Manuale Utente per la Memoria Persistente in gawk

9 febbraio 2025
gawk versione 5.3.1
pm-gawk versione 2022.10Oct.30.1667172241 (Avon 8)

Terence Kelly
tpkelly@eecs.umich.edu
tpkelly@cs.princeton.edu
tpkelly@acm.org
http://web.eecs.umich.edu/~tpkelly/pma/
https://dl.acm.org/profile/81100523747
https://queue.acm.org/DrillBits

Traduzione italiana di Antonio Giovanni Colombo

1 Introduzione

GNU AWK (`gawk`) 5.2, rilasciato nel mese di settembre 2022, ha introdotto la nuova funzionalità della *memoria persistente* che facilita la preparazione di script AWK e in certi casi ne migliora le prestazioni. La nuova funzionalità, chiamata "pm-`gawk`", può "ricordare" le variabili e le funzioni definite in uno script, che possono quindi essere utilizzate in successive esecuzioni di script `gawk` e consente di passare variabili fra script non correlati fra loro senza serializzare/analizzare file di testo—in maniera praticamente trasparente. pm-`gawk` *non* richiede presenza di memoria non-volatile sul computer in cui viene eseguito, e neppure altre infrastrutture insolite; funziona sui normali computer e sui normali sistemi operativi che sono in uso generale da decenni.

La documentazione principale di `gawk`[1] spiega il funzionamento di base della nuova funzionalità di persistenza. Questo manuale supplementare fornisce ulteriori dettagli, esempi di programmi, e uno sguardo "sotto il coperchio" a pm-`gawk`. Chi abbia familiarità con `gawk` e con gli ambienti software di tipo Unix, può proseguire direttamente leggendo:

Si può trovare la versione più recente di questo manuale, e anche la "versione del regista", nel sito web dedicato all'allocatore di memoria persistente usato in pm-`gawk`:

`http://web.eecs.umich.edu/~tpkelly/pma/`

La traduzione italiana più recente di questo manuale si può trovare in:

`https://sites.google.com/view/gawkdoc-it/home-page`

Due pubblicazioni descrivono [in inglese] l'allocatore di memoria persistente, e le prime esperienze con un prototipo di pm-`gawk` preparato a partire da una diramazione della distribuzione sorgente ufficiale di `gawk`:

- `https://queue.acm.org/detail.cfm?id=3534855`
- `http://nvmw.ucsd.edu/nvmw2022-program/nvmw2022-data/nvmw2022-paper35-final_version_your_extended_abstract.pdf`

[1] Vedere `https://www.gnu.org/software/gawk/manual/`, `man gawk` e `info gawk`.

Sentitevi liberi di inviarmi domande, suggerimenti, e resoconti di esperienze a:

tpkelly@eecs.umich.edu (preferito)
tpkelly@cs.princeton.edu
tpkelly@acm.org

2 Per iniziare

Ecco pm-gawk in azione, partendo dal prompt ('$') della shell bash:

```
$ truncate -s 4096000 heap.pma
$ export GAWK_PERSIST_FILE=heap.pma
$ gawk 'BEGIN{un_valore = 47}'
$ gawk 'BEGIN{un_valore += 7; print un_valore}'
54                      # '7' => non pm-gawk, errore => mal compilato
```

Per prima cosa, il comando truncate crea un *file sparso* vuoto (tutto a zeri binari) in cui pm-gawk immagazzinerà le variabili dello script; la sua dimensione dev'essere un multiplo della dimensione di una pagina del sistema di paginazione (4 KB). Poi, il comando export imposta una variabile di ambiente che consente a pm-gawk di trovare il file sparso; se gawk *non* trova questa variabile d'ambiente, la memoria persistente non è attivata. Il terzo comando esegue uno script AWK di una riga che inizializza la variabile un_valore, la quale sarà poi salvata nel file sparso, e in questo modo "sopravviverà" al programma interpretato che l'ha inizializzata. Infine, il quarto comando invoca pm-gawk utilizzando uno script AWK di una riga, *differente* dal precedente, il quale incrementa e visualizza script un_valore. La riga di output mostra che in effetti pm-gawk ha "ricordato" un_valore utilizzandolo in script differenti e non correlati fra loro. Per non usare la persistenza finché non lo si desideri ancora, si può impedire a gawk di trovare il nome del file sparso, dando il comando unset GAWK_PERSIST_FILE. Per "dimenticare" completamente le variabili dello script, basta cancellare il file sparso che le contiene.

Si veda la Appendice A [Installazione], pagina 24, per due problemi comuni e per come risolverli. Se eseguite l'esempio qui sopra e pm-gawk non funziona alla *seconda* chiamata, probabilmente il vostro pm-gawk era stato compilato in maniera non corretta. Se viene stampato '7' invece che '54', il programma eseguibile gawk trovato nelle librerie del vostro $PATH sono state compilate senza la funzionalità della memoria persistente.

Attivare e disattivare la memoria persistente usando i comandi export e unset per gestire la variabile d'ambiente richiede attenzione: dimenticando di usare unset quando non si vuole più usare la memoria persistente si può andare incontro a sorprese. Fortunatamente bash consente di passare a un programma le variabili d'ambiente più esplicitamente, comando per comando:

```
$ rm heap.pma                    # ripartire da zero
$ unset GAWK_PERSIST_FILE        # eliminare la variabile d'ambiente
$ truncate -s 4096000 heap.pma   # creare il nuovo file sparso

$ GAWK_PERSIST_FILE=heap.pma gawk 'BEGIN{un_valore = 47}'
$ gawk 'BEGIN{un_valore += 7; print un_valore}'
7
$ GAWK_PERSIST_FILE=heap.pma gawk 'BEGIN{un_valore += 7; print un_valore}'
54
```

La prima invocazione di gawk premette la variabile d'ambiente speciale nella riga di comando, prima di chiamare il comando gawk, e quindi attiva pm-gawk. La seconda invocazione di gawk, tuttavia, *non* vede la variabile d'ambiente, e quindi non utilizza la variabile dello script contenuta nel file sparso. La terza invocazione di gawk vede la variabile d'ambiente speciale, e quindi usa la variabile dello script contenuta nel file sparso.

Mentre la situazione è già migliorata rispetto all'uso di variabili d'ambiente "generali", il passaggio di variabili d'ambiente è un metodo prolisso e scomodo. Definire un *alias* di shell permette di immettere meno caratteri e di semplificare la visualizzazione:

```
$ alias pm='GAWK_PERSIST_FILE=heap.pma'
$ pm gawk 'BEGIN{print ++un_valore}'
55
$ pm gawk 'BEGIN{print ++un_valore}'
56
```

3 Esempi

Il primo esempio usa pm-`gawk` per analizzare le parole contenute nel testo dei libri di Mark Twain *Tom Sawyer* e *Huckleberry Finn* [nell'originale inglese] disponibili in https://gutenberg.org/files/74/74-0.txt e https://gutenberg.org/files/76/76-0.txt. Per prima cosa, i caratteri non-alfabetici sono convertiti nel carattere "a capo" [*newline*] (in modo che ogni riga contenga al massimo una parola), e il testo è poi fatto diventare tutto in caratteri minuscoli.

```
$ tr -c a-zA-Z '\n' < 74-0.txt | tr A-Z a-z > sawyer.txt
$ tr -c a-zA-Z '\n' < 76-0.txt | tr A-Z a-z > finn.txt
```

È facile contare la frequenza delle parole, usando i vettori associativi di AWK. pm-`gawk` rende persistenti tali vettori, in modi da evitare di rileggere nuovamente tutta la raccolta di testi dati ogni volta che ci poniamo una nuova domanda ("leggi una volta, analizza felicemente da allora in poi"):

```
$ truncate -s 100M twain.pma
$ export GAWK_PERSIST_FILE=twain.pma
$ gawk '{ts[$1]++}' sawyer.txt                    # crea lista parole
$ gawk 'BEGIN{print ts["work"], ts["play"]}'      # domanda frequenza
92 11
$ gawk 'BEGIN{print ts["necktie"], ts["knife"]}'  # domanda frequenza
2 27
```

Il comando `truncate` genera un file sparso sufficiente a contenere tutti i dati che dovrebbe poi contenere, più un abbondante spazio ulteriore. (Come vedremo più avanti, nel Sezione 4.3 [File sparsi], pagina 14, non c'è spreco di risorse). Il comando `export` assicura che le successive invocazioni di `gawk` attiveranno pm-`gawk`. Il primo comando pm-`gawk` immagazzina la frequenza con cui ricorrono le parole nel libro *Tom Sawyer* nel vettore associativo `ts[]`. Poiché tale vettore è persistente, i successivi comandi pm-`gawk` possono utilizzarlo senza dover rileggere il file di input.

