Dispense per il corso di algoritmica per il web

Sebastiano Vigna

7 novembre 2022

Indice

1 Notazione e definizioni di base

Il prodotto cartesiano degli insiemi X e Y è l'insieme $X \times Y = \{ \langle x, y \rangle \mid x \in X \wedge y \in Y \}$ delle coppie ordinate degli elementi di X e Y. La definizione si estende per ricorsione a n insiemi. Al prodotto cartesiano $X_1 \times X_2 \times \cdots \times X_n$ sono naturalmente[1] associate le *proiezioni* $\pi_1, \pi_2, \ldots, \pi_n$ definite da

$$\pi_i(\langle x_1, x_2, \ldots, x_n \rangle) = x_i.$$

Poniamo

$$X^n = \overbrace{X \times X \times \cdots \times X}^{n \text{ volte}}$$

e $X^0 = \{ * \}$ (qualunque insieme con un solo elemento). La *somma disgiunta* degli insiemi X e Y è, intuitivamente, un'unione di X e Y che però tiene separati gli elementi comuni ("evita i conflitti"). Formalmente,

$$X + Y = X \times \{ 0 \} \cup Y \times \{ 1 \}.$$

Di solito ometteremo, con un piccolo abuso di notazione, la seconda coordinata (0 o 1).

Una *relazione* tra gli insiemi $X_1, X_2, \ldots, X_n$ è un sottoinsieme R del prodotto cartesiano $X_1 \times X_2 \times \cdots \times X_n$. Se $n = 2$ è uso scrivere $x \; R \; y$ per $\langle x, y \rangle \in R$. Una relazione tra due insiemi è detta *binaria*. Se R è una relazione binaria tra X e Y, X è detto il *dominio* di R, ed è denotato da $\operatorname{dom} R$, mentre Y è detto il *codominio* di R, ed è denotato da $\operatorname{cod} R$. Il *rango* o *insieme di definizione* di R è l'insieme $\operatorname{ran} R = \{ x \in X \mid \exists y \in Y \; x \; R \; y \}$, e in generale può non coincidere con il dominio di R. L'*immagine* di R è l'insieme $\operatorname{imm} R = \{ y \in Y \mid \exists x \in X \; x \; R \; y \}$, e in generale può non coincidere con il codominio di R. Una relazione binaria R tra X e Y è *monodroma* se per ogni $x \in X$ esiste al più un $y \in Y$ tale che $x \; R \; y$. È *totale* se per ogni $x \in X$ esiste un $y \in Y$ tale che $x \; R \; y$, cioè se $\operatorname{ran} R = \operatorname{dom} R$. È *iniettiva* se per ogni $y \in Y$ esiste al più un $x \in X$ tale che $x \; R \; y$. È *suriettiva* se per ogni $y \in Y$ esiste un $x \in X$ tale che $x \; R \; y$, cioè se $\operatorname{imm} R = \operatorname{cod} R$. È *biiettiva* se è sia iniettiva che suriettiva.

Una *funzione* da X a Y è una relazione monodroma e totale tra X e Y (notate che l'ordine è rilevante); in tal caso scriviamo $f : X \to Y$ per dire che f "va da X a Y". Se f è una funzione da X a Y è uso scrivere $f(x)$ per l'unico $y \in Y$ tale che $x \; f \; y$. Diremo che f *mappa* x in $f(x)$, o, in simboli, $x \mapsto f(x)$. Le nozioni di dominio, codominio, iniettività, suriettività e biiettività vengono ereditate dalle relazioni. Se una funzione $f : X \to Y$ è biiettiva, è facile verificare che esiste una funzione *inversa* f^{-1}, che soddisfa le equazioni $f(f^{-1}(y)) = y$ e $f^{-1}(f(x)) = x$ per ogni $x \in X$ e $y \in Y$. Una *funzione parziale*[2] da X a Y è una relazione monodroma tra X e Y; una funzione parziale può non essere definita su elementi del suo dominio, fatto che

[1] L'uso dell'aggettivo "naturale", qui e altrove, è tecnico, e relativo alla teoria delle categorie. *Dixi, et salvavi animam meam*.

[2] È una sfortunata evenienza che una funzione parziale *non* sia una funzione. Ragioni storiche hanno consolidato ormai da tempo questa disgraziata nomenclatura.

denotiamo con la scrittura $f(x) = \perp$ ("$f(x)$ è indefinito" o "f è indefinita su x"), che significa $x \notin \operatorname{ran} f$. Date funzioni (parziali) $f : X \to Y$ e $g : Y \to Z$, la *composizione* $g \circ f$ di f con g è la funzione definita da $(g \circ f)(x) = g(f(x))$. Si noti che, per convenzione, $f(\perp) = \perp$ per ogni funzione (parziale) f. Dati gli insiemi X e Y, denotiamo con $Y^X = \{ f \mid f : X \to Y \}$ l'insieme delle funzioni da X a Y. Si noti che per insiemi finiti[3] $\left|Y^X\right| = |Y|^{|X|}$.

Denoteremo con n l'insieme $\{ 0, 1, 2, \ldots, n-1 \}$.[4].

Dato un insieme X, il *monoide libero su* X, denotato da X^*, è l'insieme di tutte le sequenze finite (inclusa quella vuota, normalmente denotata con ε) di elementi di X, dette *parole su* X, dotate dell'operazione di concatenazione, di cui la parola vuota è l'elemento neutro. Denoteremo con $|w|$ il numero di elementi di X della parola $w \in X$.

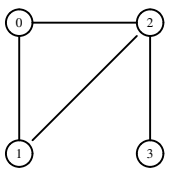
Dato un sottoinsieme A di X, possiamo associargli la sua *funzione caratteristica* $\chi_A : X \to 2$, definita da

$$\chi_A(x) = \begin{cases} 0 & \text{se } x \notin A \\ 1 & \text{se } x \in A. \end{cases}$$

Per contro, a ogni funzione $f : X \to 2$ possiamo associare il sottoinsieme di X dato dagli elementi mappati da f in 1, cioè l'insieme $\{ x \in X \mid f(x) = 1 \}$; tale corrispondenza è inversa alla precedente, ed è quindi naturalmente equivalente considerare sottoinsiemi di X o funzioni da X in 2.

Date funzioni $f, g : \mathbf{N} \to \mathbf{R}$, diremo che f è *di ordine non superiore a* g, e scriveremo $f \in O(g)$ ("f è O-grande di g") se esiste una costante $\alpha \in \mathbf{R}$ tale che $|f(n)| \leq |\alpha g(n)|$ definitivamente. Diremo che f è *di ordine non inferiore a* g, e scriveremo $f \in \Omega(g)$ se $g \in O(f)$. Diremo che f è *dello stesso ordine di* g, e scriveremo $f \in \Theta(g)$, se $f \in O(g)$ e $g \in O(f)$.

Un *grafo (semplice)* G è dato da un insieme finito di *vertici* V_G e da un insieme di *lati* $E_G \subseteq \{ \{ x, y \} \mid x, y \in V_G \wedge x \neq y \}$; ogni lato è cioè una coppia non ordinata di vertici distinti. Se $\{ x, y \} \in E_G$, diremo che x e y sono *adiacenti* in G. Un grafo può essere rappresentato graficamente disegnando i suoi vertici come punti sul piano, e rappresentando i lati come segmenti che congiungono vertici adiacenti. Per esempio, il grafo con insieme di vertici 4 e insieme di lati $\{ \{0,1\}, \{1,2\}, \{2,0\}, \{2,3\} \}$ si può rappresentare così:



L'*ordine* di G è il numero naturale $|V_G|$. Una *cricca (clique)* di G è un insieme di vertici $C \subseteq V_G$ mutuamente adiacenti (nell'esempio in figura, $\{0,1,2\}$ è una cricca). Dualmente, un *insieme*

[3] L'uguaglianza è vera in generale, utilizzando i cardinali cantoriani.

[4] In effetti, questa è la definizione di von Neumann degli interi

indipendente di G è un insieme di vertici $I \subseteq V_G$ mutuamente non adiacenti. Un *cammino* (di lunghezza n) in G è una sequenza di vertici $x_0, x_1, \ldots, x_n$ tale che x_i è adiacente a x_{i+1} ($0 \leq i < n$). Diremo che il cammino va da x_0 a x_n. Nell'esempio in figura, $1, 0, 2$ è un cammino, mentre $1, 3$ non lo è.

Un *grafo (orientato)* G è dato da un insieme di nodi N_G, da un insieme di archi A_G e da funzioni $s_G, t_G : A_G \to N_G$ (*source* e *target*) che specificano l'inizio e la fine di ogni arco. Due archi a e b tali che $s_G(a) = s_G(b)$ e $t_G(a) = t_G(b)$ sono detti *paralleli*. Un grafo senza archi è paralleli è *separato*. Il *grado positivo* (*outdegree*) $d^+(x)$ di un nodo x è il numero di archi che partono x, cioè $|s_G^{-1}(x)|$. Dualmente, il *grado negativo* (*indegree*) $d^-(x)$ è $|t_G^{-1}(x)|$. In un grafo orientato G un *cammino* (di lunghezza n) è una sequenza di vertici e archi $x_0, a_0, x_1, a_1, \ldots, a_{n-1}, x_n$ tale che $s_G(a_i) = x_i$ e $t_G(a_i) = x_{i+1}$ per $0 \leq i < n$. Diremo che il cammino va da x_0 a x_n.

Definiamo la relazione di *raggiungibilità*: $x \rightsquigarrow y$ se esiste un cammino da x a y. La relazione di equivalenza $\sim$ è ora definita da $x \sim y \iff x \rightsquigarrow y \wedge y \rightsquigarrow x$. Le classi di equivalenza di $\sim$ sono dette *componenti fortemente connesse* di G, e G è *fortemente connesso* quando è costituito da un'unica componente.

La funzione $\lambda(x)$ denota il bit più significativo dell'espansione binaria di x: quindi $\lambda(1) = \lambda(1_2) = 0$, $\lambda(2) = \lambda(10_2) = 1$, $\lambda(3) = \lambda(11_2) = 1$, e così via. Per convenzione, $\lambda(0) = -1$. Si noti che per $x > 0$ si ha $\lambda(x) = \lfloor \log x \rfloor$.

2 Crawling

Il *crawling* è l'attività di scaricamento delle pagine web. Un *crawler* visita pagine web a partire da un insieme dato di *semi*, e continua a visitare nuove pagine seguendo i collegamenti ipertestuali. Più precisamente, ad ogni istante gli URL possibili sono divisi in tre insiemi:

- l'insieme V degli URL visitati: l'URL è stato scaricato e analizzato;
- la frontiera F: l'URL è conosciuto (o perché nel seme, o perché raggiungibile dal seme tramite collegamenti ipertestuali) ma non è ancora stato visitato;
- l'insieme U degli URL sconosciuti: tutto il resto.

Da un punto di vista astratto, l'attività di crawling parte caricando la frontiera con un insieme di URL forniti esternamente, il *seme*. Dopodiché, finché la frontiera non è vuota il crawler estrae un URL dalla frontiera secondo qualche criterio, lo visita (cioè scarica la pagina corrispondente), lo analizza derivandone nuovi URL tramite i collegamenti ipertestuali contenuti nella pagina, e lo sposta nell'insieme degli URL visitati. I nuovi URL ottenuti vengono aggiunti alla frontiera se sono sconosciuti, cioè se non stanno in $V \cup F$.

Il problema fondamentale di quest'attività è la gestione della frontiera. Infatti, la frontiera è in generale ordini di grandezza più grande dell'insieme dei visitati. La prima grande differenza

tra l'attività di crawling e una normale visita di grafi è che l'insieme dei nodi sconosciuti è effettivamente sconosciuto — non ne conosciamo neppure la cardinalità.

Diverse politiche di prioritizzazione della frontiera possono poi dare luoghi a tipi diversi di crawling (per esempio, è possibile estrarre prima URL a cui si arriva da pagine che contengono delle parole chiave specifiche).

2.1 Il crivello

Il crivello è la struttura dati di base di un crawler: accetta in ingresso URL potenzialmente da visitare e permette di prelevare URL pronti per la visita. Ogni URL che viene inserito nel crivello esce una sola volta, indipendentemente da quante volte è stato inserito. In questo senso il crivello unisce le proprietà di un dizionario a quelle di una coda con priorità, e rappresenta al tempo stesso la frontiera, l'insieme dei visitati e la coda di visita. Combinare questi aspetti in una sola struttura dati permette dei risparmi notevoli dal punto di vista pratico.[5]

Una prima osservazione è che spesso per mantenere l'informazione di quali URL sono stati già visti (cioè in $V \cup F$) è preferibile sostituire gli URL con delle *firme*, cioè tramite l'output di una buona funzione di *hash* applicata gli URL stessi. Per esempio, è possibile utilizzare firme a 64 bit.

Anche semplicemente inserendo le firme in un dizionario standard (per esempio, una tabella di hash), si ottiene un grande risparmio di spazio, perché un URL ora occuperà solo 64 bit. In cambio, c'è ora la possibilità che il dizionario, visto come un dizionario di URL, restituisca dei *falsi positivi*, cioè sostenga di contenere un URL che non abbiamo mai aggiunto.

Questo fenomeno è inevitabile e dovuto alle *collisioni*, cioè al fatto che più URL necessariamente avranno la stessa firma. Se il numero di collisioni è molto piccolo, possiamo in genere ignorarle.

Se si hanno in memoria n firme, la probabilità che una nuova firma collida con una di quelle esistenti è n/u, dove u è la dimensione dell'universo delle firme. Nel caso di firme da 64 bit, $u = 2^{64}$, e quindi possiamo memorizzare 100 miliardi di URL con una probabilità di falsi positivi dell'ordine di $100 \times 10^9/2^{64} < 2^{37}/2^{64} = 1/2^{27} < 1/10^8$, cioè meno di un errore ogni cento milioni di URL.

Anche un semplice dizionario di firme in memoria che rappresenta $V \cup F$, accoppiato a una coda o pila su disco che tiene traccia di F, è sufficiente per attività di crawling di piccole dimensioni. Per dimensioni più grandi, però, è necessario utilizzare strutture dati (parzialmente o completamente su disco) che soddisfino un requisito fondamentale: l'occupazione di memoria centrale deve essere costante.

Questo è un compromesso tipico delle attività di crawling — strutture che si espandono in memoria centrale proporzionalmente alla frontiera sono troppo fragili (cioè rischiano di causare l'esaurimento della memoria, o di sovraccaricare il sistema di gestione della memoria virtuale).

[5] Si noti che è possibile riordinare ulteriormente gli URL *dopo* l'uscita dal crivello.

Più in generale, le strutture utilizzate per il un sovraccarico la struttura dovrebbe entrare una fase di *degrado grazioso* (*graceful degradation*): dovrebbe cioè ridurre le sue prestazioni, senza però interrompere all'improvviso il suo funzionamento.

2.1.1 I filtri di Bloom

La prima struttura che vedremo con queste proprietà è il *filtro di Bloom*. Un filtro di Bloom [Blo70] è una semplicissima struttura dati probabilistica che rappresenta un dizionario, cioè un insieme di elementi da un universo dato. Permette di aggiungere elementi all'insieme e chiedere se un elemento appartiene o no all'insieme, con il rischio di ottenere falsi positivi.

Un filtro di Bloom con universo X è rappresentato da un vettore $\boldsymbol{b}$ di m bit e da d funzioni di hash $f_0, f_1, \ldots, f_{d-1}$ da X in m. Per aggiungere un elemento di $x \in X$ al filtro, vengono posti a uno i bit $b_{f_k(x)}$ $(0 \leq k < d)$. Per sapere se un elemento x appartiene al filtro, si controlla se tutti i bit $b_{f_k(x)}$ $(0 \leq k < d)$ sono a uno, e solo in questo caso si risponde positivamente.

