Gaming-Simulations of Multi-Agent Information Systems Using Large Databases: The Concept and Database Algorithms

OHMORI,Tadashi HOSHI,Mamoru
Graduate School of Information Systems, The University of Electro-Communications.
E-mail: {omori,hoshi}@is.uec.ac.jp JAPAN

Abstract

Recently much attention has been paid to various usage of large history-databases, such as discoverying knowledge from log-databases. In this paper, we propose another new applications of history-databases, called a 'game-style' (*or* gaming-) simulation of multi-agent information systems; here, a 'multi-agent system' refers to the system composed of multiple agents who work on a shared and large database. This paper discusses the simulations of such multi-agent systems on a large database, where this database is chosen from history-data.

To explain our ideas, consider a trading-market system in which multiple supplier-agents work on a shared and large 'purchase-order' database; also, assume that these suppliers compete for occupying better purchase-orders there. Then we consider this competition to be a game, and simulate the game on a past 'purchase-order' database of large volume.

This paper discusses the above 'game-style' simulations of multiple agents on a shared and large database D; of course, such D must be chosen from a history-database so that it exactly imitates the simulated situations. (e.g. consider using the '1994-April order'-database to imitate the '1995-April' situation.) We discuss how to execute this simulation-problem on a centralized parallel database-system which holds all of history-data.

Our study is useful to test/verify agents' strategies with much more reality by using a real (and often very large) history-database. To clarify the concepts of our study, this paper describes examples at first. Next we formalize them as a new multi-agent problem using a shared database. After that, we propose new efficient database-algorithms for this formal problem.

1 Introduction

In recent parallel database applications, how to utilize huge history-data, such as sales-log data of terabyte-scale, has been a hot issue [4,7]. Recent popular ways are data mining, operations research, and statistic analysis [7,14]. These are used for finding useful 'knowledge' from log-data and for building new assumptions and/or new behavioral strategies about target information-systems. However, after finding such knowledge, how can we test the effects of such new assumptions or strategies in the target systems? One solution is, clearly, a simulation of the target systems under given models.

From the above motivation, this paper proposes new applications of history-databases, called a 'game-style' (*or* gaming-) simulation of multi-agent information systems; here, a 'multi-agent' information system refers to the system composed of multiple agents who work on a shared (and often very large) database. This paper discusses the simulations of such systems on a large database, where this database is chosen from history-data.

Fig.1 describes the concept of our simulation-problem: As shown in this figure, today's many information systems are composed of multiple agents who work on a shared database, D, of large volume [5,16]. (e.g. consider a market-trading system on a '1-month purchase-order' database, which has millions of advance-orders for a month.) In general, all of such D's are registered in history-data; also, the actions of agents on each of such D's are registered in history-data. Therefore, it is possible to simulate the behaviors of the agents on a given past database D_0; of course, D_0 must be chosen so that it exactly imitate the situations to be simulated. Also, as shown in Fig.1, it is general that all of the history-data are stored in a centralized database system. Thus we discuss how to compute the above simulation-problem on a today's parallel database system [4,8], assuming that the database system holds all of the history-data.

To explain our simulation-problem by an example, consider a trading-system in a market, where multiple suppliers (= the agents in Fig.1) work on a 'purchase-order' database. Also let us assume the following: (i) These suppliers compete with one another, and try to make better service-plans from this 'order'-database. That is, each supplier searches for better purchase-orders, and tries to acquire them exclusively. (For each supplier, his *service-plan* or *market-share* refers to the

Proceedings of the Fourth International Conference on
Database Systems for Advanced Applications (DASFAA'95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995

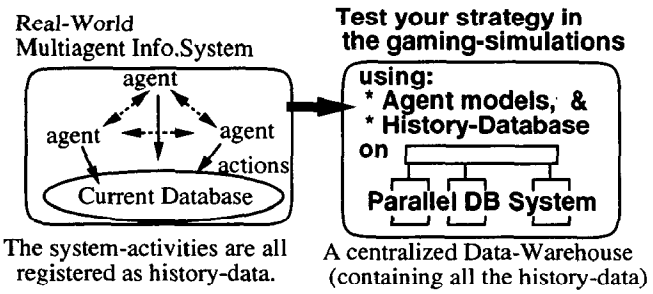

Figure 1: Gaming-simulation on database

set of those purchase-orders that he has exclusively acquired.) (ii) In order to get a better market-share, each supplier can change his behavioral strategy during the above competition. Also, he can utilize some rules of taking-over. (i.e. Under some conditions, a supplier can take away the market-share of another supplier.) Then, on these two assumptions, the simulation of this system is a 'game' among the suppliers; i.e. this is the game that each supplier tries to acquire a better market-share than the others. □

Now, in the above example, what behavioral strategy is better for a supplier? When/how should he change his strategy? To answer these questions, this paper proposes the simulations using a history-database (described in Fig.1); that is, for the above example, we give the (assumed) behavioral models of all the suppliers, and then execute the above 'game-style' (*or* gaming-) simulations on a past 'purchase-order' database. Such simulations clarify the effects of the assumed models on the basis of real (and very large) history-data. Then, from these effects, the assumed agent-models can be improved.

About related works, a gaming-simulation is originally defined as a simulation of game-style played by some human/programmed players [6]. Various gaming-simulations have been studied in business applications, such as an urban-policy simulation on a city-model or a trading simulation[6]. However those studies have used small volume of data, which are miniatured abstractions of the large databases used in the real-world [6]. Various multi-agent problems have been discussed in distributed AI[5,16], but they have not considered large database processing.

In contrast, our unique point is to use real (and large) history-databases directly in the multi-agent simulations. This point is significant because, in order to simulate 'real-world' multi-agent systems, we need to use large databases composed of history-data. In other words, there are many useful multi-agent systems that directly work on large databases. The above example (= the simulation of the market-trading system that deals with the advance-orders for a month) is such a case. As another example, consider the urban-policy simulations among some policy-making agents, and assume that these agents must decide the city-developing plans for each month; then this decision-simulation is nothing but a cooperative planning on the log-database D, where D describes all the city-activities over a 1-month period. Such D's have large volume of raw data, including monthly trading-logs, population database, etc. Also, at the early stages of these simulation-examples, we do not know any good miniatured abstractions about the used log-databases. Thus our gaming-simulations (Fig.1) need to use such large databases directly.

By using such raw history-data, our study brings two advantages: (i) We can test/improve the agents' strategies with much more reality of data. (ii) From the observed data in the simulations, we expect to discover better abstractions of the target systems; these abstractions are useful to develop better system-/agent-models, which provide a more simplified simulation of the target systems.

Because of the necessity/advantages above, our simulations directly use very large history-databases stored in secondary storages. It causes some technical problems; therefore, this paper describes the following issues: (1) First, we describe examples for the multi-agent gaming-simulations on a large database, and clarify their concepts. (2) Second, we formalize those examples as a new multi-agent problem using a database. (3) Third, we propose efficient database-algorithms to compute this formal problem on a parallel database system.

