



UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY



FACULTAD DE
INGENIERÍA

Diseño y desarrollo de un filtro de deduplicación de datos basado en técnicas de estado del arte

Informe de Proyecto de Grado presentado por

Nicolás Comerci

en cumplimiento parcial de los requerimientos para la graduación de la carrera
de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de
la República

Supervisores

Álvaro Martín,
Guillermo Dufort y Álvarez

Montevideo, 6 de agosto de 2025



Diseño y desarrollo de un filtro de deduplicación de datos
basado en técnicas de estado del arte por Nicolás Comerci
tiene licencia [CC Atribución 4.0](https://creativecommons.org/licenses/by/4.0/).

Agradecimientos

A todos los que me acompañaron en este camino.
A mi padre, que más que nadie despertó en mí la curiosidad y el hambre por el conocimiento.

Resumen

El campo de deduplicación de datos ha ganado notoriedad en la última década dada la gran cantidad de datos manejada por sistemas contemporáneos, lo común que es que estos datos presenten altos niveles de duplicación, y la eficacia de las técnicas de deduplicado para reducir estos volúmenes de datos. Más recientemente también se da un renovado interés en las áreas de detección de semejanza de datos y reducción de redundancia en datos similares, ya que pueden complementar a las técnicas de deduplicado de datos. Sin embargo, la mayoría de la actividad de estudio en estas áreas no proporcionan implementaciones de disponibilidad pública y a la vez se centran principalmente en la aplicabilidad de estas técnicas para sistemas distribuidos o de archivado de datos. En este proyecto, por tanto, nos avocamos a la tarea de relevar de la bibliografía académica disponible y actual sobre estas disciplinas para identificar cuáles son las técnicas de estado del arte en la actualidad. Habiendo identificado estas técnicas, desarrollamos una herramienta de código abierto que implementa un filtro de datos que deduplica y aplica reducción de redundancia en datos similares sobre flujos de datos arbitrarios. Estudiamos la eficacia de la aplicación de nuestra implementación del filtro como complemento preliminar al uso de compresión general sobre datos arbitrarios. De esta forma, no solo ponemos a disposición general nuestra implementación, sino que exploramos su utilidad en un tipo de aplicación que no ha visto tanto estudio en tiempos recientes.

Palabras clave: Deduplicación de datos, Reducción de datos, Codificación delta, Detección de semejanza, Compresión de datos

Índice general

1. Introducción	1
2. Marco teórico	5
2.1. Deduplicación y características básicas	5
2.1.1. Objetivo y evaluación de rendimiento	5
2.1.2. Nivel de aplicación de la deduplicación	6
2.1.3. Identificación de datos duplicados	6
2.1.4. Sistema de deduplicación	7
2.2. Cortado de bloques	8
2.2.1. FSC (Fixed-Size Chunking)	8
2.2.2. Problema de desplazamiento de bordes	9
2.2.3. Rolling Hashes	10
2.2.4. Deduplicación por ventana deslizante	11
2.2.5. Corte de bloques por contenido	11
3. Revisión de antecedentes	15
3.1. Breve historia de la teoría de la compresión	15
3.1.1. Aspectos básicos de compresión de datos	15
3.1.2. Compresión Delta	16
3.1.3. Deduplicación	17
3.2. Estado del arte	18
3.2.1. FastCDC	19
3.2.2. SS-CDC	21
3.2.3. SuperCDC	23
3.2.4. DARE	24
4. Diseño de la solución	27
4.1. Desafíos y sus resoluciones	27
4.1.1. Múltiples tipos de condiciones de corte	27
4.1.2. Salto al tamaño mínimo del bloque	28
4.1.3. Teorema de condición suficiente para la definición de un punto de corte en CDC	29
4.2. CDCZ	30
4.3. Implementación	35

4.3.1.	DupAdj y compresión delta	37
4.3.2.	Descompresión	37
4.3.3.	Disponibilidad	37
5.	Experimentación	39
5.1.	Datos de prueba	39
5.1.1.	Juego de datos LNX	40
5.1.2.	Juegos de datos LNX-IMG	40
5.1.3.	Juego de datos FslHomes	40
5.2.	Configuración del entorno de prueba	41
5.3.	CDCZ representa al estado del arte	41
5.3.1.	CDCZ es comparable con FastCDC al correr en un único hilo	41
5.3.2.	CDCZ permite paralelismo con características compara- bles a SS-CDC	42
5.3.3.	CDCZip se compara favorablemente contra LRZip	42
5.3.4.	CDCZip como preprocesador para compresión	43
6.	Conclusiones y Trabajo Futuro	47
	Referencias	49

Capítulo 1

Introducción

La compresión de datos es de vital importancia para optimizar el uso de recursos para el almacenamiento y transmisión de grandes cantidades de datos, típico en varias aplicaciones contemporáneas y novedosas como sistemas cloud, distribuidos y big data.

De particular interés son las técnicas de *deduplicación* de datos, que se centran en la detección y eficiente almacenamiento de largas cadenas de datos repetidos en largas distancias, en comparación con la compresión general que incluye análisis y transformaciones más sofisticadas de los datos. En determinados escenarios, estas técnicas tienen mejor rendimiento en términos del balance ofrecido en capacidad computacional necesaria y eficacia al reducir los datos (Xia, Jiang, Feng, Douglass, y cols., 2016). Casos de estudio típicos encuentran regularmente que las técnicas de deduplicación son aplicables para diversos tipos de datos, reportando reducciones en el volumen de datos del 50 % (Meyer y Bolosky, 2012) o incluso del 85 % (Wallace y cols., 2012).

Sin embargo, estos estudios típicamente tratan sobre sistemas de archivado y respaldo de datos, sistemas de almacenamiento en la nube o sistemas de almacenamiento distribuido. Hasta donde sabemos, no ha habido ninguna investigación o implementación sobre técnicas de deduplicación para uso conjunto con compresión de uso general desde RZip y sus implementaciones (Tridgell, 2000).

Nuestro objetivo es, por lo tanto, el desarrollo de una implementación de código abierto de un filtro de deduplicación de datos basado en técnicas de estado del arte, con el propósito de ser usado como preprocesador para compresión general de datos.

Dado que es esperable que los datos a comprimir típicamente contengan cantidades no triviales de datos duplicados, las técnicas de deduplicación modernas (al ser específicamente diseñadas para este propósito) son óptimas para atacar

este aspecto de la reducción del volumen de datos. Filtrar de antemano estos datos duplicados debería incrementar el rendimiento y eficiencia del proceso, ya que el compresor general recibe un volumen ya reducido de datos y con parte de su trabajo ya hecho, más rápido de lo que podría haberlo hecho.

El concepto no es nuevo, el algoritmo RZip, su implementación de referencia y su implementación mejorada LRZip (Kolivas, 2008) actúan como un preprocesador que elimina cadenas de datos duplicados a largas distancias y luego le aplican compresión general con algoritmos como BZip2 (Seward, 1996), GZip (Adler y Gailly, 1992) o LZMA (Pavlov, 1999). Adicionalmente, ZStandard (Collet, 2021), un compresor moderno que ha ganado popularidad rápidamente, incluye un modo de ‘largo rango’ donde aplica el algoritmo RZip para encontrar este tipo de duplicación de datos. Todos estos antecedentes están basados en el algoritmo RZip originalmente publicado en el año 2000. En el año 2024, disponemos de más de 2 décadas de innovación desde ese entonces, lo que nos motiva a usarlas con la esperanza de dar nueva vida a este concepto.

Identificamos los algoritmos FastCDC (Xia y cols., 2020), SuperCDC (Wan y cols., 2022) y SS-CDC (Ni, Lin, y Jiang, 2019) como representativos del estado del arte, dado que presentan los más altos niveles de rendimiento y eficiencia. Integramos estos tres algoritmos en nuestro nuevo algoritmo CDCZ. FastCDC y SuperCDC son integrados de forma que se heredan todos sus beneficios al ejecutar en un solo hilo, mientras que SS-CDC proporciona capacidades de paralelismo casi lineales sin comprometer las tasas de compresión.

La integración de estos tres algoritmos no es trivial, en particular dado que SS-CDC presenta ciertas incompatibilidades con los otros algoritmos. Contribuimos un teorema 1 que nos permite solucionar esta incompatibilidad y creemos puede ser de gran utilidad para el desarrollo de futuros algoritmos de deduplicación usando paralelismo.

Usando nuestro algoritmo CDCZ implementamos CDCZip, un filtro preprocesador-deduplicador que técnicamente es un compresor LZ77 (Ziv y Lempel, 1977), en línea con el antecedente de RZip.

Evaluamos CDCZip en términos de su utilidad como preproceso de los datos al aplicar métodos de compresión general como Zstandard o LZMA, y en comparación con su predecesor RZip (implementación LRZip). Observamos que CDCZip tiene la capacidad esperada de incrementar tasas de compresión y reducir los tiempos de ejecución de los compresores generales. También presenta en general un mejor rendimiento y eficiencia que LRZip.

Organizamos el resto de este documento en los siguientes capítulos:

- Capítulo 2: Se presenta el marco teórico utilizado durante el proyecto presentando los conceptos básicos de deduplicación y herramientas asociadas.

- Capítulo 3: Se realiza la revisión de antecedentes y bibliografía, primero exponiendo de forma resumida la historia del área de deduplicación de datos para entender su estado en la actualidad y luego investigando el estado del arte.
- Capítulo 4: Se presenta el diseño del software desarrollado, exponiendo los desafíos enfrentados y describiendo sus características.
- Capítulo 5: Se evalúa el rendimiento de la solución desarrollada, procesando varios tipos de datos y comparando los resultados con los esperados por algoritmos de estado del arte. Adicionalmente se investiga el impacto de la aplicación del filtro de deduplicación en las tasas de compresión y los tiempos de ejecución total al usarlo como preprocesador para métodos de compresión general.
- Capítulo 6: Se exponen nuestras conclusiones en virtud de las evaluaciones anteriores. Se plantean posibles mejoras y oportunidades de profundización o continuación de estudio relacionadas al software desarrollado y al trabajo de investigación realizado.

Capítulo 2

Marco teórico

2.1. Deduplicación y características básicas

La deduplicación es una subdisciplina de la compresión de datos que se centra en la reducción del tamaño de los datos al omitir porciones repetidas. La eliminación de datos redundantes ha existido en los procesos de compresión desde los inicios, pero el crecimiento de los volúmenes de datos y la emergente complejidad de los sistemas de almacenamiento de datos (sistemas de red, cloud, distribuidos, de archivo, etc) motiva el estudio de las técnicas de deduplicación de forma enfocada e independiente de otras técnicas de compresión.

2.1.1. Objetivo y evaluación de rendimiento

El objetivo de los sistemas de deduplicación es entonces la reducción de datos mediante la detección y adecuado manejo de datos duplicados. Por tanto la medida por excelencia para evaluar la eficacia de un proceso de deduplicación es la *tasa de compresión*, usualmente llamado *tasa de deduplicación* para aclarar que los resultados solo refieren a procesos de deduplicación y no incluyen otros tipos de compresión. Esta métrica es la relación entre el tamaño de los datos de entrada y el tamaño de los datos de la salida producida,

$$\text{tasa de deduplicación} = \frac{\text{tamaño original}}{\text{tamaño deduplicado}}$$

donde tasas mayores indican mayor eficiencia del proceso aplicado.

La otra medida fundamental para evaluar el rendimiento de un sistema de deduplicación es el *rendimiento*, esto es, la cantidad de datos procesados por unidad de tiempo, usualmente expresado en *MBps* (megabytes por segundo). Esto es importante porque diferentes casos de uso pueden tener diferentes requerimientos en términos de tiempos de ejecución y costo computacional aceptables. Un método de deduplicación con excelentes tasas de deduplicación pero

rendimiento mediocre puede ser la mejor opción para un sistema de respaldos, mientras que para un sistema de archivos de uso más dinámico los costos computacionales de ese método comprometerían el funcionamiento del sistema y por tanto un método con tasas mediocres pero rendimiento excelente sería más adecuado.

2.1.2. Nivel de aplicación de la deduplicación

La deduplicación puede hacerse en 2 niveles diferentes. El primero es a nivel de archivo, donde se buscan encontrar archivos duplicados dentro de una colección. El segundo nivel es a nivel de *bloques* ('chunk' en inglés). Esto es cuando los datos a procesar son divididos de alguna forma (proceso que se le llama *cortado de bloques*, 'chunking' en inglés) en unidades de datos a un nivel más granular: en bloques. En este nivel entonces se deduplica al omitir bloques duplicados. Las características de estos bloques y método por el que se producen es discutido la sección 2.2.

La deduplicación a nivel de archivo es muy rápida de implementar y procesar, y si bien puede dar muy buenos resultados en situaciones particulares, en general tiene un rendimiento pobre en comparación con la deduplicación a nivel de bloques (Meyer y Bolosky, 2012).

2.1.3. Identificación de datos duplicados

Tanto para la deduplicación basada en archivos como para la basada en bloques, es importante determinar cuándo estas unidades de datos son duplicadas. La forma más directa es realizar comparaciones *byte por byte*. Esto es por lo general inviable para volúmenes no triviales de datos por lo que se utilizan técnicas de hashing de datos.

Computando hashes criptográficamente seguros de los datos, se pueden obtener *firmas* que pueden ser indexadas y usadas para encontrar porciones de datos con firmas de hash coincidentes, en cuyo caso podemos presumir que estamos en presencia de dato efectivamente idénticos a otros ya procesados.

Uso de firmas de hash como test de integridad de los datos

La dependencia en estas comparaciones usando firmas de hash, sin realizar una comprobación byte por byte es llamado *comparación por hash*. Muchos sistemas no pueden realizar comparaciones byte a byte para confirmar la integridad de los datos por razones de rendimiento. La elección del hash a usar es muy importante. Por ejemplo *SHA-1* es una función de hashing que aún hoy se ve mucho en sistemas informáticos, pero en tiempos recientes se ha visto que puede ser susceptible a ataques de colisión (Google, 2017).

