



Lawrence Berkeley Laboratory

UNIVERSITY OF CALIFORNIA, BERKELEY

Information and Computing Sciences Division

RECEIVED
LAWRENCE
BERKELEY LABORATORY

OCT 19 1987

LIBRARY AND
DOCUMENTS SECTION

**Managing Distributed Derived Data:
A Preliminary Proposal**

M.C. Murphy

August 1987

For Reference

Not to be taken from this room



LBL-23861
c. 1

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

Managing Distributed Derived Data: A Preliminary Proposal

Marguerite C. Murphy† ‡

**†Computer Science Research Department
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720**

**‡Computer Science Department
San Francisco State University
San Francisco, CA 94132**

August, 1987

This research was supported by the Applied Mathematics Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under contract DE-AC03-76SF00098.

MANAGING DISTRIBUTED DERIVED DATA: A PRELIMINARY PROPOSAL

Marguerite C. Murphy
August 12, 1987

ABSTRACT

This paper introduces a new class of distributed data: derived data. Derived data is information which is generated from and/or describes source data and algorithms. Derived data differs from source data in that typically it can be regenerated whenever it is needed, or generated then stored for later use. Stored derived data may become outdated as a result of update activity to the underlying source data, however it may still be accurate enough for its intended application. In this preliminary proposal we describe several major classes of derived data, then develop a general framework for maintenance in a distributed environment.

1. INTRODUCTION

Distributed database management systems (DDBMS) provide users with the logical appearance of a single database for a collection of data which is physically distributed across multiple networked computers. The mechanisms that provide the illusion of a single database distinguish DDBMS from other distributed systems. While these mechanisms allow applications to be developed in a straightforward manner, they require multiple complex software layers in the DDBMS itself. Although several commercial DDBMS are currently available, there are still many difficult implementation issues that have not yet been fully resolved.

One major issue is the data distribution policy. One extreme is to have a fully replicated database. Although this policy provides the lowest response times and is relatively straightforward to implement, the storage requirements are large and it may be difficult to propagate data updates in a timely manner. The other extreme is to have the data partitioned in some way among the different computers. This policy minimizes the storage requirements and has no update difficulties, however locating required data and processing queries become more time consuming and complex. A hybrid policy is probably needed for many applications: distribute the data among the computers with partial replication. This policy provides both low response times

and low storage requirements, however locating remote data, propagating updates and processing non-local queries become more complex.

Other major implementation issues depend to some extent on the data distribution policy. These issues include: support for concurrent access (e.g. global record locking and update synchronization), provision of transactions and recovery mechanisms, definition and implementation of authorized user access, maintenance of global catalog information, optimization and processing of queries involving remote data, etc. While the primary purpose of these components of the DDBMS is to provide consistent shared access to the information stored in the DDBMS, their implementation depends to some extent on effective utilization of derived data--information which is generated from and/or describes source data and processing algorithms, and which must itself be managed by the DDBMS. In contrast to source data, however, derived data is typically managed in an ad hoc fashion with little explicit consideration given to maintaining its consistency.

2. STATEMENT OF THE PROBLEM

The problem addressed here is that of maintaining derived data in a distributed environment. Unlike source data, derived data can be regenerated from the underlying source data when it is needed, or it may be generated and stored for later use. If the derived data is stored, it may become invalid as a result of updates to the underlying source data. One solution to this difficulty is to recompute the derived data whenever any update activity occurs to the source data. This may not be the most efficient solution, however, since the derived data may be "good enough" for its intended use even if it is based on a slightly incorrect (or stale) version of the source data.

There are many diverse examples of distributed derived data. Although these various kinds of data are similar in many respects, they are typically managed in very different ways by the DDBMS.

- (1) Query execution plans in a compiled query environment are an example of derived data--the execution plans are generated based on statistics and metadata representing the

current state of the database and may be stored for later use. The query execution plans may be acceptable if a limited amount of data update activity occurs (i.e. if the statistics are still close to their true values), but will no longer be close to optimal if large amounts of update activity occur. If the metadata changes, the query execution plans will become unusable.

