Um estudo comparativo entre Journaling e Copy-on-Write em sistemas de arquivos

Daniel Higa, Felipe Soares, Brivaldo Alves da Silva Jr

[1]Faculdade de Computação – Universidade Federal de Mato Grosso do Sul (UFMS)
Caixa Postal 549 – 79.070-900 – Campo Grande – MS – Brazil

{daniel.higa, felipe_soares, brivaldo.junior}@ufms.br

Abstract. *The reliability of file systems depends on mechanisms capable of preserving data consistency in the presence of abrupt failures. Among the main approaches adopted are Journaling and Copy-on-Write (CoW), which follow distinct strategies to achieve this goal. This work presents a comparative analysis of these techniques, exploring their conceptual foundations and practical implementations. Journaling is examined through the Ext4 file system, highlighting the role of JBD2 in the organization of transactions, commits, and recovery after failures. In contrast, Copy-on-Write is studied based on OpenZFS, where immutable block writes and the use of atomic commits ensure structural consistency. The comparison shows that journaling preserves the traditional file system architecture by adding a protection log, whereas CoW redefines the system's internal design, enabling advanced functionalities such as snapshots and clones.*

Resumo. *A confiabilidade de sistemas de arquivos depende de mecanismos capazes de preservar a consistência dos dados diante de falhas abruptas. Entre as principais abordagens adotadas estão o Journaling e o Copy-on-Write (CoW), que seguem estratégias distintas para atingir esse objetivo. Este trabalho realiza uma análise comparativa entre essas técnicas, explorando seus fundamentos conceituais e suas implementações práticas. O journaling é examinado a partir do sistema de arquivos Ext4, destacando o papel do JBD2 na organização de transações, commits e recuperação após falhas. Em contraste, o Copy-on-Write é estudado com base no OpenZFS. A escrita imutável de blocos e o uso de commits atômicos garantem consistência estrutural. A comparação evidencia que o journaling preserva a arquitetura tradicional do sistema de arquivos ao adicionar um log de proteção, enquanto o CoW redefine o próprio desenho interno do sistema, permitindo funcionalidades avançadas como snapshots e clones.*

1. Introdução

Em ambiente computacionais, falhas inesperadas, como quedas de energia, travamentos do sistema operacional ou interrupções durante operações de escrita, podem comprometer tanto a estrutura interna do sistema de arquivos quanto os dados armazenados. Por esse motivo, diferente estratégias de proteção foram desenvolvidas com o objetivo de garantir que o sistema permaneça recuperável e operacional mesmo após falhas abruptas.

Entre essas estratégias, destacam-se duas abordagens predominantes: a técnica de *Journaling*, amplamente adotada por sistemas tradicionais, e o mecanismo de *Copy-on-Write* (CoW), característico de sistemas mais recentes, como ZFS [GIIS 2010, The FreeBSD Project 2024, Dusseau and Dusseau 2013] e Btrfs [Btrfs Developers 2025]. Este trabalho apresenta os conceitos necessários para compreender essas abordagens, discutindo os princípios de consistência, os desafios envolvidos e as soluções que motivaram a adoção de técnicas como *Journaling* e CoW.

2. Comparação Arquitetural

A implementação prática de *Copy-on-Write* (CoW) varia muito mais entre sistemas de arquivos do que o *journaling*, isso porque o CoW não é um mecanismo isolado, mas sim um paradigma estrutural que define a arquitetura do sistema de arquivos. Embora o princípio de funcionamento seja comum, a forma como ele é implementado muda drasticamente entre sistemas de arquivos como ZFS, Btrfs, APFS, WAFL e F2FS, porque cada um escolhe diferentes estruturas de árvores, políticas de alocação, tamanhos de bloco, mecanismos de transação, *caches*, controles de consistência e *pipelines* de escrita. O ZFS, por exemplo, usa a DMU, *uberblocks*, o ZIL e um *pipeline* de escrita altamente estruturado, com *checksums end-to-end*; o Btrfs utiliza múltiplas B-trees, *delayed refs* e um *log tree* separado; o APFS combina várias árvores lógicas, *checkpoints* e mecanismos de clonagem de ranges; o F2FS mistura características de *log-structured* com CoW parcial por segmentos. Assim, o termo *"Copy-on-Write"* representa muito mais um conceito de alto nível do que um mecanismo padronizado.

O *journaling*, por outro lado, tem um comportamento muito mais uniforme porque sua função, registrar no *journal* as intenções de modificação, fazer *flush* do *log*, aplicar alterações e marcar *commit*, é essencialmente a mesma em quase todos os sistemas que o utilizam. Apesar de existirem modos diferentes, o modelo geral continua sendo o de manter um *log* de *redo/undo* para garantir recuperação consistente após falhas, sem alterar profundamente a estrutura interna do *filesystem*. Por isso, comparativamente, o *journaling* varia pouco entre implementações, enquanto o CoW determina praticamente todo o design interno de um sistema de arquivos e, consequentemente, apresenta enorme diversidade técnica na prática.

