Integrità delle Query

Parte II

Indice

Capitolo 1

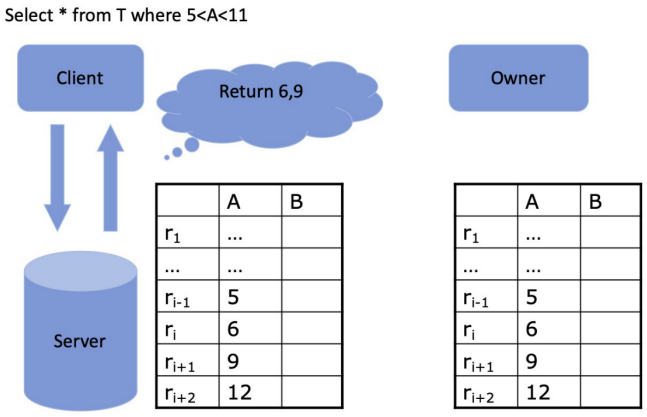
Integrità della computazione

Nel nostro scenario di riferimento potremo avere più *data owner*; queste sono entità di cui ci fidiamo.
Il problema è che questi dati potrebbero essere affidati a dei *cloud provider* esterni, e che possano essere soggetti a delle *computazioni*; questo potrebbe essere un problema sia in termini di confidenzialità che in termini di **integrità**: *"chi mi dice che la tua computazione sia integra?"*.

1.1 Esempi

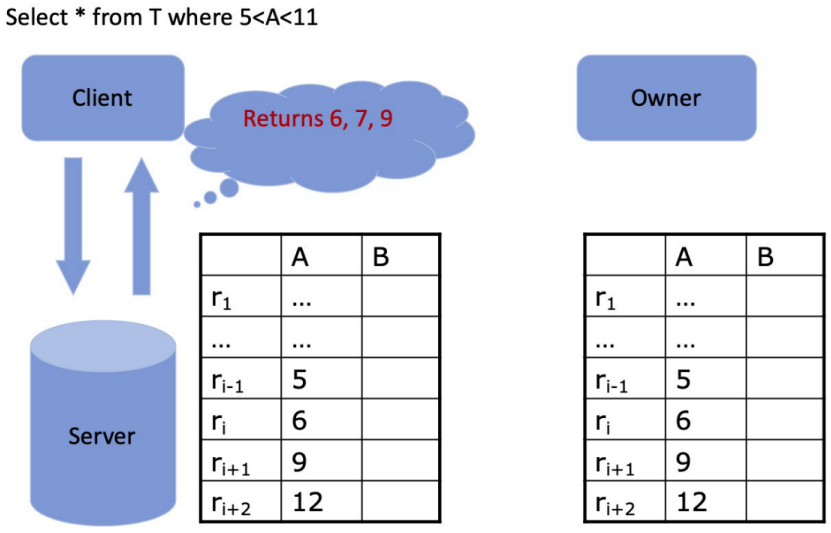
Esempio di una query

Abbiamo l'owner che affida i propri dati ad un provider esterno; abbiamo poi un client che effettua una query.

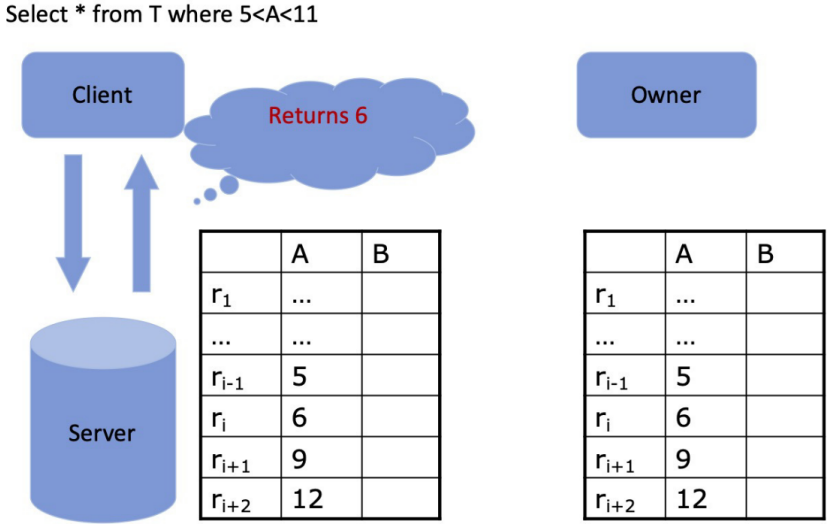


Esempio di query: iniezione

Viene iniettata un'informazione fasulla; *magari mi conviene dirti una cosa piuttosto che un'altra, le tue azioni dipendono da quello che ti dico...*

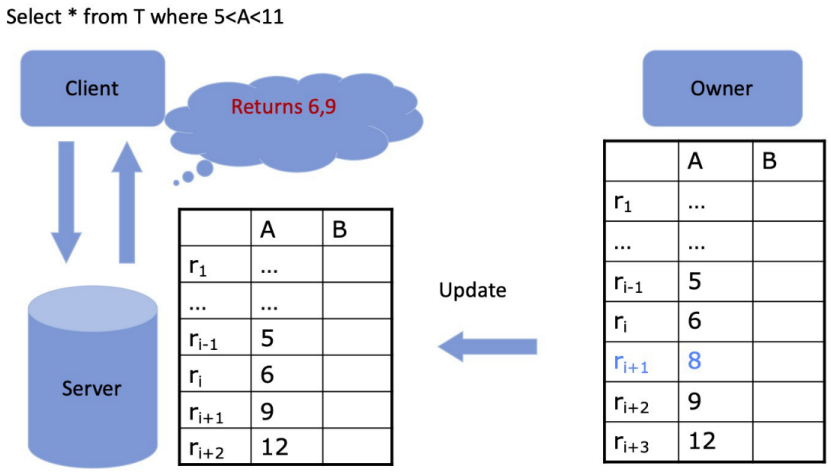


Esempio di query: drop



Esempio di query: omissione

I dati potrebbero essere dinamici, dunque potrebbero essere richieste delle operazioni di update.