Concerning the above issue (3), our study apparently needs concurrent/iterative execution of agents. We first describe that such an agent is a transaction reading/writing large bulks of data. Thereafter we propose a dedicated concurrency-control scheduler [11] for such transactions. For the iterative operation, we propose a variation of a rule-base [2,3] algorithm.

Section 2 describes examples. Section 3 describes general concepts of our gaming-simulations, and formalizes them with database-processing models. Section 4 proposes our algorithms, followed by experiments in Section 5. Section 6 discusses the applicability of our study. Section 7 summarizes the paper.

2 Examples

2.1 Market-share competition game

To clarify the concept of our study, this section describes an example and its database-processing code. The used example is the market-trading competition, mentioned roughly in Section 1:

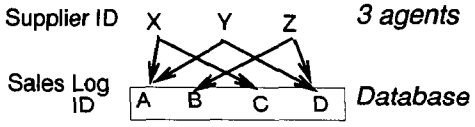

Figure 2: mapping from suppliers to log-files

Table 1: Conditions of Market-Competition game
(a): Selection Conditions of 3 suppliers on 4 areas.

Supplier	Area1,2	Filter Predicate P1 (= P2)
X	A, C	Period $\in$ Range1
Y	A, D	Customer $\in$ Range2
Z	B, D	Item $\in$ Range3

(b): Constraint Conditions
S(x)= Gain of a supplier, x, under a given plan).

1:	S(x) < limit0 for all x. and
2:	S(x) − S(y) < limit1 for all (x,y) = (X,Y), (Y,Z), (Z,X).

[EX.1]: Consider the following trading-system in a market: This market has three suppliers working on a (shared) 'purchase-order' database (denoted by DB_0). The suppliers compete with one another in order to make their service-plans (= market-shares) from this current database DB_0: i.e. this is the game that each supplier tries to get a better market-share than the others. [1] Also we assume the following: (1) The market has four areas. (To simulate this market, we use 'sales-log' files of these areas; i.e. these files are used as the current purchase-order database (= DB_0).) (2) Each supplier has a behavioral strategy. Using the strategy, he searches for better purchase-orders to be served in the market, and tries to acquire them. If this trial succeeds, he supplies those orders exclusively. (3) The game is over when there remain no purchase-orders in DB_0. (4) When the suppliers have finally made their service-plans, these plans must satisfy some constraints. □

In EX.1, we assume that the purchase-order database is very large. (This assumption is often true in real cases. For example, consider that the market deals with the advance-orders (= the orders having an appointed time of supply) in a week. If many retail-stores issue such '1-week advance-orders' on the basis of their weekly sales-log data, then the raw 'order'-database has millions of records. In this case, the suppliers must make weekly service-plans from this large database.)

[1]For each supplier, his service-plan (or market-share) refers to the set of those purchase-orders that he has exclusively acquired. We assume that a purchase-order is exclusively served by one supplier. (i.e. one purchase-order is not shared by multiple suppliers.)

To simulate this system, we use the following relational database. (file_name[list of attribute_names] gives the relational schema of a file.) In the following, each file is the history-data describing a situation of a past time; we choose these files from a history-database in order to imitate the simulated situation:

[Database] 1: A sales-log file per area= Area[Item, Customer, Period, Quantity]. Its tuple $[I, C', T, Q]$ originally means, "In an area (Area), a customer-class (C') purchased an item-class (I) by a quantity (Q) at a time-period (T).". During one simulation, this tuple means also that the same order is now issued in the current purchase-order database. [2] There are four areas: A, B, C, D. All the suppliers share these files.

2: A capacity-file per supplier = Supplier[Item, Customer, Period, Quantity]. Its tuple $[I, C', T, Q]$ means, "When a supplier (Supplier) supplys the item-class (I) for the customer-class (C') at a time-period (T), the maximum amount to be supplied is Q.". The suppliers are X, Y, Z. Originally, these files describe the suppliers' capacities that were available at a past time. In a simulation-run, we use these files to express the current states of the suppliers. (Note: each file is denoted by its area-ID (= A, B, C, or D) or its supplier-ID (= X, Y, or Z). [Item,Customer,Period] is the key attribute for each file.) □

In a simulation, we use the above sales-log files as the current database DB_0. Then, Fig.2 shows which sales-log files are manipulated by the suppliers. As shown in Fig.2, we assume the following behavior for each supplier x (= X, Y, or Z):
x chooses two areas, *Area1* and *Area2*, and then selects some tuples under filtering predicates, *P1* and *P2*, respectively. (Table.1-(a) describes these areas/predicates for each supplier.) Then, by using these tuples, the supplier x decides the purchase-orders to be served by the following strategy:
[Strategy]: Let an *entry* refer to a value (of a tuple) at the attributes [Item,Customer,Period]. Then, for a given entry E, assume that x has a capacity $Q3$; also assume that the two filtered log-files, σ_{P1}(Area1) and σ_{P2}(Area2), have the tuples whose quantities are $Q1$ and $Q2$, respectively, at the entry E. Also, for the entry E, SUM refers to $(Q1 + Q2 + Q3)$. Then x tries to occupy the tuples whose SUM's are as great as possible. (Note: If x has exclusively acquired a tuple t, it means that x supplies the entry of t by the amount of $(Q1 + Q2)$. SUM is set to 0 if $Q1 = Q2 = 0$) □

When the game is over, all the suppliers will have decided their final service-plans. At this time, these

[2]In a simulation, the time-period (T) of a tuple is interpreted as an appointed time of supply or delivery. The customer-class (C') represents a retail-storeID with its target class of customers.

```
LineNo  rule_x (K) {
  1     T = L ⋈ σ_P1(R1) ⋈ σ_P2(R2) ;
  2     Rank tuples of T by using their SUM's ;
  3     Put back some acquired tuples to R1, R2;
  4     R2 = R2 − σ_SUM≥K(T) ;
  5     R1 = R1 − σ_SUM≥K(T);
  6     Append σ_SUM≥K(T) to the answer_file of x;
  7   }
```

(R1,R2 = log-files of Area1, Area2. L = the supplier's capacity-file. SUM = gain of a join-result tuple. The answer_file of x holds the acquired tuples of x. A set-update operation is expressed by relational set-difference.)

Figure 3: Rule for the supplier x (= X, Y, or Z)

```
LineNo.
 1  i = MAX;  K = Rank[i];
 2  while ( loop_count < limit ) {
 3   CoBegin /* all rules run concurrently */
 4      rule_X(K); rule_Y(K); rule_Z(K);
 5   CoEnd   /* wait until all rules end.  */
 6   if (no tuples in the database, AND
          Table.1(b) is true )  then exit;
 7   i = i-1;  K = Rank[i];
 8  }
```

(rule_x= the rule of Fig.3-form for the supplier x.)

Figure 4: simulation-code for EX.1. (C-lang. style)

plans must satisfy all the constraints of Table.1-(b). In this table, the function $S(x)$ represents the total gain of x. ($S(x)$ is equal to $\Sigma(SUM)$ for all the tuples that the supplier, x, has decided to supply.) Thus, in a simulation-run, each supplier x works in order to increase $S(x)$ over the gains of the other suppliers.

