Integrità delle Query

Parte II

Indice

Capitolo 1

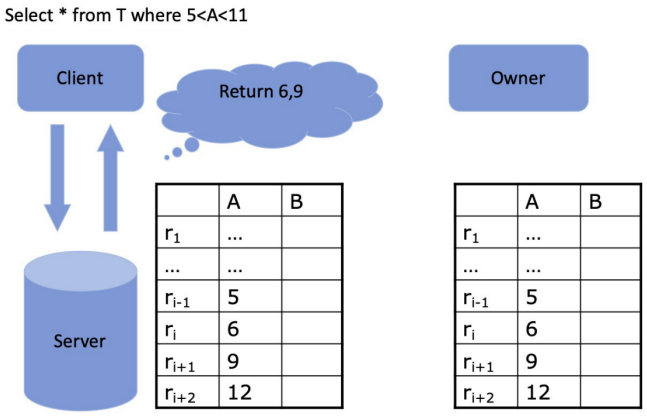
Integrità della computazione

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

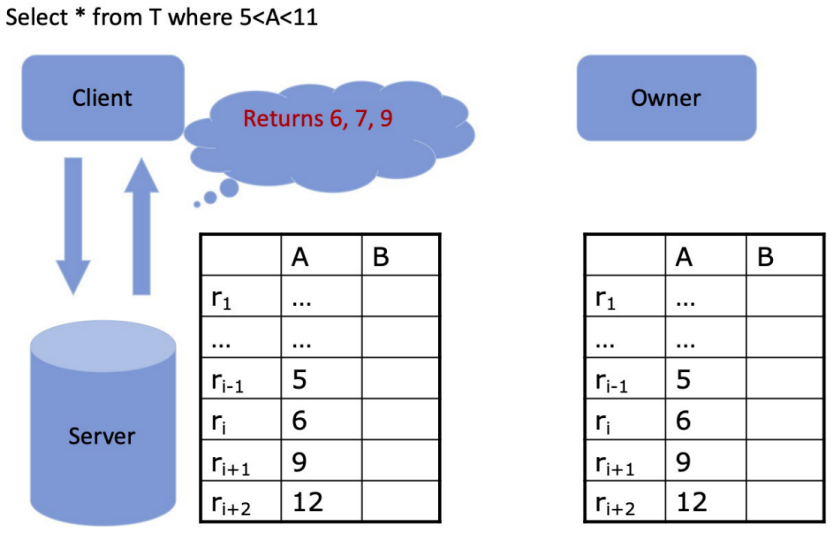
Esempio di una query

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

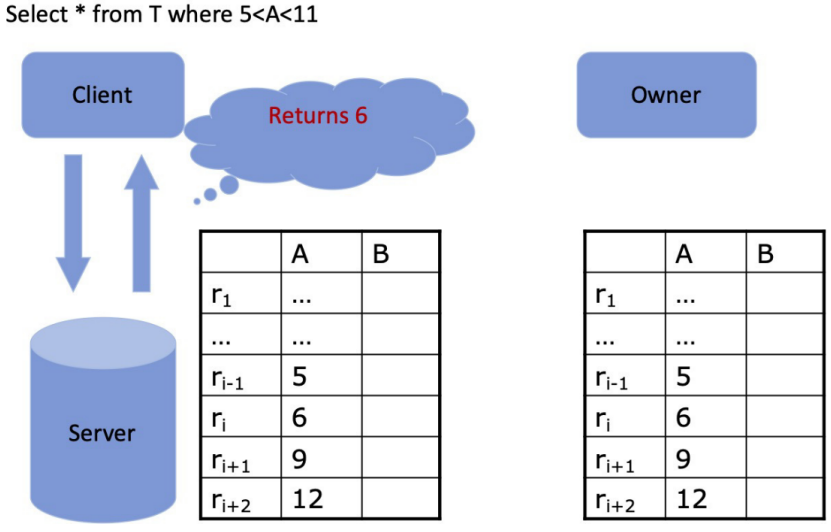


Esempio di query: iniezione

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

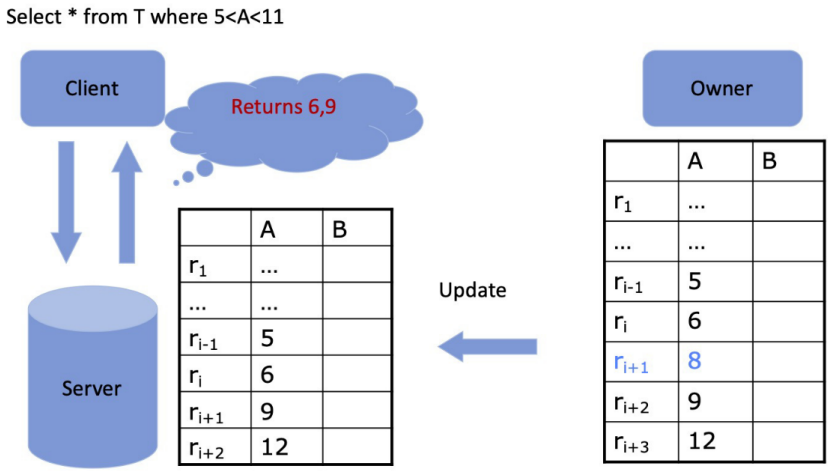


