IJCSC International Journal of Computer Science & Communication Vol. 1, No. 2, July-December 2010, pp. 223-226

Distributed Database & SD-SQL Server Architecture

S.B.Rathod[1], S.R.Deshmukh[2] & H.R.Vyawahare[3]

[1,2,3]Department of Computer Science & IT, S.G.B.Amaravati University, Amravati, India
Email: [1]Seemarathod47@rediffmail.com,[2]deshmukhshilpa87@yahoo.co.in, [3]harsha_vyawahare@rediffmail.com

ABSTRACT

Databases are now often huge and growing at a high rate. Large tables are then typically hash or range partitioned into segments stored at different storage sites. Current Data Base Management Systems (DBSs), e.g., SQL Server, Oracle or DB2, provide static partitioning only. The database administrator (DBA) in need to spread these tables over new nodes has to manually redistribute the database (DB). A better solution has become urgent. The structure of data to be stored by a Data Base Management System (DBMS) is usually decided by a database administrator. Individual users and applications are generally interested in only a subset of the data stored in the database. Often, they wish to see this subset structured in a way which reflects their particular needs. Since it is not generally possible to structure a database so as to please all of its users, some mechanism is needed whereby each user can view the data according to his (her) own requirements. The representation of the data structure as seen by a user is often referred to as an external schema; the view mechanism is a means by which a DBMS can support various external schemas.

Keywords: View Maintenance, SD-SQL Server Architecture, Scalable Database Management.

1. INTRODUCTION

1.1. Background

1.1.1. Constraints in Distributed Databases

A distributed database is a database distributed between several sites. The reasons for the data distribution may include the inherent distributed nature of the data or performance reasons. In a distributed database the data at each site is not necessarily an independent entity, but can be rather related to the data stored on the other sites. This relationship together with the integrity assertions on the data are expressed in the form of data constraints. Figure 1 shows a classification of the possible data constraints that can hold on a distributed database.

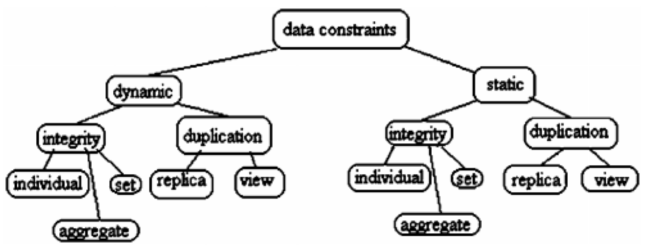

Fig.1: Distributed Database Data Constraint Classification

At the top level there are two type of constraints dynamic and static. To check whether a *static constraint* holds, at any given time point, we need a static snapshot of the database instance. On the other hand, to verify whether a *dynamic constraint* holds we need to have information about how the database instance evolves over time. An example of a dynamic constraint is: within every 5 minutes of the life of the database instance something must happen, e.g. an update must occur. On an orthogonal plane, data constraints can be integrity or duplication.

Integrity constraints are independent of how the data is distributed or duplicated If an integrity constraint is based on a single collection of data objects and a single variable it is called *individual*. *Set constraints* are those that are based on more than one data collections or on more than one variables. For example, in the distributed database shown in Figure 2, individual integrity constraints may include the primary key constraints on the tables *R1* and *R2*, whiles a set integrity constraint may include a referential foreign key constraint between the two tables. *Aggregate constraints* are integrity constraints involving aggregate operators such as min, max, sum, count and average. The cash present at every supermarket, at any time instance, should be less than $10,000 is an example of a static individual aggregate data constraint on the distributed database of a chain-store supermarket.

Note, that integrity constraints can be both dynamic and static. In particular a *dynamic integrity constraint* may state that if an update, which violates a constraint, is performed, a compensating update which restores the constraint should be performed within two minutes. On the other hand, static integrity constraints require the integrity constraint to be true at any given time instance.