Espandere la nostra analisi per includere un secondo libro è facile. Creiamo un nuovo vettore associativo `hf[]` con le frequenza delle parole contenute in *Huckleberry Finn*:

```
$ gawk '{hf[$1]++}' finn.txt
```

A questo punto si può liberamente accedere ai dati relativi a entrambi i libri, in maniera semplice ed efficiente, senza dover ogni volta analizzare nuovamente la raccolta di testi in input:

```
$ gawk 'BEGIN{print ts["river"], hf["river"]}'
26 142
```

pm-`gawk` sembra avere un ciclo leggi-valorizza-stampa, che invita a effettuare delle "conversazioni" interattive con i dati disponibili. Se ci interessa controllare quali parole in *Finn* sono assenti da *Sawyer*, le risposte (fra cui "flapdoodle," "yellocution," e "sockdolager") sono facili da trovare:

```
$ gawk 'BEGIN{for(w in hf) if (!(w in ts)) print w}'
```

Le voci riguardo alla morte di Mark Twain possono risultare essere esagerate. Se egli pubblicherà nuovi libri in futuro, sarà facile incorporarli nella nostra analisi incrementalmente, uno alla volta. I benefici in termini di efficienza di un tale modo di procedere incrementale per alcuni compiti di AWK, come l'analisi di file di log, sono trattati [in inglese] in `https://queue.acm.org/detail.cfm?id=3534855` e nel saggio sullo stesso tema, citato là e più sotto, nella sezione Capitolo 4 [Prestazioni], pagina 10.

Esercizio: Lo script AWK "Markov" riportato a pagina 79 del libro di Kernighan & Pike *The Practice of Programming* [disponibile online in inglese] genera un testo a caso, ma che ricorda quello dei libri di un dato autore, usando una semplice tecnica di modellazione statistica. Questo script consiste di una fase di "apprendimento" o "addestramento" seguita da una fase di emissione di output. Si usi pm-`gawk` per separare le due fasi e per consentire al modello statistico di aggiungere incrementalmente nuovi libri a quelli già presenti.

Il nostro secondo esempio prende in esame un altro tema, che è un punto di forza di AWK, l'analisi di dati. Per semplicità creeremo due piccoli file contenenti dati numerici.

```
$ printf '1\n2\n3\n4\n5\n' > A.dat
$ printf '5\n6\n7\n8\n9\n' > B.dat
```

Uno script AWK *non*-persistente può calcolare le statistiche sommarie di base:

```
$ cat sommario_convenzionale.awk
    1 == NR  { min = max = $1 }
    min > $1 { min  = $1 }
    max < $1 { max  = $1 }
             { sum += $1 }
    END { print "min: " min " max: " max " media: " sum/NR }

$ gawk -f sommario_convenzionale.awk A.dat B.dat
min: 1 max: 9 media: 5
```

Volendo usare pm-`gawk` per fare lo stesso, per prima cosa occorre creare un file sparso che contenga le variabili del nostro script AWK e dire a pm-`gawk` dove sono memorizzate, utilizzando la solita variabile d'ambiente:

```
$ truncate -s 10M statistiche.pma
$ export GAWK_PERSIST_FILE=statistiche.pma
```

pm-`gawk` richiede di modificare lo script AWK visto sopra per assicurarsi che le variabili `min` e `max` siano inizializzate una volta sola, al primo utilizzo del file sparso e *non* ogni volta che eseguiamo lo script. Inoltre, mentre variabili definite dallo script come `min` mantengono il loro valore a ogni successiva invocazione di pm-`gawk`, le variabili proprie di AWK, come `NR` sono inizializzate di nuovo a ogni invocazione di pm-`gawk` e quindi non possono essere usate nello stesso modo. Ecco una versione dello script modificata per pm-`gawk`:

```
$ cat sommario_persistente.awk
    ! init   { min = max = $1; init = 1 }
    min > $1 { min = $1 }
    max < $1 { max = $1 }
             { sum += $1; ++n }
    END { print "min: " min " max: " max " media: " sum/n }
```

Si noti la regola differente nella prima riga, e l'introduzione della variabile `n` al posto di `NR`. Quando viene usata con pm-`gawk`, questa nuova logica di inizializzazione supporta lo stesso tipo di trattamento incrementale già visto nel precedente scenario di analisi del testo. Per esempio, possiamo inserire i nostri file in input uno alla volta:

```
$ gawk -f sommario_persistente.awk A.dat
min: 1 max: 5 media: 3

$ gawk -f sommario_persistente.awk B.dat
min: 1 max: 9 media: 5
```

Come ci aspettavamo, dopo che la seconda invocazione di pm-`gawk` ha letto il secondo file, l'output è lo stesso che si ottiene dallo script non-persistente, che legge entrambi i file in input.

Esercizio: Modificare gli script AWK usati sopra per calcolare anche la mediana e la moda (o le mode), sia usando `gawk` convenzionale che pm-`gawk`. (La mediana è il valore di

mezzo di una lista ordinata; se la lista contiene un numero pari di elementi, va calcolata la media dei due numeri centrali. La moda è il valore (o i valori) che sono presenti con maggiore frequenza in una lista).

I nostri ultimi esempi mostrano come pm-gawk consente di mantenere in un file sparso sia i dati che le *funzioni* definite dall'utente, le quali quindi, possono essere, in seguito, liberamente utilizzate in script AWK indipendenti fra loro.

La sequenza di comandi di shell mostrata qui sotto invoca pm-gawk per creare prima e per utilizzare poi una funzione definita dall'utente. Le successive invocazioni utilizzano parecchi script AWK, tra loro differenti, che comunicano attraverso il file sparso. Ogni invocazione può aggiungere funzioni definite dall'utente e aggiungere o togliere dati al file sparso, che verrà utilizzato dalle successive invocazioni di pm-gawk.

```
$ truncate -s 10M funzioni.pma
$ export GAWK_PERSIST_FILE=funzioni.pma
$ gawk 'function conta(A,t) {for(i in A)t++; return t+0}'
$ gawk 'BEGIN { a["x"] = 4; a["y"] = 5; a["z"] = 6 }'
$ gawk 'BEGIN { print conta(a) }'
3
$ gawk 'BEGIN { delete a["x"] }'
$ gawk 'BEGIN { print conta(a) }'
2
$ gawk 'BEGIN { delete a }'
$ gawk 'BEGIN { print conta(a) }'
0
$ gawk 'BEGIN { for (i=0; i<47; i++) a[i]=i }'
$ gawk 'BEGIN { print conta(a) }'
47
```

Il primo comando pm-gawk crea la funzione definita dall'utente `conta()`, che restituisce il numero di elementi in un dato vettore associativo; si noti che la variabile `t` è locale a `conta()`, e non globale [ossia vale "0" ad ogni invocazione di `conta()`]. Il successivo comando pm-gawk riempie un vettore associativo persistente con tre elementi; non sorprende quindi che la chiamata a `conta()` nel seguente comando pm-gawk trovi questi tre elementi. I due comandi pm-gawk seguenti rispettivamente cancellano un elemento del vettore e stampano il contatore, ora ridotto a due. I due comandi seguenti cancellano i restanti elementi del vettore, e quindi viene stampato un contatore a zero. Infine, gli ultimi due comandi pm-gawk riempiono il vettore con 47 elementi, e visualizzano il contatore aggiornato.

Il seguente script di shell invoca più volte pm-gawk per creare un insieme di funzioni definite dall'utente che effettuano delle operazioni di base su polinomi di secondo grado: calcolo del valore in un dato punto, calcolo del discriminante, e utilizzo della formula quadratica per trovare le radici. Dopo di questo, l'espressione $x^2 + x - 12$ viene scomposta come $(x - 3)(x + 4)$.

```
#!/bin/sh
rm -f                    polinomi.pma
truncate -s 10M          polinomi.pma
export GAWK_PERSIST_FILE=polinomi.pma
gawk 'function q(x) { return a*x^2 + b*x + c }'
gawk 'function p(x) { return "q(" x ") = " q(x) }'
gawk 'BEGIN { print p(2) }'           # valuta e stampa
gawk 'BEGIN{ a = 1; b = 1; c = -12 }' # nuovi coefficienti
gawk 'BEGIN { print p(2) }'           # valuta/stampa ancora
gawk 'function d(s) { return s * sqrt(b^2 - 4*a*c)}'
gawk 'BEGIN{ print "discriminante (deve essere >=0): " d(1)}'
gawk 'function r(s) { return (-b + d(s))/(2*a)}'
gawk 'BEGIN{ print "root: " r( 1) "   " p(r( 1)) }'
gawk 'BEGIN{ print "root: " r(-1) "   " p(r(-1)) }'
gawk 'function abs(n) { return n >= 0 ? n : -n }'
```

```
gawk 'function sgn(x) { return x >= 0 ? "- " : "+ " } '
gawk 'function f(s) { return "(x " sgn(r(s)) abs(r(s))}'
gawk 'BEGIN{ print "factor: " f( 1) ")" }'
gawk 'BEGIN{ print "factor: " f(-1) ")" }'
rm -f polinomi.pma
```

4 Prestazioni

Questo capitolo spiega parecchi vantaggi prestazionali che si possono ottenere dall'implementazione della memorie persistente in pm-`gawk`, mostra come una regolazione del Sistema Operativo che si sta usando migliori talvolta le prestazioni e presenta delle misure sperimentali di prestazione. Per restare sul pratico, useremo esempi da un Sistema Operativo GNU/Linux—programmi di utilità GNU in ambiente OS Linux—ma i principi valgono anche per gli altri sistemi operativi moderni.