1.2 Integrità di storage e computazione

Il data owner e gli utenti necessitano di meccanismi che assicurino l'integrità dei risultati delle query. Una query è integra se rispetta:

- **Correttezza:** il risultato viene calcolato sui dati veri dell'owner (primo esempio)
- **Completezza:** il risultato calcolati su tutti i dati (secondo esempio)
- **Freschezza:** il risultato è calcolato sull'ultima versione dei dati che l'owner ha dato (terzo esempio)

Ci sono due diversi tipi di approcci per rispondere al problema di integrità, ciascuno con i suoi vantaggi e svantaggi:

- **Deterministico:** *se il risultato di una computazione è integro, sono sicuro al 100% che sia integro*

  Queste tecniche vengono implementate in modo che l'owner dà al provider, oltre ai dati da gestire, anche delle strutture ausiliarie che vengono sfruttate per verificare l'integrità della computazione

- **Probabilistico:** *ti dico sempre se è integro o no, ma non con certezza assoluta ma con una certa probabilità; c'è della probabilità di fare degli errori*

  Perché si usano questi approcci? Sono tecniche che hanno lo svantaggio di non avere la certezza assoluta ma che hanno altri vantaggi (che vedremo

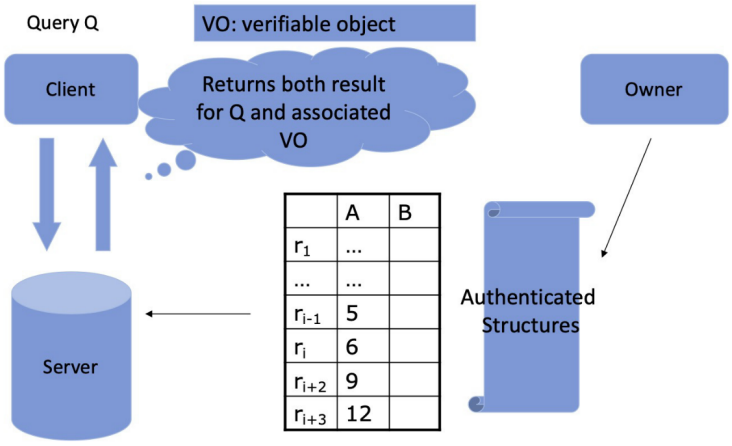
più avanti); il fatto che non avere certezza sia un problema dipende da caso a caso.

In queste tecniche il *qualcosa di auisiliario* sono dei "dati finti" (marcatori) che "aggiungo" ai dati veri; dalla presenza o meno capisco se la query è integra o no (se prima c'erano e poi non ci sono più, probabilmente c'è un errore)

Capitolo 2

Approcci deterministici

L'idea è che il proprietario *dà fuori* i dati e una struttura da lui calcolata. Quando il client vuole fare una computazione, restituisce oltre al risultato anche un *qualcosa in più* usando la struttura dati; questo prende il nome di **verification object**: è ciò che permette di verificare se il risultato della query è integro.



2.1 Approccio basato su firma

Questa tecnica si preoccupa di verificare l'integrità solo per una tipologia particolare di query, ovvero quelle che coinvolgono un solo attributo della relazione; ad esempio $x = 5, 4 < x < 5, \ldots$ l'idea è:

- ordinare le tuple rispetto al valore dell'attribuo preso in considerazione
- applicare una firma alle tuple, non singolarmente ma in coppie tra loro consecutive

  $(t_1, s_1), (t_2, s_2) \ldots (t_n, s_n), cons_i = \epsilon(t_i | t_{i+1})$

- oltre ai dati, vengono date al provider anche le firme

A questo punto quando un client vuole eseguire una computazione, ad esempio $a < x < b$:

- vengono restituite le tuple (e le firme associate) $[a-1, b+1] \rightarrow$ voglio anche la tupla immediatamente precedente ed immediatamente successiva
- le *cose aggiunte* al risultato vero e proprio per verificare l'integrità sono:
  - tuple precedente e successiva
  - firme associate alle tuple

$\Rightarrow$ l'idea è che il client tramite le firme può verificare se il risultato è integro. Questo metodo non è molto utilizzato perché:

- limitazione sulle query
- costosa sia in termini di computazione delle firme, sia nei termini di informazioni aggiuntive che ti devo dare (lineare rispetto al risultato)

Query result: t_3, t_4, t_5, t_6
VO: $t_2, t_7, s_3, s_4, s_5, s_6$

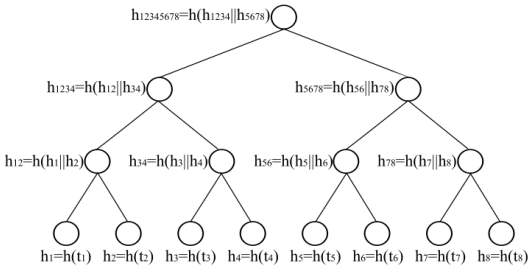
Figura 2.1: VO sta per verification object

2.2 Merkle hash tree

Questa tecnica può essere utilizzata per risolvere lo stesso tipo di query viste nella sezione precedente, ma in maniera più efficiente.

