Incremental View Maintenance For MIPMap

1. Problem Statement and MIPMap

Assume we have a source schema, a target schema and a set of tuple generating dependencies (TGDs). We can think of relations of the target schema as **materialized views** of the source schema. Can we do something better than computing the materialized view from scratch every time we have new incoming batches of source data? Ideally, can we infer an executable of minimal execution load by taking into account the **modifications** (new insertions, new updates, new deletions) done on the data of the source schema and the existing **tgd rules?** In the literature of data integration this concept is called **incremental view maintenance**. In our case, we need to adapt techniques used for incremental view maintenance to achieve **incremental data exchange**.

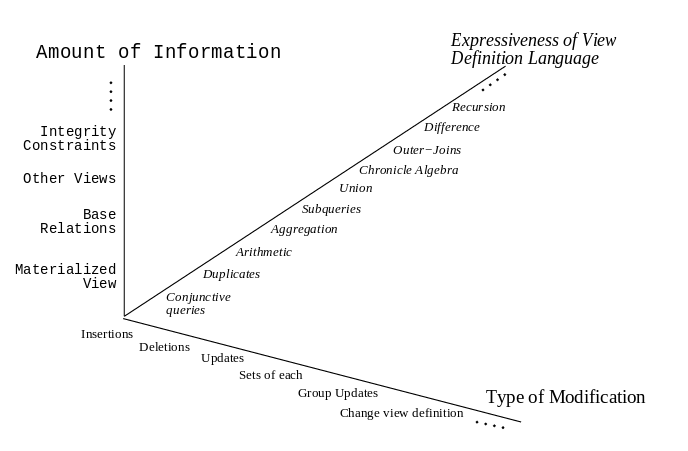
To understand why this concept is important for MIPMap consider the following use case. Let *hbp\_diags(h\_id,..., d\_code)* be a relation of the source schema representing the possible diagnostics of a hospital, *hbp\_patient(h\_id, p\_id,...)* to be a relation of the source schema representing a patient of a hospital and *diagnostic*(d\_id, p\_id, d\_code) be a relation of the target schema representing diagnostic events. Consider the following tgd rule:



Assume we receive a first batch of new data that contain the tuple t1: *hbp\_patient(1,p1,...)* but no hbp\_diags entries related to h\_id = 1. This implies that the aforementioned rule will not be triggered for t1, hence the diagnostic table will not be populated with any information about p1 (related to h\_id = 1). Consider now that a second batch of data arrives that contains the tuple t2: *hbp\_diags(1,…, code1)*. Again, t is not triggered. In order for t to be triggered we need to have both t1 and t2 available simultaneously. Hence, assuming the absence of an incremental data exchange functionality currently in MIPMap, we are forced to re-run data exchange from scratch, an operation with a potentially high computational cost.

2. Working with Materialized Views: Problem Dimensions

If we think of the target schema relations as materialized views of the source schema relations, we can identify four dimensions along which the problem can be studied, as defined in [1]. The implemented solution for MIPMap should take into account our standing with regard to each one of them. Below, we present the dimensions along with remarks on the needs of MIPMap.



* **Information Dimension:** Is there access to all source schema relations while doing the maintenance? What about integrity constraints and keys? (**MIPMap**: we can assume a full information setting.)
* **Modification Dimension**: Which operations are we planning to support with our proposed view maintenance problem? Moreover, conceptually what does an “update” operation mean (insert-delete or stand-alone operation)? (**MIPMap**: initially our focus will be on supporting insert operations)
* **Language Dimension:** How is the materialized view expressed? Examples are SQL, subset of SQL, DataLog etc.Do we need to support duplicates or data aggregation? (**MIPMap**: the target schema relations are expressed using tuple generating dependencies that are eventually translated to SQL queries, whereas MIPMap doesn’t support aggregations).
* **Instance Dimension:** Does the view maintenance algorithm work for every instance of the database of just for some subsets of data? (**MIPMap**: Ideally our solution should support all instances of ιnsert)

To get a better understanding on why determining where we stand on each of the problem dimensions is important for sorting the details of our approach, we present another motivating example from [1] with regard to the **information** dimension.

Assume the following source relation with car parts:



And the following materialized view that is defined as follows:



Notice that the materialized view contains all car parts with a cost greater than 1000.

Assume we insert a new tuple to *part.* Obviously, no action should be takenif the field part\_cost of the tuple has a value of 1000 or less. Conversely, if part\_cost> 1000 we need to check whether the materialized view would change.