Many processing strategies have been proposed for distributed queries. Perhaps the simplest technique is to determine what subset of the database is needed to process the query, then materialize that subset on a single site for processing. Many semi-join based strategies use this approach. Although data can be restricted in parallel on local sites, the bulk of the processing is performed sequentially on the materialization site. More complex techniques attempt to perform as much on-site processing in parallel on local sites as possible and attempt to balance message traffic costs with on-site processing costs. In all cases, the query access plan consists of two major parts: local access plans for on-site processing, and a global plan to assemble the query result on the proper site (and possibly to oversee and coordinate the local access plans). Other complex techniques attempt to coordinate the many local sites without an explicit global access plan (i.e. all the sites are given the query and can decide what to do on their own). In this case, the global access plan is implicit in the algorithm used by each local site.

The various global access plans are another type of derived data. Depending on how query optimization is performed, the global plans may be based upon either local plans or statistics describing the local and remote source data.

- (2) Another kind of derived data is the statistics that are used during query optimization. These statistics describe the source data stored in the database and may include relation cardinality, number of unique values in each attribute, maximum and minimum attribute values, or more detailed descriptions of the distributions of attribute values. When the source data changes as a result of update activity, the statistics may become outdated, although they may still be accurate enough for use in query optimization.

If distributed queries are optimized at a site other than the one on which the data resides, the remote site must have a private copy of the statistics describing the source data. The private copy may become out of date with respect to the statistics on-site with the data, which may themselves become out of date with respect to the source data.

- (3) View materializations are a kind of derived data that is explicitly visible to the user. The view may be accurate as of some time, and if little update activity has occurred in the meanwhile, it may still be acceptable. Another kind of view is a "snapshot in time," where the view explicitly represents the data base as it was at some prior time (or under some collection of hypothetical update activity).

If a view contains data located on a remote site, there may be a much stronger motivation for materializing local copies: the costs associated with processing a query on remote data are typically much higher than those associated with processing using local data. This is due to increased access costs (network i/o is typically more expensive than disk i/o) as well as more complex query optimization and processing strategies.

- (4) Stored summary data is very similar to a materialized view containing an aggregate function and all of the same considerations apply as above.
- (5) Metadata is the collection of data maintained about information stored in the database, primarily for use by the DBMS in processing user queries. This information includes: information about relations (name, owner, date created, date last updated, attributes, etc.), information about attributes (name, type, constraints on values, etc.), information about databases (name, creator/owner, relations, etc.), as well as information about special entities provided by the DBMS (e.g. descriptions of forms, graphs, menus, applications, etc). This information is used by the DBMS software to store data and process user requests. The collection of information is sometimes called a data dictionary or system catalog. Most relational systems use an integrated data dictionary--the metadata is stored in relations maintained by the DBMS. The metadata itself is usually created by the DBMS as the user creates/modifies the entities which the metadata describes. Although

the metadata may be viewed by the user, its primary purpose is to assist the DBMS in correctly processing queries.

Local metadata is derived data which is bound very closely to the source data. In fact, much of the local metadata can be considered part of the storage mechanism itself rather than independent data maintained by the DBMS. Local metadata describing relations, attributes and databases cannot become outdated as a result of update activity-- all updates must be consistent with the metadata or the system could not process them. Any changes to the local metadata must be reflected by corresponding changes in the source data (i.e. changing the type of an attribute value from character to integer requires reformatting all of the tuples in the relation). Local metadata describing special entities may become outdated as a result of changes to the underlying data schemas. When this happens, the metadata becomes unusable.

