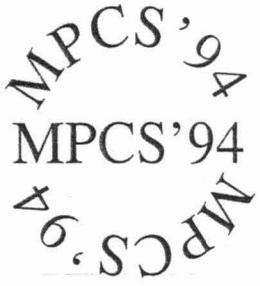


First International Conference on

Massively Parallel Computing Systems:

The Challenges of General-Purpose and Special-Purpose Computing

May 2-6, 1994
Ischia, Italy

IEEE Computer Society Press

The Institute of Electrical and Electronics Engineers, Inc.

94-LB

Load Balancing Policy in a Massively Parallel Semantic Database

Naphtali Rishe, Artyom Shaposhnikov, and Wei Sun

School of Computer Science, Florida International University
University Park, Miami, FL 33199; E-mail: rishen@fiu.edu

Abstract. We are developing a massively parallel semantic database machine. Our basic semantic storage structure insures balanced load for most parts of the database. For the other parts of the database, a load balancing algorithm is proposed herein, which allows inexpensive dynamic rebalancing without substantial negative impact on queries and transactions.

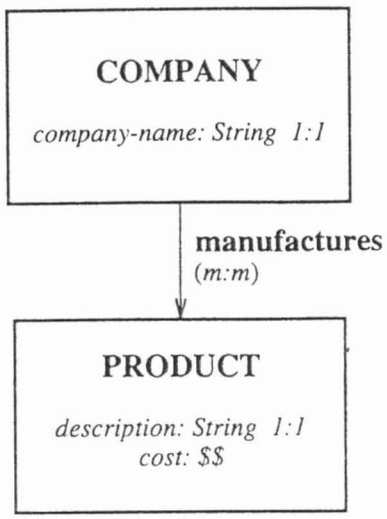
Keywords: DBMS, massive parallelism, semantic data models, load balancing, database machine.

The Semantic Binary Database Model

The semantic database models in general, and the Semantic Binary Model SBM ([Rishe-92-DDS] and others) in particular, represent the information of an application's world as a collection of elementary facts categorizing objects or establishing relationships of various kinds between pairs of objects. The central notion of semantic models is the concept of an abstract *object*, which is any real world entity that we wish to store information about in the database. The objects are categorized into classes according to their common properties. These classes, called *categories*, need not be disjoint — that is, one object may belong to several of them. Further, an arbitrary structure of subcategories and supercategories can be defined. The representation of the objects in the computer is invisible to the user, who perceives the objects as real-world entities, whether tangible, such as persons or cars, or intangible, such as observations, meetings, or desires. The database is perceived by its user as a set of facts about objects. These facts are of three types: facts stating that an object belongs to a category: xC; facts stating that there is a relationship between objects: xRy; and facts relating objects to data, such as numbers, texts, dates, images, tabulated or analytical functions, etc: xRv. The relationships can be of arbitrary kinds; for example, stating that there is a many-to-many relation *address* between the category of persons and texts means that one person may have an address, several addresses, or no address at all. For example consider a schema of a small semantic database:

- ☐ *COMPANY* — category
- ☐ *PRODUCT* — category
- ☐ *company-name* — attribute of *COMPANY*, range: *String* (*1:1*)
- ☐ *description* — attribute of *PRODUCT*, range: *String* (*1:1*)
- ☐ *cost* — attribute of *PRODUCT*, range: *$$* (*m:1*)
- ☐ *manufactures* — relation from *COMPANY* to *PRODUCT* (*m:m*)

This work is sponsored in part by grants from US DOD/BMDO & ARO, NATO, and Enterprise Florida.



Example 1. A sub-schema of a database.

Storage structure

Efficient storage structure for semantic models has been proposed in [Rishe-89-EO] and [Rishe-91-FS]. The collection of facts forming the database is represented by a file structure which ensures approximately 1 disk access to retrieve any of the following:

1. For a given abstract object x, verify/find what categories the object belongs to.
2. For a given category, find its objects.
3. For a given abstract object x and relation R, retrieve all/certain y such that xRy.
4. For a given abstract object y and relation R, retrieve all/certain abstract objects x such that xRy.
5. For a given abstract object x, retrieve (in one access) all (or several) of its direct and/or inverse relationships, *i.e.* relations R and objects y such that xRy or yRx. The relation R in xRy may be an attribute, *i.e.* a relation between abstract objects and concrete objects.
6. For a given relation (attribute) R and a given concrete object y, find all abstract objects such that xRy.
7. For a given relation (attribute) R and a given range of concrete objects $[y_1, y_2]$, find all objects x and y such that xRy and $y_1 \le y \le y_2$.