Let the new tuple be **part(p1, 5000, c15).**

* In a full information setting (access to the materialized view, knowledge of the full local schema and its constraints) we can check whether there is a tuple with p1 in the materialized view. If there is one, the view once again remains unchanged. If there isn’t one it is inserted. Even better, when part\_no is a primary key we can infer that we don’t need to lookup the materialized view at all. It is obvious that an insert operation should be performed.
* Alternatively we can perform a lookup to the local relation and and check whether there is an existing tuple with p1 along with an other contract with a cost >= 1000. Then, this tuple would have no contribution to the materialized view. In this case we exploit our knowledge of the materialized view definition (the SQL query above that populates it).

Hence, the information available is critical on choosing a methodology to efficiently maintain the materialized view. We refer the reader to [1] for examples of maintaining materialized views with regard to other problem dimensions (how expressive is the view definition language, operations supported etc.)

3. Self-maintainable materialized views

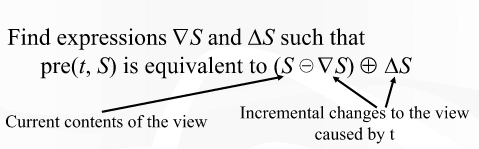
Another important concept in the theory of materialized views maintenance is self maintainability. Views that can be maintained using only the materialized view and key constraints are called self-maintenable.

4. Incremental View Maintenance: The Textbook Approach

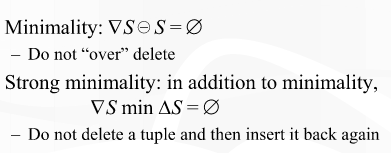
In a data warehousing environment (as is the case of Human Brain) we can basically view the warehouse as a set of materialized views based on one or many underlying data sources. New batches of data arrive to the source databases and existing data can be changed. We need to maintain the materialized view in an efficient way (e.g. not recompute from scratch every time).

A prominent paper in the literature is by Griffin and Libkin titled *Incremental Maintenance for Views With Duplicates* [2]*.* In the context of this report, we shall regard this as the textbook approach (see also [3] lecture slides on incremental view maintenance). The main point is as follows. Assume a set of local relations and a materialized view present. Now, imagine the local relations are altered after a batch of modification. We need to identify two subsets of tuples: the tuples of the local relations that need to be **inserted to the materialized view** and the subset of tuples that need to be **deleted from the materialized view**. The approach is to materialize two additional local relations containing the subsets of tuples mentioned above. For a relation S we notate these extra relations as (to be inserted) and (to be deleted). This method is sometimes called **algebraic differencing (**as opposed to the algorithmic approach).

Formally, the problem is defined below for a transaction t (a batch of new data).



pre(t,S) represents a full recomputation of the materialized view. The idea is to find an equal expression to compute pre(t,S) efficiently. Notice that theoretically  is a possible solution as well (basically, we don’t do anything incrementally and just compute the materialized view from scratch). Intuitively, a good solution should not look similar to this extreme. For a “good” solution we need “good” Deltas. Roughly, in this model good Deltas are those that adhere as much as possible to the following rules:



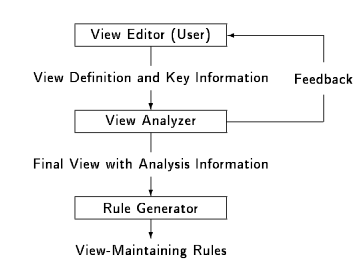
The challenge here is to compute such Deltas efficiently. In [2], the authors suggest an algorithm that propagates changes from base relations to materialized views based on equational reasoning and prove that their algorithm maintains a notion of minimality while efficiently handling the occurrence of duplicates.

**Note:** We refer the reader to [3] for bag algebra concepts, a necessary framework to efficiently handle duplicates.

4. Incremental View Maintenance with Production Rules

An alternative way to perform incremental view maintenance in contrast with [2] is presented in [4]. Instead of the algebraic approach the authors focus on *production rules****.*** A production rule in a DBMS is a set of data manipulation operations that are executed automatically when certain events occur or conditions are met. So, when source relations are modified (e.g. new batches of data arrive) the production rules are triggered and the view is modified. Once, again the baseline solution as in the case of algebraic differencing is to have a “dummy” rule that recomputes the materialized view from scratch. The authors present a framework that automatically derives production rules for many classes of materialized views definitions.

