Dispense per il corso di algoritmica per il web

Sebastiano Vigna

7 novembre 2022

Contents

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 X e Y. La definizionesi estende per ricorsione a n insiemi. Al prodotto cartesiano $X_1 \times X_2 \times \cdots \times X_n$ sono naturalmente associate le *proiezioni* $\pi_1, \pi_2, \ldots, \pi_n$ definite da

$$\pi_i(\langle x_1, x_2, \ldots, x_n\rangle) = x_i \tag{1}$$

poniamo

$$X^n = \overbrace{X \times X \times \cdots \times X}^{n \text{ volte}} \tag{2}$$

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

$$X + Y = X \times \{0\} \cup Y \times \{1\} \tag{3}$$

Di solito ometteremo, con un piccolo abuso di notazione, la seconda coordinata. Una *relazione* tra gli insiemi X_1, X_2, ..., X_n é un sottoinsieme R del prodotto cartesiano $X_0 \times X_1 \times \cdots \times X_n$. Se $n = 2$ si tende a scrivere $x\ R\ y$ per $\langle x, y\rangle \in \mathbb{R}$. Una relazione tra due insiemi é detta *binaria*. Se R é una relazione binaria tra X e Y, X é detto *dominio* di R, ed é denotato da $\mathrm{dom}(R)$, mentre Y é detto *codominio* di R, ed é denotato da $\mathrm{cod}(R)$. Il *rango* o *insieme di definizione* di R é l'insieme $\mathrm{ran}(R) = \{x \in X \mid \exists y \in Y, x\ R\ y\}$, e in generale puó non coincidere con il dominio di R. L'*immagine* di R é l'insieme $\mathrm{imm}(R) = \{y \in Y \mid \exists x \in X, x\ R\ y\}$, e in generale puó non coincidere con il codominio di R. Una relazione binaria R tra X e Y é *monodroma* se per ogni $x \in X$ esiste al piú un $y \in Y$ tale che $x\ R\ y$. É *totale* se per ogni $x \in X$ esiste un $y \in Y$ tale che $x\ R$, cioé se $\mathrm{ran}(R) = \mathrm{dom}(R)$. É *iniettiva* se per ogni $y \in Y$ esiste al piú un $x \in X$ tale che $x\ R\ y$. É *suriettiva* se per ogni $y \in Y$ esiste un $x \in X$ tale che $x\ R\ y$, cioé se $\mathrm{imm}(R) = \mathrm{cod}(R)$. É *biiettiva* se la relazione é sia iniettiva che suriettiva. Una *funzione* da X a Y é una relazione monodroma e totale tra X e Y (notate che l'ordine é rilevante[1]); in tal caso scriviamo $f : X \to Y$ per dire che f "va da X a Y". Se f é una funzione da X a Y é 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, iniettivitá, suriettivitá e biiettivitá vengono ereditate dalle relazioni. Se una funzione $f : X \to Y$ é biiettiva, é facile verificare che esiste una funzione inversa f^{-1}, che soddisfa le equazioni $f(f^{-1}(y)) = y$ e $f^{-1}(f(x)) = x$

[1]Secondo la mia interpretazione, una funzione é monodroma e totale perché una funzione é definita come una relazione in cui ogni elemento del dominio é mappato in uno e un solo elemento del codominio, dunque:

- monodroma garantisce che ogni elemento dell'insieme di definizione ha un'unica immagine.
- totale garantisce che $\mathrm{dom}(R) = \mathrm{ran}(R)$.

per ogni $x \in X$ e $y \in Y$. Una *funzione parziale* (che tecnicamente non é una funzione perché non é definita sull'interezza del suo dominio) da X a Y é una relazione monodroma tra X e Y; una funzione parziale puó non essere definita su elementi del suo dominio, fatto che denotiamo con la scrittura $f(x) = \perp$ ("$f(x)$ é indefinito" o "f é indefinita su x"), che significa che $x \notin \text{ran}(f)$. Date funzioni parziali $f : X \to Y$ e $g : Y \to Z$, la *composizione* $g \circ f$ di f con g é 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 $Y^X = \{f | f : X \to Y\}$ l'insieme delle funzioni da X a Y. Si noti che per insiemi finiti[2] $|Y^X| = |Y|^{|X|}$.
Denoteremo con n l'insieme $\{0, 1, \ldots, n-1\}$.
Dato un insieme X, il *monoide libero* su X, denotato da X^*, é l'insieme di tutte le sequenze finite, (inclusa quella vuota, normalmente denotata da ε) di elementi di X, dette *parole* su X, dotate dell'operazione di concatenazione, di cui la parola vuota é 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:

$$\chi_A = \begin{cases} 0, & \text{se } x \notin A \\ 1, & \text{se } x \in A \end{cases} \tag{4}$$

