Corso: Gestione ed elaborazione di grandi moli di dati
Lezione del: Approfondimento
Argomento: Compressione e ricerca di dati XML: XBZip e XBZipIndex
Scribes: Salvagnin Elisa, Silvestri Francesco, Tagliacollo Claudia, Vandin Fabio

Abstract

Uno dei *challenge* nell'elaborazione di documenti di grandi dimensioni è quello di navigare e ricercare dati direttamente nella loro rappresentazione compressa in maniera efficiente. Un approccio di questo tipo applicato a documenti XML è stato introdotto in [FLMM06]; questo lavoro specializza un approccio più generale descritto in [FLMM05] che utilizza la trasformata XBW. Viene qui fornita una descrizione del primo articolo con eventuali puntualizzazioni estratte dal secondo.

Nella Sezione 1 vengono descritti brevemente il linguaggio XML e l'oggetto DOM; nella Sezione 2 viene introdotta la trasformata XBW applicata a documenti XML; nelle Sezioni 3 e 4 vengono introdotte le operazioni di navigazione e di ricerca nella versione compressa di un documento XML; nella Sezione 5 vengono forniti i dettagli implementativi.

1 Introduzione

1.1 XML: utilità e problematiche

XML (*eXtensible Markup Language*)[XML] è un liguaggio di *markup*, come HTML, che serve per definire la struttura di un documento. Esso utilizza dei *tag* che servono a chiarire la semantica del contenuto di un documento (mentre HTML opera a livello di presentazione). Esso è estensibile, cioè l'utente può definire dei nuovi *tag*, ed uno dei motivi per cui è stato introdotto è che permette di maneggiare in modo semplice i documenti strutturati. Il suo utilizzo sta crescendo molto rapidamente e si stima che nel 2008 almeno il 40% del traffico della rete sarà costituito da documenti in formato XML. Un classico, seppur semplice, esempio di documento XML si può trovare in Figura 1.

In un documento XML si possono riconoscere i seguenti elementi:

- **Tag**: può essere di apertura (ad esempio <book>) o di chiusura (</book>) e serve per definire la struttura di un documento.
- **Attributo**: rappresenta le proprietà di un *tag*. Si veda ad esempio l'attributo id del *tag* <book>.

- **Valori di attributo**: è il valore assunto da un attributo; ad esempio nel secondo *tag* <book> il valore dell'attributo id è 2.

- **Contenuto testuale**: tutto ciò che non è un *tag* ed è contenuto tra un *tag* di apertura ed uno di chiusura. Ad esempio la stringa J.Austin contenuta tra i *tag* <author> e </author>.

Ci si riferisce ai valori di attributo e ai contenuti testuali con il termine generale PCDATA.

La crescente popolarità di XML è dovuta al fatto che esso codifica molte *meta-informazioni* in un formato facilmente interpretabile dalla struttura dei *tag*. Ovviamento tutto ciò ha un costo, che porta a tre principali problematiche nell'elaborazione di documenti in formato XML. Per prima cosa, i documenti XML hanno una intrinseca struttura ad albero (*DOM tree*) e questo rende molte delle operazioni di interesse, quali ad esempio indicizzazione, navigazione e ricerca, più complicate di quanto esse non siano su altre strutture dati. Il secondo punto riguarda la dimensione delle collezioni di file XML: poiché in un file XML la struttura è ripetuta in maniera estensiva per ognuno degli oggetti presenti nel file (si veda ad esempio la Figura 1), i documenti a cui è applicato lo schema XML hanno una dimensione molto maggiore di quella che avrebbe un documento contenente informazione non strutturata. Infine, le *query* in XML sono più complicate di quelle possibili in SQL, includendo ad esempio vincoli sul cammino della struttura ad albero oppure ricerca di sottostringhe sul contenuto dei vari campi.

1.2 Rappresentazione di documenti XML tramite alberi DOM

Il *DOM* (*Document Object Model*) [DOM] è una struttura (più opportunamente un oggetto) che permette la rappresentazione e gestione di documenti XML. Come è consuetudine nella programmazione ad oggetti, la struttura interna del DOM non è descritta, ma è definita solo la sua interfaccia verso l'esterno. Una possibile implementazione della struttura interna utilizza un albero $\mathcal{T}$ (*albero DOM*) ordinato ed etichettato (vedi Figura 1). Le relazioni tra i nodi dell'albero e le componenti del documento XML d sono:

- Ogni tag di apertura <t> genera un **tag node** con etichetta <t.

- Ogni attributo a genera un **attribute node** con etichetta @a.

- Ogni PCDATAρ di un tag generano due nodi: il **text-skip node** e il **content node** rispettivamente con etichetta = e $\emptyset\rho$, dove $\emptyset$ non appartiene all'alfabeto di d.

Sia $\rightarrow$ la relazione "*padre di*" e σ una sottostringa ben formata di d; la costruzione di $\mathcal{T}$ è ricorsiva:

- Se σ è il contenuto testuale di un nodo allora $\mathcal{T}$ è composto da due nodi: $= \rightarrow \emptyset\sigma$.

- Se $\sigma =$<t $a_1 = \rho_1 \ldots a_k{=}\rho_k$>$\tau$</t>, allora l'albero è composto da un tag node <t con m figli: k di questi sono i sottoalberi associati agli attributi e ai rispettivi valori, ovvero

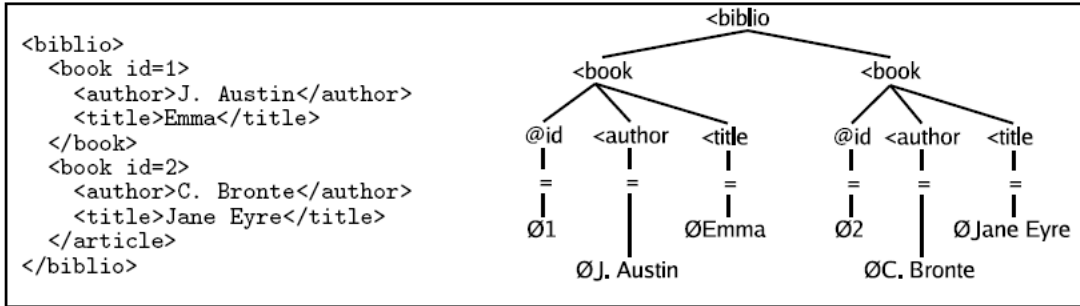


Figura 1: Un esempio di codice XML e il *DOM tree* associato

$@a_i \rightarrow \emptyset\rho_i$; i rimanenti $m-k$ sono i sottoalberi ottenuti ricorsivamente dalle stringhe dei tag annidati in τ.

