Online Reorganization of Databases

GARY H. SOCKUT and BALAKRISHNA R. IYER

IBM Silicon Valley Laboratory

In practice, any database management system sometimes needs reorganization, that is, a change in some aspect of the logical and/or physical arrangement of a database. In traditional practice, many types of reorganization have required denying access to a database (taking the database offline) during reorganization. Taking a database offline can be unacceptable for a highly available (24-hour) database, for example, a database serving electronic commerce or armed forces, or for a very large database. A solution is to reorganize online (concurrently with usage of the database, incrementally during users' activities, or interpretively). This article is a tutorial and survey on requirements, issues, and strategies for online reorganization. It analyzes the issues and then presents the strategies, which use the issues. The issues, most of which involve design trade-offs, include use of partitions, the locus of control for the process that reorganizes (a background process or users' activities), reorganization by copying to newly allocated storage (as opposed to reorganizing in place), use of differential files, references to data that has moved, performance, and activation of reorganization. The article surveys online strategies in three categories of reorganization. The first category, maintenance, involves restoring the physical arrangement of data instances without changing the database definition. This category includes restoration of clustering, reorganization of an index, rebalancing of parallel or distributed data, garbage collection for persistent storage, and cleaning (reclamation of space) in a log-structured file system. The second category involves changing the physical database definition; topics include construction of indexes, conversion between B$^+$-trees and linear hash files, and redefinition (e.g., splitting) of partitions. The third category involves changing the logical database definition. Some examples are changing a column's data type, changing the inheritance hierarchy of object classes, and changing a relationship from one-to-many to many-to-many. The survey encompasses both research and commercial implementations, and this article points out several open research topics. As highly available or very large databases continue to become more common and more important in the world economy, the importance of online reorganization is likely to continue growing.

Categories and Subject Descriptors: H.2.7 [**Database Management**]: Database Administration—*Reorganization; maintenance; logging*; H.2.4 [**Database Management**]: Systems—*Concurrency; distributed databases; parallel databases*; C.4 [**Performance of Systems**]: Availability; H.2.2 [**Database Management**]: Physical Design—*Access methods*; H.2.1 [**Database Management**]: Logical Design—*Schema and subschema*; D.2.9 [**Software Engineering**]: Management—*Life cycle*; D.4.9 [**Operating Systems**]: Systems Programs and Utilities; D.4.3 [**Operating Systems**]: File Systems Management—*Access methods; distributed file systems; file organization*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software maintenance*; E.5 [**Files**]: *Organization/structure*; D.4.2 [**Operating Systems**]: Storage Management—*Allocation/deallocation strategies; garbage collection*; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*; C.2.4 [**Computer-Communication**

Authors' addresses: G. H. Sockut (corresponding author), MIT Lincoln Laboratory, 244 Wood Street, Lexington MA 02420, email: ghs@acm.org, URL: http://alum.mit.edu/www/ghs; B. R. Iyer, IBM Silicon Valley Laboratory, 555 Bailey Ave., San Jose, CA 95141; email: balaiyer@us.ibm.com.

DOI 10.1145/1541880.1541881 http://doi.acm.org/10.1145/1541880.1541881

Networks]: Distributed Systems—*Distributed databases*; D.4.1 [**Operating Systems**]: Process Management—*Concurrency*; *scheduling*; *synchronization*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Clustering, concurrent reorganization, indexes, log-structured file systems, maintenance, online reorganization, redefinition, reorganization, restructuring, schema evolution, very large databases

ACM Reference Format:

Sockut, G. H. and Iyer, B. R. 2009. Online reorganization of databases. ACM Comput. Surv. 41, 3, Article 14 (July 2009), 136 pages. DOI = 10.1145/1541880.1541881 http://doi.acm.org/10.1145/1541880.1541881

1. INTRODUCTION

Reorganization of a database is a change in some aspect of the logical and/or physical arrangement of the database. An example of reorganization is restoration of *clustering*, which is the practice of storing instances near each other if they meet certain criteria. Traditionally, during many types of reorganization, the data in the area being reorganized has been *offline* or only partially available; users could not update (and often could not even query) data in that area. Figure 1 shows a possible traditional scheduling of activities for a database. The horizontal dimension represents time. Ordinarily (during the periods marked "OPERATIONS"), users can access the database. During the period marked "MAINTENANCE WINDOW," the database can be offline for procedures like offline reorganization.

However, a highly available database (a database that should be fully available 24 hours per day, 7 days per week whenever possible) should not go offline for significant periods, of course. High availability can shrink the maintenance window below the time required for reorganization, as the left-pointing arrow in Figure 1 shows. Applications that require high availability include electronic commerce, armed forces, utilities (e.g., telecommunication), reservations, finance (especially global finance), process control, hospitals, and police. Even if a database is not highly available, reorganization of a very large database can require much longer than some maximum tolerable offline period, that is, the maintenance window, as the right-pointing arrow at the bottom shows. The maximum tolerable period of unavailability is specific to the application. According to one group of database customers, the maximum tolerable period can range from 0 to 5 hours.

These considerations call for the ability to reorganize a database *online* (concurrently with usage, incrementally during users' activities, or interpretively), so that users can query and update the database during most or all phases of the reorganization. Many people have stated the need for this ability [Schubert 1974, p. 51; Meltzer 1975, p. 367; 1976, p. 19; Silberschatz et al. 1991, p. 117; Keyes 1992; DeWitt and Gray 1992, p. 97; Valduriez 1993, p. 150; Graefe 1993, p. 95; Selinger 1993, p. 672; Mohan 1993b; Bratsberg and Torbjørnsen 2000]. As dependence on computers and the amount of information in databases both grow, the number of highly available or very large databases will grow, so the importance of online reorganization will continue to grow.

1.1. Reorganization in General

Before discussing online reorganization, we briefly discuss reorganization in general, that is, topics that apply to both offline and online reorganization. We have identified three *general categories* of reorganization (maintenance, physical

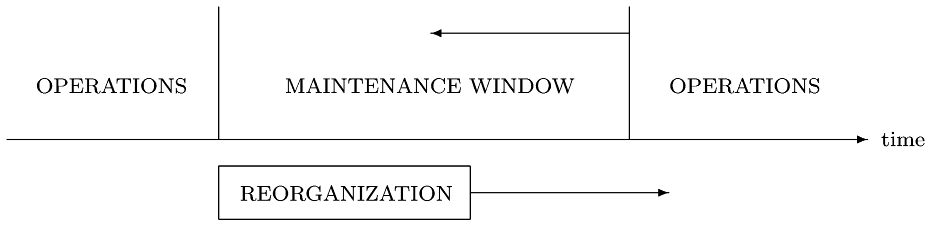

Fig. 1. Scheduling of offline reorganization during a maintenance window.

redefinition, and logical redefinition), each of which can be subdivided into more *specific* categories.