Patients

	SSN	Name	Disease
t_1	123-45-6789	Alice	Asthma
t_2	234-56-7891	Bob	Asthma
t_3	345-67-8912	Carol	Asthma
t_4	456-78-9123	David	Bronchitis
t_5	567-89-1234	Eva	Bronchitis
t_6	678-91-2345	Frank	Gastritis
t_7	789-12-3456	Gary	Gastritis
t_8	891-23-4567	Hilary	Diabetes



Merkle hash tree over attribute SSN

L'idea è:

- ordinare i valori dell'attributo preso in considerazione
- si applica una funzione di hash alle tuple (foglie dell'albero)
  - nel livello delle foglie ci sono 2^L elementi
  - i nodi intermedi vengono calcolati applicando la stessa funzione di hash alla concatenazione degli hash dei figli
    → l'idea è che l'hash di un nodo dipende dall'hash dei figli
  - se l'albero non è completo, tipicamente si aggiungono delle tuple *null* per renderlo completo

⇒ la quantità di informazioni aggiuntive non è più lineare rispetto al risultato ma è **logaritmica**.

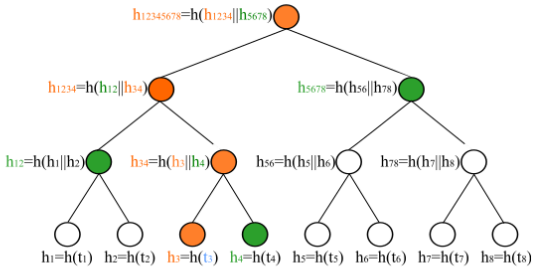
2.2.1 Merkle hash tree verification

- L'idea è che il cloud provider fornisce un ***verification object*** per permettere al client di **ricostruire l'hash associato alla radice**
- Il risultato della query è **corretto e completo** la radice calcolata corrisponde a quella già conosciuta
  - se c'è una tupla mancante o non corretta, la radice calcolata sarà diversa da quella già conosciuta

SELECT *
FROM Patients
WHERE SSN = '345-67-8912'

Patients

	SSN	Name	Disease
t_1	123-45-6789	Alice	Asthma
t_2	234-56-7891	Bob	Asthma
t_3	345-67-8912	Carol	Asthma
t_4	456-78-9123	David	Bronchitis
t_5	567-89-1234	Eva	Bronchitis
t_6	678-91-2345	Frank	Gastritis
t_7	789-12-3456	Gary	Gastritis
t_8	891-23-4567	Hilary	Diabetes



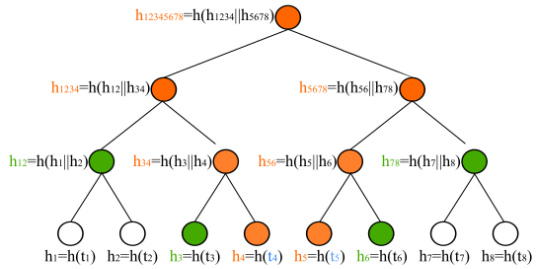
Result: t_3

Verification Object: h_4, h_{12}, h_{5678}
$h_3 = h(t_3)$
$h_{34} = h(h_3||h_4)$
$h_{1234} = h(h_{12}||h_{34})$
$h_{12345678} = h(h_{1234}||h_{5678})$

SELECT *
FROM Patients
WHERE SSN $\geq$ 456* and SSN $\leq$ 567*

Patients

	SSN	Name	Disease
t_1	123-45-6789	Alice	Asthma
t_2	234-56-7891	Bob	Asthma
t_3	345-67-8912	Carol	Asthma
t_4	456-78-9123	David	Bronchitis
t_5	567-89-1234	Eva	Bronchitis
t_6	678-91-2345	Frank	Gastritis
t_7	789-12-3456	Gary	Gastritis
t_8	891-23-4567	Hilary	Diabetes



Result: t_4, t_5

Verification Object: h_3, h_6, h_{12}, h_{78}
$h_4 = h(t_4)$, $h_5 = h(t_5)$
$h_{34} = h(h_3||h_4)$, $h_{56} = h(h_5||h_6)$
$h_{1234} = h(h_{12}||h_{34})$, $h_{5678} = h(h_{56}||h_{78})$
$h_{12345678} = h(h_{1234}||h_{5678})$

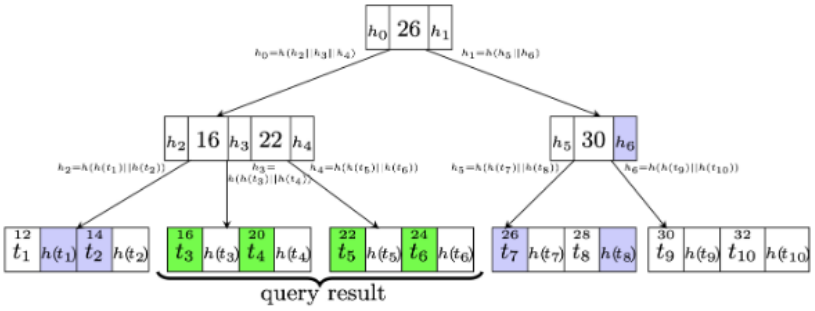
Con la tecnica delle firme, il costo della verifica è lineare rispetto alla dimensione del risultato; mentre con la tecnica dell'albero il costo è sempre pari a $\log n$ (n *numero di foglie*)