Per contro, a ogni funzione $f : X \to 2$ possiamo associare il sottoinsieme di X dato dagli elementi mappati da f in 1, cioé l'insieme $\{x \in X \mid f(x) = 1\}$; tale corrispondenza é inversa alla precedente, ed é quindi naturalmente equivalente considerare sottoinsiemi di X o funzioni di X in 2. Date due funzioni f, $g : \mathbb{N} \to \mathbb{R}$, diremo che f é di *ordine non superiore* a g, e scriveremo che $f \in \mathcal{O}(g)$ ("f é $\mathcal{O}$-grande di g") se esiste una costante $a \in \mathbb{R}$ tale che $|f(n_0)| \leq |ag(n_0)|$ definitivamente. Diremo che f é di *ordine non inferiore* a g, e scriveremo che $f \in \Omega(g)$ se $g \in \mathcal{O}(f)$. Diremo che f é *dello stesso ordine di* g e scriveremo $f \in \Theta(g)$, se $f \in \mathcal{O}(g)$ e $g \in \Omega(f)$.
Un *grafo semplice* G é 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 é cioé una coppia non ordinata di vertici distinti. Se $\{x, y\} \in E_G$, diremo che x e y sono vertici *adiacenti* in G. Un grafo puó essere rappresentato graficamente disegnando i suoi vertici come punti sul piano, e rappresentato i lati come segmenti che congiungono vertici adiacenti. Per esempio, il grafo con insieme di vertici 4 e insieme di lati $\{\{0, 1\}, \{1, 2\}, \{2, 0\}, \{2, 3\}\}$ puó essere rappresentato come segue:

L'*ordine* di G é il numero naturale $|V_G|$. Una *cricca* o una *clique* di G é un insieme di vertici $C \subseteq V_G$ mutualmente adiacenti (nell'esempio in figura $\{0, 1, 2\}$ é una cricca). Dualmente, un *insieme indipendente* di G é un insieme di vertici $I \subseteq V_G$ mutualmente non adiacenti. Un *cammino* di lunghezza n in G é una sequenza di vertici $x_0, \ldots, x_n$ tale che x_i é adiacente a x_{i+1} con $(0 \leq i < n)$. Diremo che il cammino va da x_0 a x_n. Nell'esempio in figura, 0, 1, 2 é un

[2]Si noti che l'uguaglianza é vera in generale, utilizzando i cardinali cantoriani

cammino, 1, 3 non lo é.
Un grafo *orientato* G é dato da un insieme di nodi N_G e un insieme di archi A_G e da funzioni s_G, $t_G : A_G \to n_G$ (*source*, *target*) che spcificano 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 paralleli é detto *separato*. Il *grado positivo* o *outdegree* $d^+(x)$ di un nodo x é il numero di archi uscenti da x, cioé $|s_G^{-1}(x)|$. Dualmente, il *grado negativo* o *indegree* $d^-(x)$ di un nodo x é il numero di archi entranti in x, cioé $|t_G^{-1}(x)|$. In un grafo orientato G un *cammino* di lunghezza n é 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 *raggiungibilitá*: $x \rightsquigarrow y$ se esiste un cammino da x a y. La relazione di equivalenza $\sim$ é ora definita da $x \sim y \iff x \rightsquigarrow y \wedge y \rightsquigarrow x$. Le classu di equivalenza di $\sim$ sono dette *componenti fortemente connesse* di G, G é *fortemente connesso* quando é costituito da una sola componente.
La funzione $\lambda(x)$ denota il bit piú significativo dell'espansione binaria di x: quindi $\lambda(1) = \lambda(1_2) = 0$, $\lambda(2) = \lambda(10_2) = 1$ etc...
Per convenzione $\lambda(0) = -1$. Si noti che per $x > 0$ si ha $\lambda(x) = \lfloor \log x \rfloor$.