We use *maintenance* to mean restoring the physical arrangement of data instances without changing the database definition. This article usually uses *data instance* to mean a record in persistent storage, not a record in main storage after retrieval. In many storage organizations, users' updates can cause *structural degradation*, which is the physical rearrangement of data instances in a way that causes performance degradation or depletion of storage allocation. *Performance degradation* means an increase in users' response time and/or a decrease in throughput. As an example of structural degradation, fullness of data pages can cause *overflow*, that is, placement of data away from a physical location that would yield efficient access. Examples of maintenance include restoration of clustering (perhaps with removal of overflow) and garbage collection for persistent storage. Maintenance occurs repeatedly (periodically or upon demand).

Physical redefinition means changing the physical database definition, and it typically includes eventual changing of data instances to match the new definition. Examples of physical redefinition include (1) construction, dropping, or renaming of a nonunique index and (2) conversion between indexing and hashing. Over the lifetime of a typical database, several different physical and/or logical redefinitions will be necessary.

If a type of reorganization retains the initial basic physical definition but changes the values of *parameters* of that definition, then the reorganization could fit in either category above (maintenance or physical redefinition). An example is repeatedly increasing the size of storage available for a growing database.

The third category is *logical redefinition* (sometimes called *schema evolution* or *restructuring*), which means changing the logical database definition. Changing a logical definition sometimes requires changing data instances, the physical definition, *view* definitions (specifications of user-visible representations of parts of a database), objects' methods, and application programs. Examples of logical redefinition include adding, deleting, combining, splitting, or renaming columns (or tables), changing the logical representation of a column (e.g., from inches to centimeters), adding, deleting, or renaming a method, changing the inheritance hierarchy of object classes, and changing a relationship from one-to-many to many-to-many. This article sometimes uses terms and examples from specific data models (relational, object-oriented, entity-relationship, hierarchical, and CODASYL), but the concepts from most of our discussions can apply to many data models. If a database needs several types of reorganization (e.g., combining two columns and converting from hashing to indexing), we might perform those types together, to save time. For some types of redefinition, a database management system (*DBMS*) might check the redefinition request immediately but defer the reorganization of data instances until the next maintenance [Friske 2004b].

Sockut and Goldberg [1979] discuss reasons for reorganization in general (e.g., a change in characteristics of usage) and categorize many examples of reorganization.

They describe reorganization strategies, commercial facilities, case studies, and considerations for database administration. Most of the discussions are generic to offline and online reorganization. Many circumstances can make reorganization necessary, and any DBMS can need some type of reorganization.

1.2. Overview of this Article

It is possible to describe techniques and classify the work in online reorganization via two dimensions. One dimension involves specific *categories* of reorganization, for example, (1) restoration of clustering and (2) conversion between indexing and hashing. The other dimension involves *issues*, for example, (1) use of partitions and (2) references to data that has moved, where each issue can apply to several specific categories. This article uses both dimensions. It discusses requirements and issues in reorganizing databases online, and it surveys work on online reorganization. An earlier survey [Sockut and Iyer 1996] discusses only IBM work in online reorganization. An introductory knowledge of database management should suffice for understanding this article.

Section 2 discusses requirements for online reorganization, including applications that require online reorganization and required characteristics of strategies. Section 3 analyzes *issues* that arise in online reorganization. We discuss most of these issues independently of strategies for specific categories of reorganization; the issues can apply to a variety of categories. The issues, most of which involve trade-offs, include use of partitions, the process (locus of control) that reorganizes, reorganization by copying to newly allocated storage, use of differential files, references to data that has moved, general issues in performance, and activation of maintenance. The requirements and issues can be a partial checklist for the design of strategies for online reorganization.

The next three sections describe online strategies for specific *categories* of reorganization within the three more general categories (maintenance, physical redefinition, and logical redefinition). Readers who are interested in only one category can skip the other categories. Our discussions of these strategies use the generic requirements and issues of Sections 2 and 3 and many less generic issues. Because the discussions implicitly include the analyses and trade-offs that the earlier sections present, the generic issues provide a link between strategies for different categories of reorganization, and the issues avoid redundancy among descriptions of different strategies that share a resolution of an issue. The specific categories within maintenance include restoration of clustering, reorganization of an index, rebalancing of parallel or distributed data, garbage collection for persistent storage, and cleaning (reclamation of space) in a log-structured file system. Section 5 discusses physical redefinition, including construction of indexes, conversion between B^+-trees and linear hash files, and redefinition (e.g., splitting) of partitions. Section 6 describes logical redefinition, for example, adding columns and changing the inheritance hierarchy of object classes. Within logical redefinition, the discussions include interpretation and incremental reorganization during users' activities, use of a background process for reorganization, and some additional topics. Many of the strategies for online reorganization have appeared in research contexts, and many have appeared in commercial contexts. For each individual topic (e.g., restoration of clustering or construction of indexes), we concentrate on techniques for online reorganization for that topic. We omit many articles that describe other aspects of the topic (e.g., optimal layouts for clustering or optimal choices of indexes). Section 7 is a summary.

We typically define terms when we first use them, and a glossary briefly restates the definitions of some of the terms that we use frequently.

2. REQUIREMENTS FOR ONLINE REORGANIZATION

We will discuss two aspects of requirements: applications that require online reorganization (in more detail than the introduction sketched) and required characteristics of strategies that perform online reorganization.

2.1. Applications that Require Online Reorganization

During many types of reorganization, traditionally the area being reorganized has been offline, as we stated earlier. Here, we will describe more details of the need for online reorganization.

For some highly available databases, bringing any part of the database offline for a significant period may cause a significant financial loss. Mohan [1993b, p. 288] notes that some organizations, for example, credit bureaus, can appraise the monetary cost of unavailability of a database for a period, since they route activities to competitors during periods of unavailability of their own databases. Ho et al. [1994] cite a survey of computer users, reporting an average cost of $1300 per minute of unavailability of applications. If a database that serves electronic commerce is offline, a customer will not get a quick, positive response from a mouse click. The customer may then turn to a competitor.

Another consequence of unavailability is inconvenience. For example, a hotel conglomerate's reservation database was offline for 17 hours for recompilation of records to a new database format [Radosevich 1993]. During the recompilation, customers calling the central telephone number for reservations were told to call the individual hotels. An executive said that if there were "some way of converting safely without bringing the system down, we would do it."