Distributed metadata (metadata describing remote data) may or may not be closely bound to the source data, depending on how it is used. If the distributed metadata is provided solely for user convenience (i.e. for browsing through the various collections of data available on the system), then slight inaccuracies can be tolerated. If the distributed metadata is used for query processing, however, then even slight inaccuracies will make it useless. For example, if distributed metadata is used to initially parse a query into an internal representation for transmission to the local site containing the data, even a slight inaccuracy in the metadata (i.e. an incorrect attribute name or type) will make the internal representation unintelligible to the local site. In this situation, the distributed metadata must be as closely bound to the source data as local metadata.

- (6) Replicated data with a primary site update policy can be considered derived data--the primary copy is the source data and all of the copies are derived data. All updates are applied to the primary copy, then eventually propagate out to the other copies. During the propagation phase, the copies may be out of date with respect to the primary copy, depending upon how closely bound the copies are to the primary copy.

All of these different kinds of derived data may become out of date, or less accurate, as a result of update activity to the underlying source data. The degree to which out of date derived data is acceptable depends on how tightly bound the derived data is to the source data. If the derived data is loosely bound to the source data, some amount of inaccuracy in the derived data can be tolerated.

3. MOTIVATIONS AND RELATED WORK

The primary motivation for this work is to develop a new way of looking at a collection of problems traditionally treated separately using a variety of ad hoc techniques. The exercise will hopefully lead to development of a general framework for dealing with this class of data which in turn will provide developers of distributed DBMS with a tool for designing systems which handle all kinds of derived data in a similar manner. A second motivation is efficiency. Most existing strategies regenerate derived data whenever update activities occur, or continue to use stale derived data until it becomes completely unusable. The approach taken in this work is to quantify the inaccuracy of derived data and to develop techniques which only perform regeneration when the derived data becomes too inaccurate for the given application to use.

Many ad hoc techniques have been proposed for maintaining the consistency of particular classes of derived data. For example, the System R distributed database management system has a mechanism for invalidating compiled query access plans when the underlying metadata changes [Dani82a]. The POREL DBMS has a mechanism for invalidating plans either when the metadata changes or when the statistics become too inaccurate [Walt84a]. There is a substantial literature on updating replicated data using a primary site policy. See, for example [Ston79a], for an introduction to this area. A limited amount of work has been done as well on efficient techniques for updating (accurate) view materializations, with or without embedded aggregate functions [Hans87a]. Little if any work has attempted to treat all kinds of derived data in a consistent manner. The accuracy of derived data has not been previously quantified, although [Bern81a] describe a technique for controlling response times by allowing the application to choose the minimum timestamp that must be associated with the data. Use of small timestamps allows the

application to access old data without waiting for update activity to propagate to the site, and use of large timestamps forces the application to access the most current data.

4. PROPOSED SOLUTION

There are two important decisions that must be made for all classes of derived data: *when* to regenerate the derived data and *how* to perform the regeneration. If the derived data is tightly bound to the source data it can be regenerated either at the time the source data is updated, or at the time the derived data is accessed. If the derived data is not tightly bound to the source data, it does not need to be regenerated until it becomes too inaccurate to use. Some quantitative measure of inaccuracy is needed to make this technique feasible.

4.1. DEFINITIONS

Let $S(t)$ be the collection of source data managed by the DDBMS at time t , and for $i=1,n$ let $S_i(t)$ be the subset of the source data located on site i , and $s_i(t)$ a particular item of source data on site i . Let $d_i^k(t)$ be a particular item of derived data, element k , on site i , derived at time t . The superscript will be omitted when no confusion can result. The collection of derived data on site i , $d_i^k(t)$, $k = 1, \text{number of elements of derived data on site } i$, is denoted by $D_i(\vec{t})$, where \vec{t} is the collection of times corresponding to the elements of derived data. $D_i(\vec{t})$ will be denoted by D_i when the derivation times are irrelevant. $D(\vec{t})$ is the collection of derived data on all sites, and will likewise be denoted by D when the derivation times are irrelevant.