$\rightarrow$ quando la dimensione del risultato supera $\log n$, sarà più efficiente usare l'albero

2.3 Merkle B-tree

Ciascun nodo contiene più informazioni:

- Nodo foglia:
  - chiave k_i
  - puntatore all'area di memoria che contiene l'informazione vera e propria
  - funzione di hash applicata alla tupla con chiave k_i
- Nodo interni:
  - chiave
  - puntatori ai nodi figli
  - funzione di hash applicata alla concatenazione di tutti gli hash che appaiono nel nodo puntato dal puntatore



Query result: t_3, t_4, t_5, t_6
VO: $t_2, t_7, h(t_1), h(t_8), h_6$

Per semplicità la tupla nei nodi foglia è memorizzata direttamente all'interno del nodo.
Concettualmente il meccanismo di funzionamento e verifica è lo stesso dell'albero visto precedentemente; possiamo vedere questa versione come una sua generalizzazione in cui ogni nodo può contenere più chiavi.

- l'hash associato al nodo radice è un *summary* di tutte le informazioni che contiene l'albero

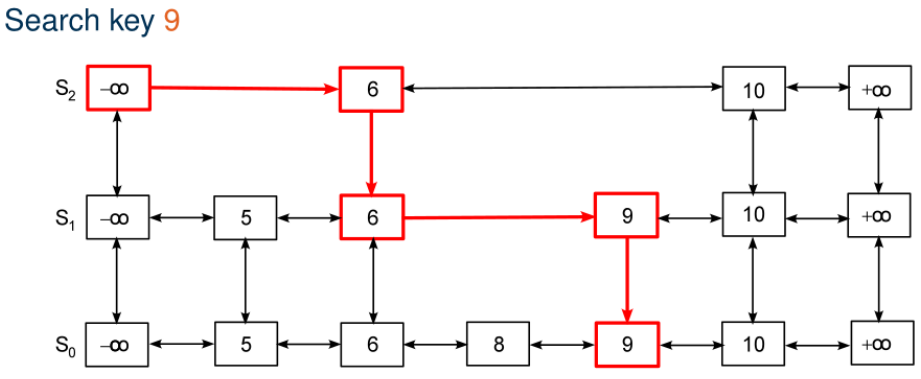
2.4 Skip list

Ha lo svantaggio che può essere usata solo con query di uguaglianza, ma ha il vantaggio di poter essere integrata in modo efficiente in un DBMS.
Una **skip list** è una lista

- Una **skip list** per un set di elementi è una serie di liste $S_1, S_2, \ldots, S_n$, tale che:
  - S_0 contiene tutti gli elementi ordinati rispetto a un qualche attributo e le sentinelle $+\infty$ e $-\infty$
  - S_i contiene un sottoinsieme degli elementi della lista S_{i-1} con una probabilità p (le sentinelle sono sempre inlcuse)
- Ha il vantaggio che tutte le operazioni vengono fatte in tempo $O(log(n))$, per cui è molto efficiente
  - `find(x)`
  - `delete(x)`
  - `insert(x)`

2.4.1 Search operation

- Si inizia dall'elemento sentinella nella top list ($-\infty$ nella lista più in alto)
- Vado avanti finché trovo un valore $\leq$ di quello che sto cercando (*hop forward*)
- Nel caso in cui ce ne fosse uno maggiore, allora scendo nella lista sotto (*top down*) e proseguo la ricerca con lo stesso procedimento

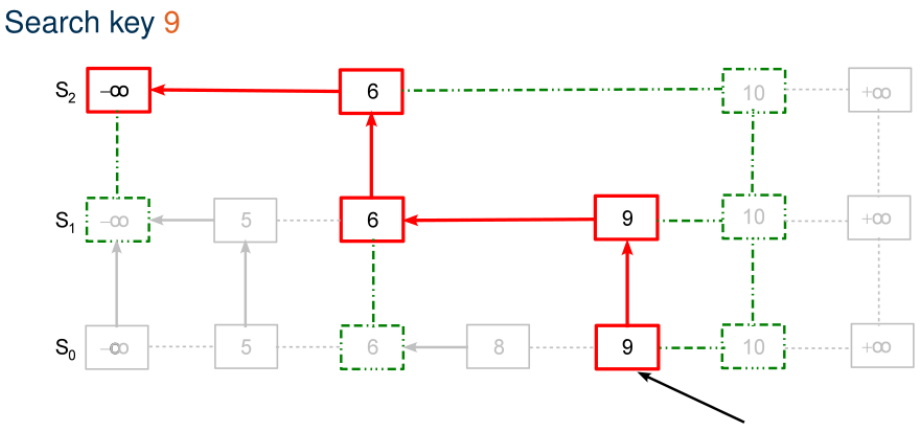


2.4.2 Authenticated skip list

Ad ognuno dei nodi si applica una funzione di hash h:

- resistente a collisioni
- commutativa ($h(x, y) = h(y, x)$)

Ad ogni nodo viene associata una etichetta che, insieme ai *verification object*, permettono di ricostruire il cammino della ricerca nel senso opposto, allo scopo di verificare la computazione.