Intuitivamente, ogni volta che un elemento viene aggiunto al filtro la conoscenza della presenza dell'elemento viene sparsa in d bit a caso, che vengono interrogati quando è necessario sapere se l'elemento fa parte dell'insieme: è però possibile che i d bit in questione siano stati messi a uno dall'aggiunta di *altri* elementi, ed è quindi possibile avere dei *falsi posiviti* — risposte positive all'appartenenza di elementi che non sono mai stati aggiunti al filtro.

Il nome "filtro" in effetti deriva dall'idea che la struttura dovrebbe venire usata per filtrare le richieste a una struttura dati esatta soggiacente più lenta. Se si prevede che la maggior parte delle richieste avrà risposta negativa, un filtro di Bloom può ridurre significativamente gli accessi alla struttura soggiacente. Di fatto, i filtri di Bloom sono risultati estremamente pratici per mantenere insiemi di grande dimensione in memoria, in particolare quando le dimensioni delle chiavi sono significative (per esempio, URL).

Andiamo ora a vedere qual è la probabilità di un falso positivo. Con un'analisi ragionevolmente precisa (quella che presenta Bloom in [Blo70]) saremo in grado di fornire valori ottimi di m e d data la probabilità di falsi positivi desiderata e il massimo numero di elementi nel filtro. In questo modo saremo in grado di scegliere la struttura dati meno ingombrante per ottenere una probabilità di falsi positivi scelta a piacere.

In realtà, l'analisi fornisce una stima della probabilità di osservare *un positivo, vero o falso che sia*, dopo n inserimenti: ovviamente questa probabilità è una maggiorazione della probabilità di un falso positivo.

La probabilità che dopo n inserimenti uno specifico bit sia zero è

$$\left(1 - \frac{1}{m}\right)^{dn}.$$

dato che la probabilità che un singolo bit sia zero, se ne imposto uno a caso, è $1 - 1/m$, e assumiamo che le funzioni di hash siano uniformemente distribuite e indipendenti.

Per ottenere un positivo, dobbiamo trovare uno in tutti i punti del vettore che controlliamo. Questo avviene con probabilità

$$\varphi = \left(1 - \left(1 - \frac{1}{m}\right)^{dn}\right)^d \approx \left(1 - e^{-\frac{dn}{m}}\right)^d,$$

dove abbiamo utilizzato il fatto che $(1 + \alpha/n)^n \to e^\alpha$ per $n \to \infty$.[6]

Poniamo ora $p = e^{-nd/m}$, sicché $d = -(m/n)\ln p$. Il nostro scopo è quindi quello di minimizzare

$$(1-p)^{-(m/n)\ln p} = e^{-(m/n)\ln p \ln(1-p)}.$$

La derivata prima è dunque

$$-\frac{m}{n} e^{-(m/n)\ln p \ln(1-p)} \left(\frac{\ln(1-p)}{p} - \frac{\ln p}{1-p}\right),$$

e si azzera quando

$$(1-p)\ln(1-p) = p \ln p.$$

Dato che sia a sinistra che a destra abbiamo la stessa funzione $x \ln x$, una soluzione è certamente data da $1 - p = p$, cioè $p = 1/2$. Per convincerci che questa è anche l'unica soluzione andiamo a studiare $g(p) = p \ln p - (1-p)\ln(1-p)$. Agli estremi la funzioni vale zero, dato che

$$\lim_{p\to 0} p \ln p = \lim_{p\to 0} \frac{\ln p}{1/p} = \lim_{p\to 0} \frac{1/p}{-1/p^2} = \lim_{p\to 0} -p = 0.$$

La derivata è invece

$$g'(p) = \ln(1-p) + \ln p + 2.$$

Chiaramente $g'(p)$ va a meno infinito in zero e uno, ma in $p = 1/2$ è positiva ($2\ln 2 < 2$, perché $2 < e$), e $p = 1/2$ è il solo massimo (dato che la derivata $g''(p) = 1/p - 1/(1-p)$ ha un solo zero in $p = 1/2$). Concludiamo che $g(p)$ ha esattamente un massimo e un minimo in $[0\,..\,1]$, e quindi esattamente uno zero in $(0\,..\,1)$.

Alla fine, la probabilità di (falsi) positivi è minimizzata da $d \approx m \ln 2/n$, e in tal caso la probabilità di un (falso) positivo è 2^{-d}. Vale a dire, possiamo migliorare esponenzialmente la probabilità di errore aumentando linearmente il numero di funzioni di hash e il numero di bit a $m \approx dn/\ln 2 \approx 1{,}44\, dn$.

Alcune osservazioni tecniche:

[6] Si noti che stiamo qui utilizzando un'ulteriore approssimazione, e cioè che i d eventi siano indipendenti. Equivalentemente, stiamo trattando il numero di bit impostati a uno come se fosse esattamente uguale alla sua media, anziché solo concentrato attorno alla sua media.

- È abbastanza intuitivo, ed è possibile dimostrare, che per evere falsi positivi con probabilità 2^k occorre utilizzare almeno k bit per elemento. Un filtro di Bloom perde quindi 44% in spazio rispetto al minimo possibile.

- In linea di principio il filtro di Bloom ha una modelità di accesso alla memoria pessima, perché richiede d accessi casuali, e quindi potenzialmente d fallimenti di cache.

- Ciononostante, il dimensionamento ottimo di un filtro di Bloom è lineare nel numero di chiavi attese. Questo fa sì che se dividiamo le chiavi in k segmenti utilizzando una funzione di hash e costruiamo un filtro per segmento l'occupazione in spazio non aumenta. Dimensionando k in modo che i segmenti abbiano la dimensione di una o due linee di cache si può abbattere il numero di fallimenti di cache dei positivi (questa implementazione è detta *block*). In questo caso però l'approssimazione che abbiamo utilizzato diventa non più così precisa; inoltre, la divisione delle chiavi in segmenti non è mai uniforme, ma segue invece una distribuzione binomiale negativa. Questi fattori peggiorano la probabilità di errore [PSS10].

- Dall'analisi che abbiamo effettuato, $1/2$ è anche la probabilità di un bit a zero. Questo significa che bastano in media due accessi medi per ottenere un risultato negativo. Il filtro di Bloom è quindi potenzialmente lento solo quando il risultato è positivo, e se in effetti a valle si trova una struttura dati esatta più lenta il costo dell'accesso alla struttura rende irrilevante l'alto costo dei positivi.

- In teoria per utilizzare un filtro di Bloom dobbiamo calcolare d funzioni di hash diverse, il che può essere molto costoso in termini di tempo. In realtà, Kirsch and Mitzenmacher hanno dimostrato che estraendo due numberi interi a 64 bit a e b tramite una funzione di hash, i numeri $ai + b$, $0 \leq i < d$ sono d hash sufficienti a replicare l'analisi condotta utilizzando funzioni indipendenti e pienamente casuali.

- Se un filtro di Bloom viene utilizzato per rappresentare gli URL già visti, soddisfa pienamente le nostre richieste: utilizza una quantità di memoria constante e degrada graziosamente—semplicemente, una volta superato il numero di URL per cui il filtro è dimensionato la probabilità di falsi positivi aumenterà fino ad arrivare a 1.

2.1.2 Crivelli basati su database NoSQL

Un modo più sofisticato (e con un degrado più grazioso) di implementare un crivello è quello di utilizzare un cosiddetto *database NoSQL*, che consiste semplicemente in una struttura parzialmente su disco che permette di memorizzare coppie chiave/valore utilizzando una quantità limitata di memoria centrale.

Uno degli esempi classici di database NoSQL è il BerkeleyDB, che permette di memorizzare coppie chiave/valore in maniera non ordinata o ordinata tramite una hash table e B-tree parzialemente su disco. La memoria centrale è utilizzata come cache per accelerare le operazioni su disco. Si tratta quindi di una versione parzialmente su disco di strutture dati classiche.

Un approccio più sistematico, implementato inizialemente a Google sotto il nome di BigTable, è l'LSM tree [OCGO96] (*Log-Structured Merge tree*, cioè *albero di fusione strutturato a registri*).

BigTable è stato successivamente reimplementato come un progetto a sorgente aperto, LevelDB, che è stato poi utilizzato come base per altri database NoSQL come RocksDB, l'implementazione di Facebook, che è utilizzata da commoncrawler, un crawler a sorgente aperto.

Gli LSM tree sono basati su un concetto relativamente semplice, ma necessitano di un'implementazione accurata che sfrutti parallelismo e concorrenza per essere efficienti.

Il contenuto di un LSM-tree è dato da vari *livelli*, ognuno dei quali contiene un sottoinsieme delle coppie chiave/valore che si intende rappresentare. Ogni livello ha una dimensione di base che cresce di un fattore dato rispetto al livello precedente (per esempio, 10), ma ha una certa elasticità nel dimensionamento (per esempio, può essere grande fino al doppio della sua dimensione di base).

Il primo livello di un LSM-tree risiede in memoria centrale, ha dimensione limitata a priori ed è implementato tramite un normale dizionario ordinato (tipicamente, un albero rosso-nero o un B-albero).

I livelli successivi sono memorizzati semplicemente come registri (*log*): sono una successione immutabile di coppie chiave/valore ordinate.

Il primo aspetto importante di un LSM-tree è che una chiave può comparire *in più livelli*: il valore associato è quello che compare nel livello più alto in cui è possibile trovare la chiave. Un'interrogazione in lettura consiste quindi in una ricerca della chiave a partire dal primo livello: non appena la chiave viene trovata, si sa il suo valore.

La parte interessante è quella di scrittura: la coppia chiave/valore viene inizialmente inserita nel primo livello. Se a questo punto il primo livello eccede la sua dimensione massima, un insieme di chiavi viene estratto dal primo livello e fuso con il secondo livello in modo da riportare il primo livello alla sua dimensione naturale. Si noti che in questa operazione è persino possibile che il secondo livello non venga del tutto modificato.

A questo punto la fusione viene scaricata ricorsivamente sui livelli successivi: se il secondo livello eccede la sua dimensione massima, di nuovo ne viene estratto un insieme di chiavi che viene fuso con il terzo livello, e così via. Se tutti i livelli eccedono la loro dimensione massima, alla fine della fusione viene creato un nuovo livello.

Si noti che tutte le operazioni su disco avvengono sequenzialmente, e che tutti i dati memorizzati su disco sono *immutabili*. Queste due caratteristiche rendono le fusioni estremamente efficienti nelle architetture moderne, e semplificano notevolmente la gestione degli accessi paralleli.

Per cancellare una associazione chiave/valore viene inserita una coppia con la stessa chiave e un valore detto *lapide* (*tombstone*); la tecnica è simile a quella utilizzata per le tabelle di *hash*. La lapide viene trattata come ogni altro valore, ma in fase di ricerca se troviamo una lapide ci fermiamo e consideriamo la chiave come assente dall'albero. Se una lapide arriva all'ultimo livello viene scartata.

Ci sono a questo punto numerose e importanti questioni ingegneristiche e implementative:

- Il formato in cui vengono memorizzati i livelli può non essere uniforme, e può dipendere dalla memoria di massa soggiacente; per esempio, un formato che funziona bene per i dischi a stato solido può non essere adatto a un disco rigido o a un nastro.
- Ogni livello può essere frammentato in file più piccoli per permettere di selezionare più liberamente le chiavi da fondere, e per rendere più semplice operare le fusioni in parallelo. In questo caso, ogni chiave compare in un solo frammento.
- Ogni frammento può essere arricchito con un dizionario approssimato con falsi positivi a bassa precisione (vedremo, ad esempio, un filtro di Bloom) che evita di cercare la maggior parte delle chiavi che non compaiono nel frammento. La bassa precisione fa sì che l'occupazione in spazio del dizionario sia poco influente.
- Ogni frammento può contenere un indice sparso che memorizza le posizioni di sottoinsieme chiavi campionate a intervalli regolari; in questo modo, in fase di ricerca è possibile identificare rapidamente la zona del frammento che potenzialmente può contenere la chiave, e poi procedere con una ricerca binaria (se le coppie chiave/valore sono tutte della stessa lunghezza) o lineare. La frequenza di campionamento bilancia lo spazio addizionale occupato e la velocità di accesso.
- Anche in assenza di fusioni di livelli, in genere un LSM tree ha dei processi concorrenti di *compattamento* che cercano di evitare che ci siano troppe copie della stessa chiave nei vari livelli, e soprattutto che non ci siano in giro troppe lapidi.
- Infine, tutte le operazioni di fusione non vengono effettuate veramente durante gli inserimenti, ma piuttosto vengono svolte con continuità da processi concorrenti.

Ci sono anche soluzioni ibride, che reinseriscono parzialmente gli alberi bilanciati negli LSM tree. Questo tipo di tecnologia è in continua evoluzione, anche perché diverse implentazioni o politiche di aggiornamento posso essere adatte a diversi carichi di lavoro.

Si noti che al crescere della frontiera un LSM tree diventa più lento (e occupa più memoria di massa), ma l'utilizzo di memoria centrale rimane costante, e non si ha un decremento di precisione.

2.1.3 Un crivello offline

Un modo meno responsivo ma molto semplice di implementare il crivello che effettua una visita in ampiezza e richiede memoria costante senza utilizzare strutture dati consiste nel tenere traccia, in ogni istante, di tre file:

- un file Z di URL già visti (cioè $V \cup F$), in ordine lessicografico;
- un file F di URL da visitare (cioè F), in ordine di scoperta;
- un file A, di lunghezza limitata a priori, che accumula temporaneamente gli URL incontrati durante la visita.

All'inizio dell'attività di crawl, Z e F sono inizializzati utilizzando il seme, e A è vuoto. Durante l'attività di crawl, gli URL da visitare vengono estratti da F (eventualmente alterandone l'ordine seguendo un criterio di priorità), e i nuovi URL che vengono incontrati vengono accumulati nel file A.

Quando F è vuoto, o A raggiunge la sua lunghezza massima, procediamo a un'operazione di *fusione* come segue:

- A viene ordinato e deduplicato, e il risultato è il file A';
- Z e A' vengono fusi per ottenere un file Z' che andrà a rimpiazzare Z;
- durante la fusione, gli URL che compaiono in A' ma non in Z vengono accodati a F.

La fusione di Z e A' può essere effettuata con letture sequenziali perché i due file sono ordinati. È evidente che ogni URL che viene incontrato dal crawler viene accodato a F esattamente una volta, e cioè durante la fusione che avviene dopo la prima volta che compare in A.

Questo tipo di organizzazione non è particolarmente performante se viene effettuata dalla macchina che effettua l'attività di crawling, sebbene l'ordinamento di A si possa effettuare in memoria costante. Se però è possibile ordinare e fondere file utilizzando un framework di ordinamento distributo, come MapReduce [DG08], o la sua implementazione a sorgente aperto, Hadoop, le prestazioni possono essere molto migliorate, e la semplicità del codice può giocare a favore di questa scelta.

Va notato che l'ordinamento effettuato su A altera l'ordine di accodamento. Per recuperare l'effetto di una visita in ampiezza è necessario recuperare l'ordine originale degli URL. Questo risultato si può ottenere, per esempio, memorizzando in A oltre agli URL la posizione ordinale della loro prima occorrenza, e riordinando i nuovi URL scoperti in tale ordine prima di accodarli a F. È anche possibile tenere A quando si crea A', e durante la fusione mantenere invece di una

lista di URL scoperti una lista di *posizioni in A* di URL scoperti. A quel punto è sufficiente ordinare la lista di posizioni e scandirla in parallelo con A per estrarre sequenzialmente e nell'ordina di accodamento in A gli URL scoperti.

Infine, da un punto di vista pratico è conveniente mantenere in Z non gli URL già visti, ma le loro firme. Per fare funzionare correttamente il passo di fusione è però a questo punto necessario ordinare e deduplicare A utilizzando come chiavi non gli URL, ma le loro firme.