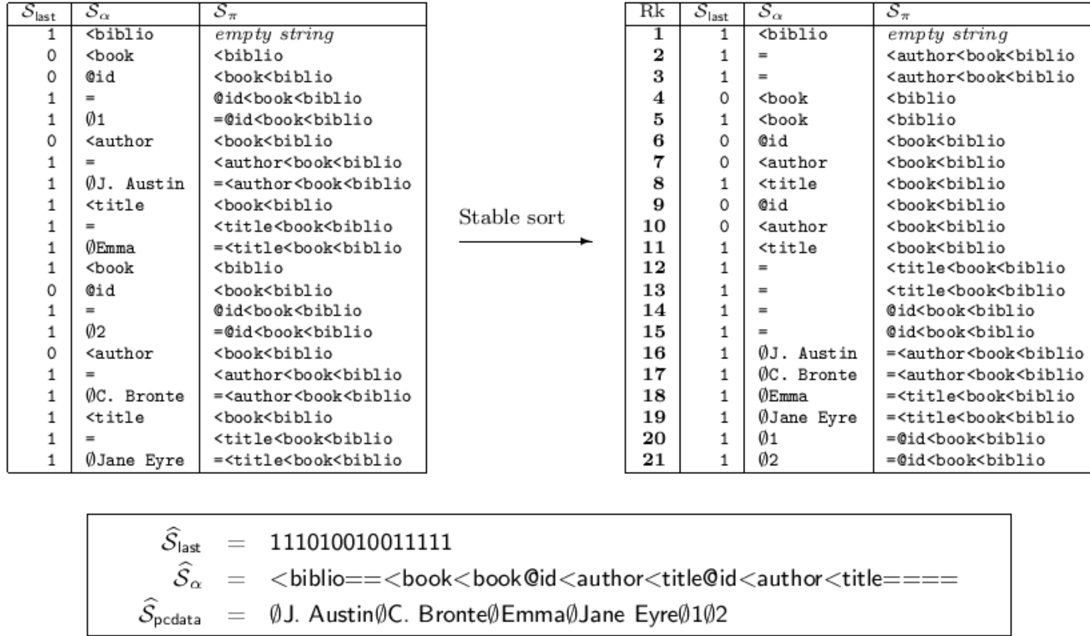
È interessante notare che la struttura di d è memorizzata nei nodi interni, mentre le foglie contengono solo i PCDATA. Inoltre, queste ultime sono riconoscibili perché la propria etichetta inizia sempre con il simbolo $\emptyset$.

2 La trasformata XBW

La trasformata XBW permette di rappresentare un albero in una nuova struttura più facilmente comprimibile. Essa è stata introdotta in [FLMM05] per un generico albero etichettato e in [FLMM06] per il *DOM tree*; quest'ultima versione verrà presentata qui di seguito.

Sia $t = n + l$ il numero totale di nodi dell'albero DOM $\mathcal{T}$ associato al documento XML d; in particolare, n indica il numero di nodi interni, mentre l il numero di foglie. Per ogni nodo $u \in \mathcal{T}$ si indichi con $\alpha[u]$, $\pi[u]$ e $\text{LAST}[u]$ rispettivamente: l'etichetta del nodo u; la stringa ottenuta concatenando l'etichetta dei nodi nel percorso radice-u capovolto (per la radice è la stringa vuota ε); una *flag* binaria per indicare la posizione di u rispetto ai suoi fratelli, il cui valore è 1 se u è il figlio posizionato più a destra rispetto al proprio padre, 0 altrimenti (per la radice vale 1). Ad esempio, per il nodo u con etichetta $\emptyset$J.Austin in Figura 1: $\alpha[u] = \emptyset\text{J.Austin}$, $\pi[u] = \text{=<author<book<biblio}$, $\text{LAST}[u] = 1$.

Per ogni nodo u dell'albero si definisce una tripletta $s[u] =< \text{LAST}[u], \alpha[u], \pi[u] >$ che verrà inserita in un multi-insieme ordinato $\mathcal{S}$; inoltre, si definiscono con $\mathcal{S}_{\text{LAST}}[i], \mathcal{S}_\alpha[i], \mathcal{S}_\pi[i]$ le rispettive componenti della tripletta nella posizione i di $\mathcal{S}$ ($\mathcal{S}[i]$). Di seguito si indicherà un nodo con il suo indice in $\mathcal{S}$ (*rank*). La trasformata XBW di $\mathcal{T}$ è descritta nel seguente pseudocodice:



$\mathcal{S}_{\text{last}}$	$\mathcal{S}_\alpha$	$\mathcal{S}_\pi$
1	<biblio	*empty string*
0	<book	<biblio
0	@id	<book<biblio
1	=	@id<book<biblio
1	∅1	=@id<book<biblio
0	<author	<book<biblio
1	=	<author<book<biblio
1	∅J. Austin	=<author<book<biblio
1	<title	<book<biblio
1	=	<title<book<biblio
1	∅Emma	=<title<book<biblio
1	<book	<biblio
0	@id	<book<biblio
1	=	@id<book<biblio
1	∅2	=@id<book<biblio
0	<author	<book<biblio
1	=	<author<book<biblio
1	∅C. Bronte	=<author<book<biblio
1	<title	<book<biblio
1	=	<title<book<biblio
1	∅Jane Eyre	=<title<book<biblio

Stable sort →

Rk	$\mathcal{S}_{\text{last}}$	$\mathcal{S}_\alpha$	$\mathcal{S}_\pi$
1	1	<biblio	*empty string*
2	1	=	<author<book<biblio
3	1	=	<author<book<biblio
4	0	<book	<biblio
5	1	<book	<biblio
6	0	@id	<book<biblio
7	0	<author	<book<biblio
8	1	<title	<book<biblio
9	0	@id	<book<biblio
10	0	<author	<book<biblio
11	1	<title	<book<biblio
12	1	=	<title<book<biblio
13	1	=	<title<book<biblio
14	1	=	@id<book<biblio
15	1	=	@id<book<biblio
16	1	∅J. Austin	=<author<book<biblio
17	1	∅C. Bronte	=<author<book<biblio
18	1	∅Emma	=<title<book<biblio
19	1	∅Jane Eyre	=<title<book<biblio
20	1	∅1	=@id<book<biblio
21	1	∅2	=@id<book<biblio

$\widehat{\mathcal{S}}_{\text{last}}$ = 111010010011111
$\widehat{\mathcal{S}}_\alpha$ = <biblio==<book<book@id<author<title@id<author<title====
$\widehat{\mathcal{S}}_{\text{pcdata}}$ = ∅J. Austin∅C. Bronte∅Emma∅Jane Eyre∅1∅2

Figura 2: La trasformata XBW applicata all'albero nell'esempio 1

Algorithm 1: XBW($\mathcal{T}$)

1.1 **begin**
1.2 Visita $\mathcal{T}$ in *pre-order*[1] e per ogni nodo u inserisci in $\mathcal{S}$ la tripletta $s[u] =< \text{LAST}[u], \alpha[u], \pi[u] >$;
1.3 Ordina stabilmente $\mathcal{S}$ rispetto alla componente π;
1.4 $\widehat{\mathcal{S}}_{\text{LAST}} \leftarrow \mathcal{S}_{\text{LAST}}[1, n]$;
1.5 $\widehat{\mathcal{S}}_\alpha \leftarrow \mathcal{S}_\alpha[1, n]$;
1.6 $\widehat{\mathcal{S}}_{\text{PCDATA}} \leftarrow \mathcal{S}_\alpha[n+1, t]$;
1.7 XBW$\leftarrow< \widehat{\mathcal{S}}_{\text{LAST}}, \widehat{\mathcal{S}}_\alpha, \widehat{\mathcal{S}}_{\text{PCDATA}} >$;
1.8 **return** XBW;
1.9 **end**