Capitolo 3

Approcci probabilistici

Lo svantaggio è che non si ha più la certezza sui risultati, ma hanno il vantaggio di coprire più tipologie di query.

3.1 Introduzione

Possono essere usate due tecniche principali, che possono essere combinate tra loro per ottenere una maggiore efficacia.

3.1.1 Fake tuples

L'idea è mettere dentro ai dati originali dei dati fasulli e li mischio; dopo posso controllare la loro presenza nel risultato della query (al client restituisco anche le tuple fasulle, lui controllerà che ci sono tutte le tuple fasulle che si aspetta). Questo meccanismo ha una serie di problematiche che devono essere affrontate:

- le tuple *fake* **non devono essere riconoscibili** da quelle reali
- si utilizzano **dati criptati** per proteggerli (questo ci aiuta nel punto precedente)
- si associa a ciascuna tupla una *informazione aggiuntiva* per verificare l'**autenticità dei dati**; posso pensare di applicare una funzione di hash alla concatenazione dei valori degli attributi della tupla; questa informazione aggiuntiva la uso anche per distinguere le tuple fasulle da quelle originali

Approccio random

Quando il client ottiene il risultato di una query, poi deve filtrare le tuple fasulle facendo una ulteriore query in locale

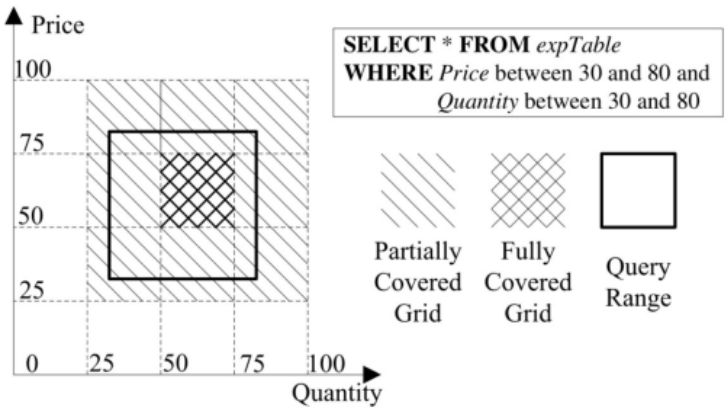
⇒ **il client deve tenersi una copia delle tuple fasulle e computare una query**

. . . bisogna pensare ad un altro approccio più efficiente

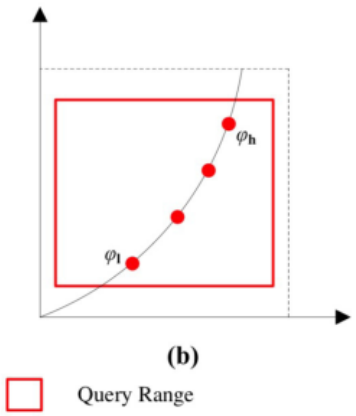
Approccio determinstico

Viene tenuta localmente la funzione usata per generare le tuple fasulle (al posto delle tuple fasulle); per semplificare il passo di verifica, l'idea è:

- si partizionano i domini
- per ogni partizione mi segno quante tuple fasulle gli appartengono
- quando eseguo una query, saprò se una certa partizione del dominio sarà *coperta* parzialmente o completamente dal risultato della query
- per verificare farò il conteggio delle tuple fasulle



Per calcolare il numero di tuple in una determinata sezione, io conosco la funzione usata per generare le tuple; per cui mi basta calcolare i punti di intersezione per fare il conteggio (è importante che la funzione sia monotona).

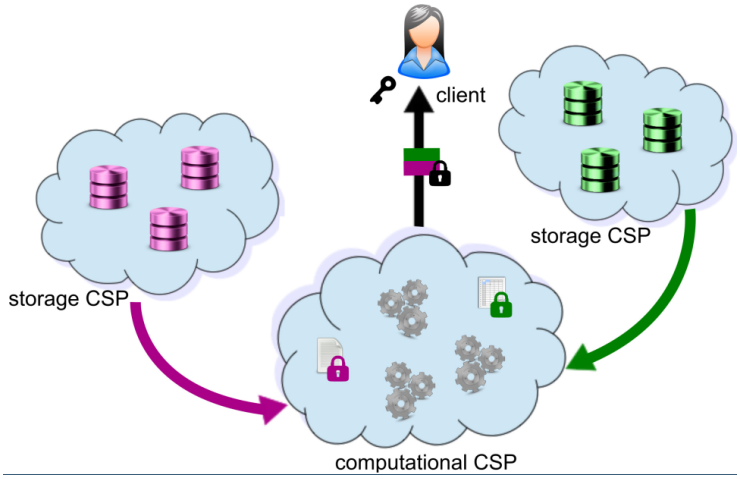


3.1.2 Duplicazione di tuple

L'idea è quella di non usare tuple fake ma di duplicare alcune tuple reali; la verifica viene fatta contando il numero di tuple doppie che mi aspetto di avere.

3.2 Computazione con provider multipli

Lo scenario di riferimento è quello con multiple sorgenti informative; supponiamo che le entità che hanno in mano i dati siano fidate, ma che le computazioni (magari perché sono costose) vengano fatte sfruttando le risorse computazionali di qualcun'altro (costa meno rispetto a gestirle direttamente).
Questa entità che gestisce le computazioni non è fidata; anzi, meno costa probabilmente meno è fidata . . . devo verificare il risultato che viene restituito.
In questo contesto vengono combinate le tecniche viste in precedenza.