Esempio di query: drop



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

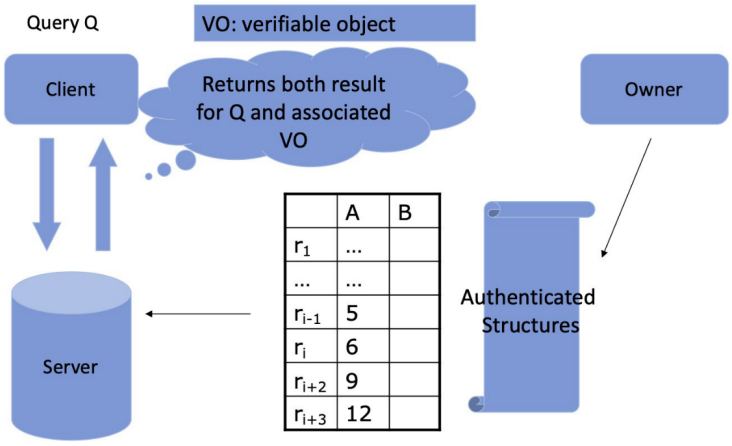
più avanti); il fatto che non avere certezza sia un problema dipende da caso a caso.

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: $t_2, t_7, s_3, s_4, s_5, s_6$

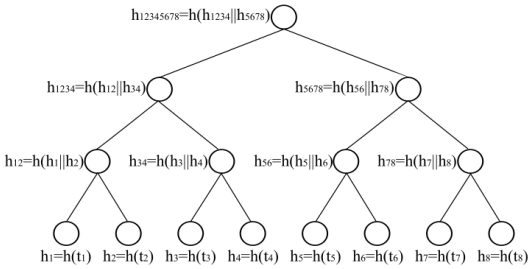
Figura 2.1: VO sta per verification object

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**.