Si noti che al crescere della frontiera il crivello offline diventa più lento (e occupa più memoria di massa), ma l'utilizzo di memoria centrale rimane costante, e non si ha un decremento di precisione.

2.1.4 Il crivello di Mercator

Il crivello originale di Mercator [HN99] è una versione parzialmente in memoria del crivello offline. Le firme degli elementi di A vengono mantenute in un vettore in memoria, evitando di eseguire ordinamenti su disco.

Il crivello è formato da un vettore S in memoria centrale che contiene firme di URL, inizialmente vuoto, che viene riempito incrementalmente. Il vettore è fisso e di dimensione n. Su disco, invece, teniamo un file Z che contiene tutte le firme dei URL sinora mai incontrati, e un file ausiliario A, inizialmente vuoto.

Ogni volta che un URL u viene inserito nel crivello, aggiungiamo $h(u)$ a S e u al file A. Il punto chiave è che cosa succede quando S arriva a contenere n firme; operiamo allora uno *scarico* nel seguente modo:

1. Ordiniamo S *indirettamente* per valore. Vale a dire, ordiniamo *stabilmente* un vettore V di lunghezza n che contiene esattamente i numeri in n utilizzando come chiave il valore $S[i]$. A questo punto $V[i]$ contiene l'indice in S della firma di rango i (cioè l'i-esima in ordine), e quindi le firme $S[V[i]]$ sono in ordine crescente al crescere di i.

2. Utilizzando questa proprietà, deduplichiamo S: andiamo cioè a marcare come inutili tutte le occorrenze di firme duplicate, tenendo per buona solo la prima. Si noti che stiamo sfruttando il fatto che l'algoritmo di ordinamento utilizzato è stabile (altrimenti potremmo marcare occorrenze successive alla prima).

3. Ora fondiamo Z con le firme marcate in un nuovo file Z'. Possiamo farlo in tempo lineare e scandendo Z in sequenza, dato che gli $S[V[i]]$ sono ordinati. Marchiamo le firme in S che non sono duplicate e che non compaiono in Z.

4. Infine, scandiamo A e S in parallelo, e per ogni firma marcata in S diamo in output l'URL corrispondente in A. Si noti che la scansione di A è puramente sequenziale, dato che S è nello stesso ordine (ed è questa la ragione per cui utilizziamo un ordinamento indiretto).

5. S e A vengono svuotati, e Z sostituito da Z'.

Innanzitutto, si noti the Z, alla fine di uno scarico, contiene di nuovo le firme di tutti gli URL mai incontrati. Inoltre, in output abbiamo dato tutti e soli gli URL la cui firma non era parte di Z, e quindi (modulo collisioni tra firme) tutti e soli gli URL sconosciuti. Infine, gli URL in output sono stati ovviamente emessi nell'ordine di accodamento in A.

2.2 Gestione dei quasi-duplicati

Durante l'attività di crawling è comune trovare pagine che sono quasi identiche (varianti dello stesso sito, calendari, gallerie di immagini, ecc.). In dipendenza dal tipo di crawling (vengono scaricate le immagini?), queste pagine andrebbero considerate duplicate e non ulteriormente elaborate.

Un modo semplice ma efficace di gestire in memoria centrale il problema dei quasi-duplicati è calcolare una forma normalizzata del testo di una pagina (per esempio, eliminando la marcatura, le date, ecc.) e memorizzarla in un dizionario approssimato, come i filtri di Bloom.

Metodi molto più sofisticati per la rilevazione dei duplicati possono essere utilizzati offline, prima dell'indicizzazione.

Un metodo efficace di gestione online del problema, che è stato utilizzato per qualche tempo dal crawler di Google [MJDS07], è quello di porre in un dizionario (eventualmente approssimato) uno hash generato dall'algoritmo SimHash di Charikar [Cha02]. L'algoritmo genera hash che sono simili (nel senso che hanno distanza di Hamming bassa) per pagine simili. In particolare, si può usare l'identità di SimHash come definizione di quasi-duplicato.

Per calcolare SimHash dobbiamo prima di tutto stabilire il numero b di bit dello hash, e fissare una buona funzione di hash h che mappa stringhe in hash di b bit. A un maggiore numero di bit corrisponderà una nozione più accurata di somiglianza. A questo punto il testo della pagina (in forma normalizzata) viene trasformato in un insieme di segnali S: un modo banale è utilizzare le parole del testo come segnali, ma è più accurato considerare n-grammi per n piccolo (tipicamente tra 3 e 5).

A ogni segnale $s \in S$ associamo ora uno hash $h(s)$. Il SimHash del testo ha il bit i ($0 \leq i < b$) impostato a uno se e solo se

$$|\{s \in S \mid \text{il bit } i \text{ di } h(s) \text{ è uno}\}| > |\{s \in S \mid \text{ il bit } i \text{ di } h(s) \text{ è zero}\}|.$$

Due documenti che hanno lo stesso SimHash sono molto simili, e la somiglianza diventa sempre meno significativa se si permettono distanze di Hamming superiori. Si noti che è banale *pesare* i segnali in modo che alcuni siano più importanti di altri.

Trovare elementi a breve distanza di Hamming è un problema interessante una cui soluzione pratica per distanze piccole è descritta in [MJDS07].

2.3 Gestione della politeness

Uno dei problemi pratici che rende l'attività di crawling diversa da una semplice visita è la gestione della *gentilezza* (*politeness*): non si dovrebbe eccedere nella quantità di tempo dedicato allo scaricamento da un singolo sito (pena, in genere, email furiose o taglio del traffico dal vostro IP).

Ci sono due modi fondamentali di operare questa limitazione:

1. limitare il tempo tra una richiesta e l'altra;
2. limitare la frazione del tempo di scaricamento rispetto al tempo di non-scaricamento.

Nel primo caso, dato un intervallo di tempo t (diciamo, quattro secondi) dobbiamo aspettare t tra la fine di una richiesta e l'inizio della successiva per lo stesso sito. Nel secondo caso, data una frazione p e un tempo di scaricamento massimo s (diciamo, un secondo) dobbiamo fare in modo che la proporzione tra il tempo di scaricamento e quello di non-scaricamento sia p. Si noti che questa condizione contempla anche una misurazione effettiva del tempo di scaricamento, dato che risorse particolarmente lente potrebbero richiedere un tempo maggiore di s.

La seconda soluzione è più interessante, perché permette di sfruttare una caratteristica della versione 1.1 del protocollo HTTP: è possibile cioè effettuare richieste multiple attraverso la stessa connessione TCP, evitando la (lenta) apertura e chiusura di una connessione per ogni risorsa scaricata. Le attività di scaricamento terminano non appena si supera la soglia s, con tempo di scaricamento effettivo s', e a questo punto si aspetta per tempo s'/p, in maniera da forzare la gentilezza.

Per implementare questo tipo di politica, però, è necessario alterare l'ordine di visita, dato che visitando gli URL nell'ordine in cui escono dal crivello si potrebbe incorrere in attese a vuoto consistenti.

2.4 La coda degli host

Un altro problema finora lasciato in parte è il ruolo della concorrenza. Certamente vorremo scaricare contemporaneamente da più siti: per farlo, possiamo istanziare molti flussi (*thread*) di esecuzione (nell'ordine delle migliaia) che si occupano di scaricare i dati, e saranno quindi sempre occupati in attività di I/O. Le pagine scaricate possono essere poi analizzate da un gruppo di flussi più ridotto (non è una buona idea avere migliaia di thread con un carico computazionale significativo). Si noti che, al di là delle questioni di gentilezza, non possiamo permetterci che due flussi accedano allo stesso sito.

Questi problemi vengono risolti riorganizzando gli URL che escono dal crivello. Consideriamo una *coda con priorità* contenente i siti noti al crawler. A ogni sito assegnamo come priorità il

primo istante di tempo in cui è possibile scaricare dal sito senza violare le politiche di gentilezza. La coda restituisce gli elementi in ordine *inverso* (cioè in cima alla coda c'è il minimo).

Per ogni sito manteniamo una coda (FIFO, nel caso di una visita in ampiezza). Quando degli URL vengono emessi dal crivello, questi vengono accodati alla coda associata al loro sito.

Ogni flusso del crawler ora procede iterativamente nel seguente modo: estrae il sito in cima alla coda (eventualmente aspettando il tempo necessario a far sì che la cima sia scaricabile), procede a scaricare una o più risorse, e infine riaccoda il sito modificando la sua prorità in maniera adeguata alla politica di gentilezza (per esempio, impostando la priorità all'istante di tempo corrente più un intervallo di tempo prefissato).

Se c'è un URL disponibile per lo scaricamento, il sito associato deve essere stato pronto per lo scaricamento prima del tempo corrente. Quindi o è in cima alla coda, o in cima alla coda c'è un sito che era pronto per lo scaricamento ancora prima. In ogni caso, la cima della coda è scaricabile. È quindi possibile scaricare un URL *se e solo se* la cima della coda è scaricabile.

Questo meccanismo rende automatica l'esclusività del download tra flussi: gli elementi della coda agiscono come token che rappresentano l'autorizzazione a scaricare da un certo sito. Dato che solo un flusso può avere in un certo istante di tempo il token associato a un certo sito, le regole di gentilezza non possono essere violate.

Il costo della coda è logaritmico: estrarre e reinserire un sito è quindi un'operazione relativamente poco costosa, che però può diventare problematica nel caso di concorrenza intensa.

Infine, nel caso sia necessaria una politica di gentilezza da applicare agli indirizzi IP possiamo organizzare gli IP in una coda come sopra: ogni IP ha una coda con priorità di host associata. La priorità di un IP è il massimo tra il proprio istante di tempo e quello del sito in cima alla coda associata. In questo modo, possiamo visitare un sito solo se è arrivato il momento da scaricare sia dal sito stesso, che dal suo indirizzo IP.

2.5 Tecniche di programmazione concorrente lock-free

Nella gestione delle strutture dati che abbiamo discusso è possibile che l'elevata concorrenza renda le strutture stesse poco efficienti. Se il numero di host è molto grande, per esempio, i flussi di scaricamento rischiano di restare in coda per molto tempo sul semaforo che controlla la coda degli host.

È possibile alleviare il conflitto tra i flussi utilizzando delle tecniche di programmazione nonbloccanti, dette *lock-free*. Una struttura dati lock-free non utilizza semafori: utilizza invece istruzioni hardware che permettono implementazioni efficienti. Quella che useremo è la CAS (*compare-and-swap*): dato un indirizzo p e due valori a, b, la CAS controlla che il valore memorizzato in p sia a e in questo caso lo sostituisce atomicamente con b, restituendo un valore vero se la sostituzione è avvenuta.

Per dare un'idea delle tecniche lock-free, l'algoritmo 1 mostra come aggiungere concorrentemente un nodo a una lista con puntatori; l'implementazione è dovuta a Harris [Har01]. La correttezza dell'algoritmo è banale, ma si noti anche che se l'istruzione CAS fallisce, un altro flusso ha effettuato un inserimento: quindi, per ogni ciclo eseguito da uno dei flussi concorrenti c'è stato *progresso* nell'aggiornamento della struttura. Detto altrimenti: non è possibile dire quante volte uno specifico flusso debba eseguire il ciclo prima di avere successo nell'inserimento, ma a ogni fallimento corrisponde un successo di un altro flusso. L'algoritmo per la cancellazione

Algorithm 1 Algoritmo lock-free per aggiungere un nodo (n) a una lista (dopo p).

```
0  do
1     t ← p.next
2     n.next ← t
3  while ! CAS(&p.next, t, n)
```

è più complesso: l'idea di utilizzare la stessa tecnica non funziona perché è possibile cancellare un nodo mentre al nodo sta venendo aggiunto un nodo successivo, cancellando così l'effetto dell'inserimento.

Si noti che non è richiesta nessuna sincronizzazione in lettura: la lista non è mai in uno stato incoerente, per cui può essere scandita tramite lo stesso algoritmo utilizzato per una normale lista collegata. Inoltre, in pratica, per evitare che in condizioni di elevata concorrenza i conflitti tra le istruzioni CAS diventino troppo costosi si utilizza una tecnica di *arretramento esponenziale* (*exponential backoff*): il ciclo viene ripetuto con un ritardo che cresce esponenzialmente.

Molti linguaggi moderni offrono strutture lock-free pronte per l'uso, come la `ConcurrentLinkedQueue` di Java, che implementa l'algoritmo di Michael e Scott [MS96].

3 Tecniche di distribuzione del carico

Per coordinare un insieme A di agenti indipendenti che effettuano attività di crawling è necessario assegnare in qualche modo ciascun URL a uno specifico agente: denoteremo con $\delta_A(-) : U \to A$ la funzione che dato l'insieme di agenti A assegna a ciascun URL l'agente che ne è responsabile. Assumiamo qui che gli agenti in A siano identici (in particolare, che abbiano a disposizione le stesse risorse).

Una richiesta banale ma necessaria è che la funzione sia *bilanciata*, cioè che

$$\left|\delta_A^{-1}(a)\right| \approx \frac{|U|}{|A|}.$$

Nel caso l'insieme degli agenti sia statico, basta ad esempio numerare gli agenti a partire da zero, fissare una funzione di hash h applicabile agli URL, e dato un URL u assegnargli l'agente $h(u) \bmod |A|$.

Un caso più interessante è quello in cui l'insieme degli agenti può variare nel tempo. In questo frangente non solo vogliamo una proprietà di bilanciamento, ma chiediamo che sia vera la properità di *controvarianza*

$$\delta_B^{-1}(a) \subseteq \delta_A^{-1}(a) \text{ se } B \supseteq A, a \in A.$$

Questa proprietà dice che aumentando l'insieme degli agenti, gli insiemi di URL associati agli agenti preesistenti si riducono. In particolare, se si aggiunge un agente gli agenti preesistenti non si vedono assegnare nessun nuovo URL.

Questo problema è comune ad altri contesti, come il caching nelle *content delivery network* e i sistemi di calcolo distribuito *peer to peer*. È facile convincersi che una strategia basata su hashing e modulo è disastrosa da questo punto di vista. Vedremo ora tre diversi modi di implementare quest'idea.

3.1 Permutazioni aleatorie

Il primo approccio per realizzare una funzione bilanciata e controvariante è assumere che ci sia un universo P di possibili agenti. A ogni istante dato l'insieme di agenti effettivamente utili è un insieme $A \subseteq P$. Per semplicità, assumiamo che P sia formato dai primi $|P|$ numeri naturali, ma in realtà è solo necessario che P sia totalmente ordinato.

Fissiamo una volta per tutte un generatore di numeri pseudocasuali. La strategia ora è la seguente: dato un URL u, inizializziamo il generatore utilizzando u (per esempio, passando u attraverso una funzione di hash prefissata) e utilizziamolo per generare una permutazione aleatoria di P scelta in maniera uniforme. A questo punto, l'agente scelto è il primo agente in A nell'ordine indotto dalla permutazione così calcolata.

Chiaramente, dato che le permutazioni vengono generate in maniera uniforme approssimativamente la stessa frazione di URL viene assegnata a ogni agente. Inoltre, se consideriamo un insieme $B \supseteq A$ le uniche variazioni di assegnazione consistono nello spostamento di URL verso elementi di $B \setminus A$, spostamento che avviene esattamente quando uno di tali elementi precede tutti quelli di A nella permutazione associata all'URL.

In sostanza, la permutazione fornisce un ordine di preferenza per assegnare u all'interno dell'insieme degli agenti possibili P. Il primo agente effettivamente disponibile (cioè, in A) sarà responsabile per il crawling di u. Dato che la permutazione è generata tramite un generatore comune a tutti gli agenti e utilizza lo stesso seme (u) tutti gli agenti calcolano lo stesso ordine di preferenza.