4.1 Accesso in tempo costante a vettori

pm-`gawk` conserva l'efficienza dell'accesso ai dati quando delle strutture dei dati sono create da uno script e utilizzate più tardi da uno script differente.

Si consideri il vettore associativo usato per analizzare i libri di Mark Twain nel Capitolo 3 [Esempi], pagina 4. Abbiamo creato i vettori `ts[]` ed `hf[]` leggendo i file `sawyer.txt` e `finn.txt`. Se N denota il volume totale di questi file, la costruzione del vettore associativo richiede un tempo proporzionale a N, ossia il "tempo atteso è *O(N)*" nel gergo dell'analisi asintotica. Se W è il numero di parole differenti nei file di input, la dimensione dei vettori associativi sarà proporzionale a W, o *O(W)*. L'accesso a singoli elementi di vettore richiede solo il tempo *costante* o tempo atteso *O(1)* non *O(N)* o *O(W)*, poiché `gawk` implementa i vettori come tabelle hash.

Il vantaggio di prestazioni di pm-`gawk` nasce quando script differenti creano e utilizzano vettori associativi. Ritrovare un elemento di un vettore persistente creato da un precedente programma pm-`gawk`, come fatto più sopra in `BEGIN{print ts["river"], hf["river"]}`, richiede ancora un tempo *O(1)*, che è asintoticamente superiore alle alternative. Ricostruire ingenuamente i vettori a partire dai file di input originali in ogni programma `gawk` che accede ai vettori richiederebbe, naturalmente, un tempo *O(N)*—un costo molto alto, se i testi della raccolta in input sono numerosi. Scaricare i vettori su dei file e quindi caricarli nuovamente quando necessario ridurrebbe il tempo di preparazione all'accesso a *O(W)*. Ciò può essere in pratica un miglioramento notevole. N è all'incirca 19 volte più ampio di W nel nostro esempio con la raccolta di testi di Twain. In ogni caso, *O(W)* rimane molto più lento di *O(1)*, che si ottiene usando pm-`gawk`. Come vedremo nel Sezione 4.6 [Risultati], pagina 21, la differenza non è solo teorica.

L'implementazione della memoria persistente utilizzata da pm-`gawk` lo mette in grado di evitare una mole di lavoro proporzionale a a N o W nell'accedere agli elementi di un vettore associativo persistente. Sotto il coperchio, pm-`gawk` immagazzina le variabili AWK definite in uno script, come i vettori associativi, in un frammento persistente di un file mappato in memoria (il file sparso). Quando uno script AWK utilizza un elemento in un vettore associativo, pm-`gawk` lo ricerca nella tabella hash, dalla quale a sua volta si accede alla memoria del file sparso. I moderni sistemi operativi implementano i file mappati in memoria in modo tale che questi accessi alla memoria richiedono una quantità minima di movimento di dati: solo le parti di file sparso che contengono i dati richiesti sono "paginati" e resi disponibili nella memoria del programma pm-`gawk`. Nel caso peggiore, il file sparso non è nella parte di filesystem già in memoria, e quindi le pagine richieste devono essere effettivamente lette da disco. Le nostre analisi asintotiche di efficienza rimangono valide a prescindere dal fatto che il file sparso sia già in memoria, oppure no. L'intero file sparso *non* è acceduto solo per poter accedere a un elemento di un vettore associativo persistente.

La memoria persistente quindi, rende pm-`gawk` in grado di offrire la flessibilità di separare la fase di acquisizione dati dalle indagini vere e proprie, senza la complicazione e il tempo in più richiesti per serializzare e caricare le strutture dati, e senza sacrificare un tempo di accesso costante ai vettori associativi, il che rende gli script AWK comodi e produttivi.

4.2 Memoria virtuale e file enormi

I file piccoli raramente turbano il piacere di usare AWK, causando problemi di prestazioni, con o senza la memoria persistente. Quando la dimensione delle strutture dati interne dell'interprete `gawk` si avvicina alla capacità della memoria fisica, comunque, per ottenere delle prestazioni accettabili occorre comprendere il funzionamento dei moderni sistemi operativi e talora operare delle regolazioni sugli stessi. Fortunatamente pm-`gawk` offre nuove possibilità di controllo per utenti attenti alle prestazioni, che abbiano a che fare con grossi file di dati. Una frase concisa esprime la filosofia sottesa: Evitare di paginare favorisce una prestazione ottimale e previene delle perplessità.

I moderni sistemi operativi offrono una *memoria virtuale* che si propone di apparire sia maggiore della memoria fisica disponibile (DRAM – che è piccola) che più veloce dei dischi fisici disponibili (che sono lenti). Quando l'utilizzo di memoria di un programma si avvicina alla capacità massima della DRAM, il sistema della memoria virtuale, in maniera trasparente, *pagina* (sposta) i dati del programma tra la DRAM e un'*area di swap* su un disco. La paginazione può degradare le prestazioni in maniera leggera o in maniera pesante, a seconda di come viene acceduta la memoria del programma in esecuzione. Accessi causali a grandi strutture di dati possono provocare una paginazione eccessiva e dei rallentamenti molto vistosi. Sfortunatamente, le tabelle hash che stanno dietro ai tipici vettori associativi di AWK richiedono strutturalmente degli accessi casuali alla memoria, e quindi dei grossi vettori associativi possono essere una fonte di problemi.

La memoria persistente cambia le regola a nostro favore: il Sistema Operativo pagina dei dati verso il *file sparso* di pm-`gawk` invece che all'area swap. Ciò non cambierà di molto le prestazioni se il file sparso risiede in un filesystem tradizionale, su un disco. Sui sistemi di tipo Unix, tuttavia, si può creare il file sparso su un filesystem che risiede in memoria, come `/dev/shm/`, la qual cosa elimina del tutto la paginazione a dischi esterni, strutturalmente più lenti. Mettere temporaneamente il file sparso su un tale filesystem è un accorgimento ragionevole, con due avvertenze. Primo, occorre tenere a mente che i filesystem che risiedono in memoria DRAM cessano di esistere quando la macchina viene riavviata o si impianta. In secondo luogo l'utilizzo di memoria di pm-`gawk` non può eccedere la memoria DRAM disponibile, se si crea il file sparso in un filesystem che risiede nella memoria DRAM.

La regolazione dei parametri di paginazione del Sistema Operativo può migliorare le prestazioni se si decide di eseguire pm-`gawk` utilizzando un file sparso che risieda su un dispositivo disco convenzionale. Alcuni sistemi operativi hanno delle abitudini malsane riguardo alle pagine in memoria modificate ("sporche", ovvero diverse rispetto alle pagine corrispondenti presenti nell'*area di swap*). Per esempio, il Sistema Operativo può scrivere le pagine di memoria "sporche" così modificate al file sparso in maniera periodica e/o quando il Sistema Operativo ritenga che "troppa" memoria sia "sporca". Tale "zelante" comportamento può degradare notevolmente le prestazioni, senza recare beneficio alcuno a pm-`gawk`. Fortunatamente alcuni Sistemi Operativi consentono di modificare i parametri di paginazione in modo che la riscrittura sia "pigra" e non "zelante". Per Linux, vedere la discussione sui parametro `dirty_*` in https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html. Cambiare questi parametri può prevenire un'inutile paginazione "zelante":[1]

```
$ echo 100    | sudo tee /proc/sys/vm/dirty_background_ratio
```

[1] La maniera contorta di effettuare la modifica utilizzando il comando `tee` è spiegata in https://askubuntu.com/questions/1098059/which-is-the-right-way-to-drop-caches-in-lubuntu.

```
$ echo 100    | sudo tee /proc/sys/vm/dirty_ratio
$ echo 300000 | sudo tee /proc/sys/vm/dirty_expire_centisecs
$ echo 50000  | sudo tee /proc/sys/vm/dirty_writeback_centisecs
```

La regolazione dei parametri può essere di aiuto sia per `gawk` non-persistente che per pm-`gawk`.

[**Avvertenza**: La regolazione del Sistema Operativo è un'arte occulta, potreste essere di parere diverso].

4.3 File sparsi

Per risparmiare le risorse di memoria su disco, il file usato come deposito da pm-`gawk` dovrebbe essere creato come un *file sparso*: un file la cui dimensione logica è maggiore dello spazio che occupa su disco. I filesystem moderni supportano i file sparsi, che si possono facilmente definire col comando di utilità `truncate`, come visto negli esempi fin qui fatti.

