

分散型データベースにおける同期制御のための 階層型プロトコル

山崎晴明, 足田定幸, 吉田勇, 川上英, 松下温
(沖電気工業株式会社, 研究所)

1. はじめに

近年の通信ネットワーク技術の発展は, コンピュータ処理の様々な分野に影響を与えており, データベース処理の分野においても, 単一の大規模なデータベースと比べて, 多くの利点を持つ分散データベースというアトラクティブなアプローチを可能とするに至っている。

CCA (Computer Corporation of America) 社の J. Rothnie 等は, このような分散データベースシステムの持つ利点として, 1) 高信頼性, 2) 通信コストの節減, 3) 拡張の容易性等を指摘しているが[1], 分散に伴なうスケジューリング方式, 共有資源の管理形態, データの最適配置等考慮すべき技術的問題も数多く残されている。さらに, データベースの *consistency* をいかにして維持していくかということ, は, 分散データベースシステムにおける主要な技術的問題のひとつとなっている。このようなデータの *consistency* には, 2つの意味がある。ひとつは, 冗長データが同一値を持つというシンタクティックな *consistency* であり, 他のひとつはデータ間に意味的な矛盾がないというセマンティックな *consistency* である。前者についてはこれまでに数多くの論文が発表されているが, 後者についての報告は少ない[2][3][4][5][7][8]。

本稿では, これら2種の *consistency* を保証するための階層型処理構造を提案する。オ2節では, 本稿で使用される用語を定義し, オ3節では階層型処理の概要を述べる。オ4節では, 階層型処理を行うためのアルゴリズムを示し, あわせて障害の検出および回復方式について簡単に述べる。

2. *Consistency* 維持のための同期制御方式

本稿では, データベースモデルとしてリレーショナルモデルを想定する。ひとつのリレーションはフラグメントの集合から成り, ひとつのフラグメントは *tuple* の集合より成るものとする。また本提案のデータベースシステムでは, リレーションは, フラグメント単位でネットワーク中に分散されるものとする。すなわち, あるフラグメントがどのデータベースサイトに配置されているか, またひとつのデータベースサイトがどのフラグメントを保持しているかは, ネットワーク内の全データベースサイトが保持する *Directory* により管理されることになる。ここでデータベースサイトとは分散データベースネットワークのノードとして定義され, ローカルなデータベースマネジメントシステム自身を意味する。したがって, 分散データベースシステムは, 通信路によって結合されたデータベースサイトの集合ということになる。またあるフラグメントは, いくつかのサイトで冗長に保持される。フラグメントの *up-date* は, データベースマネジメントシステムにより生成されるフラグメントプロセスと呼ばれるプロセスによって実行される。このフラグメントプロセスは, 対- (フラグメント *identifier*, *site identifier*) によって識別

される。同一のフラグメントで異なるサイトに冗長に格納されたものに対するフラグメントプロセスの集合を“closed Update Group”と呼ぶ。

一般に、update を実行する query は、トランザクションと呼ばれる各 closed update group 毎の query に分解される。このとき、closed update group 内の、冗長に格納されたフラグメントはすべて同一の値を持たなければならない（シンタクティックな Consistency）し、また各 closed update group 内の意味的な関係も正しく維持されなければならない（セマンティックな Consistency）。

