Vol. 42 No. 12 IPSJ Journal Dec. 2001

Regular Paper

A Method for Dynamic Reorganization of a Database

Vlad Ingar Wietrzyk,[†] Katsuya Tanaka[††] and Makoto Takizawa[††]

We consider the problem of on-line database reorganization. The types of reorganization that we discuss are restoration of clustering, purging of old data, creation of a backup copy, compaction, and construction of indexes. The contributions of this paper are both of theoretical and of experimental nature. We present an algorithm for maintaining the biconnected components of a graph during a sequence of edge insertions and deletions. We complement our algorithms with encouraging experimental results.

1. Introduction

The goal of dynamic reorganization is to arrive at an optimal data layout based on observed access patterns. To accomplish this, the reorganizer must solve three problems. First, it needs to keep track of the history of previous events. Second, it must find a layout that would deliver near-optimal performance for the observed access patterns, assuming that past is a good predictor of future access patterns. Third, it must analyze the difference between the current layout and the desired layout and if necessary, issue I/O requests to correct the difference.

Given a set of objects, the links between these objects may be represented by a labeled, directed graph. We call this graph a *total object graph* (*TOG*). The (*TOG*) is an abstraction which allows us to represent a complete database as well as any subset of objects of the database. The clustering algorithm operates on a set of objects which consists of the objects newly created by the transaction and those objects which have been accessed by the transaction. This set is described by a *Minimum Spanning Tree* (*MST*) which is a subgraph of the TOG that describes the whole database. This (*MST*) represents the objects which must persist at the end of the transaction. Objects are placed in logical clusters, whose size can be unlimited. To group as many objects as possible, we select those objects which were already stored on disk and group some of their newly created components with them. Objects already existing on disk are reclustered—this operation effectively reorganizes database on-line with additional effect of increased performance due to updated references.

The clustering algorithms presented so far in the literature aim at determining a good initial placement of an object, but none of them take into account *dynamic evolution.*

Concurrent reorganization/continuous reorganization of the physical data structure while application process has full access to the database, is an objective of our ongoing research. Our objective is to dynamically adapt physical database organizations, on-line, according to the access patterns of the users without adding significant overhead on processing workload. Clustering is one of the most effective ways to improve the performance of object base applications. Consequently, many research proposals exist for techniques computing good object placements depending on the application profile. However, clustering tool is only one of many necessary to provide for on-line database reorganization. In this paper we have shown how to approach reorganization of object database on-line by means of dynamic reclustering. Our method is effective in *I*/*O* and interruptible without loss of work. This approach is incremental, therefore system performance improves as reorganization progresses. Due to the fact that our method operates on-line, it is not necessary to reserve space for a new full copy of the database for reorganization. Since we want to keep main operational data structures in main memory, it is essential to be able to reconstruct them after system failure—we do just that by the integration of an area of main memory, the reliable memory into a database system. This has also a positive ef-

† School of Computing, University of Western Sydney
†† Department of Computers and Systems Engineering, Tokyo Denki University

fect in substantially redusing I/O traffic, which results in improvements to the overall performance of the DBMS. Our extensive fault tests show that mapping reliable memory into the database memory space does not substantially hurt reliability.

To summarize, the new aspects of our approach to on-line reorganization of the database include the following research contributions. The novel approach to incremental on-line maintenance of good clustering which employs the maintenance of the minimum spanning tree of a dynamic object graph under the presence of concurrent additions and deletions of data objects. We designed and implemented the Large Complex Object Manager (LCOM) on top of the VERSANT disk manager, for managing variable-size storage clusters. During operation our system is incrementally saving all work already done while maintaining consistency after system crashes.

The rest of this paper is organized as follows. Related work on on-line reorganization of the DBMS is discussed in section 2. Section 3 introduces the VERSANT DBMS, the storage system used in our experiments and the way statistical profile of user application software is captured by the mechanism that, actively interface an Object Database Management System to application programs. Section 4 describes new algorithms designed for minimizing object access time under the presence of concurrent additions and deletions of objects. The application of reliable memory for the purpose of enhancing reliability and performance is demonstrated in section 5. In this section we also discuss concurrency control and recoverability under the operation of the reliable memory and its possible effect on the state-of-the-art recovery schemes (e.g., ARIES[11)]). In section 6 we discuss the question of *optimal cluster size*. Section 7 presents preliminary experiments and results. Section 8 concludes the paper with a summary and a short discussion of future research.

2. Related Works

Though there has been much work in the area of on-line reorganization in recent years, there has been hardly any work that considers on-line incremental cluster modification under concurrent additions and delitions of data. Perhaps Ref. 12) is one of the few research papers that discusses an approach for on-line index modification. In Ref. 12) records are moved from one location to another location one at a time. All references and the record move operations are encapsulated in a database transaction. However this technique is too slow for massive reorganization of DBMS, because changes to the secondary index pages are performed immediately which would necessitate a disk I/O for each index leaf page that is not in the buffer at the time of the access. In Ref. 13), an online algorithm for the reorganization of key sequenced files is presented. Here approach used to move a chunk of data from one disk to another is to make a copy of the data on the destination disk. However, updates are allowed on only one copy, and at the correct time, a switch is performed synchronously to transfer the updates to the new copy. A method for reorganization together with an algorithm specifically tailored for restoration of clustering is presented in Ref. 14). This clustering approach sorts the records of the file in ascending order of the composed key. The composed key $K_1, K_2, \cdots, K_k$ is selected such that the attributes appear in nonincreasing order of their probability of appearence in a query. Since sorting is computionally expensive, i.e., producing a total ordering of all records, they apply a partial-sort approach that restricts the sorting to those records that can fit into the buffer. However, experimental evidence has shown that the partial-sort clustering method performs very poorly.

A clustering algorithm may be either static or dynamic depending upon the time at which the clustering of objects is performed.

- A static clustering scheme, does not recluster objects after they are created. Although initially this offers a good placement policy for newly created complex objects, it does not consider the dynamic evolution of objects.
- Dynamic clustering is done during normal database operation when objects are accessed concurrently by user application packages. However, dynamic clustering can generate additional workload on the database system, so it is important to identify when reorganization should be done. If the reorganization is not properly managed, reclustering may degrade the overall system performance.

Some researchers propose a cost model to evaluate the benefit and cost of reclustering. A fully dynamic reclustering scheme will reduce the access time, and hence the overhead imposed on the system by reorganization could offset the benefit. Our estimation of reorganization cost is based on copy-and-reclaim method. Reorganizing clusters of objects saved on physical pages is justified only if reclustering overhead (T_{reorg}) together with the time taken to write all pages back to the disk (T_{clus}) is smaller than the summary time of present accesses to the related pages residing in the scattered form (T_{scat}).

$$T_{reorg} + T_{clus} < T_{scat}$$

In principle, it is not beneficial to perform reclustering when access frequencies are small, since the reorganization costs taken to reshuffle scattered pages on the disk can be prohibitive. In case that there is no contiguous space available on the disk, a total reorganization will be necessary. Many research suggestions have been made to consider reclustering of objects in ODBMS; however the proposed solutions are not practically dynamic. In reality, they only control placement of objects when objects are modified.

Based on our research, we believe that dynamic reorganization is important and necessary while at the same time somewhat dangerous since disk space reorganization may take prohibitive costs to stabilize and prove too costly. Before reclustering, costs and benefits must be precisely estimated. However, to collect the whole set of statistics for every object, especially during the whole object life may prove to be very expensive and quite unrealistic.