3. Journaling

O *journal* é um *log* de alterações que o sistema de arquivos usa para proteger metadados (e dados) contra inconsistências. Antes de aplicar mudanças permanentes às estruturas do sistema de arquivos, registra-se no *journal* o suficiente para poder reaplicar (ou descartar) as mudanças depois do *reboot*, de forma que o sistema fique consistente.

Para estudar esse mecanismo na prática, inclusive suas garantias, limites e interações com o *kernel* e o *hardware*, este trabalho toma como exemplo o sistema de arquivos Ext4 [Linux Kernel Documentation 2024], cujo subsistema de *journaling* constitui um caso ainda relevante e bem documentado dentro do *kernel* Linux.

3.1. Ext4

O Ext4 (como Ext3) usa a camada de *journaling* JBD2 (*Journaling Block Device v2*) [Oracle Linux 2020, LWN.net 2021, Linux FSDevel Mailing List 2019]. JBD2 é responsável pelo formato em disco do *journal*, pela gestão de transações, *commits*, *revokes*

e pelo *replay* do *journal* no *boot*. O *driver* do Ext4 no *kernel* usa a API do JBD2 para agrupar alterações em transações e para forçar o *sync* do *journal* quando solicitado (por exemplo em `fsync`).

3.2. Layout em disco

O *journal* é uma área especial, interna à própria partição Ext4 ou em um dispositivo externo, com um formato bem definido. No início fica o superbloco do *journal*, que contém os metadados essenciais sobre o próprio *journal*, como tamanho, versão, número sequencial das transações e diversas *flags* que indicam o estado atual. Cada transação registrada começa com um bloco descritor, que lista exatamente quais blocos do *filesystem* fazem parte daquela operação, apontando seus números e quantidades. Em seguida vêm os blocos de dados, que são cópias dos blocos reais (de metadados e, dependendo do modo de *journaling*, também blocos de dados de arquivos) que estão sendo temporariamente protegidos. Depois que todos os blocos necessários são gravados, o *kernel* escreve o bloco de *commit*, um registro pequeno cuja presença e validade marcam que a transação foi completada (*checkpoint*) com sucesso e pode ser reaplicada de forma segura durante a recuperação. O *journal* também pode conter blocos de revogação, usados quando um bloco que já havia sido registrado é modificado novamente antes do *checkpoint*. Esses registros instruem o processo de *recovery* a ignorar blocos antigos para evitar aplicar dados obsoletos. Quando habilitado, o mecanismo é reforçado por *checksums*, que cobrem o conteúdo do *journal* para detectar corrupção e garantir que apenas transações íntegras sejam consideradas durante a recuperação.

3.3. Ciclo de vida de uma transação

O ciclo de vida de uma transação no JBD2 descreve como o sistema de arquivos organiza, registra e aplica modificações de forma segura, desde sua criação no *kernel* até os processos de *commit*, *checkpoint* e recuperação após falhas, garantindo consistência mesmo diante de interrupções abruptas.

3.3.1. Início da transação

Uma transação no JBD2 começa quando o *kernel*, por meio do Ext4, identifica que precisa agrupar um conjunto de modificações relacionadas, por exemplo, criar um *inode*, alterar *bitmaps* de blocos livres, atualizar entradas de diretório ou ajustar ponteiros de blocos. Em vez de realizar essas alterações diretamente no disco, uma nova transação é iniciada, que serve como uma fronteira lógica garantindo que essas mudanças de metadados sejam aplicadas de forma consistente.

3.3.2. Preparação / coleta de blocos

Antes que qualquer dado seja gravado no *journal*, o Ext4 identifica e coleta todos os blocos que serão alterados (*inodes*, *bitmaps*, diretórios e blocos de indireção). Esses blocos são copiados para *buffers* que servirão como a versão "pré-escrita" no *journal*. No modo **`data=journal`**, tanto dados quanto metadados entram no *journal*. Nos modos `data=ordered` e `data=writeback`, apenas metadados são registrados (com a

diferença de que `ordered` garante que os dados sejam escritos no disco antes do *commit*).

3.3.3. Escrita no *journal* (fase de *logging*)

Após reunir os blocos relevantes, o *kernel* inicia a escrita no *journal*. Essa fase segue uma ordem estrita: primeiro são gravados os blocos de *descriptor*, que listam quais blocos pertencem à transação; em seguida vêm os blocos de dados propriamente ditos (metadados ou dados completos, dependendo do modo). Essa sequência garante que, durante a recuperação, o *kernel* saiba exatamente o que aplicar e em que ordem.