この2つの Consistency の例を図2.1に示す。

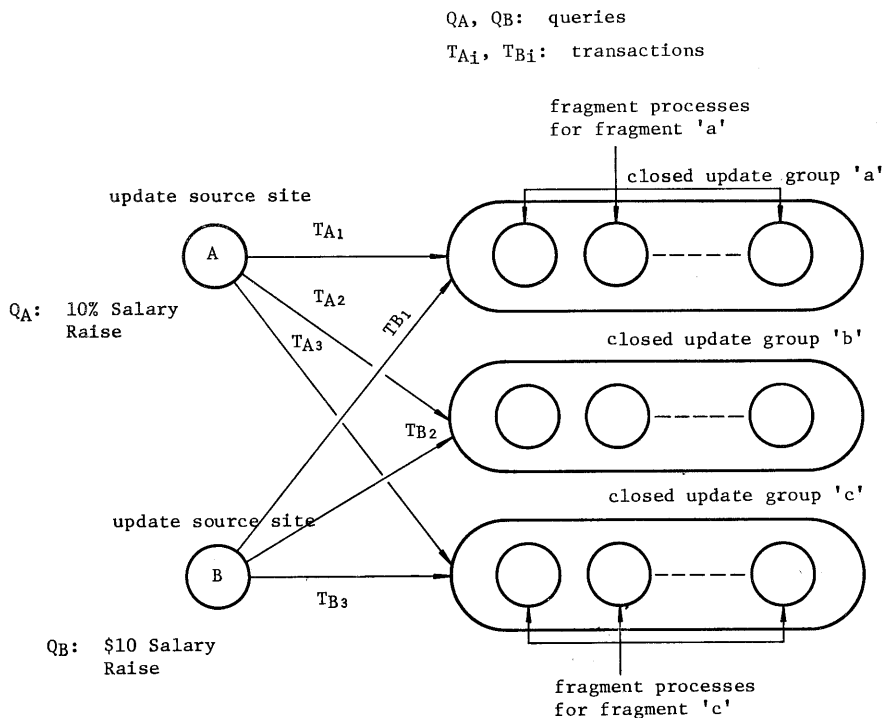


Fig. 2.1 Two types of consistency

ここで、employee リレーションは3つのフラグメントから成り、各々のフラグメントは、冗長に配置されているものとする。したがってこれらのフラグメントに対するフラグメントプロセスは3つの closed update group を形成することになる（図2.1参照）。今 source site B から従業員の給料を10ドル上げる query が入力され、ほとんど同時に Source site A から給料を10%上げる query が入力されたものとする。このとき、サイトAから出されたトランザクションが closed update group a および b で処理され、またサイトBから出されたトランザクションが group c で処理されたものとする。したがって、サイトAより group C

へのトランザクションおよびサイトBより group a, b へのトランザクションは前の処理の終了後実行されることになる。group a, b と group c との間でデータの意味的な矛盾が生ずることになる。すなわち、group aおよびbに属する従業員の給料は10%上がった後10ドル加えられたのに対し、group c に属する従業員の給料は10ドル加えられた後10%上げられたことになるからである。したがって、この場合には、セマンティックな consistency は保証されないが、各 closed update group 内のフラグメントはすべて同一値を持つという意味でシンタクティックな consistency は保証されたことになる。

分散データベースにおいては、これら2種の consistency は、常に維持されなければならない。SDD-1 [3][4] では、特にセマンティックな consistency を維持するためのプロトコルを数種提案している。そのうち、プロトコル4と呼ばれるものはネットワーク中の全サイトを一度に lock するものである。この locking メカニズムでは、update するため制御をすべて source サイトに集中することになり、パフォーマンスのボトルネックを生ずる可能性がある。INGRES [5] では、データベース update のため、2つのプロトコルを提案している。ひとつは、シンタクティックな consistency を保証するものであり、他のひとつはシンタクティック、セマンティック両方の consistency を保証するものである。後者のプロトコルも、SDD-1 と同様、パフォーマンスのボトルネックを生ずる可能性がある。

3. 階層型処理構造

本節では、第2節で述べたパフォーマンスのボトルネックを解消するための、階層型の処理構造を示す。一般的に、図3.1に示すように update のための query は2以上の closed update group における処理を必要とする。ここで、ユーザからの update 要求を受け付けるプロセスを“source”と呼ぶ。一方、closed update group 内のフラグメントプロセスのうち source からのトランザクションを受け付けるプロセスを“master”と呼び、その他の closed update group 内のフラグメントプロセスを“slave”と呼ぶ。さらに、source と master とによって形成されるプロセスの集合を“related update group”と呼ぶ。

ユーザから出された query は、まず source によって受け付けられ、source はそれをトランザクションに分解した後、master に送出する。次に、master により、closed update group 内のフラグメントの update が実行される。したがって、source と master 間で処理負荷が分散されることになる、パフォーマンスボトルネックの解消が計られることになる。

source と master および master と slave 間で行なわれるインタラクションは第4節で述べるように 2-phase commit 法に基づいたものとなっている。

図3.2は、セマンティックな consistency を維持するために related update group を形成した例を示したものである。ここで、employee リレーション R は、フラグメント F_i ($i=1, 2, 3$) に分割され、各々のフラグメントは master i により処理されるものとする。また、ユーザから source に与えられた query Q は、全従業員のサラリー (SAL) を10%上げるもので、次のような形式をしているものとする。

$$Q = \begin{bmatrix} \text{update } R \\ R. \text{ SAL} := R. \text{ SAL} * 1.1 \end{bmatrix}$$