L'albero viene inizialmente visitato in *pre-order* e, per ogni nodo u analizzato, la tupla $s[u]$ viene inserita nell'insieme $\mathcal{S}$. L'ordinamento deve essere di tipo lessicografico, stabile, e indicando con $<_r$ la relazione d'ordine deve valere che: $\varepsilon <_r \texttt{<} <_r \texttt{@} <_r \texttt{=}$[2]. La stabilità permette di mantenere l'ordine relativo introdotto dalla visita in *pre-order* delle tuple con componenti π uguali. Di seguito, $\mathcal{S}$ indicherà il multi-insieme dopo l'ordinamento lessicografico, se non diversamente specificato. Come si può notare dallo pseudocodice, il risultato della trasformata

[1]Visita in *pre-order*: si visita prima il contenuto del nodo e poi, ricorsivamente, tutti i sottoalberi da sinistra a destra.

[2]Si noti che il nuovo simbolo ∅ non è mai presente nella componente π.

non include $\mathcal{S}_\pi[1,t]$, ovvero il percorso tra il nodo e la radice; questa informazione è, infatti, ricavabile dalle proprietà di $\widehat{\mathcal{S}}_{\text{LAST}}$ e $\widehat{\mathcal{S}}_\alpha$. Il vettore delle componenti π è utilizzato solo nella fase di ordinamento, ma può essere sostituito da puntatori verso i nodi di $\mathcal{T}$: ciò diminuisce sostanzialmente lo spazio richiesto dall'algoritmo XBW (lineare invece che quadratico nel numero di nodi).

Il multi-insieme $\mathcal{S}$ gode delle seguenti caratteristiche: $\mathcal{S}_{\text{LAST}}$ ha n *flag* con valore pari ad 1 e $t-n=l$ *flag* a 0; $\mathcal{S}_\alpha$ contiene tutte le etichette dei nodi; $\mathcal{S}_\pi$ contiene tutti i percorsi verso la radice. Inoltre valgono le seguenti proprietà strutturali su $\mathcal{S}$:

1. $\mathcal{S}_{\text{LAST}}[n+1,t] = \overline{1}^T$, $\widehat{\mathcal{S}}_\alpha$ contiene solo nodi interni e $\widehat{\mathcal{S}}_{\text{PCDATA}}$ solo contenuti testuali e valori di attributo (ovvero i PCDATA).
   Prova: Nell'ordinamento lessicografico il carattere = è *maggiore* rispetto a tutti i rimanenti simboli dell'alfabeto di d, il percorso verso la radice di una foglia inizia sempre con= per costruzione e la componente π dei nodi interni ha sempre prefisso diverso da =. Da ciò si conclude che tutte le foglie verranno posizionate nelle tuple finali di $\mathcal{S}$. Questo spiega perché $\mathcal{S}_{\text{LAST}}[n+1,t]$ non appartiene all'output della trasformata.

2. La prima tripletta di $\mathcal{S}$ è associata alla radice, infatti $\mathcal{S}_\pi[1] = \varepsilon$.

3. Siano v_1 e v_2 due nodi tali che $\mathcal{S}_\pi[v_1] = \mathcal{S}_\pi[v_2]$. Allora i due nodi si trovano allo stesso livello di $\mathcal{T}$ e v_1 si trova alla sinistra di v_2 in $\mathcal{T}$ se e solo se v_1 precede v_2 in $\mathcal{S}$.
   Prova: È evidente che i due nodi si trovano allo stesso livello perché la componente π rappresenta il percorso (unico per ogni nodo) nodo-radice. La seconda parte è garantita dalla visita iniziale in *pre-order* di $\mathcal{T}$ e dall'ordinamento stabile.
   Esempio: Si considerino le due triplette $\mathcal{S}[6]$ e $\mathcal{S}[9]$ nella Figura 2: entrambe corrispondono a due nodi con profondità 2 e la prima tupla corrisponde al nodo di sinistra, mentre la seconda a quello di destra (Figura 1).

4. Siano $u_1, \dots u_k$ figli del nodo u presi da sinistra a destra, allora $s[u_1] \dots s[u_k]$ giacciono in $\mathcal{S}$ in quest'ordine e adiacentemente. Inoltre, $\mathcal{S}_{\text{LAST}}[u_k] = 1$ e $\mathcal{S}_{\text{LAST}}[u_i] = 0$, $i \neq k$.
   Prova: La visita iniziale in *pre-order* dispone i figli in $\mathcal{S}$ nell'ordine sinistra-destra, senza interporre tra questi alcuna tripletta con la stessa componente π. Poiché i k nodi hanno la stessa componente π, l'ordinamento li unirà in k posizioni adiacenti e manterrà l'ordine relativo per la stabilità.
   Esempio: Il nodo 4 con etichetta <book ha tre figli contenuti in $\mathcal{S}[6,8]$; inoltre $\mathcal{S}_{\text{LAST}}[8] = 1$ mentre $\mathcal{S}_{\text{LAST}}[6] = \mathcal{S}_{\text{LAST}}[7] = 0$.

5. Siano v_1 e v_2 due nodi con la stessa etichetta ($\mathcal{S}_\alpha[v_1] = \mathcal{S}_\alpha[v_2] = \beta$); se $\mathcal{S}[v_1]$ precede $\mathcal{S}[v_2]$, allora tutti i figli di v_1 precedono i figli di v_2 in $\mathcal{S}$.
   Prova: si indichi con u_1 e u_2 rispettivamente un figlio di v_1 e uno di v_2; per la definizione di π, $\mathcal{S}_\pi[u_1] = \beta\mathcal{S}_\pi[v_1]$ e $\mathcal{S}_\pi[u_2] = \beta\mathcal{S}_\pi[v_2]$; per ipotesi $\mathcal{S}_\pi[v_1] \leq \mathcal{S}_\pi[v_2]$, quindi i figli di

v_1 precedono quelli di v_2 in $\mathcal{S}$.

Esempio: Si considerino i nodi 4 e 5 con etichetta <book nell'esempio in Figura 2: i rispettivi figli si trovano in $\mathcal{S}[6, 8]$ e $\mathcal{S}[9, 11]$.

La trasformata XBW del documento d occupa, nel caso peggiore, $(17/8)n + l$ byte in più rispetto alla taglia di d, ma in media la dimensione è il 90% della taglia iniziale. In [FLMM05] è presentata la trasformata XBW inversa per un albero generico (anche non DOM), la quale permette di ricostruire l'albero data la sua trasformata $< \mathcal{S}_{\text{LAST}}, \mathcal{S}_\alpha >$ in tempo lineare nel numero di nodi. Dato che $< \mathcal{S}_{\text{LAST}}, \mathcal{S}_\alpha >$ è facilmente ottenibile da $< \widehat{\mathcal{S}}_{\text{LAST}}, \widehat{\mathcal{S}}_\alpha, \widehat{\mathcal{S}}_{\text{PCDATA}} >$, questa procedura permette anche di ottenere l'albero DOM dalla sua trasformata. Si rimanda all'articolo originale per maggiori dettagli.