2.2 Coding

This subsection describes a database coding of EX.1. In the following, an *agent* refers to a component to manipulate a database. Also, a *rule* refers to the procedure specifying the actions of an agent. Thus in EX.1, each supplier is an agent.

Fig.3 lists the procedure of a supplier-agent x. In this figure, the 1st line (denoted by `line1`) first makes the outerjoin of the three files on the entry of [Item, Customer,Period] [3]. Next, `line2` classifies the result-table of join-operation (abbreviated to 'join-result' or 'join table'), T, into some ranks (Rank[MAX] to Rank[0]) [4]. This ranked view is used to decide the target tuples: At `line4` and 5, this rule decides an appropriate set of tuples to be served, and then acquires them through the set-operations (= deletion) from R1 and R2. `Line3` of Fig.3 expresses a negotiation [5,16]; here, the agent puts back some acquired tuples to R1 (*or* R2) if a constraint related to himself is false, or if these tuples are taken away by another agent.

Fig.3 exemplifies that in our general simulation-problem (of Fig.1), every agent describes his strategies/actions by set-operations. In other words, we aim to test such 'set-oriented' behavioral policies of agents in the simulations. As the result, every rule is designed to read/update large bulks of data. This is a great difference from traditional 'tuple-oriented' rule-systems.

Using the procedure of the agents, Fig.4 describes the overall simulation-code of EX.1. From here on, let us abbreviate the j-th execution of the inside of the while-loop to 'j-th loop' or just 'loop'.($j = 1, 2, \ldots$). Then, each loop in Fig.4 represents one turn of play among all the agents; there, between `CoBegin` and `CoEnd`, all the rules are executed concurrently without fixed orders. Because of this concurrent execution, every rule of Fig.3-form must exclusively lock its shared files, R1 and R2 (denoted in Fig.3); but its capacity-file, L, is a 'read-only' file. Under this concurrency control, we run each rule as a serializable transaction in Fig.4 [5]. When all the rules have done their actions for one loop, the code starts the next loop until the game is over.

2.3 Other games

The EX.1 becomes a more realistic gaming-simulation by the following five extensions; they are easily implemented in Fig.4 and Fig.3:

(1) The assumption that every agent does not have global information of the whole system. e.g. In EX.1, for each (supplier-) agent, he does not know any constraints that are unrelated to himself; also, he does not know the strategies of the other agents; and he cannot access the capacity files of the other suppliers. About the other agents, an agent knows only the values of the gain-functions $S()$.

(2) In EX.1, at every turn of play, an agent tries to get some new tuples. We can limit the total amount of

[3] In Fig.3, R1 (or R2) refers to the sales-log file of *Area1* (or *Area2*) for a supplier x. (See Table.1-a.) L refers to the capacity-file of x. Then the result of join (called 'join-result' or 'join table'), T, has the schema of [Item, Customer, Period, SUM, tid of R1's tuple, tid of R2's tuple]. We use a full outerjoin [9] for T; i.e. If a relation R_i has no tuple matched with a given entry Key, then the outerjoin-table T (= $R1 \bowtie R2 \bowtie R3$) has a tuple of the entry Key where $Q_i = 0$ and tid=NULL for such i. ($i = 1, 2, 3$). To update a tuple, we use its tuple-identifier (tid).

[4] the i-th rank = { t | t is a tuple s.t. Rank[$i+1$] > t's SUM ≥ Rank[i]}. (i = MAX, MAX-1, ..., 1, 0)

[5] It is because the strategies/actions of a rule are specified by set-operations, and because we want to know their exact effects on a database. To assure this exactness, the atomic execution of a rule is necessary.

these tuples per turn. This 'limit' is an abstraction of 'cost' or 'power' per turn. It makes a fair competition among the agents.

(3) To get a better market-share, an agent can use 'taking-over' conditions: i.e. If some conditions hold, an agent X can force another agent Y to release tuples of Y (= a part of Y's market-share), and then X can take them over.

(4) An agent can be a state-transition model; thus he can change his behavioral strategies during a game.

(5) Playing an 'evolving' game on several rounds. (e.g. In EX.1, the first round uses the order-log database of the 1st week, and the second round uses that of the 2nd week, and so on.) Here, one round means playing one game to its end. Then an 'evolving' game means that, at every new round, the agent-models or the database get evolved according to the result of the previous round. (e.g. some agents generate group-cooperation, or the database-schema gets evolved, etc.) This evolution provides more realistic games, although it uses large databases.

3 Formalizing the Problem

3.1 Concept of gaming-simulation

Section 1 and 2 have described our study with various examples. The following points (1 to 5) summarize the concept of our gaming-simulations using database:

(1) Our problem is to simulate actions of the agents who work on a large database (as shown in Fig.1).

(2) This simulation uses a large database (D) directly.

(3) D is chosen from history-data; and a centralized parallel database system holds all of those history-data in its secondary storages. Thus the simulation must be computed on this database system. □

The above point 2 is necessary because we aim to simulate real multi-agent systems in the following three cases: The first case is that a multi-agent system works on a current-state database of large volume (e.g. the case of market-trading games in Section 1 and 2, which deal with the advance-orders in a week). The second case is that multiple agents make cooperative planning for a long-term period, and that this planning uses a large database which describes the raw/statistic data over this period. (e.g. the case of urban-policy cooperative planning for each month, in Section 1).

In addition, about the point 2, there is the third case that the simulations are simplified on the time-scale. To explain it, assume that in the example EX.1, the purchase-order database has all the raw data that describe the system-activities over a month; also, assume that the agents use *week-level* behavioral strategies (= the strategies depending on the days of a week. Such strategies do NOT depend on the time-scale of 1-month.) Then we can consider that the simulation of such EX.1 has simplified the agent-models on the time-scale. (It is because, on the '1-month' raw database, each agent is designed to use his week-level strategy for all the weeks.) Clearly, this case is not an exact simulation over the 1-month data, but a simplified one on the time-scale. This simplification is effective to test/develop better behavioral strategies on long-term history-data. Section 6.2 will discuss this advantage.

Because of the above useful cases, our gaming-simulations (of Fig.1) use very large databases, which are held in a centralized database system. Also,

(4) we assume that our gaming-simulations are executed by a single (human) user [6]. i.e. To run a simulation, the single user must give all the agent-/system-models of target systems. (*or* some of the agent-/system-models may be prepared as the programmed ones. Also, the user may not know the programmed agent-models.) In many cases, each agent-model is designed to use a local strategy without global information. □

As a result, by using our gaming-simulations, the single user can know the effects of various agent-/system- models on the basis of real history-data; also, such knowledge enables him to test/improve the models with great reality. These are main advantages of our study. (e.g. the single user can test/improve one-agent's strategy against the other programmed-agents; *or* he can develop a better group-cooperation strategy in an 'evolving' game of Section 2.3.) Furthermore,

(5) the tested agent-models (= strategies/actions) are described by set-operations. (See Section 2.2.) □

The point 5 is assumed because such a 'set-oriented' agent-model is necessary to roughly sketch the behavioral principles of an agent. (e.g. A typical agent-model rewrites/deletes/inserts a large part of a database through general database-views.) Such a rough sketch and its improvement are the significant part for decision-making; thus we aim to test/sophisticate such 'set-oriented' agent-models through the gaming-simulations on a database.