3.3.4. *Commit*

Finalmente, quando todos os blocos foram enviados ao *journal*, o sistema grava o bloco de *commit*, que é a marcação final indicando que a transação foi registrada com sucesso. Se *checksums* estiverem habilitados, eles são computados e incluídos para permitir validação durante o *replay*. O *commit* é o ponto crítico no ciclo de vida da transação. Quando o bloco de *commit* é escrito e confirmado no dispositivo de armazenamento, idealmente com *flushes* que garantem persistência real, a transação passa a ser considerada consolidada. Isso significa que, mesmo que ocorra uma falha logo em seguida, as informações necessárias para restaurar o *filesystem* para esse ponto consistente já estão asseguradas no *journal*. É apenas após essa etapa que o Ext4 pode planejar a atualização dos blocos originais em seus locais definitivos.

3.3.5. *Checkpoint / writeback* para o *filesystem*

Após o *commit*, o *kernel* pode, de forma imediata ou adiada, iniciar o processo de *checkpoint*. Esse procedimento aplica os blocos registrados no *journal* aos seus locais finais no *filesystem*, substituindo os blocos antigos pelos novos. Por exemplo, o *bitmap* atualizado é escrito no bloco correspondente do grupo, e o *inode* recém-modificado é colocado em sua posição na *inode table*. Quando todos os blocos da transação atual foram devidamente aplicados, o espaço correspondente no *journal* pode ser liberado para reutilização. Se algum bloco tiver sido modificado repetidamente antes de ser submetido ao *checkpoint*, os mecanismos de revogação e ordenação evitam que versões antigas sejam reaplicadas durante o *recovery*.

3.3.6. *Recovery* (após *crash*)

Se ocorrer uma falha crítica antes que as transações sejam executadas e totalmente salvas, o *mount* subsequente do *filesystem* ativa o processo de *recovery*. O *kernel* lê o *journal* do Ext4 e reexecuta apenas as transações cujo *commit* foi encontrado e validado, ignorando qualquer transação incompleta. O *replay* segue as listas de *descriptor* para saber quais blocos aplicar e usa os registros de revogação para pular blocos que foram invalidados por mudanças posteriores. Com isso, o *filesystem* é restaurado para o estado consistente mais

recente que havia sido totalmente registrado no *journal*, garantindo integridade mesmo frente a falhas abruptas.

3.4. Modos de *journaling*

O Ext4 [Baeldung 2022] oferece três modos principais de *journaling*, cada um refletindo uma estratégia para balancear integridade e desempenho. Esses modos determinam se apenas metadados ou também dados de arquivos serão registrados no *journal*, bem como a ordem em que os blocos são transferidos para o disco.

No modo **`data=journal`**, o mais seguro porém também o mais custoso, tanto os metadados quanto os blocos de dados de arquivos são primeiro escritos integralmente no *journal* antes de serem aplicados aos seus locais definitivos. Isso garante que, em caso de *crash*, todo o conteúdo necessário para restaurar o estado mais recente do *filesystem* está preservado no *journal*, reduzindo praticamente a zero o risco de perda ou inconsistência de dados. O lado negativo é o impacto significativo no desempenho, já que todo bloco modificado é gravado duas vezes (no *journal* e depois no destino), e técnicas como `delayed allocation` são desativadas. Assim, esse modo costuma ser recomendado apenas em cenário de máxima exigência de integridade.

O modo **`data=ordered`**, que é o padrão do Ext4, fornece um equilíbrio entre segurança e desempenho. Apenas os metadados são registrados no *journal*, mas o sistema garante que todos os blocos de dados relacionados a esses metadados sejam gravados no disco antes do *commit* da transação no *journal*. Isso evita situações em que um *inode* atualizado aponta para blocos que ainda não foram devidamente escritos ou que contêm dados antigos, garantindo certa proteção sem penalidades tão severas no *throughput* de escrita.

Por fim, o modo **`data=writeback`** mantém o *journaling* apenas para metadados, porém sem impor qualquer relação de ordenação entre a escrita de dados e o *commit*. Essa ausência de garantia significa que, após um *crash*, os metadados poderão estar corretos, mas o conteúdo de arquivos pode não refletir a última modificação, o chamado *stale data*. Apesar desse risco, o modo *writeback* oferece o melhor desempenho entre os três, sendo útil em cenários em que a integridade dos dados em si não é crítica e a prioridade é maximizar a velocidade de I/O.