Iniziamo col creare dapprima un file convenzionale *non* sparso, usando il comando `echo`:

```
$ echo ciao > denso
$ ls -l denso
-rw-rw-r--. 1 me me 5 Aug  5 23:08 denso
$ du -h denso
4.0K    denso
```

Il comando di utilità `ls` ci mostra che il file `denso` è lungo cinque byte (quattro per le lettere in "ciao" più uno per il carattere di ritorno a capo). Il comando di utilità `du` ci informa che questo file consuma 4 KB di memoria—un blocco su disco, lo spazio minimo occupato su disco da un file di tipo non sparso. Usiamo poi il comando di utilità `truncate` per creare un file sparso logicamente enorme, e controlliamo quanto spazio occupa su disco:

```
$ truncate -s 1T sparso
$ ls -l sparso
-rw-rw-r--. 1 me me 1099511627776 Aug  5 22:33 sparso
$ du -h sparso
0       sparso
```

Mentre `ls` ci restituisce la dimensione logica del file che ci attendevamo (un Terabyte, o 2 elevato alla quarantesima potenza), il comando `du` ci informa che il file occupa *zero* byte di memoria. Il filesystem aggiungerà delle risorse di memoria fisica a fronte di questo file man mano che dei dati vengono scritti su di esso; se si va a leggere parti del file che non sono state ancora scritte, il risultato sono dei record a zeri binari.

L'aspetto "paghi solo quello che usi" dello spazio occupato dai file sparsi assicura a chi usa pm-`gawk` sia la convenienza che il controllo. Se il vostro filesystem supporta i file sparsi, utilizzateli creando dei file sparsi di dimensioni abbondanti per pm-`gawk`. La loro dimensione logica non costa nulla e l'allocazione di memoria persistente da parte di pm-`gawk` continuerà a funzionare finché le risorse di memoria fisica disponibili nel filesystem si esauriranno. Se, al contrario, si vuole *evitare* che un file sparso consumi troppo spazio su disco, basta definire la sua dimensione iniziale a qualsiasi limite si voglia porre; il file sparso non utilizzerà più spazio disco di quello specificato. Se si copiano dei file sparsi usando il comando GNU `cp`, esso crea per default delle copie dei file che sono a loro volta dei file sparsi.

Per massimizzare il risparmio di memoria potremmo voler "riorganizzare" i file sparsi, togliendo da questi loro la memoria liberata, che conteneva dati cancellati, di cui pm-`gawk` non ha più bisogno. Un programma di utilità a sé stante, `pma_sam`, è disponibile per fare ciò nel sito web `pma`.

La cifratura di un filesystem può impedire la creazione di file sparsi: se il valore in chiaro della posizione all'interno di un file è tutto a zeri binari il corrispondente valore cifrato dev'essere diverso da una sequenza di zeri! Effettuare la cifratura a livello della memoria [ossia codificando il dato prima di scriverlo], invece che lasciarla fare al filesystem, può offrire un livello accettabile di sicurezza, pur consentendo al filesystem l'implementazione di file sparsi.

Talora si potrebbe preferire un file “normale” (invece che un file sparso), per contenere le variabili e le funzioni della memoria persistente. Per esempio, si potrebbe voler verificare che l'allocazione di memoria interna di pm-`gawk` funzioni correttamente fin quando le informazioni di memoria persistente hanno riempito completamente l'intero file. Il comando di utilità `fallocate` può preparare il file in questione:

```
$ fallocate -l 1M megabyte
$ ls -l megabyte
-rw-rw-r--. 1 me me 1048576 Aug  5 23:18 megabyte
$ du -h megabyte
1.0M    megabyte  # Abbiamo un megabyte, sia logico che fisico.
```

4.4 Persistenza rispetto a durata

Senza dubbio, la regola generale più importante per ottenere delle buone prestazioni dai computer è, "paga solo per ciò che utilizzi".[1] Per applicare questa regola a pm-`gawk` dobbiamo distinguere due concetti che sono frequentemente confusi tra loro: la persistenza e la durata.[2] (Un terzo concetto, logicamente distinto è trattato nel Capitolo 5 [Integrità dei dati], pagina 22.)

I dati *persistenti* sopravvivono ai programmi che ne fanno uso, ma non durano necessariamente per sempre. Per esempio, come spiegato in '`man mq_overview`', le code di messaggi sono persistenti, perché esistono finché il sistema non viene spento. I dati *durevoli* risiedono su un mezzo fisico che mantiene il suo contenuto anche se non è connesso a una presa di corrente. Per esempio, gli hard-disk e i dischi a stato solido contengono dati durevoli. La confusione sorge perché la persistenza e la durata sono spesso correlate: i dati nei normali filesystem che risiedono su HDD o SSD sono tipicamente sia persistenti che durevoli. La familiarità con comandi di utilità come `fsync()` ed `msync()` potrebbe indurre a credere che la durata è un sottoassieme della persistenza, ma in effetti le due caratteristiche sono ortogonali: i dati in un'area di swap sono durevoli, ma non persistenti; i dati in filesystem che risiedono in memoria come `/dev/shm/` sono persistenti, ma non durevoli.

La durata spesso costa di più rispetto alla persistenza. Per questo motivo, gli utenti pm-`gawk` attenti alle prestazioni pagano il costo in più per ottenere la durata solo qualora la semplice persistenza non sia sufficiente. Due modi per evitare il costo della durata sono stati discussi nel Sezione 4.2 [Memoria virtuale e file enormi], pagina 12: Si può porre il file sparso usata da pm-`gawk` in un filesystem che risiede nella memoria DRAM del computer, e si può inibire l'eccessiva riscrittura del file sparso. Espedienti come questo permettono di togliere il sovraccarico della durata dal tempo necessario per analisi di dati effettuate in molti stadi. Questo vale perfino se si desidera che i file sparsi rimangano poi durevoli: consentono a pm-`gawk` di funzionare a piena velocità usando solo la persistenza; forzano per il file utilizzato la durabilità (usando i comandi `cp` e `sync` secondo necessità) dopo che l'output è stato prodotto, in vista della successiva fase di analisi, e dopo che il programma pm-`gawk` che usa il file sparso è terminato.

La sperimentazione utilizzando dati creati in modo casuale aiuta a comprendere come la persistenza e la durata influenzano le prestazioni. Si può scrivere un piccolo programma in C o in AWK per generare un flusso di testo casuale, o si può improvvisare un generatore usando semplicemente la riga di comando:

```
$ openssl rand --base64 1000000 | tr -c a-zA-Z '\n' > casuale.dat
```

---

[1] In effetti, questa regola è largamente ignorata, in maniere sorprendenti. Alcuni ben noti algoritmi, descritti in libri di testo, continuano la loro esecuzione ben oltre il momento in cui hanno calcolato tutto ciò che dovevano calcolare.
Vedere https://queue.acm.org/detail.cfm?id=3424304.

[2] Recentemente il termine "Memoria Persistente" è stato talora usato per indicare un nuovo tipo di memoria non-volatile, indirizzabile byte per byte—una pratica infelice, che contraddice delle sensate convenzioni in essere da tempo, e che causa una confusione ingiustificata. Tale tipo di Memoria-Non-Volatile fornisce durata [e non persistenza]. La Memoria Persistente qui descritta è invece un'astrazione software, che non richiede l'utilizzo di Memoria-Non-Volatile [hardware]. Vedere https://queue.acm.org/detail.cfm?id=3358957.

Modificando la dimensione degli input casuali si può, per esempio, trovare quando le prestazioni "vanno sotto terra", perché l'uso di memoria di pm-`gawk` eccede la capacità della memoria DRAM del computer e la paginazione ha inizio.

Le sperimentazioni richiedono un'attenzione meticolosa, specie quando il file sparso si trova in un filesystem che risiede nella memoria del computer. Non tener conto della cache del filesystem presente nella memoria DRAM può facilmente fuorviare nell'interpretazione dei risultati e rendere inaffidabile la ripetibilità. Fortunatamente il Sistema Operativo Linux ci permette di svuotare la cache del filesystem e quindi simulare una condizione di "partenza a freddo" quale si avrebbe subito dopo una ripartenza del computer. In tale situazione, l'accesso ai file normali sulla memoria durevole avverrebbe leggendo dal dispositivo fisico, e non dalla cache in memoria. Una spiegazione [in inglese] riguardo ai comandi `sync` e `/proc/sys/vm/drop_caches` si trova in
https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html.

4.5 Esperimenti

Lo script di C-shell (`csh`) che segue illustra i concetti e implementa i suggerimenti presentati in questo capitolo. È servito per produrre i risultati trattati nel Sezione 4.6 [Risultati], pagina 21, è stato eseguito in una ventina di minuti, su un laptop non troppo recente. Per riprodurre i miei esperimenti, copiate/incollate le righe e mettetele in un file; assicuratevi che non ci siano righe omesse o inserite più di una volta.