Come è già stato detto, la trasformata XBW non è progettata per comprimere un albero, ma per rappresentarlo in una forma più facilmente comprimibile. A tal scopo, la trasformata di $\mathcal{T}$ gode della proprietà di *omogeneità locale*, ovvero tende a raggruppare etichette racchiuse dagli stessi *tag*. Per chiarezza si guardi nuovamente l'esempio in Figura 2: per le proprietà di $\mathcal{S}$, tuple con lo stesso prefisso della componente π vengono raggruppate in *cluster*; ma questo implica che le triplette nei *cluster* abbiano etichette *simili* tra loro perché rappresentano aree semantiche simili. Ad esempio, le tuple con percorso =@id<book<biblio conterranno numeri perché rappresentano codici identificativi di libri, mentre le triplette con percorso =<title<book<biblio conterranno principalmente termini in lingua inglese. Infine, si guardino le etichette dei nodi con percorso <biblio: sono sempre uguali a <book. Queste caratteristiche possono essere sfruttate dai tradizionali algoritmi di compressione, che migliorano le loro prestazioni se il documento da comprimere contiene blocchi omogenei (si pensi alla *finestra* di LZ77 [ZL77] o alle versioni di LZ78 [ZL78] che svuotano il dizionario quando questo è pieno). Inoltre ogni *cluster* può essere codificato con un algoritmo ottimizzato per il suo contenuto.

3 Operazioni di navigazione

Le operazioni di navigazione in un documento XML (cioè l'interfaccia di un oggetto DOM) possono essere eseguite efficientemente anche sulla trasformata XBW del documento; in particolare, le operazioni principali sono: `GetChildren`, `GetParent`. Come si vedrà di seguito, entrambe sono riconducibili alle operazioni fondamentali di `rank` e `select`, la cui implementazione verrà descritta nella sezione 5.3. $\texttt{rank}_c(A, q)$ calcola la frequenza del simbolo c nel vettore $A[1, q]$; $\texttt{select}_c(A, q)$, invece, restituisce la posizione in A della q-esima occorrenza di c.

Si supponga di disporre delle seguenti strutture ausiliarie:

- Una tabella H: per ogni etichetta β di un nodo interno, $H[\beta]$ rappresenta la posizione (*rank*) di β nell'ordinamento lessicografico delle etichette dei nodi interni (prese senza

doppioni). Poiché il numero di etichette distinte è basso (corrisponde al numero di tag e attributi), questa tabella occupa uno spazio limitato. Per l'esempio in Figura 2:

Etichetta β	<author	<biblio	<book	<title	@id
$H[\beta]$	1	2	3	4	5

Con $H^{-1}[i]$ si indicherà l'etichetta con rango i.

- L'array associativo[3] F: per ogni nodo interno con etichetta β si definisca $F[\beta]$ come il *rank* della prima tripletta in $\mathcal{S}$ in cui la componente π ha prefisso β. Si utilizza un array associativo perché β rappresenta una stringa, ma si può ricorrere ad un array tradizionale se si utilizza la tabella di *ranking* descritta precedentemente. Per l'esempio in Figura 2:

Etichetta β	<author	<biblio	<book	<title	@id
$F[\beta]$	2	4	6	12	14

- L'array binario E: $E[i] = 1$, $1 \leq i \leq n$, se e solo se esiste un nodo con etichetta β tale che $F[\beta] = i$ (per la radice $E[1] = 0$). Più semplicemente, E evidenzia le posizioni i di $\mathcal{S}$ in cui l'etichetta del primo nodo del percorso $\mathcal{S}_\pi[i]$ è diversa da quella di $\mathcal{S}_\pi[i-1]$. Lo spazio richiesto da questo array è di n bit. Per l'esempio in Figura 2:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E[i]	0	1	0	1	0	1	0	0	0	0	0	1	0	1	0

3.1 `GetChildren`(i)

`GetChildren`(i) restituisce l'intervallo (**First**,**Last**) di $\mathcal{S}$ che contiene i figli del nodo nella posizione i di $\mathcal{S}$.

[3]Un array è associativo se ogni cella è indicizzata da una stringa invece che da un intero. Il linguaggio di programmazione PHP possiede questa struttura dati.

Algorithm 2: `GetChildren`(i)

```
2.1  begin
2.2      if (i > n) then
2.3          return NIL;                                      // i è una foglia
2.4      end
2.5      c ← Ŝ_α[i];
2.6      k ← rank_c(Ŝ_α[i], i);
2.7      y ← F[c];
2.8      z ← rank_1(Ŝ_LAST, y − 1);
2.9      if (y <= n) then
2.10         First ← select_1(Ŝ_LAST, z + k − 1) + 1;        // ovvero c è diverso da =
2.11         Last ← select_1(Ŝ_LAST, z + k);
2.12     end
2.13     else
             // Si esegue la ricerca in Ŝ_PCDATA
2.14         First ← Last ← select_1(Ŝ_PCDATA, k);
2.15     end
2.16     return (First, Last);
2.17 end
```

L'etichetta del nodo i, ovvero c (linea 2.5), compare k volte in $\widehat{\mathcal{S}}_\alpha[1,i]$ (linea 2.6) e appare per la prima volta in $\mathcal{S}[y]$ come prefisso della componente π (quest'ultima affermazione segue dalla definizione di F, linea 2.7); z rappresenta il numero di triplette con *flag* uguale a 1 prima di y (linea 2.8). Per le caratteristiche del multi-insieme, tutti i figli di i occupano posizioni adiacenti comprese tra $\mathcal{S}[\mathbf{First}]$ e $\mathcal{S}[\mathbf{Last}]$, estremi inclusi (proprietà 4). $\mathcal{S}[\mathbf{First}]$ è la tupla successiva alla $(k-1)$-esima tripletta con *flag* LAST uguale ad 1 partendo da y; $\mathcal{S}[\mathbf{Last}]$, invece, è la k-esima tupla con *flag* ad 1 partendo da y. Poiché `select` conta le occorrenze partendo dalla prima posizione di $\widehat{\mathcal{S}}_{\text{LAST}}$, i valori **First** e **Last** devono essere ridefiniti rispetto alla posizione iniziale: $\mathcal{S}[\mathbf{First}]$ è la tupla successiva alla $(z+k-1)$-esima tripletta con *flag* LAST uguale ad 1; analogamente, $\mathcal{S}[\mathbf{Last}]$ è la $(z+k-1)$-esima tripletta con *flag* pari a 1 (linee 2.10-2.11). Particolare attenzione bisogna porre quando i è uno *skip-node*: in questo caso l'*unico* figlio si trova nella k-esima posizione di $\widehat{\mathcal{S}}_{\text{PCDATA}}$.

Esempio: Si supponga di voler cercare i figli del nodo $i = 5$ in Figura 2. L'etichetta di $\mathcal{S}[5]$ è $c =$<book: questa compare $k = 2$ volte in $\mathcal{S}[1,5]$ e la sua prima occorrenza come prefisso di una componente π è nella posizione $y = 6$. I suoi figli saranno quindi inclusi tra $\mathcal{S}[9]$ e $\mathcal{S}[11]$.