Source は, Q をトランザクション (T_1, T_2, T_3) に分割し, 単一の related update group を形成する。これにより, セマンティックな consistency が保証されることとなる。

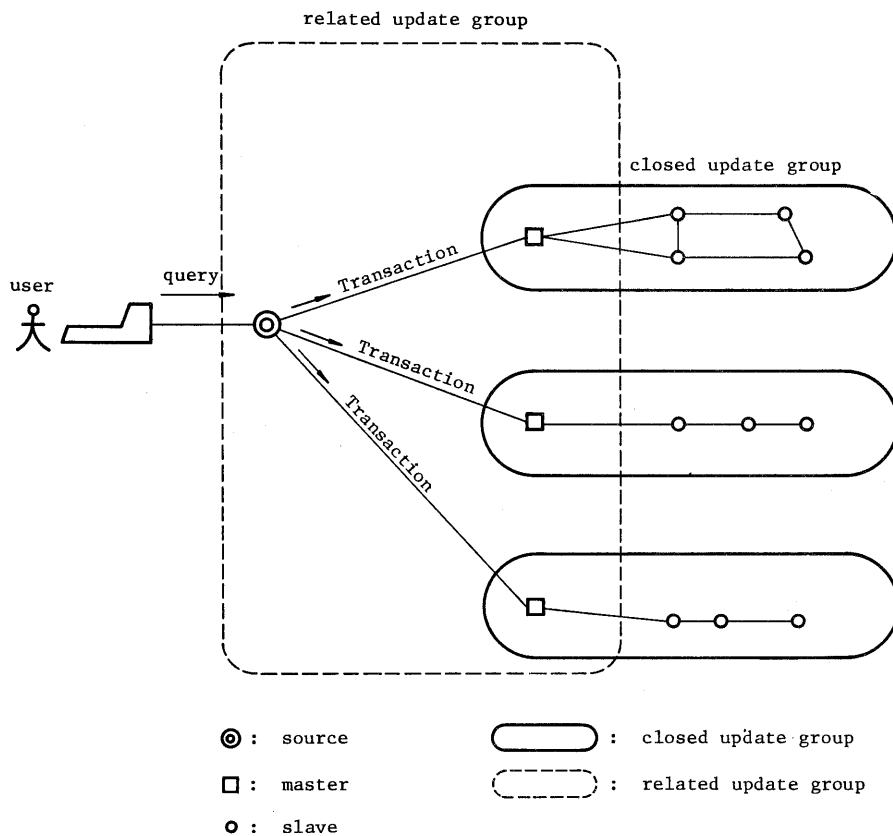


Fig. 3.1 Relationship between a closed update group and a related update group

$$Q = \begin{bmatrix} \text{update R} \\ \text{R. SAL:} = \dot{\text{R. SAL}} * 1.1 \end{bmatrix}$$

$$T_1 = \begin{bmatrix} \text{update F}_1 \\ \text{F}_1. \text{ SAL:} = \text{F}_1. \text{ SAL} * 1.1 \end{bmatrix}$$

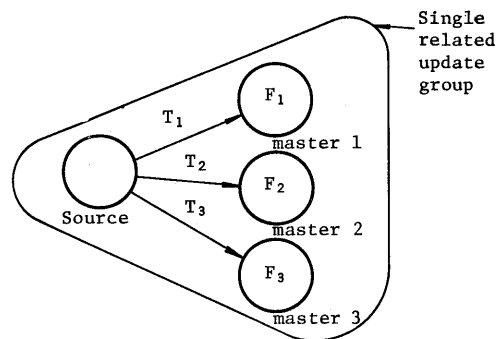


Fig.3.2 Formation of a single related update group

一方、図3.3は、各トランザクション毎に独立な related update group を形成した例であり、この場合、セマンティックな consistency は保証されないことになる。

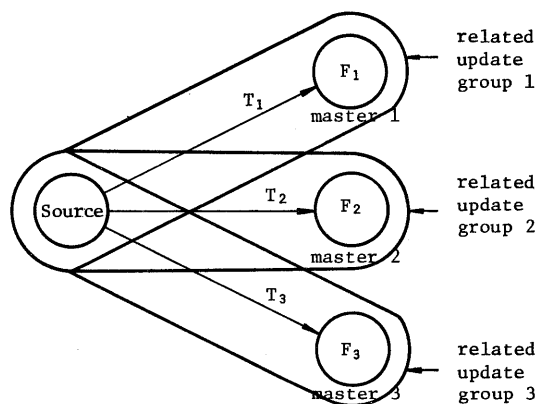


Fig. 3.3 Formation of multiple related update groups

図3.4は、2つの related update group 間で競合が生じた場合を示したものである。ここで競合とは、ひとつの master が複数の source から複数のトランザクションを同時に受け取る状態をいう。図3.4においては、master 1, 2, 3 は、source A, Bによって、競合状態におかれている。