To illustrate these definitions, take a simple example of two elements of derived data located on a given site, say l . One is a count of the number of ships in the Persian Gulf and the other is a count of the number of aircraft within striking range. Assume for simplicity that the source data is stored on a single remote site, site r , and consists of two relations, SHIPS and PLANES, with one tuple for each ship or aircraft, respectively. $S(t_1)$ is equal to $S_r(t_1)$ and consists of the two relations, SHIPS and PLANES, at time t_1 . $s_r(t_1)$ is a particular item of source data on site r , that is either the SHIPS or PLANES relation. The two items of derived data, $d_l^1(t_2)$ and $d_l^2(t_3)$, are the two counts, made at times t_2 and t_3 respectively. $D(\vec{t})$ is equal to

$D_i(\vec{t})$ and denotes the collection of both counts and \vec{t} is equal to (t_2, t_3) .

Each item of derived data is a function of the source and derived data at the time of generation: $d_i(t_j) = f[S(t_j), D(\vec{t}_j)]$, where f depends on the particular type of the item of derived data. Table 1 summarizes typical f functions for the six types of derived data under consideration.

Returning to our example above, count is a type of summary data and the algorithm to compute it is to determine the number of tuples in the corresponding relation. The count of the number of ships in the Persian Gulf, say, at 11:00 am on Thursday is the number of tuples in the SHIPS relation at 11:00 am on Thursday.

4.2. INACCURACY

Derived data may become inaccurate over time as updates occur to the underlying source data. The inaccuracy is measured as the fractional change in the value of the derived data between the time it was created (t_j) and the time it is accessed (t_k):

$$\beta = \left| \frac{\Delta[d_i(t_j), d_i(t_k)]}{V[d_i(t_j)]} \right|$$

where Δ is a function which calculates the value of the difference between a single item of derived data at two distinct times, and V is a function which calculates the value of an item of derived data at a fixed time. If the value of the difference in the derived data between times t_j and t_k is

Type of Derived Data	f function
Query Plan	Algorithm which generates optimal query plan as a function of source data statistics and metadata
Statistics	1. Sampling algorithm to estimate 2. Algorithm to calculate exactly
Metadata	Local: Happens automatically Remote: Send new metadata from source data site
View Materialization	Query which defines view
Summary Data	Algorithm which computes summary data
Replicated Data	Send new copy from source data site

Table 1. Functions to Generate Derived Data

n times the value at time t_j , then $\beta = |n|$. In particular, if there is no change in the derived data, $\beta = 0$. Table 2 summarizes typical Δ and V functions for the six types of derived data.

4.3. REGENERATION TECHNIQUES

There are several ways in which an item of derived data created at time t_j can be brought up to date as of time t_k . The simplest way is to recalculate the derived data from the source and derived data present at time t_k :

$$d_i(t_k) = f [S(t_k), D(\vec{t}_k)]$$

Some kinds of derived data can be regenerated without recomputation by adjusting their current value to compensate for update activity during the period between t_j and t_k :

$$d_i(t_k) = g [d_i(t_j)]$$

where g depends on f and one or more of $S(t_k)$, $S(t_j)$, $D(\vec{t}_j)$, t_k , t_j . Note that $S(t_k)$ typically must be known in order to determine the g function required to perform the computation of

Type of Derived Data		Function
Query Plan	V	Cost of Optimal Plan
	Δ	Zero if optimal plan has not changed, otherwise difference between costs of optimal plans at times t_j and t_k
Statistics	V	Value of Statistic (plus confidence interval if estimated)
	Δ	Difference between values at times t_j and t_k
Metadata	V	1. Binary Indicator (correct/incorrect) 2. Vector of indicators, value = #on/total #
	Δ	1. Binary indicator, zero if incorrect at t_k , one if correct 2. Vector of indicators at t_k , value = #on/total #
View Materialization	V	Number of tuples in view
	Δ	Number of invalid tuples (missing + extra)
Summary Data	V	Value of summary data
	Δ	Difference between values at times t_j and t_k
Replicated Data	V	Number of tuples in copy
	Δ	Number of invalid tuples (missing + extra)

Table 2. V and Δ Functions.