3.2 Problem definition

In the rest of this paper, we formalize our simulation-problem, and discuss its efficient computation on a parallel database system. This subsection describes our formalized multi-agent problem; we call it a distributed constraint-satisfaction problem (DCSP) on

[6] Originally, the term *gaming-simulation* refers to a game played interactively by some human players [6]. This paper uses this term as a traditional (single-user) simulation on database, although such original usage is possible.

a relational database, denoted by DB-DCSP. DCSP is a constraint-satisfaction problem modified for distributed agents, such that every agent does not know global knowledge [5,16]. Then DB-DCSP is a DCSP except that agents have relational variables and share a database as the global information:

[Def. of DB-DCSP] Consider a multi-agent system where the agents, denoted by A_i $(i = 1, \ldots, n)$, share a relational database D (= a finite set of relations). For each agent A_i, assume the following:

1: A_i has a unique variable x_i. The value of x_i is a relation [7]. A_i allocates tuples to x_i, where those tuples are chosen from D. The allocation has two distinct modes: i.e. the allocation of a tuple to only one variable (*exclusive mode*) or to some variables (*shared mode*). [8]

2: Among the agents, there are boolean constraint-predicates $P_k(x_1, .., x_n)$. $(k = 1, \ldots, m)$. A_i knows only the constraints including x_i, and can evaluate them. He knows no other information about the other agents.

3: A_i can have a private relation L_i, such that the other agents cannot read or write L_i.

On these assumptions, DB-DCSP is defined as follows: "For every A_i $(i = 1, \ldots, n)$, A_i must find the value of x_i so that all P_k are true. Also A_i must work as specified in a given behavioral model". □

Apparently all the previous examples are generalized to DB-DCSP. In general, agents work towards cooperative or competitive goals [5]; thus, EX.1 is generalized to a DB-DCSP having competitive goals.

3.3 Processing models

Here we describe processing models of DB-DCSP on a 'Shared-Nothing' [4] parallel database system.

[Platform/Cost] As a platform, we use a database system having parallel (computer-)nodes interconnected by a network [4,8]. Fig.5 shows such a system of 8-nodes. In general, every relation is partitioned into fragments, and these fragments are stored at the distinct nodes of Fig.5. By this partitioning, the nodes of Fig.5 execute a database operation in parallel.

Because we aim to use very large database, we assume that all the (not temporary) relations are stored in disks. (no memory-resident relations). Thus, we consider only the following costs: the I/O cost to access bulks-of-data from/to disks (called 'bulk-access'),

[7] a relation is a set of tuples. This paper uses the term, 'relation', as a conceptual relation-table.

[8] A_i can read (= do the shared allocation of) or update (= do the exclusive allocation of) a tuple t to x_i only when t is not allocated exclusively to another variable x_j $(j \neq i)$.

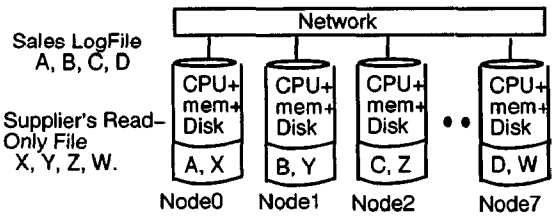

Figure 5: a parallel database system of 8-nodes

and the load-balancing of these data-accesses on the parallel nodes.

[Data Placement] We consider a range-partitioned relation [4] or a partially-declustered relation [4] in Fig.5. (They have the advantages of both less message-overhead and easy database-management in Fig.5). To model these placements, we redefine the term a **file** as follows: a *file* refers to some subclusters of a relation, so that one rule-step (= a 'file-read' or 'file-write' step of a rule) accesses only one *file* [9].

Example: Fig.5 describes a file-placement for EX.1 of Section 2.1: there, Node0 has the files A and X; Node1 has B and Y; and so on. Thus, in Fig.5, each file of EX.1 is not further declustered on some nodes, but is placed on one node. [10] □

In general, in Fig.5, a file (F) redefined above can be further declustered on some nodes; the number of these nodes is called the *Degree of Declustering* (denoted by DD) of F [4] ($DD = 1, 2, \ldots$). Thus Fig.5 is the case where $DD = 1$ for each file.

[Transaction model of a rule] As described in Section 2.2, in DB-DCSP, every rule (for an agent-model) is a transaction specified by set-operations. Then, such a rule is a transaction reading/writing large bulks of data, called a *Bulk-Accessing Transaction* (BTX) [11]. This is explained as follows: In general, data-placements and file-organizations are designed independently of given rules; hence (1) a rule cannot use good indices of an operand-file, *and* (2)

[9] Such a file is semantically-clustered so that one rule-step does not access two files. We intend here that a rule-step has a filtering predicate, and that the predicate hits the subclusters represented by one file. e.g. In the range-partitioning policy, a relation is partitioned into subrange-fragments. Then, in EX.1, each 'order-log' file (= A, B, C, or D) is considered to be a semantically-clustered subrange-fragment of the global 'order-log' relation (of all the areas). The capacity-files are interpreted in the same manner.

[10] 'Declustering a file on k-nodes' $(k = 1, 2, \ldots)$ means 'splitting the file into k-fragments, and then placing the k-fragments on the *k-distinct* nodes'. To abbreviate this meaning, we say '$DD = k$ for a file', or 'a file has DD of k', or 'a file declustered on k nodes'. e.g. In Fig.5, 'declustering the file A on 2 nodes' means that A is further split into two fragments, and that the fragments are placed on two distinct nodes.

STEP
1: get S-Lock of L, and build the hash-table, T, of L.
2: get X-Lock of R1, and outerjoin T with R1.
3: get X-Lock of R2, and outerjoin T with R2.
4: rank T, and find the tuples to be updated.
5: update R2 through tuple-identifiers.
6: update R1 through tuple-identifiers.
7: commit, and release all the locks.

(S-lock = Shared lock. X-lock = Exclusive lock. At Step2&3, T is probed by R1&R2, and is used as the outerjoin-result table. At Step4, 'rank' means classifying T into the ranks. The steps are executed sequentially.

Figure 6: BTX-model for a Fig.3-form rule

at the start of Fig.4, a rule must scan the files declustered on a very small number of the nodes (e.g. In Fig.5, $DD = 1$ for every file); then, owing to (1), when a rule wants to execute a read-/write-step having a filtering-operation, it must lock the whole operand-file and perform a file-scanning operation; hence, such a rule has to access bulks-of-data. (The point (2) above easily causes load-unbalancing, as discussed later.)

A BTX is formally defined as follows [11]:

[BTX model]: a BTX is a sequence of 'file-read' or 'file-update' steps. Each of these steps scans the major part of a file. (Index may be used, but a file-scan is a frequent operation.) Also, each step uses its operand-file as a locking-granule, which represents any given predicate-lock. As a locking model, a BTX obeys the two-phase locking manner. [11] (Appendix.A describes the cost model of a BTX). □

As an example, Fig.6 describes the BTX model (= its step-sequence) that corresponds to a Fig.3-form rule. We use here the hybrid hash join [4] and assume that the hash table or its ranking is resident in main memory. [12] (In Fig.7-(a), we illustrate the notations of two BTX's, T1 and T2.)