Lo script misura le prestazioni di quattro differenti modi di effettuare delle indagini sulla frequenza delle parole contenute in una raccolta di testi. L'approccio ingenuo, leggere la raccolta di testi e metterla in un vettore associativo ad ogni esecuzione di una *query*; scrivere una volta una rappresentazione in formato testo della tabella con la frequenza delle parole e caricarla all'inizio di ogni nuova *query*; usare l'estensione di `gawk rwarray` per scaricare e in seguito ricaricare un vettore associativo; usare pm-`gawk` per mantenere un vettore associativo persistente.

I commenti iniziali spiegano i prerequisiti. Le righe 8–10 gestiscono i parametri in input: la directory in cui eseguire i test e dove si trovano i file necessari, compreso il file sparso, il cronometro usato per misurare i tempi di esecuzione, e altre caratteristiche di prestazione, come l'uso massimo di memoria e la dimensione dell'input. La dimensione di default dell'input richiede da parte di pm-`gawk` l'utilizzo di una memoria di quasi 3GB, che è sufficientemente grande da produrre risultati interessanti, e sufficientemente piccola per essere contenuta nella memoria DRAM del computer, evitando (sui computer in uso oggi) che sia necessaria la paginazione. Le righe 13–14 definiscono un cronometro improvvisato.

Due sezioni dello script sono rilevanti se la directory di default in cui eseguire i test viene cambiata da un filesystem nella memoria del computer `/dev/shm/` a una directory su un normale filesystem che risiede su un disco esterno. Le righe 15–17 definiscono il meccanismo per eliminare gli eventuali dati presenti nella cache DRAM; le righe lines 23–30 impostano i parametri del *kernel* che scoraggiano il ricorso frequente alla paginazione.

Le righe 37–70 creano, compilano ed eseguono un piccolo programma scritto in C, che genera una raccolta di testi casuale. Il programma è veloce, flessibile e deterministico, ossia genera sempre lo stesso output casuale, se eseguito con gli stessi parametri.

Le righe 71–100 testano i quattro differenti approcci ad AWK a fronte dello stesso input casuale, registrando separatamente il tempo di preparazione e quello di *query* del vettore associativo che contiene la frequenza delle parole.

```
#!/bin/csh -f
# Imposta la var. d'amb. PMG al percorso dell'eseguibile pm-gawk e AWKLIBPATH       #   2
#    per trovare rwarray.so                                                        #   3
# Serve "sudo" per funzionare; si consideri se inserire in /etc/sudoers:           #   4
#    Defaults:vostro_nome_utente    !autentica                                     #   5
echo 'inizio:      ' `date` `date +%s`                                            #   6
unsetenv GAWK_PERSIST_FILE  # disabilita persistenza per ora                      #   7
set dir = '/dev/shm'        # dove si trova il file sparso etc.                  #   8
set tmr = '/usr/bin/time'   # volendo, disponibile anche "time" della shell      #   9
set isz = 1073741824        # dimensione input, 1GB                              #  10
# set isz = 100000000       # input piccolo per fare test veloci                 #  11
cd $dir   # tick/tock/tyme sotto sono timer improvvisati, errore ~2ms           #  12
alias tick 'set t1 = `date +%s.%N` ; alias tock 'set t2 = `date +%s.%N`'         #  13
alias tyme '$PMG -v t1=$t1 -v t2=$t2 "BEGIN{print t2-t1}"'                       #  14
alias tsync 'tick ; sync ; tock ; echo "sync time:  " `tyme`'                    #  15
```

```
alias drop_caches 'echo 3 | sudo tee /proc/sys/vm/drop_caches'                      #  16
alias snd 'tsync; drop_caches'                                                      #  17
echo "pm-gawk:     $PMG" ; echo 'std gawk:   ' `which gawk`                         #  18
echo "run dir:     $dir" ; echo 'pwd:        ' `pwd`                                #  19
echo 'dir content:'      ; ls -l $dir |& $PMG '{print "             " $0}'         #  20
echo 'timer:      ' $tmr ; echo 'AWKLIBPATH: ' $AWKLIBPATH                          #  21
echo 'Parametri Sistema Operativo:' ; set vm = '/proc/sys/vm/dirty'                 #  22
sudo sh -c "echo 100     > ${vm}_background_ratio"     # rimettere questi           #  23
sudo sh -c "echo 100     > ${vm}_ratio"                # parametri di paginazione   #  24
sudo sh -c "echo 1080000 > ${vm}_expire_centisecs"     # ai loro valori di default  #  25
sudo sh -c "echo 1080000 > ${vm}_writeback_centisecs"  # se necessario              #  26
foreach d ( ${vm}_background_ratio  ${vm}_ratio \                                   #  27
            ${vm}_expire_centisecs  ${vm}_writeback_centisecs )                     #  28
    printf "             %-38s %7d\n" $d `cat $d`                                   #  29
end                                                                                 #  30
tick ; tock ; echo 'timr ovrhd: ' `tyme` 's (circa 2ms per TK)'                     #  31
tick ; $PMG 'BEGIN{print "pm-gawk?     yes"}'                                       #  32
tock ; echo 'pmg ovrhd:  ' `tyme` 's (circa 4-5 ms per TK)'                         #  33
set inp = 'input.dat'                                                               #  34
echo 'input dime. ' $isz                                                            #  35
echo "input file:  $inp"                                                            #  36
set rg = rgen  # crea e compila programma C per generare input casuali              #  37
rm -f $inp  $rg.c $rg                                                               #  38
cat <<EOF > $rg.c                                                                   #  39
// genera N parole casuali, una per riga, nessuna riga bianca                      #  40
// caratteri sono e.g. 'abcdefg@' dove '@' diventa 'a capo'                        #  41
#include <stdio.h>                                                                 #  42
#include <stdlib.h>                                                                #  43
#include <string.h>                                                                #  44
#define RCH     c = a[rand() % L];                                                 #  45
#define PICK    do { RCH } while (0)                                               #  46
#define PICKCH  do { RCH } while (c == '@')                                        #  47
#define FP(...) fprintf(stderr, __VA_ARGS__)                                       #  48
int main(int argc, char *argv[]) {                                                 #  49
  if (4 != argc) {                                                                 #  50
    FP("usage:  %s  charset  N  seed\n",                                           #  51
                    argv[0]);   return 1; }                                        #  52
  char c, *a =      argv[1];    size_t L = strlen(a);                              #  53
  long int N = atol(argv[2]);                                                      #  54
  srand(       atol(argv[3]));                                                     #  55
  if (2 > N) { FP("N == %ld < 2\n", N); return 2; }                                #  56
  PICKCH;                                                                          #  57
  for (;;) {                                                                       #  58
    if (2 == N) { PICKCH; putchar(c); putchar('\n'); break; }                      #  59
    if ('@' == c) { putchar('\n'); PICKCH; }                                       #  60
    else          { putchar( c  ); PICK;   }                                       #  61
    if (0 >= --N) break;                                                           #  62
  }                                                                                #  63
}                                                                                  #  64
EOF                                                                                #  65
gcc -std=c11 -Wall -Wextra -O3 -o $rg $rg.c                                        #  66
set t = '@@@@@@@' ; set c = "abcdefghijklmnopqrstuvwxyz$t$t$t$t$t$t"               #  67
tick ; ./$rg "$c" $isz 47 > $inp ; tock ; echo 'gen time:   ' `tyme`               #  68
echo "input file:  $inp"                                                           #  69
echo 'input wc:   ' `wc < $inp` ; echo 'input uniq: ' `sort -u $inp | wc`         #  70
```

```
snd ################################################################################    #  71
tick ; $tmr $PMG '{n[$1]++}END{print "output: " n["foo"]}' $inp                        #  72
tock ; echo 'T ingenuo O(N):          ' `tyme` ; echo ''                               #  73
rm -f rwa                                                                              #  74
snd ################################################################################    #  75
echo ''                                                                                #  76
tick ; $tmr $PMG -l rwarray '{n[$1]++}END{print "writea",writea("rwa",n)}' $inp        #  77
tock ; echo 'T rwarray prepara O(N):  ' `tyme` ; echo ''                               #  78
snd  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #    #  79
tick ; $tmr $PMG -l rwarray 'BEGIN{print "reada",reada("rwa",n); \                     #  80
                                print "output: " n["foo"]}'                            #  81
tock ; echo 'T rwarray query O(W):   ' `tyme` ; echo ''                                #  82
rm -f ft                                                                               #  83
snd ################################################################################    #  84
tick ; $tmr $PMG '{n[$1]++}END{for(w in n)print n[w], w}' $inp > ft                    #  85
tock ; echo 'T freqtbl prepara O(N):  ' `tyme` ; echo ''                               #  86
snd  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #    #  87
tick ; $tmr $PMG '{n[$2] = $1}END{print "output: " n["foo"]}' ft                       #  88
tock ; echo 'T freqtbl query O(W):   ' `tyme` ; echo ''                                #  89
rm -f heap.pma                                                                         #  90
snd ################################################################################    #  91
truncate -s 3G heap.pma  # allargare se serve                                          #  92
setenv GAWK_PERSIST_FILE heap.pma                                                      #  93
tick ; $tmr $PMG '{n[$1]++}' $inp                                                      #  94
tock ; echo 'T pm-gawk prepara O(N):  ' `tyme` ; echo ''                               #  95
snd  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #    #  96
tick ; $tmr $PMG 'BEGIN{print "output: " n["foo"]}'                                    #  97
tock ; echo 'T pm-gawk query O(1):    ' `tyme` ; echo ''                               #  98
unsetenv GAWK_PERSIST_FILE                                                             #  99
snd ################################################################################    # 100
echo 'Nota:  tutte le righe di output sopra dovrebbero essere identiche' ; echo ''     # 101
echo 'contenuto dir:' ; ls -l $dir |& $PMG '{print "              " $0}'               # 102
echo '' ; echo 'memoria usata:'                                                        # 103
foreach f ( rwa ft heap.pma )  # compressione molto lenta, la omettiamo               # 104
    echo "    $f " `du -BK $dir/$f` # `xz --best < $dir/$f | wc -c` 'bytes xz'       # 105
end                                                                                    # 106
echo '' ; echo 'fine:       ' `date` `date +%s` ; echo ''                              # 107
```