Si noti è che possibile generare una tale permutazione in tempo e spazio lineare in $|P|$. La tecnica, nota come *Knuth shuffle* o *Fisher–Yates shuffle*, consiste nell'inizializzare un vettore di $|P|$ elementi con i primi $|P|$ numeri naturali (o con gli elementi di P, nel caso generale).

A questo punto si eseguono $|P|$ iterazioni: all'iterazione di indice i (a partire da zero) si scambia l'elemento del vettore di posto i con uno dei seguenti $|P| - i$ scelto a caso uniformemente (si noti che i stesso è una scelta possibile). Alla fine dell'algoritmo, il vettore contiene una permutazione aleatoria scelta in maniera uniforme.

In realtà non è necessario generare l'intera permutazione: non appena completiamo l'iterazione i, se troviamo in posizione i un elemento di A possiamo restituirlo, dato che nei passi successivi non verrà più spostato. Inoltre, le modifiche ad A avvengono in tempo costante (dato che non c'è nulla da fare).

3.2 Min hashing

Il secondo approccio non richiede che ci sia un universo predeterminato o ordinato. Fissiamo una funzione di hash aleatoria $h(-,-)$ a due argomenti che prende un URL e un agente. L'agente all'interno di A responsabile dell'URL u è quello che realizza il minimo tra tutti i valori $h(u, a)$, $a \in A$.

Di nuovo, assumendo che h sia aleatoria il bilanciamento è banale. Ma anche la proprietà di controvarianza è elementare: se $B \supseteq A$, gli unici URL che cambiano assegnamento sono quegli URL u per cui esiste un $b \in B \setminus A$ tale che $h(b, u) < h(a, u)$ per ogni $a \in A$.

Si noti che questo metodo richiede tempo di calcolo proporzionale ad A, e spazio costante (basta tenere traccia del minimo elemento trovato). Anche in questo caso, le modifiche ad A richiedono tempo costante dato che non c'è nulla da fare.

3.3 Hashing coerente

L'ultimo approccio è il più sofisticato. Consideriamo idealmente una circonferenza di lunghezza unitaria e una funzione di hash aleatoria h che mappa gli URL nell'intervallo $[0\,..\,1)$ (cioè sulla circonferenza). Fissiamo inoltre un generatore di numeri pseudocasuali.

Per ogni agente $a \in A$, utilizziamo a per inizializzare il generatore e scegliere C posizioni pseudoaleatorie sul cerchio (as esempio, in pratica, $C = 300$): queste saranno le *repliche* dell'agente a. A questo punto per trovare l'agente responsabile di un URL u partiamo dalla posizione $h(u)$ e proseguiamo in senso orario finché non troviamo una replica: l'agente associato è quello responsabile per u.

In pratica, il cerchio viene diviso in segmenti massimali che non contengono repliche. Associamo a ogni segmento la replica successiva in senso orario, e tutti gli URL mappati sul segmento avranno come agente responsabile quello associato alla replica. La funzione così definita

è ovviamente controvariante. L'effetto di aggiungere un nuovo agente è semplicemente quello di spezzare tramite le nuove repliche i segmenti preesistenti: la prima parte verrà mappata sul nuovo agente, mentre la seconda continuerà a essere mappata sull'agente precedente.

La parte delicata è in questo caso il bilanciamento: se C viene scelto sufficientemente grande (rispetto al numero di agenti possibili; si veda [KLL+97]) i segmenti risultano così piccoli da poter dimostrare che la funzione è bilanciata con alta probabilità.

Si noti che la struttura può essere realizzata in maniera interamente discreta memorizzando interi che rappresentano (essenzialmente, in virgola fissa) le posizioni delle repliche in un dizionario ordinato (per esempio, un albero binario bilanciato). A questo punto dato un URL u è sufficiente generare uno hash intero e trovare il minimo maggiorante presente nel dizionario (eventualmente debordando alla fine dell'insieme e restituendo quindi il primo elemento).

Questo metodo richiede quindi spazio lineare in $|A|$ e tempo logaritmico in $|A|$. Aggiungere o togliere un elemento ad A richiede, contrariamente ai casi precedenti, tempo logaritmico in $|A|$.

3.4 Note generali

In tutti i metodi che abbiamo delineato c'è una componente pseudoaleatoria. Questo fa sì che la distribuzione degli URL agli agenti non sia veramente bilanciata, ma segua piuttosto una distribuzione binomiale negativa. Considerato però il grande numero di elementi in giorno, il risultato approssima bene una distribuzione bilanciata.

Sia il min hashing che lo hashing coerente possono presentare *collisioni* — situazioni in cui non sappiamo come scegliere l'output perché due minimi, o due repliche, coincidono: in tal caso, è necessario disambiguare deterministicamente il risultato (per esempio, imponendo un ordine arbitrario sugli agenti, e restituendo il minimo).

Tutti i metodo permettono (eventualmente con uno sforzo computazionale aggiuntivo) di sapere quale sarebbe il *prossimo* agente responsabile per un URL se quello corrente non fosse presente. Nel caso delle permutazioni aleatorie è sufficiente cercare l'elemento successivo in A nella permutazioni (prolungando di quanto serve lo shuffle). Nel caso del min hashing è necessario tenere traccia di *due valori*: il minimo e il valore immediatamente successivo. Infine, nel caso dell'hashing coerente si continua a procedere in senso orario fino a giungere alla replica di un nuovo agente.

4 Codici istantanei

Un *codice* è un insieme $C \subseteq 2^*$, cioè un insieme di parole binarie. Si noti che per ovvie ragioni di cardinalità C è al più numerabile.

Definiamo l'*ordinamento per prefissi* delle sequenze in 2^* come segue:

$$x \preceq y \iff \exists z \quad y = xz.$$

Vale a dire, $x \preceq y$ se e solo se un prefisso di y coincide con x. Ricordiamo che in un ordine parziale due element x e y sono *inconfrontabili* se nessuno dei due è minore o uguale all'altro.

Un codice è detto *istantaneo* o *a decodifica istantanea* o *privo di prefissi* (*prefix code*) se ogni coppia di parole distinte del codice è inconfrontabile. L'effetto pratico di questa proprietà è che a fronte di una parola w formata da una concatenazione di parole del codice non esiste una diversa concatenazione che dà w. In particolare, leggendo uno a uno i bit di w è possibile ottenere in maniera istantanea le parole del codice che lo compongono.

Ad esempio, il codice $\{0, 1\}$ è istantaneo, ma il codice $\{0, 01, 1\}$ non lo è. Di fronte alla stringa 001, nel primo caso sappiamo che è formata da 0, 0 e 1, ma nel secondo non possiamo decidere tra 0 seguito da 01 o 0, 0 e 1.

Un codice è *completo* o *non ridondante* se per ogni parola $w \in 2^*$ è confrontabile con qualche parola del codice (cioè esiste una parola del codice di cui w è prefisso o una parola del codice che è un prefisso di w). Il primo dei due codici summenzionato è completo, mentre il secondo no.

Quando un codice istantaneo è completo, non è possibile più aggiungere parole al codice senza violare l'istantaneità; inoltre, qualunque parola *infinita* è decomponibile in maniera unica come sequenza di parole del codice, e qualunque parola *finita* è decomponibile in maniera unica come sequenza di parole del codice più un prefisso di qualche parola del codice.

Dimostriamo ora la fondamentale *disuguaglianza di Kraft–McMillan*:

Teorema 1 Sia $C \subseteq 2^*$ un codice. Se C è istantaneo, allora

$$\sum_{w \in C} 2^{-|w|} \leq 1$$

e C è completo se e solo se vale l'uguaglianza. Inoltre, data una sequenza (eventualmente infinita) $t_0, t_1, \ldots, t_{n-1}, \ldots$ che soddisfa

$$\sum_n 2^{-t_n} \leq 1$$

esiste un codice istantaneo formato da parole $w_0, w_1, \ldots, w_{n-1}, \ldots$ tali che $|w_i| = t_i$.

Dimostrazione. Un *diadico* è un razionale della forma $k2^{-h}$. A ogni parola $w \in 2^*$ possiamo associare un sottointervallo semiaperto di $[0\,..\,1)$ con estremi diadici come segue:

- se w è la parola vuota, l'intervallo è $[0\,..\,1)$;

- altrimenti, se $[x\,..\,y)$ è l'intervallo associato a w privato dell'ultimo simbolo l'intervallo associato a w è $[x\,..\,(x+y)/2)$ se l'ultimo simbolo era uno zero, $[(x+y)/2\,..\,y)$ altrimenti.[7]

Si noti che:

- l'intervallo associato a una parola di lunghezza n ha lunghezza 2^{-n};
- se $v \preceq w$ l'intervallo associato a v contiene quello associato a w;
- due parole sono inconfrontabili se e solo se i corrispondenti intervalli sono disgiunti; infatti, se v e w sono inconfrontabili e z è il loro massimo prefisso comune, assumendo senza perdita di generalità che $z0 \preceq v$ e $z1 \preceq w$ l'intervallo di v sarà contenuto in quello di $z0$ e l'intervallo di w in quello di $z1$: dato che gli intervalli di $z0$ e $z1$ sono disgiunti per definizione, lo sono anche quelli di v e w;
- dato un qualunque intervallo diadico $[k2^{-h}\,..\,(k+1)2^{-h})$ esiste un'unica parola di lunghezza h a cui è associato l'intervallo, vale a dire, la parola formata dalla rappresentazione binaria di k allineata ad h bit; questo è certamente vero per $h = 0$, e data una parola w con $|w| = h+1$ se l'intervallo associato a w privato dell'ultimo carattere è $[k2^{-h}\,..\,(k+1)2^{-h})$ l'intervallo associato a w è $[(2k)2^{-h-1}\,..\,(2k+1)2^{-h-1})$ se l'ultimo carattere di w è 0, $[(2k+1)2^{-h-1}\,..\,2(k+1)2^{-h-1})$ altrimenti.

Sia ora C un codice istantaneo. La sommatoria contenuta nell'enunciato del teorema è la somma delle lunghezze degli intervalli associati alle parole di C; questi intervalli sono disgiunti e la loro unione forma un sottoinsieme di $[0\,..\,1)$ che ha necessariamente lunghezza minore o uguale 1.

Se la sommatoria è strettamente minore di uno, deve esserci un intervallo scoperto, diciamo $[x\,..\,y)$. Questo intervallo contiene necessariamente un sottointervallo della forma $[k2^{-h}\,..\,(k+1)2^{-h})$ per qualche h e k. Ma allora la parola associata a quest'ultimo potrebbe essere aggiunta al codice (essendo inconfrontabile con tutte le altre), che quindi risulta incompleto. D'altra parte, se il codice è incompleto l'intervallo corrispondente a una parola inconfrontabile con tutte quelle del codice è necessariamente scoperto, e rende la somma strettamente minore di 1.

Andiamo ora a dimostrare l'ultima parte dell'enunciato, assumendo senza perdita di generalità che la sequenza $t_0, t_1, \ldots, t_{n-1}, \ldots$ sia monotona non decrescente.

Genereremo le parole $w_0, w_1, \ldots, w_{n-1}, \ldots$ in maniera miope (*greedy*). Sia d l'estremo destro della parte di intervallo unitario correntemente coperta dagli intervalli associati alle parole già generate; inizialmente, $d = 0$. Manterremo vero l'invariante che prima dell'emissione della

[7] Un modo semplice di rappresentare intuitivamente gli intervalli associati alle parole è quello di inserire le parole del codice in un albero in cui ogni nodo ha al più due figli etichettati da 0 e 1. A questo punto le foglie dell'albero corrispondono alle parole (dato che il codice è istantaneo), e l'estremo sinistro dell'intervallo associato a una foglia è dato da $k2^{-h}$, dove h è la profondità della foglia e k è il numero descritto dalle etichette attraversate per arrivare alla foglia.

parola w_n si ha $d = k2^{-t_n}$ per qualche k, il che ci permetterà di scegliere come w_n l'unica parola di lunghezza t_n il cui intervallo ha estremo sinistro d. Dato che l'intervallo associato a ogni nuova parola è disgiunto dall'unione dei precedenti, le parole generate saranno tutte inconfrontabili.

L'invariante è ovviamente vero quando $n = 0$. Dopo aver generato w_n, d viene aggiornato sommandogli 2^{-t_n}, e diventa quindi $(k+1)2^{-t_n}$. Ma dato che $(k+1)2^{-t_n} = \left((k+1)2^{t_{n+1}-t_n}\right)2^{-t_{n+1}}$ e $t_{n+1} \geq t_n$, l'invariante viene mantenuto. ∎

4.1 Codici istantanei per gli interi

Alcuni dei metodi più utilizzati di compressione degli indici utilizzano codici istantanei per gli interi. Questa scelta può apparire a prima vista opinabile, dato che tutti i numeri che compaiono in un indice hanno delle limitazioni superiori naturali e facili da calcolare, e quindi potrebbe essere più efficiente calcolare un codice istantaneo per il solo sottoinsieme di interi effettivamente utilizzato.

In realtà, se si è interessati a collezioni documentali di grandi dimensioni la semplicità teorica e implementativa dei codici per gli interi li rende molto interessanti.

Innanzitutto si noti che un codice istantaneo per gli interi è numerabile. L'associazione tra interi e parole del codice va specificata di volta in volta, anche se in tutti i codici che vedremo l'associazione è semplicemente data dall'ordinamento prima per lunghezza e poi lessicografico delle parole. Inoltre, assumeremo che le parole rappresentino numeri naturali, e quindi la parola minima (cioè lessicograficamente minima tra quelle di lunghezza minima) rappresenti lo zero.[8]

La rappresentazione più elementare di un intero n è quella *binaria*, che però non è istantanea (le prime parole sono 0, 1, 10, 11, 100). È possibile ovviamente rendere istantanea la rappresentazione dei primi 2^k interi allineando a k bit le rappresentazioni binarie.

Chiameremo *rappresentazione binaria ridotta* di x la rappresentazione binaria di $x + 1$ privata del bit più significativo; anch'essa non è istantanea. La lunghezza della parola di codice per x è quindi $\lambda(x + 1)$. Le prime parole sono ε, 0, 1, 00, 01, 10.

Un ruolo importante nella costruzione di codici istantenei è svolto dai *codici binari minimali* — codici istantanei e completi per i primi k interi che utilizzano un numero variabile di bit. Esistono diverse possibilità per le scelte delle parole del codice[9], ma in quanto segue diremo che il codice binario minimale di x (nei primi k interi) è definito come segue: sia $s = \lceil \log k \rceil$; se $x < 2^s - k$, allora x è codificato dalla x-esima parola binaria (in ordine lessicografico) di lunghezza $s - 1$; altrimenti, x è codificato utilizzando la $(x - k + 2^s)$-esima parola binaria di lunghezza s.

[8] Questa scelta non è uniforme in letteratura, e in effetti si possono trovare nello stesso libro due codici per gli interi che, a seconda della bisogna, vengono numerati a partire da zero o da uno.

[9] In realtà, un codice binario minimale è semplicemente un codice ottimo per la distribuzione uniforme, il che spiega perché sono possibili scelte diverse per le parole del codice.

	k						
	1	2	3	4	5	6	7
0	ε	0	0	00	00	00	00
1		1	10	01	01	01	010
2			11	10	10	100	011
3				11	110	101	100
4					111	110	101
5						111	110
6							111

Tabella 1: Esempi di codici binari minimali.

La base di tutti codici istantanei per gli interi è il codice *unario*. Il codice unario rappresenta il naturale x tramite x zeri seguiti da un uno.[10] La lunghezza della parola di codice per x è quindi $x+1$. Il codice è banalmente istantaneo e completo. Le prime parole del codice sono 1, 01, 001, 0001.

