may be materialized as a separate table in the database.

In relational systems, a view is defined using the **create view** command:

- ***create view*** < v > **as** < *query expression* >

The name of the view in the above example is < v >, and the schema and content of the view are derived on demand by the evaluation of < query expression >, which should be a legal expression supported by the database management system. Different vendor systems may impose some constraints on the form of < query expression >; for instance, they may disallow references to temporary tables. When the data in the table(s) mentioned in < query expression > changes, the data in view < v > changes also.

Consider the following view definitions:

- *Create view v1 as select Name, Age from Personnel where Department = "Sales"*
- *Create view v2 as select Wages.Name, Wages. Salary from Personnel, Wages where Personnel.Name = Wages.Name and Personnel.Department = "Sales"*

The first expression defines a view termed *v1* that contains the name and age attributes from database table *Personnel*. The instance of view *v1* consists of the subset of personnel data restricted to those working at department *Sales*. View *v2* defined by the second expression contains the result of the join between tables *Personnel* and *Wages* for employees working at the sales department.

Cross-References

► View Maintenance Aspects
► Views

Recommended Reading

1. Adiba ME, Lindsay BG. Database snapshots. In: Proceedings of the 6th International Conference Symposium on Very Data Bases; 1980. p. 86–91.
2. Dayal U, Bernstein P. On the correct translation of update operations on relational views. ACM Trans Database Syst. 1982;8(3):381–416.
3. Gupta A, Jagadish HV, Mumick IS. Data integration using self-maintainable views. In: Advances in Database Technology, Proceedings of the 5th International Conference on Extending Database Technology; 1996. p. 140–44.
4. Gupta H, Harinarayan V, Rajaraman A, Jeffrey DU. Index selection for OLAP. In: Proceedings of the 13th International Conference on Data Engineering; 1997. p. 208–19.
5. Kotidis Y, Roussopoulos N. DynaMat: a dynamic view management system for data warehouses. In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 1999. p. 371–82.
6. Roussopoulos N. View indexing in relational databases. ACM Trans Database Syst. 1982;7(2):258–90.
7. Roussopoulos N. An incremental access method for viewCache: concept, algorithms, and cost analysis. ACM Trans Database Syst. 1991;16(3):535–63.

View Maintenance

Alexandros Labrinidis[1] and Yannis Sismanis[2]
[1]Department of Computer Science, University of Pittsburgh, Pittsburgh, PA, USA
[2]IBM Almaden Research Center, Almaden, CA, USA

Synonyms

Materialized view maintenance; View update

Definition

View maintenance typically refers to the updating of a *materialized view* (also known as a derived relation) to make it consistent with the base relations it is derived from. Such an update typically happens *immediately*, with the transaction that updates the base relations also updating the materialized views. However, such immediate updates impose significant overheads on update transactions that cannot be tolerated by many applications. *Deferred view maintenance*, on the other hand, allows the view to become inconsistent with its definition, and a *refresh operation* is required to establish consistency. Typically, under deferred maintenance, a view is *incrementally* updated only just before data is retrieved from it (i.e., on-demand, just before a query is performed on the view).

Historical Background

Early systems that supported views did so in their "pure form," i.e., by storing just the view definition and using query rewriting to take advantage of views in other queries [11].

Incremental view maintenance is introduced in [1] through a technique to efficiently detect relevant updates to materialized views, thus streamlining their maintenance.

Deferred view maintenance is introduced in [10] as a scheme for materializing copies of views on workstations attached to a mainframe that maintains a shared global database. The workstations update local copies of the views while processing queries. In [7], deferred view maintenance is defined as the application of incremental view maintenance whenever desired, unlike the immediate view maintenance, where any database update triggers the incremental view maintenance algorithm. [5] has a nice survey of view maintenance techniques.

Foundations

Algorithms and techniques for maintenance of materialized views can be classified according to three different criteria:

- Whether the view is recomputed from scratch or not: *recomputation* versus *incremental* maintenance.
- Whether the view is updated whenever the base data change or not: *immediate* versus *deferred* maintenance.
- Whether queries can be executed while the view is being updated or not: *online* versus *offline* maintenance.