4.6 Risultati

Eseguendo lo script nel Sezione 4.5 [Esperimenti], pagina 18, con i parametri di default in un laptop non troppo recente ha dato i risultati riassunti nella tabella che segue. Esperimenti più ampi, non riferiti qui, conducono a risultati qualitativamente simili. Occorre tenere presente che le misurazioni di prestazione sono spesso influenzate da fattori apparentemente irrilevanti. Per esempio, il programma in esecuzione può avere il vantaggio di una CPU raffreddata meglio; ulteriori esecuzioni possono essere effettuati con una CPU più calda, e di conseguenza subire gli effetti delle variazioni di *clock* messe in atto dall'apparato di regolazione termica di un moderno processore. Molto più in generale, le misurazioni di prestazione sono faccende delicate. Se si eseguono degli script, per i quali la facilità di preparazione prevale sulla necessità di esecuzione veloce, la funzione propria delle misurazioni di prestazione è quella di testare qualitativamente delle ipotesi come quelle derivanti da un'analisi asintotica, e di fornire un'idea approssimata di quando i vari approcci possibili siano preferibili in pratica.

AWK script		tempo es. (sec)	massimo uso memoria (K)	memoria intermedia (K)
ingenuo	O(N)	242,132	2.081.360	n/a
rwarray prep.	O(N)	250,288	2.846.868	156.832
rwarray query	O(W)	11,653	2.081.444	
freqtbl prep.	O(N)	288,408	2.400.120	69.112
freqtbl query	O(W)	11,624	2.336.616	
pm-gawk prep.	O(N)	251,946	2.079.520	2.076.608
pm-gawk query	O(1)	0,026	3.252	

I risultati sono consistenti con l'analisi asintotica nel Sezione 4.1 [Accesso in tempo costante a vettori], pagina 10. Tutti e quattro gli approcci richiedono circa quattro minuti per la lettura dei dati in input. L'approccio ingenuo deve farlo ogni volta che si effettua una *query*, ma gli altri tre approcci costruiscono un unico vettore associativo, usato per ognuna delle successive *query*. Gli approcci `freqtbl` ed `rwarray` costruiscono un vettore associativo di parole, lo scaricano su un file intermedio, che rileggono prima di ogni *query*; il primo dei due lo fa manualmente mentre l'altro usa un'estensione `gawk`. Entrambi possono eseguire una nuova *query* in una decina di secondi, non in quattro minuti. Come previsto dall'analisi asintotica, eseguire un lavoro in un tempo proporzionale al numero di parole nel vettore è preferibile a eseguire il lavoro in un tempo proporzionale alla dimensione della raccolta di testi in input: il tempo *O(W)* è minore di *O(N)*. E, sempre come previsto, le *query* di pm-`gawk`'s a tempo costante sono ancor più veloci, all'incirca di un paio di ordini di grandezza. Per i calcoli qui presi in esame, pm-`gawk` fa la differenza fra un batter d'occhio e un tempo di risposta sufficientemente lungo perché la mente dell'utente cominci a pensare ad altro.

Mentre `freqtbl` ed `rwarray` ricostruiscono un vettore associativo, prima di accedere a un elemento dello stesso, pm-`gawk` immagazzina nella memoria persistente un vettore associativo pronto per l'uso. Questo è il motivo per cui il suo file intermedio (il file sparso) è molto più grande degli altri due file intermedi, e la ragione per cui il file sparso è quasi grande quanto il massimo uso di memoria da parte di pm-`gawk` nella fase di preparazione

del vettore associativo, mentre l'uso di memoria è molto piccolo quando usato da una *query* che accede a un singolo elemento del vettore. Il vantaggio che deriva dall'avere un file sparso grande è il tempo di accesso *O(1)* invece che *O(W)*—un classico compromesso fra tempo e spazio. Se la memoria su disco è una risorsa scarsa, tutti e tre i file intermedi possono essere compressi, quello `freqtbl` di un fattore all'incirca 2,7x, `rwarray` all'incirca 5,6x, e pm-`gawk` all'incirca 11x, usando il comando `xz`. La compressione usa molto la memoria ed è lenta, un altro compromesso fra tempo e spazio.

5 Integrità dei dati

Contrattempi come le mancanze di corrente, gli errori del *kernel*, i bug negli script, e i refusi sulla riga-dei-comandi possono causare danni ai vostri dati, ma ci sono delle precauzioni che servono a diminuire questi rischi. Negli scenari che prevedono l'uso di script basta di solito creare un backup dei file importanti, in tempi appropriati. Per quanto semplice questo sembri, è necessario prestare attenzione per ottenere una protezione genuina e per ridurre il costo del backup. Quel che segue è una maniera prudente e poco costosa di creare un backup del file sparso fra un uso e l'altro:

```
$ nome_backup=sparso_bkup`date +%s`
$ cp --reflink=always sparso.pma $nome_backup.pma
$ chmod a-w $nome_backup.pma
$ sync
$ touch $nome_backup.fatto
$ chmod a-w $nome_backup.fatto
$ sync
$ ls -l sparso*
-rw-rw-r--. 1 me me 4096000 Aug  6 15:53 sparso.pma
-r--r--r--. 1 me me       0 Aug  6 16:16 sparso_bkup1659827771.fatto
-r--r--r--. 1 me me 4096000 Aug  6 16:16 sparso_bkup1659827771.pma
```

La marcatura temporale aggiunta al nome nei file di backup rende facile trovare la copia più recente se il file sparso è danneggiato, anche se l'informazione sull'ultima data di modifica sia stata inavvertitamente modificata.

L'opzione `--reflink` del comando `cp` riduce sia l'utilizzo di memoria su disco che il tempo richiesto per effettuare la copia. Allo stesso modo un cui i file sparsi offrono della memoria su disco del tipo "paga solo per ciò che utilizzi", la copia di tipo reflink offre della memoria del tipo "paga solo per ciò che *modifichi*".[1] Una copia reflink condivide memoria col file originale. Il filesystem garantisce che ulteriori modifiche a uno dei due file non riguarderà l'altro. La copia con reflink non è disponibile per tutti i filesystem; al momento è supportata dai filesystem di tipo XFS, BtrFS, e OCFS2.[2] Fortunatamente è possibile

[1] La chiamata-a-sistema che implementa la copiatura di tipo reflink è descritta nella in '`man ioctl_ficlone`'.

[2] L'opzione `--reflink` crea copie di un file sparso mantenendolo tale. Se la copiatura con reflink non è disponibile, si dovrebbe usare invece l'opzione `--sparse=always`.

installare, per esempio, un filesystem XFS *all'interno di un file ordinario* in qualche altro filesystem, come `ext4`.[3]

Dopo aver creato una copia di backup del file sparso, usiamo il comando `sync` per forzare la scrittura dalla memoria a un disco. In caso contrario la copia potrebbe risiedere sono nella memoria volatile DRAM—nella cache del filesystem—dove una caduta del sistema o una mancanza di corrente potrebbero corromperla.[4] Dopo aver usato `sync` sul backup, creiamo e forziamo la scrittura con `sync` di un file "indicatore di successo" con una estensione `.fatto` per prevenire un pericoloso evento, che potrebbe succedere: ci può essere una mancanza di corrente *mentre* un file di backup è copiato dal file sparso originale, lasciando uno dei file (o entrambi) corrotti in memoria—una possibilità particolarmente preoccupante, per lavori che vengono eseguiti senza un operatore presente. Dopo una ripartenza, ogni file `.fatto` attesta che il corrispondente backup è andato a buon fine, facilitando il riconoscimento del backup riuscito più recente.

Per finire, se si è seriamente intenzionati a ovviare a eventuali problemi hardware/software si deve essere "addestrati come per combattere" testando il vostro hardware/software contro dei problemi che realisticamente possono presentarsi. Per un realistico test dei problemi correlati alle mancanze di corrente, vedere [in inglese] https://queue.acm.org/detail.cfm?id=3400902.