Details of requirements for availability are specific to each application, and unavailability of some facilities can have serious consequences other than financial loss or inconvenience. For example, the National Crime Information Center (NCIC) of the U.S. government's Federal Bureau of Investigation (FBI) is available 24 hours per day whenever possible. It maintains up-to-date information on crimes, for example, theft of cars. Section 3.3.3 explains how reorganization in a legacy implementation of the NCIC includes copying data between disks while allowing queries and deletions. Allowing just queries is straightforward, but allowing deletions required an extra effort in the design. The justification for the extra effort is prevention of situations like this: A stolen car is recovered during reorganization, the owner drives it away, and shortly thereafter the police see the car and mistakenly arrest the owner because the record of the stolen car has not yet been deleted.

Utilities, process control, hospitals, and armed forces can also have their own requirements for availability, with very serious consequences of unavailability. Even for less essential applications, many *database administrators* (*DBAs*) prefer 24-hour availability.

Even without a preference for 24-hour availability, reorganization of a very large database can require much longer than the maximum tolerable offline period. As examples of very large databases, Mohan [1993b, p. 280] mentions a database containing several terabytes of data and the desire for a database containing petabytes. In the 2005 results of a periodic survey of very large databases, the largest commercial database exceeded 100 terabytes [Winter 2005]. Wiederhold [1983, p. 516] considers offline reorganization such an important problem for very large databases that he *defines* a very large database as "a database whose reorganization by reloading takes a longer time than the users can afford to have the database unavailable."

2.2. Required Characteristics of Strategies

To serve applications that require online reorganization, we have identified several required characteristics of strategies that perform online reorganization, in the areas of correctness, performance, and tolerance of errors [Sockut and Goldberg 1976, pp. 13–14; Sockut 1977, pp. 1.29–1.32].

Users must be able to query and update correctly. Similarly, the actions of reorganization must be correct. Correct operation requires concurrency control (including deadlock resolution).[1] For maintenance performed by a background process, Bastani et al. [1988] develop analysis to assure that algorithms can satisfy invariant conditions and do not introduce errors into users' activities. An example of an invariant condition is that the elements of an array are ordered by their values.

Correctness also implies that reorganization and users' transactions must be able to track the status of each other's actions on data when necessary, for example, to correct references to that data. This tracking can lead to complexity. During online reorganization, a user transaction might perform some low-level actions that it would not perform in a serial execution, for example, following pointers to structures that have moved due to reorganization [Goodman and Shasha 1985; Shasha and Goodman 1988]. Similarly, online reorganization might need actions that offline reorganization does not need, for example, reading users' updates from a log, as described in Section 3.3.3.

The degradation (if any) of users' performance must be tolerable; the maximum tolerable amount of degradation is, of course, application-specific. Section 3.6 discusses considerations for users' performance and reorganization's performance, including tradeoffs between them. Also, our later descriptions of many of the individual strategies for online reorganization include performance issues. The reorganization's consumption of space must also be tolerable.

Progress by reorganization is another performance requirement. If a process (separate from users' activities) executes in the background to reorganize all the data instances, the process must eventually complete its work. For example, if a reorganization process scans records from A through Z (reorganizing as it scans), it must eventually reach the last Z record. As another example, suppose that reorganization includes reading users' saved updates from a log and applying these updates to a reorganized copy of the data. The reorganization's reading of the updates must eventually catch up to the users' writing of them.

For logical or physical redefinition, the above requirement for progress and completion of reorganization is self-explanatory. For maintenance, we have identified these refinements to the requirement:

—For the long term (including the periods of reorganization and any periods of no reorganization), the net rate of reorganization's *decrease* in structural degradation must match the net rate of users' *increase* in structural degradation [Söderlund 1980, p. 115; 1981a, pp. 506–507; 1981b, pp. 27–28]. Without this property, structural degradation and thus performance degradation would increase and eventually become permanently unacceptable.

—For a background process that reorganizes, we define *completion* as one complete pass through the area being reorganized. New structural degradation can occur (before completion) in structures that reorganization has already processed, for example, in the A through J records while reorganization is processing the K records. This new degradation can remain in A through J after completion. The next pass of

[1]For simplicity, this article usually does not distinguish locking from latching.

reorganization can remove this new degradation, although even newer degradation can occur during (and remain after) that next pass. We do not impose a stronger requirement that each pass of reorganization must remove all structural degradation that forms during that pass. We consider this stronger requirement unnecessary if the reorganization satisfies the previous refinement (preventing a long-term increase in the total amount of degradation). Similarly, developers of at least one strategy for online reorganization [Smith 1990] consider the new degradation tolerable. Also, for benchmarks used for an online garbage collector for volatile storage, the size of the new garbage (degradation) ranged only up to 4.1% of the size of the reachable data [Nettles and O'Toole 1993, p. 224].

For tolerance of errors, data must be recoverable during online reorganization. Here are examples of relationships between recovery and online reorganization:

—Reorganization by copying can create a backup of the new, reorganized copy of data, as discussed in Section 3.3.1 [Mix 1994] and Section 3.3.3 [Sockut et al. 1997]. This backup is a basis for recoverability.

—Amsaleg et al. [1995; 1999] impose constraints on their garbage collection to avoid possible problems that could result from interactions between garbage collection and recovery, described in Section 4.4.3.

—If a user transaction omits a redundant insertion in an index because reorganization has already performed the insertion, it may still write a log entry for the omitted insertion. The log entry ensures correctness of possible later rollback of the transaction if a problem occurs, as we discuss in Section 5.1 [Mohan and Narang 1992, 361–362].

Many published descriptions of strategies for online reorganization that we survey discuss issues of recovery, and many do not.

Errors might also arise in the actions of reorganization. Although most executions of reorganization may complete without the need to restart, reorganization must be restartable if an error occurs. Restarting from the beginning of reorganization is acceptable, but it can involve redoing a large amount of work, of course. *Checkpointing* (persistent recording) of reorganization information enables the reorganization to reduce the work by restarting from the most recent checkpoint instead of restarting from the beginning of reorganization. For example, Zou and Salzberg [1996a; 1998] checkpoint during reorganization in place (discussed in Section 3.5.2), Hartman and Ousterhout [1995] checkpoint during cleaning in a log-structured organization (Section 4.5), and Mohan and Narang [1992] checkpoint during index construction (described in Section 5.1). The costs of checkpointing are time and space during reorganization.