The entire database can be stored in a single file. This file contains all of the facts of the database (xC and xRy) and also additional information called inverted facts: Cx, Ryx. The inverted facts allow to keep answers to the queries 2, 4, 6, 7 in a contiguous segment of data in the database and answer them with one disk access. Whereas the direct facts xC and xRy allow to answer the queries 1, 3, and 5 with one disk access. The file is maintained as a B-tree. The variation of the B-tree used allows both sequential access according to the lexicographic order of the items comprising the facts and the inverted facts, as well as random access by arbitrary prefixes of such facts and inverted facts. The facts which are close to each other in the lexicographic order reside close in the file. (Notice, that although technically the B-tree-key is the entire fact, it is of varying length and on the average is only several bytes long, which is the average size of the encoded fact xRy.)

For example, consider the instantaneous database of Example 1. The following set of facts can be a part of a logical instantaneous database:

1. object1 COMPANY
2. object1 ***COMPANY-NAME*** 'IBM'
3. object1 ***MANUFACTURED*** object2
4. object1 ***MANUFACTURED*** object3
5. object2 ***PRODUCT***
6. object2 ***COST*** 600
7. object2 ***DESCRIPTION*** 'IBM/SYSTEM-2'
8. object3 ***PRODUCT***
9. object3 ***COST*** 100
10. object3 ***DESCRIPTION*** 'MONOCHROMATIC- MONITOR'

The additional inverted facts stored in the database are:

1. ***COMPANY*** object1
2. ***COMPANY-NAME*** 'IBM' object1

3. object2 ***MANUFACTURED–BY*** object1

4. object3 ***MANUFACTURED–BY*** object1

5. ***COST*** 100 object2

6. ***COST*** 600 object3

7. ***DESCRIPTION*** 'IBM/SYSTEM-2' object2

8. ***DESCRIPTION*** 'MONOCHROMATIC-MONITOR' object3

9. ***PRODUCT*** object2

10. ***PRODUCT*** object3

To answer the elementary query "Find all objects manufactured by object1", we find all the facts whose prefix is object1_***MANUFACTURED***. ('_' denotes concatenation.) These entries are clustered together in the sorted order of direct facts.

To answer the elementary query "Find all products costing between $0 and $800", we find all the facts whose prefix is in the range from ***COST***_0 to ***COST***_800. These entries are clustered together in the sorted order of inverted facts.

In a massively parallel version that we are implementing, the B-tree is partitioned into many small fragments, each residing on a separate storage unit (*e.g.*, a disk or non-volatile memory) associated with a fairly powerful processor. This disk-processor pair is called a *node*. Each node can retrieve the information from the disk, perform the necessary processing on the data and deliver the result to the user, or to the other processors. Similarly for updates: the node verifies all of the relevant integrity constraints and then stores the updated information on the disk. Many database fragments can be queried or updated concurrently.

The queries and transactions will enter into the network through host interfaces. Every host interface would have a copy of the Partitioning Map (PM) of the entire database. Since the whole database is a lexicographically ordered file represented by a set of B-trees, the map needs to contain for each node only a small number of facts: lexicographically minimal and maximal facts for each B-tree fragment that is stored on that node. The map changes only when the database is re-partitioned. The distribution policy that we propose in this work ensures that repartitioning is rare, inexpensive, localizable, does not interfere with the normal operation of the system, and invisible to the system until all of the shifting of data is complete.

Most of the physical facts that are in our implementation of a semantic binary database start with an abstract object. These facts are ordered by the encoding of the abstract objects, which assigns a unique quasi-random number to each abstract object. Since there are so many of these facts, and since the objects are randomly ordered, we can assume that traffic to each partition of these facts will be balanced over time. Other facts in a semantic binary database start with an inverted category, or an inverted attribute (i.e. a relation between an abstract object and a printable value). It is possible that at some time there may be a need to access a certain attribute or category more often than other attributes or categories. The same may be true for a specific range of values of a given attribute. Since all facts that refer to a particular inverted attribute or inverted category are clustered together, this may cause a higher load on some processor/disk pairs than on others. Since load imbalances can occur in some kinds of facts but not others, the file containing the facts will be split into two subfiles. The first subfile will contain all the facts that begin with an abstract object. The second subfile will contain the facts that begin with an inverted attribute or category. Additionally there is a third subfile containing long data items: texts, images, etc., which are pointed to from facts. Each subfile will be initially partitioned evenly over all the processor/disk pairs in the system. The first subfile is already balanced; the second and third subfiles require a block placement algorithm.

Request Execution Scheme

Each host in the system will have a copy of the Partitioning Map (PM). The Partitioning Map is a small semantic database containing information about data distribution in the system. Figure 1 describes a semantic schema of the partitioning map.

