Survey

sugli algoritmi e sulle architetture

usate dai search engine

v: 1.1.2

Antonio Gulli
gulli@ideare.com

Survey sugli algoritmi e sulle architetture usate dai search engine

Le dimensioni del Web sono cresciute in modo ragguardevole. Da poche centinaia di web server disponibili nel 1994, abbiamo raggiunto nel 2002 diversi miliardi di pagine HTML. Assumendo una dimensione media di 5-10K per pagina e prendendo solamente in considerazione i contenuti testuali, abbiamo la necessità di trattare dati per diverse decine di terabytes. Studi effettuati da Lawrence [13] stimavano già nel 98-99 un tasso di crescita biennale pari al 100% per il Web. Questa tendenza è stata confermata, su un insieme campione, anche da [1] Garcia-Molina nel 2002. Il suo studio evidenza, inoltre, che la percentuale di pagine che cambia giornalmente sia pari al 40% nei domini .com e pari al 23% su tutto il campione di sampling.

I motori di ricerca sono universalmente ritenuti come lo strumento di accesso al Web. Le dimensioni dei data set raggiunte e la frequenza di cambiamento hanno dato vita ad una insieme sorprendentemente vasto di ricerche nei più disparati settori della Computer Science. Dall'Information Retrieval, rivista per adattarsi alle esigenze del web, alla Algoritmica, principalmente in funzione di un uso efficiente di strutture dati in memoria esterna, alle computazioni su Architetture Parallele e Distribuite, per garantire scalabilità ed efficienza alle soluzioni proposte (con particolare attenzione al costo realizzativi), alla Computazioni Numerica, per lo sviluppo di strategie efficaci di ranking dei documenti, allo sviluppo di Sistemi Operativi, in particolare modo per ciò che riguarda l'uso intensivo delle risorse macchina (memoria virtuale, socket, file system, CPU), alle sviluppo di architetture di rete (per connessioni ad elevata banda).

Nel seguito viene esposta una sintesi dei principali risultati conseguiti nelle suddette aree di ricerca. Per brevità altre aree non saranno prese in considerazione. Tra di esse le nascenti tecniche di analisi semantica, data-mining, latent semantic indexing, indicizzazione di sequenze genetiche ed altre. L'analisi si concentrerà sul modello di sviluppo seguito da Google. Questo motore è nato come un prototipo di ricerca all'interno di Stanford e, di conseguenza, molti dei risultati iniziali sono pubblici e sono serviti da stimolo per successive ricerche accademiche. Laddove noti pubblicamente verranno presi in considerazione anche alcuni elementi costituenti il motore di ricerca Altavista.

L'autore del presente articolo è stato il supervisore allo sviluppo di motori di ricerca Web per la search e la classificazione sia di dati testuali (pagine Web, PDF ecc) che di oltre 50 formati comprendendo Audio, Video ed Immagini.

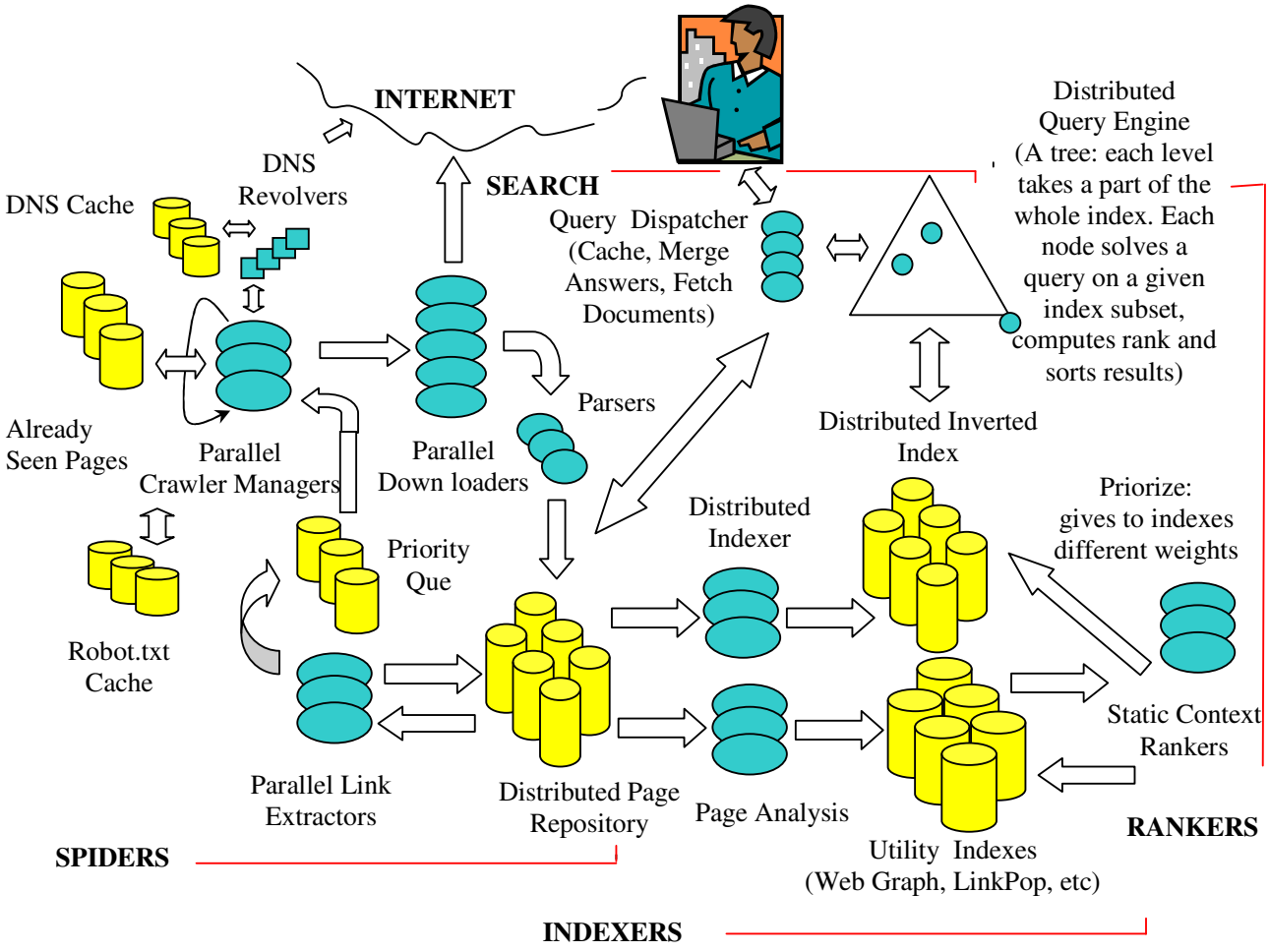
Architettura Generale

L'architettura generale di un motore di ricerca è descritta in figura 1. In giallo sono evidenziate le principali strutture dati usate, mentre in azzurro i moduli funzionali. Quasi tutte le strutture dati sono partizionate su una sottostante architettura con centinaia o migliaia di server connessi in rete. Per esempio Google usa circa 8.000 piccoli server del tutto simili ai dektop-PC. Circa il 10% usati sono usati per la parte di spidering, circa il 65% per rispondere alla search, il rimanente 25% per attività di indicizzazione e ranking.

Spider Module

Compito dello spider (crawler) è quello di raccogliere i documenti dal Web a partire da un set S_0 fornito in input. Da un punto di vista algoritmico questo task può essere modellato come la visita di un grafo orientato $G_{web}= (N_{web}, E_{web})$ con N_{web} pari all'insieme delle pagine Web presenti su Internet ed E_{web} corrispondenti ai link tra pagine. Più precisamente, in [14] è stato dimostrato che il numero di incoming link sul Web segue una distribuzione di Zipf ovvero che un ristretto numero di pagine ha un elevatissimo numero di link entranti, mentre la maggior parte delle pagine ha pochi

link entranti. Sempre in [14] si rappresenta un campione estratto dal web ricorrendo alla figura del "papillon" con un nucleo centrale di circa 50 milioni di nodi fortemente connessi, con circa 40 milioni di nodi in ingresso e circa 40 milioni di nodi in uscita (le ali del papillon). Vi sono poi circa altri 40 milioni di nodi debolmente connessi alle ali del papillon e svariate componenti con pochi link di connessione.

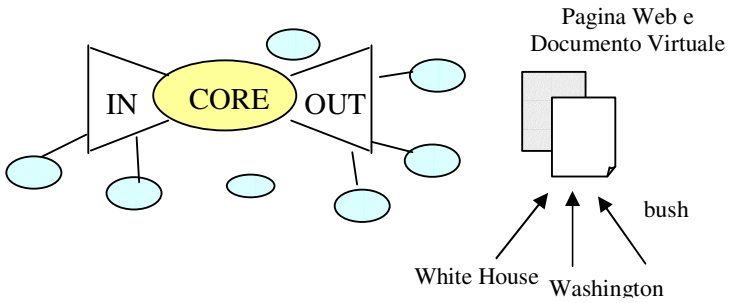


Uno spider visita il grafo servendosi di due strutture dati di appoggio "Already Seen Pages" una struttura che tiene traccia delle pagine già raggiunte ed una coda con priorità "Priority Que" usata per selezionare il prossimo set di pagine da visitare. Il processo va avanti senza sosta: i dati raccolti sono mantenuti all'interno del "Page Repository". A partire dalle pagine raccolte, nuove pagine sono visitate semplicemente seguendo i link estratti da quelle già conosciute. Durante la visita le pagine sono arricchite da vari tag. Tra gli altri un codice MD5, per il controllo di univocità, il tempo di ultima visita, etc. Si noti che spesso come insieme S_0 si adottano i link presenti nella Open Directory Project ODP [23] (pagine Web classificate a mano da editori umani). La ricerca in questo settore fornisce risposte alle seguenti domande:

1. **Quale politica adottare durante la visita del grafo?** Oltre alle tradizionali visite DFS e BFS, in [7] si suggerisce di privilegiare le pagine con maggior numero di link entranti o con più elevato PageRank (maggiori dettagli sul PageRank saranno forniti in seguito). L'idea è quella di visitare per prime le pagine più importanti. Sono state sviluppate anche delle politiche di similarità (analizzate meglio in seguito nel contesto del ranking) che "guidano"

lo spider verso "focus" di interesse specifici. La politica prescelta è implementata dai "Crawler Managers", mentre i "Dow loaders" forniscono i meccanismi per scaricare le pagine prescelte. Generalmente questi moduli si occupano anche delle necessarie operazioni di rete (DNS, HTTP, HTTPS, ecc) . Alcuni recenti studi sembrano evidenziare che per larghi data-set la politica BFS si comporta altrettanto bene di quella più onerosa basata su PageRank [26].

2. **Con quale politica aggiornare le pagine già visitate?** Vista l'elevata frequenza con cui si modificano le pagine sul Web, uno spider deve essere in grado di costruire dei modelli di aggiornamento che siano in grado di cogliere i cambiamenti repentini di tendenza. La politica prescelta viene realizzando alterando le priorità nella coda. Per esempio, si può supporre che ad ogni pagina ciascun downloader associ il tempo di ultima visita. Si noti che il protocollo HTTP permette di conoscere quando una pagina è stata modificata sul Web server remoto, senza doverla scaricare integralmente. La frequenza di aggiornamento viene quindi dinamicamente assegnata in funzione del numero di volte in cui la pagina è stata trovata modificata nelle precedenti visite.



3. **Come raggiungere un'elevata performance che permetta di scaricare miliardi di pagine Web da decine di milioni di Web server distinti?** Quasi tutti i motori di ricerca seguono un approccio distribuito. I link che puntano a nuove pagine da scaricare sono, di continuo, ripartiti su un cluster di server interconnesso in rete locale. Le politiche di distribuzione, decise dai Crawler Manager, tengono conto di problemi bilanciamento di carico sia sui nodi locali che sui web server remoti. Si cerca, inoltre, di minimizzare le comunicazioni intra-node. Queste sono comunque necessarie per mantenere condivise le due strutture dati "Already Seen Pages" e "Priority Queue". Le tecniche proposte sono tipiche delle programmazione parallela. In [8] ed [9] si propone la distribuzione delle URL secondo una funzione hash applicata alla stessa URL o al sito a cui l'URL appartiene. Nel primo caso, in [8] si suggerisce di minimizzare le comunicazioni necessarie a condividere la struttura "Already Seen" ricorrendo a batch update al prezzo di visitare, in qualche raro caso, più volte lo stesso nodo. Altra strategia utile è il caching in ciascun nodo locale delle URL più frequentemente visitate. In [9] si riporta come la percentuale di hits rate in cache raggiunga tranquillamente 85%.
   Se la funzione hash ripartisce per sito, le comunicazioni necessarie si riducono, al prezzo di un subottimale bilanciamento. Le strutture "Already Seen" e "Priority Queue" possono infatti essere in tutto od in parte localizzate sul singolo "Crawler Manager".
   Non necessariamente il sistema di spidering deve stare nello stesso posto del search engine, in [9] si suggerisce una ripartizione geografica che minimizzi il numero di border gateway attraversati. Google usa 4 data center. Dopo essere stati indicizzati, i dati raccolti sono scambiati tra le varie sedi per essere indicizzati.
   Ponendoci ad un livello d'astrazione alto, possiamo descrivere gli algoritmi usati dai vari moduli (Link Extractor, Crawler Manager e Downloader) nel modo seguente:

Link Extractor:
```
while(<ci sono pagine da cui estrarre i link>){
  <prendi una pagina p dal page repository>
  <estrai i link contenuti nel tag a href>
  <estrai i link contenuti in javascript>
  <estrai          …..
  <estrai i link contenuti nei frameset>
  <inserisci i link estratti nella priority que ciascuna con
    una priorità dipendente dalla politica scelta>
 <marca p come pagina da cui abbiamo estratto i link>
}
```

Downloaders:
```
while(<ci sono url assegnate dai crawler manager>){
  <estrai le url dalla coda di assegnamento>
  <scaricala le pagine p_i associate alla url dalla rete>
  <invia le p_i al page repository>
}
```

Crawler Manager:
```
<estrai un bunch di url dalla "priority que" in ordine>
while(<ci sono url assegnate dai crawler manager>){
  <estrai le URL ed assegnale ad S>
  foreach u ∈ S {
        if ( (u ∉ "Already Seen Page" ) ||
            ( u ∈ "Already Seen Page"  && (<sul Web server la pagina è più recente> )
          && ( <u è un url accettata dal robot.txt del sito>) ) {
           <risolvi u rispetto al DNS>
           <invia u ai downloaders, in coda>
        }
```

Chiaramente implementazioni specifiche possono ulteriormente raffinare questa descrizione di alto livello, usare forme di programmazioni multithread e comunicazioni bufferizzate per aumentare l'efficienza del sistema di spidering. Si noti che in questo caso abbiamo utilizzato una strategia "incrementale". Durante la visita, lo spider aggiorna le pagine se queste sono state già incontrate ma sono state modificate in remoto rispetto al tempo di ultima visita. Seguendo questa politica si può pensare di introdurre un tempo massimo di expire, superato il quale le pagine non più aggiornate sono automaticamente rimosse dal page repository e marcate come non valide nell'indice.

4. **Come garantire assieme alla performance la necessaria flessibilità e robustezza?** In [8] viene studiato l'overhead di comunicazioni e si conclude che con una programmazione multithread e con uso intensivo di I/O asincrono su socket le comunicazioni possono essere ammortizzate dalle elaborazioni di svolte dal modulo "Parser".
5. **Quali politiche di bilanciamento usare e come garantire la fault tolerance?** Il problema della fault-tolerance è stato studiato in diversi contesti. Recentemente è stata proposta soluzione interessate nel contesto del Web caching [37] che ha trovato applicazioni anche nel settore del Peer-to-Peer [25] e, recentemente, nel campo dello spidering [21]. I dettagli del consistent hashing teorico esulano dallo scopo del presente articolo e possono essere trovati in [36]. La necessita è quella di associare la semplicità di un mapping ottenuto attraverso l'uso di una funzione hash (di cui si conosce la distribuzione pressoché uniforme come MD5), con la flessibilità di potere aggiungere nuovi server senza dovere ri-mappare l'intero dataset. La stessa flessibilità è richiesta nel caso in cui un server per qualche motivo si rende indisponibile. L'idea innovativa è quella di normalizzare lo spazio delle chiavi nell'intervallo circolare [0,1] e di mappare gli stessi server nel medesimo intervallo (a ciascun server viene associata una chiave univoca). Se un nuovo server si rende disponibile, gli sono assegnati solo i dati "più vicini". Se un server fallisce, i suoi dati sono presi in consegna dai server a lui "più vicini". In nessun caso, tutti i dati distribuiti devono essere ri-mappati. In [36] si dimostra come costruire una simile funzione garantendo un buon bilanciamento, una buona distribuzione delle chiavi e come minimizzare le comunicazioni con delle repliche dei dati costruite a hoc su server mirror sempre distribuiti nell'intervallo [0,1].