3. ISSUES IN ONLINE REORGANIZATION

Now we will discuss issues that arise in the design of online reorganization [Sockut 1977, pp. 1.32–1.41]. The issues involve trade-offs and policies that require decisions. These issues include use of partitions, the process that reorganizes (a background process or users' activities), reorganization by copying to newly allocated storage (as opposed to reorganizing in place), use of differential files, references to data that has moved, general issues in performance, and activation of maintenance. Most of the discussions can apply to all categories of reorganization.

3.1. Use of Partitions

Some DBMSs permit division of a table or database into *partitions*, which can exist even while no reorganization occurs. A partition may be a unit of online reorganization. A partition may also be a unit of offline reorganization or other utilities, during usage or offline reorganization of other partitions [Strickland et al. 1982, p. 492]; [Mohan 1993b, p. 291]. If a purpose of reorganization is to improve performance, reorganization of one partition at a time may benefit users gradually, before completion of reorganization of the entire database.

One design decision is to set the granularity of partitioning. We can consider lack of partitioning as a degenerate case of partitioning, whose granularity is an entire table or database. With a fine enough granularity of partitioning, *offline* reorganization of one partition at a time might be fast enough to approximate 24-hour availability. However, this approach has some disadvantages. In some DBMSs, making the granularity fine can slow the routing of users' accesses into the appropriate areas, and it can also increase the total space required for the partitions' storage descriptors. It also increases the probability that areas of growth and areas of shrinkage will be in different partitions. This increase in the probability increases the likely variation among the partitions' growth rates, thus increasing the total recommended amount of free space to reserve in the database. Also, offline reorganization, like some strategies for online reorganization, has a prerequisite period of *quiescing* of users' activities (blocking of new activities and waiting for existing activities to finish). Offline reorganization of a heavily accessed partition could block most user activity. Finally, in some cases, reorganization requires corresponding changes in indexes (some of which might not be partitioned) or in cross-references from other partitions.

Stonebraker [1989] defines (and explains several uses for) *partial indexes*, which reference only the records that satisfy specified conditions. Use of a partial index to divide an index and/or a data area into partitions allows reorganization and hence lockout of only one partition at a time [Stonebraker et al. 1988, p. 329]. Section 5.1 explains more about partial indexes.

3.2. The Process that Reorganizes

If reorganization changes, creates, or deletes data instances, the designer of reorganization must also choose the process (locus of control) that performs these activities. We will discuss approaches for this reorganization process and then considerations that are specific to maintenance.

3.2.1. Approaches. We have identified two approaches for the process for online reorganization. Also, a third approach is an alternative to changing of instances in persistent storage.

The first approach is to use a *reorganizer*, which is a process that executes in the background (or on a separate processor) to sweep through an area and change the instances. Users and the reorganizer use any required concurrency control, of course. Some authors use the term *eager* reorganization to denote reorganization by sweeping through an area, either in the background (as in this approach) or offline. A reorganizer can reorganize in place or reorganize by copying, as we discuss in Section 3.3. Deferral of some tasks to a reorganizer can improve user performance [Moitra et al. 1988]. The use of a background process relieves user activities of extra work and avoids wasting idle time of a processor [Lampson 1984]. Golding et al. [1995] discuss the detection of slack periods, which are opportunities to perform background tasks. Lumb et al. [2000] analyze the performance for filling disks' rotational latency periods with background tasks.

The second approach is *incremental reorganization during users' activities* (typically, only the activities that reference instances that need change), for example, Gerritsen and Morgan [1976]. This approach, which some authors call *lazy* reorganization, can slow some activities. However, this approach can avoid extra locking and extra data access for some types of reorganization, for example, moving instances among the pages that an activity references. These advantages involving avoidance are less applicable to reorganization that requires access to more pages than just the pages that contain the referenced instances. Incremental reorganization also avoids contention between a user and a reorganizer, although it might increase the contention between two users. Incremental reorganization can also avoid change to instances that will be deleted (or will be the subject of another redefinition) without first being read; changing such instances could be a wasted effort. For incremental reorganization, typically we reorganize in place, but we might instead reorganize by copying, as we discuss in Sections 4.4.4 and 4.4.5. Typically, the incremental reorganization executes as part of a user's transaction. An alternative is to reorganize in a separate system transaction and to have the user transaction wait for the system transaction [Sun et al. 2005; Sun 2005, pp. 89–111].

Two variations of the second approach have similarities to the first approach:

—During some user activities, we incrementally reorganize some instances (but not necessarily the instances that those activities reference) [Baker 1978, p. 283; Kolodner 1990, p. 191; 1992. p. 29]. Each increment resumes where the previous increment stopped.

—For logical or physical redefinition, if the coexistence of several versions of the database definition significantly increases performance overhead or management overhead, we might wish to limit the period when several versions exist. To accomplish this limiting, we can add a process that references (and thus incrementally reorganizes) all instances that still need change.

The third approach, unlike the first two approaches, is an alternative to actual reorganization of instances in persistent storage. Instead, we interpret instances as users access them (e.g., by materializing new columns); Section 6 includes interpretive techniques for logical redefinition. Interpretation gives the user the appearance of changing the instances appropriately, and interpretation does not take the database offline. From the perspective of the user, therefore, interpretation can constitute one of the strategies for online reorganization, despite its lack of change to instances in persistent storage. Reorganization takes effect immediately after specification of the new definition. However, interpretation has a cost, of course, and sometimes the cost motivates *eventual* change to instances in persistent storage.

It is possible to combine approaches. For example, Gerritsen and Morgan [1976] use both the second and third approaches, as we explain in Section 6.1. Friske [2004b] initially uses the third approach (interpretation) for queries and the second approach (incremental reorganization) for updates, and the DBMS eventually uses the first approach (a reorganizer) to complete the reorganization, as we explain in Section 6.2.

3.2.2. Considerations for Maintenance. We continue the analysis of approaches by describing the work of Weikum [1989], who discusses issues in the process that performs maintenance. Most of his examples use clustering in an object-oriented system, but most of the underlying principles can also apply to other contexts of maintenance. Maintenance can reduce disk input/output, and concurrency can achieve high throughput. Performing maintenance incrementally during a user activity can reduce concurrency by increasing the length of the activity (e.g., with additional input/output) and by requiring more locks. Deferral of some maintenance tasks to a reorganizer can shorten a user activity and thus improve throughput, but deferral (e.g., by creating overflow) might slow later activities. Some possible tasks to defer are (1) reclaiming space that is logically deleted and (2) restoring clustering. The author's discussion includes the following questions, whose best answers under various conditions can be difficult to decide:

(1) Under what circumstances should a maintenance task execute during a user activity instead of being deferred? Possible answers include:

—Always.
—Only if a user activity has no constraints on response time. This policy can still slow concurrent activities by holding locks during the maintenance task.
—Only if the task adds no extra input/output cost to the activity. This policy can still add a processing cost.
—Only if the task adds no extra locking to the activity. This condition rarely applies if a maintenance task involves many pages.
—Only if extra locking does not cause a conflict.
—Only if the current system workload is low enough to prevent the task from causing a bottleneck. It is difficult to predict whether a task would delay other user activities.
—Never.