Esses modos definem o comportamento de segurança do Ext4 e são o principal mecanismo para ajustar o *trade-off* entre consistência e desempenho

3.5. *Checksums* do *journal*

O Ext4 tem suporte a ***checksums* no *journal***. Cada bloco de *journal* pode ter *checksum* associado, isso protege contra corrupções silenciosas (mídia danificada ou bugs). O *kernel* e ferramentas como `e2fsprogs` entendem e usam esse mecanismo quando a funcionalidade está ativa (`journal_inum`/`journal_checksum` e *flags* relevantes). *Checksums* são especialmente importantes porque o *replay* do *journal* confia que o *log* esteja íntegro.

3.6. *Delayed allocation* e interações com *journaling*

O Ext4 emprega a técnica de ***delayed allocation*** (`delalloc`) para melhorar o desempenho de escrita. Em vez de alocar imediatamente os blocos físicos no disco, o *kernel* adia essa decisão até o momento do *flush*. Isso permite escolher blocos contíguos e reduzir

fragmentação. No entanto, esse comportamento interage diretamente com o mecanismo de *journaling* e, portanto, precisa ser coordenado para garantir consistência.

No modo **`data=journal`**, o uso de `delalloc` é praticamente incompatível. Como todos os dados precisam ser registrados no *journal* imediatamente, não há como adiar a alocação física: o kernel precisa saber onde os dados ficarão e copiá-los para o journal sem atraso. Por isso, o Ext4 normalmente desativa `delalloc` nesse modo, sacrificando otimizações em favor da integridade.

No modo **`data=ordered`**, que depende da garantia de que os blocos de dados sejam gravados no disco antes do *commit* dos metadados, o *delayed allocation* pode introduzir um risco aparente: quando chega o momento de realizar o *commit* da transação, parte dos dados relacionados ainda pode não estar fisicamente escrita. Para resolver isso, o Ext4 implementa mecanismos que rastreiam quais blocos de dados precisam ser forçados ao disco antes do *commit* do *journal*. Quando as estruturas de metadados da transação estão prestes a ser registradas, o *kernel* dispara um *flush* explícito dos dados associados, garantindo a ordenação correta. Assim, `delalloc` continua habilitado, mas com salvaguardas que preservam o princípio do modo *ordered*.

3.7. *Replay* e limites do *recovery*

Quando o *kernel* detecta, na montagem de um volume Ext4, que existe um *journal* com entradas não resolvidas, a camada de *journaling* JBD2 entra em ação para reaplicar transações que estão no *commit* (*replay*). A intenção do *replay* é restaurar as estruturas do sistema de arquivos ao ponto consistente mais recente que foi totalmente registrado no *journal*.

3.7.1. Varredura inicial e construção do *replay*

JBD2 percorre o *log* circular lendo *headers* de bloco de *journal*. Para cada entrada, distingue tipos (*descriptor*, *data*, *commit*, *revoke*). A camada valida: o tipo do bloco (*header magic/version*), o ***sequence number*** associado à transação e o ***checksums* do bloco** (*header* e *payload*). Apenas transações cujo bloco de ***commit*** foi encontrado e cuja integridade (*checksums*) está correta são candidatas ao *replay*. Entradas com *checksum* inválido, *header* corrompido ou sem *commit* são tratadas como incompletas e ignoradas.

3.7.2. Tratamento de *revokes*

Antes de aplicar blocos, o JBD2 processa a área de ***revogation records*** associada à transação (ou à sequência), que lista blocos que foram invalidados (*revoked*) posteriormente. Para cada bloco listado no *revoke table*, o JBD2 marca o respectivo *buffer* como inválido/ignorável para *replay*. Isso evita que versões antigas sejam reaplicadas quando uma versão mais nova já foi escrita no local definitivo antes do *crash*.

3.7.3. Ordem de *replay* e atomicidade lógica

O *replay* é aplicado por transação em ordem crescente de *sequence number*, respeitando a ordem global do *journal*. Dentro de uma transação, os *descriptors* já impõem uma ordem

lógica; entretanto, o *replay* pode agrupar submissões de I/O para ganho de *throughput*, desde que mantenha a semântica exigida (i.e., aplicar os blocos da transação antes de considerar a transação "*done*"). O objetivo é reconstituir o mesmo efeito que o *checkpoint* teria produzido se o sistema não tivesse falhado.

4. *Copy-on-Write*

Copy-on-write (CoW) em sistemas de arquivos é uma técnica na qual **nenhuma modificação é feita *in-place* nos blocos existentes**. Em vez disso, qualquer alteração em dados ou metadados provoca a alocação de novos blocos, seguida da gravação da versão modificada nesses blocos recém-alocados, e somente depois uma atualização atômica de ponteiros faz com que a árvore de metadados passe a referenciar a nova versão, garantindo consistência estrutural mesmo em presença de falhas de energia, pois nenhuma estrutura ancestral é sobrescrita até que todos os seus filhos atualizados estejam persistidos.