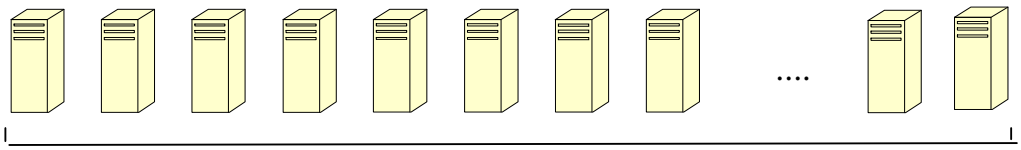
6. **Come trattare i documenti ipertestuali scaricati dalla rete?** In genere, ciascuno spider usa una libreria di parser scritti per estrarre informazioni testuali, dette "features" da differenti formati (HTML, PDF, postscript ed anche metatag da video, immagini ed audio). Il parser può memorizzare il contesto interno al documento in cui la singola parola è incontrata. Per esempio, nel linguaggio HTML il parser può ricordare che una particolare frase è stata incontrata all'interno dei tag <TITLE></TITLE>. Questa informazione sarà poi sfruttata dal modulo d'indicizzazione. Una possibilità sempre più frequentemente utilizzata è quella di memorizzare le pagine scaricate nel page repository direttamente in HTML, eventualmente preoccupandosi di trasformarle in questo formato se all'origine sono disponibili in formato differente. L'occupazione di spazio può crescere un pò, ma è possibile fornire una funzione di "cache" per tutte le pagine indicizzate se queste non sono raggiungibili nel Web server d'origine. La funzione di parsing è quindi spostata a valle del page repository e prima della fase d'indicizzazione vera e propria. Un ulteriore funzione che può essere richiesta al parser è quella di filtrare le parole rispetto ad una stop word list, di effettuare lo stemming ed il case unfolding.
7. **Come minimizzare l'impatto di carico sui Web server ?** Soluzioni standard sono politiche di throttling che introducono delle pause tra un download ed il successivo diretto e di politiche di "data scattering" che distribuiscono ad uno stesso crawler porzioni di siti distinti. E' compito dei crawler manager intervallare il download tra i vari siti. In questo modo si massimizza l'occupazione di banda locale e si riduce, nel contempo, il carico sui Web server remoti. E' buona norma, inoltre, seguire lo standard Robot exclusion che permette ad ogni amministratore di Web server di indicare, a ciascuno spider, cosa scaricare e cosa non prendere in considerazione. Per ogni sito queste informazioni sono disponibili in un file "Robot.txt" che è cachato dagli spider.
8. **Come sfruttare le informazioni ipertestuali presenti nelle pagine?** L'estrazione dei link oltre ad essere indispensabile per la navigazione può essere usata in modo indiretto per desumere il "contenuto" di una pagina a partire dal "contesto" in cui questa è stata posta. L'esempio classico è il seguente: Supponiamo di non avere ancora raggiunto la pagina whitehouse.org ma avere collezionato un considerevole numero di link che vi puntano e che ciascuno abbia associata all'anchor una piccola descrizione. Ecco che possiamo costruire una sorta di "documento virtuale" che affianca la pagina reale, se questa già presente, o la rappresenta, se non ancora raggiunta. Si noti che questo documento virtuale è potenzialmente d'elevata qualità perché rappresenta "il consenso su quello che la gente dice" intorno al sito whitehouse.org. Si stima che circa il 30-40% dell'indice di Google sia costruito ricorrendo a questa tecnica.
9. **Quando indicizzare i documenti?** In linea di principio lo spidering non si ferma mai e l'indicizzazione dei documenti potrebbe avvenire in parallelo alla raccolta. Per ragioni d'ottimizzazione delle risorse, tuttavia, spesso si preferisce adottare un update incrementale. L'indice è suddiviso in più partizioni che sono aggiornate a turno, sostituendo quelle precedenti. Questa strategia potrebbe introdurre dei vincoli al modo in cui le URL vengono partizionate sugli spider per non causare sovrapposizioni tra i dati contenuti nelle varie partizioni dell'indice. In Google, un ciclo completo di update impiega circa 28 giorni e molti ritengono questo un valore non accettabile.

La risposta a molte di queste domande non è ancora del tutto soddisfacente. Soprattutto se si considera che un moderno sistema di spider, che intenda collezionare due miliardi di pagine in una settimana, deve garantire un peak load di circa 3500 pagine il secondo distribuite per i vari nodi d'elaborazione. Il lettore interessato ad approfondire l'argomento può consultare anche le references [46] e [47] dove viene descritto in dettaglio il sistema di spidering di Altavista.

Storage Manager

Lo storage manager è centrale al motore di ricerca. E' acceduto sia dagli spider per memorizzare le informazioni recuperate dalla rete, sia dagli indexer per costruire gli indici, che dai query dispatcher una volta che i documenti significativi in risposta ad una query sono stati individuati. In figura 1 tutte le rappresentazioni di strutture dati fanno un uso implicito di questo modulo che non è espressamente disegnato. La ricerca in questo settore risponde a domande che hanno un impatto diretto con la realizzazione dei sistemi operativi. L'argomento può essere ulteriormente approfondito in [19] dove è descritto un prototipo del Web repository utilizzato da Google.

1. **Come storare in memoria esterna diverse decine di terabytes e garantire, al contempo, la scalabilità della soluzione?** In google hanno stimato che per ragioni di costo e di scalabilità è conveniente usare un approccio distribuito sempre basato su PC desktop-like, gli "storage nodes". Tale soluzione è stata preferita ad un approccio basato su soluzioni standard ma costose quali quelle RAID. La tolleranza ai guasti è garantita grazie al controllo software ed una parziale replicazione dei dati. Si richiede che ogni singolo nodo sia in grado di trattare, in memoria virtuale, file indirizzati a 64bit. La maggior parte delle strutture dati in memoria esterna possono difatti eccedere i 4Gb. Per esempio in [1], lo storage manager distribuisce le pagine Web scaricate dagli spider ricorrendo ad una ripartizione hash. Questa soluzione è adottata anche da Google.



HASH. es: (docID mod M), con M numero server

2. **Come sfruttare al meglio le gerarchie di memoria (memoria fisica, disk cache) durante le varie operazioni?** Le risposte a queste domande dipendono dalla politiche usate per la gestione delle strutture dati. In ogni caso il modulo deve garantire alcuni meccanismi di base come l'accesso efficiente in modalità random (per esempio per updatare una singola pagina Web) o in modalità stream (per esempio per indicizzare un bunch di pagine Web), o come l'update e la cancellazione. In [1], sempre per il "Page Repository" i dischi sono visti in modalità log, ovvero ogni nuovo dato è semplicemente appeso ai precedenti (spesso in modo compresso) e l'accesso avviene per chiave ricorrendo ad un soprastante β-albero. Con questa soluzione, la lettura sequenziale dal disco (per esempio in indicizzazione) sfrutta al meglio le gerarchie di memoria. Altri sistemi usano esclusivamente un β-albero per memorizzare le chiavi ed i valori. Infine un'altra architettura spesso usata è quella hashed-log in cui ad ogni chiave dell'hash corrisponde un valore organizzato come una struttura a log. In [19] si confrontano le performance di queste strutture alla luce di differenti parametri, mentre in [11] sono analizzate varie tecniche di batch update e d'in-memory sorting che riducono il costo computazione delle operazioni in memoria esterna sia bufferizando le richieste sia riordinandole, dove possibile, per ridurre i tempi di seek sul disco.

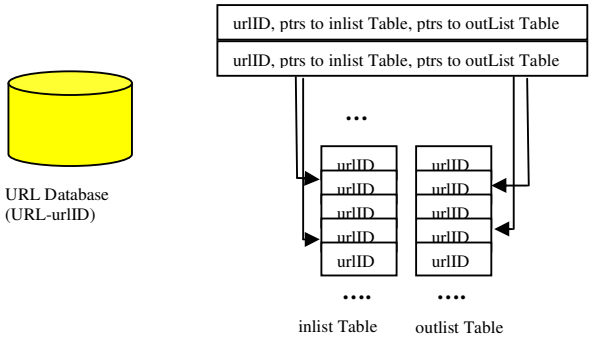
Si noti che questo settore di ricerca si collega direttamente con la realizzazione dei file system e dei network file system su cluster di workstation con elevato numero di nodi. Per esempio il sistema WebBase [19], realizzato sopra il tradizionale file system di Linux, è capace di processare sino a 2800 pagine/sec in streaming , 33 pagine/sec in accesso random, 3200 pagine/sec per addition con

buffer, generando solo un 1% d'overhead sul disco per strutture dati catalogo rispetto ai dati memorizzati.

Page Analysis Module

Le pagine collezionate dallo spider sono periodicamente analizzate dal modulo "Page Analysis" per estrarne informazioni significative. La fase d'indicizzazione vera e propria avviene in parallelo. Le domande a cui la ricerca cerca di rispondere sono le seguenti:

1. **Come rappresentare il grafo G_{web} costruito incrementalmente dagli spider?** Molti studi [4][6] evidenziano che una pagina contiene in media dai 5 ai 10 link uscenti (secondo cosa viene o meno filtrato). Di questi solo 1 o 2 inter-dominio. Anche escludendo i link intra-dominio, i cgi-bin, i link pubblicitari che hanno un basso contenuto informativo, è necessario sviluppare strutture dati su memoria esterna che siano in grado di scalare efficacemente con la dimensione del problema. Ogni singola URL in N_{web} deve essere memorizzata in modo compresso, essendo stimata $| N_{web} | = c*10^9$, ed $E_{web}= d * | N_{web} |$ con c pari a 2-3 e d pari a 5-10. Il problema è quello classico in Algoritmica di rappresentare un grafo fortemente sparso e con un elevatissimo numero di nodi. Chiaramente le strategie utilizzate dipendono strettamente dal tipo di "query" che si vuole effettuare su questo tipo d'indice. Per esempio, Google rappresenta semplicemente ogni singola URL con un intero, assegnato sequenzialmente, e gli archi come coppie (INT, INT). Questa è una rappresentazione efficiente per il successivo calcolo del PageRank. Una soluzione diversa è stata proposta in [15]. La struttura dati usata per memorizzare le URL è rappresentata in figura. Un database è usato per memorizzare le corrispondenze tra URL ed urlID univoci. Per ogni URL sono memorizzate i link entranti ed uscenti. Probabilmente uno degli esempi più significativi di modulo di analisi delle pagine noto in letteratura è il Link Database, correntemente usato in produzione da Altavista, e descritto in [45]



2. **Come utilizzare al meglio le informazioni "contestuali" contenute nelle pagine?** Abbiamo già visto che ad ogni pagina è possibile associare un "documento virtuale" costituito dalle finestre di testo intorno alle diverse "anchor" dei link entranti in una pagina. Secondo lo stesso principio queste informazioni possono essere usate, oltre che per l'indicizzazione di pagine Web, anche per il trattamento d'immagini, formati audio e video semplicemente assumendo che l'unione, filtrata, delle finestre di testo sia una descrizione dell'oggetto riferito. La tecnica è stata adottata sia da Google che da Altavista. Sono state poi sviluppati software d'estrazione di thumbnail, still-frame e meta-informazioni dai vari formati che esulano dallo scopo di questo articolo.

E' probabile che la ricerca fornirà risposte sempre più efficienti alle precedenti due domande in considerazione della crescita dei data set analizzati. Per ragioni di brevità non abbiamo preso in

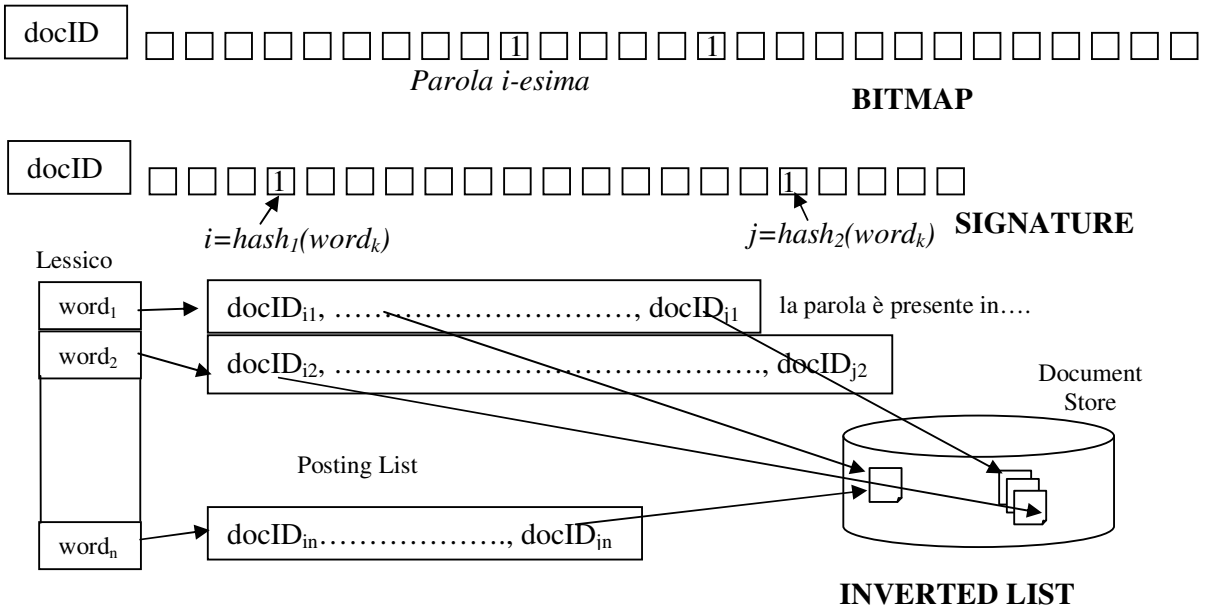
considerazione tecniche di data-mining e d'analisi semantica dei documenti che può essere applicata in questa fase o in cascata all'indicizzazione vera e propria.

Indexer Module

Una delle componenti più importanti del motore di ricerca è quella che svolge la funzione d'indicizzazione dei documenti. Questa fase è successiva alla raccolta dei documenti dalla rete e precedente alla fase di search vera e propria. Le domande poste sono le seguenti:

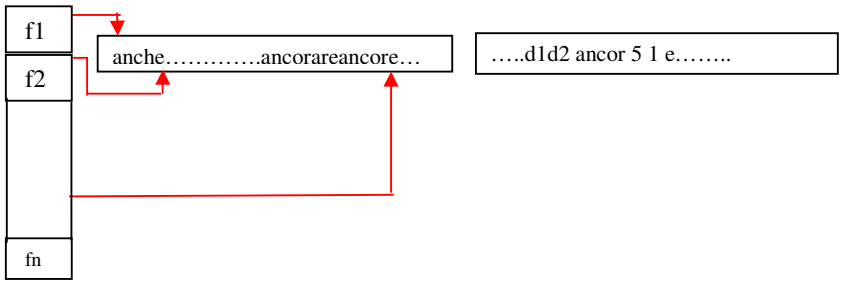
1. **In che modo costruire strutture dati efficienti che garantiscano velocità nelle risposte alle query e, nel contempo, una ridotta occupazione di spazio?** Sono state proposte molte soluzioni, ciascuna delle quali ottimizzata per il tipo di query specifico che si intende soddisfare[40]. Noi prenderemo in considerazione le tre più comuni per i motori di ricerca che sono caratterizzati da query booleane con lunghezza media 1.2 e, nella maggior parte dei casi, non superiore alle 3 keywords [31]. Il lettore interessato può trovare interessati le strutture dati alternative descritte nelle reference [42] e [43]

    - La prima è basata su bitmap. Supponiamo di avere un insieme di mappe di bit, ciascuna lunga quanto il numero di parole distinte contenute nella collezione da indicizzare. Ecco che il singolo documento può essere rappresentato da una bitmap in cui il bit *i*-esimo è settato se la corrispondente parola è presente nel documento. Si noti che le bitmap prodotte sono molto sparse e quindi potenzialmente comprimibili con discreti risultati. Un'interrogazione trasforma la query in una bitmap ed individua le bitmap che soddisfano la ricerca con operazioni booleane sui bit.
    - Il problema con le bitmap è l'occupazione di spazio. La tecnica basata su "signature" prova a superarlo, utilizzando funzioni hash. Per creare una descrizione del documento, ogni singola parola è inviata a diverse funzioni hash. Ciascuna di queste setta una bit sulla mappa di bit che costituisce la signature. In questo modo l'occupazione di spazio si riduce. In fase di fetch è introdotta una probabilità non nulla di introdurre casi di "false match" che possono essere risolti soltanto attraverso una scansione lineare dei documenti selezionati.
    - Quasi tutti i motori di ricerca Web, come pure i sistemi di Information Retrieval più recenti, usano la tecnica delle "liste invertite". In [17] Moffat dimostra sia analiticamente che con prove sperimentali la superiorità di questa tecnica rispetto alle prime due. Con il termine "liste invertite" si indicano tre strutture dati distinte. Il *lessico* contiene la lista ordinata di tutte le keywords presenti nella collezione indicizzata e, per ogni parola, un puntatore alla rispettiva posting list. La *posting list* è una struttura a log. In essa sono salvate una dietro l'altra singole *liste invertite* che, per ogni parola, memorizzano i documenti in cui le stesse parole appaiono. Facoltativamente, all'interno di una posting list possono essere memorizzate delle *hit list* ovverosia, fissato un documento, la lista di tutte le posizioni all'interno di questo in cui la parola occorre. Il *document store* memorizza i documenti assegnando a ciascuno un docID unico. Il *document store* usa tecniche standard di memorizzazione. Per esempio, in Google il document store è una struttura a log, in cui i documenti sono semplicemente appesi uno dietro l'altro compressi e i puntatori presenti nelle posting list permettono un accesso random. Vedremo che le liste invertite sono particolarmente efficienti per risolvere query con operatori booleani.