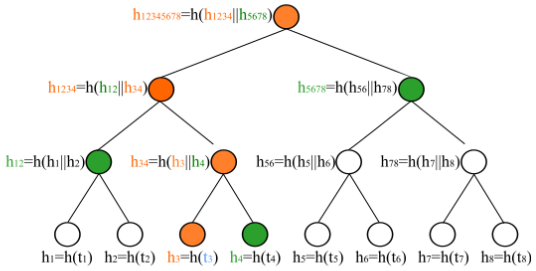
2.2.1 Merkle hash tree verification

- 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



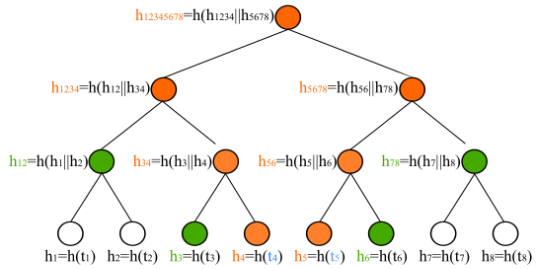
Result: t_3

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

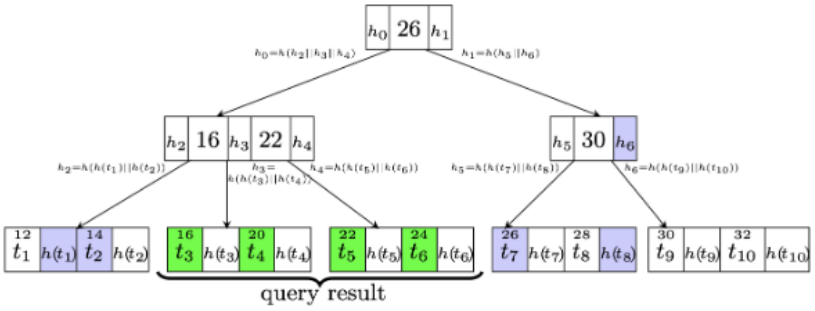
Con la tecnica delle firme, il costo della verifica è lineare rispetto alla dimensione del risultato; mentre con la tecnica dell'albero il costo è sempre pari a $\log n$ (n *numero di foglie*)

$\rightarrow$ 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

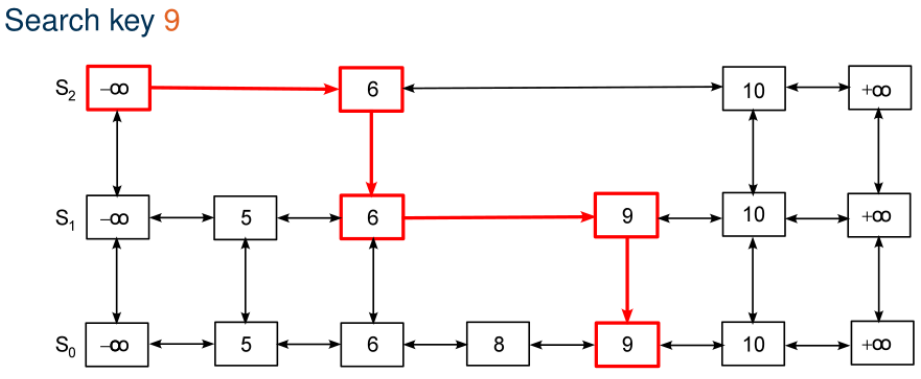
2.4 Skip list

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

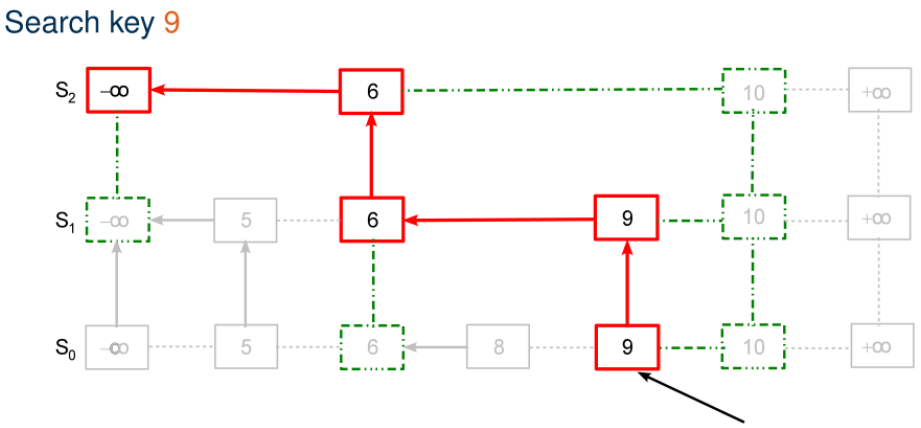


2.4.2 Authenticated skip list

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

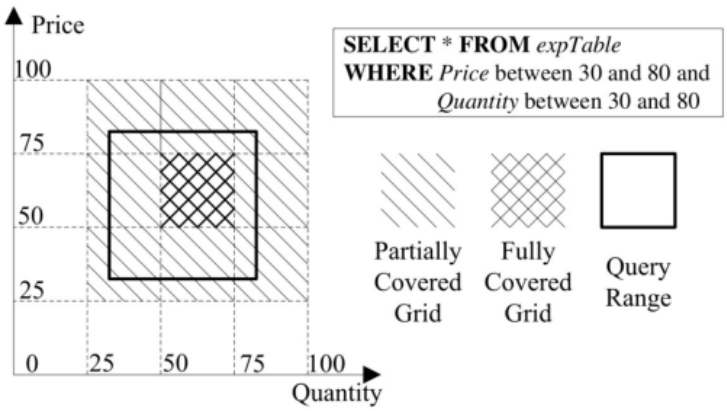
⇒ **il client deve tenersi una copia delle tuple fasulle e computare una query**