Si nuestro sistema de deduplicación detecta de una colisión de hashes espuria, registraría a un bloque como duplicado de otro a pesar de que los datos no coinciden realmente. Esto llevaría a la corrupción de los datos, ya que al reconstruir los datos se insertaría ese bloque que en realidad no es el original. Es vital entonces elegir funciones de hashing con muy buenas características, de forma que la posibilidad de encontrar estas colisiones espurias sea ínfima al punto de ser prácticamente insignificante.

Si bien hay estudios sobre los peligros del uso de comparación por hash (Henson, 2003), el consenso es que la comparación por hash es razonablemente segura en sistemas de deduplicación (Black, 2006). La comparación por hash es usada en múltiples sistemas en producción y los peligros se pueden hacer aún más improbables usando hashes criptográficos fuertes y modernos como *SHA-256* (Xia, Jiang, Feng, Douglass, y cols., 2016).

Huellas y el uso de hashes débiles

Es también común el uso de hashes criptográficamente débiles pero más rápidos de computar, obteniendo *huellas* de los datos que se usan de manera similar a las firmas obtenidas usando funciones de hash más confiables. Sin embargo, como la colisión de estas huellas no es un indicio tan fuerte de que los datos sean idénticos (dada la debilidad del hashing), por lo que en caso de colisión se deben calcular las firmas de hash de los datos candidatos a ser duplicados o incluso realizar comparaciones byte por byte.

Dependiendo de la redundancia presente en los datos a procesar, esto puede llevar a mayor rendimiento si gran cantidad de los datos no presentan duplicación y por lo tanto no es necesario el cálculo de las firmas de hash para la mayoría de ellos. En caso contrario, se podría llegar a reducir el rendimiento.

2.1.4. Sistema de deduplicación

Un sistema de deduplicación no solo debe identificar archivos o cadenas duplicadas en los datos, usualmente debe almacenarlos, indexarlos y mantener cierta información sobre ellos para poder realizar la deduplicación y luego poder reconstruir los datos.

En la figura 2.1 se puede ver un esquema del flujo de trabajo de un sistema típico de deduplicación de datos a nivel de bloques. Los datos ingresan al sistema y son cortados en bloques como es descrito en la sección 2.2. Luego a estos bloques se les calculan sus firmas de hash, las cuales son buscadas en los índices del sistema. Si el índice detecta una colisión de hash entonces el bloque es considerado duplicado, en caso contrario se considera nuevo y se ingresa su firma a los índices. Finalmente, si el bloque no era duplicado puede ser sometido a algoritmos de compresión general y luego almacenado en algún tipo de medio.

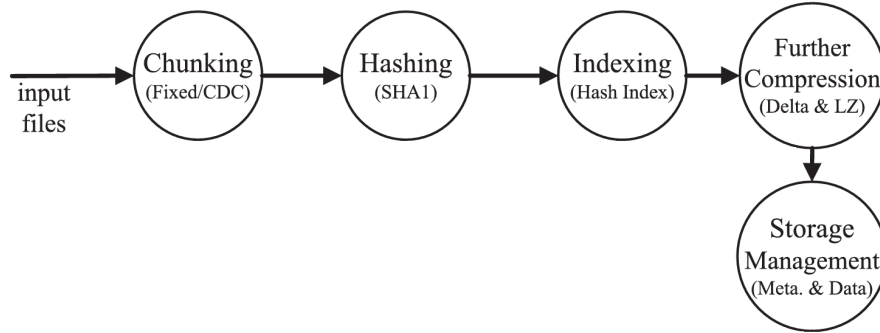


Figura 2.1: Flujo de trabajo de un sistema de deduplicación (Xia, Jiang, Feng, Douglass, y cols., 2016)

En la siguiente sección se describe la etapa de cortado de bloques. En tanto a las etapas de *Hashing*, *Indexing*, *Compresión extra* y *Manejo de almacenamiento* no nos detenemos a presentarlas ya que las consideraciones necesarias para estas fases no son de especial interés para la deduplicación, al ser similares a las usadas por cualquier sistema de almacenamiento de datos no volátiles y las características necesarias para cada una de estas fases son altamente dependientes del caso de uso. El manejo de todos estos aspectos es diferente si la deduplicación se realiza sobre un sistema de archivos, un sistema de archivado de datos o en nuestro caso un filtro secuencial de datos. El corazón de la deduplicación está en la fase de cortado de bloques; cómo se obtienen estos bloques y cómo se comparan nuevos datos contra sus hashes es lo que determina las capacidades de deduplicación de datos del sistema.

2.2. Cortado de bloques

Para la aplicación de deduplicación a nivel de bloques es necesario determinar cómo se va a realizar el cortado de bloques. El método de cortado de bloques usado tiene profundas consecuencias para el resto del proceso de deduplicación y por tanto las tasas de deduplicación y rendimiento resultantes.

2.2.1. FSC (Fixed-Size Chunking)

La primera y más directa alternativa es *Fixed-Size Chunking* (‘Cortado de bloques de tamaño fijo’), usualmente abreviado *FSC*. Para realizar este tipo de cortado de bloques simplemente se elige un tamaño que es utilizado para todos los bloques. La detección de datos duplicados es entonces bastante simple. Al obtener un nuevo bloque se compara su firma o huella contra un índice de los hashes de datos los bloques ya procesados y en caso de colisión obtenemos un bloque que podemos presumir es duplicado del actual, o por lo menos es un

buen candidato a serlo (en el caso de colisión por huellas).

FSC por tanto permite hacer deduplicación de forma muy simple de implementar y con costos computacionales muy reducidos lo que llevan a que estos sistemas pueden tener rendimiento de datos muy elevadas. Por desgracia las tasas de deduplicación son pobres en general, principalmente por el *problema de desplazamiento de bordes*

2.2.2. Problema de desplazamiento de bordes

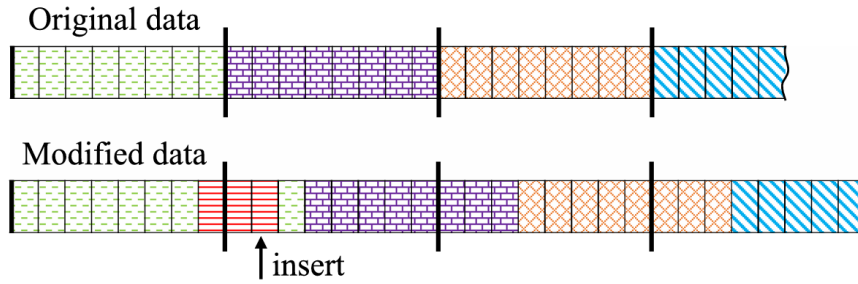


Figura 2.2: Inserción de datos (en rojo) causa el problema de desplazamiento de bordes, ya que FSC ahora produce bloques diferentes(Jin y cols., 2023)

El *Problema de desplazamiento de bordes* ('Boundary-Shift Problem') es uno de los problemas principales a resolver para lograr deduplicación efectiva. Consideremos a modo de ejemplo la cadena 'aaaa bbbb aaaa' y el uso de FSC con bloques de 4 caracteres. Nos quedan 3 bloques: 'aaaa', 'bbbb' y 'aaaa' nuevamente. Es claro que el primero y último bloque son idénticos y podrían ser deduplicados. Ahora consideremos la cadena 'aaaa bbbb c aaaa' donde a la cadena original le insertamos un carácter antes de la repetición de 'aaaa'. En este caso FSC determina que los bloques son: 'aaaa', 'bbbb', 'caaa' y 'a', lo cual hace imposible deduplicar la segunda ocurrencia de la cadena 'aaaa' ya que es separada en diferentes bloques con hashes que no coinciden. En la figura 2.2 se muestra una inserción de este tipo y se puede observar gráficamente al problema de desplazamiento de bordes, literalmente se puede ver como los bloques originales, identificados cada uno por un color diferente, quedan desplazados en relación a los bordes delimitados por las líneas negras gruesas que identifican las fronteras de los cortes de FSC.

Este es el problema de desplazamiento de bordes: el uso de esquemas inocentes como FSC normalmente producen resultados ineficientes ya que pequeñas inserciones o borrados entre los datos a comparar eliminan la posibilidad de deduplicar los datos. En casos patológicos (pero reales) la inserción de un solo

byte puede costar gigabytes de datos duplicados que no son detectados.

Afortunadamente, han surgido técnicas para mitigar este problema en gran medida, de forma que inserciones o borraduras de datos tienen impactos limitados en las tasas de deduplicación.

2.2.3. Rolling Hashes

Para lidiar con el problema de desplazamiento de bordes es necesario aplicar métodos más complejos durante la fase de cortado de bloques o la fase de detección de datos duplicados. Estos métodos recurren en general al cálculo de hashes para ciertas ventanas de datos en todos y cada uno de los bytes de los datos de entrada. La elección del función de hash es entonces clave, ya que los costos computacionales del cálculo de tantos hashes puede ser prohibitivo, a menos que se use una función de hash particularmente apta para este uso.

Un *Rolling Hash* es una función de hashing que se aplica secuencialmente en una ventana de datos deslizante de tal forma que al avanzar la ventana de una posición a la siguiente, se puede usar el resultado del hash en la posición anterior para calcular el hash de la posición actual con un costo computacional muy reducido.

Tomemos como ejemplo sencillo la suma de los bytes dentro de la ventana de bytes como función hash. Si c_i es el valor del byte en la posición i , α es el tamaño de la ventana de datos y n es la posición del byte actual (el final de la ventana de datos) tenemos

$$S(n) = \sum_{i=n-\alpha}^n c_i,$$

y al avanzar a la siguiente posición tenemos

$$S(n+1) = \sum_{i=n+1-\alpha}^{n+1} c_i = S(n) - c_{n-\alpha} + c_{n+1}.$$

Por tanto si disponemos del hash anterior no necesitamos sumar los α bytes, sino que podemos tomar el valor anterior y simplemente restarle el valor del byte que sale de la ventana de datos y sumarle el valor del byte que entra. Es claro que excepto para valores muy pequeños de α esto implica una reducción en la cantidad de operaciones necesarias para el cálculo del hash.

Este hash en particular, por su simpleza, tiene malas cualidades (como gran cantidad de colisiones) y por tanto no es usado en deduplicación, pero ejemplifica de forma simple el concepto de rolling hash.

En la práctica el rolling hash más utilizado para deduplicación es el algoritmo de Rabin ([Rabin, 1981](#)). Las *Rabin Fingerprints* pueden ser computadas en base de una anterior mediante las siguientes operaciones por cada byte: 2 XOR, 1 OR, 2 *left shift* y 2 *lookups* de array ([Xia y cols., 2014](#)).

El uso de rolling hashes permite entonces el cálculo de hashes en cada byte de los datos de entrada de forma eficiente (tiempo de ejecución de orden lineal), lo que es explotado por los métodos de deduplicación para evitar el problema de desplazamiento de bordes.

2.2.4. Deduplicación por ventana deslizante

La técnica de *Deduplicación por ventana deslizante* permite detectar datos duplicados, aliviando los efectos del problema de desplazamiento de bordes sin necesitar cambiar el cortado de bloques FSC. El primer algoritmo conocido de este tipo es RZip, una adaptación del algoritmo RSync usado comúnmente para la sincronización de archivos remotos. ([Tridgell, 2000](#)).

Se usa un rolling hash sobre una ventana de datos de tamaño igual a la de los bloques, deslizada byte por byte y se usan las huellas para encontrar una coincidencia con algún bloque conocido por el sistema.

El problema de desplazamiento de bordes se puede entonces evitar, ya que el cálculo de los hashes y prueba de colisión en todos y cada uno de los bytes garantiza que los bloques duplicados puedan ser detectados sin verse afectados por posibles inserciones o borraduras de datos. Es el uso de rolling hashes en la fase de detección, no de cortado de bloques que permite lidiar con el problema con esta técnica.

Sin embargo, se requiere el cálculo y, más importantemente, la prueba de colisiones para todos y cada uno de los bytes del flujo de entrada, lo que puede ser computacionalmente muy costoso.

2.2.5. Corte de bloques por contenido

El *corte de bloques por contenido*, abreviado *CDC* por su sigla en inglés (Content-Defined Chunking), fue originalmente propuesto en LBFS ([A. Muthitacharoen y Mazieres, 2001](#)). Ataca el problema de desplazamiento de bordes durante la fase de cortado de bloques definiendo puntos de corte en términos de los contenidos de los datos de forma que los bloques pueden ser de tamaño variable. Consiste en usar algún criterio de corte, en la localidad del byte a considerar, para definir las fronteras de los bloques. Esto es, una cantidad limitada de bytes previos y/o subsecuentes, incluyendo el byte mismo, influyen en la decisión de cortar un bloque en esa posición o no.

Las implementaciones y técnicas más comunes están basadas en rolling hashes. Los rolling hashes generan huellas en base a bytes dentro de la ventana de datos desde el byte actual, por lo que imponer un criterio de corte sobre estas huellas es, en definitiva, imponer un criterio de corte en base a contenidos.

Condición sobre huellas de un rolling hash para CDC

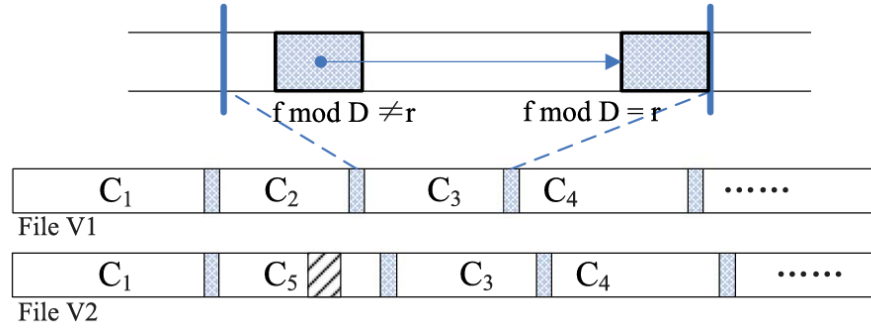


Figura 2.3: Content-Defined Chunking (CDC) encontrando datos similares en dos archivos, evitando el problema de desplazamiento de bordes (Xia, Jiang, Feng, Douglass, y cols., 2016)

La definición de condiciones de corte es importante, entre otras cosas, porque determina el tamaño de los bloques que se generan. Supongamos que queremos bloques de tamaño aproximadamente ϕ y modelemos a las huellas como números aleatorios uniformemente distribuidos (lo que es una aproximación razonable para funciones de hash apropiadas).