2. **In che modo ridurre lo spazio occupato da una lista invertita?** Questa domanda è stata oggetto di studi intensivi e sono state proposte molteplici ottimizzazioni. Le liste invertite possono occupare dalle decine di gigabyte alle decine di terabytes. E' importante sottolineare che i tempi di accesso in memoria fisica possono essere sino ad un milione di volte inferiori ai tempi di accesso al disco. Tutte le tecniche sono quindi orientate ad uso ottimale delle gerarchie di memoria:

   - **Lessico:** per ragioni di efficienza il lessico dovrebbe essere mantenuto in memoria principale, ordinato per chiave, in modo tale che su di esso si possano effettuare delle ricerche binarie. Google adottava [2] una soluzione molto semplice in cui ogni parola è rappresentata da un Word di 24bit. E' probabile che adesso sia usata una rappresentazione più efficiente. Le parole di un linguaggio hanno dimensione variabile. Per questa ragione rappresentare il lessico come un array di dimensione $num_words * \max_word_lenght$ sprecherebbe un numero elevato di bit per ciascuna parola.



   Una rappresentazione più efficiente è data in figura. Il lessico contiene una serie di puntatori ad una struttura a log in cui le parole sono memorizzate una di seguito all'altra. Utilizzando un livello di de-referenziazione la ricerca usata sul lessico può essere sempre quella binaria.
   Inoltre, si osservi che spesso parole consecutive in un elenco ordinato condividono lo stesso prefisso. Per esempio le parole "ancor-are" ed "ancor-e" condividono il prefisso comune "ancor". Possiamo sfruttare questa caratteristica per ridurre l'occupazione di spazio. Assieme a ciascuna parola sono quindi memorizzati due interi. Il primo indica quanti caratteri prefissi sono condivisi con la precedente parola ed il secondo quanti caratteri costituiscono il suffisso della presente parola. Per

esempio, per rappresentare “ancor-e” subito dopo “ancor-are” usiamo (5,1,e). Questa codifica è chiamata front-coding ed è estremamente compatta. Il problema con questo tipo di rappresentazione è che essa non consente una ricerca binaria diretta perché per conoscere il contenuto dalla *i*-esima sezione è necessario decodificare la sezione *i-1*. La soluzione proposta è quella di applicare un k-front-coding, ovvero ogni k posizioni lasciare memorizzata la parola non codificata. Lo spazio viene, di conseguenza, diviso a blocchi. Tra i vari blocchi è possibile applicare la ricerca binaria e una volta identificato il blocco questo è scandito linearmente.
Sono ovviamente possibili altre rappresentazioni ricorrendo a tabelle hash o alberi di ricerca. Per esempio Google dichiara di usare un hash con puntatori ad un array lineare che memorizza le parole una di seguito all’altra. In genere il lessico è storato sul disco e caricato in memoria durante la search. Un’ottimizzazione è quella di cachare in RAM solo i termini maggiormente cercati ricorrendo ad algoritmi di sostituzione quale per esempio LRU, al costo di un maggiore tempo di ricerca.
Una nota: il front-coding può avere molteplici applicazioni. Per esempio, Altavista lo ha usato nel suo Connectivity Server [15] per rappresentare in modo compatto le URL nel database (URL, urlID). Le URL sono mantenute ordinate secondo l’ordinamento lessicografico e utilizzando il front-coding l’occupazione in spazio scende del 70%.

- **Posting List**. la posting list può occupare una porzione significativa della lista invertita. Per ogni parola è necessario memorizzare sia il documento che la posizione nel documento in cui questa occorre. Supponendo di rappresentare a 32 bit unsigned i docID (per un massimo di poco più di 4miliardi di documenti distinti) ed a 16 bit unsigned le posizioni all’interno del documento (per un massimo di 65335 posizioni), si comprende che una realizzazione naive delle posting può produrre un’occupazione in spazio eguale o superiore a quella degli stessi documenti da indicizzare.
  Fortunatamente una semplice osservazione permette di risparmiare molto spazio. Nella posting i riferimenti ai documenti possono essere memorizzati secondo un qualunque ordine. Scegliamo quindi quello ascendente, dal più piccolo al più grande docID. Vedremo che questa scelta produrrà molteplici benefici. Secondo lo stesso principio memorizziamo le occorrenze all’interno di un documento in ordine crescente. L’idea chiave, è che se i docID sono ordinati per valori crescenti, allora si risparmia spazio se si memorizza in ogni posizione lo scarto tra il corrente docID ed il precedente. Questo strategia è detta delta-encoding.

  $d_1=3, d_2=14, d_3=16, d_4=17, d_5=18, d_6=20, d_7=23, d_8=26, \ldots\ldots\ldots\ldots$

  $d_1=3, d_2=9, d_3=2, d_4=1, d_5=1, d_6=2, d_7=3, d_8=3, \ldots\ldots$

  Chiaramente è necessario disporre di codifiche a lunghezza variabile per rappresentare le differenze. Uno studio completo sull’argomento è svolto da Moffat in [12]. Sempre Moffat in [17] dimostra che questa tecnica comprime sensibilmente le posting list che tipicamente occupano il 30-35% dello spazio totale delle pagine nel Document Store. Si noti che la compressione è tanto elevata tanto più la lista è densa di informazioni. Nel caso limite, se una posting list riferisce tutti i documenti è sufficiente un bit per indicare la differenza e la lista invertita diviene in tutto e per tutto simile ad una bitmap. Molti studi evidenziano che sul Web le posting list risultanti riflettono spesso una distribuzione di Zipf con pochi termini presenti in un numero elevatissimo di documenti e la stragrande maggioranza di termini presenti in

pochi documenti. Chiaramente il trade-off pagato per il risparmio di spazio è espresso in termini di risorse di computazione. In fase di search, si richiede che le posting list vengano decodificare sequenzialmente per ricostruire il reale docID a partire dalla differenza con il precedente.
In letteratura, tutti concordano sul fatto che raggiungere un equilibrio soddisfacente tra l'occupazione di spazio sul disco e l'aumento del tempo di servizio a query time sia un compito molto difficile.
Tra le molte tecniche di codifica a lunghezza variabili dei numeri ne analizziamo due. Il lettore può approfondire l'argomento in [12] e [51]. L'idea è quella di codificare le parole frequenti in modo breve e quelle meno frequenti con una rappresentazione un po' più lunga. Prima di tutto un po' di analisi statistica. Supponiamo di conoscere un'informazione globale *f* il numero di puntatori prodotti dalla inversione della collezione (il numero totale di coppie "documento, parola"). Sia *N* il numero totale di documenti ed *m* il numero totale di parole. Assumendo una distribuzione uniforme, il valore $p = \frac{f}{N * m}$ rappresenta la probabilità che una qualunque documento contenga un termine qualunque scelto casualmente. L'occorrenza dei puntatori può essere vista come una processo di Bernoulli, dove gli *f* puntatori siano selezionati dalle $N * m$ coppie "parola-documento". Sotto queste assunzioni la possibilità di avere una distanza *x* è la probabilità di avere *x-1* non occorrenze di una particolare parola, seguita da una occorrenza della parola ovverosia $(1-p)^X p$. In statistica questa è la distribuzione geometrica corrispondente ad *x* prove di Bernoulli con probabilità *p.* Le probabilità cumulative sono $\sum_{i=1}^{x-1}(1-p)^{i-1} p = 1-(1-p)^{x-1}$ e $\sum_{i=1}^{x}(1-p)^{i-1} p = 1-(1-p)^{x}$. Adesso notiamo che, dato un parametro intero *b*, ogni numero x>0 può essere rappresentato su *base unaria* (sequenza di uno) come *q+1*, essendo *q* quoziente pari a $\left\lfloor \frac{x-1}{b} \right\rfloor$. Il resto $r = x - qb - 1$ è rappresentato su *base binaria* in $\lfloor \log b \rfloor$ bit. Per esempio se *x*=11 e *b*=3 allora q=(11-1)/3=3 che è rappresentato come 111_1 seguito da 11_2, essendo r=11-3*3=2. E' stato dimostrato in [12] che scegliendo *b* in modo tale da soddisfare $(1-p)^b + (1-p)^{b+1} \le 1 \le (1-p)^{b-1} + (1-p)^b$ si generano delle rappresentazioni ottimali ed in media il numero di bit necessario a codificare la lista invertita è minimizzata. Si dimostra che $b = \left\lceil \frac{\log(2-p)}{-\log(1-p)} \right\rceil \cong \frac{\log_e 2}{p} \cong 0.69 * \frac{N * m}{f}$ è una soluzione per l'equazione precedente. Il metodo descritto è noto come modello *Global Bernoulli.* Dato *x* e scelto *b*, il quoziente è codificato in base unaria ed il resto in base binaria.
Un altro metodo sfrutta la conoscenza di una informazione locale ovvero: sia f_t la frequenza del termine *t* (questa informazione è locale alla posting list) e riapplichiamo tutto quanto detto sino adesso. Ecco che otteniamo il metodo *Local Bernoulli.* In questo caso, se la parola è molto frequente, è facile dimostrare che si ottiene b≈1 e la posting list diviene simile ad una bitmap, come avevamo già ipotizzato.
Il delta-encoding è una strategia che ha una larga applicazione nei motori di ricerca non solo a livello di indicizzazione. Per esempio, Altavista la usa nel proprio Link Database per codificare le urlID nelle liste di adiacenza che rappresentano i link entrati ed uscenti da una pagina [45].

Concludiamo, notando che questa rappresentazione è solo una tra le molteplici utilizzabili e che la giusta scelta dipende dal tipo di applicazione che si intende realizzare per le posting list e dai compromessi in termini di risparmio dello spazio disco e consumo di CPU che si è disposti ad accettare. Infine, facciamo notare che è oramai una scelta comune quella di accedere la posting list memorizzata sui dischi con cached I/O (es. Google) per sfruttare i benefici della località nelle strutture dati tra query successive.

- **Document Store:** Come già detto il document store può essere una struttura dati gestita come un log appendendo le pagine una dopo le altre in modo compresso. Essa non è generalmente caricata in memoria RAM, ma è acceduta sul disco quando necessario.

3. **Come costruire efficientemente una lista invertita?** Una trattazione completa dell'argomento è riportata da Moffat in [12]. Partiamo dall'algoritmo naive. Supponiamo di rappresentare la collezione da indicizzare come una matrice A di dimensione *n*m*. Per riga abbiamo gli *n* documenti. Ogni colonna rappresenta una specifica parola estratta dall'insieme delle *m* parole distinte presenti nella collezione. La posizione a_{ij} rappresenta il numero di volte in cui la parola rappresentata dalla colonna *j* appare nel documento *i.* Costruire una lista invertita equivale ad calcolare A^T, la matrice trasposta di A.

	word 1	word 2	word 3	word 4	word 5	word 6	…..
doc 1	**2**	**1**	**--**	**5**	**---**	**4**	
doc 2	**--**	**1**	**--**	**4**	**--**	**2**	
doc 3	**--**	**--**	**--**	**--**	**---**	**1**	
doc 4	**--**	**--**	**1**	**--**	**--**	**--**	
….							

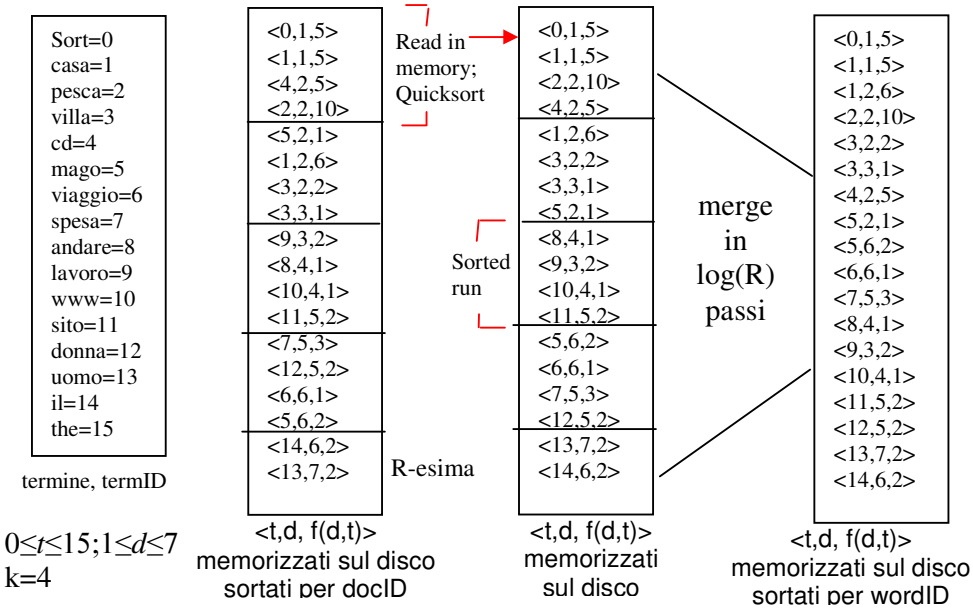
Sul Web la dimensione della matrice A è dell'ordine dei terabytes e non può essere mantenuta in memoria fisica. Anche ricorrendo a macchine con memoria virtuale, avremmo un numero di page fault elevatissimo e tempi di calcolo lunghissimi. La matrice è acceduta infatti per colonna ed è probabile di incorrere in un page fault per ogni accesso. Utilizzando questo metodo l'inversione della TREC (circa 2Gb) richiederebbe circa 2mesi di tempo su un desktop-PC. Sono stati suggeriti molti metodi alternativi, tra questi il "Sort Based Multi-Way Merge in-place" indicizza la TREC in meno di due ore su un desktop-PC!

L'idea è di suddividere il lavoro in più passi ed in ciascun passo sfruttare al meglio la memoria fisica disponibile. Per semplicità assumiamo che le posting list memorizzino solo i docID e non la posizione all'interno del documento della parola. La generalizzazione è facilmente ricavabile da quanto diremo. Si suppone di utilizzare una struttura a dizionario *S* (per esempio un albero binario) che memorizzi i termini già visti. Inizialmente, il testo è suddiviso in parole e memorizzato sul disco (memoria esterna) usando un file temporaneo. Per ogni termine, si salva sul file la tripla $< t, d, f(d,t) >$ contenente il termine *t* presente nel documento *d* in numero di volte pari ad *f(d,t).* Se il termine $t \notin S$, allora *t* è inserito in S.