Il codice γ [Eli75] codifica un intero x scrivendo il numero di bit della rappresentazione binaria ridotta di x in unario, seguito dalla rappresentazione binaria ridotta di x. La lunghezza della parola di codice per x è quindi $2\lambda(x+1)+1$. Il fatto che il codice sia istantaneo e completo si ottiene immediatamente dal fatto che lo è il codice unario. Le prime parole del codice sono 1, 010, 011, 00100, 00101, 00110, 00111, 0001000.

Analogamente, il codice δ [Eli75] codifica un intero x scrivendo il numero di bit della rappresentazione binaria ridotta di x in γ, seguito dalla rappresentazione binaria ridotta x. La lunghezza della parola di codice per x è quindi $2\lambda(\lambda(x+1)+1)+1+\lambda(x+1)$. Il fatto che il codice sia istantaneo e completo si ottiene immediatamente dal fatto che lo è il codice γ. Le prime parole del codice sono 1, 0100, 0101, 01100, 01101, 01110, 01111, 00100000, 00100001.

È possibile ovviamente continuare in questa direzione, ma, come vedremo, senza vantaggi significativi.

Il *codice di Golomb di modulo b* [Gol80] codifica un intero x scrivendo il quoziente della divisione di x per b in unario, seguito dal resto in binario minimale. La lunghezza della parola di codice per x è quindi $1+\lfloor x/b \rfloor + \lambda(b) + [x \geq 2^{\lceil \log b \rceil} - b]$. Il fatto che il codice sia istantaneo e completo si ottiene immediatamente dal fatto che lo sono il codice unario e il binario minimale. Per $b = 3$ le prime parole del codice sono 10, 110, 111, 010, 0110, 0111.

Infine, conviene ricordare i *codici a blocchi a lunghezza variabile (variable-length block codes)*, come il codice variabile a nibble o a byte. L'idea è che ogni parola è formata da un numero variabile di blocchi di k bit (4 per il nibble, 8 per il byte). Il primo bit, detto di *continuazione*,

[10] È possibile trovare in letteratura la scelta inversa, che ha il vantaggio di fare coincidere l'ordine lessicografico delle parole con l'ordine naturale dei valori rappresentati.

non fa parte dell'intero codificato, e serve a specificare (se diverso da zero) che il codice non termina in quel blocco. Un intero x viene scritto in notazione binaria, allineato a un multiplo to $k-1$ bit, diviso in blocchi di $k-1$ bit, e rappresentato tramite la sequenza dei suddetti blocchi, ciascuno preceduto da un bit di continuazione uguale 1, tranne nell'ultimo blocco. La lunghezza della parola di codice per x è quindi $\lceil(\log x+1)/k\rceil(k+1)$. I codici a lunghezza variabile sono ovviamente istantanei ma non completi, dato che, ad esempio, una lista di blocchi lunghezza maggiore di uno che contiene solo a bit a zero (a parte quelli di continuazione) è inconfrontabile con tutte le parole del codice. È possible rendere il codice più semplice da predire in fase di decodifica spostando tutti i bit di continuazione all'inizio della parola di codice. In particolare, spesso il primo byte di un codice a byte a lunghezza variabile conterrà il numero esatto di ulteriori byte da leggere.[11]

	γ	δ	Golomb ($b=3$)	nibble
0	1	1	10	1000
1	010	0100	110	1001
2	011	0101	111	1010
3	00100	01100	010	1011
4	00101	01101	0110	1100
5	00110	01110	0111	1101
6	00111	01111	0010	1110
7	0001000	00100000	00110	1111
8	0001001	00100001	00111	00011000
9	0001010	00100010	00010	00011001
10	0001011	00100011	000110	00011010
11	0001100	00100100	000111	00011011
12	0001101	00100101	000010	00011100
13	0001110	00100110	0000110	00011101
14	0001111	00100111	0000111	00011110
15	000010000	001011001	0000010	00011111

Tabella 2: Esempi di codici.

4.1.1 Caratteristiche matematiche dei codici

Esistono delle caratteristiche intrinseche dei codici istantanei per gli interi che permettono di classificarli e distinguerne il comportamento. In particolare, un codice è *universale* se per qualunque distribuzione p sugli interi monotona non crescente (cioè $p(i) \geq p(i+1)$) il valore atteso

[11] Questa tecnica è utilizzata dalla codifica UTF-8.

della lunghezza di una parola rispetto a p è minore o uguale all'entropia di p a meno di costanti additive e moltiplicative (indipendenti da p). Vale a dire, se $\ell(x)$ è la lunghezza della parola di codice per x e $H(p)$ denota l'entropia (nel senso di Shannon) di una distribuzione p esistono costanti c e d tali che per ogni distribuzione p monotona non crescente si ha

$$\sum_{x \in \mathbf{N}} \ell(x) p(x) \leq c H(p) + d.$$

L'idea compare nel lavoro di Levenshtein dove viene introdotto il primo codice universale. Il codice è detto *asintoticamente ottimo* quando a destra il limite superiore è della forma $f(H(p))$ con $\lim_{x \to \infty} f(x) = 1$.

Il codice unario e i codici di Golomb non sono universali, mentre lo sono γ e δ. Inoltre, δ è asintoticamente ottimo, mentre γ non lo è.

4.1.2 Codifiche alternative

È possibile utilizzare tecniche standard quali la codifica di Huffmann o la codifica aritmetica per la codifica di ogni parte di un indice. Ci sono però delle considerazioni ovvie che mostrano come questi metodi, utilizzati direttamente, non siano di fatto implementabili. La codifica di Huffman richiederebbe un numero di parole di codice esorbitante, e le parole dovrebbero essere calcolate separatamente per ogni termine. La codifica aritmetica richiede alcuni bit di scarico prima di poter essere interrotta, e quindi non si presta ad essere inframmezzata da altri dati (come i conteggi e le posizioni). Inoltre, per quanto efficientemente implementata, è estremamente lenta.

PFOR-DELTA. Un approccio che ha rivoluzionato il modo di memorizzare le liste di affissioni è quello proposto in [ZHNB06], comunemente chiamato PFOR-DELTA o FOR (*Frame Of Reference*). L'idea è molto semplice: si sceglie una dimensione di blocco B (di solito, $B = 128$ o $B = 256$), e per ogni blocco consecutivo di B scarti si trova l'intero b tale che il 90% degli scarti sono esprimibili in 2^b bit.

A questo punto viene scritto un vettore di B interi di b bit, che descrive correttamente il 90% degli scarti. Il rimanente 10% degli scarti, le *eccezioni*, viene scritto separatamente in un vettore di interi (di lunghezza sufficiente a descrivere il massimo scarto). Le posizioni del primo vettore corrispondenti alle eccezioni vengono riempite con un intero che rappresenta la distanza dalla prossima eccezione (può essere necessario aggiungere delle eccezioni fittizie per far sì che la distanza dalla prossima eccezione sia sempre esprimibile in b bit).

In fase di decodifica, si copiano i primi B valori a b bit in un vettore di interi. Quest'operazione è molto veloce, in particolare se vengono creati cicli srotolati diversi[12] per ogni possibile valore

[12] Sono migliaia di righe di codice, ma possono venire generate automaticamente.

di b. A questo punto si passa attraverso la lista delle eccezioni, che vengono copiate dal secondo vettore nelle posizioni corrette.

Questo approccio fa sì che il processore esegua dei cicli estremamente predicibili, e riesce a decodificare un numero di scarti al secondo significativamente più alto delle tecniche basate su codici. La performance di compressione è di solito buona, anche se è difficile dare garanzie teoriche. Lo svantaggio (che può risultare significativo) è che è sempre necessario decodificare B elementi.

Sistemi di numerazione asimmetrica Nel 2013, Duda [Dud13] ha introdotto un nuovo metodo di codifica di sequenze di simboli che ha rivoluzionato il campo della compressione dati. I *sistemi di numerazione asimmetrica* hanno velocità confrontabile o superiore a un codice di Huffman, ma compression confrontabile con la codifica aritmetica, e quindi ottima (vicina all'entropia di Shannon). Sono alla base di tutti i più moderni sistemi di compressione, come `zstd` di Facebook o lo standard JPEG XL.

L'idea alla base dei sistemi di numerazione asimmetrica è che per comprimere una sequenza di simboli vorremmo idealmente utilizzare un intero molto grande. Se una sequenza è codificata da x, vorremmo che aggiungere un simbolo s che ha probabilità di comparire p_s portasse a una codifica $x' \approx x/p_s$, perché in questo modo $\log(x') = \log x + \log \frac{1}{p_s}$. Aggiungendo cioè s alla sequenza si spendono $\log \frac{1}{p_s}$ bit. Complessivamente, il costo del messaggio sarà l'entropia di Shannon.

Possiamo vedere questo meccanismo nella codifica banale della sorgente più semplice, con zero e uno equiprobabili. Se x è la codifica dei simboli visti sinora, $x' = 2x$ o $x' = 2x + 1$ a seconda del simbolo che compare. Il risultato è un intero che in base 2 è descritto dalla sequenza di zeri e uni da codificare.

Se vogliamo decodificare la sequenza x', l'ultimo simbolo codificato è semplicemente $x' \bmod 2$, e la sequenza rimanente $\lfloor x/2 \rfloor$. Iterando la procedura otteniamo tutte le cifre *in ordine inverso rispetto alla codifica*.

Questa è una caratteristica comune a tutti i sistemi di numerazione asimmetrica: si codifica in una direzione, si decodifica in direzione opposta, come in una pila. In fase di compressione il problema è risolto comprimendo a blocchi a rovescio, in modo che in fase di decompressione l'output sia già nella direzione corretta.

È chiaro che il meccanismo descritto per il caso di due simboli funziona allo stesso modo per k simboli equiprobabili: staremo semplicemente codificando ogni sequenza di simboli tramite l'intero rappresentato dai simboli stessi in base k.

Vediamo ora come generalizzare quest'idea a un numero arbitrario di simboli con distribuzione arbitraria. Definiremo un'operazione *push* che prende l'intero che rappresenta una sequenza, un nuovo simbolo, e restituisce la rappresentazione della sequenza col simbolo aggiunto, e un'ope-

razione *pop* che prende un intero che rappresenta una sequenza e restituisce l'ultimo simbolo, e la rappresentazione della sequenza privata dell'ultimo simbolo.

Sia $\Sigma = [0\,..\,k)$ l'insieme dei simboli e per ogni $s \in \Sigma$ sia p_s la sua probabilità. Innanzitutto fissiamo una *precisione* d: tutte le probabilità verranno approssimate da qui in poi tramite multipli di $1/2^d$: quindi per ogni s abbiamo $p_s = f_s/2^d$. Definiamo $f_n = 2^d$. Più grande è d, maggiore è la precisione con cui rappresentiamo le probabilità iniziali (e quindi migliore la compressione).

Prendiamo ora i primi 2^d interi e dividiamoli in n segmenti contigui di lunghezza f_s. A simboli più frequenti corrisponderanno quindi segmenti più lunghi. Definiamo la cumulativa degli f_s

$$c_s = \sum_{t<s} f_t \text{ per } s \in [0..n].$$

Notiamo ora che ogni elemento x di 2^d sta in uno dei segmenti, e quindi ha associato un simbolo s. Più precisamente, esiste esattamente un s tale che $c_s \le x < c_{s+1}$, e poniamo $\mathrm{sym}(x) = s$. Inoltre, $\mathrm{sym}\big(f_s\big) = s$. In questo modo, se prendiamo un elemento x di 2^d uniformemente a caso, $\mathrm{sym}(x) \in \Sigma$ ha esattamente distribuzione p.

Procediamo ora a definire le operazioni:

$$\begin{aligned}\mathrm{push}(x, s) &= \big(\lfloor x/f_s \rfloor \ll d\big) + c_s + x \bmod f_s\\ \mathrm{pop}(x) &= \langle (x \gg d) \cdot f_s + x \bmod 2^d - c_s, s\rangle \text{ dove } s = \mathrm{sym}\big(x \bmod 2^d\big).\end{aligned}$$

È facile vedere che le due operazioni sono una inversa dell'altra. Inoltre, $\mathrm{push}(x, s) \approx (x/f_s) \cdot 2^d = x \cdot \big(f_s/2^d\big) = x/p_s$, come volevamo.

Provamo ora su un semplice esempio in cui abbiamo tre simboli: 0 con probabilità $9/10$, 1 con probabilità $1/30$ e 2 con probabilità $2/30$. Abbiamo quindi $\log \frac{1}{p_0} \approx 0.152$, $\log \frac{1}{p_1} \approx 4.9$ e $\log \frac{1}{p_2} \approx 3.9$, e l'entropia è ≈ 0.558. Utilizzando $d = 10$ abbiamo probabilità $922/1024$, $34/1024$ e $68/1024$. Consideriamo la sequenza

$$0, 0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0$$

Partiamo da $x = 0$. Ora,

$$\begin{aligned}
\text{push}(0,0) &= \big(\lfloor 0/922 \rfloor \ll 10\big) + 0 + 0 = 0 && \text{(1 bit)}\\
\text{push}(0,0) &= \big(\lfloor 0/922 \rfloor \ll 10\big) + 0 + 0 = 0 && \text{(1 bit/2)}\\
\text{push}(0,0) &= \big(\lfloor 0/922 \rfloor \ll 10\big) + 0 + 0 = 0 && \text{(1 bit/3)}\\
\text{push}(0,0) &= \big(\lfloor 0/922 \rfloor \ll 10\big) + 0 + 0 = 0 && \text{(1 bit/4)}\\
\text{push}(0,0) &= \big(\lfloor 0/922 \rfloor \ll 10\big) + 0 + 0 = 0 && \text{(1 bit/5)}\\
\text{push}(0,1) &= \big(\lfloor 0/34 \rfloor \ll 10\big) + 922 + 0 = 922 && \text{(9 bit/6)}\\
\text{push}(922,0) &= \big(\lfloor 922/922 \rfloor \ll 10\big) + 0 + 0 = 1024 && \text{(10 bit/7)}\\
\text{push}(1024,0) &= \big(\lfloor 1024/922 \rfloor \ll 10\big) + 0 + 102 = 1126 && \text{(11 bit/8)}\\
\text{push}(1126,0) &= \big(\lfloor 1126/922 \rfloor \ll 10\big) + 0 + 204 = 1228 && \text{(11 bit/9)}\\
\text{push}(1228,2) &= \big(\lfloor 1228/68 \rfloor \ll 10\big) + 956 + 4 = 19392 && \text{(15 bit/10)}\\
\text{push}(19392,0) &= \big(\lfloor 19392/922 \rfloor \ll 10\big) + 0 + 30 = 21534 && \text{(15 bit/11)}\\
\text{push}(21534,0) &= \big(\lfloor 21534/922 \rfloor \ll 10\big) + 0 + 328 = 23880 && \text{(15 bit/12)}
\end{aligned}$$

Nelle implementazioni pratiche, viene effettuata una *rinormalizzazione*: quando l'intero calcolato diventa troppo grande da maneggiare, la metà superiore dei suoi bit viene scritta in un buffer di output e si continua utilizzando la metà inferiore. La stessa operazione, invertita, consente di rileggere. La rinormalizzazione comporta un'ulteriore piccola perdita di compressione, ma permette di operare solo su interi di dimensione fissa.

4.2 Distribuzioni intese

Ogni codice istantaneo completo per gli interi definisce implicitamente una *distribuzione intesa* sugli interi che assegna a w la probabilità $2^{-|w|}$. Il codice, in effetti, risulta *ottimo* per la distribuzione associata. La scelta di un codice istantaneo può quindi essere ricondotta a considerazioni sulla distribuzione statistica degli interi che si intendono rappresentare.

