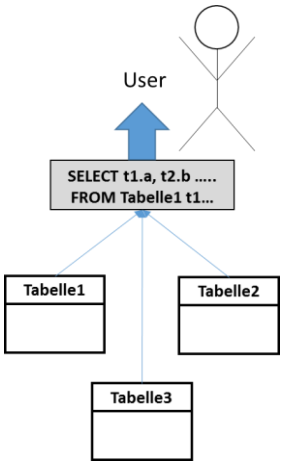
1 Definition

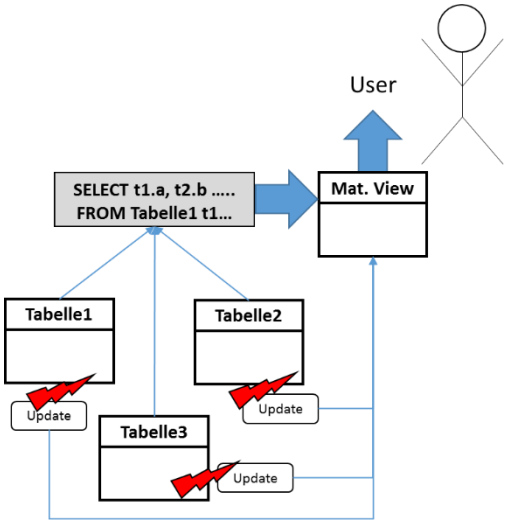
Eine „Materialized View“ ist eine Tabelle, welche das Ergebnis eines Select Statements enthält. Die meisten kommerziellen RDBMS Hersteller bieten derartige Funktionalitäten an, wobei sie mitunter eigene Namen (wie bspw. IBM mit Materialized Query Tables „MQT“ oder Microsoft mit „Indexed Views“) verwenden. MySQL hat keine derartige Funktionalität, wobei sie mit den vorhandenen Mitteln selbst erstellt werden kann. Der Sinn einer Matrialized View ist, die Performance von Abfragen zu verbessern, indem sie gecached werden.

2 Konzept

Die Idee hinter einer View ist unter anderem, dass komplexe Abfragelogik nicht in jeder einzelnen Abfrage realisiert werden muss, sondern diese Abfragelogik als SELECT Statement in einer View gespeichert wird. Diese kann nun von anderen SQL Abfragen genutzt werden.



Die Problematik dabei ist aber, dass üblicherweise die Komplexität der Abfrage auch für Performanceprobleme sorgt. Gerade bei großen Tabellen mit vielen Joins kann dies zu erheblichen Zeitverzögerungen führen.



Das Prinzip der Materialized View ist es nun, nicht (nur) das SELECT Statement zu speichern, sondern das Ergebnis des Selects. Dadurch werden die komplizierten und zeitaufwändigen Aktionen auf der Datenbank nur bei Erstellung der Materialized View durchgeführt.

Dies führt aber in der Konsequenz zu dem Problem, dass bei Änderung der Daten in den Tabellen, auf die das SELECT Statement zugreift, auch die Materialized View nachgezogen werden muss. Eine gewisse Zeit ist also die Materialized View nicht synchron mit den eigentlichen Daten der ursprünglichen Tabellen.

3 Umsetzung in MySQL

Damit das Konzept verstanden wird, realisieren wir eine Materialized View in MySQL selbst. Dazu nehmen wir aus der Datenbank „WarpShop“ die View „ProdPriceFromTo“ und erweitern sie zur Materialized View. Um einen Vergleichstest durchführen zu können, nennen wir die Materialized View „PreisVonBisMat“.