The task of clustering in an on-line, interruptible and adaptive manner has largely been ignored. The majority of known cluster analysis algorithms in the literature are static and, therefore, cannot adapt to changes in usage patterns[6)]. None of the adaptive algorithms address the problem of performing reclustering on-line. Our research is concerned with creating a cluster analysis algorithm that is capable of adapting variable size complex object placement in the scenario of changing usage patterns by different access methods. To keep additional workload introduced due to reorganization as low as possible, we perform cluster analysis only on those objects that have been accessed since the last reorganization and which are already in main memory; to support that we make use of Versant ODBMS event notification mechanism.

3. The VERSANT Storage System

We use the VERSANT database management system developed at *Versant Object Technology Corporation*, *California* as the database in our experiments[10)]. VERSANT is a fourth-generation DBMS combining the direct modeling of complex graph-structured data with the power of today's leading object programming languages. VERSANT has certain unique features which are relevant to our experiments, but our results should apply to more conventional databases as well[4)].

Optimal clustering is achieved only when data relevant to a particular query is collocated physically on disk. Since a particular data element can only reside in a single place within the database, it seems unlikely that the physical location of that element will be optimal for all patterns of access. This behavior is reinforced by the current lack of adaptable buffering strategies in particular buffer replacement strategies. As a direct effect, some likely to be used object can be swapped out because the buffer replacement scheme does not consider that referenced objects are strongly related to currently-used objects. In order to address these issues related to clustering we designed and implemented new clustering algorithms.

3.1 Monitoring Objects

Statistical profile of user application software is captured by the mechanism, which actively interfaces an Object Database Management System to application programs. The database monitor mechanism observes which and how values of attributes of database objects change as a function of time. By means of Versant SQL we can specify the derived or stored associated attribute. The DBMS signals that the state of derived data has changed by invoking tracking procedures, which is a procedure triggered by the changes in a state of the database. Possible tasks for tracking procedures include counting the number of references between objects, or passing information to other systems. Optimization is done by minimizing the amount of monitored data and by allowing monitors to be active only while applications need them. The client localization of monitors can be done by associating monitors with client processes. When no client needs them the system deactivates monitors. The application may run on a client workstation other than the database server. The system keeps track of which application processes on which workstations monitor which data, and automatically notifies application processes when monitored data change. Calling tracking procedures performs notification. External calls may interrupt client processes and execute asynchronously, enabling normal operations, without the ODBMS waiting for the tracking procedures to terminate.

4. Algorithms Designed for Automatic Database Reorganization

In this section, we present the algorithm without the details of logging and recovery.

Database accesses in an Object DBMS are very complex due to the rich variety of type constructors provided. Additionally ODBMS often use navigation-like access among object hierarchy, therefore inter-object references often generate random disk access if the entire database does not fit in main memory. Dynamical hierarchical clustering, where records remain clustered in a hierarchical order after numerous insertions and deletions, has generated very little publicity, and remains an unsolved problem.

4.1 Reorganization Strategy Offered by Commercial Object Database Systems

Usually placement of objects on disk depends on control information given by the database administrator describing the placement of objects. Often physical placement of object on disk is bounded to schema information. Strictly speaking, the placement strategy that has been implemented in existing commercial systems aims at determining a good initial placement of an object, but does not take into account dynamic evolution. The same is relevant to the policy of supplying user hints, which aims only at good initial placement of objects.

Persistence is implemented by associating a reference count with each object. When an object becomes persistent, so do all of its components; conversly, when the reference count of an object drops to 0, the reference counter of each of its subcomponents is decreased. A reference count strategy was adopted to implement the reachability-based persistence scheme. Other widely known techniques, such as scavenging and mark-and-sweep, add some overhead to database performance. Marking schemes force the database to be quiscent for long periods of time because of the amount of data that have to be tested.

Scavenging on disk brings a high performance overhead because of the I/O costs involved in copying and clustering on disk is compromised. In our case, clustering information is collected dynamically during database operation and promotion is delayed until transaction commit time. This is because at update time the object manager has only partial and incomplete view of the graph of objects.

4.2 Frequency of Access and Reachibility Index in an OODB

Our method does not assume any prior knowledge of user queries and their frequencies, except that the object graph structure is available in main memory.

The clustering problem is closely related to the partitioning problem in which a hypergraph is to be partitioned into several disjunct subgraphs.

We used the cost analysis in order to derive automatically an optimal object placement graph. The naive approach, which consists in building all the possible subgraphs and computing their cost functions on-line and then selecting optimal, is not tractable. Such a combinatorial structure is exponential with respect to the number of edges covered by the subgraphs related to the particular methods.

Instead we used a minimum spanning tree, computed dynamically from augmented object graph. We make use of a *path-method* (*pm*) which is a mechanism to retrieve information relevant to one class, in an object-oriented database (OODB), that is not stored with that class but with some other class. For more information on *path-method* see Ref. 31). Our algorithm uses numerical *reacheability index* between pairs of classes as a guide for the traversal of an object graph.

Our method is greatly improved by assigning a *reachibility index* value from the range [0, 1] to each edge in the object graph. The sum of the weights—(D_i) on the n outgoing connections of a node (class) conforms to the following constraint: $\sum_{i=1}^{n} D_i = 0.5 * n$. From this formula's sum value, each connection is assigned a weight from the interval [0, 1], reflecting its relative frequency of traversal. We combined *frequency of access* with *reachibility index.* The *reacheability index* of a connection from a class c_{alpha} to a class c_{beta} is a measure of its significance according to the frequency of traversing this connection relative to all connections involving class c_{alpha}. To express it in a different way; *reacheability index* characterizes the strength of the connection between two classes. The *reacheability index* values are assigned relevant to the frequencies of use of the connections accumulated during the operation of the database.

The importance of a path is measured by the *Access Relevance Index* $ARI(P)$. To compute various *access relevance* values we utilized approach used in Ref. 31). We can obtain the $ARI(P)$ of a path P by applying a triangular-norm (*t-norm*) to the set of *reacheability indexes* of the edges of the path. We have chosen the *t-norm* $PRODUCT$, for which the $ARI(P)$ of a path P is the product of the *reacheability indexes* of all its edges. To emphasize the fact that we are considering access weights and not probabilities, we prefer to use the term "weighting function" (D_F) instead of the term *t-norm*. By applying techniques commonly used with fuzzy sets, we will use a co-norm to select one of the paths connecting the pair of nodes c_α and c_β. In particular, we will use the $MAXIMUM$ co-norm. To compute the access relevance-AR from c_α to c_β, we are applying the $MAXIMUM$ function to the access relevance indexes-ARI of all paths from c_α to c_β.

Definition 1. The most significant connection from c_α to c_β is the path P with the maximum $ARI(P)$.

<u>*Theorem*</u>: The $ARI(P)$ of the most significant path P from c_α to c_β is equivalent to the $AR(c_\alpha, c_\beta)$.

<u>*Proof*</u>

To choose between different paths, we are using the $MAXIMUM$ co-norm. Therefore the path P that will be selected to define the $AR(c_\alpha, c_\beta)$ value is the one with the maximum $ARI(P)$. □

The generation of joins in relational databases parallels the problem of *pm* generation. One approach to this problem is the universal schema interface[30] and the other is the generation of implicit join. However, there is a fundamental difference between *pm* presented in this paper and the generations of joins in both methods described above. Execution of the generated *pm* requires just the fast traversal of the connections defining the *path-method*. Usually, however joins are used to gather information stored in different relations—which may be huge. Normally joins require a large overhead for deriving query-results.