. . . bisogna pensare ad un altro approccio più efficiente

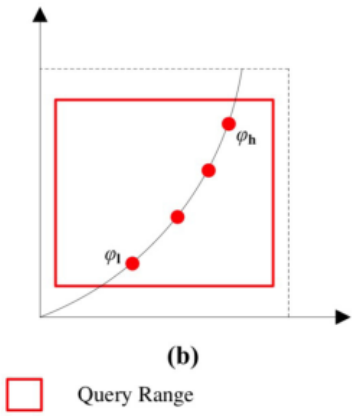
Approccio determinstico

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).

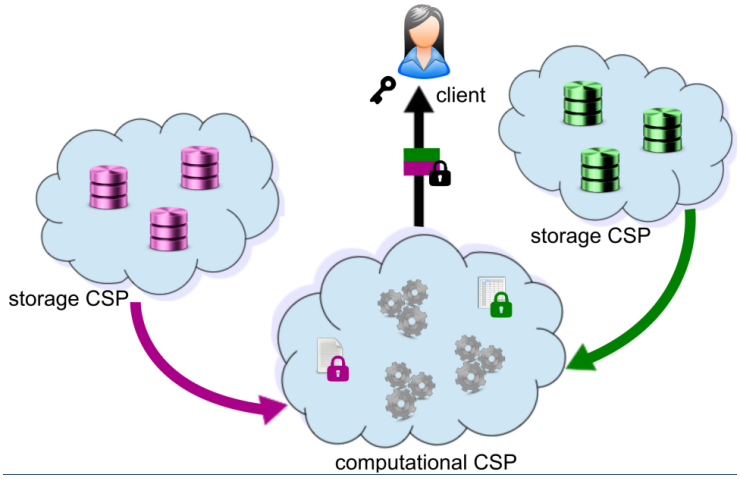


3.1.2 Duplicazione di tuple

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

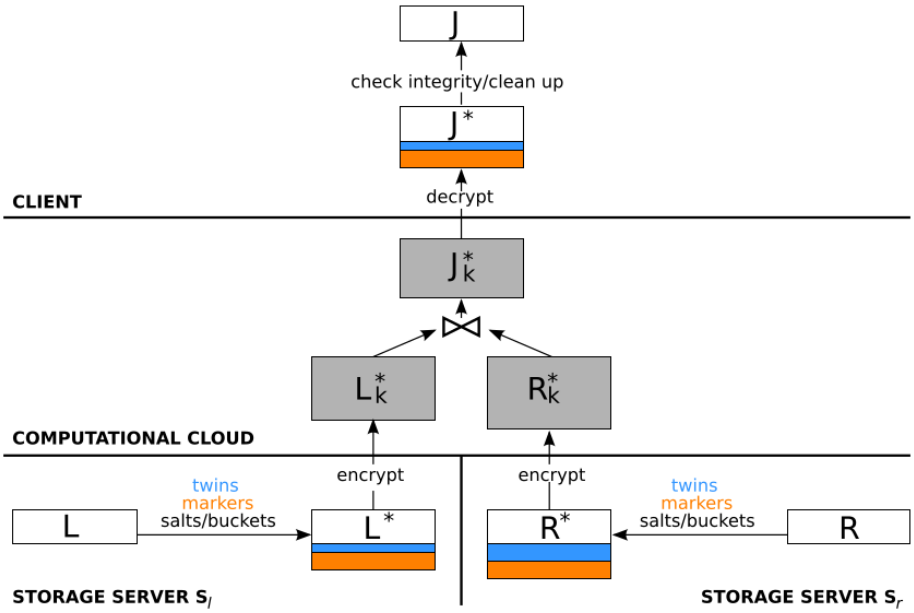
Questo tipo di query è quello tipicamente più costoso; le tecniche di protezione sono:

- **dati criptati**
- **markers** (tuple fake)
- **twins** (tuple duplicate)
- **salts/buckets** per ottenere occorrenze *flattened*

L'idea è:

- ho due storage server che sono fidati e che hanno in mano i dati veri e propri

- ciascune dei due server mettono 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

La relazione criptata contiene due campi: mmmmh