Al codice unario è associata una distribuzione geometrica

$$\frac{1}{2^{x+1}}$$

mentre la distribuzione associata a γ è

$$\frac{1}{2^{2\lfloor \log(x+1) \rfloor + 1}} \approx \frac{1}{2(x+1)^2}$$

e quella associata a δ è

$$\frac{1}{2^{2\lfloor\log(\lfloor\log(x+1)\rfloor+1)\rfloor+1+\lfloor\log(x+1)\rfloor}} \approx \frac{1}{2(x+1)(\log(x+1)+1)^2}.$$

Il codice di Golomb ha, naturalmente, una distribuzione dipendente dal modulo, che però è geometrica:

$$\frac{1}{2^{\lfloor x/b\rfloor+\lambda(b)+[n\geq 2^{\lceil\log b\rceil}-b]}} \approx \frac{1}{b\left(\sqrt[b]{2}\right)^x}$$

Infine, sebbene i codici a lunghezza variabile non siano completi, a meno di un fattore di normalizzazione è comunque possibile osservarne la distribuzione intesa:

$$\frac{1}{2^{\lceil(\log x)/k\rceil(k+1)}} \approx \frac{1}{(x+1)^{1+\frac{1}{k}}}.$$

4.3 Codici per gli indici

Nella scelta dei codici istantanei da utilizzare per la compressione di un indice è fondamentale la conoscenza della distribuzione degli interi da comprimere. In alcuni casi è possibile creare un modello statistico che spiega la distribuzione empirica riscontrata.

Per quanto riguarda frequenze e conteggi, la presenza significativa degli *hapax legomena* rende il codice γ quello usato più di frequente.

La questione degli scarti tra puntatori a documenti è più complessa. Il *modello Bernouilliano* di distribuzione prevede che una termine con frequenza f compaia in una collezione di N documenti con probabilità $p = f/N$ indipendentemente in ogni documento. L'assunzione di indipendenza ha l'effetto di semplificare enormemente la distribuzione degli scarti, che risulta una geometrica di ragione p: lo scarto $x > 0$ compare cioè con probabilità $p(1-p)^{(x-1)}$.

Abbiamo già notato come il codice di Golomb abbia distribuzione intesa geometrica: si tratta quindi di trovare il modulo adatto a p, e un risultato importante di teoria dei codici dice che il codice ottimo per una geometrica di ragione p è un Golomb di modulo

$$b = \left\lceil -\frac{\log(2-p)}{\log(1-p)} \right\rceil.$$

Una controindicazione può essere però la *correlazione* tra documenti adiacenti. Per esempio, se i documenti nella collezione provengono da un crawl e sono nell'ordine di visita, è molto probabile che documenti vicini contengano termini simili. Questo fatto sposta significativamente da una geometrica la distribuzione degli scarti.

Per quanto riguarda le posizioni, non esistono modelli noti e affidabili degli scarti. Si utilizza quindi un codice dalle buone prestazioni generali come il δ. È anche possibile utilizzare Golomb, assumendo un modello Bernouilliano anche per le posizioni, ma per calcolare il modulo

è necessario avere la lunghezza del documento, che quindi deve essere disponibile in memoria centrale.

Alla fine, in ogni caso, considerazioni come la facilità di implentazione, la velocità di decodifica, etc., possono essere forze determinanti nella scelta delle tecniche di codifica.

4.4 Problemi implementativi

I codici istantanei comprimono in maniera ottima rispetto alle distribuzioni intese. Riducono quindi la dimensione dell'indice in maniera significativa, cosa particolarmente utile se l'indice verrà caricato in tutto o in parte in memoria centrale. In effetti, esprimenti condotto all'inizio dell'attività di ricerca sugli indici inversi hanno mostrato che un indice compresso è significativamente più veloce di un indice non compresso, dato che l'I/O è ridotto in maniera significativa e il prezzo da pagare nella decodifica è di pochi cicli macchina per intero.

L'implementazione della lettura dei codici istantanei va però effettuata con una certa cura se si vogliono ottenere prestazioni ragionevoli. L'idea più importante è quella di mantenere un *bit buffer* che contiene una finestra sul file che contiene le liste di affissioni. Le manipolazioni relative alla decodifica dei codici dovrebbero essere confinate al bit buffer nella stragrande maggioranza dei casi. In particolare, un numero significativo di prefissi (per esempio, 2^{16}) può essere decodificato tramite una tabella che contiene, per ogni prefisso, il numero di bit immediatamente decodificabili e l'intero corrispondente, o l'indicazione che è necessario procedere a una decodifica manuale.

4.5 Salti

Non è possibile ottenere in tempo costante un elemento arbitrario di una lista compressa per scarti: è necessario decodificare gli elementi precedenti. In realtà, più problematica è l'impossibilità di saltare rapidamente al primo elemento della lista maggiore o uguale a un limite inferiore b, operazione detta comunemente "salto" (*skip*). I salti sono fondamentali, come vedremo, per la risoluzione veloce delle interrogazioni congiunte.

La tecnica di base per ovviare a questo inconvieniente è quella di memorizzare, insieme alla lista di affissioni, una *tabella di salto* (*skip table*) che memorizza, dato un *quanto* q, il valore degli elementi di indice iq, $i \geq 0$, e la loro posizione (espressa in bit) nella lista di affissioni. Quando si vuole effettuare un salto, e più precisamente quando si vuole trovare il minimo elemento maggiore o uguale a b, si cerca nella tabella (per esempio tramite una ricerca dicotomica o una *ricerca esponenziale*) il massimo l'elemento di posto iq minore o uguale a b, e si comincia a decodificare la lista per cercare il minimo maggiorante di b. Al più q elementi dovranno essere decodificati, e q deve essere scelto sperimentalmente in modo da al tempo stesso non accrescere eccessivamente la dimensione dell'indice e fornire un significativo aumento di prestazioni.

Si noti che una volta che è possibile saltare all'interno della lista dei puntatori ai documenti, è necessario mettere in piedi strutture di accesso per conteggio e posizioni, se presenti e memorizzate in separatamente. Se cioè è possibile accedere in modo diretto ai puntatori documentali di indice iq, deve essere possibile accedere allo stesso modo alle parti rimanenti dell'indice.

5 Gestione della lista dei termini

La lista dei termini di un indice può essere molto ingombrante. Esistono diversi metodi per rappresentarla: in questo contesto, ne vedremo uno piuttosto sofisticato, che ha il grosso vantaggio di essere in grado, data una lista di stringhe, di restituire il numero ordinale di qualunque elemento della lista, ma di *non memorizzare la lista stessa*: in effetti, l'occupazione di memoria sarà di 1,23 n interi di $\log n$ bit, dove n è il numero delle stringhe

Una funzione di hash per un insieme di chiavi $X \subseteq U$, dove U è l'universo di tutte le chiavi possibili, è una funzione da $f : X \to m$. Quando la funzione è iniettiva (cioè non esistono *collisioni*) lo hash è detto *perfetto*. Se $|X| = m$, lo hash è detto *minimale*. Se, infine, esiste un ordine lineare su X e si ha $x \leq y$ se e solo se $f(x) \leq f(y)$ diremo che f *preserva l'ordine*. Il nostro scopo è costruire, per un insieme di stringhe arbitrario, una funzione di hash minimale, perfetta, e che preservi l'ordine. (Si noti che di per sé non stiamo chiedendo qual è il risultato di f su $U \setminus X$: questo problema verrà risolto a parte.)

Esistono numerose tecniche per la costruzione di hash perfetti [CHM97]. Quello che andiamo a descrivere è basato sulla teoria dei grafi casuali, e utilizza la randomizzazione per la costruzione della struttura dati. Quella che descriveremo è in effetti una tecnica completamente generale per memorizzare una funzione statica $f : X \to 2^b$ dall'insieme X all'insieme di valori 2^b. Scegliendo $b = \lceil \log n \rceil$ e mappando ogni termine di un indice al suo rango lessicografico otterremo il risultato richiesto.

L'idea della costruzione è la seguente: prendiamo due funzioni di hash aleatorie $h : U \to m$ e $g : U \to m$ con $m \geq n$, dove n è la cardinalità di X, e consideriamo un vettore $\boldsymbol{w}$ di interi a b bit che sia soluzione del sistema

$$w_{h(x)} \oplus w_{g(x)} = f(x) \qquad x \in X$$

Chiaramente, memorizzando h, g e il vettore $\boldsymbol{w}$ abbiamo memorizzato f: per calcolare $f(x)$ è sufficiente calcolare $h(x)$ e $g(x)$, recuperare dal vettore $\boldsymbol{w}$ i valori $w_{h(x)}$ e $w_{g(x)}$ e restituire $w_{h(x)} \oplus w_{g(x)}$.

Ora, è chiaro che abbiamo a che fare con un sistema di n equazioni nelle variabili w_i, che potrebbe non avere soluzione. Possiamo però rappresentare i vincoli imposti dal sistema come segue: costruiamo un grafo G non orientato con m vertici e n lati in cui per ogni $x \in X$ abbiamo che $h(x)$ è adiacente a $g(x)$ tramite un lato etichettato da x.

Andiamo ora a *esfoliare* il grafo così costruito. Partiamo cioè da una qualunque foglia v (un vertice di grado 0 o 1), e la rimuoviamo dal grafo. Se è adiacente a un altro vertice, rimuoviamo anche il lato corrispondente, che sarà etichettato, diciamo, da $x \in X$, e mettiamo la coppia $\langle v, x \rangle$ in uno stack.

Continuiamo finché non ci sono più vertici rimasti, o incontriamo un ciclo. Se non incontriamo alcun ciclo, alla fine ogni equazione del sistema originale comparirà nello stack, e risolvendo le equazioni una alla volta estraendole dallo stack avremo la certezza di poter risolvere il sistema, perché ogni volta che estraiamo una coppia $\langle v, x \rangle$ dallo stack possiamo risolvere l'equazione

$$w_{h(x)} \oplus w_{g(x)} = f(x) \qquad x \in X$$

sfruttando il fatto che $v = h(x)$ o $v = g(x)$, e v non è stato certamente assegnato precedentemente, dato che ogni vertice in una coppia non compare mai in coppie successive. Essenzialmente, stiamo *triangolando* la matrice associata al sistema (si vedano i lucidi animati online).

Ora, assumendo che h e g siano casuali e indipendenti, il grafo che andiamo a costruire è un grafo casuale di m vertici con n archi, e un risultato importante di teoria dei grafi casuali dice che per n sufficientemente grande quando $m > 2{,}09\,n$ il grafo è *quasi sempre* privo di cicli (quasi sempre significa che il rapporto tra il numero di grafi privi di cicli e il numero totale di grafi tende a uno). In sostanza, scegliendo bene h e g quasi tutti i grafi che otterremo permetteranno di risolvere il sistema.

Il caso ora descritto non è in realtà ottimo. La teoria degli ipergrafi casuali ci dice che utilizzando 3-ipergrafi (cioè un insieme di sottoinsiemi di ordine 3 dell'insieme dei vertici) è possibile dare una nozione di aciclicità che permette di risolvere i sistemi nel caso di *tre* funzioni di hash, ma in questo caso il limite che garantisce l'aciclicità è $m > 1{,}23\,n$ — un miglioramento netto rispetto al caso di ordine 2.

Alla fine, per memorizzare la funzione statica $f : X \to 2^b$ dovremo utilizzare solo $1{,}23b$ bit.

5.1 Firme

La funzione così costruita, per quanto sia minimale, perfetta e preservi l'ordine non permette di riconoscere se un elemento fa parte di X o no. Per ovviare all'inconveniente, utilizziamo un insieme di *firme* associato all'insieme X delle chiavi. Consideriamo cioè una funzione $s : U \to 2^r$ che associ a ogni chiave possibile una sequenza di r bit "casuale", nel senso che la probabilità che $s(x) = s(y)$ se x e y sono presi uniformemente a caso da U è 2^{-r} (per esempio, una buona funzione di hash). Consideriamo l'enumerazione ordinata $x_0 \leq x_1 \leq \cdots x_{n-1}$ degli elementi di X, e, oltre a f, memorizziamo ora una tabella di n firme a r bit $S = \langle s(x_0), s(x_1), \ldots, s(x_{n-1}) \rangle$.

Per interrogare la struttura risultante su input $x \in U$, agiamo come segue:

1. calcoliamo $f(x)$ (che è un numero in n);

2. recuperiamo $S_{f(x)}$;

3. se $S_{f(x)} = s(x)$, restituiamo $f(x)$; altrimenti, restituiamo -1 (a indicare che x non fa parte di X).

Si noti che se $x \in X$, $f(x)$ è il suo rango in X, e quindi $S_{f(x)}$ conterrà per definizione $s(x)$. Se invece $x \notin X$, $S_{f(x)}$ sarà una firma arbitraria, e quindi restituiremo quasi sempre -1, tranne nel caso ci sia una collisione di firme, il che avviene, come detto, con probabilità 2^{-r}. Tarando il numero r di bit delle firme è quindi possibile bilanciare spazio occupato e precisione della struttura.

5.2 Ottimizzazioni

È possibile migliorare ulteriormente l'occupazione in spazio quando b non è troppo piccolo utilizzando un *vettore compatto* per memorizzare i valori della soluzione del sistema. In effetti, per come abbiamo descritto il processo di soluzione non è possibile che vengano poste a valori diversi da zero più di n delle variabili. Se potessimo memorizzare, con una piccola perdita in spazio, *solo i valori diversi da zero* potremmi ridurre l'occupazione di memoria a quasi nb bit.

Per farlo, consideriamo un vettore di bit $\boldsymbol{b}$ che contiene in posizione i un uno se la variabile corrispondente w_i è diversa da zero, e zero altrimenti. Definiamo l'operatore di *rango*

$$\operatorname{rank}(\boldsymbol{b}, i) = \left|\left\{j < i | b_j \neq 0\right\}\right|,$$

che conta il numero di uni a sinistra della posizione i. Chiaramente, se mettiamo in un vettore C i valori delle variabili w_i diversi da zero, abbiamo

$$w_i = \begin{cases} 0 & \text{se } b_i = 0; \\ C[\operatorname{rank}(\boldsymbol{b}, i)] & \text{se } b_i \neq 0. \end{cases}$$

Quanto spazio occorre per calcolare rapidamente il rango? Esistono soluzioni molto sofisticate che occupano spazio $o(n)$, ma qui proponiamo un metodo molto semplice: scelto un quanto q, memorizziamo in una tabella R i valori di $\operatorname{rank}(\boldsymbol{b}, kq)$. Per calcolare il rango in posizione p, basta recuperare $R[\lfloor p/q \rfloor]$ e completare il calcolo contando gli uni dalla posizione $q\lfloor p/q \rfloor$ alla posizione p (utilizzando le istruzioni di conteggio delle CPU moderne). Per q fissato, il tempo del calcolo è costante. Lo spazio occupato dipende anch'esso da q, che permette quindi di bilanciare spazio occupato e velocità.

Alla fine, la funzione occuperà $nb + 1{,}23\,n$ bit, più il necessario per la struttura di rango (per esempio, se $q = 512$ saranno necessari altri $0{,}125\,n$ bit). Si noti che è impossibile scrivere una tale funzione in meno di nn bit, dato che esistono $(2^b)^n$ funzioni da un insieme di n a un insieme di 2^b elementi.

5.3 Auto-firma

Un utilizzo interessante delle funzioni che abbiamo descritto è quello di memorizzare efficientemente un dizionario statico approssimato. Per farlo, consideriamo un insieme $X \subseteq U$ che vogliamo rappresentare, e una funzione di firma $s : U \to 2^r$.

Possiamo utilizzare la tecnica generale di rappresentazione delle funzioni per scrivere in $nr + 1{,}23\,n$ bit la funzione $f : X \to 2^r$ data da $f(x) = s(x)$. La funzione f, cioè, mappa ogni chiave nella propria firma. Il dizionario viene a questo punto interrogato come segue:

1. calcoliamo $f(x)$;
2. se $f(x) = s(x)$, restituiamo "appartiene a X"; altrimenti, restituiamo "non appartiene a X".