Table 1 The meaning of various coefficients.

P	path.
c_x	(x)-node of the object graph.
ARI	access relevance index.
$ARI(P)$	the access relevance index
	of a path P.
D	an access weight of a connection
	in the OODB, where $0 \le D \le 1$.
AR	the access relevance–a number
	reflecting the strength
	of the strongest
	path between two nodes.
D_F	weighting function expressing
	the strength of connection
	between two nodes.
trb_{c_x}	attribute of the class (node) c_x.

Definition 2. The *Access relevance* $AR(c_\alpha, c_\beta)$ from a node c_α to a node c_β is a number describing the strength of the strongest path between those two nodes, including into account all the paths from c_α to c_β.

We can describe formally, that if a single path $P(c_\alpha, c_\beta) = c_\alpha(= c_{i_1}), c_{i_2}, c_{i_3}, \ldots, c_{i_k}(= c_t)$ connects two nodes c_{alpha}, c_{beta}, then the access relevance index of P is computed by applying the following formula:

$$\sum_{e \in \delta(V_1, V_2, \ldots, V_m)} d_e$$

The *access relevance* of this pair of nodes, if there are m paths $P_1, \ldots, P_m$ from c_α to c_β, is expressed by the following formula:

$$AR(c_\alpha, c_\beta) = \max_{j=1}^{m} ARI(P_j)$$
$$= \max_{j=1}^{m} \prod_{(c_{i_r}, c_{i_{r+1}}) \in P_j} [D(c_{i_r}, c_{i_{r+1}})]$$

By definition $AR(c_\alpha, c_\alpha) = 1$. To derive a path with the highest product of *reacheability indices* of all its edges, we have to maximize the $PRODUCT$ weighting function. **Table 1** summarizes the coefficients.

In what follows, we assume that, using an efficient algorithm, access relevances-(AR) for OODB are already computed and stored. We shall also assume that all edge weights are nonnegative. (If not, we can add a positive value to each edge weight to give an equivalent problem with all edge weights nonnegative.)

4.3 The Algorithm for Maintaining Minimum Spanning Trees in Dynamic Graphs

The basic idea of our dynamic clustering method is to maintain a *minimum spanning tree*, derived from weighted object graph subjected to objects insertions, deletions and updates. For object instances of nodes shared by several edges, the edge with the highest weight is selected. The object subgraph is kept in primary memory. Once a minimum spanning tree is built, objects can be clustered dynamically and automatically according to the frequency of references between them. The object collection need not already exist, but can grow from any number of insertions. Automatic clustering algorithms, which are scarcely found in the research literature, focus usually on reorganizing an existing database in order to match most common access patterns. Those algorithms allow object database clustering to degrade over time and then to employ frequent reclustering to improve performance. In contrast, our research reported here focuses on ensuring good initial placement followed by dynamic maintenance of clusters, in the hope of making static reorganization unnecessary. In case of frequent changes of user's access patterns our technique based on *reachibility index* values continuously reflect those changes in the augmented object graph.

The application of combinatorial structures to the theory of databases, motivated us to study the node capacitated graph partitioning problem. Formally, given a graph $G = (V, E)$, an integer $\mathcal{L} \in \mathcal{N}$, edge weights d_e for $e \in E$, node weights c_i for $i \in V$ and a capacity F, the problem is to find a partition $(V_1, V_2, \ldots, V_m)$ of V with $m \leq \mathcal{L}$, $\sum_{i \in V_j} c_i \leq F$ for $j = 1, \ldots, m$ and such that

$$\sum_{e \in \delta(V_1, V_2, \ldots, V_m)} d_e$$

is minimized, where $\delta(V_1, V_2, \ldots, V_m)$ denotes the set of edges whose end nodes belong to different elements of the partition, typically called a *multicut*. When $m = 2$, we also use the notation $\delta(V_1, V_2)$=$\{e = (i, j) : i \in V_1, j \in V_2\}$ even if $V_1 \cup V_2 \subset V$. For convenience, we introduce the term *cluster*. A cluster is a subset of V (possibly empty). A *cluster partition* $(V_1, V_2, \ldots, V_{\mathcal{L}})$ into $\mathcal{L}$ disjoint clusters ($V_i \cap V_j = \emptyset$ for all i different from j, $\cup_{i=1}^{\mathcal{L}} V_i = V$, $V_i \subseteq V$ for all i) correspond to a partition $(V_1, V_2, \ldots, V_m)$ of V, $1 \leqslant m \leqslant \mathcal{L}$, where $V_1, V_2, \ldots, V_m$ are the nonempty clusters in the cluster partition.

Throughout this paper, we shall wish to deal with graphs that have maximum vertex degree 3. Let us now recall from Ref. 32) how to transform our graph into a graph in which every vertex has degree no grater than three. A well-known transformation in graph theory is used. By ∞ we designate a sufficiently large value, say equal to the largest value that can be represented in a single word of memory. For each vertex V of degree $d > 3$ and neighbors $w_0, w_1, \ldots, w_{d-1}$, replace V with new vertices $V_0, V_1, \ldots, V_{d-1}$. Add edges $\{(V_i, V_{i+1}) \mid i = 0, \ldots, d-2\}$, each of cost $-\infty$, and edge (V_{d-1}, V_0) of cost ∞, and replace edges $\{(w_i, V) \mid i = 0, \ldots, d-1\}$ with $\{(w_i, V_i) \mid i = 0, \ldots, d-1\}$, of corresponding costs. The resulting minimum spanning tree for the transformed graph will be a minimum spanning tree for the original graph with every edge of cost $-\infty$ added. (The value $-\infty$ is used to ensure that these edges are not swapped out when identifying a best swap. For the purpose of identifying the cost of the minimum spanning tree in the original graph, treat the $-\infty$ as 0.)

We next define some terms that serve as the foundation for data structures from Ref. 32) that we wish to use. Let $G = (V, E)$ be a connected undirected graph with maximum vertex degree at most 3, and let MST-Minimum Spanning Tree be a subgraph of G that is a tree. A *vertex cluster* with respect to MST is a set of vertices such that the subgraph of MST induced on the cluster is connected.

Now, we will present our first design of the clustering algorithm as follows: a clustering algorithm takes as input a random sequence of nodes of a configuration Directed Acyclic Graph (DAG), and generates a clustering sequence of nodes. Our approach is to generate and cluster a minimum spanning tree of the nodes of a given object graph configuration.

Our strategy for clustering objects is mainly based on the estimated inter-object communication volume represented by the weight between pairs of objects. In order to keep all clusters within a reasonable size, we impose a limit on cluster size: P_{max}, which is a page size. When the total size of objects of P_x and P_y is less than P_{max}; $\{P_x + P_y \leq P_{max}\}$, they can be clustered together by moving objects from P_y to P_x. The result of this clustering process is a set of pages $P_1, P_2, \ldots$, each of which is a set of o_i objects, $\{1 \leq o_i \leq N\}$, and the size of each page, P_i, satisfies the condition

$\{1 \le P - i \le P_{max}\}$.

Let N be the number of objects in the object net of application software system; w_{v_1,v_2} is the weight representing totality of references between objects v_1 and v_2.