All the above dimensions are typically orthogonal. We explain the different options below.

View Recomputation

Recomputing a materialized view from the base relations it is derived from is the most general technique of updating. As such, it can be applied on any type of view, regardless of the complexity of the query definition. The disadvantage is that, in most cases, such recomputation is costly, and, in many cases, the view can be updated *incrementally* instead, at a fraction of the cost.

Incremental View Maintenance

It is possible to update a materialized view *incrementally* for many types of view definitions (i.e., queries). One such class is the general case of SPJ views (i.e., views whose definition is just a select-project-join query).

For example, assume that we have a view V defined over two relations R and S through a natural join (i.e., $V = R \bowtie S$; for simplicity of the presentation we ignore the selection and projection operators). Further, let us assume that we have a set of deleted tuples from relation R, denoted as R_D; a set of inserted tuples into relation R, denoted as R_I (i.e., $R' = R \cup R_I - R_D$). Also, assume a set of deleted tuples from relation S, denoted as S_D; and a set of inserted tuples into relation S, denoted as S_I (i.e., $S' = S \cup S_I - S_D$). We trivially represent base relation updates as pairs of deletions and insertions.

Given the above, the updated version of V, i.e., V', should be $V' = R' \bowtie S' = (R \cup R_I - R_D) \bowtie (S \cup S_I - S_D)$. By expanding this further, and grouping all the deletions from V as V_D and all the insertions to V as V_I, we have that: $V_D = (R_D \bowtie (S \cup S_I)) \bowtie ((R \cup R_I) \bowtie S_D)$, and $V_I = (R_I \bowtie S) \cup (R \bowtie S_I) \cup (R_I \bowtie S_I)$, so that $V' = V \cup V_I - V_D$. This, incrementally computed formula, should be less costly to compute than recomputing the entire join from scratch.

The problem of incrementally updating materialized views is difficult in the general case, but there are additional classes of queries (i.e., besides SPJ views) that it can be solved for [6].

Immediate View Maintenance

The default way of updating materialized views is to do so *immediately*, i.e., batch together, in a single transaction, the updating of the base relations and the updating of the materialized views that are derived from these relations. However, many applications cannot tolerate this delay, especially if they are interactive and users are expecting an answer at transaction commit.

Deferred View Maintenance

Incremental deferred view maintenance requires (i) techniques for checking what views are affected by an update to the basic tables, (ii) auxiliary tables that maintain certain information like updates and deletes since the last view refresh and finally (iii) techniques for propagating the changes from the base tuples to the view tuples without fully recomputing the view relation.

First, Buneman in [2], proposes a technique for the efficient implementation of alerters and triggers that checks each update operation prior to execution to see whether it can cause a view to change. In [1], an efficient method for identifying updates that cannot possibly affect the views is described. Such *irrelevant updates* are then removed from consideration while differentially updating the views.

In [7], the hypothetical relations technique developed in [12] is adapted to the purpose of storing and indexing the deltas to the base tables. The main idea is to use a single table *AD* that stores deletions and insertions for the base tables (updates can be modeled as a deletion followed by an update). Whenever a view is accessed, the base tables and the *AD* table need to be accessed (in order to check for new or deleted tuples). A bloom filter however, is used to check if a tuple from the base relation exists in *AD* significantly reducing irrelevant accesses to *AD*.

In [4], the authors demonstrate that the ordering of the updates from the base tuples to the view tuples is critical and call this phenomenon *state bug*. Typically, an “incremental query” – during the refresh operation – avoids recomputing the full view and only incrementally computes the delta view to bring it up to date, based on updates/deletes made to the base tables. Such incremental queries can evaluated in two states: The *pre-update state*, where the base table updates have not been applied yet or the *post-update state* where changes have been applied. In most techniques a pre-update state is assumed which severely limits the class of updates and views considered. The post-update state allows for a much larger class of view to be deferred maintained, however direct application of pre-update techniques results in incorrect answers (state bug) and new techniques are proposed.