Para este fin, necesitamos una condición que se cumpla con probabilidad $p = 1/\phi$. La función $fp \bmod \phi$ divide el espacio de todas las posibles huellas de hash fp en ϕ ‘cubetas’ del mismo tamaño (en tanto ϕ sea potencia de 2). Esto significa que la probabilidad de fp de encontrarse en una de estas ‘cubetas’ en particular es de $1/\phi$. Por lo tanto, si definimos como condición de corte $fp \bmod \phi = r$, para algún r fijo $0 \leq r < \phi$, tenemos la condición deseada, que nos da bloques de tamaño promedio ϕ .

Al realizar la fase de cortado de bloques de esta forma se evita en gran parte el problema de desplazamiento de bordes ya que inserciones o borrados en los datos solo distorsionan el flujo de datos en alguna localidad del evento, por lo que eventualmente se llega a bytes donde su localidad no fue distorsionada y por tanto los bloques se vuelven a cortar de la misma forma como antes, como se puede ver en la figura 2.3.

Vemos entonces que en CDC el problema de desplazamiento de bordes es atacado durante la etapa de cortado de bloques, las pruebas de colisión de hash no se realizan byte por byte, sino que el hash del bloque completo se busca en el índice de bloques procesados anteriormente para detectar duplicaciones, lo que es computacionalmente mucho más eficiente.

Imposición de tamaños mínimos y máximos para bloques en CDC

Es común imponer condiciones de corte de tamaño mínimo, ya que si los bloques son demasiado pequeños se puede incurrir en costos demasiado grandes de metadatos e indexado de las firmas. También es común imponer cotas superiores para el tamaño de bloque, para evitar casos patológicos donde podría nunca satisfacerse la condición de corte y por lo tanto los bloques podrían ser indefinidamente grandes.

Capítulo 3

Revisión de antecedentes

3.1. Breve historia de la teoría de la compresión

Esta sección es en gran parte una adaptación y resumen de (Xia, Jiang, Feng, Douglas, y cols., 2016).

3.1.1. Aspectos básicos de compresión de datos

La compresión de datos ha sido formalizada como teoría desde los orígenes de la teoría de la información. En su publicación (Shannon, 1948), Claude E. Shannon presenta el concepto de entropía de una fuente estocástica de datos y establece una representación óptima de estos datos (en términos del uso de menor cantidad media de bits para su representación). Queda de esta forma determinado el dominio de la compresión de datos sin pérdida, que es el estudio y desarrollo de técnicas con las cuales se pueden transformar representaciones de datos a representaciones que requieren menor cantidad esperada de bits.

Desde ese punto de partida se produjo el desarrollo de técnicas fundamentales para la compresión de datos que siguen en uso hasta el día de hoy, como por ejemplo la codificación de Huffman (Huffman, 1952) y la codificación aritmética (Rissanen, 1976). Estas técnicas requieren conocer o estimar una distribución de probabilidad que gobierna la fuente de datos, lo cual no siempre es práctico. Esto motivó el desarrollo de técnicas de compresión universal, como por ejemplo las basadas en modelo de diccionario como el conocido algoritmo LZ77 (Ziv y Lempel, 1977). El formato DEFLATE (Deutsch, 1996) combina LZ77 con codificación de Huffman, y es posiblemente el formato de compresión de mayor adopción en la historia, al ser utilizado por ficheros y formatos tan populares como ZIP, GZIP y PNG (entre otros). Incluso formatos significativamente más recientes, como ZStandard (Collet, 2021) usan una combinación de LZ77 con otras técnicas.

3.1.2. Compresión Delta

Hasta este punto hemos considerado los elementos básicos de la compresión de datos en general. Sin embargo las necesidades de almacenamiento y transmisión de datos presentan otros problemas que requieren soluciones específicas. Consideremos el caso de uso donde una persona tiene un Laptop y una PC de escritorio y queremos que sus archivos (en sus versiones actualizadas) estén disponibles en ambas de forma indistinta. Es claro que transmitir la totalidad de los datos de un computador al otro constantemente tiene un costo prohibitivo. Otro problema similar ocurrió fruto de la distribución de Software a finales de los 80 y principios de los 90, donde usando CDs se pudo distribuir Software que hubiera sido impráctico distribuir usando el internet de la época. Si se necesitaba distribuir una versión actualizada del Software, era entonces impráctico que los usuarios pudieran descargar la nueva versión en su totalidad, y enviar CDs actualizados por correo (que es algo que de hecho se hacía) tiene grandes costos.

Estas problemáticas llevaron a la siguiente idea: usar datos ya disponibles para reducir los datos que deben ser transferidos. De esta manera no es necesario enviar todos los datos entre una PC y un Laptop para mantener sus datos sincronizados, ni es necesario transmitir la totalidad de la nueva versión de un paquete de Software. Solo los datos que han cambiado, la diferencia (o *delta*), necesita ser transmitida o almacenada. Nos detenemos a ver los algoritmos RSync y XDelta, hitos importantes en la línea de tiempo de la compresión delta, que son adecuados para solucionar los problemas de sincronización de datos y actualización mencionados e introducen conceptos que luego serían fundacionales y reutilizados en el área de deduplicación.

RSync

RSync ([Tridgell y Mackerras, 1996](#)) es un algoritmo de sincronización de sistemas de archivos. En primer lugar realiza chequeos basados en los metadatos disponibles por el sistema de archivos, para identificar archivos faltantes o potencialmente modificados. Comúnmente se comparan los tamaños de archivo y fechas de modificación para archivos cuya ruta del sistema de archivos de origen y objetivo coinciden. Si el tamaño de archivo o fecha de modificación difieren, entonces se realiza un chequeo basado en los contenidos de los archivos. Es aquí donde la compresión delta ocurre, el archivo de origen es dividido en bloques usando FSC, por lo que quedan bloques de un tamaño predefinido (usualmente 4KB) y se calculan las huellas de estos bloques usando un rolling hash llamado Adler32. Las huellas son transmitidas (o simplemente usadas si el sistema de archivo no es remoto y no se usan las capacidades de Cliente-Servidor de RSync) para usarse en la comparación de los datos del archivo objetivo. Se desliza entonces el rolling hash sobre el archivo objetivo, buscando sobre cada byte una coincidencia entre la huella generada y alguna de los bloques recibidos del archivo del sistema de origen. Los datos que coinciden son verificados usando firmas de hash (producidas por el algoritmo MD5, aunque implementaciones modernas

pueden usar otros), y si son efectivamente idénticos su transmisión es omitida. De esta manera RSync permite la sincronización de sistemas de archivos (posiblemente remotos) de manera eficiente, omitiendo la transferencia innecesaria de datos coincidentes.

XDelta

XDelta (MacDonald, 2000) es un algoritmo de codificación delta que analiza un archivo original contra otro ‘actualizado’ y produce un *archivo delta* (comúnmente conocido como *parche*) generando instrucciones COPY/INSERT (los datos coincidentes entre ambas versiones son copiados mientras los datos nuevos son insertados). Usa una adaptación bastante directa del algoritmo RSync para encontrar datos coincidentes para generar instrucciones COPY, aunque usando tamaños de bloques más pequeños. Este algoritmo fue el estado del arte y ha sido usado como estándar de industria para distribución de parches o actualizaciones para Software, entre otras aplicaciones, por más de una década.

Compresión delta como precursor de la deduplicación

Notar cómo los algoritmos RSync y XDelta se basan en encontrar datos idénticos y omitirlos, con pequeños ajustes para adecuarse a sus casos de uso. No es sorprendente entonces que estos algoritmos terminen teniendo muchos aspectos en común o incluso ser en su totalidad muy similares a los de deduplicación. Si bien la deduplicación se parece más al caso general en términos de que no hay datos de origen y de objetivo como en RSync y XDelta, sino por lo general un repositorio único de donde se ingresan o extraen datos, la forma de encontrar datos duplicados van a tomar a estos algoritmos como punto de partida. Efectivamente, la separación de los datos en bloques, la computación de huellas y firmas de hash y el uso de rolling hashes son técnicas recurrentes de la deduplicación.

En particular la deduplicación con la técnica de deduplicación por ventana deslizante 2.2.4 está basada en una adaptación del algoritmo RSync. De hecho, el mismo autor describe el algoritmo RZip (Tridgell, 2000) que es precisamente este tipo de adaptación, donde el algoritmo RSync es aplicado sobre un mismo archivo, y las coincidencias son usadas para comprimir el archivo agregando los bloques repetidos a un diccionario de compresión.

3.1.3. Deduplicación

A principios de los 2000, en el contexto de un constante aumento de volúmenes de datos, se hizo necesario el desarrollo de soluciones escalables y transparentes para la reducción de estos volúmenes de datos. La deduplicación una herramienta para este fin, omitiendo el almacenamiento y/o transferencia duplicada de estos, de forma eficiente y eficaz. En particular los costos computacionales suelen ser inferiores a los de técnicas de compresión generales y pueden

encontrar duplicaciones de datos a distancias más largas. Las técnicas de deduplicación logran estas características construyendo índices de los hashes de los bloques de datos, evitando mantener los datos en sí mismos en memoria RAM, lo cual permite detecciones de cadenas más largas y encontradas a mayores distancias. Como contraparte, este procesamiento a escala gruesa en general es incapaz de eliminar redundancia más fina, como sí lo hacen algoritmos de compresión general.

Nuevamente presentamos dos ejemplos pioneros de aplicaciones del área.

Venti

Venti (Quinlan y Dorward, 2002) es un sistema de archivado que guarda los datos realizando cortado de bloques FSC. Es uno de los primeros sistemas que implementan deduplicación en datos a gran escala, siendo una implementación de deduplicación basada en FSC sin mayor sofisticación. Por tanto sufre del problema de desplazamiento de bordes y sus tasas de deduplicación no son particularmente buenas en general. Sin embargo los resultados obtenidos por Venti mostraron satisfactoriamente cómo incluso implementaciones poco sofisticadas de deduplicación pueden resultar en reducciones en el tamaño de los datos a almacenar, con costos muy bajos de cómputo y de complejidad de desarrollo.

LBFS

LBFS (A. Muthitacharoen y Mazieres, 2001) (*Low Bandwidth File System*) es un sistema de archivos especialmente diseñado para su uso eficiente como almacenamiento de red. En particular, y como su nombre indica, su objetivo es reducir la transferencia de datos requerida de tal forma que el sistema de archivos sea efectivamente utilizable en ambientes de bajo ancho de banda (donde las velocidades de transferencia disponibles son escasas). Para esto usa técnicas comunes de compresión, y de forma más relevante, deduplicación de datos. Datos ya disponibles en el servidor no necesitan ser enviados si estos pueden ser identificados, y esto ocurre de forma transparente a nivel del sistema de archivos.

LBFS implementa cortado de bloques CDC y por tanto tiene en general buenas tasas de deduplicación, al no sufrir en gran medida del problema de desplazamiento de bordes. En el caso de uso de LBFS, entornos donde las tasas de transmisión de datos son el claro cuello de botella, la complejidad agregada y costos computacionales incrementados por este tipo de deduplicación (aplicada de forma conjunta con otros métodos de compresión) terminan representando una relación de compromiso muy favorable.

3.2. Estado del arte

En los años recientes se han propuesto diferentes alternativas y mejoras con las cuales se pueden incrementar el rendimiento o tasas de deduplicación compa-

rado con los métodos ya tradicionales de deduplicación usando CDC con *Rabin fingerprints* (usualmente llamado *RabinCDC*) o deduplicación por ventana deslizante.

Presentamos una selección de propuestas disponibles en la bibliografía académica que representan mejoras significativas sobre los procesos de deduplicación tradicionales basados en RabinCDC o RZip y que por tanto consideramos que deben ser integradas en nuestra solución para estar a la altura del estado del arte. Esta selección es no exhaustiva, pero incluye las referencias que consideramos más importantes en cuanto al balance entre su complejidad algorítmica y de implementación contra los beneficios de rendimiento y/o eficiencia que proporcionan.

3.2.1. FastCDC

FastCDC (Xia y cols., 2020) es una consolidación de mejoras sobre CDC tradicional basado en Rabin, que las combina de forma conjunta para obtener rendimiento muy superior mientras mantiene tasas de deduplicación muy similares. En los resultados presentados se aprecia una aceleración de 300 % en comparación con CDC tradicional basado en Rabin.

Gear rolling hash

Una de las diferencias fundamentales de FastCDC con sus predecesores es el intercambio de la función usada para computar las huellas con el rolling hash. En vez de usar Rabin se usa Gear que computa las huellas basandose en 256 números aleatorios de forma que el cómputo de una huella usando una anterior es $fp = (fp \ll 1) + Gear[b]$ donde b es el byte actual y $Gear[b]$ es un valor predefinido para cada posible byte b (Gear dispone de una tabla con 256 números aleatorios, uno por cada posible byte). El cálculo de fp se reduce módulo 2^n , donde n es la cantidad de bits usados para la representación de las huellas. Gear puede computar las huellas usando solo una adición, un left shift y un lookup de array, mientras que Rabin en comparación requiere adicionalmente al menos otro lookup de array y 2 operaciones sobre bits. Por tanto el uso de Gear termina representando un incremento substancial en performance.