In order to minimize intercluster connectivity and maximize concurrency, the objective function can be expressed as follows:

$$IR = \min \left\{ \sum_{i}^{N-1} \sum_{j}^{N} \sum_{k}^{P} \; w_{v_1,v_2}\, \lambda_{ik}(1 - \lambda_{jk}) \right\}$$

where:

$$\lambda_{ik} = \begin{cases} 1 & \text{if object is clustered on page } k \\ 0 & \text{otherwise} \end{cases}$$
$$\{(1 \le i \le N-1) \ \text{ and } \ (1 \le k \le P)\}$$

In the description of the first algorithm the following notations are used, see also Ref. 4). The edges of G are stored in the list E, i.e., $E(i)$ is the pair of the end vertices of the *i-th* edge. The list W contains the edge weights: $W(i)$ is the weight of the *i-th* edge. ET is the set of edges of the current forest T, p is the number of its components, E_1 is the set of minimum-weight edges for the current forest T. In order to improve the performance of the construction of the minimum-weight edge sets, the following scratchpad, auxiliary work lists are used: $COMP(j)$ is the index of the component of the current forest which contains the vertex j; $MWE(i)$ is the index of the minimum-weight edge for the *i-th* component of the growing forest; $MW(i)$ is the weight of the edge $MWE(i)$.

Algorithm Cluster Object Net

```
Input:   Weighted, undirected graph G
                   with n vertices and edge list E
Output: Clustering Mapping
Compute MST (Min.WeightSpan.Tree)
{
  1. Initialization step: Set $ET \leftarrow \emptyset$,
     $COMP(i) \leftarrow i, MW(i) \leftarrow \infty$ for $i = 1,$
     $2, \ldots, n. p \leftarrow n$.
       //Operations $2-8$ gradually build-
       //up the set $E_1$ of minimum-
       //weight edges for the forest $T$.
  2. $k \leftarrow 1$.
  3. Let $E(k) = uv; i \leftarrow COMP(u)$,
                              $j \leftarrow COMP(v)$.
  4. If $i \ne j$ then go to step 5,
                    otherwise go to step 7.
  5. If $w(uv) = W(k) < MW(j)$ then
          $MW(j) \leftarrow w(uv)$,
          $MWE(j) = k$.
  6. If $w(uv) = W(k) < MW(i)$ then
          $MW(i) \leftarrow w(uv)$,
          $MWE(i) = k$.
  7. If $k = |EG|$ then go to step 8,
          otherwise k ← k + 1
          and go to step 3.
        //Immediately before the
        //execution of step 8, the first$p$
        //entries of $MWE$ contain the
        //indices of edges from the
        //minimum-weight set for $T$
  8. Examine the first $p$ elements of $MWE$
     and create the set $E_1$ of the minimum -
     weight edges for the forest $T$.
  9. $ET \leftarrow ET \cup E_1$.
        //This is the edge set
        //for the "new" forest T'
  10. Using the depth-first search, extract
        the connected components of the
        "new" forest $T' = T' \cup E_1$.
        //The list $COMP$ and the
          value
        //of $p$ are updated
  11. IF $p = 1$ then terminate
        //$ET$ is the edge set of the
        //minimum spanning tree
          otherwise go to step 2.
  RETURN $M_{ST} = \{ET\}$
}
```

At the first iteration the algorithm handles the spanning forest of a graph G consisting of $n = |G|$ single-vertex components. Each iteration acts as follows. First, it constructs a set M of minimum-weight edges for a forest T produced before this iteration.

This can be done during a single scan of the set EG. Let e be the edge to be examined next: if e induces a cycle in M, then e is discarded. Otherwise, edge e is chosen and the new forest will be $M \in \{e\}$. Then using the depth-first search it extracts the connected components of the graph $T' = T + M$, which is a forest. This forest is passed to the next iteration; clearly, T' has fewer components than T.

Theorem:

The algorithm ***Cluster Object Net*** *constructs a minimal spanning tree in time* $\theta(|EG| \log_2 |G|)$.

Proof

It is easy to see that after each iteration of

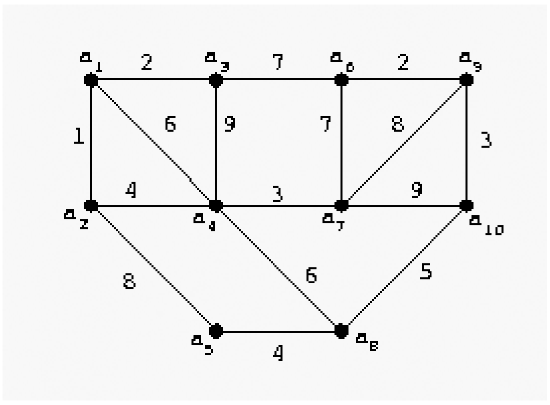

Fig. 1 Graph example.

Step 8; E_1 is the set of minimum-weight edges for the current forest T. Therefore $T' = T + E_1$ is a spannig forest. This means that the algorithm indeed constructs a minimal spanning tree of G [5]. Now, let us estimate the time complexity of the algorithm. A single iteration of Steps 3–6 takes $\theta(|EG|)$. This time is also sufficient for Steps 8–11. Thus, the transition from T to $T' = T + E_1$; a single iteration takes time $\theta(|EG|)$.

Now we can estimate the number of iterations of the algorithm. Since an edge may be minimal at most for two components of T, at each iteration $|E_1| \geq p(T)/2$. And since $T + E_1$ is a forest, $p(T+E_1) \leq p(T)/2$; at each iteration the number of components decreases at least twice. This means tht the number of iterations of the algorithm is at most $\theta(\log_2 |G|)$, hence the algorithm constructs a minimal spanning tree in time $\theta(|EG| \log_2 |G|)$. □

Let us apply algorithm ***Cluster Object Net*** to the graph in **Fig. 1**. At the first iteration we shall obtain the set $E_1 = \{a_1a_2, a_1a_3, a_4a_7, a_5a_8, a_6a_9, a_9a_{10}\}$ of minium weight for the forest of single-vertex components. The spanning forests obtained at the first, second and the third iterations are shown in **Fig. 2**. The last of them is the minimal spanning tree.

To speed up this computation, the above minimum spanning tree algorithm is augmented with Boruvka algorithm [2]. The basic idea in Boruvka's algorithm is to contract simultaneously the minimum weight edges incident on each of the vertices in G. Boruvka's algorithm thus reduces the MST problem in a $|G|$—vertex graph with $|EG|$ edges to the MST problem in a $(|G/2|)$—vertex graph with at most $|EG|$ edges. Our final design uses randomization in conjunction with Boruvka's algorithm, which further improves its computational speed. First, MST computes in $\theta(E)$ time the B-th smallest edge t in G with a fast randomized algorithm for selection. The algorithm will recourse on subgraphs that are not necessarily connected. When the input graph G is not connected, a spanning tree does not exist and we generalize the notion of a minimum spanning tree to that of a minimum spanning forest (MSF).

The only use of randomization in the MST algorithm is in the use of random sampling to identify and eliminate edges that are guaranteed not to belong to the MST. The random-sampling result is the key to the $(\theta|EG|)$ bound. Next, it computes in $\theta(E)$ the reduced set of edges $R = \{e \in E \mid cost(e) \leq cost(t)\}$. Consecutively, minimum spanning tree algorithm applies Boruvka's algorithm to the reduced edge set R in time $\theta(B \log B)$. The forest obtained after this step is denoted by M. If M has the same number of connected components as G or M is connected, then M is the minimum spanning tree or forest of G. Otherwise, minimum spanning tree applies Boruvka's algorithm to G.