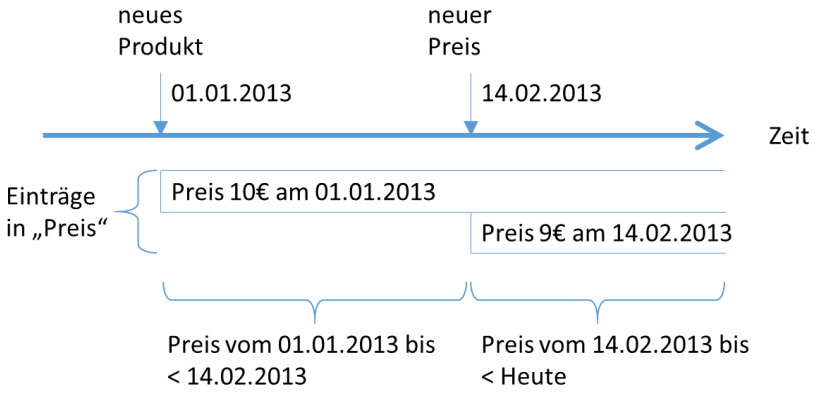
Doch bevor wir beginnen, müssen wir uns über die Funktionalität dieser View Gedanken machen. Sehen wir uns zuerst die Definition der ProdPriceFromTo View an:

```
SELECT p1.ID, p1.ProductID, p1.Price, p1.fromDate,
  MIN(COALESCE(p2.fromDate, '9999-12-31')) AS toDate
FROM ProductPrice p1 LEFT OUTER JOIN ProductPrice p2
  ON( p1.ProductID = p2.ProductID AND p1.fromDate < p2.fromDate)
GROUP BY p1.ID;
```

Um die View zu verstehen, müssen wir uns erstmal das Konzept der Tabelle „ProductPrice“ ansehen. Hier sind für jedes Produkt alle historischen Preise gespeichert:

Feld:	Datentyp:	Bedeutung:
ID	INT	Technischer Schlüssel zur eindeutigen Identifikation des Preis - Datensatzes.
ProductID	INT	Fremdschlüssel zur Identifikation des Produktes, für das die Preisinformation abgelegt wird.
Price	DECIMAL(8,2)	Preis des Produktes zum angegebenen Zeitpunkt.
fromDate	DATE	Datum, zu dem der Preis eingestellt wurde. Dies ist beim erstmaligen Einstellen des Produktes der Fall und bei jeder Preisänderung.

Hier ein Beispiel für ein Produkt mit dem Preis von 10€, welches am 01.01.2013 erstmalig eingestellt wurde. Am 14.02.2013 wurde ein neuer Preis von 9€ abgelegt. Insofern finden wir in der Preis-Tabelle zwei Einträge zu dem Produkt.



Das Problem ist, dass wir am 01.01.2013 noch nicht wissen konnten, dass am 14.02.2013 ein neuer Preis gilt. Insofern trägt der Datensatz für die 10€ auch kein „bis“ Datum. Dieses wird nun von der View ProdPriceFromTo automatisch erzeugt. Gehen wir die View nun Schritt für Schritt durch:

```
SELECT p1.ID, p1.ProductID, p1.Price, p1.fromDate
```

Die Felder ID, ProductID, Preis und fromDate können 1:1 übernommen werden. Lediglich das toDate ist nicht in der Tabelle vorhanden und muss nun berechnet werden. Da das Datum allerdings nicht vom aktuellen Datensatz, sondern vom zeitlich nächsten Datensatz (also dem „Nachfolger“) ermittelt werden kann, müssen wir die Tabelle „Preis“ zweimal selektieren und mit einem Join über die ProductID verbinden. Dies muss aber mit einem OuterJoin passieren, da der letzte (also aktuellste) Eintrag keinen „Nachfolger“ hat:

```
FROM ProductPrice p1 LEFT OUTER JOIN ProductPrice p2
  ON( p1.ProductID = p2.ProductID AND p1.fromDate < p2.fromDate)
     GROUP BY p1.ID;
```

Hier müssen aber nur diejenigen Datensätze verjoined werden, welche vom Datum her größer als der Vorgänger ist. Somit liegt p1 immer zeitlich vor p2. Wer genau hinsieht erkennt, dass wir bei mehr als einem „Nachfolger“ alle joinen. Beim späteren Group By pro ID wird jedoch immer nur einer genommen. Da wir aber das jüngste darauf folgende Daten benötigen (also das erste nach dem aktuellen), greifen wir mit Hilfe eines MIN() das Datum ab, wodurch pro Gruppe immer das richtige Datum erhalten.