1: Crea una dizionario S; Crea un file temporaneo vuoto sul disco
2: per ogni documento *d* nella collezione da indicizzare 1≤d≤N
  2.1: leggi il documento *d* e suddividilo in termini
  2.2: per ogni termine $t \in d$
    calcola *f(d,t).*
    Se $t \notin S$ allora insert(S, t).
    Scrivi il record <t,d, f(d,t> sul disco appendendolo al file temporaneo

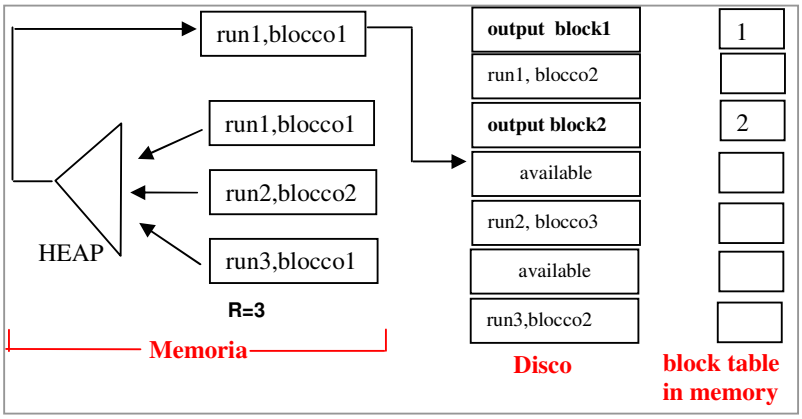
(*t* può essere rappresentato dal term number in S)
3: Sia *k* il numero di record che si può mantenere in memoria
3.1 repeat
leggi *k* blocchi dal file temporaneo
Quicksort per *t* non decrescente ed in caso di *t* uguali per *d* non decrescente.
scrivi i k blocchi (sorted run) sul file temporaneo
until blocchi disponibili.
4. Merge le *R* sorted run considerandole a coppie con una scansione lineare in log(R)
5. Per ogni termine 1≤t≤n contenuto in S
inizializza una nuova sezione della posting list, appesa alle precedenti
salva *t* nel lessico, salva il puntatore alla nuova sezione nel lessico
leggi sequenzialmente tutte le triple <t,d, f(d,t)> e scrivile nella nuova sezione
(se serve comprimile, prima della scrittura)

Al passo successivo, Il sorting di queste triple è fatto ricorrendo ad un merge sort su memoria esterna. In pratica, i files sono letti a blocchi di *k* (essendo *k* il numero massimo di record che si può mantenere in memoria RAM) e ciascun blocco è ordinato in memoria fisica ricorrendo ai normali algoritmi d'ordinamento quali il Quicksort. Il risultato è nuovamente scritto sul file temporaneo in blocchi detti "sorted runs". Il passo finale è il merge delle sorted runs, considerandole a coppie con una scansione lineare di ogni blocco costituente (in *log(R)*, con *R* numero di sorted run). Il risultato finale è quello di ordinare i record per chiave primaria non decrescente *t* e nel caso di *t* uguali per *d* non decrescenti. Vediamo un esempio in figura.



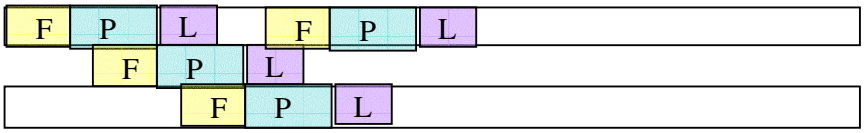
Utilizzando questo algoritmo il tempo di completamento per indicizzare la TREC scende a 20 ore. Notiamo che lo spazio disco occupato è notevole. Se prendiamo in considerazione, per esempio, l'ultima passata delle merge sul disco possiamo verificare che siano necessari due file. Uno contenente le due meta ordinate, l'altro che conterrà il risultato finale. Di conseguenza, l'occupazione del disco può essere addirittura superiore a quella dei documenti da indicizzare.
Vi sono due possibili ottimizzazioni. Si noti che possiamo risparmiare spazio sul disco salvando nel file temporaneo ciascun docID come differenza dal docID precedente. Il trade-off è un aumento di carico di CPU. Per contro abbiamo un notevole riduzione dei tempi di trasferimento dei dati dal disco.

La seconda ottimizzazione è usare una operazione di merge a h vie in cui le sorted run sono prese in considerazione a gruppi di h alla volta. In memoria è mantenuta una struttura dati heap per individuare efficientemente la minima tripla $< t, d, f(d,t) >$ da produrre in output. L'heap viene, via via, caricato con le triple lette dal disco portando in memoria blocchi di dimensione b da h diverse sorted run. Ciascun blocco b_i può essere rimpiazzato, in memoria, da un nuovo blocco b_j quando l'ultima tripla associata al blocco b_i entra nell'heap. Il blocco b_i dal file temporaneo non serve più e lo spazio usato è disponibile per essere riutilzzato. Supponiamo che l'ultima parte di ciascuna sorted run sia estesa con una operazione di padding alla dimensione b e che, analogamente, il blocco di output prodotto dall'heap sia esteso con una operazione di padding alla dimensione b. Possiamo pensare di gestire il file temporaneo in modo analogo alla tecnica di paginazione in memoria virtuale. Il blocco in output sostituisce quindi sul file temporaneo il blocco in input non più usato. Abbiamo quindi bisogno di una tabella di blocchi in memoria che ci dica quale è la posizione di ogni blocco logico nel file temporaneo / di output ordinato. All'inizio abbiamo h caricati in memoria e quindi h blocchi disponibili sul file. Di conseguenza, h blocchi in output possono andare direttamente sul file temporaneo. Questo libera almeno un blocco ed il processo può continuare sino ad esaurire i blocchi presi in input ed avere svuotato l'heap. In figura un esempio:

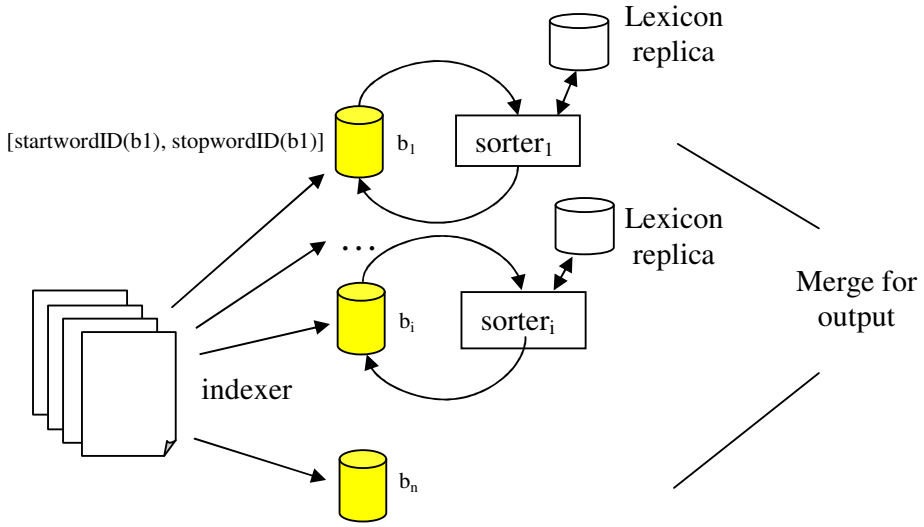


Alla fine, il file è pseudo ordinato. Ciascun blocco lo è al proprio interno e la tavola dei blocchi ci permette di ricostruire il loro ordinamento. Usando due buffer di appoggio di dimensioni b in memoria e la tabella dei blocchi, possiamo permutare in tempo lineare il file temporaneo ed ottenere la posting list completa. Lo spazio rimasto inutilizzato alla fine può essere rilasciato troncando il file.

4. **Come costruire efficientemente una lista invertita sfruttando la programmazione multithread?** L'intera indicizzazione può essere ottimizzata ricorrendo ad una pipeline software come suggerito in [1]. La computazione è suddivisa in tre stadi. Nel primo il documento è letto e parserizzato. Nel secondo le posting sono ordinate in-place, prima per termine, poi per locazione usando il quicksort. Nel terzo le sorted run prodotte sono salvate nel disco. Il processing (parsing, quicksort, ..) è ovviamente CPU-bound è può essere fatto da un thread separato rispetto all'attività I/O bound svolta dal flushing. Supponendo di avere un flusso continuo di documenti da indicizzare lo schema delle dipendenze temporali è riportato in figura:



5. **Come costruire efficientemente una lista invertita in modo distribuito su una rete di workstation?** Esistono due diversi approcci: il partizionamento del data-set **word-based** ed il partizionamento del data-set **document-based**.
   Nel 1998, Google usava un partizionamento word-based. L'indicizzazione è fatta in due passi. Nel primo passo un processo "indexer" legge il testo e ripartisce le triple $< t, docID, f(docID,t) >$ su n file separati detti "barrels" (In [2] si suggerisce n=64). A ciascun barrel b_i, $1 \le i \le n$ viene assegnato il range $[\text{startwordID}(b_i), \text{stopwordID}(b_i)]$ di wordID. Ogni t viene trasformato nel corrispondente wordID(t). Il barrel b_i riceve la tripla $< wordID(t) - \max_ wordID(b_i), docID, f(docID,t) >$ in base all'intervallo a cui $wordID(t)$ appartiene. La componente $wordID(t) - start_{wordID}(b_i)$ serve per risparmiare spazio rappresentando $wordID(t)$ come differenza dall'inizio dell'intervallo $start_{wordID}(b_i)$. I barrels prodotti dall'indexer sono già sortati per docID, con la condizione aggiuntiva che ciascun barrel contiene solo specifici range di wordID. Da questo punto in poi, ciascun barrel può essere trattato in parallelo ed in modo indipendente dai rimanenti. Nel secondo passo, i barrels sono ripartiti su server distinti. Su ciascun server gira un processo "sorter" che prende i barrel e li ri-ordina per wordID usando le tecniche già viste. Il problema con questa tecnica è che non è possibile inserire un nuovo documento se non re indicizzando l'intera collezione. Difatti, ogni nuovo documento potrebbe potenzialmente contenere parole da assegnare a tutti gli n barrels usati.

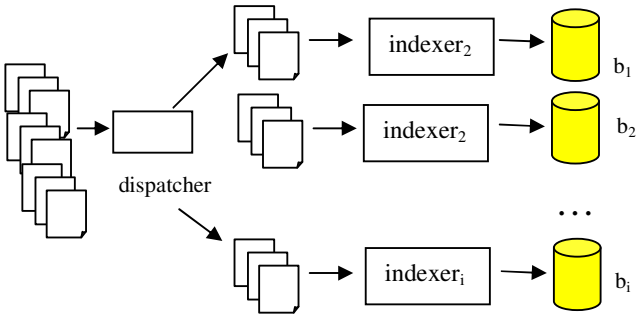


$$< wordID(t) - \max_ wordID(b_i), docID, f(docID,t) >$$

Inoltre, è necessario condividere l'indice durante la fase finale realizzata dai sorter per gestire parole non incontrate prima. Google dichiarava di usare un piccolo trucco. Un lessico iniziale di 14 milioni di parole viene replicato su ciascun sorter. Ciascun sorter lavora quindi in parallelo e produce un log file delle nuove parole incontrate. L'insieme di tutti i log file viene usato nella fase finale di merge degli indici prodotti in parallelo dai sorter. Google dichiarava di usare due set di inverted barrels. Uno, detto short

barrels, con le posting corrispondenti a parole situate in un anchor o nel titolo, ed un altro, detto long barrels, per tutte le rimanenti parole. Le ricerche sono effettuate sullo short barrels e se non vi sono sufficienti risultati sul long barrels. Il meccanismo equivale ad assegnare un peso implicito alle diverse collezioni.

Il metodo di partizionamento **document-based** si rivela più flessibile ed è probabile che lo stesso Google lo abbia adottato da un certo punto in poi. Il principio è molto semplice. L'insieme dei documenti viene suddiviso in diverse collezioni, ciascuna indicizzata in parallelo ed indipendentemente alle altre (non è necessaria alcuna condivisione di strutture dati).



I vantaggi sono molteplici. Per esempio la suddivisione può essere fatta sfruttando il giudizio statico dato dal PageRank (di cui parleremo estesamente a breve) e creando indici con "livelli di qualità" crescenti. Durante la search si interrogano gli indici uno dopo l'altro arrestandosi non appena si collezionano un numero di risultati sufficienti. Il partizionamento può essere sfruttato efficientemente anche dagli spider. Ciascuna partizione è aggiornata ed indicizzata con frequenza dipendente dal suo "livello di qualità".
Un problema che si presenta con la strategia di partizionamento "document-based" è tenere traccia delle informazioni statistiche globali alla collezione da indicizzare. Per esempio supponiamo di volere usare le misure di rank TF*IDF (di cui parleremo a breve). Notiamo che la componente IDF possa essere calcolata solo dopo avere analizzato tutte le distinte partizioni. Sono state proposte molte soluzioni. La più semplice pone $IDF=IDF_{partizione}$, ovverosia approssima l'IDF dell'intera collezione con quello calcolato localmente alla singola partizione. Questo metodo funziona bene su poche partizioni di grandi dimensioni. In alternativa si utilizza un protocollo in due passi. Nel primo passo, ciascun indexer broadcasta a tutti i rimanenti i valori calcolati localmente e rimane in attesa della conferma di ricezione. Nel secondo passo, quando tutte le conferme sono arrivate, ciascun indexer aggiorna i dati locali sulla base dei valori ricevuti da remoto.
In sintesi, questa semplice strategia permette di esplicitare un buon livello di parallelismo sia a tempo di indicizzazione che a tempo di search.
Spesso in letteratura la strategia di partizionamento word-based è spesso chiamata "global inverted" mentre quella document-based "local inverted".
Infine, notiamo che i moderni motori di ricerca non aggiornano le collezioni in modo dinamico ma preferiscono costruirle from scratch, alternando le fasi di spidering a quelle di indicizzazione.

La ricerca nel settore dell'indicizzazione ha raggiunto dei risultati di riguardo, ma non si è certamente conclusa. I risultati già acquisiti devono continuamente essere confrontati con le esigenze di trattamento di dataset sempre più grandi, con particolare attenzione al trade-off tra

lospazio occupato ed il tempo necessario per processare le query. Alla fine del 2002, Google ha il più grande indice del Web. Esso è composto da circa 2,5 miliardi di documenti, pari tuttavia a meno del 10% del Web stimato.

Ranking Module

1. Within Document Ranking

Le tecniche di ranking in questa categoria attribuiscono un peso alle pagine Web in dipendenza dal loro contenuto. Questo è un settore di studio classico per l'Information Retrieval. La ricerca ha fornito delle risposte alle seguenti domande:

1. **Come capire se una singola parola all'interno di una pagina è molto importante?** L'idea è che una parola è molto importante se la sua frequenza è molto alta all'interno della pagina identificata, e se, contemporaneamente, la stessa parola è poco frequente all'interno dell'intero data set di documenti indicizzati. Analiticamente, detto TF(t,d) la frequenza del termine *t* all'interno del documento *d* ed IDF(t) l'inverso della frequenza del termine *t* all'interno di tutta la collezione, abbiamo che il peso associato è:

$$w(t,d) = TF(t,d) * IDF(t)$$

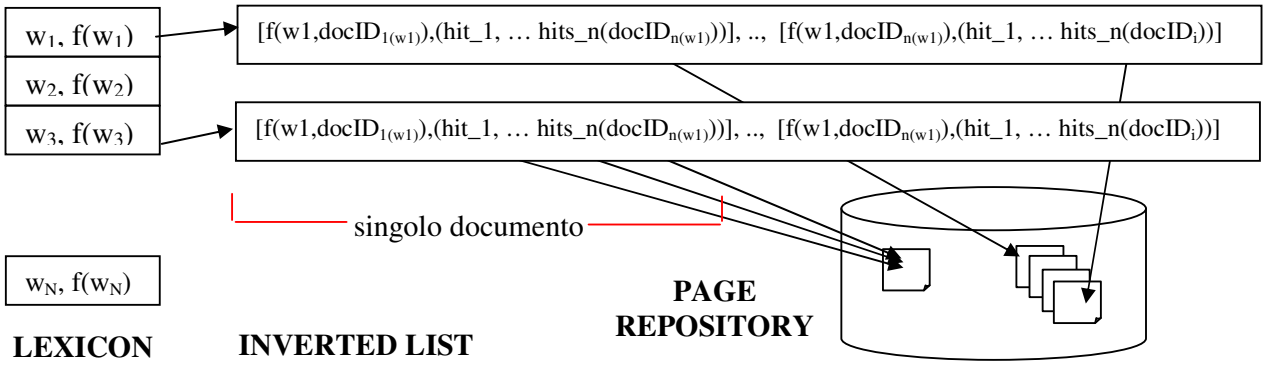
In letteratura sono stare proposte molteplici definizioni più o meno complesse per *IDF(t)*, ma tutte molto simili per bontà dei risultati forniti. Tra le altre:

$$IDF(t) = \frac{1}{f(t)} \qquad IDF(t) = \log(1 + \frac{N}{f(t)}) \qquad IDF(t) = \log(1 + \frac{f_{\max}}{f(t)})$$

dove *f(t)* è la frequenza del termine *t* nella collezione indicizzata, *N* pari al numero di documenti presenti nella collezione indicizzata ed f_{max} pari alla frequenza massima di un termine nella collezione. Analogamente abbiamo molteplici definizioni per *TF(t,d)*:

$$TF(t,d) = 1 \qquad TF(t,d) = f(t,d) \qquad TF(t,d) = \frac{f(t,d)}{lenght(d)} \qquad TF(t,d) = \log(1 + \frac{f(t,d)}{lenght(d)})$$

dove *f(t,d)* è la frequenza del termine *t* all'interno del documento *d*, *length(d)* la lunghezza del documento. Per velocizzare quanto più possibile il calcolo del rank in risposta alla query si memorizzando i valori necessari alla calcolo di TF*IDF all'interno delle posting list. Per esempio la *f(t)* è premessa alla *hit list*, mentre la *f(t,d)* rappresenta il numero di volte in cui il singolo termine appare in un documento. Per gli indici costruiti con granularità alla parola questo valore non è altro che la lunghezza della hit list per una dato termine ed una dato documento. Anche questo valore è memorizzato quindi in posting list. Generalmente il termine *length(d)* è usato per minimizzare l'impatto della lunghezza dei documenti. Tuttavia la sua memorizzazione non può agevolmente realizzata all'interno della posting list. Molti motori di ricerca [12] usano la rappresentazione seguente del lessico e della posting list associata. Le posting list possono essere ordinate per docID crescenti, e quindi potenzialmente compresse con le tecniche già viste, oppure ordinate per frequenze crescenti riducendo, come vedremo, il costo della selezione dei documenti al prezzo di una minore compressione delle stesse.