Minimum spanning tree algorithm has a worst-case running time of $\theta\,(E \log E)$, therefore it is not asymptotically more efficient than Boruvka's algorithm in the worst case. With the support and help of the theory of random graphs[8], the following can be observed: If an appropriate choice of B is made—namely $B = \Omega\,(n \log n)$, in case of random graphs the probability that minimum spanning tree algorithm applies Boruvka's algorithm to the entire graph G is exceptionally low. As a consequence, the choice of particular $B = \Theta\,(n \log n)$ generates an average running time of $\theta\,(E{+}n : \log^2 n)$ for minimum spanning tree (MST) algorithm.

We now describe a simple fully dynamic minimum spanning tree algorithm (DMST). DMST is an appropriate implementation of a partially dynamic data structure, which is itself a combination of the linking and cutting trees introduced by Tarjan and Sleator and our MST algorithm. During its operations, DMST maintains two data structures. It stores all the edges of graph G in a priority queue Q according to their cost and it also maintains the minimum spanning tree T of the graph G as a linking and cutting tree.

In case when a new edge e is inserted into object graph G, DMST updates T and Q in time $\theta\,(\log n)$. In case when an edge e is deleted

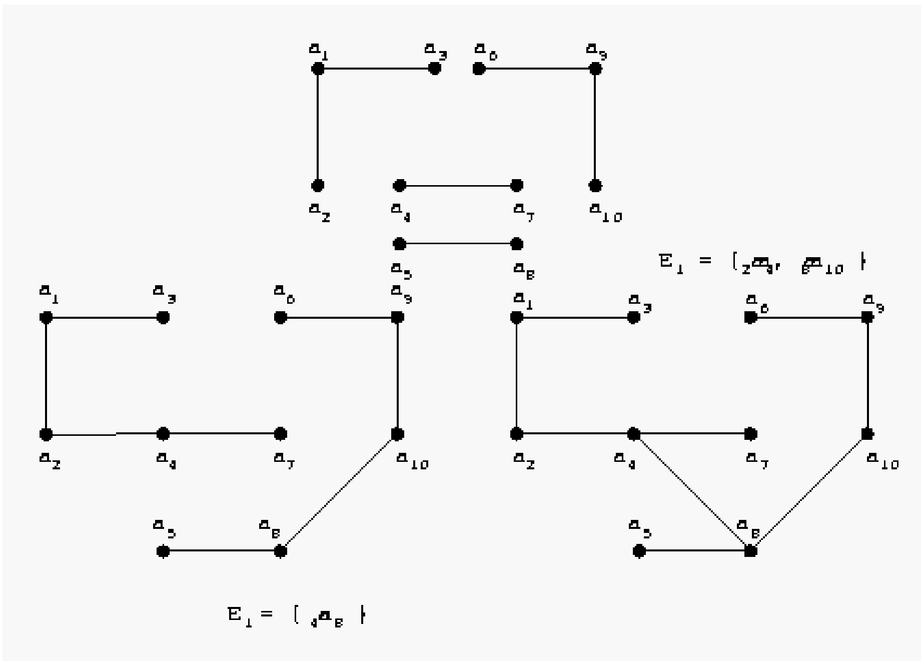

Fig. 2 Applying the algorithm ***Cluster Object Net*** to the ***Graph Example***.

from graph G and it is a non-tree edge, DMST does nothing else - it only deletes e from Q in $\theta\,(\log n)$ time. In case when tree edge is deleted from Q, DMST calls minimum spanning tree (MST) algorithm on the edges in Q. As a result of those operations DMST requires $\theta\,(\log n)$ time plus the running time of MST in case of a tree edge deletion. The expensive case happens with *probability* n/m, if the edge to be deleted is chosen uniformly at random from object graph G, resulting in an average running time of $\theta\,\{\log n + (n/m) \times (m + n\log^2 n)\} = \theta\,\{n + (n\log n)^2/m\}$ for DMST algorithm.

The above analysis suggests that running time of DMST will decrease as the graph density increases—this actually coincides with the results of our experiments. The algorithm exercises a deferred and incremental approach to make the changes in the cluster pages already in main memory. Before the pages from the buffer are flushed out, DMST cluster algorithm updates object graph immediately. However, the pending changes are stored in special memory resident tables. The buffer manager consults those tables when fresh cluster pages are brought into the buffer by a new user request. Although there is a finite number of feasible read sequences, it is not known if there is an efficient algorithm for computing the optimum sequence with minimum disk seek time. Trying to read all the sequences in order to fined the optimum sequence is computationally very expensive. As a result of its prohibitive expense, research has been focusing on using heuristics.

5. Application of Reliable Memory for Databases

Safe *RAM* allows systems that support reliable updates, such as databse and transaction processing systems, to perform more efficiently. Significant improvement in response time can always be realized, and a throughput improvement can be realized by limiting disk access due to the use of the safe *RAM* 15). We make use of an area of main memory, maintained by the operating system, that buffers file system data. Our design uses a memory interface to reliable memory with virtual memory protection to protect against wild stores to dirty, commited buffers–this approach was suggested in Ref. 16). In our design we used reliable memory to store the log. Keeping the log in reliable memory removes all synchronous disk writes from the critical path of a transaction 15). This technique decreases transaction commit time and can help to reduce lock contention with the resulting benefit of increased concurrency. By storing important system information in the stable memory we can help to improve recovery time. We did achieve reduction of time needed to scan the log to find the last checkpoint by storing a pointer in reliable mamory. In our experiments we used the reliable memory to store the database buffer cache. Theoretically this makes all buffer cache changes permanent without writing to disk. Similar to the force-at-commit policy, this eliminates the need for checkpoints and a redo log in recovering system crashes—partial redo[17),18)]. According to Ref. 19), the redo log is still necessary in recovering from media failures—global redo; however, redundant disk storage makes this scenario less probable. Because redundant undo log records can be purged after a transaction commits, removing the redo log has a result that no log records need be written if memory has enough capacity to contain the undo records for all current transactions in progress[20),21)]. Also, according to Ref. 20), storing the databse buffer cache in reliable memory allows the system to begin operation after a crash with the contents of a warm cache, see also Ref. 22). In summary reliable memory also allows state-of-the-art recovery schemes e.g., ARIES family of protocols[11)] to become much more simpler. Since, for example, the need for fuzzy checkpoints is reduced because transactions can commit faster.

We also present a new solution to an old, yet very important problem known as the "on-line object placement problem", which can be stated more precisely as the problem of choosing a disk page to hold a newly allocated object. To the best of our knowledge, the database literature contains no studies of object allocation with the objective to optimize storage utilization and placement performance, with the exception of excellent treatment of the subject in Ref. 23). Since our approach provides for incremental on-line clustering which allows also concurrent updates to data, the object allocation problem is solved by our clustering algorithm as described in Section 4 above.

5.1 Buffer Policies

In the *shared disks scheme*, each site broadcasts its updates to other sites. However, a broadcast is required only if the transaction updates one or more pages.

Table 2 summarizes the parameters of our system.

The probability that transaction requires a broadcast, p_{bdcst} is given by:

$$p_{bdcst} = 1 - \prod_{i=1}^{D_P} (1 - p_{u_i})^{T\psi_i}$$

Table 2 The parameters of our system.

p_{bdcst}	the probability that transaction requires a broadcast.
D_P	number of database partitions.
p_{u_i}	probability of update for pages of partition i.
T	number of pages accessed per transaction.
ψ_i	probability of accessing i_{th} partition.

