias de caching, junto con la evaluación incremental, alinean el sistema con la filosofía de Blender de "*no actualizar nada que no haya cambiado*", mejorando notablemente la performance en escenas con muchos modificadores.

## Inspiración del sistema Geometry Nodes (evaluación interna)

*Diagrama simplificado del flujo de datos en Blender 2.8+: los datos originales de la escena (DNA data) permanecen inalterados, mientras el grafo de dependencias crea copias evaluadas para aplicar modificadores, constraints y nodos de forma no destructiva [1]. Esto permite tener múltiples estados evaluados conviviendo (por ejemplo, en distintas ventanas o vista previa de render) sin modificar la fuente.*

El sistema **Geometry Nodes** de Blender es un modelo a seguir para lograr una evaluación no destructiva eficiente. Al igual que los modificadores convencionales, un nodo de geometría opera sobre una *copia* de los datos del objeto, generando una **malla evaluada** nueva en vez de alterar la malla original. Internamente, Blender implementa un mecanismo de *copy-on-write*: cada vez que un objeto entra al pipeline de modificaciones, se crea una copia ligera de su estructura de datos (por ejemplo, un contenedor Mesh) donde se aplicarán los cambios. Los datos pesados (como vértices y caras) pueden compartirse entre original y copia hasta que alguna operación los modifique, evitando duplicar en memoria información innecesaria. Este diseño asegura que **ningún cambio se aplica al ID original** en el `.blend` — el grafo de dependencias almacena todos los resultados evaluados por separado [1]. Gracias a ello, es posible, por ejemplo, tener una vista 3D mostrando la malla con Geometry Nodes aplicados, mientras el dato base permanece intacto y puede revertirse al desactivar el modificador.

Otra característica importante de Geometry Nodes es su **evaluación diferida y paralelismo**. Al construir un grafo de nodos puro (sin efectos secundarios), Blender puede calcular nodos en paralelo en muchos casos, y solo materializa los resultados cuando es necesario. Además, el sistema marca las

dependencias: si un nodo de entrada no cambia, los nodos posteriores pueden reutilizar el resultado previo (comportamiento de tipo *cache interno* del grafo). Esto contrasta con la aproximación manual actual del addon, donde se recalcúla todo secuencialmente. Emular aunque sea parte de esta filosofía podría mejorar File Nodes. Por ejemplo, separando la evaluación de cada *FileNodeTree* en su propio contexto (similar a cómo cada modificador de geometría tiene su contexto de evaluación), se aislarían mejor los cambios y se podrían actualizar nodos individuales sin reprocesar toda la pila.

Cabe destacar también cómo Blender maneja la composición de múltiples modificadores: la **salida evaluada de uno es la entrada del siguiente** automáticamente. En File Nodes, lograr esto con copias implicaría que la escena/objetos resultantes de un gráfico se pasen como base al siguiente. Geometry Nodes no modifica la escena global sino que entrega un resultado (una nueva geometría) al siguiente modificador en la cola – extrapolando esa idea, File Nodes podría tratar cada *File Modifier* como una función que toma una escena (o sus datablocks) de entrada y devuelve una escena/colección modificada como salida. Implementado correctamente, esto permitiría apilar varios nodos de archivo sin colisiones, igual que se apilan modificadores de malla.

## Recomendaciones para rediseñar File Nodes (no destructivo)

Integrando lo anterior, a continuación se proponen **enfoques viables** y recomendaciones concretas para rediseñar el sistema de evaluación del addon:

- **Uso de duplicados temporales de datos:** En lugar de modificar directamente los objetos/escena original, crear *copias temporales* sobre las que se apliquen los nodos. Por ejemplo, si un nodo va a cambiar propiedades de `bpy.data.objects["Cube"]`, primero duplicar ese objeto (`obj_copy = original_obj.copy()`) junto con los datos necesarios (`obj_copy.data = original_obj.data.copy()` si hará cambios en la malla), y luego aplicar las modificaciones al duplicado. Finalmente, el duplicado puede vincularse a la escena en lugar del original (ej. enlazar `obj_copy` a las mismas colecciones que tenía el original, ocultando el original). Esto lograría que la **versión modificada** del objeto sea visible en el visor 3D, mientras el original queda intacto en segundo plano. Al desactivar o recalcular, simplemente se eliminaría/eliminarían los duplicados y se mostraría de nuevo el original. Este patrón se puede extender a *Escenas completas*: crear una escena clon de trabajo (mediante `bpy.data.scenes.new()` y copiando linking de los contenidos originales) y aplicar en ella todos los modificadores nodales, manteniendo la escena original sin tocar. Aunque Blender no soporta mostrar dos escenas simultáneamente en la misma ventana, se podría hacer que la vista activa apunte a la escena evaluada (enganchando la cámara y layers equivalentes) para dar la ilusión de continuidad. Si esto resultara muy complejo, la duplicación a nivel de objetos/colecciones específicas afectadas puede ser más granular y fácil de controlar.