2. **Come dare maggior peso alle parole se queste appaiono molto vicine?** Data una query composta da molti termini $Q = \{q_1, ...q_n\}$ è chiaro che i documenti che contengono le parole cercate molto vicine sono potenzialmente più rilevanti di quei documenti che contengono le parole a distanza elevata. Nella figura precedente le posting list memorizzano, per ogni documento, la posizione in cui la parola occorre. Un algoritmo può quindi sfruttare queste informazioni e considerare finestre di testo sempre più grandi ed attribuire ai risultati un "bonus" secondo una funzione decrescente al crescere della lunghezza della finestra. Questo tipo di ricerche sono dette per "proximity" e la maggior parte dei motori di ricerca le supporta. Altre ricerche frequenti sono quelle per "phrase search", che può essere vista come un caso particolare di "proximity" in cui si richiede che ogni parola disti dalle altre di una singola posizione. Alcuni motori, adottano soluzioni di compromesso partizionando il singolo documento in blocchi di eguale lunghezza e memorizzando solo il numero di blocco nella posting list per ridurre l'occupazione in spazio e garantire in ogni caso una forma di proximity. Questa soluzione è adottata da Altavista. In sintesi, l'indice può avere una diversa granularità (al documento, al blocco alla parola).
3. **Come dare maggior peso alle parole se queste sono contenute all'interno di un particolare meta-tag?** Il linguaggio HTML, come tutti i linguaggi di markup, permette di definire delle sezioni di testo all'interno di particolari tag. Per esempio si potrebbe assumere che il testo contenuto dentro un <H1></H1> sia più importante del testo contenuto in un <H6></H6>. Per realizzare questa tecnica è necessario introdurre nella posting list una bitmap che salvi il tag di markup all'interno del quale la parola è stata incontrata in fase di indicizzazione. Per esempio con b bit possiamo rappresentare 2^b diversi tag. Si noti che questa tecnica può risultare utile non solo sul Web ma anche in altri campi. Per esempio per indicizzare in full-text i campi di una tabella contenuta all'interno di un database relazione.

La ricerca nel settore delle tecniche di ranking within document, va avanti da oltre 20 anni e possiamo dire che vi sono risultati consolidati che funzionano molto bene su collezioni di documenti omogenei, i cui contenuti siano rigorosamente controllati dagli autori originali (per esempio giornali, riviste, cd-rom).
Con lo sviluppo del Web è divenuto imperativo essere ai primi posti nei motori di ricerca. In fondo, la notorietà è un ottimo modo per accrescere le proprie possibilità di business!!
La prima generazione di motori di ricerca si basava esclusivamente su tecniche di ranking within document. Purtroppo, gli autori di pagine Web hanno cominciato a modificare il contenuto delle pagine per alterare arbitrariamente il rank. Per esempio, molte pagine "adult oriented" hanno introdotto decine o centinaia di keywords nascoste (non visualizzate dai browser) per essere segnalate come risposta anche in contesti del tutto diversi. La ricerca ha fornito delle risposte sviluppando nuove tecniche di rank, più oggettive, che descriveremo nel prossimo paragrafo.

2. Link Popularity Ranking

Le tecniche di analisi che cadono in questa categoria attribuiscono staticamente un peso alle singole pagine in dipendenza del "contesto" in cui le stesse sono posizionate. Le domande cui la ricerca vuole rispondere sono:

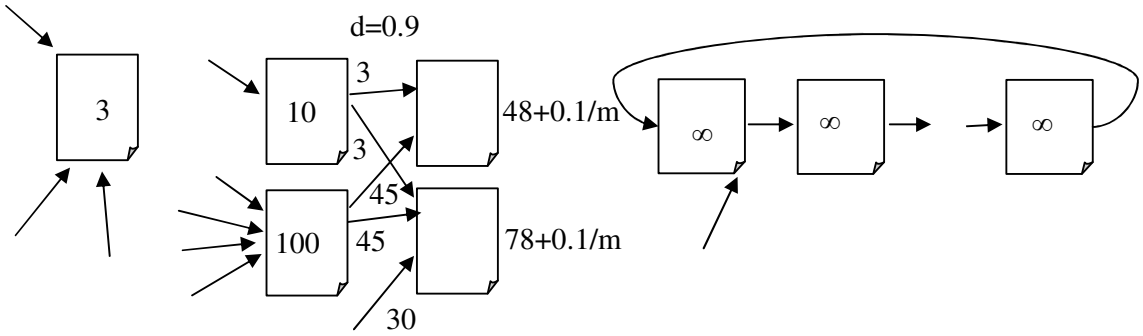
1. **Come assegnare un giudizio ad una pagina in modo indipendente dal suo contenuto e quindi con una oggettività maggiore di quella espressa dall'autore originario?** L'idea non è del tutto nuova. Nel campo della bibliografico un articolo è tanto più popolare quanto più è riferito. Ad esempio, per i database l'articolo in cui Codd descrive il modello relazionale è molto riferito da successivi lavori e quindi implicitamente molto importante. Applicando questo semplice principio, ad ogni singolo di nodo di N_{web} è attribuito un peso pari al numero dei link entranti.
   Sul web, in cui per principio non vi è alcun controllo sul modo in cui le pagine sono reciprocamente collegate, questa soluzione è soggetta a spam. Un individuo può arbitrariamente alterare i rank calcolati semplicemente avvalendosi di servizi, disponibili allo scopo sulla rete che creano ad hoc batterie di "fake-link". Anche con una attenta politica di filtraggio (link intra-site, link cgi-bin link reciproci quando scoperti) la tecnica può dare cattivi risultati.
2. **Come rafforzare il giudizio ricavato dal "contesto" per evitare che questo sia arbitrariamente alterato?** Anche in questo caso l'idea non è nuova. Supponiamo che un terzo soggetto scriva un lavoro sempre sul modello relazionale e questo sia in seguito riferito dallo stesso Codd. Ecco che automaticamente anche questo lavoro assume notevole importanza. Applicando il concetto al Web, stiamo definendo una relazione ricorsiva come la seguente:

$$r(i) = \sum_{j \in B(i)} r(j) / N(j) \qquad \text{(R.1)}$$

   Detto *B(i)* l'insieme di tutte le pagine *j* che puntano ad *i* ed *N(j)* il numero di link uscenti dalla pagina *j-esima*, la pagina *i*-esima riceve un rank *r(i)* pari alla somma dei contributi dati dai rank *r(j)* attribuiti alle pagine *j* in *B(i).* Si noti che, usando un principio democratico, ciascuna pagina *j* diffonde equamente la propria importanza lungo i link uscenti (r(j) / N(j)). Per il momento non prendiamo in considerazione i metodi iterativi e matriciali conosciuti per risolvere questa relazione ricorsiva ed occupiamoci di due piccoli problemi di natura tecnica. In figura 3.c abbiamo descritto il caso limite di un ciclo di pagine che ha un unico link entrante. Secondo il modello descritto, a ciascuna pagina è attribuito un peso sempre maggiore perché una volta entrati nel ciclo si continua ad iterare, attribuendo ad ogni passaggio peso sempre maggiore. L'idea per risolvere questo "problema della trappola" è quella di usare una costante moltiplicativa che continui a ripartire una parte del rank calcolato su una pagina su tutti i link uscenti, e ne riservi una parte su tutte le rimanenti pagine. Analiticamente otteniamo:

$$r(i) = d * \sum_{j \in B(i)} r(j) / N(j) + (1 - d) / m \qquad \text{(R.2)}$$

   essendo $0<d<1$ ed *m* la cardinalità di tutte le pagine analizzate. E' come se ogni singola pagina avesse una "uscita di sicurezza", a bassa priorità, che la collega a tutte le altre. In genere si sceglie $d = 0.9$. La precedente formula descrive il PageRank[5], l'algoritmo di ranking alla base del successo e dell'accuratezza di Google. Un ulteriore piccolo problema è quello delle pagine che hanno solo link entranti. Il loro contributo al calcolo iterativo del rank è nullo. Per questa ragione esse possono essere omesse inizialmente e re-introdotte, al costo di un ulteriore passo per diffondere i risultati ottenuti, una volta che il metodo iterativo è arrivato a convergenza.



Siamo di fronte ad un problema di Calcolo Computazionale che può essere risolto con le tecniche classiche offerte da questa branchia della Computer Science. La (R.1) può essere letta in forma matriciale[6] come $\lambda r = A^T r$ posto λ=1, avendo il vettore r = [r(1), ……, r(m)] ed $a_{i,j} = \frac{1}{N(i)}$ se esiste l'arco orientato (i, j), altrimenti 0. In modo analogo la (R.2) può essere vista in forma matriciale come $\lambda r = (1-d) * e + dA^T r$. Il PageRank corrisponde dunque al calcolo dei principali autovettori che corrispondono agli autovalori 1, per la matrice $A^{T.}$

Per il calcolo, Google usa il tradizionale metodo delle potenze ripetute partendo da un vettore iniziale che è ripetutamente moltiplicato per A^T sino a convergenza verso gli autovettori.

Scelto un ε>0 che rappresenta l'errore desiderato prima di arrivare a convergenza, l'algoritmo è il seguente:

```
foreach s { source[s]= 1 / m }
residual = 1;
while (residual > ε)
        foreach i, dest[i] = 0
        while (not links.eof()){
                Links.read(source, r, dest1, dest2, …. destr);
                for j = 1 … r do
                        dest[destj] = dest[destj] + source[source] / r
        }
        foreach i, dest[i] = d * dest[i] + (1-d)  / m
        residual = ‖ source – dest ‖   // eventualmente calcolato solo in alcune iterazioni
        source = dest;
}
```

Il vettore source è inizializato con una distribuzione uniforme all'inverso del numero di pagine *m* contenuto nell'intera collezione. In ogni istante contiene il valore di rank corrente. Il vettore dest contiene invece i prossimi valori di rank. Sino a quando il valore residual, dato dalla norma della differenza tra source e dest, non scende sotto una soglia si legge la struttura dati Links, usando i correnti valori di rank mantenuti in source, e si calcolano i prossimi valori di rank salvandoli in dest.

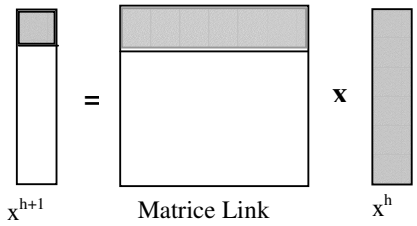
In [5] e [6] si dimostra come l'algoritmo converga in poche decine di iterazioni (tipicamente meno di 100), in particolare se sono rimossi i link intra-host. Dalla teoria di calcolo numerico sappiamo che la convergenza dipende dal gap tra il principale ed il secondario autovalore (nelle simulazione, la topologia di G_{web} produce un gap maggiore dell'1% e questo garantisce una veloce convergenza). Quando l'algoritmo arriva a convergenza abbiamo $A^T x^{(k)} = x^{(k*1)} = x^{(k)}$, il che significa che $x^{(k)}$ è un autovettore di A^T.

Una diversa soluzione di (R.2) può essere data interpretando la formula come un sistema di equazioni. In [6] si studia come risolvere il sistema sia con il metodo di Jacobi, che richiama il metodo delle potenze ripetute, sia con quello di Gauss-Siedel secondo la formula: $x_i^{(k*1)} = (1-\alpha) + \alpha \sum_{j<i} a_{ij} x_j^{(k+1)} + \alpha \sum_{j<i} a_{ij} x_j^{(k)}$ utilizzando sempre il più recente $x_i^{(k*1)}$ se j < i. Il metodo garantisce sufficiente velocità, al costo di un sort per riga della matrice A.
Al momento, non sono state ancora testati altri metodi di convergenza noti in letteratura.
Una diversa lettura della (R.2) può essere data modellando la visita del grafo G_{web} come un processo stocastico in una sorta di cammino randomico: Per ogni pagina l'utente seleziona, con probabilità *d,* uno dei link uscenti senza fare tra di essi distinzioni di sorta. Ogni tanto accade che l'utente "si stanchi" con probabilità (1-d) e salti verso una nuova pagina. Tale modello è noto come "random surfer". Il surfer compie un cammino randomico Markoviano, nel senso che ad ogni passo il comportamento viene deciso solo sulla base dello stato corrente e non delle decisioni prese ai passi successivi. Si può dimostrare che secondo questo modello, attribuire ad una pagina PageRank *r* equivale ad stimare la probabilità di visita di quella pagina. Intuitivamente la distribuzione di equilibrio per un cammino infinito è equivalente alla frazione di dei passi nel cammino che ritornano verso ciascuna pagina Web. Il concetto del "random surfer" è approfondito in [34] ed in [44].
Nel contesto dello spidering, il PageRank può essere anche usato come politica di selezione delle pagine.
Una piccola ottimizzazione recentemente introdotta in Google, è stata quella di assegnare alle pagine indicizzate dalla Open Directory Project (pagine che sono classificate a mano da editori umani) un peso più alto come se esse fossero già state riferite da altre pagine importanti (gli esseri umani, appunto). In [28] si dimostra che, sebbene il PageRank ed un rank molto più semplice basato sul conteggio dei numeri di link entranti, seguano la stessa distribuzione di probabilità $\frac{1}{r^{2.1}}$ i due rank hanno una bassa correlazione. In altre parole, le due unità di misure producono per la stessa pagina risultati diversi.

3. **Come calcolare efficientemente il PageRank su memoria esterna?** Nell'articolo di introduzione al PageRank [5], Larry Page riporta che a partire da un indice di 24 milioni di pagine e considerando una media di 10 link per pagina, per ottenere il calcolo del PageRank in 5 giorni è necessario garantire un peak load di 550 link per secondo. Questo nel 1998. Oggi con 2 Miliardi di pagine, sotto le stesse condizioni Google dovrebbe sostenere un sorprendente peak load di 55.000 link al secondo. Usando un approccio naïve potremmo rappresentare la matrice sparsa dei link come in tabella.

Source Id (32 bit)	**Out Degree (16 bit)**	**Dest Nodes (32 bit)**
0	5	5,12,1024,23456
1	1037	4,12,…,….
….	…	…

Le dimensioni di questa struttura dati possono raggiungere le diverse terabytes ed anche supponendo di lavorare su una macchina con memoria virtuale otterremmo tempi di completamento lunghissimi a causa dei continui fault derivanti dalla paginazione. Anche in questo caso, come durante la creazione delle liste invertite è necessario ricorrere ad una strategia che faccia un uso efficiente della memoria esterna. In [18] Haveliwala propone il partizionamento a blocchi della matrice dei link con un certo grado di replicazione per soddisfare i vincoli imposti dal calcolo del PageRank. Il vettore $x^{(h+1)}$ è costruito a blocchi.



Supponiamo di partizionare la matrice dei Link in β file distinti, ciascuno di dimensione tale potere essere caricato interamente in memoria fisica. La partizione produce i file $Link_{0, \ldots}$ $Link_{\beta-1}$ tali che il campo destinazione in $Link_i$ deve ricadere nell'intervallo $[\beta * i, \beta * (i+1)]$. In questo modo stiamo introducendo un certo grado di replicazione a causa della possibile necessità di memorizzare in diversi $Link_i$ lo stesso "SourceID", il campo "OutDegree" e il campo "Local Out" che contiene il numero di link uscenti associati ad uno specifico "Source Id" per la corrente partizione.