In our experiments we validated our analysis of buffer models and integrated system models for the response times using the Distributed Database VERSANT.

In our experiments we used the reliable memory to store the database buffer cache. Theoretically this makes all buffer cache changes permanent without writing to disk. Similar to the force-at-commit policy, this eliminates the need for checkpoints and a redo log in recovering system crashes–partial redo[17),18)]. According to Ref. 19), the redo log is still necessary in recovering from media failures—global redo; however, redundant disk storage makes this scenario less probable. Because redundant undo log records can be purged after a transaction commits, removing the redo log has a result that no log records need be written if memory has enough capacity to contain the undo records for all current transactions in progress[20),21)]. Also, according to Ref. 20), storing the databse buffer cache in reliable memory allows the system to begin operation after a crash with the contents of a warm cache, see also Ref. 22).

To integrate buffer management with the recovery model, we guarantee that a modified segment is flushed to the database only after the log records associated with those modifications have been written. Outside of that constraint, the buffer manager is free to use any appropriate buffer replacement policy.

Our recovery algorithm can also be extended to deal with a site failure without performing a complete system restart, so long as the global lock manager data has not been lost, or can be regenerated from the other sites. In other case, a full site failure, as with regular system recovery, has a redo pass, followed by rollback of in-progress operations.

In our implementation, log flushes are triggerd by the release of a lock from a site, in order to support repeating of history and correct rollback of multi-level actions during crash recovery. By comparison "*the super fast method*" of ARIES-SD, does not describe flushes to protect the early release of locks, making it unclear how that scheme supports logical undo and high-concurrency operations.

In summary reliable memory also allows state-of-the-art recovery schemes (e.g., ARIES family of protocols[11]) to become much more simpler. For example, the need for fuzzy checkpoints is reduced because transactions can commit faster.

6. Variable-Size Storage Clusters

Related issue to the "*on-line object placement problem*" is the question of *optimal cluster size*. Our experimental system operates under the assumption that the exchange granularity between disk and main memory is the logical cluster. To group as many objects as possibly, we select those objects which were already stored on disk and group some of the created objects with them. In that way objects already existing on disk are reclustered, which effectively reorganizes ODBMS on-line. Simply using larger fix-size storage clusters is not a solution because of low storage utilization. The enlargement of storage cluster actually brings up countercurrent perfomance issues. On the one hand if we use a small cluster size we get high I/O cost from the large directory structure and the loading of numerous storage cluster and we get high CPU cost from the composition of highly decomposed objects. On the other hand, if we increase the cluster size too much in order to reduce object decomposition and index size, we may end up with increased I/O cost due to the large cluster size and increased CPU cost due to increased cost for the search within the storage cluster and its increased number of objects. To overcome wasted space problem we designed storage clusters with variable size: multi-page clusters with a varying number of pages. When a large object is created it is stored in a sequence of large variable-size segments, each consisting of physically contiguous disk pages. Thus, the segments that comprise the large object may have sizes that vary drastically.

7. Preliminary Experimental Study

The clustering algorithm has been implemented in our experimental system.

We would like to emphasize that at this stage of our research efforts the study limits itself to exploring the feasibility of the techniques we have proposed. In order to test our method for on-line reorganization (specifically, for restoration of clustering), we conducted a series of object clustering experiments. While performing experiments we considered only the time that the algorithms used in recomputing a new solution after each update; we did not measure the time necessary for initializing the data structures and for loading the initial object graph.

For our experiments we used a Pentium Pro 200 MHz based workstation supported by Intel-initiated PCI-ISA bus system, with 128 MB main memory and two disk drives: one 3.2 AT/S gigabyte holding the software—VERSANT ODMS Release 5.0 and 6.4 AT/S gigabyte accommodating the database. Both disk drives model Quantum FireballTM ST. We have adapted hypermodel benchmark[9] to test our clustering algorithms. The benchmark itself comprises operations for:

- *creating* the initial test database with clustering
- *range queries*
- *incremental modifications* of attributes at a number of randomly selected n odes
- *closure operations*
- *reference lookups* (so-called group lookups)

We present in this section only a few experimental results concerning our experiences with the hypermodel benchmark.

We found that the size of workstation buffer pool has no effect on the performance of the sequential scan query and the results are not included.

7.1 Processing of Multiple Queries Concurrently

The main objective in a heavily loaded system is to improve the average response time of queries under a given system throughput. The throughput is established for a given mix of queries by the arrival rate of queries per second. The issue concerning us is to define how the policy of multi-page requests determines the average response time when multiple queries are concurrently processed at a time? This problem is treated in direct relation between: "*multi-page clusters with a varying number of pages*" and "*the citerion that the response time of a query is bounded by its I/O time*". This subsection presents the results of a preliminary performance comparison of multi-page requests which supports multi-page clusters with a varying number of pages and single-page requests when multiple queries are processed concur-

Table 3 Query signature parameters.

QS_F	query signature.
P_T	the number of target pages.
C_F	the number of cylinders occupied by a file F.
ξ	the average number of queries which arrive in one second.
QT_i	i-th query type.
$p\omega_i$	probabilty of a query being of the i-th query type.
P_{T_i}	the number of target pages of quert type $QT - i$.
C_i	the number of file cylinders of query type QT_i.

rently.

A *query signature* is defined by one or several query types. Every query type is characterized by the parameters listed in **Table 3**.

Additionally to the query types, a query signature is characterized by the arrival rate of the queries. In our experiments we assume that the arrivals of queries follow an exponential distribution where ξ designates the average number of queries which arrive in one second time interval. Generally, a query signature QS_F can be described by the following string of parameters:

$$QS_F = (\xi,\ p\omega_1,\ QT_1\,(P_{T_1},\ C_1), \ldots, p\omega_i,\ QT_k\,(N_k,\ C_k))$$

QT_i is the i-th query type which executes stream of queries with probability $p\omega_i$, where $1 \leq i \leq k$.

Firstly, we performed experiments on the following query signatures:

$$QS_{F1}(\xi) = (\xi,\ 1.0,\ QT(15,\ 5))$$

We varied the values of parameter ξ from 1 to 20. **Figure 3** shows the average response time in *ms*. As can be seen from it, when $\xi > 5$ the average response time per query dramatically increases for queries using single-page requests. However, multi-page requests still offer low response time of the queries.

In the next group of experiments, the query signatures are examined where parameter ξ is varying in the range between $1 \leq \xi \leq 30$, each of them comprising four different query types. It is assumed that the query type that will access only one page will occur with probability 60%, however probability of a query that will access 100 target pages is only 10%.

$$QS_{F2}(\xi) = (\xi,\ 0.6,\ QT(1,\ 1),\ 0.1,$$
$$QT(20,\ 5),\ 0.1, QT(50,\ 5),\ 0.2, QT(100,\ 10))$$

We studied important problem related to how the average response times of small queries performes when all queries are executed using

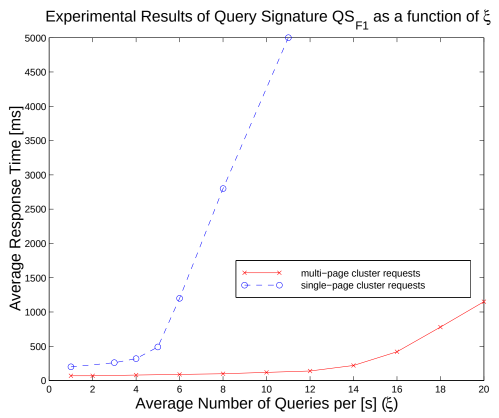

Fig. 3 Experimental results of query signature QS_{F1} as a function of ξ.