Als nächstes müssen wir uns um den letzten Datensatz pro ProductID kümmern. Hierzu nehmen wir uns denjenigen, bei dem das Datum p2.Datum „null“ ist – das muss also der letzte (aktuellste) Datensatz pro ProductID sein, da dieser über den outer JOIN erzeugt wurde. Dort müssen wir anstatt dem „null“ Wert das heutige Datum (plus 1) setzen. Die eleganteste Möglichkeit hierfür ist die „COALESCE(field, value)“ Funktion. Diese wertet die Spalte „field“ aus und übernimmt sie 1:1, sofern kein „null“ Wert gefunden wurde. Sollte der Inhalt der Spalte allerdings „null“ sein, so wird stattdessen der Wert „value“ ausgegeben:

```
  MIN(COALESCE(p2.fromDate, '9999-12-31')) AS toDate
```

Zum Schluss gruppieren wir die Ergebnisse nach p1.ID, um die gleiche Granularität wie „Preis“ zu haben:

```
GROUP BY p1.ID;
```

Nun, da wir die View „ProdPriceFromTo“ verstanden haben, können wir uns um die Materialisierung der View kümmern. Dazu machen wir uns über die Datenquellen Gedanken. Wir haben hier eine sehr einfache Situation, in der wir lediglich eine Tabelle als Quelle haben (diese nutzen wir in der View zwar zweimal, aber eben zweimal die gleiche Tabelle). Insofern benötigen wir auch nur auf dieser Tabelle die Trigger, welche sich um die Aktualisierung der Materialized View kümmern.

Trigger-auslöser:	**Wann:**	**Bedeutung:**
INSERT	oft – dies ist der Standardfall	Jede Preisänderung und jedes neue Produkt würde zu einem INSERT Statement führen. Dies ist also für uns die „Top Priorität“. Für jedes INSERT müssen wir mindestens einen Datensatz anlegen und ggf. einen anpassen.
UPDATE	selten	Bei einem Update müssen wir zuerst analysieren, was geändert wird. Wenn Keys verändert werden, müssen wir alle betroffenen Datensätze in unserer View neu anlegen. Bei Datumsveränderungen müssen die zeitlich angrenzenden Datensätze angepasst werden. Bei Preisanpassungen lediglich der Preis.
DELETE	selten	Bei einem DELETE müssen die Datensätze mit der gleichen ID ebenfalls gelöscht werden und die betroffenen Datumswerte angepasst werden.

Beginnen wir nun mit dem eigentlichen (initialen) Erstellen der Materialzed View. Hierzu müssen wir die Tabelle erzeugen:

```
CREATE TABLE ProdPriceFromToMat (
      ID INT NOT NULL PRIMARY KEY,
      ProductID INT NOT NULL,
      Price DECIMAL(8,2) NOT NULL,
      fromDate DATE,
      toDate DATE
      );
CREATE INDEX id_ProductIDMat ON ProdPriceFromToMat (ProductID);
CREATE INDEX id_FromDateMat ON ProdPriceFromToMat (fromDate);
CREATE INDEX id_ToDateMat ON ProdPriceFromToMat (toDate);
```

Danach füllen wir die Tabelle initial. Hierbei kombinieren wir das INSERT mit dem SELECT Statement. Da wir beim Datum aufgrund der Vereinfachung eine Änderung vornehmen, müssen wir das SELECT der View explizit angeben und das Datum von current_date() auf 9999-12-31 anpassen:

```
INSERT INTO ProdPriceFromToMat (ID, ProductID, Price, fromDate, toDate)
SELECT p1.ID, p1.ProductID, p1.Price, p1.fromDate,
MIN(COALESCE(p2.fromDate, '9999-12-31'))  AS toDate
FROM ProductPrice p1
LEFT JOIN ProductPrice p2
  ON( p1.ProductID = p2.ProductID AND p1.fromDate < p2.fromDate)
GROUP BY p1.ID;
```

Nun haben wir die View materialisiert. Ein kurzer Test mit einem simplen count(*) auf die ProdPriceFromTo und ProdPriceFromToMat zeigt, dass wir hier einen erheblichen Performancegewinn verbuchen können.