(2) When deferring a task, what should the user activity do instead? Some possibilities are creating overflow and adding an entry to a list of deferred tasks.

(3) What information should the list of deferred tasks contain to describe the required maintenance? Choices include everything, nothing (if a reorganizer can discover the requirements by examining the database), and the addresses of overflowed blocks (information that can be added cheaply to a list of tasks).

(4) When should deferred tasks execute? Tasks might execute periodically, when a task becomes essential, within a time limit, during low workloads, or when degradation exceeds a threshold.

(5) How should deferred tasks execute? Possible answers include short background activities and allowance of further deferral. For lock contention between a user activity and a deferred task, Weikum suggests aborting the deferred task.

Weikum also discusses several points that are specific to certain structures or to the clustering that involves those structures:

—If some storage is volatile, then a user activity that makes some objects persistent must copy those objects to the persistent area. It seems reasonable to cluster those objects at the same time, that is, during the activity. Copeland et al. [1990] describe a system that performs such copying and clustering during the activity.
—If a key (e.g., with an associated index) determines the clustering of data, then deferral of maintenance implies that overflow might form during a user activity.
—If an access path (e.g., an index) implements key-based clustering, then changing data implies changing the index when necessary.

3.3. Reorganization by Copying

If reorganization of an area of a database changes preexisting data instances, another decision involves reorganizing the area in place vs. reorganizing by copying the area to newly allocated storage. Figure 2 shows two approaches for reorganization in place (A and B) and one approach for reorganization by copying (C). The small rectangles represent pages, and a set of four contiguous pages in this simple figure represents the area being reorganized. The letters r through u represent locations of data instances before reorganization, and arrows denote movement or copying of data instances. Variations on the approaches are possible. Under our definitions, *reorganization in place* writes

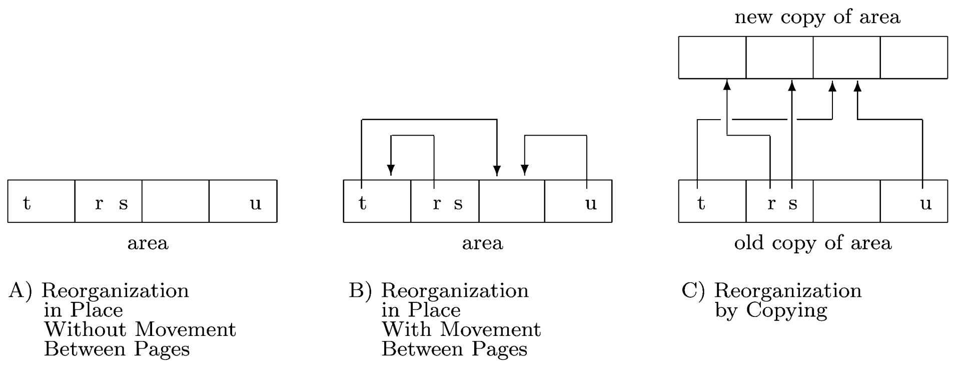
new copy of area
t
r s
u
area
old copy of area
A) Reorganization in Place Without Movement Between Pages
B) Reorganization in Place With Movement Between Pages
C) Reorganization by Copying

Fig. 2. Reorganization in place and reorganization by copying.

the area being reorganized, but it does *not* construct a new copy of the entire area being reorganized, for example, an entire table:

—In approach A for reorganization in place, the reorganization does not move any data instances between pages.

—In approach B for reorganization in place, the reorganization moves groups of one or more data instances between pages, but the reorganization does not necessarily move all the instances. Here, a page might be both the origin of one data instance's movement and the destination of a different instance's movement.

As approach C depicts, *reorganization by copying* can involve unloading *all* the data instances from the area (specifically, from what we call the *old copy* of the area) and reloading *all* the instances in a *new copy* of the area in newly allocated storage. Here, each page can be either the origin of copying or the destination of copying but not both, although a later reorganization might copy in the opposite direction. Approach C includes switching user accesses from the old copy to the new. Reorganization by copying writes the new copy of the area being reorganized, but it does not write the old copy.

Some approaches can be inappropriate for some types of reorganization. For example, approach A does not change the clustering, so strategies for restoration of clustering use approach B or approach C. For garbage collection, we later survey strategies that use approach A and strategies that use approach C, but we are unaware of any strategies that use approach B.

There are trade-offs between reorganizing in place and reorganizing by copying. If reorganization moves data, and users release some locks before the end of a transaction (thus enabling a reorganizer to operate concurrently on the formerly locked data), one advantage of reorganization by copying is simple, efficient assurance of correctness. This advantage is particularly important if reorganization includes changing the assignment of records to pages, as in restoration of clustering. With approach B for reorganization in place and users' releasing of locks before the end of a transaction, a user transaction that *scans* a table (reads a range of data sequentially) might incorrectly experience multiple reading or skipped reading of many records that reorganization moves across the transaction's position in the scanning. If, instead of releasing some locks early, user transactions set page-level locks and hold them until the end of the transaction, reorganization in place also assures correctness simply [Omiecinski et al. 1992; Omiecinski et al. 1994; Salzberg and Dimock 1992; Zou and Salzberg 1996a; Zou and Salzberg 1998]. However, this latter style of locking can dramatically reduce users' concurrency and thus throughput. Iyer and Sockut [2002] present an approach for correctness with early release of locks, described in Section 3.5.5. Huras et al. [2005] also present an approach for correctness with early release of locks, described in Section 4.1.2.

Another advantage of reorganization by copying is that it might cause less contention with users and thus less degradation of users' performance during the copying, if it reads but does not write the copy that users access. Concurrency control may also be simpler.

An advantage of reorganization by copying an entire table (and reconstructing all its indexes) is avoidance of the need to *change* individual references in preexisting indexes when data has moved. This changing can be a requirement when using approach B in Figure 2 for reorganizing the table in place (or when reorganizing just a partition of the table via approach B or approach C). However, some strategies for reorganization by copying can require changing individual references in a copy of the log, as we describe later. Section 3.5 discusses changing of references in indexes, logs, data, and users' activities.