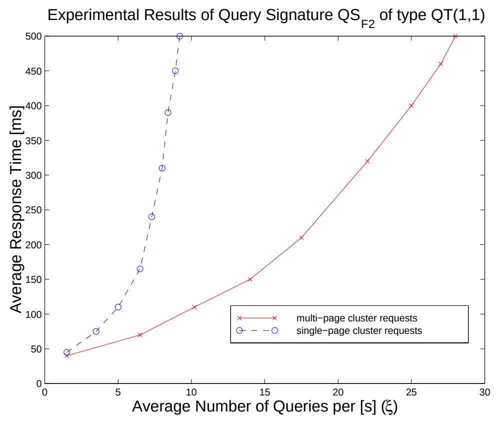

Fig. 4 Experimental results of query signature QS_{F2} of type QT(1,1).

multi-page clusters with a varying number of pages.

We addressed this question for our query signature $QS_{F2}(\xi)$, by plotting the average response time of the queries of type $QT(1,\ 1)$ in **Fig. 4**. The query of type $QT(1,\ 1)$ cannot take advantage from multi-page cluster request. Therefore, response time show similar results on both graphs for a small value of ξ - a lowly loaded system. In case of a heavily loaded system, the average response time of $QT(1,\ 1)$ queries type is distinctly lower when multi-page cluster requests are executed in comparison to using single-page cluster queries.

7.2 Static vs. Dynamic Reclustering

Figure 5 depicts a comparison between the average miss rates as a function of time in case where dynamic, on-line reclustering was active and operational and where it was not.

Reductions in the average miss rate were gen-

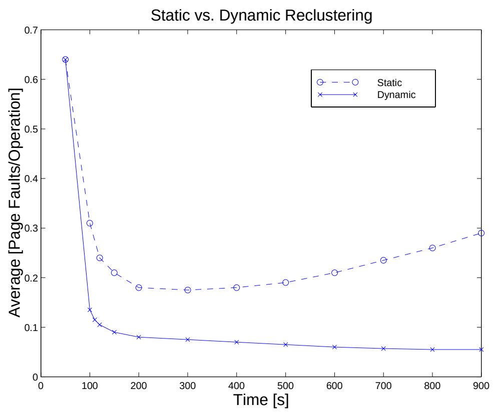

Fig. 5 Static vs. dynamic reclustering.

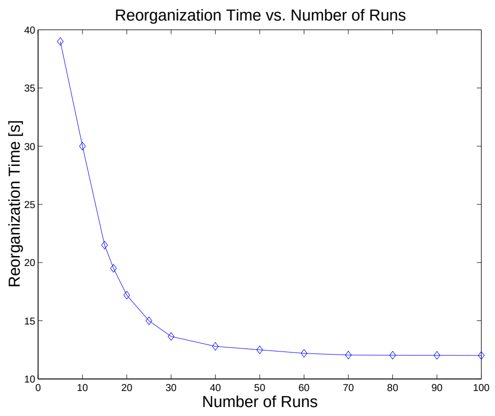

Fig. 6 Reorganization time vs. number of runs.

erally noticed in case when DMST algorithm was active. A decreased miss rate was expected for those user processes that utilized the part of the database after it has been reclustered. Indeed this effect was observed in the miss rate results of our benchmark runs.

Figure 6 shows relationships between the number of reorganizations and the average duration of those reorganizations. With the number of reorganizations increasing during a run, the time interval between them decreases.

This is due to the fact that as the frequency of reorganizations increases during a particular run, the statistics that are gathered to support clustering cover a smaller part of the database, since it makes references to the smaller number of objects. This analysis is also supported by looking at Fig. 4, showing that reorganization duration decreases as the number of reorganizations during a particular run increases. We can deduce (from Fig. 4) that this behavior will increase with concurrency, since user transactions do not need to wait as long to gain access to the necessary database locks.

8. Conclusion and Future Work

We believe that a wide class of applications such as *CAM* and *CASE* systems can greatly benefit form an on-line dynamic reorganization via clustering since they normally use low number of competing concurrent transactions.

Since we want to keep main operational data structures in main memory, it is essential to be able to reconstruct them after system failure—we do this by the integration of an area of main memory—the reliable memory into a database system.This has also a positive effect in substantially reducing I/O traffic, which improves the overall performance of the DBMS. Our tests show that mapping reliable memory into the database memory space does not substantially hurt reliability. Our main objective of incremental on–line reorganization is to change a part of the database without affecting on-going transactions for very long. To perform reorganization more efficiently, the ways to group to be reorganized are examined. Our methodology has advantages over many other proposed or cited approaches because it does not require any beforehand knowledge of the query frequencies, nor does it trigger a reclustering process based on trace analysis. We have also considered making online reorganization restartable. Our method incrementally saves all work already done while maintains consistency after system crashes.

To address the shortcomings of fixed size clusters, we designed variable storage clusters, a new clustering mechanism that supports clusters whose sizes vary dynamically according to the access patterns being observed.

We demonstrated that on-line reclustering is possible, but more research is required to demonstrate its suitability in many practical situations, especially in the presence of greater degrees of concurrently operating transactions.

For our purposes, the value of this experimental work is the comparison of the relative performance of the new algorithms under a range of different operational scenarios. In that sense, the results are used to demonstrate practical feasibility of the proposed approach.

Acknowledgments Also we would like to acknowledge *Intel Corporation* for making their tools available for our experiments. *Microsoft, Windows and Windows NT* are trademarks of

Microsoft Corporation. *VERSANT* is a trademark of *Versant Technology Corporation*. *Intel* and *Pentium* are trademarks of *Intel Corporation*. We wish to acknowledge the anonymous referees for their helpful comments, which led us to an improved presentation of this paper.

References

1) Wietrzyk, V.: Performance Evaluation of New Clustering Algorithms in Object-Oriented Database Systems, *International Conference on Database and Expert Systems Applications*, Zurich, Switzerland (Sep. 1996).

2) Wietrzyk, V. and Orgun, M.A.: A Foundation for High Performance Object Database Systems, *Databases for the Millennium 2000 in Proc. 9th International Conference on Management of Data*, Hyderabad (Dec. 1998).

3) Wietrzyk, V.: Effective Clustering and Data Caching in Client-Serve DBMS Architecture, *Trends in Information Technology in Proc. International Conference on Information Technology*, Bhubaneswar (Dec. 1998).

4) Wietrzyk, V. and Orgun, M.A.: VERSANT Architecture: Supporting High-Performance Object Databases, *International Database Engineering and Applications Symposium, IDEAS98*, Cardiff, U.K., July, IEEE Computer Society Press, Los Alamitos, CA (1998).

5) Wietrzyk, V.I. and Orgun, M.A.: Dynamic reorganization of Object Databases, *International Database Engineering and Applications Symposium, IDEAS99*, Montreal, Canada, August, IEEE Computer Society Press, Los Alamitos, CA (1999).

6) Bancilon, F., Delobel, C. and Kanellakis, P.: *Building an Object-Oriented Database System: The Story of O2*, Morgan Kaufmann, San Mateo, CA (1992).

7) DeWitt, J.D., Maier, D., Futtersack, P. and Velez, F.: A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems, *Proc. 16th VLDB Conference*, Brisbane, Australia (1990).