Nun kümmern wir uns um das INSERT. Hierbei gehen wir vereinfacht vor. Bei einem neuen Preis für ein Produkt löschen wir alle Einträge dieses Produktes in ProdPriceFromToMat und erzeugen einfach alle neu:

```
DELIMITER $$
CREATE TRIGGER INS_ProdPriceFromToMat AFTER INSERT ON ProductPrice
  FOR EACH ROW BEGIN
    REPLACE INTO ProdPriceFromToMat (ID, ProductID, Price,
      fromDate, toDate)
    SELECT p1.ID, p1.ProductID, p1.Price, p1.fromDate,
```

```
    MIN(COALESCE(p2.fromDate, '9999-12-31'))  AS toDate FROM
      ProductPrice p1 LEFT OUTER JOIN ProductPrice p2
     ON( p1.ProductID = p2.ProductID AND p1.fromDate < p2.fromDate)
      WHERE p1.ProductID = NEW.ProductID
    GROUP BY p1.ID;
END$$
DELIMITER ;
```

Bevor wir unseren Trigger testen können, müssen wir hier erstmal die Voraussetzungen schaffen. Die Datenbank ist derzeit noch in einem Zustand, der für die reine Abfrage ausgelegt ist. Die IDs wurden noch nicht als Autoincrement Felder festgelegt. Dies müssen wir nun (zumindest für die Preis Tabelle) ändern:

```
ALTER TABLE ProductPrice CHANGE ID ID INT AUTO_INCREMENT;
```

Danach sehen wir uns die Daten eines einzelnen Produkts an:

```
SELECT * FROM ProductPrice WHERE ProductID = 1007765 ORDER BY ID;
```

Und zum Vergleich aus unserer Materialisierten View:

```
SELECT * FROM ProdPriceFromToMat WHERE ProductID = 1007765 ORDER BY ID;
```

Bis auf das DatumBis des letzten Datensatzes müssen die Werte nun identisch sein. Nun fügen Sie für das Produkt einen neuen Datensatz ein und prüfen, ob der Trigger seine Arbeit getan hat.

4 Aufgabe

Erstellen sie die Materialized View für Update und Delete.
Hinweise:
Nun fehlen noch die Situationen, dass Datensätze gelöscht oder geändert werden können. Die Fälle sind zwar selten, führen jedoch trotzdem zu einer Änderung des Datenbestandes, welche in der Tabelle ProdPriceFromToMat berücksichtigt werden müssen. Die Trigger für **UPDATE und DELETE** werden sehen genauso aus, wie der INSERT Trigger. Lediglich beim DELETE Trigger müssen vorher noch alle Datensätze, welche zum gleichen Zeitpunkt (oder später) wie der geänderte Datensatz zu dem Produkt aufgetreten sind, gelöscht werden:

```
DELIMITER $$
CREATE TRIGGER DEL_ProdPriceFromToMat AFTER INSERT ON ProductPrice
  FOR EACH ROW BEGIN
    DELETE FROM ProdPriceFromToMat
    WHERE ProductID = OLD.ProductID
    AND fromDate >= OLD.fromDate;

    REPLACE INTO ProdPriceFromToMat (ID, ProductID, Price,
      fromDate, toDate)
      SELECT p1.ID, p1.ProductID, p1.Price, p1.fromDate,
      MIN(COALESCE(p2.fromDate, '9999-12-31'))  AS toDate
      FROM ProductPrice p1
      LEFT OUTER JOIN ProductPrice p2
      ON( p1.ProductID = p2.ProductID AND p1.fromDate < p2.fromDate)
      WHERE p1.ProductID = NEW.ProductID
      GROUP BY p1.ID;
  END$$
DELIMITER ;
```