The partitioning map contains a set of ranges and their lexicographical bounds — the *low–bound* and the *hi–bound* values. When a

query or transaction arrives, the host will identify its lexicographical bounds; then the host will use the partitioning map to determine a set of ranges that needs to be retrieved or updated.

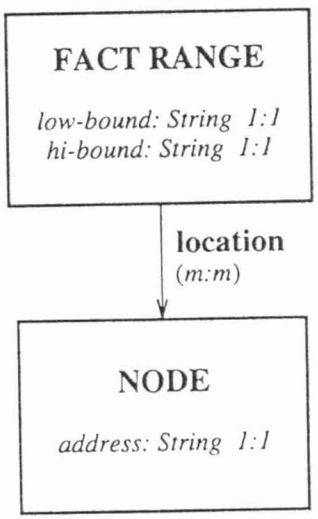


Figure 1. A Partitioning map

The partitioning map will be replicated among hosts. However, this does not imply that we need a global data structure; the algorithm described below allows to perform updating of the partitioning database gradually, without locking and interrupting all hosts.

A range can be obtained from the node pointed by the *location* reference in the partitioning database. This node should have either the range or a reference to another node.

To perform the load balancing we will need to move ranges from one node to another. A moved range will be accessible via an indirect reference that is left on its previous location. Such an indirect access slows down the system, especially when the range is frequently accessed by users. To allow a direct access to the moved range we need to update the *location* reference in the partitioning database. We will not perform such an update at once for all the host interfaces. The update will be performed when a host executes the first query or transaction that refers to the range that was transferred. The node that actually holds the range will send to the host the results and a request to update the partitioning map. This means that the first transaction will have to travel a little longer than all subsequent transactions. The second and other queries or transactions will be executed directly by the node pointed by the *location* reference.

A data structure at each node to support indirect referencing will be exactly the same as the partitioning map described above. We will call this data structure a local partitioning map.

Each range of facts will be represented as a separate B-tree structure that will reside on the node pointed to by the partitioning map. Consider now a case when a range has been moved several times from one node to another. We will have a multiple indirection in references to the actual range location. To reduce the number of possible indirect references in the system each range will have a "native" node. A native node will always contain a reference to the actual range location. When the range is transferred from a source node to a destination node, the native node will be notified and the source node will include a reference to the native node in its local partitioning database.

Load Balancing Policy

In the idle time, the host interfaces will send the recently accumulated data and work load statistics from nodes to a Global Performance Analyzer (GPA). The GPA is a computer that analyzes the statistic information obtained and generates preferable directions of data transfer for each node.

The statistics report will contain only the changes since the previous report:

- Changes in data partitioning
- Number of accesses for each range
- Free space on each node

The GPA will use a heuristic search algorithm that uses a choice function to select a small number of possible data movements in the system. The final state will be estimated by a static evaluation function S. The GPA will select the data movement with the lowest value of the resulting static evaluation S.

The choice function should comply with the following strategies:

1. Whenever possible the load balancing should be achieved by joining ranges together. Joining will result in faster query execution and smaller partitioning maps.

2. A criteria for preferable direction of a range transfer is a lexicographic proximity to the lexicographically closest range on the destination node.

3. If a range is exceptionally highly loaded by accesses or it is an exceptionally big range — split the range into several parts and transfer them to other nodes.

Each node will be characterized by two parameters:

1. The amount of free disk space on the node, F

2. The percentage of idle time I. In other words the I is:
$$I = \frac{Idle}{T}, \text{ where}$$
T is a given time interval and $Idle$ is the node's idle time during the time T.

The resulting state will be estimated by the following parameters:

1. A - the total amount of data that will be necessary to transfer in the system

2. D_F — the mean square deviation of F

3. D_I — the mean square deviation of I

4. M — total number of ranges in the system

The static evaluation function can be represented as:

$$S = C_1 * A + C_2 * D_F + C_3 * D_I + C_4 * M,$$

where C_1, C_2, C_3, and C_4 are constants.

References

[Rishe-89-EO] N. Rishe. "Efficient Organization of Semantic Databases" *Foundations of Data Organization and Algorithms*. (FODO-89) W. Litwin and H.-J. Schek, eds. *Springer-Verlag Lecture Notes in Computer Science*, vol. 367, pp. 114-127, 1989.

[Rishe-91-FS] N. Rishe. "A File Structure for Semantic Databases." *Information Systems*, 16, *4* (1991), pp. 375-385.

[Rishe-92-DDS] N. Rishe. *Database Design: The Semantic Modeling Approach*. McGraw-Hill, 1992, 528 pp.