[3] Vedere https://www.usenix.org/system/files/login/articles/login_winter19_08_kelly.pdf.

[4] In alcuni Sistemi Operativi il comando `sync` non garantisce molto, ma in ambiente Linux `sync` termina sono quando tutti i dati di filesystem sono stati scritti su una memoria durevole. Se nel vosgtro sistema `sync` è inaffidabile, si può scrivere un piccolo programma in C che invochi la chiamata-a-sistema `fsync()` per forzare la scrittura di un file. Per maggiore sicurezza, è meglio chiamare `fsync()` per ogni directory a livello superiore nel percorso del file (`realpath()`) fino a giungere alla *root*.

6 Ringraziamenti

Haris Volos, Zi Fan Tan e Jianan Li hanno sviluppato una versione prototipo persistente di `gawk`, a partire da una diramazione del codice sorgente di `gawk`. I consigli che ho ricevuto dal manutentore di `gawk`, Arnold Robbins, e che ho inoltrato a loro, sono risultati essere molto utili. Robbins, inoltre, ha implementato, documentato e testato pm-`gawk` per la versione ufficiale di `gawk`; strada facendo, egli ha suggerito numerosi miglioramenti per l'allocatore di memoria `pma` che sta dietro a pm-`gawk`. Corinna Vinschen ha suggerito altri miglioramenti a `pma` e testato pm-`gawk` in ambiente Cygwin. Nelson H. F. Beebe ha fornito accesso a macchine Solaris durante la fase di test. Robbins, Volos, Li, Tan, Jon Bentley e Hans Boehm hanno rivisto le bozze di questo manuale utente, e le loro osservazioni sono state utili. Bentley ha suggerito l'esempio minimo/massimo/media nel Capitolo 3 [Esempi], pagina 4, e anche l'esercizio di rendere persistente lo script "Markov", tratto dal testo di Kernighan e Pike. Volos ha fornito e testato i suggerimenti sulla regolazione dei parametri del Sistema Operativo nella Sezione 4.2 [Memoria virtuale e file enormi], pagina 12. Stan Park ha fornito degli approfondimenti riguardo a memoria virtuale, filesystem e programmi di utilità. Antonio Giovanni Colombo ha tradotto questo manuale in italiano; vedere il file `doc/it/pm-gawk.texi` nella distribuzione di `gawk`.

Appendice A Installazione

Gli utenti possono ottenere la funzionalità pm-`gawk` tramite il sistema di gestione dei pacchetti software o effettuando un'installazione manuale. Entrambi gli approcci hanno dei pro e dei contro.

Pacchetti Delegare il compito di installare al sistema di gestione dei pacchetti software, disponibile nelle più importanti distribuzioni Linux, compreso Ubuntu e Fedora, è più semplice—in teoria. I pacchetti software resi disponibili in alcune distribuzioni Linux, tuttavia, sono "indietro" anche di anni rispetto agli ultimi rilasci software di un particolare comando. Per questo motivo, funzionalità relativamente nuove potrebbero essere disponibili solo in rilasci "successivi", ma non nei pacchetti "regolari" presenti nella distribuzione Linux. Se la prima riga restituita dal comando `gawk --version` indica la versione 5.2 o successive e contiene "`PMA`," la versione disponibile di pm-`gawk` supporta la memoria persistente; in caso contrario, vedere see "Installazione Manuale" più sotto.

Un altro problema con il software installato tramite un gestore di pacchetti è che il software può essere stato compilato/preparato in maniera non corretta. Per esempio, a febbraio 2025, il `gawk` 5.3.0 installato dal gestore pacchetti in Fedora 41 è stato creato, in maniera errata, come un programma indipendente dalla posizione di esecuzione (PIE — Position Independent Executable), nonostante il metodo di compilazione inserito nella distribuzione ufficiale di `gawk` disabiliti in modo deliberato e in maniera esplicita la costruzione di un programma di tipo PIE. Il risultato, come evidenziato in Capitolo 2 [Per iniziare], pagina 3, è che pm-`gawk` va in errore la seconda volta che viene invocato. I dettagli del problema con il pacchetto distribuito da Fedora mentre ci sono dei "lavori in corso", a inizio febbraio 2025, sono disponibili in

`https://bugzilla.redhat.com/show_bug.cgi?id=2341653`

Per mezzo del programma di utilità `file` si può controllare se `gawk` è stato compilato in maniera errata. In ambiente Fedora 41, per esempio:

```
$ which gawk
/usr/bin/gawk
$ file /usr/bin/gawk
/usr/bin/gawk: ELF 64-bit LSB pie ...
```

La parolina "`pie`" è probabilmente da incolpare se il vostro pm-`gawk` va in errore alla *seconda* invocazione. Per default, i programmi PIE vengono eseguiti in modalità ASLR (address-space layout randomization—rendere casuali alcuni indirizzi usati nei programmi), qualcosa che `gawk` dalla versione 5.2 fino alla corrente versione 5.3.1 non supporta.

(Nota: L'allocatore di memoria persistente che rende possibile la memoria persistente `gawk`—la libreria di funzioni `pma`—è perfettamente compatibile con le funzionalità PIE e ASLR. L'incompatibilità fra `gawk` e ASLR sorge dalle strutture interne dei dati dell'interprete, che richiedono che i puntatori alle funzioni siano consistenti fra un'invocazione e l'altra del programma. ASLR introduce delle inconsistenze non desiderate in questi puntatori.)

Nell'attesa di una correzione elegante e definitiva per il problema PIE di Fedora 41, si può evitare il problema nella maniera qui indicata: Eseguire l'interprete pm-`gawk` tramite '`setarch -R`', disabilitando in questo modo ASLR anche se il programma è PIE. Ecco il primo esempio da Capitolo 2 [Per iniziare], pagina 3, dapprima senza, e poi con la correzione proposta:

```
$ truncate -s 4096000 heap.pma
$ export GAWK_PERSIST_FILE=heap.pma
$ gawk 'BEGIN{myvar = 47}'
$ gawk 'BEGIN{myvar += 7; print myvar}'
Segmentation fault (core dumped)    # ...grazie al PIE

$ rm -f heap.pma                    # scarta file usato prima
$ truncate -s 4096000 heap.pma      # inizia di nuovo
$ setarch -R gawk 'BEGIN{myvar = 47}'
$ setarch -R gawk 'BEGIN{myvar += 7; print myvar}'
54                                  # come dev'essere
```

Si consiglia di definire un alias di shell per ottenere, quando si invoca il familiare ed ergonomico `gawk`, il verboso e prolisso '`setarch -R gawk`'.

Alternativamente, potrebbe essere possibile disabilitare la funzionalità ASLR per l'intero sistema operativo, usando il comando `sysctl` per modificare variabili come `kernel.randomize_va_space`. Non ho esplorato questa possibilità, che andrebbe comunque usata con cautela. Occorre essere consapevoli di tutte le implicazioni, prima di richiedere un così brusco cambiamento a livello dell'intero sistema.

È ragionevole aspettarsi che, in circostanze normali, una versione preparata correttamente di pm-`gawk` sarà prima o poi disponibile nel `gawk` installato da un gestore di pacchetti nelle principali distribuzioni GNU/Linux. Nel frattempo, un'installazione manuale è una modalità abbastanza semplice per avere a disposizione un pm-`gawk` che supporti la memoria permanente.

Installazione Manuale La compilazione manuale, a partire dal codice sorgente, permette di ottenere l'ultima versione stabile disponibile di `gawk`, seguendo la "ricetta" fornita dal manutentore, senza modifiche introdotte da intermediari. Un'installazione eseguita in

questo modo sarà esente da problemi come quello del PIE sopra descritto. L'ultima versione stabile può essere scaricata qui:

https://ftp.gnu.org/gnu/gawk/
https://ftp.gnu.org/gnu/gawk/gawk-5.3.1.tar.xz

Chi desideri la versione "più fresca" disponibile, e voglia avventurarsi nel fai-da-te, può scaricare la versione "master" mantenuta sotto `git` da
http://git.savannah.gnu.org/cgit/gawk.git/snapshot/gawk-master.tar.gz
Dopo aver espanso il file scaricato, eseguite `./bootstrap.sh`, `./configure`, `make` e `make check`, e quindi potete provare ad eseguire alcuni degli esempi presentati sopra.

A febbraio 2025, pm-`gawk` è supportato da buona parte, ma non da tutte le principali piattaforme di tipo Unix. Il processo di compilazione di `gawk` decide se includere o meno la funzionalità della memoria persistente; vedere la documentazione `gawk` e il file `./m4/pma.m4` nella distribuzione `gawk`.

Appendice B Debugging

Per bug non correlati alla persistenza, vedere la documentazione `gawk`, e.g., *GAWK: Effective AWK Programming*, disponibile in https://www.gnu.org/software/gawk/manual/. [La versione italiana dello stesso libro è disponibile in https://sites.google.com/view/gawkdoc-it/home-page.]