One *disadvantage* of reorganization by copying is that it can require more disk space for the area being reorganized.

Also, copying can require a *transition* between directing users' accesses into the old copy and directing them into the new copy. In many implementations, a disadvantage of copying is a period of limited or nonexistent access by users during the transition.

In some cases, a third disadvantage of reorganization by copying involves a delay in the benefit for users (a delay in giving them access to a reorganized area). Reorganization in place might *begin* to benefit users *immediately*, as the system begins to reorganize the area that users access. In contrast, for reorganization by copying, the benefit begins only *after* the transition mentioned above, which occurs near the end of reorganization in some strategies. However, some copying strategies make the transition and benefit users before the end:

—Some strategies for garbage collection by copying, which we discuss in Section 4.4, make the transition near the beginning of reorganization. Here, user activities might benefit from clustering before the completion of reorganization, but some reorganization activities can occur during user activities and thus slow those activities.

—If the transition is incremental, for example, in accordance with values of a key [Omiecinski 1988; 1989], the benefit occurs incrementally, but user activities can incur the overhead of determining which copy of the area being reorganized contains the desired data.

We will discuss (1) techniques for reorganization by copying, (2) allowance of updates during backup, (3) fuzzy reorganization (a strategy that allows updates during reorganization by copying and that has similarities to strategies for backup), and (4) possible transitions between accessing the old and new copies.

3.3.1. Techniques for Reorganization by Copying. Here, we sketch three techniques for reorganization by copying. The first technique includes two alternatives, namely simple copying and simple unloading and reloading:

—We copy (in reorganized form) from the original copy to a new copy. Both copies are in the DBMS's normal format, that is, a format that is appropriate for users to access.

—We unload from the original copy to an unload copy; then we reload (in reorganized form) from the unload copy to a new copy. The data in the unload copy is not necessarily in the DBMS's normal format; for example, it might be on a tape. Also, it has not necessarily been reorganized.

Section 3.3.3 includes strategies that use this technique.

The second technique involves temporary replication of the area to be reorganized, for a database that is not ordinarily replicated. We copy the area without reorganizing; then we reorganize the copy offline (in place or by reorganizing yet another copy). The copy is in the DBMS's normal format, although we will reorganize the copy before allowing users to access it. The lack of reorganization of the *initial* version of the copy distinguishes this technique from the first technique.

The third technique applies to a permanently replicated database. We reorganize one copy at a time offline (in place or by reorganizing yet another copy). Permanence of the replication distinguishes this technique from the second technique. Section 3.3.3 includes a strategy for online reorganization in the context of replicated databases.

Suppose that a DBMS has a reorganization facility that allows queries but not updates of the area being reorganized during unloading, and it brings the area completely offline during reloading. Mix [1994] describes a strategy (using the first technique) that extends query-only access to cover the reloading step. This strategy needs no modification to the DBMS's normal facility for reorganization. Here are the main steps:

(1) Quiesce updates of the area to be reorganized, while continuing to allow users to query the area.
(2) Apply the unloading step of the DBMS's normal facility for reorganization. Continue to allow query-only access to the area.
(3) Apply the reloading step of the normal facility, but direct the reloading into a new copy of the area. Continue to allow query-only access to the old copy of the area.
(4) Quiesce queries of the old copy.
(5) Exchange the names of the files that underlie the old and new copies of the area being reorganized. This exchange modifies a logical-to-physical mapping, and it will cause the DBMS to direct users' future accesses into the new copy. The area is offline during this step.
(6) Create a backup copy of the new copy as a basis for future recoverability, and allow users to query and update the new copy. Here are two possible sequences for this step: (A) Allow users to query the new copy, and create a backup copy while allowing just queries. After the backup completes, allow users to query and update the new copy. (B) Section 3.3.2 includes strategies that allow users' updates during creation of a backup. By using one of these strategies, allow users to query and update the new copy as soon as the backup begins, instead of waiting for the backup to complete.
(7) Delete the old copy.

Mix's description includes the associated tasks that a DBA must perform. Mix [1992] also describes an earlier version of his strategy, using the *second* technique. Instead of unloading from the old copy, this earlier version temporarily replicates the area (creating a new copy) and then unloads from the new copy.

3.3.2. Allowance of Updates During Backup. During the serializable unloading or other complete reading (e.g., querying) of a table, often a DBMS allows concurrent queries but not updates. An interleaved execution of transactions is *serializable* if it has the same effect as some serial execution of those transactions [Bernstein and Goodman 1981, p. 191]. The strategies that we sketch below for unloading, reloading, copying, or complete reading allow concurrent updates; most of the strategies support serializability. These strategies were designed for *backup* or *long queries*, not reorganization, so we do not survey them in detail. However, in Section 3.3.3, we will explain how concepts from these strategies for backup can lead to a strategy for *reorganization* by copying.

Rosenkrantz [1978] defines several strategies to produce a copy of the database that reflects the state of the database at a certain time. An unloader process traverses the database, and the system tracks its position. Users' updates can also cause unloading, as we describe for each of the strategies:

(1) One strategy produces a copy that reflects the state of the database at the time that the unloading *starts*. When a user updates a not-yet-unloaded part of the database, the system unloads the old value. The first unloaded copy of each part is correct.
(2) In a variation on the strategy above, the system uses a bit array to track which parts the users' updates have unloaded, to avoid unloading parts more than once.
(3) Another strategy produces a copy that reflects the state of the database at the time that the unloading *ends*. When a user updates an already-unloaded part of the database, the system unloads the new value. Here the last unloaded copy of each part is correct.
(4) A final strategy produces a copy that reflects the state of the database at some *intermediate time*. This strategy operates like strategy 3 until the system has unloaded some portion of the database; then it operates like strategy 1.

Rosenkrantz analyzes the performance of each strategy.