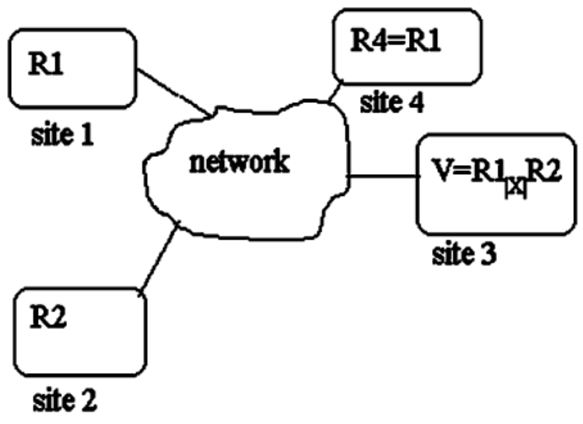

Fig.2: A Distributed Database Example

On a orthogonal plain, a *duplication constraint* specifies at what sites *replicated* or *derived* data is to be stored. We will refer to the first type of duplication constraints as *replica constraints* and to the second type as *view constraints*. In the example of Figure 2, the constraint that the table R_1 is to be replicated at *site 4* as R_4 is a replica constraint, while the constraint that *site 3* contains a table with derived data from tables R_1 and R_2 is a view constraint. From now on, we will refer to the stored query result over existing data sets as a *materialized view*. In the example from Figure 2, V is a materialized view because it stores the result of the query which performs a join between the tables R_1 and R_2. We will refer to the tables R_1 and R_2 as the *underlying tables* of the materialized view V.

Note that replica constraints are a special kind of view constraints. This is the case because a replica constraint can be expressed as a view constraint in which the query producing the view is the identity query. As well, note that duplication constraints can be both dynamic and static. In the example of Figure 2, a dynamic constraint may state that if the table $R1$ is updated, the view V has to be updated accordingly within five minutes to reflect the changes.

There is another type of constraints related to distributed databases and databases in general - *update constraints*. Those constraints specify what kind of updates are allowed on the data. More specifically an update constraint consists of a triple (P+,P-,P+-) associated with each collection of data items, where each of the Ps is a predicate formula with at most one free variable, specifying what type of add, delete and modify operations respectively are allowed on the collection.

2. MATERIALIZED VIEW MAINTENANCE

A materialized view is the stored result of a retrieve query. As the data, on which the view is based (the so called underlying data) changes, so should the materialized view to reflect the changes. Materialized view maintenance algorithms do exactly that, i.e. they propagating updates to the underlying data sources to the materialized view. In this section we will first classify existing materialized view maintenance algorithms and then we will describe some of the more prominent ones.

2.1. Classification of Materialized View Maintenance Algorithms

As the base data, on which materialized views are based, change so should the views themselves in order to correctly reflect the new state of the database. The update of a view is called *view refresh* and the ongoing process of synchronizing the view with the underlying data is called *view maintenance*. There are two general types of view maintenance algorithms - *immediate* and *deferred*. As the names show, in the first type of algorithms a view is updated during the update to the base data, while in the second type of algorithms the view refresh is done somewhere in the future, as a result of the triggering of some event. Such event may include the activation of an alarm clock, user interaction with the system, or may be triggered by a component of the system.