4 The Database Algorithms

4.1 Technical issues to be solved

For any DB-DCSP, its database-processing code can be described in the form of Fig.4. Thus the rest of

[11] i.e. a BTX must acquire a shared(S) (or an exclusive(X)) lock on its operand-file for its read(or write)-step, and then releases all the locks at its commitment. If a BTX1 requests a lock to a file and if another BTX2 has held a conflicting lock on this file, the requester (BTX1) gets blocked until the current lock gets released. (an X-lock conflicts with all the other locks.)

[12] We assume that a BTX performs its steps sequentially (i.e. one step after one). Parallelism within a BTX is represented by executing one step in parallel; i.e. if a step wants to access an operand-file whose $DD \geq 2$, then DD-nodes (= those holding the file) execute this step in parallel.

(a): Transaction models of T1 and T2.
T1: r1(A:1) → r1(B:3) → w1(A:1).
T2: r2(C:1) → w2(A:1) → w2(C:1).
(Note: $r_i(F:C)$ (or $w_i(F:C)$) refers to a read (or write)-step of T_i to the file F, where C = I/O cost of the step. A BTX has the step-sequence denoted by $step_1 \rightarrow ... \rightarrow step_n$. T1 is a BTX having 3 steps, where it first joins the files, A and B, and then updates 50% of A.)

(b): WTPG for when T1, T2 have started.

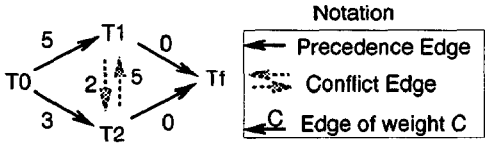

Figure 7: Examples of BTX's and a WTPG

this paper discusses executing the code of Fig.4-form on the processing models of Section 3. That is, we consider the situation that a DB-DCSP uses a very large database stored in a parallel database system. Then, to compute a DB-DCSP (Fig.4-form code) on such a parallel database system (of Fig.5), we adopt the following general execution-manner:

At each loop of a Fig.4-code, do the following processing: First, all of the triggered rules (= BTX's) are submitted to the concurrency-control scheduler. Next, the scheduler dynamically schedules appropriate BTX's, and executes them on the database system. When all the rules end their actions that are specified for this loop, the next loop starts the same processing. □

Clearly, this manner needs two inherent operations of BTX's: i.e. concurrency-control scheduling and iterative execution. For these operations, this section proposes the following algorithms:

(1) Effective concurrency-control scheduling, tuned for BTX's: This is necessary because, at the start of DB-DCSP, every rule (= a BTX) scans the files declustered on a few computing-nodes. That is, in the case of Fig.5, we must consider that a rule may scan a file whose $DD = 1$ to 4 on the 8-node machine. [13] This fact easily causes load-unbalancing between the nodes. Thus, for better load-balancing, a scheduler must execute much more rules concurrently.

Remember that the rules are BTX's. Their scheduling has high degree of the locking/resource contention [14], because each BTX needs coarse-grain (i.e. file-level) locking and must access large bulks of

[13] e.g. To execute EX.1, consider starting the Fig.4-code on the file-placement of Fig.5. Then, each rule of Fig.3-form (= a BTX of Fig.6) scans the files sequentially, and also $DD = 1$ for each of the files. Thus it causes load-unbalancing.

[14] i.e. the contention caused by both locking-conflict and resource-level congestion [1,11]

data[11]. This high-degree contention heavily limits the degree of concurrency [1,11]. Thus, in order to extract rich concurrency of BTX's, a scheduler must reduce this contention as greatly as possible. Traditional locking protocols are for short transactions, and thus lack this ability [11].

Owing to this reason, we have developed a dedicated concurrency-control scheduler of BTX's [11]. We use it for the concurrent execution of the rules.

(2) Differential computation of a rule at the 2nd and further loops; this is to reduce bulks-of-data accessed at the iterative execution of a rule (= a BTX). The same idea are used in traditional rulesystems, but they are tuned for 'tuple-oriented' rules, which update only a few tuples per loop [2,3,15]. In contrast, DB-DCSP's use a BTX having set-oriented database-operations, such as 'bulk-read' or 'bulk-update' ones [15]; thus we modify TREAT [2] for such BTX's.

4.2 The scheduler

This subsection introduces our concurrency-control way of BTX's, called the Locally-Optimized WTPG (LOW) scheduler [11]. This scheduler uses a serialization-graph [10] called a *Weighted Transaction-Precedence Graph* (WTPG); here, a WTPG describes the serialization orders between BTX's, and does also the locking/resource contention in a schedule. By using such a WTPG, LOW is aimed at reducing the locking/resource contention in the BTX's scheduling. This subsection explains only the configuration of a WTPG and how LOW uses it; readers can find the formal definitions in [11,12].

[WTPG]: Fig.7-(b) illustrates a WTPG, in which the two BTX's ($T1$ and $T2$ in Fig.7-(a)) have just started [16]. To build a WTPG, every transaction must declare, when it starts, its step-sequence including its I/O-cost information. (i.e. Fig.7-(a) are such sequences of both $T1$ and $T2$). Then, at the current scheduling-state S_{now}, a WTPG is built as follows:

- The nodes represent transactions (= those working at S_{now}, as shown in Fig.7-b);

- Suppose that two transactions (Ti and Tj) have declared a conflicting access to a common locking-granule. Then, between such Ti and Tj, we set a pair of shaded edges (called a conflict-edge, denoted by (Ti, Tj)). It means that Ti conflicts with Tj in serialization-order, and that the precedence-order between them is not determined yet. If this precedence-order is determined, then (Ti, Tj) is replaced by an appropriate *precedence-edge* immediately. (e.g. In Fig.7-b, $(T1, T2)$ is set because $T1$ conflicts with $T2$ on the file A.)

- A precedence-edge (e.g. a solid edge in Fig.7-b), denoted by $\{Ti \rightarrow Tj\}$, means that at S_{now}, Ti is serialized before Tj.

- Every edge has a cost (*or* called a weight), which represents the degree of the locking-/resource- contention (at S_{now}): (i) For an edge $\{T0 \rightarrow Ti\}$, it has the cost C_R, which represents the resource-contention degree of Ti. (C_R = the I/O cost that Ti has left before its commitment.) C_R is adjusted when a schedule proceeds. (ii) If an edge exists from Ti to Tj $(i, j \neq 0)$, it has the cost (C_L) to represent the locking-contention degree. (C_L = the I/O cost that Tj will have left if Ti blocks Tj.) We can set up these costs by the declared transaction-patterns [11]. □

[Function $E(q)$: q = a lock-request]

Phase 1: In the WTPG of the current scheduling-state, find all the conflict-edges (Ti, Tj), such that Ti gets serialized before Tj when granting q. Then replace all of such edges temporarily by $\{Ti \rightarrow Tj\}$. Next, delete all the remaining conflict-edges temporarily.