- **Empleo de "proxies" u overrides:** Para datos enlazados de librerías externas, considerar usar *Library Overrides* en lugar de modificar directamente los vinculados. Por ejemplo, si `Read Blend File` trae un objeto linkeado y luego un nodo pretende moverlo o cambiarle el material, se podría crear un **override local** de ese objeto (Blender 3.x lo permite) de modo que los cambios queden registrados como anulación no destructiva sobre el link. Esto encaja con la filosofía de no destructividad: el archivo original permanece intocado y Blender sabe aplicar la diferencia (override) solo en el contexto local. Para el addon, podría significar que en vez de almacenar manualmente `original_value` para cada propiedad de un objeto linkeado, se delega al sistema de overrides mantener esa relación original–modificado. No obstante, las librerías override con muchos cambios pueden agregar complejidad, por lo que esta técnica conviene aplicarla especialmente a casos de uso claros (transformaciones, visibilidad, etc. de

objetos linkados). Alternativamente, el término *proxy* también podría implementarse creando *objetos vacíos representativos* o *custom datablocks* que actúen como capa intermedia: por ejemplo, un `Empty` que siga la transformación deseada y conduzca al objeto real mediante drivers. Esto es más artesanal, pero ilustra que podemos manipular proxies en lugar del objeto final directo.

- **Arquitectura de evaluación en paralelo:** Para mejorar la performance, se puede diseñar el sistema para evaluar modificadores en **contextos aislados**, potencialmente en paralelo. Por ejemplo, cada FileNodeTree podría evaluarse en una copia separada de la escena (o en un subset de datos duplicados) *simultáneamente*, siempre y cuando sus dependencias lo permitan, y luego combinar los resultados. Si bien Python en Blender corre en un solo hilo (y el GIL limita la verdadera concurrencia), se podría explotar la evaluación diferida: calcular nodos "pesados" (como la carga de archivos) en background threads o usar `bpy.app.timers` para no trabar la interfaz, y luego integrar el resultado cuando esté listo. Otra idea es que, dado que los distintos nodos del grafo ya se evalúan de forma funcional (sin estado persistente entre ejecuciones, según el diseño del addon), **nada impide recalcular solo un subárbol** si sabemos que sus inputs cambiaron mientras el resto permanece igual. Implementar un *sistema de dependencias interno* en el addon (inspirado en el depsgraph) sería ambicioso, pero se puede empezar por marcar cada modificador o nodo con un flag de "dirty" cuando alguno de sus inputs cambia, y solo entonces recalcúlalo. En escenas con muchos modificadores, esto ahorraría recomputaciones redundantes.

- **Sistema de cachés y persistencia de datos:** Complementariamente, conviene introducir **cachés para datos estáticos o repetitivos**. Un caso claro es el de la carga de .blend: el nodo *Read Blend File* podría mantener en la clase del nodo una referencia a los datos linkados la primera vez (por ejemplo, almacenar los `bpy.data.objects` obtenidos en un diccionario global usando la ruta de archivo como clave). Si el nodo se vuelve a evaluar con la misma filepath y ya tiene los datos en memoria, simplemente reutilizarlos en lugar de volver a hacer `bpy.data.libraries.load` (que es costoso). Habría que gestionar la vida de esos datos (por ejemplo, removerlos del caché si el nodo se elimina o si el archivo cambia), pero Blender ya retiene los datablocks linkados hasta que se cierre el archivo, así que el costo es principalmente controlar las referencias duplicadas. Igualmente, para otros nodos que realizan búsquedas (ej. *Get Item by Name/Index* sobre una lista) se puede almacenar el resultado mientras la lista origen no cambie. Un **sistema de cache por nodo** puede integrarse fácilmente dado que la evaluación actual ya pasa un diccionario de outputs; ese diccionario podría compararse con el de la ejecución anterior para decidir si algo cambió. Esto aporta consistencia (los resultados no oscilan inesperadamente) y eficiencia (no recalcular nodos que producirían lo mismo).