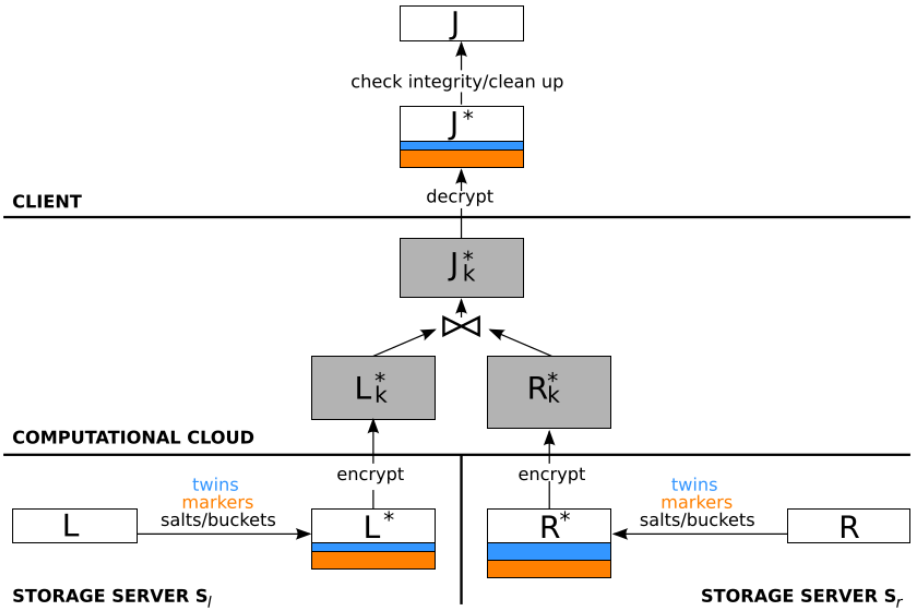
3.3 Approccio probabilistico per query di join

Questo tipo di query è quello tipicamente più costoso; le tecniche di protezione sono:

- **dati criptati**: proteggo i dati criptandoli perché non mi fido dell'entità che esegue la computazione
- **markers** (tuple fake): prima di consegnarli aggiungo tuple fasulle
- **twins** (tuple replicate): prima di consegnarli replico alcune tuple
- **salts/buckets**: voglio evitare che chi prende i miei dati (a cui ho aggiunto le informazioni di controllo) non deve essere in grado di distinguere le informazioni di controllo dai dati veri e propri

L'idea è:

- ho due storage server che sono fidati e che hanno in mano i dati veri e propri
- ciascuno dei due server mette dentro *markers* e *twins*
- criptano i dati e li danno a *qualcun'alro (computational cloud)*, che non è più grado di distinguere i dati veri da quelli spuri
- viene fatta la operazione di join
- chi ha richiesto l'esecuzione della query decripta il risultato, fa tutte le verifiche opportune (markers e twins) e si tiene il risultato



3.3.1 On-the-fly encryption

Devo criptare separatamente ogni attributo di join per proteggerli (gli altri attributi al computational cloud non interessano... anche se li cripto tutti insieme va bene lo stesso).
Il client genera la chiave di criptazione che viene comunicata agli storage server tramite il computational cloud; gli storage server utilizzano questa chiave per criptare separatamente:

- attributo di join
- tutti gli altri attributi

Danno queste relazioni criptate al computational cloud che computerà il join (è importante affinché il computational cloud possa eseguire il join che i server usino la stessa chiave).

R_l

	I	Attr
l_1	a	Ann
l_2	b	Beth
l_3	c	Cloe

R_r

	I	Attr
r_1	a	flu
r_2	a	asthma
r_3	b	ulcer
r_4	e	hernia
r_5	e	flu
r_6	e	cancer

J

	L.I	L.Attr	R.I	R.Attr	
l_1	a	Ann	a	flu	r_1
l_1	a	Ann	a	asthma	r_2
l_2	b	Beth	b	ulcer	r_3

R_{lk}

I_k	L.Tuple$_k$
α	λ_1
β	λ_2
γ	λ_3

R_{rk}

I_k	R.Tuple$_k$
α	ρ_1
α	ρ_2
β	ρ_3
ε	ρ_4
ε	ρ_5
ε	ρ_6

J_k

L.I_k	L.Attr$_k$	R.I_k	R.Attr$_k$
α	λ_1	α	ρ_1
α	λ_1	α	ρ_2
β	λ_2	β	ρ_3

3.3.2 Markers

Gli storage server iniettano delle tuple fasulle; dato poi il computational cloud lavora sul criptato, non mi devo neanche preoccupare di generare delle tuple *simili al vero*.
Tuttavia, per gli attributi di join il valore deve essere generato con attenzione, perché voglio evitare che tuple fasulle si combinino con tuple vere; devo avere la garanzia che quel valore possa essere un valore reale per l'attributo di join.