When a view is generated or “compiled” (think of the SQL query defining the materialized view running once), the proposed framework derives a set of production rules as executables by performing syntactic analysis on the view definition SQL query.

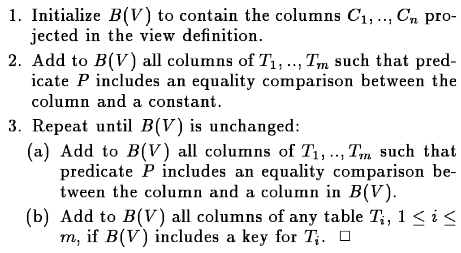


There are two main drawbacks with this way of handling things. Firstly, there is no support for duplicate tuples in the materialized view. Secondly, should the system detect that efficient rules cannot be derived for a source relation the user is asked to act. Possibly, the user may have to alter the source schema or add additional constraints. In the above diagram, this interaction is the *Feedback* arrow.

Consider a materialized view of the following form:

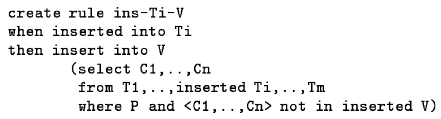


where Ti, i=1,..,m are source relations, Cj, j=1,..,n are columns and P is a predicate. We define a set of columns B(V) called **Bounded Columns** in the following way:



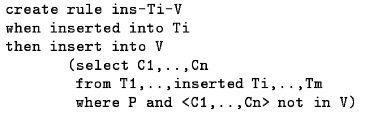
We label Ti relations as **safe** and **unsafe** for the maintenance of V**.** Safe relations provide a theoretical guarantee that there are efficient maintenance rules for its underlying data.Safe relations are those which B(V) contains a **key** for Ti.

Let us focus on the “new insert” case, namely assume we want to insert tuples to relation Ti. In the case of a safe relation the insert production rule is the following:



where **inserted Ti** is an artificial relation of the inserted tuples in the new data batch (a Delta for V). Since the relation is safe, it is again guaranteed that the insertion cannot create more duplicates. However, we need to be wary that a different rule (e.g. ins-Tj-V, j != i then duplicates may appear. Hence, before inserting a new tuple with this rule we ensure that the tuple has not been already inserted (by checking against **inserted V)**.

The approach is similar when Ti is labeled as unsafe. To guarantee that no duplicates are created in this case we check against **V** instead of **inserted V.** Hence, the rule is:



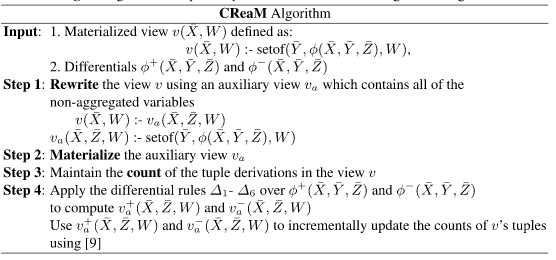
For delete and update operations things may get a bit harder. We refer the reader to the paper for a more complex presentations of other production rules.

5. Incremental View Maintenance of Aggregate Views

Currently MIMPAP does not support aggregations. However, this extension should not be difficult to implement and it is a feature that would be nice to have. [5] presents a state of the art approach of maintaining materialized views populated by aggregation tgds. The proposed algorithm called *CreaM* guarantees optimality and self-maintainability for views that are defined over a single database table and is still very efficient when when aggregate views are defined over multiple source relations.

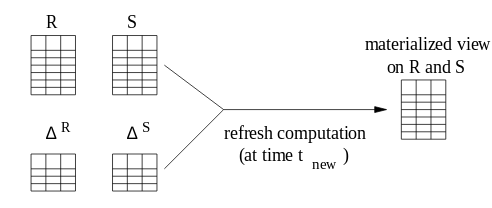
The authors compute differentials of the materialized view. The correctness is ensured by defining a set of *differential rules.* These are operations that are performed on the materialized view and ensure that the differential relations are correctly computed in response to a change to the underlying source relations.

The algorithm is the following:

The speed up to update the materialized view with n tuples is *n/logn* or *logn* depending on whether the materialized view is indexed or not.