$d_i(t_k)$. In some cases no g function exists. Table 3 contains typical g functions for the six types of derived data.

If the derived data is changing in a predictable fashion, its current value might be estimated based on its past behavior:

$$\hat{d}_i(t_k) = E [d_i(t_k) | d_i(t_j), t_k]$$

That is, $\hat{d}_i(t_k)$ is the expected value of d_i at time t_k given its value at time t_j . Note that typically $S(t_k)$ need not be known in order to compute $\hat{d}_i(t_k)$, however the value obtained is an estimate and may differ from the true value that would be obtained by either of the previous two techniques.

4.4. SIMPLE REGENERATION STRATEGIES

Assume initially that a single item of derived data on site n is based on remote source data stored on a single site, site m. That is, $d_n(t_j) = f[S_m(t_j)]$. In this situation, a two part regeneration strategy can be used to insure that queries do not access derived data which is too inaccurate.

First, the site containing the derived data also stores a β -value for the derived data, β_{dev} , which is the most recent value of β received from the source data site. Queries arrive at the derived data site and have a maximum β -value associated with them, β_{max} , where this is the largest value of β_{dev} which the application can tolerate. If $\beta_{max} < \beta_{dev}$, then the derived data must be regenerated before the query is processed.

Type of Derived Data	g function
Query Plan	None
Statistics	Adjust Value to Compensate for Updates
Metadata	None
View Materialization	Insert & Delete on View
Summary Data	Adjust Value to Compensate for Updates
Replicated Data	Insert & Delete on Copy

Table 3. Functions to Regenerate Derived Data

Second, the source data site maintains a β -value, β_{cur} , which is recomputed after every update transaction. If $|\beta_{dev} - \beta_{cur}|$ is greater than some threshold δ , the new value of β_{cur} is sent to the derived data site and becomes the current value of β_{dev} . This insures that the β_{dev} value will be accurate to within δ .

This strategy will minimize the number of times the derived data is regenerated, however query response times may vary greatly depending upon how up to date the derived data is when the query arrives. One way to make response times more predictable is to have another β -value, β_{reg} , associated with each element of derived data. Whenever β_{dev} (or β_{cur} depending on who initiates regeneration), increases above β_{reg} , the data is regenerated automatically. This insures that queries with $\beta_{max} < \beta_{reg}$ will always execute without regeneration.

Any of the techniques described above can be used to regenerate the derived data: the source data site can recompute the derived data based on local data and send the result to the derived data site; the source data site can determine the proper g function, based on local data, and send this *function* to the derived data site for execution; the derived data site can estimate the current value of the derived data based on local information only.

With this strategy, the problem of maintaining distributed derived data becomes one of maintaining β -values and regenerating the derived data as needed. Intuitively, savings result because maintaining β -values is less expensive than maintaining the derived data, particularly when updates result in little net change or queries can be executed against relatively inaccurate data.

4.5. COMPLEX REGENERATION STRATEGIES

Complex derived data depends on source data stored on multiple sites. There are two general ways in which complex derived data can be computed: replicate the remote source data on the derived data site and perform the computations there; or, decompose the complex derived data computations into local and global computations, perform the local computations on site with the source data and transmit the results to the derived data site for global computations. For example, the minimum of data stored on multiple sites can be computed by replicating the

data on the derived data site and taking the minimum, or it could be computed by first calculating the minimum stored on each site, transmitting these values to the final site and calculating the global minimum there.

Unfortunately, there is no simple way of computing β for all kinds of complex derived data. The individual source data sites do not have enough information to compute the total value of β for the complex derived data. The derived data site cannot compute β directly because it cannot compute Δ , since the exact value of the derived data after the derivation time is unknown (i.e. the value of $d_i(t_k)$ is not known, nor are any other values since time t_j).