Para estudar este mecanismo em profundidade, incluindo alocação, encadeamento de blocos, lógica de atualização transacional e interação com o mecanismo de escrita, utilizaremos a implementação do **OpenZFS** [OpenZFS Project 2023] como referência prática.

4.1. Fluxo de escrita

O fluxo de escrita no OpenZFS descreve a transformação de uma operação lógica de escrita em atualizações físicas persistentes, organizadas segundo o modelo de *Copy-on-Write* e consolidadas por meio de *Transaction Groups*. As etapas a seguir detalham esse processo desde a entrada no *kernel* até o *commit* atômico do estado do sistema.

4.1.1. Entrada da escrita

Quando um processo emite uma chamada de escrita via `syscalls` como `write` ou `pwrite`, o fluxo desce pelo *VFS* (*Virtual File System*) até alcançar o *ZFS POSIX Layer* (ZPL). O ZPL converte a operação POSIX em uma operação nativa da DMU (*Data Management Unit*), a camada lógica interna do OpenZFS responsável pela manipulação de objetos, que incluem arquivos, diretórios, estruturas de metadados, árvores de indireção e atributos estendidos. Ao receber a solicitação de escrita, a DMU realiza o mapeamento do objeto lógico para um conjunto de `dmu_bufs`. Cada `dmu_buf` representa um bloco lógico gerenciado pela DMU. Caso o bloco correspondente ainda não tenha sido carregado, ele é lido do armazenamento ou inicializado como um bloco vazio no cache ARC (*Adaptive Replacement Cache*).

Ao modificar um `dmu_buf`, a DMU marca esse *buffer* como *dirty* (sujos), ou seja, como parte de um conjunto de alterações pendentes. Esses *buffers* sujos não são imediatamente escritos no disco: eles residem no ARC, ou no L2ARC caso a memória principal seja insuficiente para acomodar todos os dados em cache (embora metadados sujos permaneçam sempre no ARC). A DMU organiza essas alterações em estruturas conhecidas como *dirty records*, cada uma associada a um bloco sujo, a um *checksum* futuro e ao TXG ao qual a alteração pertencerá. A escrita é, portanto, inteiramente absorvida pela memória no primeiro momento. Somente no ponto de sincronização os dados efetivamente descem para o pipeline físico.

4.1.2. Acúmulo por *Transaction Group* (TXG)

O ZFS organiza todas as atualizações em *Transaction Groups* (TXGs), que são intervalos contínuos de mudanças agrupadas logicamente. Um TXG está sempre em um dentre três estados: *open*, *quiescing* e *syncing*. Durante a fase *open*, todas as modificações vindas do ZPL DMU são aceitas e registradas como *dirty records* vinculados àquele TXG. Quando o TXG atinge limites de tempo (tipicamente 5 segundos), memória ou de quantidade de dados sujos, ele migra para a fase *quiescing*. Nessa fase, o TXG deixa de aceitar novas modificações, e todas as *threads* de escrita passam a contribuir para o próximo TXG aberto. Já o TXG anterior é então entregue ao *syncing thread*, uma *thread kernel* exclusiva do ZFS responsável por transformar as alterações lógicas (*dirty records*, listas de atualização e modificações em árvores) em um conjunto final de blocos físicos a serem gravados.

Esse design garante atomicidade de larga escala: cada TXG é um *commit* indivisível de todo o estado do pool. Nada dentro de um TXG se torna visível até que o *uberblock* correspondente ao TXG seja gravado com sucesso. Isso cria semântica similar à de um "*epoch*"de atomicidade, garantindo consistência sem exigir *journaling* tradicional.

4.1.3. DMU → ZIO *pipeline*

Com o TXG entrando em modo *syncing*, os *dirty records* são convertidos em operações físicas de I/O chamadas ZIOs (ZFS I/O *requests*). O subsistema ZIO funciona como uma *pipeline* modular de transformação de blocos, onde cada estágio corresponde a uma operação: primeira compressão (LZ4, ZSTD, GZIP, etc), depois *checksumming* (Fletcher4, SHA256, etc), criptografia opcional se configurado, montagem de indireções ou *gang blocks* caso um bloco grande precise ser subdividido, e seleção dos `vdevs` físicos onde cada bloco será alocado.