3.2 `GetParent`(i)

`GetParent`(i) restituisce il padre del nodo nella posizione i di $\mathcal{S}$. Questo metodo è leggermente più complesso rispetto a `GetChildren` perché richiede la conoscenza della componente π, che

non è contenuta nella trasformata. Si ricorre quindi al vettore ausiliario E definito precedentemente.

Algorithm 3: `GetParent`(i)

```
3.1  begin
3.2      if (i = 1) then
3.3          return NIL ;                                     // i è la radice
3.4      end
3.5      if (i ≤ n) then
3.6          c ← rank_1(E, i);                  // i è un nodo interno non radice
3.7          β ← H^{-1}[c];
3.8          y ← F[β] ;
3.9          k ← rank_1(Ŝ_LAST, i − 1) − rank_i(Ŝ_LAST, y − 1);
3.10         p ← select_β(Ŝ_α, k + 1);
3.11     end
3.12     else
3.13         n ← numero nodi interni;                        // i è una foglia
3.14         p ← select_=(Ŝ_α, i − n);
3.15     end
3.16     return p;
3.17 end
```

Si supponga che i sia un nodo interno e β l'etichetta del padre. Per la definizione di E, la linea 3.6 calcola il numero c di etichette distinte che compaiono come prefisso nelle componenti π delle tuple in $\mathcal{S}[i,j]$; ma c è anche il *ranking* di β rispetto a tutte le etichette dei nodi interni. Utilizzando la tabella H è possibile ricavare $\beta = H^{-1}(c)$ (linea 3.7), mentre tramite F è possibile ricavare la prima tupla $\mathcal{S}[y]$ con prefisso β (linea 3.8). k è il numero di *flag* con valore 1 tra le posizioni y e $i-1$ di $\mathcal{S}$, ma per le proprietà precedenti il padre di i è la $k+1$-esima occorrenza di β in $\widehat{\mathcal{S}}_\alpha$ (linee 3.9 e 3.10). Nel caso i sia un figlio, non si possono utilizzare le strutture dati precedenti perché la posizione i non è inclusa né in $\widehat{\mathcal{S}}_{\text{LAST}}$ né in $\widehat{\mathcal{S}}_\alpha$. Si ottiene facilmente che β è uguale a = e il padre di i è la $(i-n)$-esima occorrenza di = in $\widehat{\mathcal{S}}_\alpha$ (linee 3.13 e 3.14), dove n è il numero di nodi interni.

Esempio: Si supponga di voler calcolare il padre di $\mathcal{S}[10]$: si ottiene che $c=3$, $H[c] =$ `<book`, $y=6$ $k=1$; quindi il padre di $\mathcal{S}[10]$ è $\mathcal{S}[5]$. Invece, il padre di $\mathcal{S}[16]$ è $\mathcal{S}[2]$ perchè $i=16$ e $n=15$ (numero di nodi interni).

Si noti che tutte le operazioni sono definite su $\mathcal{S}$, ma la loro implementazione utilizza solo le strutture ausiliarie e $\widehat{\mathcal{S}}_{\text{LAST}}$, $\widehat{\mathcal{S}}_\alpha$ e $\widehat{\mathcal{S}}_{\text{PCDATA}}$, ovvero l'output della trasformata di $\mathcal{T}$. Inoltre, `GetChildren` e `GetParent` utilizzano un numero costante di chiamate a `rank` e `select` su $\widehat{\mathcal{S}}_{\text{LAST}}$ e $\widehat{\mathcal{S}}_\alpha$.

4 Operazioni di *path expression search*

Indicando con $\Pi = q_1 \cdots q_k$ la concatenazione delle etichette corrispondenti ad un sottocammino di $\mathcal{T}$, le operazioni di *path expression search* di interesse per un documento XML sono:

- `SubPathSearch`(Π): dato Π, trovare l'insieme dei nodi $\mathcal{N}$ discendenti da Π
- `ContentSearch`(Π, γ): dato Π ed una stringa γ, trovare l'insieme delle foglie di $\mathcal{T}$ tali che il cammino dalla radice al nodo sia Π= e che il PCDATA corrispondente contenga γ.

Da notare è il fatto che in entrambe le operazioni il sottocammino Π non deve essere necessariamente "ancorato" alla radice dell'albero.

4.1 `SubPathSearch`(Π)

Ci si può convincere facilmente che ciò che si vuole restituire è l'insieme di tutti i nodi u di $\mathcal{T}$ il cui cammino verso la radice (cioè il cammino ottenuto concatenando le etichette dei nodi che si trovano spostandosi da u verso la radice) abbia come prefisso $\Pi^R = q_k q_{k-1} \cdots q_2 q_1$. Si deve tener presente che, essendo l'albero generato a partire da un file XML, una ricerca del tipo `SubPathSearch`($\Pi = q_1 q_2 \cdots q_{k-1}$=) non è supportata, nel senso che il simbolo = è inserito come *label* di un nodo solamente per costruire la XBW. Per questo si assume che l'ultimo simbolo del cammino $\Pi = q_1 q_2 \cdots q_{k-1} q_k$ passato in input sia sempre diverso da =; formalmente, si pone la seguente ipotesi: $q_k \neq =$. Questa ipotesi permette di facilitare la ricerca, poiché è necessario accedere solamente agli array $\widehat{\mathcal{S}}_{\text{PCDATA}}$ e $\widehat{\mathcal{S}}_{\alpha}$ e, come spiegato sopra, essa non limita le potenzialità della procedura. Nel caso in cui $q_k = =$ la procedura `SubPathSearch`(Π) indicherà che nessun sottocammino con etichette Π è presente in $\mathcal{T}$. Dato il *sorting* effettuato su $\mathcal{S}$, le triplette che rappresentano i nodi da restituire si trovano in locazioni contigue in $\mathcal{S}$. Si indichi l'intervallo di posizioni in $\mathcal{S}$ occupate da tali triplette con $\mathcal{S}[\mathbf{First}, \mathbf{Last}]$; in questo modo $\mathcal{S}_\pi[\mathbf{First}, \mathbf{Last}]$ sono esattamente le tuple di $\mathcal{S}_\pi$ aventi come prefisso $\Pi^R = q_k q_{k-1} \cdots q_1$.

Come si può notare dallo pseudocodice, l'Algoritmo 4 trova l'intervallo $[\mathbf{First}, \mathbf{Last}]$ in k fasi distinte. L'invariante mantenuto è: alla fine della i-esima fase il parametro **First** punta alla prima entry di $\mathcal{S}$ tale che $\mathcal{S}_\pi[\mathbf{First}]$ ha come prefisso $q_i \cdots q_1$ (linea 4.8) e il parametro **Last** punta all'ultima entry di $\mathcal{S}$ tale che $\mathcal{S}_\pi[\mathbf{Last}]$ ha come prefisso $q_i \cdots q_1$ (linea 4.10). Si veda [FLMM05] per la dimostrazione. Si vede così come l'operazione di `SubPathSearch` richieda un numero di `rank` e di `select` proporzionale al numero k di etichette contenute nel sottocammino $\Pi = q_1 \cdots q_k$.