Se pm-`gawk` non si comporta secondo le attese, dovreste per prima cosa domandarvi se state usando il file sparso che intendevate usare; usare il file sbagliato è un errore comune. Altre feconde sorgenti di bug per principianti sono il fatto che una regola `BEGIN` è eseguita ogni volta che si invoca pm-`gawk`, il che non è sempre ciò che veramente si vuole, nonché il fatto che le variabili predefinite di AWK, come per esempio `NR` sono sempre inizializzate a zero ogni volta che si invoca il programma. Vedere la trattazione dell'inizializzazione per lo script minimo/massimo/media nel Capitolo 3 [Esempi], pagina 4.

Se pm-`gawk` va in errore alla *seconda* invocazione che usa un particolare file di appoggio della memoria persistente, vedere la discussione della funzionalità in Appendice A [Installazione], pagina 24.

Se si sospetta qualche altro tipo di problema connesso con la memoria persistente in pm-`gawk`, si può impostare una variabile d'ambiente per far sì che il codice `pma`, che gestisce la persistenza, invii dei messaggi diagnostici più numerosi; per i dettagli, si veda la documentazione principale di `gawk`.

Programmatori: Si può ricompilare `gawk` abilitando delle asserzioni, il che causerà degli estesi controlli di integrità all'interno del codice `pma`. Assicuratevi che il file sorgente `pma.c` sia compilato *senza* il flag `-DNDEBUG`, quando si compila `gawk` usando il comando `make`. Il programma eseguibile che ne risulta dovrebbe essere testato con input piccoli, perché i controlli di integrità possono rallentare di molto l'esecuzione. Se qualche asserzione fallisce, significa che c'è un bug da qualche parte in pm-`gawk`. Tali bug vanno segnalati a me (Terence Kelly) come pure seguendo le procedure indicate nella documentazione principale di `gawk`. Specificate quale versione di `gawk` si sta usando, e cercate di fornire uno script piccolo e semplice che permetta di riprodurre il bug in maniera affidabile.

Appendice C Storia

La funzionalità di Memoria Persistente in pm-gawk si basa su un nuovo Allocatore di Memoria Persistente, pma, le cui specifiche di progettazione [in inglese] sono descritte in https://queue.acm.org/detail.cfm?id=3534855. È istruttivo ripercorrete le fasi dell'evoluzione che ha condotto a pma e pm-gawk.

Ho preparato parecchi script AWK durante le ricerche per la mia tesi di laurea sul Web caching, 25 anni fa, la maggior parte dei quali analizzava file di log provenienti da server Web e da cache Web. Avere a disposizione gawk con la Memoria Persistente avrebbe reso questi script più piccoli, più veloci, e più facili da scrivere, ma allora non ero neppure in grado di immaginare che pm-gawk sarebbe stato possibile. Quindi ho scritto un mucchio di codice inefficiente e tedioso che manualmente creava e ricaricava variabili AWK utilizzando file di testo. Un decennio doveva passare prima che i miei colleghi e io iniziassimo a mettere insieme i pezzi che hanno reso possibile script che usano la Memoria Persistente, e un altro decennio sarebbe passato prima che pm-gawk fosse reso in grado di operare.

All'incirca nel 2011, mentre lavoravo agli HP Labs, ho sviluppato una piattaforma di calcolo distribuito tollerante agli errori, chiamata "Ken", che conteneva un allocatore di Memoria Persistente che assomigliava a una versione semplificata di pma: rendeva disponibile una interfaccia in linguaggio C simile a malloc() e allocava memoria tramite una mappatura della memoria, che era mantenuta anche su un file. L'esperienza fatta con Ken mi convinse che l'astrazione software della Memoria Persistente è molto attraente, rispetto alle alternative, per gestire dati persistenti (p.es., database relazionali e depositi chiave-valore). Sfortunatamente, l'allocatore di memoria di Ken è così profondamente interconnesso col resto di Ken da risultare praticamente inseparabile; per poter fruire dei benefici della Memoria Persistente di Ken occorreva "comprare" anche una vasta serie di altro complicato software. Quali che fossero i suoi altri pregi, Ken non è l'ideale per servire da esempio riguardo ai benefici della Memoria Persistente in se stessa.

Un altro aspetto convoluto di Ken era un meccanismo per resistere ai crash del computer che, in retrospettiva, può essere visto come un'implementazione a livello di utente di quello che fa la chiamata a sistema msync() resistente a un crash. Il primo sforzo di estrazione del dopo-Ken ha estratto il meccanismo di resistenza ai crash, e l'ha implementato nel kernel Linux, e il risultato è stato definito "failure-atomic msync()" (FAMS – msync() resistente a un crash). FAMS rafforza il significato del normale comando msync() garantendo che lo stato permanente di un file mappato in memoria rifletta sempre la più recente chiamata a msync(), perfino in presenza di eventi come una mancanza di corrente e di crash sia a livello di programma che di Sistema Operativo. Il prototipo originale del FAMS nel kernel Linux è descritto in un saggio di Park et al. negli atti del convegno EuroSys 2013. I miei colleghi e io abbiamo in seguito implementato FAMS in parecchie maniere differenti, fra cui i filesystem (FAST 2015) e le librerie a livello di utente. La mia implementazione più recente di FAMS, che utilizza la funzionalità di copiatura reflink, descritta in altra parte di questo manuale, è ora alla base di una nuova funzionalità di resistenza ai crash nel venerabile e onnipresente comando GNU dbm (gdbm) per gestire dei database (https://queue.acm.org/detail.cfm?id=3487353).

Negli ultimi anni la mia attenzione è tornata sui vantaggi della Memoria Persistente nella programmazione, che è diventata un argomento "caldo" ai tempi del COVID grazie

alla fugace disponibilità commerciale di memoria non-volatile indirizzabile a byte (Intel Optane che, per confondere le idee, era descritta dal marketing come "Memoria Persistente"). L'astrazione software della Memoria Persistente e il corrispondente stile di programmazione sono perfettamente compatibili con dei computer *convenzionali*—macchine non dotate di memoria non-volatile e neppure di altro software e hardware particolari. Parecchi saggi sono stati scritti per spiegare questo punto, per esempio [in inglese] https://queue.acm.org/detail.cfm?id=3358957.

A inizio 2022 ho scritto un nuovo allocatore di memoria persistente, a sé stante, `pma`, per facilitare l'uso della Memoria Persistente su hardware convenzionale. L'interfaccia `pma` è compatibile con la chiamata a sistema `malloc()` e, a differenza dell'allocatore presente in Ken, `pma` non è collegato a un particolare meccanismo di resistenza ai crash. Usare `pma` è semplice e piacevole.

Ken è stato integrato in prototipi derivanti sia dall'interprete di Javascript V8 che da un interprete Scheme, e quindi è parso naturale domandarsi se `pma` avrebbe analogamente potuto potenziare un linguaggio interpretato di script. GNU AWK è stata una scelta naturale perché il codice sorgente è ordinato, e perché `gawk` ha un solo manutentore principale, con una mente aperta riguardo all'aggiunta di nuove funzionalità.

Jianan Li, Zi Fan Tan, Haris Volos, e io abbiamo iniziato a prendere in esame la persistenza per `gawk` alla fine del 2021. Mentre io stavo scrivendo `pma`, essi hanno preparato un prototipo di pm-`gawk` a partire da una diramazione della distribuzione sorgente ufficiale di `gawk`. L'esperienza con il prototipo ha confermato la convenienza e i benefici per le prestazioni di pm-`gawk`, e a partire dalla primavera del 2022 Arnold Robbins ha iniziato a implementare la persistenza nella versione ufficiale di `gawk`. La funzionalità di Memoria Persistente nel comando `gawk` ufficiale è lievemente differente da quella del prototipo: la prima usa una variabile d'ambiente per comunicare all'interprete il nome del file sparso, mentre l'altra usava un'opzione obbligatoria da inserire nella riga-di-comando. Per molti aspetti, comunque, le due implementazioni sono simili. Una descrizione del prototipo, con anche delle misurazioni di prestazioni è disponibile in http://nvmw.ucsd.edu/nvmw2022-program/nvmw2022-data/nvmw2022-paper35-final_version_your_extended_abstract.pdf.

Mi piacciono parecchie cose di pm-`gawk`. Non è invadente; man mano che acquisite familiarità ed esperienza, scivola sullo sfondo della vostra programmazione. È semplice sia come concetto che come implementazione e, cosa ancora più importante, semplifica i vostri script; molto del suo valore si apprezza non tanto nel codice che mette in grado di scrivere, quanto nel codice che consente di eliminare. È tutto ciò (e anche più) di cui avevo bisogno nella mia tesi di laurea 25 anni fa. Guardando ai fatti, sembra ispirare la creatività di coloro che l'hanno adottato per primi. Sono curioso di vedere quali nuove finalità di utilizzo si possono trovare per esso.