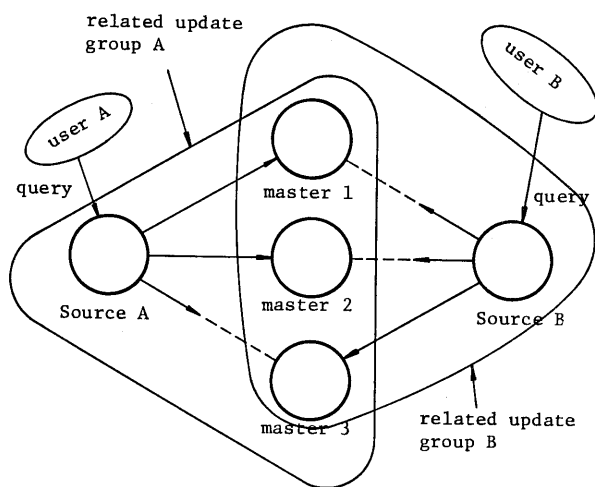


Fig. 3.4 The conflict between two related update groups

また、ユーザから source に到着した query には、priority が付加され、source から master へトランザクションが送出される時には、その priority も一緒に送られることになる。このときの priority は、query が source へ到着したときの time stamp と、source の identifier より成る。

競合状態に置かれた master は、いづれか一方の source に、競合状態が生起していることを通知するとともに、競合状態を引き起しているもう一方の source からの query の priority を報告する。報告を受けた source は、自身の priority と、報告された priority とを比較し、自身の priority の方が低いのであれば、related update group の形成を中断する。したがって、高い priority を持つ source が、related update group を形成することになり、競合状態は解除される(4節参照)。

通常、シンクティップおよびセマンティック両方の consistency を保証するためには、単一の related update group を形成することが必要となる。しかしながら、query 処理の並列度を増すためには、別個の related update group を形成した方がよいことは明らかである。したがって、あらかじめセマンティックな consistency を保つことがわかっているような query に対しては、複数の related update group を別個に形成する方式が望ましいものとなる。

次に、related update group での処理を実行したあと、closed update group 内のフラグメントの update が行なわれる。これら2つの update group は、

ともに *commitment group* と呼ばれるものの一種であり[6], 次節で述べるように, 階層型処理構造により実行される。さらに, このような *commitment group* は, 与えられた *query* に応じて, 幾段階にもネストされる。

なお, *closed update group* 内のどのフラグメントプロセスも *master* となることができる。この *source* による *master* の決定は, *source* 対応に定められた *master* 選択リストにより行なわれる。このリストは, 各サイトの *identifier* と, サイトとフラグメントとの対応テーブルより成り, サイト *identifier* のならべられた順序により *master* 選択が行なわれることになる。

次節では, *source* と *master* 間, *master* と *slave* 間で実行される階層型処理について詳述する。

4. 階層型処理構造

階層型処理のシーケンスを図4.1に示す。 *source* と *master* 間で行なわれるサブシーケンスは *related update control* と呼ばれ, *master* と *slave* 間で行なわれるサブシーケンスは *closed update control* と呼ばれる。