Source Id (32 bit)	Out Degree (16 bit)	Local Out (16 bit)	Dest Nodes (32 bit)
0	4	2	10,31
1	3	1	4
….	5	2	9,16

Intervallo Dest Nodes [0,31]

Source Id (32 bit)	Out Degree (16 bit)	Local Out (16 bit)	Dest Nodes (32 bit)
0	4	2	42,61
1	3	2	37,40
….	5	3	32,54,62

Intervallo Dest Nodes [32,63]

Per costruzione i singoli files $Link_i$ stanno in memoria fisica. In [18] si dimostra che i tempi di completamento scendono di un fattore 3 su un desktop-PC. Questo usando un fattore di bloccaggio pari a 4, su un dataset di 19 milioni di link. In corrispondenza c'è un aumento di occupazione in spazio pari al 28%. In dettaglio l'algoritmo usato è il seguente.

```
foreach s { source[s]= 1 / m }
residual = 1;
while (residual > ε)
        for i = 0 …. β-1 do {
                foreach i, dest[i] = 0
                while (not links.eof()) {
                        Links.read(source, r, dest1, dest2, …. destr);
                        for j = 1 … r do
                                dest[destj] = dest[destj] + source[source] / r
                }
                foreach i dest[i] = d * dest[i] + (1-d)  / m
                write Desti to disk
        }
        residual = ‖ source – dest ‖
        source = dest;
}
```

4. **Data una query, come costruire automaticamente quali sono le pagine autoritative per quel determinato argomento e quali quelle che contengono il maggior numero di link sull'argomento?** Un interessante studio pubblicato da Kleinberg [3] nel 1998, suddivide l'insieme delle pagine rilevanti per una data query in due distinte categorie. Le "Authorities" sono quelle pagine rilevanti per l'interrogazione sottomessa, mentre gli "Hub" sono le pagine che puntano alle Authorities. Ogni pagina *p* riceve quindi due tipi di rank. L'authority score p_a è determinato dall'hub score associato alle pagine che puntano *p*, l'hub score p_h è dato dall'authorities score associato alle pagine che sono puntate da *p* . Quello che è interessante è notare che gli Hub e le Authorities godono, quindi, di una proprietà nota

come “mutual reinforcing”: una buona Authorities è puntata da molteplici Hub ed un buon Hub è puntato da molteplici Authorities. E’ naturale quindi anche in questo caso ricorrere ad una formulazione ricorsiva. Prima di tutto è pero necessario definire alcuni insiemi di lavoro. Sia R l’insieme delle pagine ottenute in risposta ad una query (per fare questo si assume che l’indice delle pagine sia *già* stato costruito). Si considerino l’insieme S costruito aggiungendo al set R l’insieme delle pagine che puntano alle pagine in R e l’insieme delle pagine che sono puntate dalle pagine in R (il “neighbor set” viene in genere precalcolato dal page analysis module come nel connectivity server di Altavista [15]). E’ molto probabile che in S vi siano molte authorities perché esse dovrebbero essere puntate da qualcuna tra le pagine in R. E’ altrettanto probabile che vi siano degli Hub perché essi dovrebbero puntare alle pagine in R. Definito *B(i)*, l’insieme delle pagine che puntano alla pagina *i* ed *F(i)* l’insieme delle pagine puntate dalla pagina i, l’algoritmo applica iterativamente i seguenti due step I ed O:

1. inizializza a_i (authority score) ed h_i (hub score) a valori arbitrari
2. repeat

   I : $a_i = \sum_{j \in B(i)} h_j$

   O : $h_i = \sum_{j \in F(i)} a_j$

   Normalizza $\sum_i a_i^2 = 1$ ed $\sum_i h_i^2 = 1$

   until convergenza

L’interpretazione matriciale del metodo HITS è molto semplice. Sia A la matrice di adiacenza del grafo $G_{web.}$ Il generico $a_{ij} = 1$ se la pagina *i* punta alla pagina *j*, 0 altrimenti. Sia h = (h_i) ed a = (a_i) il vettore dei rank di authority ed hub. Allora le operazioni I: ed O: sono espresse da $a = A\ h$ ed $h = A^T a$, sostituendo ed introducendo due costanti per la normalizzazione otteniamo che devono essere soddisfatte $a = c_1 A\ A^T a$ ed $h = c_2 A^T A\ h$. Nuovamente ci siamo ricondotti al calcolo degli autovettori questa volta delle matrici $B = A\ A^T$ ed $C = A^T A$ rispettivamente. B è la matrice di co-citazione dove b_{ij} è il numero di siti che assieme puntano sia la pagina *i* che la pagina *j*. C è la matrice che in cui c_{ij} rappresenta il numero di siti che congiuntamente sono puntanti dalla pagina *i* e dalla pagina *j*. Una estensione di HITS è stato applicato da ricercatori della IBM nel progetto CLEVER ed in ARC [27]. L’idea è quella di tenere conto, nel calcolo delle authorities, del “mantenimento del focus di interesse”. Una pagina ha un maggior peso, se nel propagare i valori a_i ed h_i l’anchor associata ai link mantiene un certo grado di similiarietà con la query originale. Dal punto di vista algoritmico, l’unica differenza è la definizione $a_{ij} = 1 + \eta(t)$, con ή(t) numero di volte in cui il termine *t* fornito nella query appare nell’anchor associata ad al link orientato (*i, j*). In [34] viene fornita una interpretazione di HITS secondo il modello dei cammini randomici.

Un interessante risultato è stato raggiunto da Lempel [4] che da una definizione leggermente differente delle matrici. Lempel parte dalla costruzione di un grafo bipartito G_{bip}=(V_h, V_a, E). Ogni singolo nodo *n* del grafo che rappresenta S è sdoppiato in due nodi n_h e n_a (la parte Hub e la parte Authorities) e definiamo E = $\{(n_h, n_a) \mid n_h \rightarrow n_a \text{ in S}\}$. Sul grafo bipartito possiamo effettuare due cammini randomici. Ogni cammino visiterà solo i nodi da uno dei due lati del grafo. Poiché le Autorithies sono molto riferite verranno naturalmente raggiunte molto spesso nei cammini randominici, lo stesso vale per gli Hubs. Definiamo due matrici:

$$h_{i,j} = \sum_{\{k|(i_h,k_a),(j_h,k_a)\in G_{bip}} \frac{1}{\deg(i_h)*\deg(k_a)}$$

$$a_{i,j} = \sum_{\{k|(k_h,i_a),(k_h,j_a)\in G_{bip}} \frac{1}{\deg(i_a)*\deg(k_h)}$$

Un algoritmo iterativo calcola le due matrici, in modo analogo a quanto fatto in HITS. Il risultato notevole è che partendo da questa definizione si ottiene una dimostrazione analitica del fatto che il calcolo delle Autorithies assegna un rank proporzionale alla somma dei link incoming e il calcolo degli Hubs assegna un rank proporzionale alla somma dei link outgoing. Questa cosa sembra in contrasto con le premesse che hanno dato origine alla ricerca nel settore. In realtà il risultato teorico è stato confermato corretto ma lo stesso Lempel precisa che, in pratica, il sistema messo in piedi beneficia di due fattori: (a) il set S ha già selezionato un set di pagine comunque correlate alla query (b) i link sono stati accuratamente selezionati per ridurre al massimo la possibilità di spamming. Solo dopo che (a) e (b) vengono realizzati il calcolo dei rank viene effettuati.
Si noti che questo tipo di computazioni, dipendenti dalla query, possono essere effettuate solo a "search time", in genere richiedendo un tempo che va dai minuti a diverse ore. Per questo motivo non sono direttamente applicabili ai motori di ricerca che devono produrre risposte in pochi millisecondi.

5. **Dunque, come sfruttare le keywords durante il calcolo della Link Popularity?** Sono state proposte diverse soluzioni. La ricerca in questo campo è davvero agli albori e tutte le tecniche introducono comunque un overhead computazionale rilevante. Tra le altre riportiamo quella proposta da Richardson [16] che calcola i valori in parallelo all'indicizzazione, e prima della fase di search vera e propria. Prima di tutto chiariamo il perché si stiano svolgendo degli studi in questo settore. Il PageRank da un elevato peso alle pagine che sono al centro di un sub-grafo densamente connesso (in cui ci sono molte pagine che sono riferite reciprocamente). Sarebbe intuitivo pensare che i migliori risultati fossero raggiunti individuando il centro di un sub-grafo densamente connesso, e rilevante per la query. Considerando il modello del random surfer, questa è una approssimazione del comportamento del tipico navigatore sul Web. L'utente difatti di fronte ad una pagina non segue con equa probabilità tutti i link, ma piuttosto uscendo da una pagina segue quei link che hanno a che fare con il suo focus di interesse. Analiticamente abbiamo la seguente formulazione:

$$P_q(i) = d * \sum_{j\in B(i)} P_q(j)P_q(i \to j) + (1-d)P_q'(j) \quad \text{(R.3)}$$

Abbiamo introdotto delle piccole modifiche rispetto al PageRank tradizionale sostituendo il fattore di distribuzione uniforme $\frac{1}{N(j)}$ con la probabilità $P_q(i \to j)$ che la pagina *i* riferisca la pagina *j* di interesse *q* ed introducendo $P_q'(j)$, la probabilità che l'utente si stanchi e decida di passare ad una nuova pagina, in entrambi i casi dato il topic di interesse *q*. Questa la formulazione generale. Detta $R_q(j)$ una misura di rilevanza della pagina *j* alla query *q,* detto *N* il numero totale delle pagine nella collezione, detto *F(i)* il numero di pagine che riferiscono la pagina *i*, possiamo definire:

$$P_q'(j) = \frac{R_q(j)}{\sum_{k\in N} R_q(k)} \text{ ed } P_q(i \to j) = \frac{R_q(j)}{\sum_{k\in F(i)} R_q(k)}$$

$R_q\ (j)$ è una classica misura di ranking dipendente dal contenuto delle pagine Web per cui possiamo porre $R_q\ (j) = TF * IDF$. Sino al momento la (R.3) può essere risolta sempre ricorrendo al calcolo degli autovalori.
Nel caso di termini multipli $Q = \{q_1, ...q_n\}$ il calcolo deve tenere conto di una differente distribuzione di probabilità $P(q)$ ovvero il modello assume che il surfer tra tutti i termini $\{q_1, ...q_n\}$ ne selezioni uno q che guida la prossima scelta del link. Il processo viene ripetuto, per ogni termine, e si ottiene la formula finale:

$$P_Q(j) = \sum_{q \in Q} P(q) P_q(j)$$

Putroppo in [16] si dimostra come questa equazione non possa essere risolta direttamente con il calcolo degli autovalori e si suggerisce di usare la (R.3) come approssimazione di (R.4).

Chiudiamo questa rassegna degli algoritmi relativi al calcolo del rank basandosi su informazioni topologiche, invitando il lettore interessato ad approfondire il discorso partendo dalla lettura della reference [28]. L'articolo descrive un modello di simulazione del comportamento degli utenti Web che riassume le caratteristiche salienti del PageRank e di HITS, aggiungendone di nuove. Il surfer è un agente che può compiere quattro diversi tipi di azioni. Saltare in un nuovo nodo del grafo G_{web}, seguire un link in avanti a partire alla pagina corrente, seguire un link indietro a partire dalla pagine corrente, restare fermo. Per ogni azione si tengono conto delle probabilità condizionali derivanti dalla query e dal contenuto delle pagine. Altre letture interessanti sono la reference [35] che sviluppa un modello probabilistico in cui si tiene conto del contenuto delle pagine e dei link che puntano alle pagine e la reference [50] che analizza la stabilità del PageRank e di Hits intesa come variazione dei risultati in presenza di pertubarzioni sul grafo che rappresenta il Web.

Le tecniche di Link Popularity sono usate anche in contesti diversi da quelli del Web Ranking, per esempio sono state usate con successo per identificare community di pagine caratterizzate dal medesimo topic [31], per classificare in automatico le pagine, e per altre applicazioni.

La ricerca in questo settore deve rispondere ancora a molte domande. Per esempio è chiaro che siano necessari nuovi algoritmi iterativi per effettuare su memoria esterna e/o in modo distribuito il calcolo efficiente del rank. Sarebbe inoltre opportuno sviluppare dei modelli in cui si possa sfruttare il calcolo dei rank fatto nelle precedenti indicizzazioni per le pagine non modificate nel frattempo. Un risultato iniziale li si ha in [48] limitatamente alla creazione di nuovi archi sul grafo del Web e supponendo che la struttura dei nodi rimanga invariata.

Inoltre è chiaro che utilizzando il modello di Google si pongono alcuni problemi. Per essere autorevoli bisogna essere molto linkati, ma per essere molto linkati bisogna essere conosciuti da molto tempo ed appartenere ad una community di utenti molto attiva nel produrre pagine Web. Tuttavia, molto spesso si verificano degli episodi che producono "un interesse elevato" per alcuni argomenti in ristrettissimi intervalli di tempo. Ad esempio, subito dopo l'11 Settembre 2001 si è diffuso una forte attenzione per termini come "antrace" e sono stati prodotti molti documenti sull'argomento, poco riferiti appunto perché troppo nuovi. Tuttavia, il termine antrace è stato per lungo tempo ai primi posti tra quelli più cercati sui motori.
Un altro fenomeno che si sta evidenziando è il seguente: per essere autorevole è necessario essere nei primi posti di un motore di ricerca, tipicamente Google. Se non si è ai primi posti si riducono le chance di essere conosciuti e quindi anche di essere linkati. Ecco che, da un certo punto, in poi i meccanismi iterativi potrebbero essere interrotti proprio per il successo degli stessi motori. A tal

proposito, è interessante notare che in [28] è stato sviluppato un modello che intende descrivere la creazione dei link in una pagina, da parte di un Web editor, proprio basandosi sul ranking fornito da Google! .

Query Engine Module

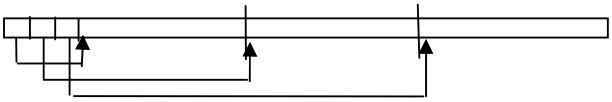
Il Query Engine riceve usa le liste invertite create in fase di indicizzazione per rispondere alle interrogazioni sottomesse dagli utenti. Il modulo deve essere progettato in modo tale scalare bene al crescere dei volumi delle interrogazioni. Per esempio google è passato da una media di 10.000 interrogazioni al giorno nel 96 poco dopo la sua creazione, ad oltre 150.000.000 di interrogazioni nel 2002. Oggi Google dedica oltre 6.500 servers esclusivamente al query processing. Nel seguito descriveremo una possibile architettura distribuita che fa uso di partizionamento degli indici in modalità document-based sfruttando un approccio basato sui livelli di qualità. Prima tuttavia concentriamoci sulla definizione del problema con algoritmi sequenziali. Rispondiamo quindi alle seguenti domande:

1. **Come rispondere alle interrogazioni sottoposte dagli utenti in forma booleana?** Uno dei motivi del successo delle liste invertite deriva dal fatto che con esse è possibile soddisfare in modo efficiente le interrogazioni con operatori booleani che sono uno standard per la maggior parte dei motori di ricerca. In [12] viene riportato il seguente algoritmo noto come "Small Intersection Set" (la parte qui rappresentata è solo quella relativa alle query AND, in modo analogo si possono processare le query in OR e i NOT). Su una query in AND un documento candidato deve apparire in tutte le posting list associate ai singoli termini, altrimenti deve essere scartato. Per questa ragione si risparmia spazio in memoria fisica e tempo se si parte sempre dalla lista associata al termine con minore frequenza, e se si processano i termini in ordine non decrescente di frequenza.

```
1. foreach query term t {
2.        cerca nel lexicon e memorizza f_t e l'indirizzo di I_t , la entry nella posting list per t
3.   }
4. Identifica il termine mt con il minimo f_t
5. Leggi la corrispondente I_t e setta la lista di candidati C ← I_t
6. Sorta la i termini per valori di f_t non decrescenti
7. foreach query term t ≠ mt in ordine non decrescente {
8.        Leggi la corrispondente It
9.        foreach d ∈ C {
10.               if d ∉ I_t then set C ← C – {d}
11.               if |C| = 0 the return "nessuna risposta"
12.       }
13.foreach d ∈ C {
14.       estrai dalla posting l'indirizzo corrispondete a d
15.       estrai dal document store il documento relativo e presentalo all'utente
16.}
```

   Una trattazione altrettanto efficiente non sarebbe realizzabile con le bitmap o le signature.
   Si noti che se le posting list vengono mantenute ordinate per *docID* non decrescenti allora è possibile controllare ottimizzare il controllo effettuato in linea 10. Supponiamo di stare analizzando il documento candidato *d* (linea 9) e di scandire linearmente la posting list associata al termine *t*. Se il *docID* correntemente raggiunto è più grande di del docID *d* allora siamo sicuri che il documento *d* non conteneva il termine *t* presente nella interrogazione booleana. Il documento *d* può essere immediatamente rimosso dalla lista di candidati. In altre parole stiamo traducendo il controllo di appartenenza della

linea 10 con una scansione lineare della posting list. Ovviamente durante la scansione provvediamo anche a decodificare la lista eventualmente compressa. Il problema che potrebbe derivare da questa strategia è la scansione di posting list molto lunghe con un conseguente elevato tempo di processing. La soluzione proposta in [12] da Moffat e di suddividere la posting list associata a ciascun termine in blocchi ripartiti per range di docID, premettendo alla posting list stessa una tabella di dispatch che memorizza la posizione di entrata per lo specifico blocco. Aumenta l'occupazione in spazio ma si riduce sensibilmente il tempo di processing perché si può "saltare" direttamente al blocco specifico ed effettuare una ricerca lineare solo all'interno del blocco.