Phase 2: In the above temporary WTPG, find the longest path (P_L) from $T0$ to Tf. $E(q)$ is set to the length of P_L. If a deadlock occurs, $E(q) = +\infty$. (Length of a path P = the sum of the costs of all the edges in P.)

Figure 8: Procedure of Function $E(q)$

The LOW scheduler always keeps the WTPG that represents the current scheduling-state. Then, using this WTPG, LOW works as follows:

[LOW strategy]: Suppose that a lock-request q is submitted. Then, LOW first computes a function, $E(q)$ (See Fig.8). $E(q)$ represents how much the contention will occur if LOW grants q now. Next, LOW grants q (1) if q is not blocked, and (2) if q causes no deadlock, and (3) if $E(q) \leq E(p)$ for all p, where p is an access-declaration that conflicts with q on the requested locking-granule. In the other cases, LOW blocks or delays q. [17] □

It is clear that this strategy generates a serializable schedule of BTX's; furthermore, by the above condition (3), LOW grants only those lock-requests that keep the 'locking/resource' contention in the lowest level. In this way, the LOW scheduler successfully reduces the locking/resource contention in BTX's

[15] a bulk-read (or bulk-update) operation means reading (or updating) large bulks of data from/to a database.

[16] $T0$ (or Tf) in Fig.7-(b) refers to the virtual transaction that is serialized before (or after) any other transaction.

[17] The delay of q means that the transaction requesting q sleeps for a while. The block of q means that the requester gets blocked. The number of p is limited by an upper bound K. If the limit is not kept, LOW delays the start of new transactions. This paper uses LOW where $K = 2$ because of its better performance [11].

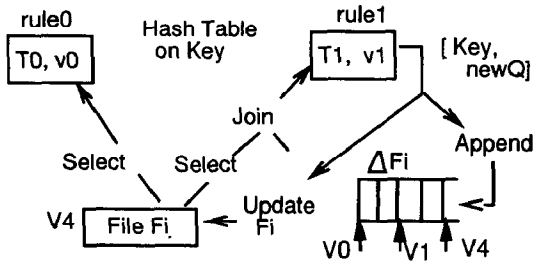

Figure 9: outerjoin-tables & differential queue-file

scheduling. Our work [11] has demonstrated that LOW successfully extracts rich concurrency of BTX's in various workloads, and that it is much better than the traditional locking protocols developed for short transactions.

4.3 Iterative execution

Here we discuss the differential computation of a rule at the n-th loop; i.e. a rule is recomputed by using the $(n-1)$-th result (for $n \geq 2$). We modify TREAT [2] in order to process bulk-update operations per loop. Our method is called OJT (= Using an OuterJoinTable per rule and differential files shared by rules.)

We discuss only the rule-form of Fig.6: i.e. a multi-way outerjoin on a common key, followed by aggregation/grouping. Many DB-DCSP's are expected to use this form. Thus we discuss the following case:

For each $i(= 1, 2, ..., n)$, consider a base-relation file F_i (of the schema [Key, Qi]), and its differential file δF_i (of the schema [Key, newQi]). (δF_i is the set of 'update-tuples', i.e., those tuples that updated F_i after a given past time.) Then we discuss the case to recompute the outerjoin (T) of all the F_i's at a loop. ($T = \sigma_1 F_1 \bowtie .. \bowtie \sigma_n F_n$. T has the schema of [Key,Q1,...,Qn]) [18] □

Now consider applying TREAT to the above case. Remember that the original TREAT is for a natural join and keeps two kinds of temporary files: i.e. a join-result table (called a conflict set) and a selected result per base relation, σF_i (called an α-node); also, in order to recompute the conflict set, TREAT originally joins δF_i's immediately with these files. Thus the original TREAT has disadvantages in the above case, such as the large temporary files, indices, and much random I/O's caused by bulk-updates.

However, in the above limited rule-form, TREAT really needs neither α-nodes nor any additional indices for join. It is because i) the outerjoin-result, T, includes all the tuples of F_i; *and* ii) both T and F_i are clustered on the key. To reduce tuple-level random I/O's, we should also use lazy-evaluation for the join between δF_i and T, as often seen [2,15].

[18] Qi refers to non-key attributes. Key refers to the key-attributes. Because of the outerjoin, when T has a tuple t whose Key = *key0* and when F_i has no tuples whose Key = *key0*, then Qi = NULL in t. If a tuple is deleted in F_i, the corresponding tuple in δF_i sets its newQi to NULL.

[OJT method]
INPUT: T = outerjoin-result of r_i at (N-1)th loop, F_i = database file, δF_i = differential file for F_i.
OUTPUT: T = outerjoin-result at N-th loop.

Phase1: From δF_i, read out all the tuples, t, in the FIFO order, where $pos(t) \geq V_{old}$. ($pos(x)$ = position-id of a tuple x in δF_i. V_{old} = viewID of F_i that r_i used at the (N-1)th loop.)

Phase2: Let t be each of the above tuples. (Let t have the value of [Key, newQi]). For all t in the read-out order, do the following operations (i) to (iii):
(i) Test if t satisfies the selection condition of r_i;
(ii) If the test-result is true, then find the tuple J in T, such that J is equi-joined with t. (Let J have the value of [Key,..., oldQi, ..., Qn]);
(iii) Update oldQi to newQi in J. (If T has no matched tuples with t, then insert a new 'outerjoin-result' tuple including t.)

Phase3: Delete the mark V_{old} from δFi. If V_{old} is the minimum among the V-marks of δFi and if $V_{min'}$ is the second minimum there, delete all the tuples t' from δFi, such that $V_{old} \leq pos(t') < V_{min'}$. Finally, put the new mark V_{now} at the last entry-position of δFi. (V_{now} = the current viewID of F_i locked by r_i.) □

Figure 10: OJT-method for a rule r_i

From these ideas, our method (OJT) uses the following data-structures: (1) For each rule, we keep its outerjoin-result T_i as a temporary file. (2) For each file F_i, we keep its differential file (δF_i) as a FIFO-queue organization. δF_i is shared by the rules accessing F_i. F_i has its view-identifier (viewID) V. (V is incremented only when a rule gets an exclusive lock on F_i.) (3) When a rule (r) wants to update F_i, then r really updates F_i, and appends the new values of the updated tuples to δF_i. At the same time, r puts a mark V_x at the first entry-position of those new tuples in δF_i, where V_x = the current viewID of F_i. □

Using these data-structures, Fig.10 describes the procedure of the OJT-method. Fig.10 is used when a rule wants to recompute its view from a given δF_i. In this figure, Phase 2 is to apply TREAT to our case; also, Phase 3 is storage deallocation of the (shared) differential files.

e.g. Fig.9 illustrates the data-structures of OJT and how it works. In this figure, the two rules (*rule*1 and *rule*0) share the database file F_i. The join-tables ($T0$ and $T1$) and the differential file (δF_i) are all clustered on the key. Assume also that *rule*1 and *rule*0 previously read F_i whose viewID = $V1$ and $V0$, respec-

tively. Now, consider that *rule*1 wants to recompute its view ($T1$) and that *rule*1 has locked F_i whose current viewID = $V4$. Then the procedure of Fig.10 works as follows: First, *rule*1 reads out all the tuples located after $V1$ in δF_i; next, *rule*1 re-computes $T1$ by the Phase 2 of Fig.10; third, $V1$ is removed from δF_i; finally, *rule*1 puts the new mark ($V4$) at the last entry of δF_i. *rule*1 uses the mark $V4$ at the next loop.□

In the OJT method, δF_i can utilize a buffered (and no-random) I/O. Thus OJT is better than the original TREAT, because OJT has neither α-node nor random I/O caused by bulk-update per loop. OJT has another advantage by declustering δFi on the parallel nodes of Fig.5; that is, it provides better load-balancing.