4.2 `ContentSearch`(Π, γ)

La spiegazione dettagliata di come può essere eseguita in maniera efficiente tale operazione è riportata in seguito, visto che essa dipende dalla struttura dati utilizzata per memorizzare

Algorithm 4: SubPathSearch($\Pi = q_1 q_2 \cdots q_k$)

```
4.1  begin
4.2      i ← 1;
4.3      First←F[q_1]; Last←F[H^{-1}[H[q_1] + 1]]−1;
4.4      while (First≤Last) and (i ≤ k − 1) do
4.5          i ← i + 1;
4.6          z ← rank_1(Ŝ_LAST, F[q_i] − 1);
4.7          k_1 ← rank_{q_i}(Ŝ_α, First − 1);
4.8          First ← select_1(Ŝ_LAST, z + k_1) + 1 ;
4.9          k_2 ← rank_{q_i}(Ŝ_α, Last);
4.10         Last ← select_1(Ŝ_LAST, z + k_2);
4.11     end
4.12     if (First > Last) then return NIL;
         // Π = q_1 q_2 ··· q_k non è presente in T oppure q_k ==
4.13     else return (First,Last);
4.14 end
```

$\widehat{\mathcal{S}}_{\text{PC}_{\text{DATA}}}$; ad un primo livello si può comunque pensare che per una ContentSearch(Π, γ) si esegua per prima cosa un SubPathSearch(Π), ottenendo così un'insieme di nodi $\mathcal{N}$, e successivamente si vada a cercare quale tra i nodi di $\mathcal{N}$ sia collegato tramite uno *skip-node* ad una foglia che contiene γ come sottostringa.

5 Implementazione

In questa sezione vengono spiegati i dettagli con cui vengono implementati i metodi discussi nelle sezioni precedenti. Si supponga di avere a disposizione le tre strutture dati descritte nella Sezione 3.

5.1 Computazione di XBW

Per quanto riguarda la computazione di XBW, una volta costruita la tabella $\mathcal{S}$ è necessario ordinare in maniera stabile le sue entry utilizzando come chiavi per l'ordinamento $\mathcal{S}_\pi$. In [FLMM05] è stato dimostrato che un tale ordinamento effettuato su un qualsiasi albero $\mathcal{T}$ i cui nodi contengono delle etichette può essere fatto in tempo lineare rispetto al numero di nodi di $\mathcal{T}$ (e quindi lineare rispetto al numero di entry di $\mathcal{S}$). Per l'implementazione di tale ordinamento applicato al caso di documenti XML è stato utilizzato un approccio più semplice: si rappresenta $\mathcal{S}_\pi$ tramite un array di puntatori ai nodi di $\mathcal{T}$ e l'ordinamento di $\mathcal{S}$ si ottiene operando su tale array di puntatori. Si perde così l'ottimalità asintotica nella complessità dell'ordinamento, che risulta essere la fase che comporta il maggior sforzo computazionale nel calcolo di XBW.

5.2 XBZip: compressione di XBW

Nel caso in cui si sia interessati solamente alla compressione di XBW, senza voler poi effettuare le operazioni di ricerca e navigazione sulla rappresentazione compressa, è sufficiente memorizzare gli array $\widehat{\mathcal{S}}_{\textsc{last}}$, $\widehat{\mathcal{S}}_\alpha$ e $\widehat{\mathcal{S}}_{\textsc{PCdata}}$ nella maniera più efficiente possibile. Sperimentalmente si vede che è più conveniente rappresentare gli array $\widehat{\mathcal{S}}_{\textsc{last}}$ e $\widehat{\mathcal{S}}_\alpha$ come un unico array $\widehat{\mathcal{S}}'_\alpha$ ottenuto da $\widehat{\mathcal{S}}_\alpha$ aggiungendo un'etichetta </ in corrispondenza dei bit uguali ad 1 in $\widehat{\mathcal{S}}_{\textsc{last}}$. Riprendendo l'esempio in Figura 2 si ricava:

$$\widehat{\mathcal{S}}'_\alpha = \text{<biblio</=</=</<book<book</@id<author<title</@id} \\ \text{<author<title</=</=</=</=</}$$

Questa strategia permette di tener conto in fase di compressione della ripetitività che può essere presente nella struttura dell'albero. Come software per la compressione di $\widehat{\mathcal{S}}'_\alpha$ e $\widehat{\mathcal{S}}_{\textsc{PCdata}}$ è stato utilizzato il compressore generico PPMDI[Shk02].

Algorithm 5: XBZip(d)

5.1 **begin**
5.2 calcola l'albero $\mathcal{T}$ di d;
5.3 $\langle \widehat{\mathcal{S}}_{\textsc{last}}, \widehat{\mathcal{S}}_\alpha, \widehat{\mathcal{S}}_{\textsc{PCdata}} \rangle \leftarrow \mathsf{XBW}(\mathcal{T})$;
5.4 ricava $\widehat{\mathcal{S}}'_\alpha$ a partire da $\widehat{\mathcal{S}}_\alpha$ e da $\widehat{\mathcal{S}}_{\textsc{last}}$;
5.5 comprimi separatamente $\widehat{\mathcal{S}}'_\alpha$ e $\widehat{\mathcal{S}}_{\textsc{PCdata}}$ utilizzando PPMDI;
5.6 **end**

Il *tool* prodotto per la compressione di XBW è XBZip e il suo pseudocodice è descritto dall'Algoritmo 5.

5.3 XBZipIndex: compressione, ricerca e navigazione

Come visto precedentemente, per supportare la ricerca e la navigazione, oltre a computare XBW è necessario utilizzare delle strutture dati che permettano di effettuare in maniera efficiente le operazioni di `rank` e `select` su $\widehat{\mathcal{S}}_{\textsc{last}}$ e $\widehat{\mathcal{S}}_\alpha$. In [FLMM05] sono descritte delle strutture dati generali per alberi con etichette che garantiscono delle prestazioni buone (spesso ottime) in termini di comportamento asintotico al caso peggiore. Invece vengono qui descritte delle strutture dati più semplici ottimizzate per l'applicazione ai documenti XML, che, pur facendo perdere l'ottimalità asintotica, permettono di ottenere dei buoni risultati nei casi reali. L'idea di base è quella di vedere gli array $\widehat{\mathcal{S}}_{\textsc{last}}$ e $\widehat{\mathcal{S}}_\alpha$ come stringhe, utilizzando quindi delle strutture dati che permettono di mantenere degli indici su tali stringhe. Lo pseudocodice della procedura che crea tali strutture dati è riportato nell'Algoritmo 6, che descrive il tool XBZipIndex.

Vengono ora forniti i dettagli per per ognuno degli array $\widehat{\mathcal{S}}_{\textsc{last}}, \widehat{\mathcal{S}}_\alpha, \widehat{\mathcal{S}}_{\textsc{PCdata}}$.