La soluzione proposta in [39] sfrutta il fatto che nella posting list si garantisce un ordinamento non decrescente dei docID. Vedremo che in altri contesti sarebbe opportuno memorizzare nella posting list i riferimenti ai documenti secondo altri ordinamenti. Per esempio è intuitivo pensare che in fase di search potremmo trarre benefici dal fatto che i documenti siano già ordinati per valori di Link Popularity crescenti. La scelta da compiere è quindi un trade-off. Un altro schema che permetta di processare velocemente le posting list è descritto in [51]

2. **Esistono altri modelli di interrogazione oltre a quello booleano?** Due modelli di retrieval che hanno riscontrato un vasto campo di applicazioni sono quello probabilistico ed il vector space. Vediamone le caratteristiche salienti.

   - **Modello probabilistico.** L'idea è di usare il teorema di Bayes per stabilire se un documento è rilevante o meno per la query. Ricordiamo che il teorema di Bayes permette di stimare la probabilità condizionale che si verifichi l'evento H_i in presente a delle evento E come:

$$\Pr(H_i \mid E) = \frac{\Pr(E \mid H_i)\Pr(H_i)}{P(E) = \sum_j \Pr(E \mid H_j)\Pr(Hj)} i = 1,....,n$$

     La base di partenza è un set *TS* di documenti usato come training e di cui si accetta *a priori* il giudizio di rilevanza. Supponiamo che $N = |TS|$ che R_t siano i documenti ritenuti rilevanti per il termine *t*, che il termine *t* appaia in f_t documenti e che *R* siano i documenti ritenuti rilevanti in senso assoluto. Possiamo quindi pre-calcolare sulla collezione usata da training le probabilità condizionali, nel modo seguente:

     Pr[documento rilevante | che il termine *t* è presente] = R_t / f_t
     Pr[documento irrilevante | che il termine *t* è presente] = 1- $(R_t / f_t) = (f_t - R_t)/f_t$
     Pr[termine *t* sia presente | il documento sia rilevante] = R_t / R
     Pr[termine *t* sia presente | il documento sia irrilevante] = $(f_t - R_t) / (N - R)$

     Ed assegnare, grazie al teorema di Bayes, il seguente peso w_t ai singoli termini:

$$w_t = \frac{R_t /(R - R_t)}{(\mathrm{f_t} - R_t)/(N - \mathrm{f_t} - (R - R_t))}$$

Assumendo che i singoli termini appaiano in modo indipendente nel documento (questa è una semplificazione imposta dal modello), otteniamo che il peso di un nuovo che entra nella collezione documento *d* è pari ad $w_d = \prod_{t \in d} w_t$. Gli usi possibili di questo modello sono molteplici. Per esempio, w_d può essere calcolato a query time partendo da w_t calcolati a index time sul training set. I documenti vengono poi sortati per w_t.
Una diversa applicazione è quella della classificazione automatica dei documenti [32]. Si suppone che un dataset di documenti sia stato già diviso in categorie e queste siano state usate per il training del classificatore. Nuovi documenti sono classificati scegliendo la categoria che massimizza la probabilità condizionale rispetto ai dati pre-calcolati.
Infine, un uso interessante è stato fatto da Haveliwala in [22]. Qui il modello probabilistico viene usato congiuntamente al calcolo del PageRank. L'idea è quella di usare, come training set a tempo di indicizzazione, i documenti contenuti nelle 16 categorie di top-level presenti in ODP [23] e di calcolare, a search time, la vicinanza della query alla categoria. Sempre ad indexing time, ad ogni pagina viene associato non un singolo valore di PageRank ma un vettore $[v_i]$ di 16 valori. Ciascun valore è calcolato usando i link presenti nelle 16 categorie di top-level di ODP come vettore di bias non uniforme che guida il salto verso una nuova pagina, quando il random surfer si "stanca" di visitare la pagina corrente. La vicinanza della query alla categoria permette di selezionare il corretto v_i. Sempre in [22] Haveliwala suggerisce un altro uso del modello probabilistico. La query Q può assumere un significato differente a seconda del "contesto" in cui è sottomessa. Per una query è possibile dare diverse definizioni di "contesto". Per esempio all'interno di un documento (una mail, un documento word od altro), il contesto di un termine è il documento stesso. Altri contesti possono essere i pattern di browsing, i bookmarks, ed i pattern di search. In sintesi, il modello probabilistico appare la scelta naturale per tenere conto dei termini contenuti nei "contesti di query".

- **Modello Vector Space:** Questo modello trae origine da una interpretazione in senso topologica della query e dei documenti. Indicando con *m* il numero di termini distinti in una collezione, possiamo immaginare sia i documenti che le query come vettori nello spazio ad *m*-dimensioni R^m. Il valore v_i nel vettore (v_i) $1 \le i \le m$ assume valore pari alla frequenza dell' *i*-esimo termine nel documento o nella query. Possiamo immaginare quindi che due documenti sono molto simili se il coseno dell'angolo tra i vettori che li rappresentano è molto piccolo. Analogamente possiamo dire che un documento è significativo per una query usando la stessa misura. Dalla geometria sappiamo che vale $X.Y = |X|.|Y|\cos\theta$ essendo X.Y il prodotto interno tra due vettori $X.Y = \sum_{i=1}^{m} x_i y_i$, $|X| = \sqrt{\sum_i x_i^2}$ ed θ l'angolo tra i vettori di conseguenza otteniamo $\cos\theta = \frac{\sum_{i=1}^{m} x_i y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}}$. I documenti significativi per una interrogazione possono essere ordinati per valori crescenti di *cos θ*. Un approfondimento è presente in [12]. La misura del coseno è, per esempio, usata in [31] per la "topic" distillation. L'algoritmo di Kleimberg è modificato ed ad ogni link si attribuisce un peso che calcola la vicinanza delle pagine rispetto ad una "query virtuale". Questa è costruita concatenando le prime 1000 parole delle pagine nel set S costruito a partire al root set.

3. **Come misurare la rilevanza delle risposte?** In letteratura vengono fornite due misure. La *recall* ovverosia il numero di documenti che sono restituiti tra gli *r* top-ranked rapportato al numero di documenti rilevanti nella collezione. E la *precision* ovverosia il numero di documenti restituiti che sono rilevanti rapportati al quelli restituiti. L'applicazione di queste misure è in larga misura soggettiva e poco adatta al Web. Per esempio, come riconoscere *quali* siano i documenti rilevanti in una collezione se non basandosi su un campione statistico e quindi sul giudizio personale degli esseri umani che ne fanno parte?. La letteratura in merito è molto vasta e la reference [41] è un ottimo punto di partenza. Un interessante metodologia è stata sviluppata da Henzinger in [44]. L'idea chiave dell'articolo è quella di costruire un campione di sampling delle pagine Web più importanti lasciando che i crawler simulino un "cammino randomico" sul Web. Se ad ogni passo si "salta" verso una pagina scelta in modo casuale con distribuzione uniforme, dopo un numero di passi sufficiente, verranno visitate con maggiora probabilità le pagine con PageRank più alto. Successivamente si verifica se le pagine appartenenti al campione di sampling costruito sono state indicizzate dai search engine, sottomettendo query in "*and"* delle keywords meno frequenti.

4. **Come ordinare i documenti che soddisfano le interrogazioni?** Abbiamo discusso ampiamente di tecniche per di ranking e abbiamo visto che esse ricadono in due categorie. Quelle Within-Document usano la query sottomessa dall'utente ed il contenuto della pagine per attribuire dinamicamente un "punteggio" alla pagina stessa. Quelle basate sulla Link Popularity attribuiscono alle pagine un "punteggio" statico dipendente dal contesto topologico in cui le stesse sono situate. Dobbiamo adesso occuparci di come combinare efficacemente questi due aspetti.
   Prima di tutto una osservazione. Supponiamo di usare solo il rank di Link Popularity e di memorizzare i documenti nella posting list direttamente secondo valori non decrescenti di Link Popularity. Ecco che non abbiamo bisogno di alcun sorting a query time. Non solo. Poiché i motori di ricerca presentano un numero ridotto *r* (in genere *r* = 10) di documenti nella pagina di risposta, estratti tra i più significativi tra i W_Q che soddisfano la query Q,, possiamo fermarci non appena abbiamo identificato i primi *r* documenti significativi. Siamo certi infatti che questi *r* siano i top-ranked. Un bel guadagno visto che quasi sempre $r << W_Q$!!. Per conciliare questa strategia con quella che richiede che nelle posting list i documenti siano ordinati per valori di docID non decrescenti, potremmo pensare di usare una funzione di mapping dei documenti che assegna i docID in base a valori di Link Popularity non decrescente.
   Ovviamente, stiamo volutamente trascurando l'uso delle componenti di rank Within-Document ed inoltre stiamo precludendoci qualunque forma di fine-tuning del rank a query time. Qualunque modifica del rank richiede una ricostruzione dell'indice. In genere si adottano strategie più flessibili. Per esempio, Google dichiarava in [2] di attribuire ad ogni documento in W_Q un rank dato dalla formula

$$R_Q(i) = \alpha * WithinDocRank + \beta * LinkPopularityRank \quad \text{(R.5)}$$

   Chiaramente è possibile definire altre funzioni che combinino opportunamente le due componenti del rank. Quello che è importante sottolineare che questa operazione avviene adesso a "search time" subito dopo avere individuato l'insieme W_Q che soddisfa la query booleana $Q = \{q_1, ...q_n\}$ e prima di fornire in agli utenti gli *r* documenti considerati top ranked.
   Di conseguenza, è necessario ordinare con un algoritmo in memoria la lista dei docID contenuti W_Q. Detto N = $|W_Q|$ questo può essere fatto in O(NlogN). In realtà possiamo

risparmiare tempo considerando che in realtà noi abbiamo bisogno di estrarre solo gli *r* docID con rank più grande tra gli *N* disponibili. Da un punto di vista algoritmico questo è un problema di selezione più che un problema di sorting ed può essere risolto con tecniche specifiche.

Per esempio, supponiamo di usare un max-heap di dimensione O(*N)* che in cui inseriamo, in tempo O(*N*), i W_Q docID da ordinare usando il rank come chiave. Con r operazioni di estrazione della radice estraiamo dall'heap gli *r* docID di rank massimo con un costo di O(*rlogN*) + O(*N*).

Un approccio diverso è quello di usare un min-heap questa volta di dimensione O(*r)* ed inserire gli iniziali *r* docID presenti in W_Q sempre usando il valore di rank come chiave. Poi, prendiamo in considerazione gli *N-r* docID rimanenti in W_Q. Il rank calcolato per ciascun docID $rank(docID_i)$ viene confrontato con la radice dell'heap, che per costruzione contiene istante per istante il minimo rank *min_rank* incontrato sino a quel momento. Se il $rank(docID_i) > min_rank$ allora possiamo estrarre la radice e inserire nell'heap il valore più grande $rank(docID_i)$. In modo automatico, l'heap porterà sulla radice il nuovo *min_rank* nell'insieme dei docID processati sino a quel momento. Se invece vale $rank(docID_i) < min_rank$ allora $docID_i$ può essere tranquillamente scartato in quanto gli elementi presenti nell'heap hanno tutti sicuramente un rank più grande. Nel caso pessimo questo algoritmo prende come tempo O(N+Nlogr). Si dimostra tuttavia che nel caso medio si impiega solo $O(r\log_e N/r) = 0.69 \log N/r$. Supponiamo una distribuzione uniforme dei valori di rank e sapendo che ad ogni istante l'heap contiene gli *r* valori visti sino a quel punto. Se abbiamo già visto *r* elementi e supponiamo di incontrare l'i-esimo con *i>r*, la probabilità che esso sia maggiore degli elementi nell'heap è *r/i*. Cosi otteniamo la formula $\sum_{i=r+1}^{N} \frac{r}{i} = r * \sum_{i=r+1}^{N} \frac{1}{i} = \sum_{i=1}^{N} \frac{1}{i} - \sum_{i=1}^{r} \frac{1}{i} = r \log_e (N/r)$ essendo $\sum_{i=1}^{n} \frac{1}{i} \cong \log_e (n+1)$. La tecnica permette quindi di risparmiare spazio e tempo.

Per ridurre ulteriormente il tempo necessario alla selezione degli *r* documenti top ranked, spesso i motori di ricerca accettano di fornire risposte subottimali ed forzano N = $|W_Q|$ a non superare un valore prefissato. Per esempio in Google, quando la lista dei candidati supera le 40.000 entry, le rimanenti vengono semplicemente scartate.

5. **Come presentare i documenti significativi per l'interrogazione sottomessa dagli utenti?** Una volta selezionati ed ordinati gli *r* docID che soddisfano la query $Q = \{q_1, ...q_n\}$ con massimo rank nell'insieme W_Q i risultati possono essere estratti dal document store e presentati agli utenti. Quasi sempre il documento *i*-esimo viene sostituito da una sua rappresentazione costituita da l'URL, il titolo ed una piccola sintesi del documento stesso. La sintesi viene spesso costruita estraendo finestre di testo che contengono le keywords q_i In letteratura questa fase viene definita come document highlight o snippets creation. La trattazione di questo argomento esula dal core del survey. Una introduzione all'argomento è fornita nel libro [49]

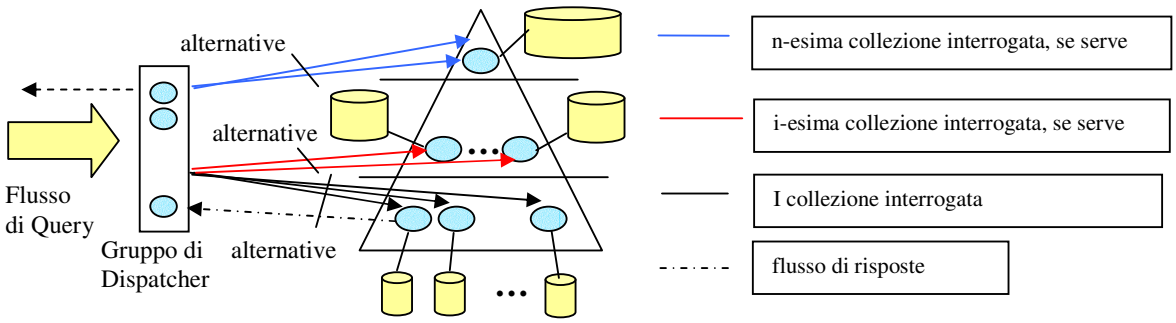
6. **Come servire le query usando un approccio distribuito?** Le soluzioni sono molteplici. Una introduzione all'argomento è presente in [38]. Ciascuna ha vantaggi e svantaggi e può fornire, o meno, risultati sub-ottimali. Supponiamo di avere un insieme di *R* nodi "query nodes" deputati a risolvere le interrogazioni ed un insieme di *S* nodi "dispatchers" deputati a smistare le interrogazioni sottomesse dagli utenti ai "query nodes" attendere le risposte, eventualmente processarle ulteriormente, e proporle agli utenti.

Cominciamo da un modello in cui si adotti una costruzione degli indici di tipo "word partition". Ciascun query node QN_j $1 \leq j \leq R$ possiede una range di parole ordinato, tra tutte quelle indicizzate nelle collezioni. Ecco che di conseguenza il singolo dispatcher D_i $1 \leq i \leq S$ può smistare la query composta da una singola keyword direttamente su uno specifico query node QN_k con una comunicazione unicast. Il query node risolve la query secondo le tecniche già viste e ritorna la risposta. I problemi sorgono quando viene sottomessa una query booleana $Q = \{q_1, ... q_n\}$. Ciascuna keyword q_r viene indirizzata sullo specifico query node $QN_{k(q_r)}$, ma questi non può applicare localmente l'algoritmo "Small Intersection Set" visto in precedenza perché non ha visione delle posting list selezionate dagli altri query node. In altre parole ciascun query node non può fare altro che inviare la posting list di propria pertinenza al nodo dispatcher D_i. Il dispatcher ricevute tutte le posting list applica l'algoritmo di "Small Intersection Set" costruisce l'insieme W_Q ed estrae da questi gli r top-ranked docID. Eseguito il fetch dal document store, i risultati vengono presentati agli utenti. Si noti che la soluzione proposta produce un notevole overhead di comunicazioni nel caso di query booleane, solo in parte mitigate dall'introduzione di meccanismi di caching a livello di query dispatcher.