4.4 The overall processing method

Until here, we have described the technical solutions for the database-operations of DB-DCSP. Remember that we follow the execution-manner of Section 4.1; thus, by using these elementary solutions, we propose the following overall method to compute a DB-DCSP (Fig.4-form code):

[Method1] At every loop, we use LOW for the concurrency-control scheduling of the rules. (Each rule is executed as a Bulk-Accessing Transaction (BTX).) The first loop executes the rules of Fig.6-form. At the 2nd and further loops, every rule uses the OJT method in order to recompute its view; thus, from the 2nd loop on, the rules of Fig.11-form are executed. (Fig.11 is the step-sequence of a rule using the OJT-method, where this rule has originally Fig.6-form.) □

To use Method1, Fig.6 and Fig.11 need some modifications: (i) At the 1st loop, a rule of Fig.6-form has to add the following steps: Just after the step3, this rule needs a step to save the outerjoin-result, T, to disks; also, the step5 and the step6 include appending the updated tuples to $\delta R2$ and to $\delta R1$, respectively. (ii) At the 2nd and further loops, a rule of Fig.11-form saves the whole (or updated) region of T after the step3. □

With the above modifications, we propose Method1 as an efficient and basic computing manner for DB-DCSP on a parallel database system. We discuss some extensions later.

5 Preliminary evaluation

This section evaluates the proposed processing-method (Method1 in Section 4.4) for DB-DCSP. Because of the space limit, we only summarize the experiments reported in [12].

In addition to Method1, we test also the following methods: Method2 (= the same as Method 1, except

STEP
1: load the outerjoin-table T to main memory.
2: get X-lock on R1, and recompute T by δR1.
3: get X-lock on R2, and recompute T by δR2.
4: rank T, and find the tuples to be updated.
5: update R2, and append the updated tuples to δR2.
6: update R1, and append the updated tuples to δR1.
7: commit, and release all the locks.
(Step2 & 3 recompute T by the procedure of Fig.10.)

Figure 11: Step-sequence of a rule using OJT

using Atomic-Static Locking (ASL) instead of LOW)[19] and Method3 (=using the OJT method from the 1st loop on. ASL is always used as a concurrency control scheduler.).[20] Method2 represents traditional locking protocols, and Method3 is the 'pure' differential-computing strategy of traditional rulebases.

The experimental conditions are as follows:

- We extend the EX.1 (of Section 2) to that of 8-areas plus 32-suppliers, and compute this DB-DCSP problem (of an extended Fig.4-form code) on a simulator of an 8-node parallel database system. The simulator has the Fig.5-configuration. It includes a rule-interpreter, reporting the run-time from the observed tupleI/O's. In the simulation, the multi-programming level of BTX's is set to 8; also, a BTX is switched at a computing-node every time when it has completed the data-access of 1-unit.
- The agents take the strategy that they try to hold major market-shares at the first two loops, and then improve their plans.
- The used database has 1200 tuples [21]
- To consider the internal parallelism of BTX's, all the database files have various Degree of Declustering (DD = 1 to 4). The differential files (those used in the OJT method) are also declustered (DD' = 1 to 8).

The following is the experimental results [12]:

1: For the 1st loop-processing time, LOW (Method 1) has the best performance; in contrast, the traditional locking-protocols (= the two-phase locking protocol(2PL) [1] and the atomic-static locking (ASL)) perform much worse because of high locking-contention. (In the experiments, at the 1st loop-processing, LOW had 1.33 - 1.75 times better performance than ASL. We tested also 2PL, but it was worse

[19] ASL is the two-phase locking protocol in which a BTX can start only if it gets all the necessary locks.

[20] Method3 needs the preprocessing that each rule computes its view and saves it. This overhead-time is included in the 1st loop-time

[21] This is small, but our result is valid for much larger database. It is because all the tested methods use mainly file-scan; thus the simulation-results can be applied for a large-file scanning if the ratio of (tuples/file) is the same. We have confirmed the result in a larger database (about 60000 tuples).

than ASL there.). It confirms that LOW is superior in the scheduling of BTX's.

2: In particular, when the base-relations have DD of 4 and when the differential files have DD' of 8 on the 8-nodes system, LOW has very good load-balancing of these nodes. (When the experiment used Method 1, the 8-nodes had the average utilization-ratio of 87% at the 1st loop-processing; there, Method 1 was over 1.33 times better than ASL and 2PL).

3: It is not efficient to use only the OJT-method from the 1st loop on. (In the experiment, at the 1st loop-processing, Method1 was over 1.57 times better than Method3.) It is because the rules issue large bulk-updates at some loops and thus enlarge the I/O cost of the differential files.

4: From the 2nd loop on, OJT efficiently computes each iteration when compared with the 1st loop-processing time. (Each loop-processing time was reduced to the 1/3 or 1/4 of the 1st loop-processing time.) However, further reduction is not achieved even at a few tuple-updates per loop. This is due to loading/saving the whole outerjoin-result tables, as well as the access-cost of the shared differential files. □

These results confirm that, for the large (and parallel) database-processing of DB-DCSP, Method 1 is better than traditional locking protocols (Method 2) and traditional rulebase strategies (Method 3).

6 Discussion

6.1 Extending the algorithms

Another study [13] of ours have extended OJT, as follows: (ext.1) The differential files (δF) are multi-dimensionally clustered; (ext.2) To reduce the load of join-result tables, we have proposed a 'current scope' of a rule. (e.g. in Fig.4, a rule has the current scope that $SUM \geq Rank[i]$ at a loop.) The scope describes a strategic preference of an agent. Then, the rule must load only the data of its 'current scope' at a loop. If necessary, the rule switches its scope to the next one by scanning its outerjoin-result table.

These extensions can accelerate the original OJT [13]. In many DB-DCSP's, we expect that agent-models use a fixed scope for some loops; thus the ext.2 greatly improves the performance.

The ext.2 is a natural extension of LEAPS [2] (LEAPS is a rule-base algorithm for database production systems.). To reduce the conflict sets, LEAPS uses a timestamp as a tuple-oriented search-priority, which cannot be applied to the bulky set-operations of DB-DCSP's. Thus the ext.2 has realized such a priority-scope by an aggregation/group-by function.

6.2 Extended usage of our study

Here we discuss how to use our study with traditional gaming-simulations [6] or data-mining studies [7].

As described in Section 3.1, our simulation-problem uses large databases of multi-agent systems with no miniatured abstractions. On the other hand, many traditional gaming-simulations do use a small and abstract model for these large databases [6].