Offline View Maintenance

Typically, maintaining materialized views is done *offline*, without allowing queries to the materialized view to execute concurrently with the processing of the materialized view updates. This simplifies the view maintenance algorithms significantly, at the expense of delaying queries. Traditionally, in data warehousing environments [3], updates of materialized views are performed at night, thus minimizing the possibility of delaying user queries.

Online View Maintenance

The need of most companies for continuous operation (especially in the presence of the Web), has precipitated the need for *online view maintenance*, where queries can be answered while the materialized views are being updated.

In a centralized setting, this is typically achieved through some sort of multi-versioning, either as *horizontal redundancy*, where extra columns are added to hold the different versions [9], or as *vertical redundancy*, where extra rows are needed to hold the different versions [8]. In a distributed setting, this is typically achieved through determination of additional queries to ask of the data sources [13].

Key Applications

Materialized views help speed up the execution of frequently accessed queries, giving interactive response times to even the most complex queries. The cost of maintaining materialized views is typically amortized over multiple accesses (i.e., queries to the view). This has been utilized/transferred in many different application domains, from data warehousing to web data management. Beyond efficient algorithms and techniques to update materialized views, special attention has also been given to the *view selection* problem: how to identify which views should be materialized, and also to the issue of how to effectively use materialized views to answer other queries (i.e., by utilizing subsumption or caching).

Cross-References

▶ Maintenance of Recursive Views

Recommended Reading

1. Blakeley JA, Larson PÅ, Tompa FW. Efficiently Updating Materialized Views. In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 1986. p. 61–71.
2. Buneman P, Clemons EK. Efficient monitoring relational databases. ACM Trans Database Syst. 1979;4(3):368–82.
3. Chaudhuri S, Dayal U. An overview of data warehousing and OLAP technology. ACM SIGMOD Rec. 1997;26(1):65–74.
4. Colby LS, Griffin T, Libkin L, Mumick IS, Trickey H. Algorithms for deferred view maintenance. In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 1996. p. 469–80.
5. Gupta A, Mumick IS. Maintenance of materialized views: problems, techniques, and applications. IEEE Data Eng Bull. 1995;18(2):3–18.
6. Gupta A, Mumick IS, Subrahmanian VS. Maintaining views incrementally. In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 1993. p. 157–66.
7. Hanson EN. A performance analysis of view materialization strategies. In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 1987. p. 440–53.
8. Labrinidis A, Roussopoulos N. A performance evaluation of online warehouse update algorithms. Technical report CS-TR-3954, Department of Computer Science, University of Marylan. 1998.
9. Quass D, Widom J. On-line warehouse view maintenance. In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 1997. p. 393–404.
10. Roussopoulos N, Kang H. Principles and techniques in the design of ADMS±. IEEE Comp. 1986;19(12):19–25.
11. Stonebraker M. Implementation of integrity constraints and views by query modification. In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 1975. p. 65–78.
12. Woodfill J, Stonebraker M. An implementation of hypothetical relations. In: Proceedings of the 9th International Conference on Very Data Bases; 1983. p. 157–66.
13. Zhuge Y, Garcia-Molina H, Hammer J, Widom J. View maintenance in a warehousing environment. In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 1995. p. 316–27.

View Maintenance Aspects

Antonios Deligiannakis
University of Athens, Athens, Greece

Definition

Database systems often define *views* in order to provide conceptual subsets of the data to different users. Each view may be very complex and require joining information from multiple *base relations*, or other views. A view can simply be used as a query modification mechanism, where user queries referring to a particular view are appropriately modified based on the definition of the view. However, in applications where fast response times to user queries are essential, views are often materialized by storing their tuples inside the database. This is extremely useful when recomputing the view from the base relations is very expensive. When changes occur to their base relations, materialized views need to be updated, with a process known as view maintenance, in order to provide fresh data to the user.

Historical Background

The use of relational views has long been proposed in relational database systems. The notion