Before we continue our classification of view maintenance algorithms, it would be useful to look at an example (see Figure 2). The materialized view V, stored at *site 3* is calculated as the inner join of two tables R_1 and R_2. Let's assume that at time 1, $R1$ is updated to R_1' and R_2 is updated to R_2'. Then V should change to $V' = R_1' \bowtie'' R_2' = (R_1 + \Delta R_1) \bowtie (R_2 + \Delta R_2) = R_1 \bowtie R_2 + R_1 \bowtie R_2 + \Delta R_1 \bowtie R_2 + \Delta R_1 \bowtie \Delta R_2$, where ΔR is used to denote the changes made to the table R, which may include insertions, deletions and modifications. As well, $R + \Delta R$ is used to represent the result of adding the changes of ΔR to R. If at time 1, V' is recomputed as $R1' \bowtie R2'$, without the old value of the view V to be used then we classify this algorithm as *direct view refresh* algorithm. If on the other hand, ΔV is calculated first and then V' is calculated as $V' = V + \Delta V$, then we have an *incremental view refresh*. In most cases the incremental algorithms are more efficient than the direct update algorithms, i.e. in the case of small changes to the base tables it is usually cheaper to compute ΔV and then V' as $V + \Delta V$, than to re compute V' from scratch. Continuing our classification, the incremental view maintenance algorithms can be divided into two types: the ones that use *auxiliary views* and the ones that don't. To illustrate the difference on our example, an algorithm of the second type could calculate ΔV as $R_1 \bowtie \Delta R2 + \Delta R_1 \bowtie R_2 + \Delta R_1 \bowtie R_2$ or as $\Delta R_1 \bowtie R_2' + \Delta R_2 \bowtie R_1' - \Delta R_1 \bowtie \Delta R_2$ where "–" is used to indicate difference. If we know, that certain integrity constraints hold on the data, the above expressions could be simplified. Note that, since neither the old values of the underlying tables – R_1 and R_2, nor the new values R_1' and R_2' are stored at *site 3*, they have to be fetched from *sites 1* and *2* when a refresh is performed, which could result in excessive network traffic if those relations are big. Another problem in these type of view maintenance algorithms is that when *site 1* receives a request for the value of *R1'*, it may have already changed to a new state. This problem is corrected by running an error correcting procedure. Both problems are eliminated if auxiliary views are used. An auxiliary views contains representative data from the underlying data, which is sufficient to perform a view refresh. Overhead of this type of algorithms is the extra storage spaced used for the auxiliary views and the extra processing time for refreshing the auxiliary views. The later is the case because auxiliary views are materialized views themselves and therefore also need to be refreshed. Just as in the case where auxiliary views are not used, knowledge on existing integrity constraints may improve the algorithm's performance. More specifically, when auxiliary views are used, the knowledge of integrity constraints may be used to reduce the size of those views, which will result in reduced storage and faster performance.

The classification of materialized view maintenance algorithms presented so in fig.

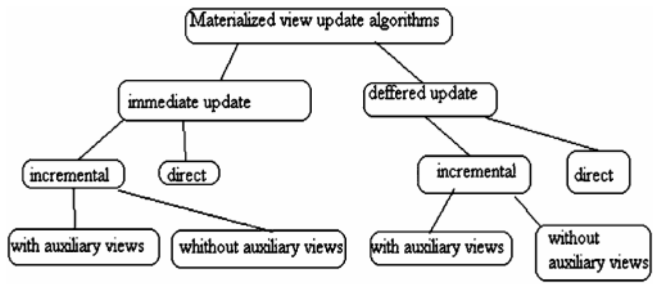

Fig. 3

3. SD-SQL SERVER ARCHITECTURE

An *SD-SQL Server database* is an SQL Server database that contains an instance of SD-SQL Server specific *manager* component. A node may carry several SD-SQL Server databases. We call an SD-SQL Server database in short *node database* (NDB). NDBs at different nodes may share a (proper) database name. Such nodes form an SD-SQL Server *scalable (distributed) database* (SDB). The common name is the *SDB name*. One of NDBs in an SDB is *primary*. It carries the meta-data registering the current NDBs, their nodes at least. SD-SQL Server provides the commands for scaling up or down an SDB, by adding or dropping NDBs. For an SDB, a node without its NDB is (an SD-SQL Server) *spare* (node). A spare for an SDB may already carry an NDB of another SDB. Fig 4 shows an SDB, but does not show spares.