Juicio de corte optimizado

El criterio de corte es cambiado de $fp \bmod D = r$ (para un D y r elegidos para obtener un tamaño promedio deseado de bloque) a $fp \& mask = 0$ (para una máscara de bits $mask$ fabricada para lograr un tamaño promedio de bloque deseado), donde ‘&’ denota la operación AND bit a bit. La omisión de la reducción modular y la comparación con r permiten juzgar las huellas más eficientemente para encontrar los puntos de corte de los bloques.

Salto al tamaño de bloque mínimo

FastCDC impone límites en el tamaño mínimo y máximo de los bloques, para evitar casos patológicos donde CDC crea bloques con tamaños muy alejados del promedio deseado. Ya que FastCDC impone un tamaño mínimo, usa la oportunidad para omitir el cálculo de las huellas para esa cantidad de bytes al principio de cada bloque.

Cortado de bloques normalizado

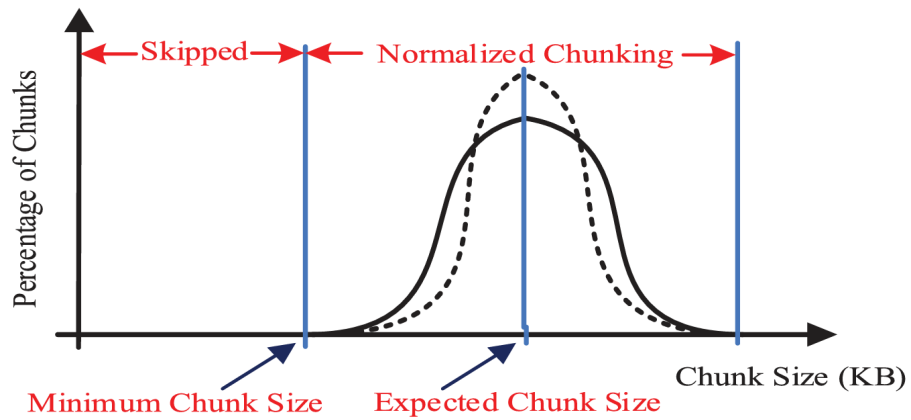


Figura 3.1: Diagrama conceptual del cortado de bloques normalizado combinado con *Salto al tamaño de bloque mínimo*. La línea punteada representa la mejora en la normalización de los bloques (Xia y cols., 2020)

La imposición de límites de tamaño mínimo y máximo en los bloques solucionan ciertos problemas pero por otro lado reducen la efectividad del proceso de deduplicación. Esto es porque se pierde en cierta medida la propiedad de corte definida estrictamente por contenido, ya que se realizan o dejan de realizar cortes según otros criterios (en este caso, el bloque es demasiado pequeño o grande). La técnica de cortado de bloques normalizado introducida por FastCDC busca solucionar este problema reduciendo la cantidad de bloques que deben ser cortados por estos límites estrictos, de forma que se mantiene la propiedad de corte definida por contenidos.

Para lograr esto FastCDC reemplaza la máscara de bits *mask* usada para la condición de corte dependiendo de si el bloque ya alcanzó un tamaño al menos igual al promedio requerido o no. Si se está por debajo del tamaño promedio se usa una máscara de bits que hace que la condición de corte sea más difícil de satisfacer y por tanto menos común que los bloques tengan tamaño muy inferior al promedio. Luego si el bloque ya tiene el tamaño promedio se usa una

máscara de bits que hace que la condición de corte sea más fácil de satisfacer, lo que hace menos probable que los bloques tengan tamaño muy superior al promedio y por tanto deban ser cortados forzosamente por el límite superior. En la figura 3.1 se puede ver una representación gráfica del efecto del cortado de bloques normalizado y cómo la distribución del tamaño de los bloques queda más concentrada alrededor del tamaño promedio deseado.

De esta forma el cortado de bloques normalizado de FastCDC evita en gran parte los efectos negativos causados por los límites mínimos y máximos aplicados forzosamente sobre los bloques.

3.2.2. SS-CDC

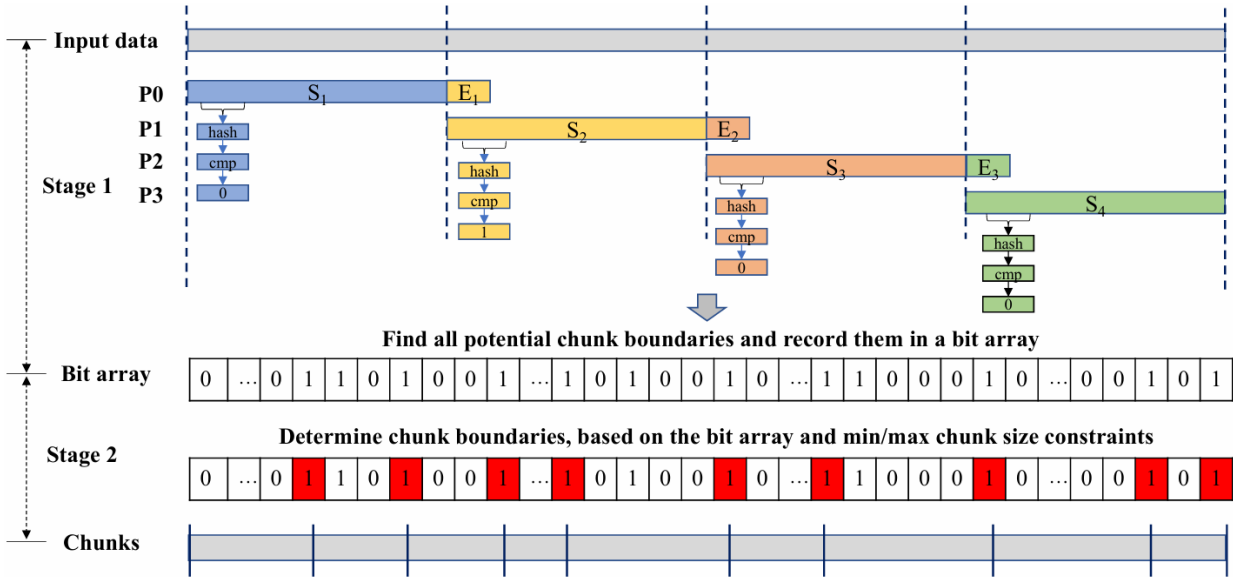


Figura 3.2: Segmentación de SS-CDC para proceso en múltiples hilos, con una primera fase de búsqueda de candidatos y una segunda fase de sección de candidatos. (Ni y cols., 2019)

SS-CDC (Ni y cols., 2019) es la primera mejora que agrega niveles significativos de paralelismo a la deduplicación basada en CDC que no compromete (reduce) las tasas de deduplicación. Logra paralelizar la fase de cortado de bloques dividiendo los datos de entrada entre los núcleos de CPU disponibles pero repitiendo y solapando ciertos datos en las fronteras de los datos proporcionados a cada uno. De esta forma los puntos de cortes encontrados pueden ser consolidados en los mismos que se hubieran encontrado durante un procesamiento en serie, mediante una segunda etapa de reconciliación.

Etapas de segmentación

En esta etapa se segmentan los datos del flujo de entrada de forma que cada hilo dispone de una cantidad fija (tamaño del segmento) de datos y unos cuantos bytes adicionales que extienden el segmento. La cantidad de bytes de extensión es $window_size - 1$, donde $window_size$ es la ventana de contexto del rolling hash usado. Cada hilo encuentra todos los posibles candidatos a puntos de corte en su segmento extendido. De esta forma, en caso de haber un candidato de corte de bloque en los primeros $window_size - 1$ bytes de un segmento, donde aún no hay una ventana completa para calcular una huella, este candidato no se pierde porque es encontrado en los bytes de extensión del hilo anterior.

Etapas de reconciliación

Al finalizar el proceso de cada núcleo de CPU, hay una rápida etapa de conciliación donde se toman todos los candidatos a punto de corte de cada uno, en forma de un arreglo de bits, y se consideran serialmente todos los candidatos a puntos de corte. Gracias a esta etapa de conciliación SS-CDC es capaz de acelerar la deduplicación basada en CDC usando paralelismo, sin incurrir en pérdidas de tasa de deduplicación por los cortes arbitrarios entre datos realizados por la separación de datos para cada núcleo.

En 3.2 se puede ver una representación gráfica de ambas etapas. Los datos son segmentados y enviados para ser ejecutados en este caso en 4 núcleos diferentes. Estos núcleos depositan los candidatos encontrados en el arreglo de bits y finalmente la etapa de reconciliación recorre el arreglo de bits y selecciona (marcado en rojo) los puntos de corte que se habrían encontrado en un procesamiento serial, descartando otros candidatos que son inválidos en virtud de los cortes seleccionados anteriormente y los límites de tamaño mínimo y máximo de los bloques.

Uso de instrucciones SIMD

Adicionalmente SS-CDC logra un nivel aún superior de paralelismo al usar instrucciones de CPU SIMD que son capaces de procesar varios datos en una sola instrucción. Si bien el juego de instrucciones utilizado, AVX-512, puede ser limitante ya que muchos CPUs modernos no lo soportan, incluso de arquitectura x86, el uso de instrucciones SIMD para el proceso de los hashes en la etapa de cortado de bloques es una característica destacable. Esto permite a SS-CDC incrementar el rendimiento por encima de lo que se puede obtener por el mero hecho de usar varios núcleos de CPU, esto es, en algunos casos, usando 8 núcleos se obtuvieron incrementos de 25.6X en comparación con el uso de 1 solo núcleo sin SIMD y 7.7X con multihilo sin SIMD.

3.2.3. SuperCDC

SuperCDC (Wan y cols., 2022) itera sobre algunas de las mejoras presentadas por FastCDC e intentos previos de aceleración del cortado de bloques usando listas con los tamaños de bloques previos y criterios de localidad de la duplicación, que permiten omitir el cálculo de los rolling hashes para ciertos bloques. SuperCDC consolida estas mejoras al mitigar sus puntos débiles, negando la pérdida en tasa de deduplicación causada por el salto al tamaño de bloque mínimo de FastCDC y reduciendo el uso de memoria de las estructuras necesarias para aplicar los criterios de localidad de la duplicación.

Recuperación de contexto tras salto por tamaño mínimo

Luego de un salto al tamaño mínimo del bloque en FastCDC, al inicializar la huella en 0 en ese punto, sin mantener el contexto de los datos previo, se pueden perder condiciones de corte (o detectar condiciones de corte espurias), lo que podría llevar a tasas de deduplicación reducidas ya que los datos no se dividen en bloques según los contenidos sino que terminan siendo determinados en cierta medida por estos saltos. SuperCDC introduce una técnica para recuperar este contexto de la huella luego del salto al tamaño mínimo, con una cantidad pequeña y constante de operaciones, con el objetivo de mejorar las tasas de deduplicación.

Para esto, simplemente se reduce el tamaño del salto en $window_size - 1$, donde $window_size$ es el tamaño de la ventana contexto del rolling hash usado, generalmente GEAR. Luego del salto, se ignora la condición de corte por $window_size - 1$ bytes, calculando el rolling hash de forma normal pero no usándolo para la detección de cortes. De esta forma, ejecutando del orden de $window_size - 1$ operaciones, se recupera el contexto en posición del byte al que se hizo el salto.

Punto de corte de respaldo

SuperCDC introduce el uso de una máscara de bits adicional a las usadas por el cortado de bloques normalizado, para ser usada en caso de que se deba cortar el bloque forzosamente por llegar al tamaño máximo. La máscara representa una condición aún más fácil de satisfacer que la usada por cortado de bloques normalizado luego de llegar al tamaño promedio del bloque, por lo que es altamente probable que esta fuese satisfecha incluso si la otra no lo ha sido. La primera posición que satisface esta condición de respaldo es guardada como el punto de corte de respaldo. En caso de alcanzarse el tamaño máximo de bloque sin que se satisfaga la condición de corte regular, se recurre al punto de corte de respaldo (si éste existe). Para estos casos, se evita la pérdida de las propiedades de los cortes basados en contenido que se habría producido de necesitar aplicar el corte por tamaño máximo, y por tanto, se aumenta potencialmente la eficiencia de deduplicado.

Lista de adyacencia para acelerar cortado de bloques

RapidCDC (Ni y Jiang, 2019) estudia el fenómeno de *localidad de la duplicación*, respaldado por resultados experimentales, que indica que con alta probabilidad un bloque duplicado es precedido y/o sucedido por otros bloques duplicados. Para aprovechar este fenómeno plantea una estructura auxiliar en memoria que permite usualmente saltarse por completo el cómputo de los rolling hashes al encontrar un bloque duplicado, si puede validar que el siguiente bloque también es duplicado y particularmente es un duplicado del siguiente del cual el actual bloque es duplicado.

SuperCDC optimiza el uso de recursos de esta técnica, usando huellas en vez de firmas para los hashes en la estructura secundaria (que ocupan menos espacio en memoria) y solo registrando el tamaño del siguiente bloque a la última ocurrencia de un bloque en particular, mientras que RapidCDC guardaba una lista con varios candidatos de siguiente bloque para cada bloque. Estos ajustes reducen significativamente la cantidad de memoria necesaria para la estructura secundaria mientras mantiene casi la misma capacidad de acelerar el proceso que RapidCDC.

3.2.4. DARE

DARE (Xia, Jiang, Feng, y Tian, 2016) presenta una técnica donde la información obtenida durante un proceso de deduplicación es subsecuentemente usada para encontrar bloques candidatos para la aplicación de compresión delta, esto es, bloques que si bien no son duplicados tienen una buena medida de similitud y por tanto redundancia de datos entre sí.

Esta técnica es llamada *Duplicate-Adjacency based Resemblance Detection and Elimination scheme (DARE)* y consiste en la realización de chequeos de similitud entre bloques que son adyacentes a otros bloques que fueron detectados como duplicados, sólo usando otros mecanismos tradicionales de detección de similitud para aquellos bloques para los que los criterios de duplicación y de adyacencia no fueron satisfechos. Este criterio de adyacencia (*Duplicate-Adjacency*, abreviado como *DupAdj*) es la principal innovación de DARE. Estos chequeos de similitud directos son comparativamente mucho más baratos computacionalmente que los procesos normalmente requeridos de detección de similitud para obtener candidatos para compresión delta.