Si noti che se $x \in X$, $f(x)$ è la sua firma $s(x)$, e quindi restituiremo "appartiene a X". Se invece $x \notin X$, $f(x)$ sarà un valore arbitrario, e quindi restituiremo "non appartiene a X" tranne nel caso ci sia una collisione di firme, il che avviene con probabilità 2^{-r}. Tarando il numero r di bit delle firme è quindi possibile bilanciare spazio occupato e precisione della struttura.

Rispetto a un filtro di Bloom, un dizionario di questeo tipo è molto più compatto (un filtro di Bloom per avere precisione 2^{-r} ha bisogno di 1,44 bit per elemento) e molto più veloce (le funzioni sono valutabili con al più tre fallimenti della cache, contro gli r di un filtro di Bloom). Per contro, non è modificabile.

6 Risoluzione delle interrogazioni

La risoluzione di un'interrogazione dipende fondamentalmente da due fattori:

- il linguaggio di interrogazione e la corrispondente semantica;
- il tipo di dati contenuti nelle affissioni dell'indice.

Per il momento, ci concentreremo sulla risoluzione di interrogazioni *booleane*, cioè in cui gli unici operatori sono la congiunzione, disgiunzione e negazione logica. Una formula del linguaggio di interrogazione è cioè un termine o la congiunzione, disgiunzione o negazione di formule.

La semantica di una formula è una lista di documenti. La semantica di un termine è data dalla lista di documenti in cui il termine compare, mentre la semantica della congiunzione e della disgiunzione sono, rispettivamente, l'intersezione e l'unione di liste. La semantica della negazione è la complementazione (rispetto all'intera collezione documentale).

Chiaramente, per risolvere un'interrogazione Booleana è sufficiente che le affissioni contengano i puntatori documentali. Inoltre, se l'indice contiene i puntatori in ordine crescente, è possibile ottenere intersezione e unione di liste in tempo lineare nel numero di puntatori.

È di grande interesse operare in maniera *pigra* — leggere cioè dalle liste in input solo i puntatori necessari ad emettere un certo prefisso dell'output. Questo permette di gestire in maniera efficiente la *terminazione anticipata* (*early termination*), tecnica con cui il calcolo della semantica viene arrestato sulla base di stime della qualità dei documenti già rinvenuti.

Per fondere liste in tempo lineare (con perdita logaritmica nel numero delle liste in ingresso) è sufficiente una *coda di priorità indiretta* che contiene (puntatori alle) liste, prioritizzate secondo il documento corrente. A ogni passo, il prossimo documento restituito sarà il documento corrente della lista in cima alla coda (cioè quella che correntemente è posizionata sul documento più piccolo). Una volta deciso il documento, avanziamo la lista in cima alla coda e aggiustiamo la coda stessa finché la lista in cima alla coda non è posizionata su un documento diverso. Si noti che in questo modo siamo anche in grado di sapere *quali* liste contengono il documento corrente.

Per quanto riguarda l'intersezione, sebbene in linea di principio il limite inferiore lineare non sia superabile, esistono diverse euristiche che permettono di accelerare la computazione nel caso l'indice fornisca la possibilità di effettuare *salti*.

In linea di principio, potremmo sempre tenere conto tramite una coda di proprità indiretta del documento minimo corrente, come nel caso precedente, ma tenere al tempo stesso traccia del massimo puntatore in una variabile. Quando coincidono, il puntatore sta nell'intersezione e può essere restituito (e immediatamente dopo, una qualunque lista viene fatta avanzare). Altrimenti, la lista che realizza il minimo viene fatta saltare fino al massimo e la coda viene aggiornata. Dato che viene effettuato al più un aggiornamento per avanzamento, il tempo richiesto è lineare nella dimensione di input, con una perdita logaritmica nel numero di input.

Da un punto di vista pratico, però, è spesso più efficiente utilizzare una soluzione che ha in linea teorica un comportamento molto peggiore (la perdita nel numero di input è *lineare*). L'idea è di fare avanzare in maniera miope le liste in ordine fino al massimo corrente m (all'inizio, il primo puntatore della prima lista) finché non sono tutte allineate. Al *primo* disallineamento (che causa un incremento di m) si ricomincia ad allineare la prima lista. Quando si arriva ad allineare l'ultima lista, si ha un puntatore da restituire.

L'aspetto interessante di questo approccio è che ordinando le liste in ordine di frequenza (quando è nota — per esempio nel caso di termini) la maggior parte degli avanzamenti verrà giocato dalle prime liste, che essendo quelle di minima frequenza genereranno pochi allineamenti. Le liste più dense subiranno pochi avanzamenti molto consistenti, che potranno essere gestiti in maniera efficiente da un sistema di salti. È anche possibile implementare una versione adattiva che mano a mano che scandisce le liste in input (non necessariamente liste di termini) ne stima la frequenza e riordina le liste al volo.

Si noti che nel caso le liste siano interamente caricabili in memoria, il problema diventa completamente diverso, ed esistono diverse soluzioni molto sofisticate al problema dell'intersezione [DLOM00]. L'osservazione fondamentale è che nell'intersecare due liste se una è molto più corta dell'altra (per esempio, esponenzialmente più corta) può essere conveniente cercare con una ricerca dicotomica gli elementi della lista più corta all'interno della lista più lunga.

Questa osservazione permette di risolvere rapidamente in memoria centrale, in maniera non pigra, intersezioni di liste. È sufficiente ordinare le liste in ordine di frequenza (questo è sempre possibile, essendo l'algoritmo non pigro) e procedere a ridurre, una lista alla volta, il numero di candidati. L'algoritmo da utilizzare dipenderà ovviamente dal numero corrente di candidati e dalla lunghezza della lista in cui li dobbiamo trovare, ma la speranza è che quando arriveremo alle liste più lunghe (le ultime) i candidati siano così pochi che sia possibile utilizzare tecniche di intersezione come quelle summenzionate.

6.1 Indici distribuiti

Le dimensioni degli indici del web sono tali da rendere poco pratico l'accumulo dell'intero indice su una sola macchina. È quindi necessario *segmentare* (o *partizionare*) l'indice complessivo in sottoindici, detti *segmenti*, le cui risposte verranno poi opportunamente combinate. È sottinteso che i segmenti saranno in genere memorizzati su un gruppo di macchine collegate in rete, e che un opportuno protocollo di comunicazione permetterà di accedere al contenuto dei segmenti in maniera remota.

In linea di principio, un indice può essere partizionato tramite una funzione che assegna a ogni affissione un segmento destinazione. Ad ogni segmento saranno associati i termini per i quali compare almeno un'affissione.

In pratica, però, è più comune scegliere un singolo criterio di partizionamento — *documentale* o *lessicale* — su cui basare il processo. In un partizionamento documentale, la collezione documentale viene divisa in blocchi (non necessariamente contigui) e a ogni blocco viene assegnato un segmento. Tutte le affissioni relative a un blocco andranno a finire nel suo segmento, e al segmento saranno associati i termini che compaiono nel blocco. Di norma, lo stesso termine comparirà in più segmenti (si pensi a termini comuni come le congiunzioni), mentre gli *hapax legomena* compariranno esattamente in un solo segmento.

In alternativa, è possibile partizionare la collezione lessicalmente: viene scelto un criterio per dividere l'insieme dei termini in più blocchi (non necessariamente lessicograficamente contigui), e a ogni blocco viene associato un segmento: la lista di affissioni relativa a un termine sarà interamente contenuta nel segmento relativo.

La ricostruzione delle liste di affissioni a partire da una segmentazione lessicale è banale: basta individuare il segmento che contiene il termine e interrogarlo. L'individuazione del segmento può non essere un problema banale se, ad esempio, i blocchi di termini non sono lessicograficamente consecutivi. Nel caso peggiore, l'unica soluzione è l'interrogazione di tutti i segmenti.

La ricostruzione delle lsite di affissioni a partire da una segmentazione documentale è più delicata, dato che nella maggior parte dei casi sarà necessario leggere vari frammenti della lista complessiva e combinarli. Questo richiede *in primis* di interrogare tutti i segmenti per sapere quali possiedono una lista di affissioni per il termine dato. Le liste vanno poi combinate — con

una semplice concatenazione e rinumerazione se i blocchi di documenti sono consecutivi, o con una fusione di liste in caso contrario.

In entrambi i casi precedenti può essere utile filtrare le richieste ai segmenti tramite dei dizionari approssimati, che possono rappresentare in maniera molto compatta l'insieme di termini presente in ogni segmento. Dimensionando i dizionari a seconda della memoria disponibile è possibile ridurre il numero di richieste inutili fatte ai segmenti.

Va notata una caratteristica importante del partizionamento documentale: è possibile cioè risolvere *direttamente* un'interrogazione complessa a livello di segmento. Questo fatto può portare a un miglioramento netto delle prestazioni, perché, ad esempio, in presenza di un'interrogazione congiunta di termini, alcuni segmenti, pur contenendo tutti i termini dell'interrogazione, potrebbero non restituire nessun documento. Nel caso di un'interrogazione disgiunta non c'è in ogni caso riduzione delle prestazioni. Va però fatto notare che se il processo di risoluzione della query costruisce artefatti (quali valori di ranking, calcolo della prossimità, ecc.) il protocollo di rete con cui ci si connette all'indice dovrà essere in grado di trasmetterli. Detto altrimenti, risolvendo e trasmettendo solo liste di affissioni il protocollo dipenderà solo dalla struttura dell'indice, mentre la risposta a interrogazioni con punteggio lo renderà dipendente dal meccanismo di assegnamento del punteggio.

D'altra parte, il partizionamento lessicale offre un'interessante possibilità: quella cioè di tenere in memoria centrale la parte dell'indice corrispondente ai termini più interessanti in qualche senso, o che risultano utilizzati più di frequente da una rilevazione empirica.

7 Una nozione astratta di sistema per il reperimento di informazioni

Nella sua accezione più generale, un sistema di reperimento di informazioni (*information-retrieval system*) è dato da una collezione documentale D (un insieme di documenti) di dimensione N, da un insieme Q di interrogazioni, e da funzione di ranking $r : Q \times D \to \mathbf{R}$ che assegna a ogni coppia data da un'interrogazione e un documento un punteggio (un numero reale). L'idea è che a fronte di un'interrogazione a ogni documento viene assegnato un punteggio: i documenti con punteggio nullo non sono considerati rilevanti, mentre quelli a punteggio non nullo sono tanto più rilevanti quanto il punteggio è alto.

Fissata un'interrogazione q il sistema assegna un rango (cioè crea una graduatoria) tra i documenti rilevanti, ordinandoli per punteggio; il rango è essenziale per restituire i documenti in un ordine specifico, in particolare quando la collezione documentale è di grandi dimensioni.

I criteri di per assegnare punteggi si dividono in *endogeni* ed *esogeni*. I due termini (non completamente formali) distinguono punteggi che utilizzano il contenuto del documento (cioè l'interno) da quelli che utilizzano struttura esterna (per esempio, il grafo dei collegamenti ipertestuali tra documenti). I criteri si dividono ulteriormente in *statici* (o indipendenti dall'interrogazione) e

dinamici (o dipendenti dall'interrogazione). Nel primo caso, il punteggio assegnato a ciascun documento è fisso e indipendente da q. Tutte le misure di centralità che abbiamo discusso possono essere utilizzate come punteggi esogeni, e quindi ci concentreremo su quelli endogeni.

La valutazione di un sistema per il reperimento di informazioni è basata sull'assunzione che a ogni interrogazione q sia assegnato un insieme di documenti *rilevanti* — quelli che un ipotetico utente considererebbe risposte valide all'interrogazione stessa. Il concetto può essere raffinato assumento un ordine (totale o parziale) sui documenti rilevanti, che deve essere il più possibile coincidente con il punteggio assegnato dal sistema.

8 Punteggi endogeni

I punteggi endogeni utilizzano il contenuto di un documento. Possono essere statici o dinamici (cioè dipendere o meno da un'interrogazione). Ci occuperemo dei metodi più classici, che si basano su un'interrogazione espressa come *bag of words* (letteralmente, "sacchetto di parole"). In questo modello, un'interrogazione è semplicemente un insieme di termini: non ci sono operatori booleani. Di solito viene utilizzata un'interpretazione implicita *disgiunta* (i documenti rilevanti sono solo quelli che contengono almeno uno dei termini), ma è anche possibile assumere un'interpretazione congiunta.

Il criterio più banale per assegnare un punteggio a un documento è quello di *conteggio*: assegnamo a un documento il punteggio dato dalla somma dei conteggi dei termini dell'interrogazione che compaiono nel documento stesso. In questo modo, documenti in cui i termini compaiono più frequentemente avranno un punteggio più elevato (la *ratio* essendo che se contengono con alta frequenza un termine desiderato devono essere più rilevanti).

Il conteggio è un metodo molto primitivo: innanzitutto si presta molto facilmente a manipolazione. Inoltre, non tiene conto del fatto che alcuni termini occorrono con grande frequenza non perché rilevanti, ma perché altamente frequenti all'interno di *ogni* documento. Per esempio, nell'interrogazione "Romeo e Giulietta" il metodo di conteggio valuterà in maniera estremamente positiva documenti contenenti un grande numero di "e".

Per ovviare a questi inconvenienti sono stati sviluppati dei *schemi di pesatura* (*weighting schemes*) che aggiustano il punteggio assegnato a ogni termine in ogni documento in modo da ovviare agli inconvenienti del conteggio.

Il primo e più classico metodo è il *TF/IDF* (*term frequency/inverse document frequency*) [Spa72]. Esso normalizza il conteggio (*term frequency*) dividendolo per il massimo conteggio all'interno del documento, o per la lunghezza del documento, e inoltre lo attenua moltiplicandolo per l'inverso della frequenza del termine, o, più frequentemente, per il logaritmo del numero di documenti diviso per la frequenza (cioè per il logaritmo dell'inverso della frequenza in senso probabilistico). In formule, se c è il conteggio di un termine t nel documento d di lunghezza ℓ

e f è la frequenza di t nella collezione documentale il peso di t nella valutazione di d è

$$\frac{c}{\ell} \log \frac{N}{f}$$

Come accennato, al posto di ℓ è possibile utilizzare il massimo conteggio di un termine che compare in c. Sono possibili molte altre varianti (per esempio, applicare un logaritmo a c).

Tornando all'esempio precedente, il termine "e" comparirà in quasi tutti i documenti, e verrà quindi pressoché ignorato dallo schema TF/IDF, dato che $N/f \approx 1$. La normalizzazione sulla lunghezza del documento, per contro, cerca di tenere conto del fatto che in documenti lunghi è naturale che un termine compaia più volte. Si noti che, comunque, il termine di conteggio compare in maniera lineare, ed è quindi sempre facilmente soggetto a manipolazione.

Lo studio degli schemi di pesatura è parte arte, parte scienza e parte magia. Numerosi schemi sono stati inventati in maniera puramente euristica, e si sono dimostrati efficaci alla prova dei fatti. Uno degli schemi più celebri è BM25 (stando agli autori, il 25° tentativo), uno schema di pesatura basato sul *modello probabilistico*. BM25 pesa un termine come segue:

$$\frac{(1+k_1)c}{k_1((1-b)+b\ell/L)+c} \log \frac{N-f+0{,}5}{f+0{,}5},$$

dove b e k_1 sono dei parametri liberi che devono essere tarati sulla collezione documentale, e L è la lunghezza media di un documento nella collezione. La parte dentro al logaritmo è una versione del punteggio IDF. Si noti che la formula non è più lineare in c. In effetti, la formula cresce monotonicamente in c, ma ha limite asintotico $k_1 + 1$ (quindi un numero eccessivo di ripetizioni del termine non influisce più di tanto).