O ZIO pipeline estrutura cada I/O como um grafo de dependências assíncronas, no qual cada etapa só dispara quando todas as etapas predecessoras completaram. Assim, um ZIO de `write` pode gerar ZIOs filhos que representam compressão, *checksum*, cópia de dados, escrita física e atualizações de metadados. Todas essas operações são agendadas no `taskq` (conjunto de *threads* do ZFS), com paralelismo profundo e alta afinidade com o hardware subjacente. Em caso de falha de um `vdev` (por exemplo, setor defeituoso ou dispositivo morto), o ZIO tenta automaticamente caminhos alternativos ou recupera réplicas já garantidas pelo RAID-Z ou espelhamento.

4.1.4. Alocação de blocos

Antes de enviar um bloco ao dispositivo físico, o ZFS precisa alocar espaço para ele. Cada `vdev` é dividido em dezenas ou centenas de `metaslabs`, pedaços lógicos do espaço físico com seus próprios *space maps*. Cada `metaslab` contém um histórico de alocações representado como uma sequência de entradas *log*-estruturadas (*alloc / free*) que formam o *space map*, complementado por um *in-memory range tree* construído durante a abertura ou criação do *pool*. O alocador consulta esses *range trees* para encontrar regiões

livres adequadas, seguindo políticas como *first-fit*, *largest-free-range* ou rotacionamento heurístico entre `metaslabs` (*metaslab groups*), de forma a mitigar fragmentação a longo prazo.

Por se tratar de um sistema CoW puro, o ZFS nunca sobrescreve blocos existentes. Cada escrita aloca um bloco novo, e o antigo permanece válido até que nenhum *block pointer* o referencie (momento no qual o sistema pode liberá-lo, usualmente durante o processo de `spa_sync`). Isso oferece segurança, atomicidade e facilita *snapshots* e clones: como as versões antigas dos blocos continuam existindo sem modificação, os *snapshots* são apenas uma imobilização de seus *block pointers*.

4.1.5. *Checksums* e block *pointers*

Quando um bloco é finalizado pelo pipeline ZIO, seu *checksum* é calculado e o valor resultante é inserido no *block pointer* que faz referência a ele. O *block pointer* é uma estrutura rica contendo múltiplos campos: endereços físicos (DVA — *Data Virtual Address*), tamanho do bloco, tamanho lógico, tipo de compressão usada, *checksums*, nível de indireção e *flags* adicionais (como *birth txg*, *fill count* e *flags* de criptografia). A árvore de *block pointers* forma a estrutura hierárquica do arquivo: data *blocks* são referenciados por *indirect blocks*, que são referenciados por níveis superiores, culminando no *root block pointer* do *dataset*. Essa estrutura lembra uma *B-tree*, mas é otimizada para CoW e não possui balanceamento dinâmico tradicional, em vez disso, substitui nós inteiros durante o *sync*.

Cada alteração descendente (em um bloco de dados) naturalmente produz alterações ascendentes (novos *block pointers* nos *indirect blocks*), pois cada bloco modificado tem um novo checksum e novo endereço físico. Isso cria uma escalada de mudanças até o *rootbp*, que é o ponto de entrada do *dataset* no *uberblock*.

4.1.6. *Commit* atômico: *uberblocks* e *TXG sync*

Quando todo o conjunto de blocos do TXG é escrito, incluindo os dados e seus metadados derivados (*indirect blocks*, *dnodes*, metadados do *dataset*, MOS — *Meta-Object Set*), o ZFS prepara o *commit* atômico final: a escrita do *uberblock*. Cada `vdev` contém um conjunto circular de *uberblocks* gravados em sua *label*. Cada *uberblock* registra, entre outros itens, o número do TXG (`ub_txg`), *timestamps*, e principalmente o *rootbp* do pool, que aponta para o topo da árvore de metadados do MOS. A escrita de um *uberblock* é extremamente pequena (algumas centenas de bytes) e é feita somente após todo o resto ter sido persistido.

O mecanismo de atomicidade é brutalmente simples: apenas o *uberblock* com maior TXG válido representa o estado mais recente do pool. Em caso de *crash* durante o *sync*, o novo *uberblock* não será escrito ou estará inconsistente, e na montagem seguinte o ZFS automaticamente escolherá o *uberblock* anterior, cujo TXG é menor mas válido. Isso garante que todas as operações do TXG sejam aplicadas integralmente ou que nenhuma delas seja efetivada. Esse modelo elimina a necessidade de journaling tradicional e fornece uma consistência transacional intrínseca baseada na imutabilidade de blocos.

4.2. Como o COW produz *snapshots* e clones "instantâneos"

O mecanismo de *snapshots* e clones no OpenZFS é uma consequência direta e inevitável do seu modelo de *Copy-on-Write* aplicado à árvore inteira de metadados, e não uma funcionalidade "extra" adicionada sobre o sistema de arquivos. Em ZFS, *snapshots* não exigem cópia de dados nem reescrita de blocos existentes; eles emergem naturalmente do fato de que *block pointers* são imutáveis após o *commit* de um TXG e que nenhuma escrita ocorre *in-place*.