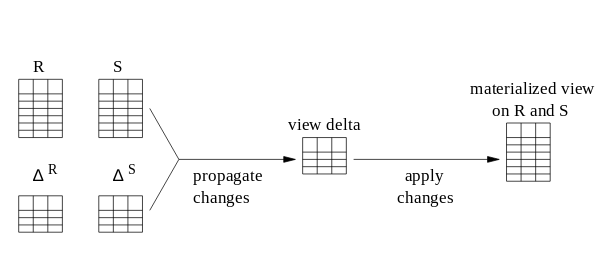
6. Asynchronous Incremental View Maintenance

Another important perspective on incremental view maintenance is offered in [6]. The intution behind the work is th fact that even though incremental view maintenance is usually less expensive that the computation of the materialized view from scratch it may still be a quite costly operation. Therefore, authors suggest a methodology of incremental view maintenance based on a series of small, asynchronous steps for additional performance gains. The proposed approach also supports *point in time refresh* an option that offers the user the flexibility to obtain the state of the materialized view at an intermediate time (between the current time of the materialized view and the time of the materialized view). However, this should be done with care so that it is ensured that the materialized view remains in a transaction-cosistent intermediate state.



The approach covers the following use case: what if the underlying tables get updated while the incremental refresh operation is running? An obvious first thought would be to handle the possibly long transaction of incremental view maintenance by using small refresh intervals. The downside of this is that the materialized view is “too updated”. There are cases when this is undesirable. For example, imagine that the materialized view contains daily data and therefore should only be updated at a designated point once per day. Moreover, there are use cases when it cannot be decided when the refresh cost should be payed.

A baseline potential solution to these issues is presented in [4ofpaper]. We could think of splitting the computation into two phases: the *propagation* phase and the *apply* phase (the process are independent apart from their natural producer/consumer relationship). During the propagation phase the changes to the view are computed and stored seperately. Later, during the apply phase the view is actually updated.



The authors propose a similar extension of the baseline described above. The key advantages of their approach are:

1) The view delta propagation is asynchronous.

2) The view delta propagation is a continuous process. A number of small transactions is being used instead of a single large transaction.

3) The changes recorded in the delta are timestamped to facilitate the control of propagation sizes and enable point in time incremental view maintenance.

7. Incremental View Maintenance in Nested Data Models

So far, all approaches presented assume that the materialized view is defined in terms of one or more flat relations. In [7]] we get an insight on working with more expressive view definitions such as the nested relational model (a tuple can have arbitrarily deeply nested structures) with arbitrary levels of nesting) or an object oriented model. The authors show that such views can be maintened by a modification of the counting algorithm [countingalgo]. The approach is to create a structure that transparently flattens records and, with the usage of appropriate data structures the view maintenance overhead is minimized. The approach works views defined by SP (select-project) and SPJ (select-project-join) operations with an arbitrary level of nesting. Note that the only nested objects that are considered are those where a tuple is just a reference to an other relation (structures where tuples are actually nested relations are not considered)

8. Conclusions

We presented a number of prominent techniques that explore different dimensions for the incremental view maintenance problem. In order to decide a modus operandi for MIPMAP, we should clearly understand where we stand on each of the dimensions defined in section 2 with regard to the current functionality or functionality to be added in the forseeable future (e.g. are we planning to support aggregations?). Then, this document and its references should act as a pool of ideas on how to proceed to the implementation.

9. References

[1] Gupta, A., & Mumick, I. S. (1995). Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, *18*(2), 3-18.

[2] Griffin, T., & Libkin, L. (1995, June). Incremental maintenance of views with duplicates. In *ACM SIGMOD Record* (Vol. 24, No. 2, pp. 328-339). ACM.

[3] Topics In Database Systems, 05-06, Lecture Notes. Link <https://www.cs.duke.edu/courses/spring02/cps296.1/lectures/07-view.pdf>

[4] Ceri, S., & Widom, J. (1991). Deriving production rules for incremental view maintenance.

[5] Mohapatra, A., & Genesereth, M. (2014, March). Incremental maintenance of aggregate views. In *International Symposium on Foundations of Information and Knowledge Systems* (pp. 399-414). Springer International Publishing.

[6] Salem, K., Beyer, K., Lindsay, B., & Cochrane, R. (2000, May). How to roll a join: Asynchronous incremental view maintenance. In *ACM SIGMOD Record* (Vol. 29, No. 2, pp. 129-140). ACM.

[7] Kawaguchi, A., Lieuwen, D., Mumick, I., & Ross, K. (1997, August). Implementing incremental view maintenance in nested data models. In *International Workshop on Database Programming Languages* (pp. 202-221). Springer Berlin Heidelberg.