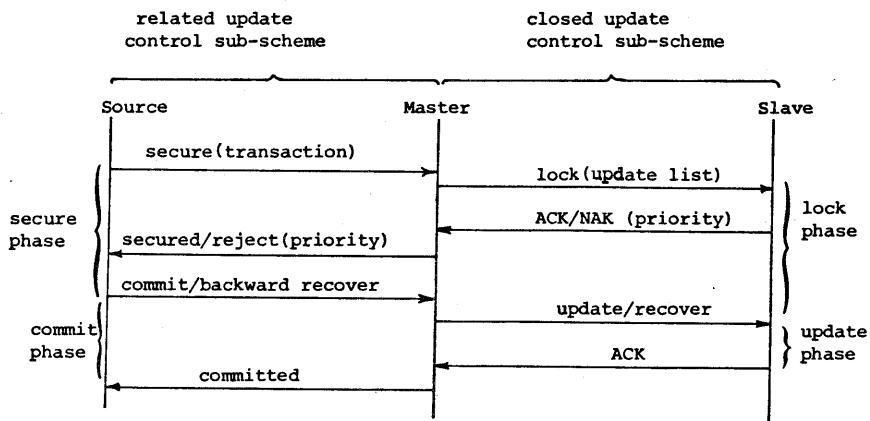


Fig. 4.1 Sequence of the hierarchical processing scheme

related update control の2つのフェーズは, それぞれセキアフェーズ, コミットフェーズと呼ばれる。一方 *closed update control* における2つのフェーズは, それぞれロックフェーズ, アップデートフェーズと呼ばれる。

Source の動作

① *source* は *related update group* 内の *master* に *secure* メッセージを送出することにより, トランザクションを割り当てる。

(2) source は, related update group内のすべての master から securedメッセージが返送されてきた場合に限る, 各 master に commitメッセージを送出する。

(3) もし, ひとつでも rejectメッセージを返送した master があれば, source は, 自身の priority と rejectメッセージのパラメータとして付加されている priority とを比較する。もし後者の方が前者よりも priority が高ければ, source は, related update groupの形成を中断し, securedメッセージを返送してきた master に対し, backward recoverメッセージを送出する。逆に, 前者の方が後者よりも priority が高ければ, source は rejectを返送した master に対し secureメッセージを送出し, securedメッセージが返送されるまでくり返す。このとき, 低い priority を持っている source は related update groupの形成を中断するため, 最終的には securedメッセージが返送されることは保証される。

Master の動作

(1) secureメッセージを受け取った master はアップデートリスト(更新された tuple のリスト)を作成する。ただし, master がすでに他のトランザクションを受けてそれを処理している最中であれば, その処理中のトランザクションの priority を, rejectメッセージに付加し, source に返送する。

(2) 次に, master は, アップデートリストを lockメッセージに付加し, closed update group内の全 slave に送付する。

(3) 次に, すべての slave から ACK が返送されたとき限り, master は securedメッセージを source に返す。もし master が, ひとつでも NAKを受け取ったのであれば, それに付加されている priority と現在 master が処理している priority とが比較される。もし後者の方が前者よりも priority が高ければ, master は, 再び NAKを返送した slave に対し, lockメッセージを送出し続ける。これは, 該当 slave が ACK を返送するまで続けられる。もし priority の高低が逆であれば, master は source に対し, secureメッセージの応答として rejectメッセージを返送する。さらに master は, secured を返送してしまっている slave に対し, アップデートリストの放棄とロック状態の解除を意味する recoverメッセージを送付する。

(4) master が source から commitメッセージを受け取ると, 補助記憶装置中のフラグメントの更新が行なわれる。したがって, この後には Backward recovery が不可能な状態となる。この後, master は, 全 slave に対し updateメッセージを送出する。

(5) master は, source から backward recoverメッセージを受け取ると, master 自身のフラグメントの更新を中止すると同時に, 全 slave に対し recoverメッセージを送出して, slave によるフラグメントの更新を中止させる。なお, この場合 source への応答は返さないものとする。

(6) master は、全 slave からの ACK を受信すると、source に対して committed メッセージを返送する。これにより、master における更新処理が終了したことになる。

slave の動作

(1) slave により lock メッセージが受信されると、該当フラグメントはロックされ、アップデイトリストが一時記憶域に格納される。このとき、もし slave が他の master からの lock メッセージを処理しているのでなければ、master に対し ACK を返送する。そうでなければ、master に NAK が送られることになる。

(2) 次に、update メッセージが受信されると、slave は、該当フラグメントの更新を実際に行ない、backward recover が不可能な状態となる。その後 slave は、master に ACK を送付する。

(3) recover メッセージが受信されると、slave は該当フラグメントの更新を中止し、フラグメントのロックを解除する。この場合、master に対する応答は返さない。

もしタイムアウトやシーケンスエラーの様な異常状態が検出されると、障害処理が異常を検出したサイトで起動される。この障害処理は、各サイトにおかれた diagnostic manager によって実行されるものとする。各サイトの diagnostic manager 間には、diagnostic connection と呼ばれるシステムジェネレーション時に確立された connection によって結合されているものとする。

これまで提案された多くの分散データベースシステムでは、この障害処理のためのプロトコルと consistency 維持のためのプロトコルとが明確に分離されておらず、また connection も同一のものを使用していたため、次のような欠点を持っていた。