- **Integración con la API de Blender y nodos futuros:** A más largo plazo, sería ideal explorar si File Nodes puede integrarse más profundamente con la infraestructura de Blender. Por ejemplo, en Blender 4.x están madurando los *Simulation Nodes* y se planea extender el concepto de Geometry Nodes a más ámbitos ("Everything Nodes"). Quizá algunas de las funcionalidades del addon podrían aprovechar **nodetrees nativos**: imaginemos exponer un NodeTree de Geometry Nodes que gestione escenas/colecciones (no existe aún oficialmente, pero hay debates sobre *Scene Nodes*). Mientras tanto, podemos apoyarnos en la API existente: por ejemplo, usar `bpy.context.scene.evaluated_depsgraph_get()` para obtener versiones evaluadas de objetos (aunque sea solo para lectura/verificación), o utilizar controladores (drivers) para activar cambios condicionalmente. Una recomendación práctica es estructurar el addon de forma modular, separando la representación de datos original, de la representación evaluada. Esto podría significar tener, por cada *FileNodeModItem*, campos que apunten al objeto/escena "evaluado" separado del original. Así se podría conmutar entre uno y otro fácilmente. La

**consistencia** se mantendría porque siempre habría una fuente única de verdad (los originales) y las diffs aplicadas por nodos en otro lado. Este patrón es parecido a cómo Blender maneja los *undo/redo*: guarda estados por separado. En el addon, podría implementarse guardando un *snapshot* de la escena original al iniciar la evaluación, y aplicando todos los mods sobre un duplicado; si algo falla, simplemente descartar el duplicado y restaurar el snapshot original, garantizando robustez.

**Comparación de enfoques:** En resumen, la estrategia actual de File Nodes (guardar atributos y restaurarlos) funciona para un MVP, pero escalará pobremente con muchas operaciones. Rediseñar hacia un modelo de **evaluación no destructiva estilo Geometry Nodes** implicará más uso de memoria temporal (por las copias), pero a cambio ofrece mayor seguridad y potencial de optimización. Usar duplicados/proxies hace la ejecución más independiente del original, reduciendo riesgos de corrupción de datos y facilitando *debugging* (se puede inspeccionar la copia sin miedo). Implementar caches y evaluaciones parciales aportará velocidad en escenas grandes, a costa de mayor complejidad lógica (invalidación de cache, seguimiento de dependencias). Lo importante es que **el núcleo del addon trate a los datablocks originales como inmutables** durante la evaluación – igual que Blender trata la "DNA data" como inmutable [1] – y construya el resultado final a partir de capas superpuestas de cambios. Esto alineará File Nodes con la filosofía de *Everything Nodes* de Blender, donde los modificadores muestran versiones modificadas del contenido sin alterar los datos de origen. Siguiendo estas recomendaciones, el addon podrá apilar nodos y modificadores de archivo de forma más confiable, ofreciendo una experiencia similar a Geometry Nodes: no destructiva, predecible y eficiente incluso con muchos nodos.

**Sugerencias técnicas resumidas:** Implementar una arquitectura de datos de "doble capa" (original + evaluada), utilizar duplicación de ID blocks para aplicar modificaciones, aprovechar library overrides para modificaciones sobre links externos, integrar un sistema básico de dependencias/caché para saltar recomputaciones innecesarias, y probar la evaluación aislada por modificador (incluso mediante escenas ocultas o collections temporales). Estas modificaciones requerirán cierta refactorización del addon, pero aportarán las bases para que File Nodes escale a escenas complejas sin sacrificar la no destructividad ni la performance del flujo de trabajo nodal que pretende brindar. Con ello, el addon se acercará al estándar que Blender marca con Geometry Nodes en cuanto a **evaluación segura y diferida**, llevando el paradigma de *Everything Nodes* un paso más allá hacia la gestión de escenas y proyectos de forma procedural. [1]

---

[1] Dependency Graph - Blender Developer Documentation
https://developer.blender.org/docs/features/core/depsgraph/