	Single keywords	**Boolean Query**	**Merge, Sortine, Retrieval**	**Caching**	**Indice a livelli**
Word based partition Index	Accesso diretto ad un singolo query node. Comunicazioni in Unicast	Accesso diretto ai singoli query nodes che lavorano in parallelo Query inviata in Multicast. La risposta verso il dispatcher contiene tutti i documenti W_{qi}	Merge e ordinamento dei risultati sono effettuate dal dispatcher. Lo stesso accade per il retrieval dei documenti	Il dispatcher può cachare le risposte.	Non conveniente. Ogni nodo ha allocato un range di parole e non agevole suddividere le collezion in base ad un giudizio a priori assegnato alle singole pagine.
Document based partition Index	Broadcast della query a tutti i query node. Multicast usando indici a livelli.	Broadcast della query a tutti i query node. Multicast usando indici a livelli. La risposta verso il dispatcher contiene solo gli r_c documenti top ranked per ciascuna collezione c	Ordinamento dei risultati effettuati, in parallelo dai singoli query nodes. Merge e retrieval effettuato dal dispatcher	Il dispatcher può cachare le risposte.	Si. E' possibile interrogare selettivamente degli indici in bas ad un giudizio statico attribuito alle pagine. Si possono ridurre i tempi di query accettando risulta sub-ottimali ed interrogando specifiche collezioni.

Una soluzione spesso più efficiente è quella di utilizzare una partizione degli indici di tipo "Document Based". In questo caso le comunicazioni necessarie dai D_i verso i QN_j $1 \leq j \leq R$ devono essere necessariamente di tipo broadcast. Il grosso vantaggio è che ciascun QN_j $1 \leq j \leq R$ può risolvere localmente le query booleane applicando l'algoritmo "small intersection set", attribuendo un rank ai propri documenti, ordinando i risultati e presentandoli al dispatcher D_i. Questi ha solo il compito di eseguire una merge, in tempo lineare, degli r_j $1 \leq j \leq R$ documenti selezionati top-ranked da ciascuna collezione locale e successivamente presentare i primi r documenti selezionati all'utente.
In tutte le due soluzioni proposte si cerca punta a ridurre l'impatto delle comunicazioni. Per questa ragione sulla rete "girano" messaggi, in forma compressa, rappresentanti i docID piuttosto che i documenti veri e propri. E si delega al dispatcher il retrieval dei documenti dal document store.
Entrambe le soluzioni, poi beneficiano dell'introduzione di meccanismi di caching a livello di dispatcher per ridurre l'utilizzo della banda di comunicazione. La soluzione basata su "Document based partition" lavora molto bene su query booleane e per contro richiede un broadcast delle query a tutti i nodi. La soluzione "Word based partition" seleziona il sottoinsieme di nodi specifico a cui inviare le query, ma per contro provoca un notevole overhead di comunicazioni perché riesce a fare poco lavoro in locale. L'argomento può essere approfondito in [19].

7. **Quale organizzazione scegliere tra la "Document based partition" e la "Word based partition"?** Quasi tutti i motori di ricerca Web hanno scelto la prima che, come abbiamo visto, permette anche una maggiore flessibilità degli update. Spesso sono adottati dei compromessi che pur fornendo un teorico ranking sub-ottimale, si rivelano ottimali da un punto di vista pratico. Supponiamo di partizionare i documenti secondo un giudizio statico, calcolato ad indexing time, derivante da tecniche di tipo Link Popularity. In pratica è come se ad ogni collezione stessimo attribuendo un giudizio *a-priori* sulla sua qualità. Parlando in termini di PageRank possiamo pensare che la collezione di livello 0 contenga i siti ritenuti "*più rilevanti*" in termini assoluti. In generale, la collezione di livello *i* contiene tutti i documenti che stanno a livello di rilevanza *i* secondo una qualche scala (per esempio quella logaritmica). Ecco che il broadcast richiesto dalla tecnica "Document based partition" può essere visto come un multicast *verso la collezione di livello i.* Se questa non fornisce sufficienti risultati il dispatcher provvede a forwardare l'interrogazione verso la collezione di livello *i+1*. Si noti che un rilevante beneficio di questa soluzione è che, potenzialmente, essa permette di organizzare i dati su ogni query node in modo tale da ottimizzare le risorse locali. Per esempio, si possono usare dischi la cui capienza non ecceda quella media correntemente disponibile in commercio. E si può cercare di organizzare i dati in modo tale che essi stiano in memoria principale, limitando la paginazione sul disco. Una proposta di architettura è data in figura. I query nodes disponibili sono organizzati ad albero. La collezione di livello *0*, più frequentemente acceduta, è replicata su ciascun nodo foglia. La collezione di livello *i*, proporzionalmente meno acceduta, è replicata su tutti nodi interni dell'albero di livello *i.* In generale le collezioni hanno una dimensione crescente al crescere dei livelli nell'albero.



L'insieme di dispatcher ricevuta una query, la invia ad un qualunque nodo foglia scelto secondo una prefissata politica di bilanciamento. Per esempio, D_i può adottare il round-robin, oppure politiche dipendenti dalla lunghezza delle code di invio verso QN_j. Se il numero di risposte individuato è *r*, allora l'interrogazione si interrompe al livello 0. Altrimenti, prosegue sui livelli successivi sino ad individuare un numero risposte sufficienti.

Collezioni dinamiche

Un aspetto meno investigato dalla ricerca è quello di trattare grossi moli di documenti che cambiano frequentemente nel tempo garantendo un aggiornamento in tempo reale dell'indice. Questo è un problema che comincia a trovare applicazioni sempre più diverse. Per esempio un motore di ricerca Web può avere un servizio di indicizzazione in tempo reale di news provenienti da diverse fonti (agenzie stampe, insiemi selezionati di siti Web, giornali on line). Un altro esempio potrebbe essere quello di un servizio che indicizza le informazioni provenienti dai mercati borsistici (stock quote, news, future). Un terzo esempio, che presumiamo avrà sempre più sviluppo, è quello relativo all'indicizzazione di informazioni provenienti da reti in cui i singoli nodi siano caratterizzati da un

comportamento dinamico. Per esempio in una rete wireless il singolo PC può essere presente o assente in modo del tutto randomico, lo stesso dicasi per reti UMTS o per sistemi di search realizzati per reti Peer-to-Peer.
Da un punto vista algoritmico, sono state proposte delle soluzioni utilizzando β-tree [33] in cui la chiave è il termine la posting list è il valore puntato dalle foglie dell'albero. Per minimizzare gli accessi al disco le chiavi vengono updatate in modo batch e si provvede a pre-ordinarle in memoria per minimizzare la distanza "di salto" sul disco e ridurre il tempo di completamento. Questa soluzione è proposta in [33]. In [19] Garcia-Molina suggerisce di segmentare le posting a blocchi in modo tale che ciascun blocco abbia approssimativamente la stessa dimensione. Questo produce dei benefici nel β-tree, ma riduce la possibilità di comprimere la posting list. Nel β-tree sono inserite le coppie (chiave, valore), eventualmente, provvedendo a replicare le chiavi tante volte quanti sono i blocchi della rispettiva posting list. La soluzione suggerita è realizzata al di sopra del Berkeley DB.
Altra soluzione proposta da Garcia-Molina in [24] è quella di memorizzare le modifiche che avvengono successivamente alla prima indicizzazione in una sorta di struttura di appoggio organizzata a log e consultata a tempo di query in cascata alle liste invertite. Ad intervalli periodici la struttura di log è "consolidata" e le posting list sono ricostruite incorporando le modifiche che sono accadute dall'ultima indicizzazione.
Tutte le soluzioni proposte si sono però dimostrate inadatte a trattare le modifiche di una grossa quantità di dati. Per esempio, supponiamo di volere costruire una rete peer to peer per condividere e cercare documenti Web. In questa rete, la caduta di un singolo nodo N_p provoca, nell'insieme dei nodi deputati all'indicizzazione, la rimozione dagli indici dei riferimenti ai documenti posseduti da N_p. Lo situazione inversa si presenta quando un nuovo nodo entra sulla rete. Considerando la frequenza con cui i nodi entrano ed escono dalla rete, il numero di documenti che essi possiedono, ed il numero totale di chiavi nei documenti, si capisce che soluzioni basati su β-tree o su strutture a log consolidate periodicamente nella posting list non siano sufficienti [1].

Bibliografia

1. Arvind Arasu and Junghoo Cho and Hector Garcia-Molina and Andreas Paepcke and Sriram Raghavan, "Searching the Web", "ACM Transactions on Internet Technology", "2001";
2. Sergey Brin and Lawrence Page, "The anatomy of a large-scale hypertextual {Web} search engine", "Computer Networks and ISDN Systems vol. 30 num 1--7, pages 107--117", "1998";
3. Jon M. Kleinberg, "Authoritative sources in a hyperlinked environment", "Journal of the ACM vol. 46, num. 5, pages 604--632", "1999";
4. R. Lempel and S. Moran, "The stochastic approach for link-structure analysis (SALSA) and the TKC effect", "Computer Networks vol. 3 num 1—6 pages 387--401", "2000";
5. Lawrence Page and Sergey Brin and Rajeev Motwani and Terry Winograd, "The PageRank Citation Ranking: Bringing Order to the Web" "Stanford Digital Library Technologies Project","1998";
6. Arvind Arasu,"PageRank Computation and the Structure of the Web: Experiments and Algorithms"
7. Junghoo Cho and Hector Garcia-Molina and Lawrence Page, "Efficient crawling through {URL} ordering", "Computer Networks and ISDN Systems, vol 30 num. 1—7 pages 161--172", "1998";
8. Junghoo Cho and Hector Garcia-Molina, "Parallel crawlers. In Proc. of the 11th International World--Wide Web Conference", "2002";
9. D. Dato, A. Gullì, G. Attardi, Web Host Enumeration Through DNS, *Web Net '97*, Toronto, 1997.
10. V. Shkapenyuk and T. Suel. "Design and implementation of a high-performance distributed Web crawler", "In Proceedings of the 18th International Conference on Data Engineering (ICDE'02), San Jose, CA Feb. 26--March 1, pages 357—368", "2002"
11. Maxim Lifantsev, Tzi-cker Chiueh "I/O-Conscious Data Preparation for Large-Scale Web Search Engines". "VLDB 2002"
12. Moffat and Bell, "Managing Gigabyte", "Witten, 1999"

13. Steve Lawrence and C. Lee Giles, "Accessibility of Information on the Web", "Nature vol. 400, pages 107--109",
14. Andrei Broder et al, "*Graph structure in the web*," Proc. 9th International World Wide Web Conference, pp. 309--320, 2000
15. Krishna Bharat, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian, "*The Connectivity Server: fast access to linkage information on the Web*, Computer Networks and ISDN Systems 30 (1998), no. 1-7, 469-477."
16. Mathew Richardson and Pedro Domingos, "The Intelligent Surfer: Probabilistic Combination of Link and Content Information in Page Rank", "Advances in Neural Information Processing Systems 14, MIT Press",
17. Zobel, Moffat, Ramamohanarao "Inverted Files Versus Signature Files for Text Indexing,. ACM Transactions on Database Systems, 23(4):453-490, December 1998"
18. Taher Haveliwala, "Efficient Computation of PageRank 1999 Technical report, Stanford University, Stanford, CA",
19. Sergey Melnik and Sriram Raghavan and Beverly Yang and Hector Garcia-Molina,"Building a distributed full-text index for the Web booktitle: World Wide Web pages 396-406. 2001",
20. Jun Hirai and Sriram Raghavan and Hector Garcia-Molina and Andreas Paepcke,"WebBase: a repository of {Web} pages Computer Networks (Amsterdam, Netherlands: 1999 volume 33,number 1—6 pages 277—293 year 2000",
21. Paolo Boldi and Bruno Codenotti and Massimo Santini and Sebastiano Vigna, "UbiCrawler: A Scalable Fully Distributed Web Crawler",
22. Taher Haveliwala, "Topic-Sensitive PageRank"
23. ODP http://www.dmoz.org
24. Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. "Incremental updates of inverted lists for text document retrieval. Technical Note STAN-CS-TN-93-1, Stanford University, 1993".
25. Ion Stoica and Robert Morris and David Karger and Frans Kaashoek and Hari Balakrishnan, "Chord: {A} Scalable {Peer-To-Peer} Lookup Service for Internet Applications 149--160",
26. Marc Najork and Janet L. Wiener, "Breadth-First Crawling Yields High-Quality Pages Proceedings of the 10th International World Wide Web Conference Elsevier Science Hong Kong 114—118 May 2001",
27. Chakrabarti and B. Dom and D. Gibson and J. Kleinberg and P. Raghavan and S. Rajagopalan, "Automatic resource list compilation by analyzing hyperlink structure and associated text, Proceedings of the 7th International World Wide Web Conference 1998",
28. Gopal Pandurangan and Prabhakara Raghavan and Eli Upfal, "Using PageRank to Characterize Web Structure, 8th Annual International Computing and Combinatorics Conference (COCOON) 2002",
29. Michelangelo Diligenti, Marco Gori, Marco Maggini, "Web Page Scoring Systems for Horizontal and Vertical Search WWW2002, May 7-11, 2002, Honolulu, Hawaii, USA. ACM 1-58113-449-5/02/0005."
30. A. Vo and A. Moffat., "*Compressed inverted files with reduced decoding overheads*. In R. Wilkinson, B. Croft, K. van Rijsbergen, A. Moffat, and J. Zobel, editors, Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval, pages 290--297, Melbourne, Australia, July 1998".
31. Krishna Bharat and Monika R. Henzinger, "Improved algorithms for topic distillation in a hyperlinked environment Proceedings of {SIGIR}-98, 21st {ACM} International Conference on Research and Development in Information Retrieval pages 104—111 1998"
32. *Tom Mitchell, "Machine Learning, McGraw Hill, 1997"*.
33. Tatu Ylönen, "An Algorithm for Full-Text Indexing, 1992",
34. Davood Rafiei and Alberto Mendelzon, "What is this Page Known for? Computing Web Page Reputations",
35. David Cohn and Thomas Hofmann, "The missing link - a probabilistic model of document content and hypertext connectivity 2001, Neural Information Processing Systems 13"

36. David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin and Rina Panigrahy. "Consistent hashing and random trees: Distributed cachine protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pages 654-663* , 1997."
37. David Karger, Tom Leighton, Danny Lewin, and Alex Sherman."Web caching with consistent hashing. In Proc. of 8th International World--Wide Web Conference, Toronto, Canada, 1999"
38. wen de Kretser and Alistair Moffat and Tim Shimmin and Justin Zobel, "Methodologies for Distributed Information Retrieval International Conference on Distributed Computing Systems 66-73 1998",
39. Alistair Moffat and Justin Zobel, "Self-Indexing Inverted Files for Fast Text Retrieval ACM Transactions on Information Systems 14 4 349—379 1996"
40. J. Zobel, A. Moffat, and K. Ramamohanarao. "Guidelines for presentation and comparison of indexing techniques. ACM SIGMOD Record, 25(3):10--15, September 1996"
41. TREC http://trec.nist.gov/
42. R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In 32nd A CM Symposium on Theory of Computing, pages 39406, 2000.
43. Paolo Ferragina and Giovanni Manzini", "Opportunistic Data Structures with Applications IEEE Symposium on Foundations of Computer Science 390-398 2000",
44. Monika R. Henzinger and Allan Heydon and Michael Mitzenmacher and Marc Najork, "Measuring index quality using random walks on the Web Computer Networks Amsterdam, Netherlands: 1999 31 11—16 1291—1303 1999",
45. K. H. Randall, R. Stata, R. G. Wickremesinghe, and J. L. Wiener. "The link database: Fast access to graphs of the Web. Research Report 175, Compaq Systems Research Center, Palo Alto, CA, 2001. 15 2001",
46. Allan Heydon and Marc Najork, "Mercator: A Scalable, Extensible Web Crawler journal World Wide Web vol.2 num.4 pages 219-229 1999",
47. M. Najork and A. Heydon, "On High-Performance Web Crawling Research Report, Compaq Systems Research Center, 2001",
48. Steve Chien and Cynthia Dwork and Ravi Kumar and D. Sivakumar, "Towards exploiting link evolution" to appear (citeseer.nj.nec.com/501969.html),
49. Brigitte Endres, "Summarizing Information" Springer 1998
50. A. Ng and A. Zheng and M. Jordan, "Stable algorithms for link analysis Andrew Y. Ng, Alice X. Zheng, and Michael I. Jordan. Stable algorithms for link analysis. In Proc. 24th Annual Intl. ACM SIGIR Conference. ACM, 2001.",
51. F. Scholer and H.E. Williams and J. Yiannis and J. Zobel, "Compression of Inverted Indexes For Fast Query Evaluation",