Then, how can we find such good abstractions (= the abstract models above)? Such abstractions are nothing but significant strategies for the agents; also, in the early stages of simulations, it is general that we do NOT know these abstractions exactly. Therefore, to discover such good abstractions, we must use the raw history-data of target systems.

The above necessity is a reason why our simulation-problem directly uses large history-databases. (The other reason is to test agents' strategies on real history-data, as described in Section 1.) That is, by running gaming-simulations on real (and very large) history-data, we can discover useful abstractions from the observed log-data in the simulations. After finding such abstractions, the target systems can be abstracted to much better and small simulation-models. In this way, our study supports testing/developing abstractions of multi-agent systems by using large history-data.

We expect that our gaming-simulations use large databases of giga-byte size having millions of records. There are three useful cases to use such a large database (as described in Section 3.1). In particular, for the above discovery-applications, the case of simplified simulations on the time-scale (in Section 3.1) is effective. It is because this case can easily test the strategies that have global views over a long period. In other words, the simplified simulations support rapid model-developing on history-data. (An example is to develop a group-cooperation model from both the component models and long-term history-data.) Also, this case uses roughly-sketched agent-models, and such rough modeling easily causes bulk-read/bulk-update operations. Thus our algorithms for BTX's are significant especially for this case.

7 Summary and future works

This paper has discussed 'game-style' (or gaming-) simulations of multi-agent systems using large (history-) databases. We have discussed how to execute these simulations on a centralized parallel database system, assuming that it holds all of the history-data.

The main contribution of this paper is to have clarified the concept of gaming-simulations using large

databases. To exemplify it, we have described the simulation of a multi-agent system using a large 'current-state' database (*the case of market-trading game*), *and* the simulation of a multi-agent cooperative planning for a long-term period (*the city-developing simulation among multiple policy-makers*). Also, we have described that our study can simulate such multi-agent systems on the simplified time-scale; this simplification is particularly useful to develop effective behavioral strategies by using long-term history data.

To assert our study, this paper has described the following contributions:

- To formalize these examples as a distributed constraint-satisfaction problem on database (DB-DCSP).

- To propose efficient database algorithms for DB-DCSP on parallel database systems: i.e. (i) a dedicated concurrency-control scheduler for the agents' transaction-models, and (ii) a TREAT-style algorithm for iterative execution of these transactions.

Concerning the algorithms, we have clarified, at first, that an agent is a transaction accessing large bulks of data (called a BTX). It is because each agent specifies his behaviors (i.e. decision-making and actions) by set-oriented criteria on databases. Next, we have adopted the LOW scheduler [11] for the concurrency-control scheduling of BTX's. LOW has the good ability to reduce the locking/resource-level conflicts in the scheduling. For the iterative operation, we have modified TREAT in order to reduce temporary files and random I/O's. This method (OJT) uses outerjoin-tables with shared differential files. The preliminary experiment has confirmed that our algorithms outperform both traditional locking protocols and the straightforward usage of differential computing.

To improve our algorithms, main functions of LOW should be executed at the compile-time. Also, DB-DCSP needs extensions for general agent-models [2,3] and complicated negotiation manners [16]. It is also interesting to connect our game-style simulations with data-mining [7], because our simulations use behavioral models of agents and because we should discover/improve such models from observed data. In addition, we need a general prototype on a parallel database system. These are our future works.

Acknowledgement: An early stage of this study was done when the first author worked for Dept. Information Science of Kyoto University, JAPAN. He would like to appreciate Prof. Matsumoto in Kyoto U. and his laboratory.

References

[1] Agrawal R., "Models for Studying Concurrency Control Performance: Alternatives and Implications", Proc. ACM-SIGMOD '85, pp.108-121 (1985).
[2] Brant D.A., et al., "Effects of Database Size on Rule System Performance: Five Case Studies", Proc. 17th Very Large Data Bases, pp.287-296 (1991).
[3] Ceri S., Widom J., "Deriving Production Rules for Incremental View Maintenance", Proc. 17th Very Large Data Bases, pp.577-589 (1991).
[4] DeWitt D.J. and Gray J., "Parallel Database Systems: The Future of Database Processing or a Passing Fad?", ACM SIGMOD Record,**19**, 4, pp.104-112 (1990).
[5] Durfee E.H., "The Distributed Artificial Intelligence Melting Pot", IEEE *Trans. Syst. Man. Cybern.* Vol.21(No.6), pp.1301-1306 (1991).
[6] Greenblat C.S., *Designing Games and Simulations*, SAGE Publications, ISBN 0-8029-2956-0, USA (1988). (Japanese translation by Arai K. et al., KYORITSU Pub., ISBN 4-320-02704-3, 1994)
[7] Han J. et al., "Knowledge Discovery in Databases: An Attribute-Oriented Approach", Proc.18th Very Large Data Bases, pp.547-559 (1992).
[8] Kiyoki Y., et al., "Software Architecture of a Parallel Processing System for Advanced Database Applications", Proc. 7th IEEE Int. Conf. Data Engineering, pp.220-229 (1991).
[9] Khoshafian S., et al., *A Guide to Developing Client/Server SQL Applications*, pp.142-143, Morgan Kaufmann (1992).
[10] Nishio S. et al., "Performance Evaluation on several cautious schedulers for database concurrency control", Proc. 5th Int. Workshop Database Machines, pp.212-225 (1987). (in *KnowledgeBase Machines and Database Machines*, Kluwer Academic Pub.).
[11] Ohmori T., Kitsuregawa M., and Tanaka H., "Scheduling Batch Transactions on Shared-Nothing Parallel Database Machines", Proc. 7th IEEE Int. Conf. Data Engineering, pp.210-219 (1991).
[12] Ohmori T., Matsumoto Y., "Parallel Database Algorithms for Solving the Planning Problems on Very Large Databases", Trans. Institute of Electronics,Information&Communication Eng., D-I, Vol.J77(No.8), pp.577-588, JAPAN (1994). *(In Japanese).*
[13] Oda,A., "Data-processing algorithms for supporting Cooperative planning using databases" *(In Japanese)*, B.Thesis, Dept. Info. Sci., Kyoto University, JAPAN (1994). (supervised by the first author)
[14] O'Neil P.E., "The Set Query Benchmark", Chapter.5 in *The Benchmark Handbook for Database and Transaction Processing*, Morgan Kaufmann Publisher (1991).
[15] Richeldi M. and Tan J., "JazzMatch: Fine-Grained Parallel Matching for Large Rule Sets", Proc. 9th IEEE Int. Conf. Data Engineering, pp.616-623 (1993).
[16] Yokoo M., Durfee E.H., Ishida T., Kuwabara K., "Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving", Proc. 12th IEEE Int. Conf. Distributed Computing Systems, 1992.

Appendix.A

[Cost Model of a BTX]: For a given BTX, when its read-step reads N-units of data-access, the I/O (= disk Input/Output access) cost of the step is set to N. When a write-step updates N-units, the step has the cost of $2N$. We neglect the I/O cost between the commitment of a BTX and its completion. (The *unit* is the data-unit of file-scanning, such as a constant number of disk tracks.) □