En la figura 3.3 se puede ver el flujo de este proceso, en una primera etapa A se detecta la duplicación de los bloques, marcada en rojo en el diagrama de los datos de entrada, y esta información se usa en la etapa B para aplicar DupAdj, como se puede ver para los bloques marcados en verde, adyacentes a los bloques rojos duplicados. Únicamente para los bloques para los que duplicación y DupAdj no ha aplicado se aplican otras estrategias de detección de similitud.

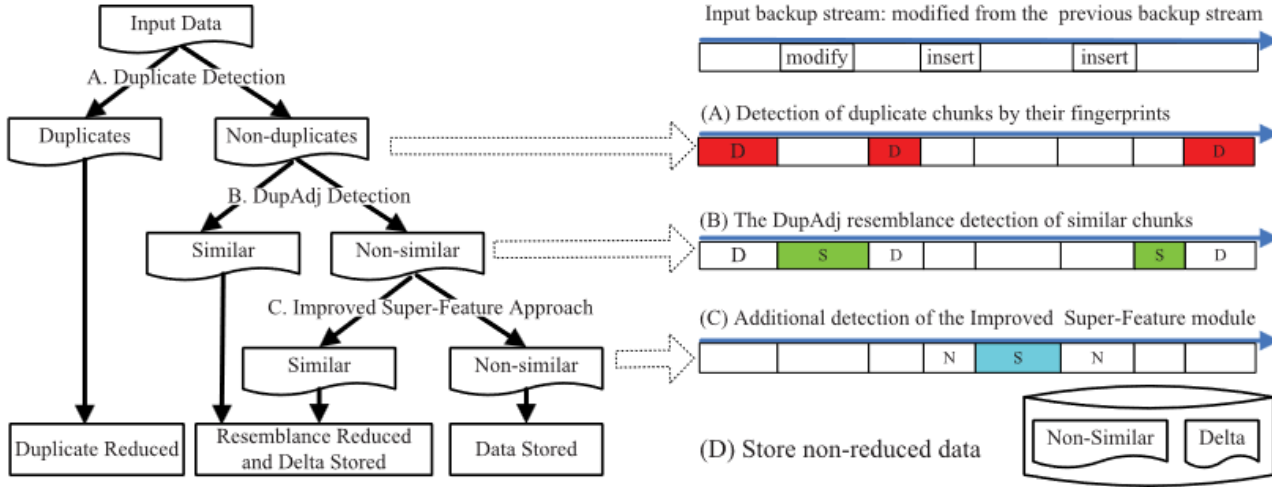


Figura 3.3: Sistema de reducción de redundancia propuesto por DARE

Si bien DARE usa la técnica propuesta DupAdj en conjunción con una versión optimizada de las técnicas tradicionales de compresión delta, en el estudio (Xia, Jiang, Feng, y Tian, 2016) presentan resultados de la aplicación de DupAdj como única estrategia de detección de similitud, con reducciones logradas de un 10 % a 50 % sobre la reducción lograda por la deduplicación en varios conjuntos de datos. Esto representa ganancias significativas, disponibles a costos computacionales muy limitados, si estamos dispuestos a usar la información de los datos obtenida durante el proceso de deduplicación.

Capítulo 4

Diseño de la solución

Habiendo reexaminado el arte previo y en particular el estado del arte en deduplicación, diseñamos e implementamos un compresor al que llamamos CDCZip. CDCZip está inspirado en el mismo criterio de diseño que RZip/LRZip, pero para la detección de datos duplicados el algoritmo RZip es reemplazado por las técnicas basadas en CDC de estado del arte expuestas anteriormente.

A excepción de la estructura secundaria para aceleración usada por SuperCDC (por sus costos en uso de memoria y dificultad de adaptación a un proceso multihilo) y el proceso SIMD de SS-CDC usando instrucciones AVX-512 (dado la escasa disponibilidad de CPUs con estas instrucciones), implementamos todas las mejoras planteadas por FastCDC, SuperCDC y SS-CDC.

La implementación de estas técnicas de forma aislada no representaría mayores problemas ya que se pueden implementar acorde a las instrucciones disponibles en sus respectivas publicaciones. Los desafíos surgen de la combinación de éstas, y de incompatibilidades que presentarían al ser implementadas sin adaptaciones.

4.1. Desafíos y sus resoluciones

4.1.1. Múltiples tipos de condiciones de corte

FastCDC y SuperCDC pueden realizar el corte de los bloques por diferentes condiciones que a su vez aplican de forma condicional. Por ejemplo la condición de corte relajada luego del tamaño promedio de FastCDC solo es válida luego de que el bloque siendo procesado tenga al menos el tamaño promedio esperado. Otro ejemplo es la condición de corte de respaldo de SuperCDC, que sólo es válida si no se pudo satisfacer otra condición de corte (otra que la de tamaño máximo).

Estas múltiples condiciones de corte y sus condicionales aplicaciones a priori no pueden ser respetadas por SS-CDC que usa un único mapa de bits para representar los candidatos para corte. Adicionalmente, durante el proceso de SS-CDC en su primera fase, no es sabido qué candidatos a cortes van a ser usados y, por tanto, no es posible saber en esa fase si se está procesando una posición de segmento que excederá el tamaño promedio de bloque una vez que se definan las fronteras de los bloques en la fase 2.

Para resolver este problema simplemente reemplazamos el mapa de bits de SS-CDC por una lista donde los candidatos a punto de corte son ingresados junto con la condición de corte que fue satisfecha para generar ese candidato, las cuales pueden ser la condición de respaldo de SuperCDC o las condiciones relajada y difícil del cortado de bloques normalizado de FastCDC. De esta forma la fase 2 de SS-CDC nuevamente dispone de toda la información requerida para seleccionar los bloques de la misma forma que se hubieran seleccionado en la implementación de un solo hilo sin segmentación.

***Submasking* de las condiciones de corte**

Usamos una técnica llamada Submasking que consiste en elegir máscaras de bits para las condiciones de forma que los bits en 1 de las máscaras para las condiciones más fáciles estén también en 1 en las máscaras para de las condiciones más difíciles de satisfacer. De esta forma, no necesitamos probar las tres condiciones de corte para cada byte, ya que la condición difícil del corte de bloques normalizado solo necesita ser probada si la condición fácil se cumplió, que a su vez solo se prueba si se cumplió la condición de corte de respaldo de SuperCDC. Esto también simplifica nuestra fase 2 de selección de puntos de corte ya que un punto que satisface la condición difícil del cortado de bloques normalizado también satisface la fácil, por lo que puede ser usado indistintamente antes o después de llegar al tamaño promedio del bloque.

4.1.2. Salto al tamaño mínimo del bloque

El salto al tamaño mínimo del bloque introducido por FastCDC es beneficioso para los tiempos de ejecución al poder omitir el proceso de los hashes de gran cantidad de bytes de los datos. Sin embargo, nuevamente por el desconocimiento de qué candidato de corte será usado finalmente por la segunda fase en SS-CDC, integrar esta técnica durante la primera fase de SS-CDC parece imposible, ya que necesitamos saber dónde comienza cada bloque para poder saltar hasta el tamaño mínimo en sus datos.

Con el fin de ser capaces de aplicar esta optimización junto con SS-CDC, investigamos en detalle el proceso de la fase 2, de selección de puntos de corte, de SS-CDC y contrastamos con la información disponible por un hilo particular operando en un segmento durante la fase 1, con el fin de identificar una condición de re-sincronización desde la cuál podamos asegurar que un candidato de corte

de bloque va a ser utilizado y por tanto el hilo puede retornar a un estado de proceso como si fuera el único hilo en ejecución.

4.1.3. Teorema de condición suficiente para la definición de un punto de corte en CDC

En esta sección presentamos una condición suficiente para determinar que una posición candidata a punto de corte por un algoritmo CDC (definida como una posición de un byte el cual satisface la condición de localidad del algoritmo) va a efectivamente ser usada para delimitar un bloque. La condición involucra únicamente a dos candidatos de punto de corte consecutivos c_0 y c_1 .

Ningún candidato a punto de corte anterior a c_0 necesita ser considerado. En particular, si aplicamos la condición suficiente en un esquema de procesamiento de datos segmentado como en SS-CDC, significa que ningún candidato a punto de corte en ningún segmento anterior necesita ser considerado.

Consideramos un algoritmo CDC que consiste únicamente de un criterio en la localidad de los bytes (que determina en cada posición si esta genera un candidato de corte) y los criterios de tamaño mínimo y máximo de los bloques. En particular el teorema presentado a continuación no es válido para algoritmos más complejos o que usan condiciones adicionales como el corte de bloques normalizado de FastCDC o las condiciones de corte de respaldo de SuperCDC. En la sección 4.2 ofrecemos una adaptación del análisis para estos casos.

Teorema 1. *Sean $m > 0$ y $M > m$ los tamaños mínimo y máximo, respectivamente, impuestos para los bloques, y sean c_0 y c_1 dos posiciones candidatas a corte consecutivas obtenidas por un algoritmo CDC. Sea $d = c_1 - c_0$ la distancia entre estas posiciones. Si se cumple $m \leq d < M - m$, entonces c_1 es seleccionada como posición de corte por el algoritmo CDC.*

Demostración. Sea c_x la posición de corte anterior más próxima a c_1 . Como c_0 es la posición candidata consecutivamente anterior a c_1 o bien es seleccionada por lo que $c_x = c_0$, ya que por hipótesis $d < M$ y por tanto la cota superior de M para el tamaño de bloques no impone un corte entre c_0 y c_1 , o es rechazada por lo que existe una posición de corte c_z tal que $c_0 - m < c_z < c_0$, porque de lo contrario, c_0 sería seleccionada como posición de corte. Definimos la distancia entre c_z y c_1 como $d_z = c_1 - c_z$ y notamos que por la anterior desigualdad se cumple $c_1 - c_0 < d_z < c_1 - c_0 + m$. Usando la hipótesis $d < M - m$ se obtiene $d_z < d + m < M$. Como en el caso anterior, no se impone un corte por el tamaño máximo de bloque entre c_z y c_1 y por tanto $c_x = c_z$. Por otro lado por la hipótesis $m \leq d$ se tiene $m \leq d < d_z$ por lo que la cota inferior del tamaño de los bloques no implica el rechazo de c_1 , sin importar si $c_x = c_0$ o $c_x = c_z$. En tanto a otros candidatos entre c_x y c_0 es claro que no existen para $c_x = c_0$ y para $c_x = c_z$ por la desigualdad $c_0 - m < c_z$ se deduce $c_0 < c_z + m$ por lo que de existir serían rechazados por la cota inferior del tamaño de bloques. Al no

existir ningún candidato elegible o corte impuesto entre c_x y c_1 , y c_1 ser elegible, concluimos que c_1 es efectivamente seleccionado como punto de corte. $\square$

Corolario 1. *Luego de encontrar un punto de corte que satisface el teorema 1, un hilo de ejecución puede independientemente realizar la selección de los puntos de corte siguientes en su segmento de forma idéntica a como serían seleccionados por la fase 2 de SS-CDC o procesamiento CDC en serie.*

Cómo las condiciones de corte en CDC son elegidas para generar bloques de un tamaño medio predeterminado, que es generalmente substancialmente mayor que el tamaño mínimo y a su vez substancialmente menor que el tamaño máximo, las condiciones del teorema 1 no son difíciles de satisfacer. Experimentalmente también observamos que nuestra implementación pasa por debajo del 1 % del tiempo de ejecución mientras los hilos buscan satisfacer esta condición.

Podemos implementar la optimización de salto al tamaño mínimo del bloque luego de encontrar un punto c_1 que satisface a la condición del teorema 1, ya que desde ese punto podemos asegurar que los saltos no descartan ningún candidato a punto de corte que podría haber sido utilizado.

Otras aplicaciones

Es importante notar que el teorema 1 puede tener otras aplicaciones a las usadas en nuestra solución.

En determinadas aplicaciones, luego de satisfacer la condición del teorema 1, los hilos podrían generar bloques que inmediatamente podrían ser enviados y/o indexados en un sistema de almacenamiento, sin necesitar que se termine el proceso de otros segmentos ni necesitar de la fase 2 de SS-CDC. La fase 2 sigue siendo necesaria para reconciliar y terminar el corte de bloques en las regiones de los segmentos, donde para los hilos la condición del teorema 1 aún no fue satisfecha, pero gran parte de los datos procesados podrían saltarse esta limitación.

4.2. CDCZ

Llamamos a nuestro algoritmo CDCZ (dado que es el algoritmo que es la base de CDCZip), y usamos las soluciones anteriormente descritas para integrar nuestro CDC basado en rolling hash GEAR con todos los beneficios descritos de FastCDC, SuperCDC y SS-CDC.

Los hilos, al recibir segmentos de datos, comienzan ejecutando una etapa simplificada con el algoritmo 1 donde para cada byte se calcula su hash GEAR y se prueban las condiciones de corte de cortado de bloques normalizado y de respaldo. Si alguna condición de corte es satisfecha se prueban todas en orden

Algoritmo 1 Corte de bloques CDCZ previo a sincronización (CDCZ_UNSYNCED)

Input buffer de datos, src ; largo del buffer, n ; huella GEAR, fp ;
máscara de respaldo, $maskB$; máscara difícil, $maskS$; máscara fácil, $maskL$

Output tupla (candidato de corte, tipo de candidato)

```

1: for  $i \leftarrow 0$  to  $n$  do
2:    $fp \leftarrow (fp \ll 1) + GEAR[src[i]]$ 
3:   if  $!(fp \& maskB)$  then
4:     if  $!(fp \& maskL)$  then
5:       if  $!(fp \& maskS)$  then
6:         return  $(i, \text{HARD})$ 
7:       else
8:         return  $(i, \text{EASY})$ 
9:       end if
10:    else
11:      return  $(i, \text{BACKUP})$ 
12:    end if
13:  end if
14: end for
15: return  $(i, \text{EOF})$ 

```

de dificultad y se retorna como candidato según la condición más estricta que se cumple.