Algorithm 6: $\mathsf{XBZipIndex}(d)$

6.1 **begin**
6.2 calcola l'albero $\mathcal{T}$ di d;
6.3 $\langle \widehat{\mathcal{S}}_{\text{LAST}}, \widehat{\mathcal{S}}_{\alpha}, \widehat{\mathcal{S}}_{\text{PCDATA}} \rangle \leftarrow \mathsf{XBW}(\mathcal{T})$;
6.4 memorizza $\widehat{\mathcal{S}}_{\text{LAST}}$ utilizzando una rappresentazione compressa che supporti le operazioni di `rank` e `select`;
6.5 memorizza $\widehat{\mathcal{S}}_{\alpha}$ utilizzando una rappresentazione compressa che supporti le operazioni di `rank` e `select`;
6.6 suddividi $\widehat{\mathcal{S}}_{\text{PCDATA}}$ in *bucket* tali che due elementi stiano nello stesso *bucket* se e solo se i due elementi hanno lo stesso cammino risalendo verso la radice;
6.7 comprimi ogni *bucket* separatamente usando un FM-INDEX;
6.8 **end**

5.3.1 $\widehat{\mathcal{S}}_{\text{LAST}}$: strutture dati ausiliarie

Come si può notare dallo pseudocodice delle operazioni di navigazione e di ricerca, le uniche operazioni che devono essere eseguite sull'array $\widehat{\mathcal{S}}_{\text{LAST}}$ sono $\texttt{rank}_1$ e $\texttt{select}_1$. Quindi è sufficiente utilizzare uno schema di memorizzazione con un livello di *bucket* (*one-level bucketing*), operando nel modo seguente: si definisce una costante L opportuna (in [FLMM06] si pone $L = 1000$ per default) e si partiziona $\widehat{\mathcal{S}}_{\text{LAST}}$ in blocchi di lunghezza variabile, ognuno dei quali deve contenere esattamente L bit posti ad 1 (tranne eventualmente il blocco contenente la parte finale di $\widehat{\mathcal{S}}_{\text{LAST}}$). Per ognuno di questi blocchi si memorizza:

- la lunghezza LP del prefisso di $\widehat{\mathcal{S}}_{\text{LAST}}$ memorizzato nei blocchi precedenti;
- il *1-blocked rank* del blocco, cioè il numero di 1 che sono contenuti nella porzione di $\widehat{\mathcal{S}}_{\text{LAST}}$ che precede questo blocco;
- il puntatore alla rappresentazione compressa del blocco ottenuta tramite GZIP.

In Figura 3 è riportato lo schema di memorizzazione per l'esempio trattato precedentemente con $L = 3$.

Come si può facilmente verificare, per eseguire l'operazione di $\texttt{select}_1$ su $\widehat{\mathcal{S}}_{\text{LAST}}$ è sufficiente eseguire una ricerca binaria sulla tabella che mantiene i valori dei *1-blocked rank* per i vari blocchi, seguita dalla decompressione e scansione di un unico blocco (quello individuato a seguito della ricerca binaria). La stessa procedura vale per $\texttt{rank}_1$ applicando, però, la ricerca binaria ad LP.

5.3.2 $\widehat{\mathcal{S}}_{\alpha}$: strutture dati ausiliarie

Anche per $\widehat{\mathcal{S}}_{\alpha}$ si utilizza uno schema di memorizzazione con un livello di *bucket*, ma questa volta si utilizza una suddivisione in blocchi di taglia costante (in [FLMM06] tale dimensione è posta per default a 8Kb). Quindi per ogni blocco si memorizza:

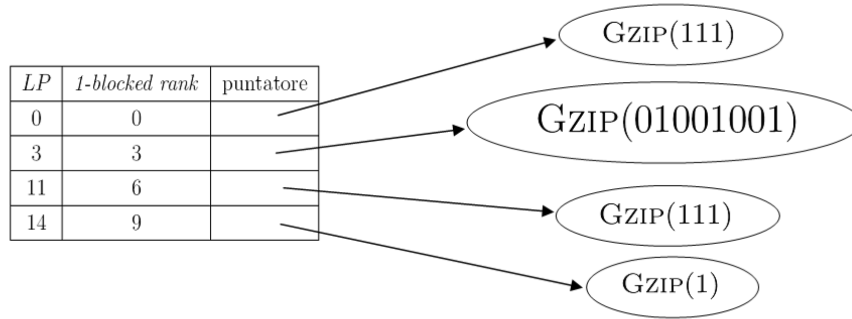


Figura 3: Esempio di memorizzazione di $\widehat{\mathcal{S}}_{\text{LAST}} = 111010010011111$ con $L = 3$

- il numero di etichette LP del prefisso di $\widehat{\mathcal{S}}_\alpha$ memorizzato nei blocchi precedenti;
- una tabella in cui, per ogni possibile etichetta β contenuta in un nodo interno, si memorizza il numero di occorrenze di β nel prefisso di $\widehat{\mathcal{S}}_\alpha$ che precede il blocco. Tale numero è chiamato *β-blocked rank*;
- il puntatore alla rappresentazione compressa del blocco ottenuta utilizzando GZIP.

Da notare è il fatto che questa suddivisione in blocchi di solito porta ad un buon *compression ratio* vista l'omogeneità locale di $\widehat{\mathcal{S}}_\alpha$. Inoltre, visto che il numero di etichette distinte presenti nei nodi interni è solitamente piccolo rispetto alla dimensione del documento, non è necessario utilizzare alcuna codifica sofisticata per memorizzare i vari *β-blocked rank*. In Figura 4 è riportato lo schema di memorizzazione per l'esempio trattato precedentemente (si assume che la dimensione dei blocchi sia la stessa per ogni blocco, tranne l'ultimo).

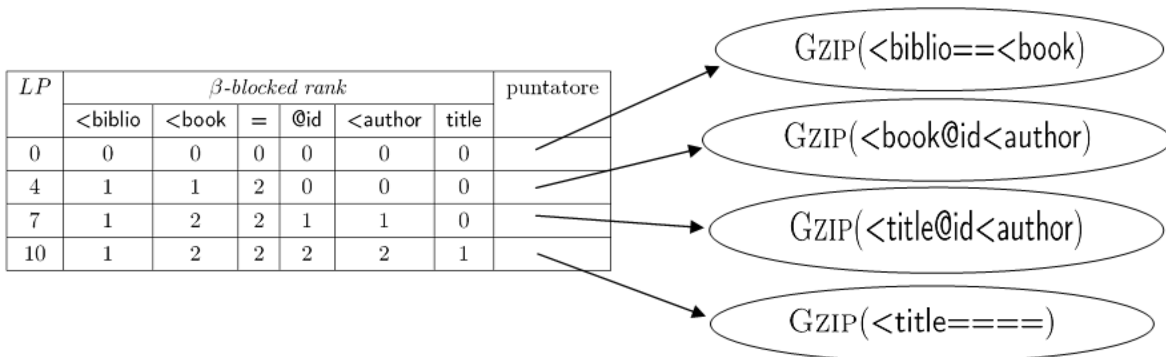


Figura 4: Esempio di memorizzazione di $\widehat{\mathcal{S}}_\alpha$

Grazie alle informazioni memorizzate, per eseguire l'istruzione $\texttt{select}_\beta(\widehat{\mathcal{S}}_\alpha, i)$ è sufficiente eseguire una ricerca binaria sulla tabella che memorizza i *β-blocked rank* per il β desiderato,

seguita dalla decompressione e scansione del blocco individuato tramite tale ricerca. La stessa procedura vale per $\texttt{rank}_\beta(\widehat{\mathcal{S}}_\alpha, i)$ applicando, però, la ricerca binaria ad LP.