R_l

	I	Attr
l_1	a	Ann
l_2	b	Beth
l_3	c	Cloe

R_r

	I	Attr
r_1	a	flu
r_2	a	asthma
r_3	b	ulcer
r_4	e	hernia
r_5	e	flu
r_6	e	cancer

J

	L.I	L.Attr	R.I	R.Attr	
l_1	a	Ann	a	flu	r_1
l_1	a	Ann	a	asthma	r_2
l_2	b	Beth	b	ulcer	r_3

$R_l{}^*$

	I	Attr
l_1	a	Ann
l_2	b	Beth
l_3	c	Cloe
m_1	x	$marker_1$

$R_r{}^*$

	I	Attr
r_1	a	flu
r_2	a	asthma
r_3	b	ulcer
r_4	e	hernia
r_5	e	flu
r_6	e	cancer
m_2	x	$marker_2$

J^*

	L.I	L.Attr	R.I	R.Attr	
l_1	a	Ann	a	flu	r_1
l_1	a	Ann	a	asthma	r_2
l_2	b	Beth	b	ulcer	r_3
m_1	x	$marker_1$	x	$marker_2$	m_2

Ad esempio, per avere certezza che non ci sia mai questa collisione, potrei:

- combinare i valori veri con un certo v
- combinare i valori fasulli con un valore distinto v'

Devo sostanzialmente inventarmi un qualunque meccanismo che mi assicuri che non ci sia collisione.

3.3.3 Twins

Gli storage server (senza coordinarsi, neanche si conoscono ...) duplicano alcune tuple; il problema è: *quali tuple vado a duplicare?*
Gli storage server devono duplicare le tuple in modo tale che queste tuple si combinino correttamente durante la operazione di join; mi devo trovare queste tuple duplicate nel risultato del join, altrimenti non avrebbe senso.

⇒ **devono duplicare le tuple con lo stesso attributo di join**

L'unico modo per avere questa garanzia senza che i server comunichino tra loro, il client comunica quali tuple devono essere duplicate; ad esempio, tutte le tuple che soddisfano una certa condizione che dipende dall'attributo di join).

R_l

	I	Attr
l_1	a	Ann
l_2	b	Beth
l_3	c	Cloe

R_r

	I	Attr
r_1	a	flu
r_2	a	asthma
r_3	b	ulcer
r_4	e	hernia
r_5	e	flu
r_6	e	cancer

J

	L.I	L.Attr	R.I	R.Attr	
l_1	a	Ann	a	flu	r_1
l_1	a	Ann	a	asthma	r_2
l_2	b	Beth	b	ulcer	r_3

R_l

	I	Attr
l_1	a	Ann
l_2	b	Beth
l_3	c	Cloe

R_r

	I	Attr
r_1	a	flu
r_2	a	asthma
r_3	b	ulcer
r_4	e	hernia
r_5	e	flu
r_6	e	cancer

J

	L.I	L.Attr	R.I	R.Attr	
l_1	a	Ann	a	flu	r_1
l_1	a	Ann	a	asthma	r_2
l_2	b	Beth	b	ulcer	r_3

3.3.4 Salts and buckets

Cercano di eliminare le **frequenze riconoscibili** delle combinazioni in join *one-to-many*.
Il problema sta nella tabella con molte tuple (*many*):

- se una tupla ha 1 sola occorrenza, potrei inferire che si tratta di un marker, dato che vengono generati con valori (per l'attributo di join) sempre diversi per evitare collisioni (non ho la certezza, ma sospetta che possa essere un marker)
- se ho tuple con lo stesso valore ripetuto per l'attributo di join, allora inferisco che sicuramente non sono marker; non possono nemmeno essere twins perchè vanno sempre a coppie (multipli di 2); quindi capisco che sono tuple vere

Questi attacchi sfruttano la frequenza con cui appaiono i valori degli attributi di join; per evitarli si cerca di appiattire queste frequenze, ovvero tutti i valori possibili appaiono tutti con la medesima frequenza; viene fatto in due modi:

- **Sali:** si combinano i valori dell'attributo di join con un valore casuale
  - ha lo svantaggio che la tabella *one* aumenta di dimensione per appiattire la frequenza delle occorrenze
  - ha il vantaggio che la dimensione del risultato del join rimane invariata
- **Bucket:** creo dei gruppi di tuple, dove tutte le tuple appartenenti ad un bucket hanno lo stesso valore per l'attributo di join
  - devo aggiungere delle tuple *dummy* nella tabella *many* per ottenere la dei gruppi con dimensione uniforme
  - i join non saranno $(1:1)$ ma saranno tutti con la stessa frequenza, quindi non posso inferire nulla
  - lo svantaggio è che aumenta la dimensione sia della tabella *many* che del risultato del join

R_l^*

	I	Attr
l_1	a	Ann
l_2	b	Beth
l_3	c	Cloe

R_r^*

	I	Attr
r_1	a	flu
r_2	a	asthma
r_3	b	ulcer
r_4	e	hernia
r_5	e	flu
r_6	e	cancer

J^*

	L.I	L.Attr	R.I	R.Attr	
l_1	a	Ann	a	flu	r_1
l_1	a	Ann	a	asthma	r_2
l_2	b	Beth	b	ulcer	r_3

R_l^*

	I	Attr
l_1	a	Ann
l_1'	a′	Ann′
l_2	b	Beth
l_2'	b′	Beth′
l_3	c	Cloe
l_3'	c′	Cloe′

R_r^*

	I	Attr
r_1	a	flu
r_2	a	asthma
r_3	b	ulcer
d_1	b	$dummy_1$
r_4	e	hernia
r_5	e	flu
r_6	e′	cancer
d_2	e′	$dummy_2$

J^*

	L.I	L.Attr	R.I	R.Attr	
l_1	a	Ann	a	flu	r_1
l_1	a	Ann	a	asthma	r_2
l_2	b	Beth	b	ulcer	r_3
l_2	b	Beth	b	$dummy_1$	d_1

Le tecniche di *salt* e *bucket* possono essere usate sia singolarmente che in combinazione.