Eventualmente el hilo logra satisfacer las condiciones del teorema 1 y por tanto cambia a una nueva etapa con el algoritmo 2, dado que se logró sincronizarse con lo que se habría procesado con un algoritmo normal de un solo hilo. Precisamente, esta nueva etapa ejecuta de manera indistinguible a que fuera el único hilo en ejecución.

El proceso conjunto de estas 2 etapas (algoritmo 3) acumula todos los candidatos de corte encontrados, tanto antes de la sincronización, como después.

Luego, como en SS-CDC, se ejecuta una etapa de reconciliación que presentamos como algoritmo 4. El uso de nuestro teorema no es suficiente para prescindir de ella, ya que sigue siendo necesario realizar cierta conciliación entre los candidatos encontrados al final de un segmento y al principio del siguiente, donde aún no se usa el algoritmo sincronizado. La etapa rechaza candidatos de corte inválidos e inserta cortes por tamaño máximo que no fuesen encontrados por los hilos en sus segmentos.

Finalizada esta última etapa, CDCZ logra seleccionar los puntos de corte de igual forma que lo haría el algoritmo FastCDC, pero enriquecido con la condición de respaldo y recuperación de contexto de SuperCDC, usufructuando el paralelismo sin penalización en las tasas de deduplicación al estilo SS-CDC.

Algoritmo 2 Corte de bloques CDCZ sincronizado (CDCZ_SYNCED)

Input buffer de datos, *src*; largo del buffer, *n*;
 máscara de respaldo, *maskB*; máscara difícil, *maskS*; máscara fácil, *maskL*;
 tamaño mínimo, *minSize*; tamaño promedio, *avgSize*; tamaño máximo,
maxSize

Output tupla (candidato de corte, tipo de candidato)

Require: $minSize \geq GEAR_WINDOW_SIZE$

```

1: backup  $\leftarrow n$ 
2: fp  $\leftarrow 0$ 
3: if  $n < minSize$  then
4:   return (n, EOF)
5: end if
6: if  $n \geq maxSize$  then
7:    $n \leftarrow maxSize$ 
8: else if  $n < avgSize$  then
9:    $avgSize \leftarrow n$ 
10: end if
11:    $\triangleright$  Salto al mínimo con recuperación de contexto
12:  $i \leftarrow minSize - (GEAR\_WINDOW\_SIZE - 1)$ 
13: for  $i$  to  $minSize - 1$  do
14:    $fp \leftarrow (fp \ll 1) + GEAR[src[i]]$ 
15: end for
16:    $\triangleright$  Previo a  $i = avgSize$  sólo maskS es válida
17: for  $i$  to  $avgSize$  do
18:    $fp \leftarrow (fp \ll 1) + GEAR[src[i]]$ 
19:   if  $!(fp \& mask\_s)$  then
20:     return (i, HARD)
21:   end if
22: end for
23:    $\triangleright$  Luego de  $i = avgSize$  usamos maskB y maskL
24: for  $i$  to  $n$  do
25:    $fp \leftarrow (fp \ll 1) + GEAR[src[i]]$ 
26:   if  $!(fp \& maskB)$  then
27:     if  $!(fp \& maskL)$  then
28:       return (i, EASY)
29:     else if  $i < backup$  then
30:        $backup \leftarrow i$ 
31:     end if
32:   end if
33: end for
34:    $\triangleright$  No encontrado, usar respaldo o truncar por tamaño máximo
35: if  $backup < n$  then
36:   return (backup, BACKUP)
37: else
38:   return (i, MAX)
39: end if

```

Algoritmo 3 Corte de bloques CDCZ combinado en hilo

Input buffer de datos, *src*; largo del buffer, *n*; offset de comienzo del hilo, *threadOffset*;
 máscara de respaldo, *maskB*; máscara difícil, *maskS*; máscara fácil, *maskL*;
 tamaño mínimo, *minSize*; tamaño promedio, *avgSize*; tamaño máximo, *maxSize*
Output lista de candidatos, *candidateList*

```

1: fp  $\leftarrow$  0
2: candidateList  $\leftarrow$  newList
3: candidateList.push((threadOffset, EOF))
4:    $\triangleright$  Corre CDCZ no sincronizado hasta satisfacer el teorema 2
5: while n > 0 do
6:   newCandidate  $\leftarrow$  CDCZ_UNSYNCED(...params)
7:   prevCandidate  $\leftarrow$  candidateList.last()
8:   candidateList.push((newCandidate[0]+prevCandidate[0], newCandidate[1]))
9:   n  $\leftarrow$  n - newCandidate[0]
10:  src  $\leftarrow$  src + newCandidate[0]
11:  if prevCandidate[1] == HARD and
12:    (newCandidate[1] == HARD and newCandidate[0] >= minSize or
13:    newCandidate[1] == EASY and newCandidate[0] >= avgSize) and
14:    newCandidate[0] + minSize < maxSize then
15:    break
16:  end if
17: end while
18:    $\triangleright$  Ya podemos correr la versión sincronizada de CDCZ
19: while n > 0 do
20:   newCandidate  $\leftarrow$  CDCZ_SYNCED(...params)
21:   prevCandidate  $\leftarrow$  candidateList.last()
22:   candidateList.push((newCandidate[0]+prevCandidate[0], newCandidate[1]))
23:   n  $\leftarrow$  n - newCandidate[0]
24:   src  $\leftarrow$  src + newCandidate[0]
25: end while
26: return candidateList

```

Algoritmo 4 Etapa de reconciliación de candidatos de CDCZ

Input lista de candidatos global, *candidateList*; lista de candidatos del siguiente hilo, *threadCandidateList*; primer offset válido para el hilo, *threadValidOffset*; bandera de último segmento, *isLastSegment*; tamaño mínimo, *minSize*; tamaño promedio, *avgSize*; tamaño máximo, *maxSize*; cola de candidatos de respaldo, *backupCandidatesQueue*

Output Ninguna. (salida depositada en la lista de candidatos global)

```

1: if candidateList.empty() then
2:   candidateList.push((0, EOF))
3: end if
4:
5: while !threadCandidateList.empty() do
6:   newCandidate  $\leftarrow$  threadCandidateList.pop()
7:   if threadValidOffset > newCandidate[0] then
8:     continue
9:   end if
10:  prevCandidate  $\leftarrow$  candidateList.last()
11:  ▷ Truncar bloques con tamaño máximo antes del nuevo candidato
12:  while prevCandidate[0] + maxSize < newCandidate[0] do
13:    ▷ Limpiar viejos backups que ya no son válidos
14:    while !backupCandidatesQueue.empty() and
15:      backupCandidatesQueue.first() < prevCandidate[0] + avgSize do
16:      backupCandidatesQueue.pop()
17:    end while
18:    backup  $\leftarrow$  backupCandidatesQueue.first()
19:    if backup  $\neq$  NULL and
20:      backup < prevCandidate[0] + maxSize then
21:      candidateList.push((backup, BACKUP))
22:      backupCandidatesQueue.pop()
23:    else
24:      candidateList.push((prevCandidate[0] + maxSize, MAX))
25:    end if
26:    prevCandidate  $\leftarrow$  candidateList.last()
27:  end while
28:  ▷ Rechazar o seleccionar el candidato según sea válido o no
29:  if newCandidate[1] == BACKUP then
30:    backupCandidatesQueue.push(newCandidate)
31:  else if newCandidate[0]  $\geq$  prevCandidate[0] + minSize then
32:    if newCandidate[1] == MAX or
33:      newCandidate[1] == HARD or
34:      (newCandidate[1] == EASY and
35:        newCandidate[0]  $\geq$  prevCandidate[0] + avgSize) then
36:      candidateList.push(newCandidate)
37:    end if
38:  end if
39: end while
40: if isLastSegment then
41:   ▷ Para finalizar el flujo de datos
   es necesario truncar por tamaño máximo hasta llegar a la última
   posición de los datos y luego insertar a ésta como punto de corte.
   Omitido por brevedad.
42: end if

```

Adaptación del teorema para CDCZ

En la formulación del teorema 1 analizamos la situación con hilos de ejecución como los usados sobre segmentos en SS-CDC. Sin embargo nosotros introducimos puntos de corte alternativos de respaldo de SuperCDC, así como los del cortado de bloques normalizado de FastCDC. En este escenario, existen varias condiciones de corte diferentes, algunas más exigentes que otras, que se aplican o no dependiendo del tamaño que haya alcanzado el bloque en construcción. En particular notamos que los puntos de corte alternativos de SuperCDC pueden ser desestimados si el bloque finalmente no necesita cortarse por la condición de tamaño máximo, y de forma similar, los puntos de corte que satisfacen la condición relajada del cortado de bloques normalizado solo se aplican después de llegar al tamaño promedio del bloque.

Teniendo en cuenta estas consideraciones, enunciamos el siguiente teorema, válido para el algoritmo CDC con corte de bloques normalizado y cortes de respaldo. Omitimos la demostración porque es análoga a la del teorema 1.

Teorema 2. *Sean $m > 0$, $a > m$ y $M > a$ los tamaños mínimo, promedio y máximo, respectivamente, impuestos para los bloques, y sean c_0 una posición candidata de corte por el criterio difícil y c_1 una posición candidata de corte por el criterio fácil o difícil obtenidas consecutivamente por un algoritmo CDC con corte de bloques normalizado. Sea $d = c_1 - c_0$ la distancia entre estas posiciones. Si se cumple $m \leq d < M - m$ y adicionalmente $a \leq d$ en caso de que c_1 fuese obtenido por la condición fácil, entonces c_1 es seleccionada como posición de corte por el algoritmo CDC.*

Para CDCZ por tanto sólo consideramos los candidatos de corte c_0 que satisfacen la condición difícil del cortado de bloques normalizado, ya que estos sólo pueden ser rechazados por las condiciones de tamaño mínimo y máximo, que son las contempladas por el teorema 1. En cambio para c_1 permitimos tanto candidatos obtenidos por la condición difícil y la fácil, requiriendo una condición adicional y más estricta en la distancia entre c_1 y c_0 en ese caso.

Se puede observar el uso del teorema 2 en el algoritmo 3 en las líneas 11-16. *prevCandidate* es la posición candidata de corte c_0 del teorema 2 y efectivamente requerimos que sea obtenida por el criterio difícil del cortado de bloques normalizado denotado en el código por ‘*HARD*’. *newCandidate* es entonces el candidato de corte c_1 del teorema con *newCandidate*[1] su tipo de condición de corte satisfecha y *newCandidate*[0] la distancia d del teorema 2. Es claro que se comprueban todas las hipótesis del teorema 2. Al satisfacer el teorema 2 se procede inmediatamente al modo de proceso sincronizado.

4.3. Implementación

Implementamos CDCZip usando C++ de forma portable por lo que nuestra implementación es multiplataforma.

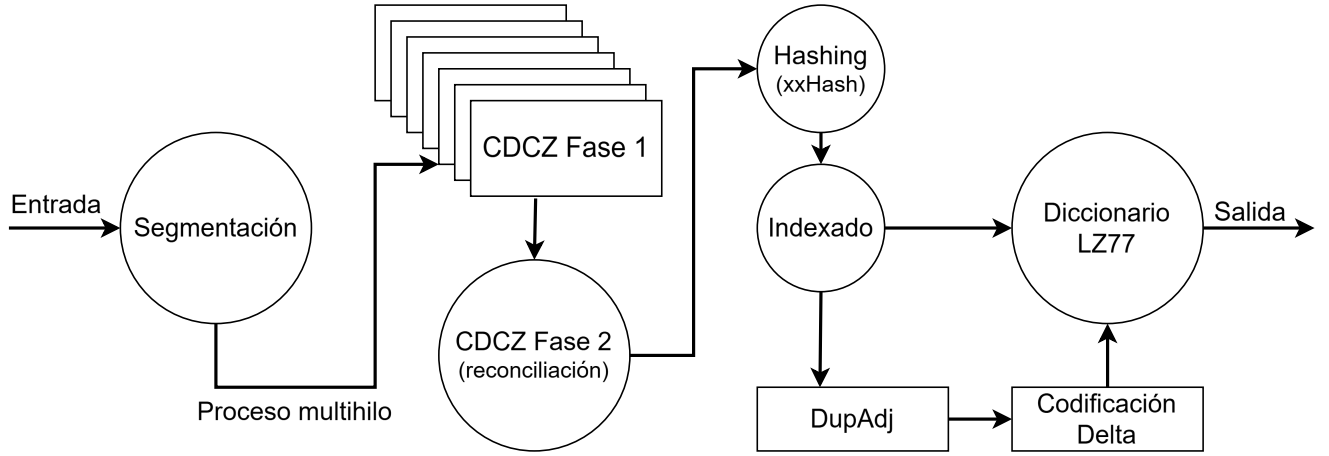


Figura 4.1: Diagrama de la implementación de CDCZip

En el diagrama 4.1 se puede ver una representación del flujo de datos en nuestra implementación. CDCZip primero lee suficientes datos como para generar segmentos para la ejecución al estilo SS-CDC en varios hilos. Estos hilos ejecutan nuestra implementación de cortado de bloques CDCZ. CDCZ incluye nuestra versión adaptada de la fase 2 de SS-CDC que es capaz de consolidar los resultados de los segmentos en los puntos de corte de bloques de la misma forma que si se hubiesen procesado en un único hilo sin usar segmentación.

Estos bloques son emitidos, sus firmas de hash calculadas e ingresadas en las estructuras de memoria mantenidas para el proceso de deduplicación por CDCZip. Para la detección misma de bloques duplicados simplemente usamos un hashmap (implementación de la librería estándar de C++) donde se busca por colisiones usando el algoritmo de hash xxHash (que es un algoritmo moderno y rápido, si bien no criptográfico, usado por Zstandard para la detección de errores). Opcionalmente se puede realizar una comprobación byte por byte. Al agregarse nuevos bloques se actualiza el diccionario de compresión.