For many kinds of complex derived data, however, the value of β can be computed by combining the V and Δ values for each remote site's derived data with terms representing the value and difference resulting from the global computation, V_g and Δ_g :

$$\beta^n = F\left(\Delta_g, V_g, \Delta_i (i=1, n), V_i (i=1, n)\right)$$

where β^n is the beta value of an item derived from data on n remote sites and Δ_i and V_i are computed on source data site i and transmitted to the derived data site for the final β computation. Note that the function parameters have been omitted for clarity. All Δ_x functions have the parameters $d_x(t_j)$ and $d_x(t_k)$, and all V_x functions have the parameter $d_x(t_j)$.

For many kinds of derived data β^n can be computed by taking the quotient of the total Δ to the total V :

$$\beta^n = \left| \frac{\Delta_g + \sum_{i=1}^n \Delta_i}{V_g + \sum_{i=1}^n V_i} \right|$$

For example, distributed query access plans have local and global components. The global plans oversee execution of the local plans and combine local results. Δ_g is the difference between the costs of the stored and optimal global plans (if the two plans are different) and V_g is the cost of the stored plan. The summation in the numerator computes the total difference in costs between the stored and optimal local plans (if the two are different). The summation in the

denominator computes the total cost of the stored local plans. The overall formula computes the ratio of the cost difference between the stored and optimal query processing plans to the cost of the stored plan (assuming the optimal plan distributes the query among the same sites in the same way).

A second example is replicated data and non-join view materializations. Δ_g and V_g are both zero in this example. The two summations compute the total number of invalid tuples and the total number of tuples in the stored complex derived data.

A third example is summary and statistics based on data stored on multiple remote sites. For count-like items (sum, cardinality, etc.) the second formula can be used with Δ_g and V_g equal to zero. The two summations compute the total deviation from the true value and the total stored value.

A final example is metadata. In this case Δ_g and V_g are again both equal to zero, and the two summations are equal to the total number of indicators on and the total number of indicators.

5. DISCUSSION AND FUTURE WORK

The framework developed in the previous section provides a basis for the design of update protocols for simple and complex derived data. Inaccuracy is handled in a consistent way which allows either the user or the system to specify tolerances for a given application. This permits "fine tuning" of the protocol during execution in order to maximize performance.

The most immediate goal for the future is to demonstrate the utility of this approach by designing a collection of update protocols for one or more of the derived data classes described in the first section of this paper. This will permit a more detailed evaluation of the resource requirements for implementing the protocol, as well as any savings resulting from its use.

6. ACKNOWLEDGEMENTS

I would like to thank Dr. Doron Rotem for serving as a sounding board for working out these ideas, as well as carefully reading through earlier drafts of this paper and offering many

helpful suggestions for its improvement.

7. REFERENCES

- [Bern81a] Bernstein, Philip A. and Goodman, Nathan, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys* **13**(2) pp. 185-221 (June 1981).
- [Dani82a] Daniels, Dean, Selinger, Patricia, Haas, Laura, Lindsay, Bruce, Mohan, C., Walker, Adrian, and Wilms, Paul, "An Introduction to Distributed Query Compilation in R*," in *Distributed Data Bases*, ed. H.J. Schneider, North-Holland Publishing Company (September 1982).
- [Hans87a] Hanson, E.N., "A Performance Analysis of View Materialization Strategies," *ACM-SIGMOD Proceedings*, (May 27-29, 1987).
- [Ston79a] Stonebraker, Michael, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Transactions on Software Engineering* **SE-5**(3)(May 1979).
- [Walt84a] Walter, B. and Neuhold, E.J., "POREL: A Distributed Database System," in *Tutorial: Recent Advances in Distributed Data Base Management*, ed. C. Mohan, IEEE Computer Society, Silver Spring, MD (1984).

*LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720*