2 Crawling

Il *crawling* é l'attivitá di scaricamento delle pagine web. Un *crawler* é un dispositivo software che visita, scarica e analizza i contenuti delle pagine web a partire da un insieme di pagine dato, detto *seme*. Il crawler procede nel suo processo di visita seguendo i collegamenti ipertestuali conenuti nelle varie pagine.
Le pagine web durante il processo di crawl si dividono in tre:

- L'insieme delle pagine *visitate*, V, che sono giá state scaricate e analizzate;
- La *frontiera*, F, che é l'insieme delle pagine conosciute ma che non sono ancora state visitate;
- Línsieme U degli URL sconosciuti.

Le differenze tra l'attivitá di crawling e una banale visita all'interno di un grafo sono molto importanti, prima di tutto c'é il fatto che un crawl ha una dimensione ignota, non conosciamo $|V_G|$; secondariamente la frontiera é un enorme problema, in quanto la sua dimensione tende a crescere molto piú velocemente dell'insieme dei visitati.
In generale l'operazione di crawling parte caricando il seme in frontiera e, finché la frontiera non é vuota, viene estratto un URL dalla frontiera, secondo determinate politiche, l'URL viene visitato (e quindi scarica la pagina corrispondente), lo analizza derivandone nuovi URL tramite i collegamenti ipertestuali contenuti all'interno della pagina e sposta l'URL nell'insieme dei visitati. I nuovi URL vengono invece aggiunti alla frontiera se sono sconosciuti, e quindi non sono in $V \cup F$.
Diverse politiche di prioritizzazione della frontiera possono poi dare luogo ad approcci diversi al processo di crawling, posso per esempio estrarre prima degli URL a cui si arriva partendo da pagine che contengono determinate parole chiave.

2.1 Il crivello

Il crivello é la struttura dati di base di un crawler, questo accetta in ingresso URL potenzialmente da visitare e permette di prelevare URL pronti alla visita. Ogni URL viene estratto una e una sola volta in tutto il processo di crawling, indipendentemente da quante volte é stato inserito all'interno della struttura. In questo senso il crivello unisce le proprietá di un dizionario a quelle di una coda con prioritá e rappresenta al tempo stesso la frontiera, l'insieme dei visitati e la coda di visita. Combinare questi aspetti in una sola struttura é un lavoro complesso ma permette risparmi notevoli dal punto di vista pratico[3].
Una prima osservazione é che spesso, per mantenere l'informazione di quali URL sono stati giá visitati ($V \cup F$) é preferibile sostituire gli URL con delle *firme*, cioé con il risultato del calcolo di $h(u)$, dove h é una funzione di hash definita

[3]Si noti che é possibile riordinare ulteriormente gli URL *dopo* l'uscita dal crivello

sulle stringhe e restituisce un hash di dimensione arbitraria, per esempio 64 bit. Questo ha due grandi benefici:

- Risparmio di spazio non trascurabile, molti URL possono essere di grandi dimensioni e salvarli sempre in un centinaio[4] si rivela una buona fonte di risparmio
- Uniformiamo le lunghezze degli URL a un valore standard

Il drawback di una soluzione del genere é che accettiamo il fatto che vi siano delle collisioni, é dunque possibile che due URL diversi vengano mappati sullo stesso valore di hash. Questo fenomeno é inevitabile, peró se abbiamo una funzione di hash che lavora su un numero di bit abbastanza grande, la probabilitá di incontrare una collisione sará cosí bassa da essere trascurabile.
Github.com implementa una soluzione del genere, viene impiegato SHA-1 (funzione di hash a 160bit) per calcolare un hash dell'URL di ciascuna delle repository nei loro database, la probabilitá di collisione é cosí bassa che é sostanzialmente impossibile. Adesso sembra che vogliano muoversi verso SHA-256.
Supponiamo ora di avere in memoria n firme, la probabilitá che una nuova firma collida con una di quelle esistenti é n/u, dove u é la dimensione dell'universo delle possibili firme. Nel caso di un sistema a 64 bit $u = 2^{64}$, e quindi possiamo memorizzare 100 miliardi di URL con una probabilitá di falsi positivi nell'ordine di $10^{11}/2^{64} < 2^{37}/2^{64} = 1/2^{27} < 1/10^8$, quindi avremo meno di un errore ogni 100 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 un'attività di crawling di piccole dmensioni. Per dimensioni più grandi è necessario ingegnarsi e impiegare delle strutture dati, in parte, o completamente su disco, che consentano di mantenere l'occupazione totale di memoria centrale costante.
Questo è un compromesso tipico delle attività di crawling - strutture che si espandono in memoria centrale proporzionalmente alla frontiera sono troppo fragili e quindi rischiano di mandare in crash il processo o di sovraccare il sistema di gestione della memoria virtuale.
Solitamente le strutture dati impiegate in ambiti di crawling importanti dovrebbero entrare in un processo di *degrado grazioso*, riducendo le performance, ma senza interrompere all'improvviso il funzionamento del programma.