Ante el eventual llenado del diccionario se hace la evicción de bloques anteriores con lógica FIFO. Cuando ocurre esta evicción, los datos y/o coincidencias LZ77 relevantes son emitidos al flujo de salida de forma que para el proceso de descompresión sólo debe mantener un diccionario del tamaño adecuado. De igual forma se retiran los bloques de las otras estructuras de memoria como el hashmap para la deduplicación.

4.3.1. DupAdj y compresión delta

Al encontrarse un bloque para el cual no se puede encontrar otro duplicado, pero es adyacente a otro bloque que sí es duplicado, se aplica la técnica DupAdj de DARE. La compresión delta usada es una adaptación del mismo proceso de deduplicación, repetimos el proceso de cortado de bloques CDC pero usando tamaños de bloque mínimo, promedio y máximo más pequeños sobre ambos bloques para los que se efectúa la compresión delta, obteniendo lo que podemos llamar *minibloques*. Usando un hashmap temporal, buscamos por repeticiones de estos minibloques del bloque a comprimir en comparación con los minibloques del bloque de referencia. De esta forma obtenemos coincidencias LZ77 para el bloque de forma que omitimos ciertos datos que están disponibles en el bloque de referencia seleccionado por DupAdj.

4.3.2. Descompresión

El proceso de descompresión no implica ningún trato especial ya que la codificación usada es la de un compresor LZ77 simple. El descompresor no requiere ningún conocimiento sobre el cortado de bloques CDC ni cuales bloques fueron cortados por el compresor. Los datos son leídos desde el flujo de entrada y el diccionario de compresión es poblado con los datos literales, mientras que las coincidencias LZ77 son leídas de ese diccionario. Los datos pueden ser descomprimidos a un archivo o a un flujo de salida mediante *pipeing*.

4.3.3. Disponibilidad

La implementación esta disponible en [GitHub](https://github.com/nicolas-comerci/CDCZip) (<https://github.com/nicolas-comerci/CDCZip>). El código tiene licencia Apache-2.0 que es una licencia de código abierto permisiva por lo que el código se puede usar para derivados de código abierto o comerciales. Se proporcionan binarios para Windows en arquitectura de CPUs AMD64 y para MacOS en arquitectura de CPUs ARM64.

Capítulo 5

Experimentación

El contexto en el que nos desarrollamos y evaluamos nuestro proyecto, entre otras cosas, incluye a RZip/LRZip como precursor, siendo éste un filtro de deduplicación basado en un algoritmo más antiguo, y ZStandard como algoritmo de compresión que incluye su propio detector de datos duplicados en rango largo (basado también en el algoritmo RZip). Por tanto determinamos que debemos evaluar el proyecto en las siguientes áreas:

1. El algoritmo de deduplicación implementado representa el estado del arte.
2. El filtro de deduplicación se compara favorablemente contra los precursores RZip/LRZip.
3. El filtro de deduplicación al ser usado como preprocesador para compresión permite mejores resultados en términos de tiempo de ejecución y/o tasa de compresión.
 - a) Usando un algoritmo de compresión con capacidades de deduplicación no modernas (ZStandard)
 - b) Usando un algoritmo de compresión sin capacidades de deduplicación (LZMA)

5.1. Datos de prueba

Para realizar estas evaluaciones debemos primero definir un conjunto de datos de prueba donde podemos correr experimentos para poder evaluar los tiempos de ejecución y tasas de compresión de CDCZ y CDCZip, así como de las referencias para las comparaciones de LRZip, LZMA y ZStandard.

Compilamos tres juegos de datos usando criterios recurrentes en publicaciones sobre deduplicación y compresión.

5.1.1. Juego de datos LNX

Este juego de datos esta comprendido por tarballs del código fuente de varias versiones del Kernel de Linux con un total de 42.4GB de datos. Representa un caso de uso donde la deduplicación es muy efectiva, ya que entre las diversas versiones hay una gran cantidad del código que no es modificado y por tanto se encuentra duplicado. El código fuente del Kernel de Linux es usado comúnmente (Xia, Jiang, Feng, y Tian, 2016)(Ni y Jiang, 2019)(Ni y cols., 2019)(Xia y cols., 2020)(Wan y cols., 2022) para estas evaluaciones precisamente por resultar en juegos de datos con alta redundancia de datos donde los algoritmos de deduplicación y compresión pueden probarse en condiciones altamente favorables. Nos permitirá evaluar cómo los algoritmos de compresión evaluados se comportan en este caso y qué beneficios podemos obtener con CDCZip en un caso de aplicación prácticamente ideal.

5.1.2. Juegos de datos LNX-IMG

A diferencia del juego de datos anterior, éste es conformado por archivos binarios, imágenes de disco de Linux para despliegues en la nube, en vez de código fuente. Incluye imágenes de las distribuciones CentOS, Fedora, RockyLinux, AlmaLinux, Ubuntu y Debian en varias versiones y para diferentes arquitecturas de CPU incluyendo x86-64, ARM64, PPC64LE y S390X. En total contiene 44.1GB de datos. El uso de varias versiones de imágenes de disco, Docker o similares es común (Ni y Jiang, 2019)(Ni y cols., 2019)(Xia y cols., 2020)(Wan y cols., 2022) ya que usualmente presentan un buen balance de compresibilidad y duplicación de datos, sin llegar a la idoneidad del juego de datos anterior.

También es importante notar que este tipo de juegos de datos tienen una aplicabilidad clara, ya que sistemas de almacenamiento o archivado y servidores repositorios de software comúnmente se encuentran en este tipo de situaciones donde tienen varias versiones de un mismo software y otros similares.

5.1.3. Juego de datos FslHomes

Construimos este juego de datos con 25.2GB de datos reales anonimizados del File system and Storage Lab (FSL) en Stony Brook University (FSL, 2014). Estos son datos reales de usuarios de su laboratorio, que consintieron a que sus datos fueran anonimizados y publicados con el fin de disponibilizar datos reales para el estudio de la deduplicación, donde un problema común es la dificultad de proporcionar datos reales de forma que las publicaciones puedan ser reproducibles, lo que es importante para cualquier área científica.

Este juego de datos busca representar el posible impacto de la compresión y deduplicación aplicada en casos de uso típicos.

Tabla 5.1: Cortado de bloques usando FastCDC contra CDCZ, un solo hilo.

	LNX		LNX-IMG		FslHomes	
	Rendimiento MB/s	Tasa de dedup.	Rendimiento MB/s	Tasa de dedup.	Rendimiento MB/s	Tasa de dedup.
FastCDC	2450.73	1.8873	2825.30	1.8999	2154.17	1.3036
CDCZ	2265.20	1.8898	2529.60	1.9194	2035.73	1.3037

5.2. Configuración del entorno de prueba

Las pruebas fueron realizadas en un equipo con un CPU Intel i7-12700K (20 hilos) y 128GB de RAM. Los experimentos del cortado de bloques son realizados en memoria RAM para evaluar al algoritmo y su implementación removiendo en la mayor medida posible cualquier cuello de botella, de esta forma nuestros resultados pueden ser a grandes rasgos comparados con otras publicaciones del área, que usan este mismo criterio.

Para deduplicación en sistemas a gran escala es común el uso de bloques relativamente grandes. Una configuración común (Xia, Jiang, Feng, Douglass, y cols., 2016) es 2KB de tamaño mínimo, 8KB de tamaño promedio y 64 KB de tamaño de tamaño máximo. Estos tamaños superan en varios órdenes de magnitud lo que se suelen usar para acompañar procesos de compresión. Por ejemplo ZStandard y su buscador de duplicados en largo rango tienen un tamaño por defecto de 64 bytes, soportando un máximo de 512 bytes.

A partir de nuestros experimentos, decidimos usar bloques de tamaño mínimo de 64 bytes, promedio de 256 bytes y máximo de 2KB, ya que esta configuración mostró el mejor balance entre tasa de deduplicación, tiempo de ejecución y consumo de recursos.

5.3. CDCZ representa al estado del arte

5.3.1. CDCZ es comparable con FastCDC al correr en un único hilo

Para evaluar este aspecto, primero comparamos nuestro algoritmo de cortado de bloques contra FastCDC corriendo sobre un solo hilo y confirmamos que CDCZ ofrece mayor deduplicación con rendimiento similar.

Como podemos ver en la tabla 5.1 nuestro algoritmo CDCZ tiene rendimiento similar si bien algo menor a FastCDC y por otro lado presenta mejores tasas de deduplicación.

Esto tiene sentido y es lo esperado, ya que la condición de corte de respaldo y recuperación del contexto luego de los saltos introducidos por SuperCDC e integrados en CDCZ significan un sacrificio en el tiempo de ejecución para lograr

Tabla 5.2: Cortado de bloques usando SS-CDC basado en GEAR contra CDCZ, 20 hilos.

	LNX		LNX-IMG		FslHomes	
	Rendimiento MB/s	Factor mejora	Rendimiento MB/s	Factor mejora	Rendimiento MB/s	Factor mejora
GEAR	1739.30		1746.60		1741.13	
SS-CDC(GEAR)	15454.60	8.8973	14549.20	8.3280	16199.80	9.3050
CDCZ	2265.20		2529.60		2035.73	
CDCZ MT	16446.17	7.2609	16270.47	6.4308	16702.40	8.2033

incrementos en las tasas de compresión, que es exactamente lo que observamos.

5.3.2. CDCZ permite paralelismo con características comparables a SS-CDC

Para comparar el rendimiento multihilo de CDCZ contra SS-CDC implementamos SS-CDC sobre cortado de bloques CDC tradicional usando hashes Gear (esto es, sin ninguna de las optimizaciones de FastCDC ni SuperCDC). Comparamos los beneficios experimentados sobre el cortado de bloques simple usando Gear otorgados por SS-CDC con los observables en la ejecución multihilo de CDCZ.

En la tabla 5.2 podemos ver cómo el proceso multihilo de CDCZ le otorga un incremento en rendimiento comparable al otorgado por SS-CDC al cortado de bloques tradicional usando GEAR. El incremento de rendimiento de CDCZ es en proporción algo menor al experimentado por GearCDC. Una explicación precisa de las causas requeriría mayor investigación. De cualquier forma, nuestros datos confirmarían que CDCZ puede lograr grandes incrementos en rendimiento mediante paralelización, sin perder eficiencia de deduplicación y logrando rendimiento aún superior a la de SS-CDC. Las tasas de deduplicación para SS-CDC fueron de 1.8895, 1.9172 y 1.2817 para los juegos de datos LNX, LNX-IMG y FslHomes respectivamente, mientras que para CDCZ multihilo fueron de **1.8898**, **1.9194** y **1.3037**. Por tanto este superior rendimiento frente a SS-CDC no sólo se obtiene sin sacrificios en las tasas de deduplicación sino que es acompañado de tasas ligeramente superiores.

5.3.3. CDCZip se compara favorablemente contra LRZip

Si bien CDCZip se basa en técnicas de estado del arte en deduplicación mientras que LRZip se basa en el algoritmo tradicional RZip, ambos cumplen la misma función de filtrar repeticiones de datos en rangos largos con el propósito de ser usados como preprocesadores para otros algoritmos de compresión. Evaluamos entonces el rendimiento de CDCZip en comparación con LRZip en términos de tiempo de ejecución y reducción de datos observada sobre nuestros

Tabla 5.3: CDCZip vs LRZip.

	LNX		LNX-IMG		FslHomes	
	Rendimiento MB/s	Tasa de dedup.	Rendimiento MB/s	Tasa de dedup.	Rendimiento MB/s	Tasa de dedup.
500MB Ventana/Diccionario						
LRZip	55.81	1.4056	57.38	1.3639	70.50	2.0889
CDCZip	134.84	1.4738	110.41	1.50	140.24	1.7264
1000MB Ventana/Diccionario						
LRZip	76.71	1.6933	64.98	1.3565	84.05	1.9680
CDCZip	139.61	14.0601	126.14	1.8773	133.01	1.8650
2000MB Ventana/Diccionario						
LRZip	106.16	2.9393	80.35	1.4663	98.12	1.8825
CDCZip	159.62	18.2753	131.27	2.2945	132.33	1.9791

juegos de datos de prueba.

En la tabla 5.3 se puede observar cómo CDCZip presenta en todos los casos tiempos de ejecución menores que LRZip, mientras logra en general mayores tasas de deduplicación. La única excepción se da para el juego de datos FslHomes, donde LRZip logra mayores reducciones para tamaños de diccionario más pequeños. En estos dos casos no favorables para la tasa de deduplicación su empeoramiento fue de 11 % en promedio, mientras que los casos más favorables (LNX con 1000 y 2000 MB de diccionario) presentaron incrementos de la tasa del 625 %, con el resto de los casos siendo más típicos con incrementos del 26 % en promedio. En suma, CDCZip muestra substancialmente mejor desempeño en la mayoría de los casos, desempeño de un orden de magnitud superior en algunos pocos, mientras en una minoría de los casos tiene desempeño modestamente inferior.

5.3.4. CDCZip como preprocesador para compresión

Nos interesa poner a prueba el rendimiento de CDCZip como preprocesador previo a un proceso de compresión, buscando confirmar que las promesas de menores tiempos de ejecución y mayores tasas de compresión realmente se materializan.

Para esto corremos primero el compresor con un tamaño de diccionario o ventana de compresión en particular, y luego realizamos dos combinaciones junto a CDCZip como preprocesador donde mantemos el tamaño combinado de diccionario o ventana de compresión pero lo distribuimos entre ambos procesos. En la primera combinación alojamos la mayoría de la memoria de el diccionario a CDCZip, mientras en la segunda repartimos en partes iguales la memoria alojada para el diccionario entre CDCZip y el compresor. Comparando los resultados de no aplicar CDCZip o de aplicarlo en diferente medida podemos apreciar el nivel de contribución de CDCZip en el proceso.