- 1) アルゴリズムが複雑。
- 2) 緊急メッセージや高プライオリティのメッセージの扱いが容易でない。
- 3) 障害時には、無駄な通信が発生し易い。

これらの欠点は、本提案の分散データベースシステムにおいては、diagnostic manager および diagnostic connection を導入して、通常処理と障害処理とを明確に分離することにより解決が図られている。また、障害時の通信量も、site 毎に置かれた diagnostic manager 間通信となるため、fragment process 間通信のときと比べて大巾に減じられたものとなっている。なお障害処理の概要は次のようである。

fragment process から障害生起の通知を受けた diagnostic manager は、障害が生起したと思われるサイトの diagnostic manager との通信を試みネットワークのパーティショニングが生じたか否かを判定する。もしパーティショニングであれば manager は自サイトがマジョリティグループかマイノリティグループに属するかの判定を行なわ

なければならぬ。そのため manager は、診断メッセージに、障害の生じたサイトの識別子を付け、全サイトに送出する。この診断メッセージを受け取った各サイトの manager は、メッセージ中の識別子を自身が管理するエラーサイトリストに登録後、応答を返す。応答を受信した manager は、その応答の個数により、自身が、マジョリティかマイノリティかを判断する。もしマジョリティであれば、応答を返してこなかったサイトはすべてエラーサイトリストに登録され、再度診断メッセージを送出する。このステップは、正常に動作しているサイトがすべて応答を返すようになるまでくり返される。もしマイノリティであれば、応答を返送してきたすべてのサイトに we are down メッセージを送出する。こうすることによ、て、マイノリティに属するすべてのフラグメントはブロックされ、パーティショニングが解除されたとき、マジョリティによ、て処理されたトランザクションをマイノリティでも処理する。このようにして全体での consistency が維持されることになるが、アルゴリズムの詳細については、本稿では紙面の都合上、省略する。

5. 結論

本稿では、分散データベースにおける consistency 維持のための同期制御に、階層処理構造を用いる方式を提案した。この方式は、処理負荷の分散化、アルゴリズムの明確化と理解のし易さ等の特長を持っている。今後は、この方式のパフォーマンスを定量的に評価し、さらに改良を加えてゆく予定である。さらに、ネットワークアーキテクチャにおける分散データベースシステムの位置、特に障害処理とネットワークにおけるマネジメント機能との関連等が今後の研究課題として残っている。

参考文献

- [1] J.B. Rothnie, N. Goodman "A survey of Research and Development in Distributed Management" Proc. Int. Conf. VLDB, 1977, PP. 48-62
- [2] R.H. Thomas, "A solution to the concurrency control problem for multiple copy database", Compcon 78, 1978, PP. 56-62
- [3] P.A. Bernstein, J.B. Rothnie, N. Goodman, C.A. Papadimitriou, "The concurrency control mechanism of SDD-1: A system for distributed Database (The fully redundant case)", IEEE, Trans. Software, 1978, vol. SE-4, no. 3, PP. 154-168.
- [4] P.A. Bernstein, D.W. Shipman, J.B. Rothnie, N. Goodman, "The concurrency control mechanism of SDD-1: A system for distributed database (The general case)" Technical report, CCA-77-09, 1977.
- [5] M. Stonebraker, "Concurrency control and consistency of Multiple copies of Data in Distributed INGRES", Proc. of the THIRD BERKELEY WORKSHOP, 1978, P.P. 235-258.
- [6] J.B. Brenner, "Preliminary View on Administration and Use of Logically Related Sessions", Contribution to ISO/TC97/SC16/WG2 N60.
- [7] C.A. Ellis "A Robust Algorithm for Updating Duplicate Databases", proc of the SECOND BERKELEY WORKSHOP, 1977, PP. 146-158.
- [8] C.H. Lee, R. Shastri, "Distributed Control schemes for Multiple-Copied File Access in a Network Environment" Compsac 77, 1977, P.P. 722-728.
- [9] Gray J. et al., "Granularity of Locks and Degrees of Consistency in a shared Data Base", IBM Research, San Jose, Ca., RJ1849, July, 1976.
- [10] H. Yamazaki, S. Hikita, I. Yoshida, S. Kawakami, Y. Matsushita, "A Hierarchical Structure for Concurrency Control in a Distributed Database System", Sixth Data Communications Symposium-1979, Pacific Grove, California, November 27-29, 1979 (To be published).