2.2 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 a falsi positivi che rappresenta un dizionario, cioè un insieme di elementi da un universo dato. Permette di aggiungere elementi all'insieme e chiedere se un elemento è presente o meno nell'insieme.

[4] valore d'esempio arbitrario

Un filtro di Bloom con universo X è rappresentato da un vettore **b** di m bit e da d funzioni di hash $f_0, \ldots, f_d$ da X in m. Per aggiungere un elemento al filtro è necessario calcolare i valori per tutte le d funzioni di hash e mettere a 1 il bit $d_{f_i(x)}$ con $0 \leq i < d$. Per sapere se un elemento è presente all'interno del filtro è comunque necessario calcolare tutte le d funzioni di hash, qualora esista $i \in [0, d) \mid d_{f_i(x)} = 0$ allora il valore non è presente nella struttura, altrimenti la risposta è positiva.
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 quell'elemento è stato memorizzato: è però possibile che i d bit siano stati messi a 1 a seguito di una serie di inserimenti precedenti, quindi la risposta a un'interrogazione per un elemento che non è presente nell'insieme risulta essere ugualmente positiva. Questo implica che a causa di collisioni sulle varie funzioni di hash noi possiamo avere dei *falsi positivi*, questo significa che il filtro da risposta positiva sebbene l'elemento non sia stato inserito nella struttura.
I filtri di Bloom sono chiamati in questa maniera perchè sono molto utili come filtri per strutture dati più lente che stanno su disco. Se si prevede che la maggior parte delle richieste avrà risposta negativa, un filtro di Bloom può ridurre significativamente gli accessi alla struttura soggiacente; oltre a questo il filtro tende a rispondere molto velocemnte a richieste che hanno risposta negativa, basta infatti che una sola delle posizioni indicate dalle funzioni di hash abbia bit a 0 per rispondere falso, mentre è necessario controllare tutte le posizioni e accedere alla struttura dati soggiacente nel caso in cui la risposta sia positiva.
Di fatto, i filtri di Bloom sono risultati estremamente pratici per mantenere insiemi di grandi dimensioni in memoria, in particolare quando le dimensioni delle chiavi sono significative (e.g. degli 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 memorizzabili 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.

2.2.1 Dimostrazione dell'efficacia dei filtri di Bloom

Per calcolare l'efficacia dei filtri di Bloom, come detto in precedenza, si considererà la probabilità di osservare un falso positivo. Una prima semplificazione lecita (seguendo l'analisi di [Blo70]) è quella di andare a calcolare la probabilità di un positivo qualunque dopo n inserimenti, che è ovviamente una maggiorazione del caso dei falsi positivi. Supponiamo di avere un vettore di m posizioni e d funzioni di hash uniformemente distribuite e indipendenti. Dopo l'n-esimo

inserimento, la probabilità che un bit sia 0 è data da:

$$P[\mathbf{b}[i] = 0 \mid N = n] = 1 - P[\mathbf{b}[i] = 1 \mid N = n] = \left(1 - \frac{1}{m}\right)^{dn}$$

Si ottiene un positivo quando tutte le posizioni controllate sono a 1, ciò avviene con probabilità

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

siccome $(1 + \alpha/n)^n \to e^\alpha$ per $n \to \infty$

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

Sia ora $p = e^{-\frac{dn}{m}}$, allora

$$\ln(p) = -\frac{dn}{m}$$
$$m\ln(p) = dn$$
$$d = -\frac{m\ln(p)}{n}$$

Voglio ora minimizzare la probabilità di ottenere un positivo φ, quindi prendo $\left(1 - e^{-\frac{dn}{m}}\right)^{d}$ e sostituisco p, quindi $(1-p)^{m/n\cdot\ln(p)}$.
Riscrivo come esponenziale:

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

Faccio la derivata rispetto a p che viene:

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

Il punto stazionario è quello per cui la parte tra parentesi si annulla, l'esponenziale è sempre > 0 quindi:

$$\frac{1}{p}\ln(1-p) - \frac{1}{1-p}\cdot\ln(p) = 0$$
$$(1-p)\cdot\ln(1-p) = p\cdot\ln(p)$$

Se $1-p = p$ allora $p = 1/2$, questa è l'unica soluzione, come prova del 9 si studi $g(p) = p\ln(p) - (1-p)\ln(1-p)$ e risulterà che agli estremi la funzione vale 0 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 invece è $\dot{g(p)} = \ln(1-p) + \ln(p) + 2$.
Chiaramente va a meno infinito in 0 e 1, ma in $p = 1/2$ è positiva, ed è l'unico punto di massimo (dato che la derivata seconda 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 $(0, 1)$. Per concludere, se $p = 1/2$ allora $d = -\frac{m \ln(p)}{n} = \frac{m \ln 2}{n}$ per quanto riguarda la probabilità di avere uno in tutti i punti che andiamo a controllare all'interno del vettore:

$$\varphi = \left(1 - e^{-\frac{dn}{m}}\right)^d = \left(1 - e^{-\ln 2}\right)^d = \left(1 - \frac{1}{2}\right)^d = 2^{-d}$$

Alla fine, la probabilità di (falsi) positivi è minimizzata da $d \approx \frac{m \ln 2}{n}$, e in tal caso la probabilità di un (falso) positivo è 2^{-d}. Vale a dire che aumentando linearmente il numero delle funzioni di hash impiegate e il numero di bit della struttura a $m \approx dn/\ln 2 \approx 1.44dn$, si ha una riduzione esponenziale del numero di (falsi) positivi che possono essere incontrati. Passiamo ora a fare alcune osservazioni tecniche:

- È abbastanza intuitivo, ed è possibile dimostrare, che per avere falsi positivi con probabililtà 2^{-d} occorre utilizzare almeno d bit per elemento. Quindi un filtro di bloom perde 44% in spazio rispetto al minimo possibile.

- In linea di principio il filtro di Bloom ha una modalità di accesso alla memoria pessima, a causa dei d accessi casuali al vettore, che possono causare d fallimenti in 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 perde di precisione; inoltre, la divisione delle chiavi in segmenti non è mai uniforme ma ha una distribuzione binomiale negativa. Questi fattori peggiorano la probabilità di errore [Sin10].

- Dall'analisi che abbiamo effettuato, 1/2 è anche la probabililtà di un bit a 0, quindi, come si diceva all'inizio del paragrafo, un filtro di Bloom è molto più efficace nel riportare il fatto che un elemento non è presente nel filtro piuttosto che a riportarne la presenza.

- 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 e Mitzenmacher hanno dimostrato che estraendo due numeri 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 centrale costante, è relativamente veloce ed affidablie e degrada graziosamente, man mano che il vettore si riempie la probabilità di falsi positivi aumenterà fino a diventare 1.

2.3 Crivelli basati su database NoSQL

Un modo più sofisticato e con 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 utlizzando 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 un B-tree parzialmente su disco. La memoria centrale è utilizzata come cache per accelerare le operazioni su disco.
Un approccio più sistematico, implementato inizialmente a Google sotto il nome di BigTable, è l'LSM tree [O'N96]. BigTable è stato successivamente reimplementato come progetto open-source, LevelDB, che è poi stato utilizzato come base per altri database NoSQL come RocksDB, l'implementazione di Facebook, che è utilizzata da commoncrawler, un crawler open-source.
Gli LSM tree sono basati su un concetto relativamente semplice, ma necessitano di un'implementazione accurata che sfrutti parallelismo e concorrenza per essere efficienti.
Un LSM-tree è diviso in 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, ma ha una certa elasticità nel dimensionamento (può essere, ad esempio, grande il doppio rispetto alla sua dimensione di base).
Il primo livello è sempre in memoria centrale e ha dimensione limitata a priori, solitamente è implmementato tramite un normale dizionario ordinato (RB-tree o B-tree).
I livelli successivi sono memorizzati sotto forma di *log* e 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, si esegue l'operazione di *scarico* in cui un gruppo di chiavi viene estratto dal primo livello e aggiunto al secondo, in modo da riportare il primo livello alla sua dimensione naturale.