5.3.3 $\widehat{\mathcal{S}}_{\mathrm{PC_{DATA}}}$: strutture dati ausiliarie

Questa è solitamente la componente di taglia maggiore di $\mathsf{XBW}(\mathcal{T})$. Come si può vedere dallo pseudocodice, le operazioni di navigazione e $\texttt{SubPathSearch}(\Pi)$ non richiedono alcuna operazione di `rank` o `select` sull'array $\widehat{\mathcal{S}}_{\mathrm{PC_{DATA}}}$, quindi è possibile ottimizzare le strutture dati per la rappresentazione di $\widehat{\mathcal{S}}_{\mathrm{PC_{DATA}}}$ con l'obiettivo di svolgere in maniera efficiente l'operazione di $\texttt{ContentSearch}(\Pi, \gamma)$, in cui Π è un cammino e γ è una qualsiasi stringa di caratteri. Per realizzare questa operazione si utilizza nuovamente uno schema di memorizzazione a *bucket*. L'appartenenza ad un *bucket* è discriminata dalla componente π della foglia. Formalmente, sia $\mathcal{S}_\pi[i, j]$ un intervallo massimale di stringhe uguali in $\mathcal{S}_\pi$; con massimale si intende $\mathcal{S}_\pi[i-1] \neq \mathcal{S}_\pi[i] = \mathcal{S}_\pi[j] \neq \mathcal{S}_\pi[j+1]$. Un *bucket* di $\widehat{\mathcal{S}}_{\mathrm{PC_{DATA}}}$ è costituito dalla concatenazione delle stringhe in $\widehat{\mathcal{S}}_{\mathrm{PC_{DATA}}}[i, j]$. Equivalentemente, due elementi di $\widehat{\mathcal{S}}_{\mathrm{PC_{DATA}}}$ sono nello stesso *bucket* se e solo se hanno lo stesso cammino verso la radice. Anche qui ci si può aspettare che in generale ogni *bucket* sia altamente comprimibile, visto che i dati contenuti in esso sono caratterizzati dallo stesso "contesto".

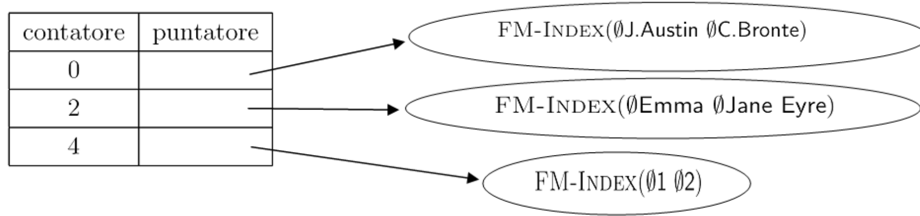


Figura 5: Esempio di memorizzazione di $\widehat{\mathcal{S}}_{\mathrm{PC_{DATA}}}$

Per ognuno dei *bucket* si memorizza:

- Un puntatore al FM-INDEX del *bucket*. Il FM-INDEX è una rappresentazione compressa di una stringa che supporta la ricerca efficiente all'interno dei *bucket*, grazie al fatto che è necessario accedere solamente ad una "piccola" porzione del *bucket* compresso. Tale porzione è piccola nel senso che essa è proporzionale alla lunghezza della stringa cercata, non alla dimensione del *bucket*; per i dettagli riguardanti FM-INDEX si veda [FM01].
- un contatore che indica il numero di stringhe PCDATA che sono contenute nei bucket che rappresentano il prefisso del blocco in $\widehat{\mathcal{S}}_{\mathrm{PC_{DATA}}}$;

In Figura 5 è riportato lo schema di memorizzazione per l'esempio trattato precedentemente. Utilizzando questa struttura dati, per realizzare l'operazione di $\texttt{ContentSearch}(\Pi, \gamma)$ si deve per prima cosa eseguire una $\texttt{SubPathSearch}(\Pi)$ per trovare i nodi il cui cammino

verso la radice hanno prefisso Π^R (linea 7.2). Fatto questo è necessario identificare i nodi che contengono un campo PCDATA: essi sono i nodi il cui cammino verso la radice ha come prefisso $=\Pi^R$ (linee 7.6-7.7). Non è possibile identificare questi nodi direttamente con una `SubPathSearch`(Π=) per l'assunzione fatta sull'input di una `SubPathSearch`(Π) (vedi sezione 4). Sia $\widehat{\mathcal{S}}_{\text{PCDATA}}[\mathbf{F},\mathbf{L}]$ la sottostringa di $\widehat{\mathcal{S}}_{\text{PCDATA}}$ che corrisponde a tali nodi. Vista la rappresentazione di $\widehat{\mathcal{S}}_{\text{PCDATA}}$, tale sottostringa corrisponde ad un numero intero di *bucket*: sia tale numero b. Infine si deve cercare la stringa γ all'interno di ognuno dei b *bucket* utilizzando il FM-INDEX corrispondente (linea 7.11), in tempo proporzionale a $|\gamma|$ per ogni *bucket*.

Algorithm 7: `ContentSearch`(Π, γ)

```
7.1  begin
7.2      (First,Last)← SubPathSearch(Π);
7.3      if ((First,Last)= NIL) then
7.4          return NIL;
7.5      else
7.6          F← rank=(Ŝα, First − 1) + 1;
7.7          L← rank=(Ŝα, Last);
7.8          sia B[i, j] l'intervallo di bucket che contiene Ŝ_PCDATA[F, L];
7.9          N ← {};
7.10         for k ← i to j do
7.11             Inserisci in N le tuple di B[k] contenenti γ;
7.12         end
7.13         return N;
7.14     end
7.15 end
```

L'Algoritmo 7 riporta lo pseudocodice per le operazioni appena descritte. Poiché l'operazione `SubPathSearch`(Π) si esegue in tempo proporzionale a $|\Pi|$, il costo complessivo di una `ContentSearch`(Π, γ) è proporzionale a $|\Pi| + b \cdot |\gamma|$.

Bibliografia

[DOM] http://www.w3.org/tr/dom-level-2-core/introduction.html.

[FLMM05] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Annual Symposium on Foundations of Computer Science (FOCS)*, pages 184–196, 2005.

[FLMM06] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and searching xml data via two zips. In *International World Wide Web Conference (WWW)*, 2006.

[FM01] Paolo Ferragina and Giovanni Manzini. An experimental study of a compressed index. *Invited paper in Information Sciences: Special Issue on Dictionary Based Compression*, 135(1-2):13–28, 2001.

[FM05] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[Shk02] Dmitry Shkarin. Ppm: One step to practicality. In *IEEE Data Compression Conference (DCC 2002)*, pages 202–211, 2002.

[XML] http://www.w3.org/tr/rec-xml/.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.