Si noti che se $k_1 = 0$ la formula si riduce alla parte IDF, mentre per k_1 molto grande la formula è approssivativamente lineare in c (giusto per avere un'idea, nelle applicazioni reali k_1 è in genere tra 1 e 3).

Il termine b serve a controllare l'influenza della lunghezza del documento (rispetto alla lunghezza media).

9 Metodi computazionali

Il principale metodo computazionale utilizzato per il calcolo di ranking basati sull'algebra lineare è il *metodo della potenza* (*power method*), che consente di calcolare con relativa facilità l'autovettore dominante di una matrice. Per semplicità, assumeremo in quanto segue che la matrice $\boldsymbol{M}$ in esame sia diagonalizzabile, ma questo non è un requisito necessario. È invece necessario che il primo autovalore sia *separato* dal secondo, e cioè che l'autovalore dominante (cioè di modulo massimo) abbia molteplicità uno. In tal caso, è facile osservare che dato un qualunque vettore $\boldsymbol{v}$ con componente non nulla lungo l'autovettore associato all'autovalore dominante, l'iterazione

$M^k \boldsymbol{v}$ produce un vettore in direzione sempre più vicina a quella dell'autovettore dominante. Più precisamente, partiamo da un vettore $\boldsymbol{v}_0$ e definiamo

$$\boldsymbol{v}_{k+1} = \frac{M\boldsymbol{v}_k}{\|M\boldsymbol{v}_k\|}.$$

Detto altrimenti, moltiplichiamo il vettore corrente per M e normalizziamo il risultato. Se $\boldsymbol{e}_0, \boldsymbol{e}_1, \ldots, \boldsymbol{e}_{n-1}$ è una base normalizzata di autovettori e $\boldsymbol{v}_0 = \sum \alpha_i \boldsymbol{e}_i$, abbiamo che

$$\boldsymbol{v}_{k+1} = M^k \boldsymbol{v}_0 = \sum \alpha_i M^k \boldsymbol{e}_i = \alpha_0 \lambda_0^k \left(\boldsymbol{e}_0 + \sum_{i>0} \frac{\alpha_i}{\alpha 0} \left(\frac{\lambda_i}{\lambda_0} \right)^k \boldsymbol{e}_i \right).$$

Per via delle assunzioni di separatezza, l'espressione tra parentesi converge a $\boldsymbol{e}_0$. Dato che dividiamo a ogni passo per la norma, $\boldsymbol{v}_k$ converge proprio a $\boldsymbol{e}_0$. Si noti inoltre che il resto è controllato a $(\lambda_1/\lambda_0)^k$: il metodo della potenza converge cioè *geometricamente* con ragione tanto più piccola quanto più grande è la separazione tra il primo e il secondo autovalore.

Il caso di PageRank. Quando si ha a che fare con una matrice stocastica, di fatto il metodo della potenza si identifica con il calcolo iterativo di un'approssimazione di una configurazione limite. Partendo da un vettore normalizzato (cioè a somma uno) si ottengono sempre vettori normalizzati, e quindi in realtà si itera semplicemente la moltiplicazione per la matrice che descrive la catena di Markov; bisogna però stare attenti al fatto che la matrice che descrive la transizione è la *trasposta* della matrice di cui vogliamo calcolare gli autovalori, o, equivalentemente, che vogliamo calcolarne gli autovalori *sinistri* e non quelli destri.

In ogni caso, il calcolo effettivo non viene eseguito in maniera matriciale, perché questo genererebbe un numero di moltiplicazioni ingestibile. In teoria, infatti, ogni nodo ha un collegamento verso ogni altro nodo (per via degli archi indotti dal fattore di attenuazione). In realtà possiamo gestire gli archi di questo tipo e quelli dovuti ai pozzi in maniera molto più semplice. Denotiamo con $\boldsymbol{d}$ il vettore caratteristico dei pozzi, e assumiamo di spostarci da un pozzo a un altro nodo scelto mediante il vettore di preferenza $\boldsymbol{v}$. La matrice che stiamo effettivamente usando è ora

$$\alpha G + \alpha \boldsymbol{d}\boldsymbol{v}^T + (1-\alpha)\mathbf{1}\boldsymbol{v}^T.$$

Se assumiamo che l'approssimazione corrente sia $\boldsymbol{x}$, la prossima sarà

$$\alpha \boldsymbol{x}^T G + \alpha \boldsymbol{x}^T \boldsymbol{d}\boldsymbol{v}^T + (1-\alpha)\boldsymbol{v}^T = \alpha \boldsymbol{x}^T G + \alpha\kappa \boldsymbol{v}^T + (1-\alpha)\boldsymbol{v}^T,$$

dove κ è la somma dell'approssimazione corrente su tutti i pozzi. La coordinata i-esima del vettore precedente dipende però dalle singole coordinate di $\boldsymbol{x}$ solo per mezzo dell'addendo $\boldsymbol{x}^T G$: il valore κ può essere calcolato facilmente a partire da $\boldsymbol{x}$ separatamente. In pratica, la coordinata i-esima della nuova approssimazione è data da una certa quantità di ranking (controllata da α) che

proviene da altri nodi che non sono pozzi, distribuita secondo G, e da due addendi addizionali, uno dovuto al teletrasporto, e uno dovuto al ranking diffuso dai pozzi. Questi ultimi dipendono solo da $\boldsymbol{v}$ e κ. Questo fa sì che il metodo della potenza possa essere implementato *iterando unicamente sui lati di G*.

Si noti che le coordinate della nuova approssimazione sono definite sulla base di quella vecchia: questo fa sì che il metodo della potenza richieda *due* vettori, uno contente l'approssimazione corrente e uno contenente l'approssimazione successiva. Quando la differenza tra i due vettori è sufficientemente piccola in una norma scelta opportunamente, l'iterazione viene terminata.

È possibile dimostrare che il secondo autovalore della matrice che utilizziamo è α o meno, e quindi la convergenza è garantita, ed è particolarmente veloce se α non è vicino a 1.

Il metodo di Gauss–Seidel. Una visione alternativa al problema di PageRank (e, più in generale, del calcolo dell'autovettore dominante) è data dalla riscrittura dell'equazione di PageRank sotto forma di sistema lineare. Per quanto abbiamo già notato, l'equazione che definisce l'autovettore dominante è, nel caso di una matrice stocastica,

$$\boldsymbol{x}^T = \alpha \boldsymbol{x}^T G + \alpha \boldsymbol{x}^T \boldsymbol{d}\boldsymbol{v}^T + (1-\alpha)\boldsymbol{v}^T,$$

che riscritta raccogliendo $\boldsymbol{x}$ ci dà

$$\boldsymbol{x}^T (I - \alpha(G + \boldsymbol{d}\boldsymbol{v}^T)) = (1-\alpha)\boldsymbol{v}^T.$$

Questa equazione può ora essere risolta tramite l'algoritmo di Gauss–Seidel,[13] che fornisce approssimazioni per le soluzioni di un'equazione della forma $M\boldsymbol{x} = \boldsymbol{b}$ partendo da un vettore $\boldsymbol{x}^{(0)}$ e calcolando

$$x_i^{(t+1)} = \left(b_i - \sum_{j<i} m_{ij} x_j^{(t+1)} - \sum_{j>i} m_{ij} x_j^{(t)} \right) \Big/ m_{ii}.$$

Riscrivere questa formula utilizzando al posto di M la matrice appena ottenuta dall'equazione di PageRank e semplificare la computazione in maniera simile a quanto abbiamo fatto per il metodo della potenza è possibile, e i dettagli (piuttosto complicati) possono essere letti nella documentazione della classe `PageRankGaussSeidel` distribuita dal LAW.

L'aspetto particolarmente interessante dell'algoritmo di Gauss–Seidel è che utilizza sempre il valore più recente calcolato per una certa componente, il che ci permette di utilizzare *un solo vettore*. Il fatto che l'algoritmo utilizzi in maniera aggressiva valori più precisi lo rende anche di fatto la sua convergenza molto più rapida.

[13] Utilizzare l'algoritmo di Gauss–Seidel richiede la verifica di una serie di condizioni sul sistema che ne garantiscono la convergenza e la stabilità numerica; in questo contesto daremo per assodato che tali condizioni sono verificate.

Bisogna però notare che le sommatorie sono indicizzate dalla prima, ma bensì dalla seconda compoente della matrice: trasferendo questa osservazione nel caso di PageRank, abbiamo bisogno dei *predecessori*, e non dei successori di un nodo per aggiornare il suo valore. Questo fa sì che dobbiamo scandire il grafo trasposto, ma al tempo stesso conoscere il grado positivo di ogni nodo per poter calcolare correttamente il suo contributo ai suoi successori; la conoscenza dei gradi positivi richiede quindi un'ulteriore vettore di interi.

Notazione

- $[\ell \,..\, r]$ Intervallo da ℓ a r (inclusi).

$X \subseteq Y$ L'insieme X è contenuto nell'insieme Y, ovvero per ogni $x \in X$ abbiamo anche $x \in Y$.

$X \subset Y$ L'insieme X è contenuto *propriamente* nell'insieme Y, ovvero $X \subseteq Y$ e $X \neq Y$.

$\mathbf{N}$ L'insieme del numeri naturali $\{0, 1, 2, \dots\}$.

n Il numero naturale n, ovvero l'insieme $\{0, 1, \dots, n-1\}$.

$\mathbf{Z}$ L'insieme del numeri interi.

$\mathbf{R}$ L'insieme del numeri reali.

$\mathbf{R}_{>0}$ L'insieme del numeri reali maggiori di zero.

$\perp$ Simbolo convenzionale per indicare la divergenza: $f(x) = \perp$ significa che f non è definita su x; si assume che $f(\perp) = \perp$.

X^n Potenza n-esima di X, vale a dire l'insieme delle n-uple ordinate di elementi di X.

X^* Chiusura di Kleene di X, vale a dire $\bigcup_{n \in \mathbf{N}} X^n$; equivalentemente, il monoide libero su X. Gli elementi di X^* sono detti *stringhe su* X.

ε La stringa vuota.

(*formula*) Notazione di Iverson: ha valore 0 se la formula è falsa, 1 se è vera [GKP94].

χ_A La funzione caratteristica dell'insieme A, vale a dire $\chi_A(x) = (x \in A)$.

Y^X L'insieme delle funzioni da X in Y.

2^X L'insieme delle parti di X (equivalentemente, l'insieme delle funzioni da X nell'insieme $2 = \{0, 1\}$).

$\operatorname{dom} R$ Dominio di R.

$\operatorname{cod} R$ Codominio di R.

$\operatorname{ran} R$ Rango di R, cioè $\{\, x \in \operatorname{dom} R \mid \exists y \in \operatorname{cod} R \; x \; R \; y \,\}$.

$\operatorname{imm} R$ Immagine di R, cioè $\{\, y \in \operatorname{cod} R \mid \exists x \in \operatorname{dom} R \; x \; R \; y \,\}$.

$\pi_i^{(n)}$ Proiezione i-esima a n argomenti: $\pi_i^{(n)}(x_1, x_2, \ldots, x_n) = x_i$.

$\varphi_z^{(n)}$ Funzione parziale ricorsiva con n argomenti e di indice z.

φ_z Funzione parziale ricorsiva con un argomento e di indice z.

$\varphi_u^{(2)}$ La funzione parziale universale: $\varphi_u^{(2)}(x, y) = \varphi_x(y)$.

Halt L'insieme di arresto $\{\, \langle\langle x, y \rangle\rangle \in \mathbf{N} \mid \varphi_u^{(2)}(x, y) \neq \perp \,\}$.

K La diagonalizzazione dell'insieme d'arresto $\{\, \langle x, x \rangle \in \mathbf{N} \mid \varphi_u^{(2)}(x, x) \neq \perp \,\}$.

$f \leq g$ Ordinamento per punti delle funzioni: $f \leq g$ sse per ogni $x \in \operatorname{dom} f = \operatorname{dom} g$ abbiamo $f(x) \leq g(x)$ (ovviamente, $\operatorname{cod} f = \operatorname{cod} g$ deve essere ordinato perché la cosa abbia senso).

$O(f)$ L'insieme $\{\, g \mid \exists \alpha \; g(x) \leq \alpha f(n)$ definitivamente $\,\}$.

$\Omega(f)$ L'insieme $\{\, g \mid \exists \alpha \; g(x) \geq \alpha f(n)$ definitivamente $\,\}$.

$\Theta(f)$ L'insieme $\{\, g \mid \exists \alpha, \beta \; \alpha f(x) \leq g(x) \leq \beta f(n)$ definitivamente $\,\}$.

TEMPO (f) La classe dei problemi risolubili in tempo f.

SPAZIO (f) La classe dei problemi risolubili in spazio f.

NTEMPO (f) La classe dei problemi con certificati verificabili in tempo f.

NSPAZIO (f) La classe dei problemi con certificati verificabili in spazio f.

P La classe dei problemi risolubili in tempo polinomiale.

NP La classe dei problemi con certificati verificabili in tempo polinomiale.

PSPAZIO La classe dei problemi risolubili in spazio polinomiale.

LOGSPAZIO La classe dei problemi risolubili in spazio logaritmico.

EXP La classe dei problemi risolubili in tempo esponenziale.

RP La classe dei problemi risolubili probabilisticamente in tempo polinomiale con errore limitato unilaterale.

$k \mid n$ k divide n (cioè $n = ak$ per qualche intero a).

$m \perp n$ m e n sono coprimi [GKP94].

$\mathbf{Z}_n$ Il gruppo dei resti modulo n (cioè classi di equivalenza di interi rispetto alla relazione di avere lo stesso resto nella divisione per n).

$\mathbf{U}_n$ Il gruppo delle unità di $\mathbf{Z}_n$ (cioè le classi associate a interi coprimi con n).

$\varphi(n)$ L'indicatore di Eulero (la cardinalità di $\mathbf{U}_n$).

Riferimenti bibliografici

[Blo70] Burton H. Bloom. Space-time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[Cha02] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pag. 380–388, 2002.

[CHM97] Zbigniew J. Czech, George Havas, e Bohdan S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1-2):1–143, 1997.

[DG08] Jeffrey Dean e Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[DLOM00] Erik D. Demaine, Alejandro López-Ortiz, e J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pag. 743–752. ACM Press, 2000.

[Dud13] Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *CoRR*, abs/1311.2540, 2013.

[Eli75] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21:194–203, 1975.

[GKP94] Ronald L. Graham, Donald E. Knuth, e Oren Patashnik. *Concrete Mathematics*. Addison–Wesley, second edizione, 1994.

[Gol80] Solomon W. Golomb. Sources which maximize the choice of a Huffman coding tree. *Inform. and Control*, 45(3):263–272, 1980.

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, volume 2180 della collana *Lecture Notes in Computer Science*, pag. 300–314, Berlin, Heidelberg, 2001. Springer-Verlag.

[HN99] Allan Heydon e Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, pag. 219–229, December 1999.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, e Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pag. 654–663, El Paso, Texas, 1997.

[MJDS07] Gurmeet Singh Manku, Arvind Jain, e Anish Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*, pag. 141–150, 2007.

[MS96] Maged M. Michael e Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pag. 267–275. ACM, 1996.

[OCGO96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, e Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[PSS10] Felix Putze, Peter Sanders, e Johannes Singler. Cache-, hash-, and space-efficient Bloom filters. *Journal of Experimental Algorithmics (JEA)*, 14, 2010.

[Spa72] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[ZHNB06] Marcin Zukowski, Sándor Héman, Niels Nes, e Peter A. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, a cura di Ling Liu, Andreas Reuter, Kyu-Young Whang, e Jianjun Zhang, pag. 59. IEEE Computer Society, 2006.

GNU Free Documentation License

Version 1.3, 3 November 2008

<http://fsf.org/>

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.