Tabla 5.4: CDCZip junto a ZStandard.

	LNX		LNX-IMG		FslHomes	
	Rendimiento MB/s	Tasa de comp.	Rendimiento MB/s	Tasa de comp.	Rendimiento MB/s	Tasa de comp.
1GB Ventana/Diccionario total						
ZStd	32.58	21.7414	41.91	3.6147	32.42	4.1661
CDCZip 960 +ZStd 64	113.96	72.9560	41.51	4.0184	34.22	4.1641
CDCZip 512 +ZStd 512	32.87	9.7751	36.61	3.5069	36.76	4.1688
2GB Ventana/Diccionario total						
ZStd	34.57	44.8066	48.62	4.5559	37.02	4.2282
CDCZip 1984 +ZStd 64	130.78	116.8774	49.91	4.9359	34.18	4.2787
CDCZip 1024 +ZStd 1024	119.28	120.9067	48.30	4.4773	38.40	4.2359

CDCZip como preprocesador junto a ZStandard

Con ZStandard realizamos dos tandas de pruebas, una con tamaño total de diccionario y ventana de compresión de 1GB, y otra con 2GB. ZStandard fue configurado para usar el nivel de compresión 17 de entre sus predefinidos, que es el más potente dentro de los recomendados para uso general por la documentación. Los resultados con ZStandard son de particular interés ya que ZStandard, para la configuración y tamaños de ventana de compresión dados, activa por defecto sus capacidades de reducción de datos en largas distancias basadas en RZip.

En la tabla 5.4 podemos ver cómo en la mayoría de los casos al menos una de las dos combinaciones con CDCZip nos permite obtener mayor compresión con menores tiempos de ejecución. La excepción es para el juego de datos LNX-IMG con 1GB de diccionario, donde incluso el resultado más favorable con CDCZip si bien implica una mejora significativa en la tasa de compresión, también implica un pequeño incremento en el tiempo de ejecución.

CDCZip como preprocesador junto a LZMA

Para LZMA nuevamente realizamos dos tandas de experimentos como con ZStandard, pero esta vez con diccionarios totales de 1GB y 512MB. Esto es porque usamos el compresor XZ (Tukaani, 2009) que no soporta diccionarios más grandes. A diferencia de ZStandard, LZMA no tiene ninguna capacidad especial integrada para detección de datos duplicados en distancias largas, por tanto es esperable que CDCZip produzca mejores resultados.

Los resultados mostrados en la tabla 5.5 de estos experimentos nos muestran que CDCZip al ser usado en combinación con compresión LZMA nos permite

Tabla 5.5: CDCZip junto a LZMA.

	LNX		LNX-IMG		FslHomes	
	Rendimiento MB/s	Tasa de comp.	Rendimiento MB/s	Tasa de comp.	Rendimiento MB/s	Tasa de comp.
512MB Ventana/Diccionario total						
XZ	21.28	8.2812	25.79	3.4454	19.88	3.5867
CDCZip 448						
+XZ 64	31.10	8.8867	45.29	3.6785	32.38	4.0794
CDCZip 256						
+XZ 256	28.16	8.0163	41.35	3.4386	30.61	3.9553
1GB Ventana/Diccionario total						
XZ	13.28	10.9844	18.39	3.6750	19.77	3.5867
CDCZip 960						
+XZ 64	111.04	81.9123	53.70	4.2899	37.95	4.1785
CDCZip 512						
+XZ 512	33.45	10.5114	44.10	3.7172	33.95	4.1290

obtener tasas de compresión superiores en menor tiempo de ejecución. Los resultados son especialmente prometedores cuando alojamos la mayoría del diccionario a CDCZip.

Capítulo 6

Conclusiones y Trabajo Futuro

Pudimos diseñar e implementar CDCZip, un filtro de deduplicación basado en técnicas de deduplicación de estado del arte para usarse como preprocesador para procesos de compresión.

Para esto fue necesario realizar un relevamiento de bibliografía con el fin de identificar técnicas de estado del arte y resolver problemas de diseño en tanto a su integración conjunta. En servicio de esto planteamos el teorema 1 de condición suficiente para la elección de un candidato a punto de corte en CDC e implementamos CDCZ, un algoritmo de CDC que consolida múltiples algoritmos de estado del arte con el fin de hacerse de todos sus beneficios.

Los resultados experimentales muestran que el uso de CDCZip puede incrementar las tasas de compresión y reducir los tiempos de ejecución sobre los juegos de datos probados.

Sin embargo, el uso óptimo de CDCZip requiere en su estado actual conocimiento a priori de las características de los datos a procesar. Usar tamaños de diccionario muy pequeños pueden hacer que las contribuciones de CDCZip al proceso de compresión sean nulas o incluso negativas, mientras que usar tamaños de diccionario muy grandes pueden ser contraproducente si un tamaño más pequeño podría producir resultados similares y sería mejor alojar más diccionario para el compresor que se usa luego de CDCZip.

Para resolver esta problemática en el futuro se puede trabajar en una implementación que esté directamente integrada junto a un algoritmo de compresión, de forma que esta división del diccionario entre dos procesos separados no necesita ocurrir. Un filtro adaptativo que reaccionara según las tasas de duplicación observados en los datos de entrada en contra de las tasas de compresión logrados

mediante otras técnicas usadas por el compresor (como codificación entrópica o aritmética) podría informar al compresor sobre cómo usar el diccionario de forma óptima.

Otras optimizaciones y refinaciones en términos de calidad de código, tiempos de ejecución y uso de memoria u otros recursos que no se pudieron hacer por limitaciones de tiempo son probablemente posibles.

En cualquier caso creemos que los resultados obtenidos son suficientemente positivos como para ameritar más investigación y trabajo en el área. Creemos que las presentes limitaciones pueden ser solucionadas de tal forma que el uso de técnicas de deduplicación podrían en el futuro tornarse en un aspecto fundamental para la compresión en escenarios de uso propicios.

Referencias

- Adler, M., y Gailly, J. (1992). *Gzip*. (<https://www.gnu.org/software/gzip/> [Accessed: 04/12/24])
- A. Muthitacharoen, B. C., y Mazieres, D. (2001). A low-bandwidth network file system. *Proc. ACM Symp. Oper. Syst. Principles, 2001*, 1–14.
- Black, J. (2006). {Compare-by-Hash}: A Reasoned Analysis.. Descargado 2024-11-21, de <https://www.usenix.org/conference/2006-usenix-annual-technical-conference/compare-hash-reasoned-analysis>
- Collet, Y. (2021). Zstandard compression and the 'application/zstd' media type. *Internet Engineering Task Force RFC*.
- Deusch, P. (1996). Deflate compressed data format specification version 1.3. *Internet Engineering Task Force RFC*.
- FSL. (2014). *Traces and snapshots public archive*. (<https://tracer.filesystems.org> [Accessed: 20/11/24])
- Google. (2017). *Announcing the first sha1 collision*. (<https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html> [Accessed: 20/11/24])
- Henson, V. (2003). An Analysis of Compare-by-hash. En *May 18, 2003* (pp. 13–18). Descargado 2024-11-21, de <https://www.usenix.org/conference/hotos-ix/analysis-compare-hash>
- Huffman, D. A. (1952, septiembre). A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9), 1098–1101. Descargado 2024-06-13, de <https://ieeexplore-ieee-org.proxy.timbo.org.uy/document/4051119> (Conference Name: Proceedings of the IRE) doi: 10.1109/JRPROC.1952.273898
- Jin, X., Liu, H., Ye, C., Liao, X., Jin, H., y Zhang, Y. (2023, septiembre). Accelerating Content-Defined Chunking for Data Deduplication Based on Speculative Jump. *IEEE Transactions on Parallel and Distributed Systems*, 34(9), 2568–2579. Descargado 2024-11-21, de <https://ieeexplore.ieee.org/document/10168293> (Conference Name: IEEE Transactions on Parallel and Distributed Systems) doi: 10.1109/TPDS.2023.3290770
- Kolivas, C. (2008). *Lrzip*. (<https://github.com/ckolivas/lrzip> [Accessed: 20/11/24])
- MacDonald, J. (2000). *File system support for delta compression* (Tesis Doctoral no publicada). Dept. Electr. Eng. Comput. Sci., Univ. California at Berkeley, Berkeley, CA, USA.

- Meyer, D. T., y Bolosky, W. J. (2012). A Study of Practical Deduplication.. Descargado 2024-05-20, de <https://www.usenix.org/conference/fast11/study-practical-deduplication> doi: 10.1145/2078861.2078864
- Ni, F., y Jiang, S. (2019, noviembre). RapidCDC: Leveraging Duplicate Locality to Accelerate Chunking in CDC-based Deduplication Systems. En *Proceedings of the ACM Symposium on Cloud Computing* (pp. 220–232). New York, NY, USA: Association for Computing Machinery. Descargado 2024-11-29, de <https://dl.acm.org/doi/10.1145/3357223.3362731> doi: 10.1145/3357223.3362731
- Ni, F., Lin, X., y Jiang, S. (2019, mayo). SS-CDC: a two-stage parallel content-defined chunking for deduplicating backup storage. En *Proceedings of the 12th ACM International Conference on Systems and Storage* (pp. 86–96). New York, NY, USA: Association for Computing Machinery. Descargado 2024-06-24, de <https://doi.org/10.1145/3319647.3325834> doi: 10.1145/3319647.3325834
- Pavlov, I. (1999). *Lzma*. (<https://7-zip.org/sdk.html> [Accessed: 04/12/24])
- Quinlan, S., y Dorward, S. (2002). Venti: A new approach to archival storage. *Proc. USENIX Conf. File Storage Technol.*, 2002, 1–13.
- Rabin, M. O. (1981). *Fingerprinting by random polynomials*. Cntr. Res. Comput. Tech., Aiken Comput. Lab.
- Rissanen, J. J. (1976, mayo). Generalized Kraft Inequality and Arithmetic Coding. *IBM Journal of Research and Development*, 20(3), 198–203. Descargado 2024-11-28, de <https://ieeexplore.ieee.org/document/5391119> (Conference Name: IBM Journal of Research and Development) doi: 10.1147/rd.203.0198
- Seward, J. (1996). *Bzip2*. (<https://sourceware.org/bzip2/> [Accessed: 04/12/24])
- Shannon, C. E. (1948, julio). A mathematical theory of communication. *The Bell System Technical Journal*, 27(3), 379–423. Descargado 2024-11-28, de <https://ieeexplore.ieee.org/document/6773024> (Conference Name: The Bell System Technical Journal) doi: 10.1002/j.1538-7305.1948.tb01338.x
- Tridgell, A. (2000, mayo). Efficient Algorithms for Sorting and Synchronization.
- Tridgell, A., y Mackerras, P. (1996). *The rsync algorithm* (Tesis Doctoral no publicada). Faculty of Engineering and Information Technology, The Australian National University.
- Tukaani. (2009). *Xz utils*. (<https://tukaani.org/xz/> [Accessed: 01/12/24])
- Wallace, G., Douglass, F., Qian, H., Shilane, P., Smaldone, S., Chamness, M., y Hsu, W. (2012). Characteristics of Backup Workloads in Production Systems.. Descargado 2024-05-20, de <https://www.usenix.org/conference/fast12/characteristics-backup-workloads-production-systems>
- Wan, B., Pu, L., Zou, X., Li, S., Wang, P., y Xia, W. (2022, octubre). SuperCDC: A Hybrid Design of High-Performance Content-Defined Chunking for Fast Deduplication. En *2022 IEEE 40th International Conference on Computer Design (ICCD)* (pp. 170–178). Descar-

- gado 2024-06-25, de <https://ieeexplore-ieee-org.proxy.timbo.org.uy/document/9978509> (ISSN: 2576-6996) doi: 10.1109/ICCD56317.2022.00034
- Xia, W., Jiang, H., Feng, D., Douglass, F., Shilane, P., Hua, Y., ... Zhou, Y. (2016, septiembre). A Comprehensive Study of the Past, Present, and Future of Data Deduplication. *Proceedings of the IEEE*, 104(9), 1681–1710. Descargado 2024-05-18, de <https://ieeexplore.ieee.org/document/7529062/> (Conference Name: Proceedings of the IEEE) doi: 10.1109/JPROC.2016.2571298
- Xia, W., Jiang, H., Feng, D., y Tian, L. (2016, junio). DARE: A Deduplication-Aware Resemblance Detection and Elimination Scheme for Data Reduction with Low Overheads. *IEEE Transactions on Computers*, 65(6), 1692–1705. Descargado 2024-06-25, de <https://ieeexplore-ieee-org.proxy.timbo.org.uy/document/7155488> (Conference Name: IEEE Transactions on Computers) doi: 10.1109/TC.2015.2456015
- Xia, W., Jiang, H., Feng, D., Tian, L., Fu, M., y Zhou, Y. (2014, septiembre). Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, 79, 258–272. Descargado 2024-06-13, de <https://www.sciencedirect.com/science/article/pii/S0166531614000790> doi: 10.1016/j.peva.2014.07.016
- Xia, W., Zou, X., Jiang, H., Zhou, Y., Liu, C., Feng, D., ... Zhang, Y. (2020, septiembre). The Design of Fast Content-Defined Chunking for Data Deduplication Based Storage Systems. *IEEE Transactions on Parallel and Distributed Systems*, 31(9), 2017–2031. Descargado 2024-06-24, de <https://ieeexplore-ieee-org.proxy.timbo.org.uy/document/9055082/> (Conference Name: IEEE Transactions on Parallel and Distributed Systems) doi: 10.1109/TPDS.2020.2984632
- Ziv, J., y Lempel, A. (1977, mayo). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3), 337–343. Descargado 2024-06-13, de <https://ieeexplore-ieee-org.proxy.timbo.org.uy/document/1055714> (Conference Name: IEEE Transactions on Information Theory) doi: 10.1109/TIT.1977.1055714