Another strategy is to produce a *fuzzy dump* (also called a *fuzzy image copy*) [Gray 1978, p. 478]; [Haerder and Reuter 1983, pp. 312–315]; [Crus 1984, pp. 183–185]; [IBM 1986, pp. 276–280]; [Mohan et al. 1992, pp. 131–134]; [Gray and Reuter 1993, pp. 622–624]; [Mohan and Narang 1993]; [Bratsberg et al. 1997]. Here we record the log's current position, and we then unload the desired part of the database. If the unloading references a page that users have written in the DBMS's main storage buffers, but the DBMS has not yet written the page to disk, we use the version that is in main storage. We can later bring the unloaded data up to date (e.g., during recovery that uses that unloaded data) by applying the log entries (starting from the recorded position). In some DBMSs, the header of each data page includes the log position that was current when the page was most recently written. Here, application of the log ignores a log entry whose position is less than or equal to the position in the header of the page that the log entry indicates, since the page already reflects that logged update [Crus 1984, p. 183]. Lomet [2000] generalizes fuzzy dumping for a system that uses logical log operations instead of physical page-oriented log operations. Hamada et al. [2000] can perform fuzzy dumping and log application for a selected logical subset of physical blocks.

In the strategy of Pu [1985; 1986] for reading an entire database (e.g., for a backup), each entity has a bit to signal whether the reader has read it. The DBMS serializes updates of not-yet-read data before the backup; it serializes updates of already-read data after the backup. It forbids updates that involve both not-yet-read and already-read data; such an update must either abort or wait until the backup process has read the update's data. Pu also describes an extension for distributed databases. Pu et al. [1988] model the performance; the probability of forbidding an update is highest when the backup process has read half the database. Ammann et al. [1995] and Lam and Lee [2006] discuss variations of Pu's strategy.

Srinivasan [1992] and Srinivasan and Carey [1992a] show how techniques designed for constructing indexes [Srinivasan and Carey 1991a], which we discuss in Section 5.1, can also process long queries or copying. Such a query operates in two phases:

(1) The query scans the relevant data (read-locking one row at a time) and extracts the data, producing intermediate results. For updates by concurrent transactions, the DBMS performs the updates in the database and saves information about those

updates in a list. If the query scans data in a particular order, for example, by key value, the DBMS can omit information about updates that it performs ahead of the scan, since the scan will later find the updated data.

(2) The query sets a read-lock on the saved information. The query adjusts its intermediate results to compensate for the saved information. The type of adjustment depends on the type of query, for example, aggregates (like average) or joins. Concurrent updates operate normally.

Concurrent copy [IBM 1993a] uses both hardware (in a storage control unit) and software to produce a copy that reflects the state of the database at the start of the copying. Updates (but not queries) are initially quiesced; then the copying begins, and updates are allowed again. During the copying, for the first update (if any) of each not-yet-copied disk track, the system saves the old version of the track in a file and then performs the update. When producing the copy, the system uses the old version of each updated track.

Langley and Moore [2008] create a backup copy as of a specified time during online reorganization by copying. They begin with a pre-reorganization backup copy and then apply log entries to bring the backup copy up to the specified time. The log includes entries for both reorganization's movement and users' updates. The backup copy eventually includes the not-yet-copied records from the old copy of the area being reorganized and the already-copied records from the new copy of the area being reorganized. Application of a log entry takes place for either the not-yet-copied records or the already-copied records, according to how far the reorganization had progressed when the log entry was written.

The strategies produce a variety of results. Strategies 1 and 2 of Rosenkrantz and concurrent copy produce results that reflect the state of the database at the start of the unloading. Strategy 3 of Rosenkrantz and the strategy of Srinivasan produce results that reflect the state at the end. Strategy 4 of Rosenkrantz produces a result that reflects the state at an intermediate time. Langley and Moore produce a result as of a specified time. Fuzzy dumping (without application of the log) and the strategy of Pu produce results that do not necessarily reflect the state at any one time, but later application of the log can produce a result that reflects the state at the end.

3.3.3. Fuzzy Reorganization (A Strategy that Allows Updates During Reorganization). Section 3.3.2 described strategies that allow updates during copying for backup. To modify such strategies to perform *reorganization* by copying, we can write the final copy in reorganized form instead of the original form. However, if the result of a strategy does not reflect the state of the database at the end of the unloading (including all updates of already-read data), then reorganization via that strategy requires an extra step of performing recent updates, to bring the final copy up to date. One way to perform these updates is to apply the log, as in fuzzy dumping. Of course, the DBA must not discard the log prematurely [Mohan and Narang 1992, pp. 369–370].

Here we describe a strategy for reorganization by copying. We call this strategy *fuzzy reorganization*, since it resembles fuzzy dumping. We will describe one possible set of steps; variations are possible. The strategy allows queries and updates during almost all steps, including a step of log application. Subsequent steps involve a period of query-only access and an offline period. These are the steps:

(1) Record the log's current position. Users can query and update during this step.

(2) Unload the old (original) copy of the area being reorganized, and reload into a new copy in reorganized form. We might copy directly instead of unloading into a file and reloading from that file. Concurrently, users can use the DBMS's normal facilities to

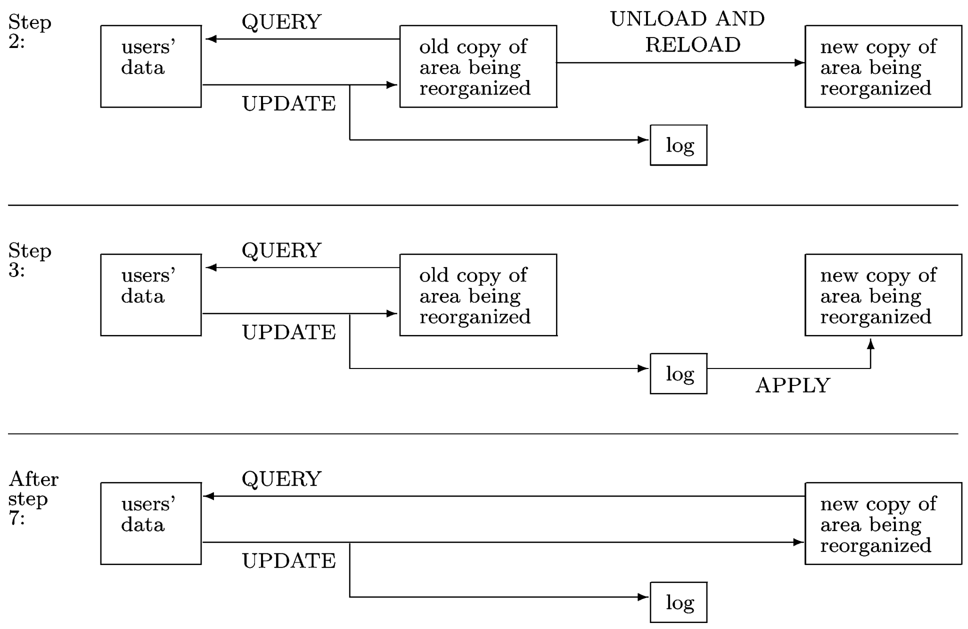
Step 2:
users' data
QUERY
UPDATE
old copy of area being reorganized
UNLOAD AND RELOAD
new copy of area being reorganized
log
Step 3:
users' data
QUERY
UPDATE
old copy of area being reorganized
new copy of area being reorganized
log
APPLY
After step 7:
users' data
QUERY
UPDATE
new copy of area being reorganized
log

Fig. 3. Some steps of fuzzy reorganization.

query and update the old copy. The top of Figure 3 shows this step. Users' querying and updating consist of copying data between the database (specifically, the old copy) and variables in users' programs (shown in the figure as "users' data"). In the log, the DBMS's normal facilities also append log entries that correspond to users' updates.