Quando um *snapshot* é criado, o ZFS executa uma operação puramente de metadados: ele captura o *root block pointer* atual do *dataset*, isto é, o ponteiro que referencia a raiz da árvore de metadados (*dnodes*, *indirect blocks* e data *blocks*) naquele exato TXG, e o registra como pertencente a um *snapshot*. Internamente, isso ocorre no MOS, onde existe uma lista ordenada de *snapshots* associada a cada *dataset*, cada um identificado por um TXG específico. O *snapshot*, portanto, é essencialmente um nome simbólico associado a um conjunto de *block pointers* já existentes, sem que nenhum bloco adicional precise ser alocado.

A partir desse momento, os *block pointers* que pertencem à árvore capturada passam a ser considerados logicamente imutáveis no contexto daquele *snapshot*. Isso não significa que o ZFS marque blocos físicos como *read-only* no disco; em vez disso, a imutabilidade é garantida estruturalmente pelo modelo COW: qualquer modificação futura em qualquer arquivo ou metadado do *dataset* ativo jamais sobrescreverá esses blocos, pois toda escrita aloca novos blocos e produz novos *block pointers*. Assim, o *snapshot* continua apontando para os blocos antigos, enquanto o *dataset* ativo passa a apontar para blocos novos. Essa separação ocorre automaticamente, sem necessidade de rastreamento explícito de versões por bloco.

É por isso que *snapshots* recém-criados ocupam zero *bytes* adicionais: eles apenas adicionam referências (*block pointers*) para blocos já existentes. O consumo de espaço cresce apenas quando o *dataset* ativo começa a modificar dados que eram anteriormente referenciados pelo *snapshot*. Cada modificação gera novos blocos via COW, enquanto os blocos antigos permanecem vivos porque ainda são referenciados pelo *snapshot*. O espaço "usado pelo *snapshot*" é, na realidade, o espaço dos blocos que deixaram de ser o *dataset* ativo e passou a referenciar versões novas.

Do ponto de vista interno, isso é implementado por meio de contagem implícita de referências. Um bloco físico só pode ser liberado quando nenhum *block pointer* em nenhuma árvore ativa (*dataset* ativo, *snapshots* ou clones) apontam para ele. O processo de liberação ocorre durante o `spa_sync`, quando o ZFS percorre as estruturas de metadados e identifica blocos cuja última referência foi removida. Até lá, o bloco permanece alocado no `metaslab` correspondente, mesmo que o *dataset* ativo já não o utilize mais.

Clones são uma extensão direta desse mesmo mecanismo. Um clone é criado a partir de um *snapshot* e consiste em um novo *dataset* cujo *root block pointer* inicial é exatamente o mesmo `rootbp` do *snapshot* de origem. Em outras palavras, o clone e o *snapshot* começam apontando para a mesma árvore completa de metadados e dados. No instante da criação, o clone não possui nenhum bloco exclusivo; todo o seu conteúdo é compartilhado com o *snapshot*. A partir desse ponto, qualquer modificação feita no clone segue exatamente o mesmo fluxo COW descrito anteriormente: novos blocos são aloca-

dos, novos *block pointers* são criados, e apenas o clone passa a referenciá-los, enquanto o *snapshot* e outros clones continuam apontando para os blocos antigos.

Essa arquitetura implica uma propriedade importante: clones e *snapshots* compartilham dados de forma transitiva e estrutural, não por meio de um mecanismo explícito de deduplicação. O compartilhamento existe porque os *block pointers* são idênticos, não porque o sistema detecta igualdade de conteúdo. Isso torna o compartilhamento extremamente barato em termos de metadados e elimina a necessidade de tabelas globais de referência como ocorre em sistemas de `dedup` clássicos. Ao mesmo tempo, isso significa que o compartilhamento só ocorre em granularidade de bloco e apenas enquanto os ponteiros forem compartilhados; uma vez que um bloco é modificado, a divergência é permanente.

Um ponto crítico é que, diferentemente de sistemas como Btrfs ou XFS com *reflinks*, o OpenZFS não implementa *Copy-on-Write* por arquivo isolado (*per-file reflink*). Não existe uma operação equivalente a `cp --reflink` que crie duas entradas de diretório apontando para os mesmos blocos de dados de forma independente. Em ZFS, o COW opera no nível de *datasets* inteiros, e o compartilhamento é viabilizado exclusivamente via *snapshots* e clones. Isso não é uma limitação acidental, mas uma consequência direta do design da DMU e da forma como *dnodes* e *block pointers* são organizados. Introduzir *reflinks* por arquivo exigiria mudanças profundas na semântica de referência de blocos e na contabilidade de espaço, razão pela qual o tema aparece apenas como discussões e propostas experimentais, e não como funcionalidade estável.

