Appunti Basi di Dati

Luca Seggiani

22 Maggio 2024

Refresh incrementale

Vediamo adesso tutte quelle metodistiche che permettono di effettuare un refresh "incrementale", che va opportunamente ad effettuare una sincronizzazione nel, attraverso le modalità già viste, su una tabella che decidiamo di mantenere aggiornata fino ad un certo istante nel tempo $t^\star$. L'incremental refresh si basa sul non accedere mai (o almeno, sul farlo in modo molto limitato) alle tabelle contenente i dati raw, ma di usare invece il meccanismo delle **log table**. Le log table, *"tabelle dei log"*, sono apposite strutture create per tenerre traccia delle modifiche effettuate su una o più tabelle. Una log table conterrà quindi, ad esempio, tutte le modifiche fatte ad una certa relazione, la data in cui tali modifiche sono state eventuate, ecc... Una log table è libera di contenere valori contenenti nelle nuove entrate aggiunte alle tabelle di interesse, come valori derivati ricavati da tali entrate. Sulla base della log table, si potrà poi eseguire:

- **Partial refresh**: un trasferimento parziale del log, cioè solamente delle modifiche che effettivamente comportano una variazione dei contenuti della materialized view;
- **Complete refresh**: un trasferimento totale e imparziale delle modifiche riportate sulla log table;
- **Rebuild**: una ricostruzione da zero della materialized view, sulla base delle informazioni contenute nella log table.

Assicurandosi che la ricostruzione della materialized view sulla base della log table non implichi accessi alle tabelle d'interesse, si può creare un sistema molto performante, in quanto i record della log table sono molto meno di quelli delle tabelle grezze. Questo significa tenere traccia di tutte le modifiche effettuate al database, e che queste siano sufficienti ad eseguire il refresh della vista materializzata. Nel caso la vista si basi su più tabelle, possono tornare utili più log table.

Partial refresh

Vediamo nel dettaglio il partial refresh. Il partial refresh si basa sul fatto che non tutti i record che vengono inseriti nella tabella d'interessa dopo il tempo $t^\star$ di aggiornamento della materialized view sono effettivamente importanti per il calcolo della materialized view. Possiamo quindi restringere le operazioni di push sulla materialized view alle sole modifiche effettivamente interessanti la materialized view. Vediamo un'esempio. Si potrà dichiarare una log table come qualsiasi altra tabella:

```
CREATE TABLE LOG_TABLE (
  Paziente CHAR(100) NOT NULL,
  DataVisita DATE
) ENGINE = InnoDB DEFAULT CHARSET=latin1;
```

Possiamo quindi definire trigger di aggiornamento della stessa, sia in caso di inserzione su paziente che su visita:

```
DELIMITER $$
DROP TRIGGER IF EXISTS push_visita $$
CREATE TRIGGER push_visita
AFTER INSERT ON Visita
FOR EACH ROW
BEGIN
  INSERT INTO LOG_TABLE
  VALUES(NEW.Paziente, CURRENT_DATE);
END $$

DROP TRIGGER IF EXISTS push_paziente $$
CREATE TRIGGER push_paziente
AFTER INSERT ON Paziente
FOR EACH ROW
BEGIN
  INSERT INTO LOG_TABLE
  VALUES(NEW.CodFiscale, NULL);
END $$
```

A questo punto saranno stabilite le procedure che manterranno in ordine la log table, e si potrà quindi procedere con l'implementazione di procedure che aggiornino la materialized view sulla base delle informazioni contenute nella stessa, magari fino a una certa data dopo $t^\star$ (si parla di refresh parziale). Notiamo adesso che è importare scegliere se concentrarsi sull'ottimizzazione delle procedure di push dalla tabella raw alla log table, o sulle procedure di refresh della materialized view. In generale, conviene ottimizzare le operazioni di push (in quanto più frequenti), e lasciare che siano le operazioni di refresh a prendere la maggior parte del tempo e delle risorse, in quanto quest'ultime verranno comunque eseguite in momenti di basso carico sul database (ad esempio di notte). Vediamo quindi la procedura d'aggiornamento parziale:

```
DROP PROCEDURE IF EXISTS on_demand_partial $$
CREATE PROCEDURE on_demand_partial(_up_to DATE)
BEGIN
  REPLACE INTO MATERIALIZED_VIEW
  WITH aggregated_log AS (
    SELECT LT.Paziente, SUM(IF(LT.DataVisita IS NULL, 0, 1)) AS NuoveVisite,
      MAX(LT.DataVisita) AS NuovaUltima
    FROM LOG_TABLE LT
    WHERE LT.DataVisita <= _up_to OR LT.DataVisita IS NULL
    GROUP BY LT.Paziente )
  SELECT COALESCE(MV.Paziente, AL.Paziente) AS Paziente,
    IF(MV.Paziente IS NULL,
      AL.NuoveVisite,
      MV.NumVisite + IF(AL.NuovaUltima IS NULL, 0, AL.NuoveVisite)),
    IF(MV.Paziente IS NULL,
      IFNULL(Al.NuovaUltima, NULL),
      Al.NuovaUltima
    )
  FROM MATERIALIZED_VIEW MV
    RIGHT OUTER JOIN
    aggregated_log AL USING(Paziente);

  DELETE FROM LOG_TABLE LT
  WHERE LT.DataVisita <= _up_to OR (LT.Paziente IS NULL AND LT.Paziente IN (SELECT Paziente
                    FROM MATERIALIZED_VIEW));
END $$
```

La procedura usa una CTE che aggrega le entrate del log in una comoda tabella che contiene, per ogni paziente, il nome, il numero di visite inserite nel log e la più recente fra queste. Questa tabella fa quindi join con la materialized view, in modo da poter inviduare i record da aggiornare e quelli da creare da zero, attraverso le funzioni COALESCE (che sceglie il primo valore non NULL che trova) e le funzioni IF. Infine, si fa il flush delle tabelle ormai inserite della log table.