To illustrate the architecture, Fig 4 shows the NDBs of some SDB, on nodes $D_1 \ldots D_i + 1$. The NDB at D_1 is a client NDB that thus carries only the images and views, especially the scalable ones. This node could be the primary one, being only of type peer or client. It interfaces the applications. The NDBs on all the other nodes till D_i are servers. They carry only the segments and do not interface any applications. The nodes could be peer or server, only. Finally, the NDB at *Di*+1 is a peer, providing all the capabilities. Its node has to be a peer node. The NDBs carry a scalable table termed *T*. The table has a scalable index *I*. We suppose that *D*1 carries the primary image of *T*, locally named *T*. The image unions the segments of *T* at servers *D*2…*Di* with the primary segment at *D*2. Peer *Di*+1 carries a secondary image of *T*. That one is supposed different, including the primary segment only. Both images are outdated. Server *Di* just split indeed its segment and created a new segment of *T* on *Di*+1. It updated the meta-data on the actual partitioning of *T* at *D*2. None of the two images refers to this segment as yet. Each will be actualized only once it gets a scalable query to *T*. The split has also created the new segment of *I*.

Notice finally in the figure that segments of *T* are all named *_D1_T*. This represents the couple (creator node, table name). We discuss details of SD-SQL Server naming rules later on. Notice here only that the name provides the uniqueness with respect to different client (peer) NDBs in an SDB. These can have each a different scalable table named *T* for the local applications. Their segments named as discussed may share a server (peer) node without the name conflict.

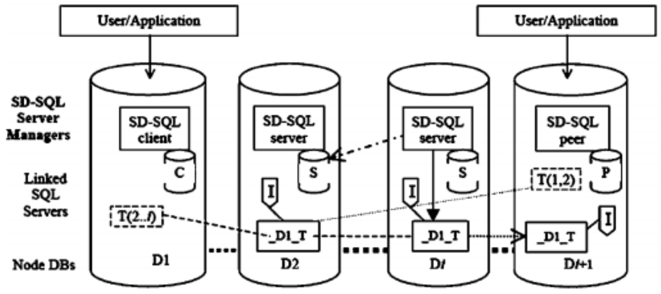

Fig. 4: SD-SQL Server Architecture

4. CONCLUSION

The approach chosen for view implementation has many advantages. Programs using views depend upon them; thus they always correctly incorporate the current semantics of any view referenced by the programs. If the views are dropped or changed, the programs will be invalidated. Views also depend on the objects referenced in the view definition statements; this allows invalidation of views when one of these objects is dropped or changed.

We have put into practice the SD-SQL Server, the "proof of concept" prototype of a new type of a DBMS, managing the scalable distributed tables. Through the scalable distributed partitioning, with respect to the current capabilities of an SQL Server database, as well as of the other known DBMSs, our system provides a much larger table, or a faster response time of a complex query, or both.

REFERENCES

[1] [Chamberlin'lS] Chamberlin,D.D., Gray,J.N. and Traiger,I.L., "Views, Authorization and Locking in a Database System", *Proc. AFIPS NCC*, **44**, 1975.

[2] [NW1 Ng, P., "Distributed Compilation and Recompilation of Database Queries", *IBM Research Laboratory RJ3375 San Jose*, Calif., January 1982.

[3] [Rothnie80] Rothnie, J.B, Bernstein, P.A., Fox, S.A, GoodmanN., Hammer, M.M., Landers, T.A..Reeve, C.L., Shipman, D.W. andWong, E., "A System for Distributed Database (SDD-1)", *ACM Transactionson Database System*, March 1980.

[4] Guinepain, S & Gruenwald, *L. Research Issues in Automatic Database Clustering*, ACM-SIGMOD, March 2005.

[5] Lejeune, *H. Technical Comparison of Oracle vs. SQL Server 2000: Focus on Performance*, December 2003.

[6] Litwin, W., Neimat, M.-A., Schneider, *D. LH*: A Scalable Distributed Data Structure*, ACM-TODS, Dec. 1996.