3.3.5 Valutazione delle query

Il client:

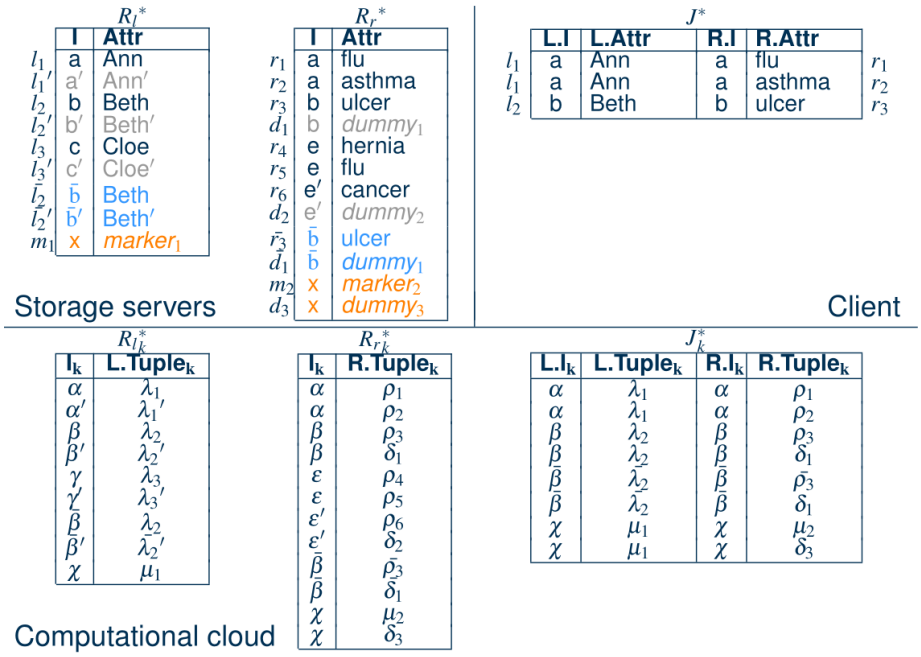
- parla con colui che esegue la query (computational cloud), non con i server; gli manda la richiesta di join e quali sono i server da coinvolgere
- tra le varie informazioni che il client manda al cloud, ci sono:

  1. Sotto-query che ciascun server deve eseguire sulla sua tabella

     ```
     SELECT * FROM L, R
     WHERE L.Name = R.Name and L.Stipendio > 1000
     ```

  2. Chiave di criptazione della query
  3. Numero di marker da iniettare
  4. Percentuale di twins
  5. Numero di sali da utilizzare

  Tutte queste informazioni protette vengono criptate con la chiave pubblica degli storage server per passarle al cloud (*honest-but-curious*).



3.3.6 Markers e twins: garanzia dell'integrità

I markers e twins offrono una protezione complementare; senza la loro combinazione potrebbero rimanere scoperti alcuni casi estremi:

- i **twins sono efficaci il doppio**, ma perdono la loro efficacia quando il computational cloud omette una grande parte delle tuple (caso estremo: omette tutte le tuple)
  - *se butto via poco, mi proteggono bene i twins perché dovrei avere la fortuna di buttare via entrambi i twins*
- i **markers permettono di rilevare casi estremi**

3.4 Semi-join

Protegge il join:

- senza introdurre salts e buckets
- supporta *one-to-one, one-to-many, many-to-many*

Dato che il problema è la frequenza delle occorrenze dei valori di join, si proietta tutto su di essi:

1. `SELECT Name WHERE I=A`

2. `SELECT DoB WHERE I=A`

3. si fa il *merge* dei risultati ottenuti

Proiettando tutto sull'attributo di join, avrò solo una occorrenza dei valori di join; in questo modo sto automaticamente appiattendo le frequenze.

- Ha il vantaggio di non aumentare la dimensione delle tabelle
- Ha lo svantaggio che il client deve si prende in carica contattare i server e di calcolare i risultati; non gli viene dato già pronto

3.5 Computational cloud distribuito

Il computational cloud non è un'unica entità, ma in questo scenario è distribuito; ci sono più nodi, ciascuno prende una parte della relazione per computare la sua parte, ed infine il risultato viene messo insieme.
il problema è verificare se il risultato ottenuto combinando i risultati parziali va bene o no

⇒ **ho il problema di come distribuire le informazioni di controllo**

3.5.1 MapReduce

È un framework distribuito con più nodi (*workers*) basato su due funzioni:

- **Map:** definita lato client, ha l'obiettivo di trasformare i dati su cui vogliamo eseguire una computazioni in un insieme di coppie
  - il primo campo è una chiave
  - il secondo campo è il valore
- **Assignement:** una funzione di assegnamento che assegna delle coppie ai workers; le coppie con la stessa chiave appartengono allo stesso worker
- **Reduce:** funzione eseguita da ciascun worker che produce il risultato parziale

Ogni worker esegue la sua funzione di reduce; successivamente, il *manager* combina i vari risultati parziali per ottenere il risultato finale.

3.5.2 On-the-fly encryption

Dato che non ci fidiamo del computational cloud, vengono create delle coppie criptate, dove:

- la chiave è il valore dell'attributo criptato
- il valore è il nome della relazione a cui appartiene

3.5.3 Markers and MapReduce