8) Andrasfai, B.: *Graph Theory*, Techno House, New York, NY, USA (1991).

9) Anderson, T.L., Berre, A.J., Mallison, M., et al.: The Hypermodel Benchmark in Bancilhon, Thanos, Tsichritzis (Eds.), *Advances in Database Technology—EDBT'90*, LNCS 416 (1990).

10) VERSANT System Manual.: VERSANT Release 5.0 (Feb. 1997).

11) Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., et al.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, *ACM Trans. Database Syst.*, Vol.17, No.1, pp.94–162 (1992).

12) Salzberg, B. and Dimock, A.: Principles of transaction-based on-line reorganization, *Proc. 18th Intl. Conf. Very Large Databases*, pp.511–520, San Mateo, CA, Morgan Kaufmann Publishers (Aug. 1992).

13) Smith, G.S.: Online reorganization of key-sequenced tables and files, *Tandem System Review*, Vol.6, No.2, pp.52–59 (1990).

14) Jakobsson, M.: Reducing block accesses in inverted files by partial clustering Inform, *Syst.*, Vol.5, No.1–5 (1980).

15) Copeland, G., Keller, T., Krishnamurthy, R. and Smith, M.: The Case for Safe RAM, *Proc. 15th Intl. Conf. Very Large Databases*, pp.327–335, Amsterdam, The Netherlands, Morgan Kaufmann Publishers (Aug. 1989).

16) Sullivan, M. and Stonebraker, M.: Using write protected data structures to improve software fault tolerance in highly available database management systems, *Proc. 1991 Intl. Conf. Very Large Data Bases* (*VLDB*), pp.171–180 (Sept. 1991).

17) Haerder, T. and Reuter, A.: Principles of transaction-Oriented Database Recovery, *ACM Computing Surveys*, Vol.15, No.4, pp.287–317 (1983).

18) Akyurek, S. and Salem, K.: Management of partially safe buffers, *IEEE Trans. Comput.*, Vol.44, No.3, pp.394–407 (1995).

19) Chen, P.M., Lee, E.K., Gibson, G.A., Katz, R.H. and Patterson, D.A.: RAID: High Performance, Reliable Secondary Storage, *ACM Computing Surveys*, Vol.26, No.2, pp.145–188 (1994).

20) Wee Teck, N.G., Aycock, C.M., Rajmani, G. and Chen, P.M.: Comparing Disk and Memory's resistance to Operating system crashes, *International Symposium on Software Reliability Engineering* (1996).

21) Agrawal, R.A. and Jagadish, H.V.: Recovery Algorithms for database Machines with Nonvolatile main memory, *Database Machines Sixth International Workshop, IWDM'89 Proc.* (June 1989).

22) Sullivan, M. and Chillarege, R.: A Comparison of Software Defects in Database Management Systems and Operating Systems, *Proc. 1992 International Symposium on Fault-Tolerant Computing*, pp.475–484 (July 1992).

23) McAuliffe, M.L., Carey, M.J. and Solomon, M.H.: Towards Effective and Efficient Free Space Management, *Proc. 1996 ACM SIGMOD International Conference on Management of Data*, pp.389–400 (1996).

24) McAuliffe, M.L., Carey, M.J. and Solomon,

M.H.: Vclusters: A Flexible, Fine-Grained Object Clustering Mechanism, *Proc. 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications* (*OOPSLA'98*), Vol.33, No.10 (1998).

25) Dröge, G. and Schek, Hans-Jörg: Query-Adaptive data Space Partitioning using Variable-Size Storage Clusters, *Proc. Third International Symposium, SSD'93*, Singapore (June 1993).

26) Weikum, G.: Set oriented disk access to large complex objects, *Proc. Int. Conf. on Data Engineering, 1989*, pp.426–433 (1989).

27) Hutflesz, A., Six, H.W. and Widmayer, P.: *Proc. Int. Conf. on Data Engineering, 1988*, pp.572–579 (1989).

28) Knuth, D.E.: *The Art of Computer Programming*, Addison-Wesley (1997).

29) Kotch, P.D.: Disk File Allocation Based on The Buddy System, *ACM Trans. Comput. Syst.*, Vol.5, No.4 (1987).

30) Maier, D. and Ullman, J.D.: Maximal Objects and the Semantic of Universal Relation Databases, *ACM Trans. Database Syst.*, Vol.8, No.1, pp.1–14 (1983).

31) Mehta, A., Geller, J., Perl, Y. and Neuhold, E.J.: The OODB Path-Method Generator (PMG) Using Precomputed Access Relevance, *Proc. 2nd Int'l Conference on Information and Knowledge Management*, Washington DC, pp.596–605 (1993).

32) Frederickson. G.N.: Data Structures for On-Line Update of Minimum Spanning Trees, with Applications, *SIAM J. Computing*, Vol.14, No.4, pp.781–798 (1985).

(Received July 19, 2001)
(Accepted October 16, 2001)

Vlad Ingar Wietrzyk obtained his M.Sc. degree from Prague University. EU and his Dip. in Computer Science from UTS, Sydney. Since 1999 he has been at the University of Western Sydney. Since 1997 until 1998 he had been a visiting researcher of Stuttgart and Mannheim Universities. He has publications in national and international conferences and workshops. In 1999 he was a visiting researcher at the Institute of Software Engineering, Montreal University. He has served on the program committees of international conferences like ICPADS, IDEAS, CIT, COMAD, ENTER. He has deliverd industrial seminars on computing to companies like VERSANT and ALCATEL. While at the Analytical Service Corporation, Sydney 1987–1995 he designed and implemented in software, a hierarchical clustering method which was the first to support the analysis of data based on groups and data exploration. His current research interests are: Object Distributed Databases, Various aspects of Information Systems Design Methodologies (including Distributed Systems), Transaction Processing in distributed systems, Concurrency Control, Distributed and Federated Database Systems, and Distributed Workflow Technology supporting Electronic Commerce. He is a member of IEEE and AIEA.

Katsuya Tanaka was born in 1971. He received his B.E. and M.E. degree in Computers and Systems Engineering from Tokyo Denki University, Japan in 1995 and 1997, respectively. From 1997 to 1999, he worked for NTT Data Corporation. Currently, he is an assistant in the Department of Computers and Systems Engineering, Tokyo Denki University. He received the D.E. degree from Dept. of Computers and Systems Engineering, Tokyo Denki University, Japan, in 2000. His research interests include distributed systems, transaction management, recovery protocols, and computer network protocols. He is a member of IEEE CS and IPSJ.

Makoto Takizawa was born in 1950. He received his B.E. and M.E. degrees in Applied Physics from Tohoku Univ., Japan, in 1973 and 1975, respectively. He received his D.E. in Computer Science from Tohoku Univ. in 1983. From 1975 to 1986, he worked for Japan Information Processing Developing Center (JIPDEC) supported by the MITI. He is currently a Professor of the Dept. of Computers and Systems Engineering, Tokyo Denki Univ. since 1986. From 1989 to 1990, he was a visiting professor of the GMD-IPSI, Germany. He is also a regular visiting professor of Keele Univ., England since 1990. He was a program co-char of IEEE ICDCS-18, 1998 and serves on the program committees of many international conferences. He chaired SIGDPS of IPSJ from 1997 to 1999. He is IPSJ fellow. His research interests include communication protocols, group communication, distributed database systems, transaction management, and security. He is a member of IEEE, ACM, and IPSJ.