A questo punto l'operazione di scarico continua ricorsivamente verso il basso fino a quando, se accade, anche l'ultimo livello eccede la propria dimensione massima, ed esegue un'operazione di scarico su un nuovo livello dell'albero.
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 arbitrario noto come *lapide*; la tecnica è simile a quella utilizzata per le tablle 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 dell'albero, questa viene scartata.
Ci sono a questo punto numerose e importanti questioni ingegneristiche e implementative da considerare:

- Il formato in cui vengono memorizzati i livelli può non essere uniforme, e può dipendere dalla tipologia di memoria di massa soggiacente. Ad esempio, un metodo di memorizzazione per nastri non può essere efficace per dischi elettromagnetici.

- 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 a filtri positivi a bassa precisione (come un filtro di Bloom) che evita l'accesso al file nel caso in cui la chiave che stiamo cercando non sia all'interno del file. La bassa precisione fa sì che l'occupazione in memoria del filtro non sia particolarmente rilevante.

- Ogni frammento può contenere un indice sparso che memorizza le posizioni di un sottoinsieme di chiavi campionate a intervalli regolari; in questo modo, in fase di ricerca è possibile identificare rapidamente la zona del frammento che potenzialmente contiene la chiave, per poi procedere con una ricerca binaria o lineare. La scelta della frequenza di campionamento consente di bilanciare lo spazio occupato dalla struttura e l'efficacia del processo di ricerca.

- Anche in assenza di fusioni di livelli, in generale in un LSM-tree vengono lasciati in esecuzione dei thread che si occupano di fare il *compattamento* della struttura:

  - Controllano che il numero di copie per chiave non sia eccessivo
  - Rimuovono eventuali lapidi in eccesso

- 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 perchè diverse implementazioni o politiche di aggiornamento possono essere adatte a diversi carichi di lavoro.
Si noti che al crescere della frontiera l'LSM-tree continua ad occupare la stessa quantità di memoria centrale e non ha decrementi di precisione, però il crivello rallenta e occupa più memoria di massa.

2.4 Un crivello offline

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

- Un file Z di URL già visitati ($V \cup F$), in ordine lessicografico.
- Un file F di URL ancora dia visitare, quindi la frontiera, 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 il crawl, gli URL da visitare vengono estratti da F (eventualmente alterandone l'ordine secondo qualche politica), e i nuovi URL che vengono incontrati vengono accumulati in A.
Quando F è vuoto o quando A raggiunge la dimensione massima si procede ad eseguire l'operazione di *fusione*:

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

La fusione di Z e A' può procedere in maniera sequenzialme perchè 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 sulla macchina che sta eseguendo 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 distribuito, come MapReduce [Ghe08], o la sua implementazione

open-source, 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'ordine 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 dedubplicare A utilizzando come chiavi le firme degli URL.
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 resta costante e non si hanno decrementi di precisione.

2.5 Il crivello di Mercator

Mercator è un crawler il cui funzionamento è stato spiegato in [Naj99], il cui crivello è una versione parzialmente in memoria del crivello offline descritto in precedenza (crivello offline). Le firme degli elementi in 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 che contiene firme di URL, inizialmente vuoto, che viene riempito incrementalmente. Il vettore è di dimensione fissa n. Su disco, invece, teniamo un file Z che contiene tutte le firme degli 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 raggiunge la massima dimensione; operiamo allora uno *scarico* nel seguente modo:

1. Ordino S indirettamente, cioè creo un vettore V di indici associati ai valori S[i], faccio ordinamento stabile sulle firme, dunque le firme S[V[i]] sono in ordine crescente al crescere di i.

2. Deduplico S, quindi elimino le occorrenze successive alla prima per ogni firma.

3. Z' = Z $\cup$ S marchiamo utili le firme in S che non compaiono in Z.

4. Scandiamo ogni entri di S e A in parallelo (sono entrambi ordinati) e diamo in output gli URL in A corrispondenti alle entry marcatre come utili al passo precedente.

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

Innanzitutto, si noti che Z, alla fine di uno scarico, contiene di nuovo le firme di tutti gli URL mai incontrati. Inoltr in output abbiamo dato tutti e soli gli URL la cui firma non era parte di Z, dunque si trattava di URL che non erano ancora stati visitati. Infine, gli URL in output sono stati ovviamente emessi nell'ordine di accodamento in A.