Outro efeito importante desse modelo é que *snapshots* participam integralmente da semântica de consistência do ZFS. Como o *snapshot* captura uma árvore inteira de *block pointers* pertencente a um TXG específico, ele representa um estado totalmente consistente do sistema de arquivos, equivalente a um ponto de *commit* atômico. Não existem *snapshots* “parciais” ou “inconsistentes”. Isso os torna ideais para backup incremental, replicação (`zfs send/receive`) e *rollback* confiável, pois cada *snapshot* corresponde exatamente a um estado que já foi confirmado via `uberblock`.

5. Conclusão

Ao longo deste trabalho, foi possível analisar de forma detalhada e comparativa os dois principais paradigmas de consistência empregados em sistemas de arquivos modernos: o *journaling* e o *Copy-on-Write*. A partir do estudo do Ext4 e de sua camada de *journaling* JBD2, observou-se que o *journaling* atua como um mecanismo de proteção acoplado a um modelo tradicional de escrita *in-place*, oferecendo garantias de recuperabilidade por meio do registro antecipado das intenções de modificação. Essa abordagem apresenta comportamento relativamente uniforme entre diferentes implementações, com complexidade delimitada e previsível, permitindo ao administrador ajustar o compromisso entre desempenho e integridade por meio de modos de operação. Em contrapartida, suas garantias estão fortemente ligadas à correta ordenação de escritas e à confiabilidade do *journal*, além de não eliminar completamente certas classes de inconsistência lógica nos dados.

Por outro lado, o estudo do *Copy-on-Write* a partir do OpenZFS evidenciou que essa técnica não se limita a um mecanismo isolado, mas define a própria arquitetura do sistema de arquivos. Ao evitar qualquer sobrescrita *in-place* e basear a persistência em *commits* atômicos de grandes conjuntos de modificações, o ZFS fornece consistência es-

trutural intrínseca, verificação de integridade de ponta a ponta e recuperação automática sem necessidade de *replay* de *logs* tradicionais. Esse modelo, embora mais complexo em termos de implementação e com custos potenciais de fragmentação, viabiliza de forma natural funcionalidades avançadas como *snapshots* e clones instantâneos, que emergem diretamente da imutabilidade dos blocos e da organização da árvore de metadados.

Dessa forma, a comparação demonstra que *journaling* e *Copy-on-Write* representam filosofias distintas de projeto: enquanto o *journaling* busca adicionar confiabilidade a um modelo clássico de sistemas de arquivos, o CoW reconstrói esse modelo a partir de princípios transacionais.

Referências

Baeldung (2022). Understanding Ext4 Journal Modes. https://www.baeldung.com/linux/ext-journal-modes. Acesso em: 12 Dez. 2025.

Btrfs Developers (2025). Btrfs Documentation. https://btrfs.readthedocs.io/en/latest/. Acesso em: 12 Dez. 2025.

Dusseau, A. C. and Dusseau, R. H. (2013). ZFS Internals. Lecture slides, University of Wisconsin–Madison. Disponível em: https://pages.cs.wisc.edu/~dusseau/Classes/CS736/CS736-F13/Lectures/2_zfs_internals.pptx. Acesso em: 12 Dez. 2025.

GIIS (2010). ZFS On-disk Format. https://www.giis.co.in/Zfs_ondiskformat.pdf. Acesso em: 12 Dez. 2025.

Linux FSDevel Mailing List (2019). Discussion on JBD2 Behavior and Recovery. https://marc.info/?l=linux-fsdevel&m=156711923530455. Acesso em: 12 Dez. 2025.

Linux Kernel Documentation (2024). The Ext4 filesystem. https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html. Acesso em: 12 Dez. 2025.

LWN.net (2021). JBD2 Scaling and Journaling Internals. https://lwn.net/Articles/842385. Acesso em: 12 Dez. 2025.

OpenZFS Project (2023). ZFS Read/Write Lecture. https://openzfs.org/wiki/Documentation/Read_Write_Lecture. Acesso em: 12 Dez. 2025.

Oracle Linux (2020). On-disk Journal Data Structures (JBD2). https://blogs.oracle.com/linux/ondisk-journal-data-structures-jbd2. Acesso em: 12 Dez. 2025.

The FreeBSD Project (2024). ZFS Handbook. https://docs.freebsd.org/en/books/handbook/zfs/. Acesso em: 12 Dez. 2025.