2.6 Gestione dei quasi duplicati

Durante l'attività di crawling è comune trovare pagine che sono quasi identiche (varianti dello stesso sito, calendari, immagini, etc. . .). In dipendenza dal tipo di crawling (quali sono le politiche del processo di crawling?), queste pagine andrebbero considerate duplicate e non ulteriormente elaborate.
Un modo semplice ma efficace di gestire il problema in memoria centrale è quello di analizzare una forma normalizzata del documento, rimuovendo marcatura, date e altri dati che sono standard ma che potrebbero risultare differenti sulla base di meccanismi automatici. Il documento dovrebbe poi essere memorizzato in un filtro di Bloom.
Metodi molto più sofisticati per la rilevazione dei duplicati possono essere utilizzati offline prima del processo di indicizzazione.
Un metodo efficace di gestione online del problema, che è stato utilizzato per qualche tempo dal crawler di Google [Sar07], è quello di porre in un dizionario (eventualmente approssimato) uno hash generato dall'algoritmo di 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 i cosiddetti *shingles* (segmenti di testo di 3-5 caratteri).
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 1 se e solo se:

$$|\{s \in S \mid h(s)[i] = 1\}| > |\{s \in S \mid h(s)[i] = 0\}$$

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 è [Sar07].

2.7 Gestione della politeness

Un altro dei problemi pratici che rende l'attività di crawling diversa da una semplice visita è la gestione della *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:

- Limitare il tempo tra una richiesta e l'altra.
- Limitare il rapporto tra il tempo di scaricamento e quello di non scaricamento.

Nel primo caso, dato un intervallo di tempo t, diciamo, quattro secondi, siamo costretti ad 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, di 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, perchè 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.

2.8 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 (sotto forma di migliaia di *visiting thread*) di esecuzione 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 (i *parsing thread*). Si noti che, al di là delle questioni di politeness, 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 sarà possibile scaricare dal sito senza violare la politeness. Si tratta di una coda di min-priorità, dunque in cima alla coda c'è il minimo.
Per ogni sito manteniamo una coda (FIFO nel caso della visita in ampiezza). Quando degli URL vengono emessi dal crivello, vengono accodati alla coda del

sito cui appartengono.
Ogni flusso del crawler procede iterativamente nel seguente modo:

1. Estrae il sito in cima alla coda (eventualmente aspettando il tempo necessario a far sì che questo sia scaricabile).

2. Procede a scaricare una o più risorse.

3. Riaccoda il sito aggiustando il timestamp di "readiness" secondo la politica di gestione della politeness.

Se c'è un URL disponibile allo scaricamento, il sito deve essere già stato reso 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. Il punto è che la cima della coda è sempre scaricabile.
Questo meccanismo rende automatica l'esclusività del download tra flussi: gli elementi della coda agiscono come *token*, quando un elemento è in cima alla coda, il thread ha in possesso un token per scaricare da quel sito fino allo scadere del tempo.
Il costo della coda è logaritmico e l'aggiunta e la rimozione sono operazioni relativamente costose (al più polilogaritmiche) ma possono diventare problematiche in momenti di concorrenza intensa.
Infine, nel caso sia necessaria una politica di gentilezza da applicare agli indirizzi IP possiamo organizzare gli IP in una lista come quella descritta sopra. Rimanendo lungo il solco tracciato dalla metafora del token: quando un thread trova in cima alla lista un IP e un URL significa che il thread è in possesso del token per scaricare i dati associati a quell'URL per quello specifico indirizzo IP per una quantità limitata di tempo.

References

[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*, pages 380–388, 2002.

[Ghe08] Jeffrey Dean Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[Naj99] Allan Heydon Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, pages 219–229, December 1999.

[O'N96] Patrick O'Neil Edward Cheng Dieter Gawlick Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[Sar07] Gurmeet Singh Manku Arvind Jain Anish Das Sarma. Detecting near-duplicates for web crawling. *In Proceedings of the 16th international conference on World Wide Web*, pages 141–150, 2007.

[Sin10] Felix Putze Peter Sanders Johannes Singler. Cache-, hash-, and space-efficient bloom filters. *Journal of Experimental Algorithmics (JEA)*, 14, 2010.