(3) Record the log's current position again. Read the log (between the two recorded positions), and apply the log entries to the new copy. This application of the log will reflect users' updates that occurred during Step (2). Users can query and update the old copy during this step. The figure shows this step.
(4) Quiesce updates of the old copy, while continuing to allow queries of the old copy.
(5) Record the log's current position again, and read and apply the log again (between the two most recently recorded positions). We need this additional step of application only to handle updates that occurred after Step 3 began. Users can query the old copy during this step.
(6) Quiesce queries of the old copy.
(7) Change the logical-to-physical mapping, for example, by exchanging the names of the files that underlie the old and new copies of the area being reorganized. The area is offline during this step. After we have changed the mapping, users' future accesses will go into the new copy, as the figure shows.
(8) Create a backup copy of the new copy as a basis for future recoverability, and allow users to query and update the new copy. For the strategy of Mix [1994], Section 3.3.1 discussed two possible sequences for this step, one of which allows updates, not just queries, *during* the backup. Another way to allow updates during this step is to create a backup copy as part of Step (2). However, such a backup copy represents a less recent time, so it would require application of more log entries during a possible later recovery. Another way to create a backup copy is to create one during Step (2)

and then append (to that backup copy) *only* the changed pages at the end of Steps (3) and (5) [Sockut et al. 1997]; this backup copy represents a recent time.

(9) Delete the old copy.

Instead of unloading the data from the *database* in Step (2), an alternative is to unload from the most recent backup copy of the data. This alternative eliminates the reorganizer's accessing and locking of the database, which users access. However, it requires the reorganizer to apply additional log entries (all the entries since the backup). Also, the backup might reside on tapes.

In some systems, an optimization is to unload *only* the blocks that have *not* been updated since the start of reorganization [Pereira 2000]; application of the log includes all the blocks that *have* been updated.

IBM [1997, pp. 2.193–2.232], Sockut et al. [1997], and Friske and Ruddy [1998] describe enhanced control of the log application in fuzzy reorganization (Steps (3) through (5) above), allowing the DBA to limit the query-only period. Here, in Step (3), we perform one or more iterations of log application while users can query and update the old copy. The first iteration applies the part of the log that reflects users' updates that occurred during Step (2). After the first iteration, each iteration applies the part of the log that reflects the updates that occurred during the previous iteration. Hopefully, the time that each iteration requires will typically be less than the time that the previous iteration required. At the end of each iteration, we use the current and previous log positions to estimate the time that the next iteration will require. We compare this estimate to the maximum desired length of Step (5) (the query-only period). A parameter of the reorganization command specifies the maximum. If the estimated time exceeds the maximum, we also perform a different comparison:

—If the range of log that the next iteration would apply is smaller than a certain fraction of the range that the current iteration just finished applying, then the reorganizer's reading of the log is catching up quickly enough to users' appending to the log. Therefore, Step (3) iterates again.

—If the range is not smaller, we are not catching up quickly enough. Actions by which the DBA may solve this problem include continuing Step (3) with a higher maximum query-only period, continuing Step (3) after increasing the execution priority of the reorganizer, continuing after quiescing updates of the old copy (Step (4)), and canceling reorganization.

If, instead, the estimated time is within the maximum desired length of Step (5), we quiesce updates (Step (4)) and perform Step (5) once, while users have query-only access.

For this iterative application of the log, the DBA's choice of a value for the parameter mentioned above (the maximum desired length of the query-only period) involves a trade-off. A small value can shorten the query-only period, of course. A large value can hasten the transition to the new copy and thus hasten the completion of reorganization, since probably fewer iterations are necessary to reduce the estimated time per iteration below the maximum.

Friske et al. [2003b] describe a method to estimate the time for each iteration of log processing, which the DBMS compares to the maximum desired length of the query-only period.

Dynamic management of buffers for log entries can speed the tasks of log processing [Banzon et al. 2006].

Løland and Hvasshovd [2006] also apply the log iteratively.

For exchanging the names of the files in Step (7), an alternative to renaming the files is to change a value in the database catalog, where the DBMS calculates the file names from the value [IBM 2001; Teng and Todd 2002]. In some systems, this technique can be faster than renaming.

One disadvantage of fuzzy reorganization is that the number of not-yet-applied log entries will increase if we suspend reorganization, for example, during a period of peak usage.

Another concern with use of a log is that a log might ordinarily omit some information that the reorganization needs [Mohan and Narang 1992, p. 369]. For example, in some DBMSs, if a user modifies just column 2 in a row of a table, the log contains the row's identifier and the value of column 2, but it ordinarily omits the value of the unchanged column 1. If a type of reorganization needs the values of both columns 1 and 2, then during reorganization, the system would need to use an option (which is available in some DBMSs) that forces inclusion of all columns (even unchanged columns) in the log.

An alternative to using the regular log is to define a trigger that records updates of the area being reorganized [Pereira 2000]. The fuzzy reorganization uses the recorded updates.

A similar alternative to using the regular log is to set up a partially redundant short log [Isip 2007]. When the DBMS appends to the regular log, it also appends to the short log if the logged activity affects the area being reorganized. The fuzzy reorganization's log processing uses the short log.

A similar optimization, to reduce log traffic in a distributed environment, is to filter out the log entries that do not refer to the copied data [Sun 2005, pp. 77–88].

A complication in fuzzy reorganization arises if the database uses *data compression* to save space. In some strategies for data compression, for example, Iyer and Wilhite [1994], bit strings that appear frequently in the data are stored as shorter (compressed) bit strings, and a *compression dictionary* maps between compressed bit strings and full bit strings. Data (stored in the database) and log entries both use compression. A log entry might contain only a contiguous *subset* of the corresponding compressed data record, namely the subset that the update has changed. In some DBMSs, for invocation of reorganization of a compressed database, the DBA chooses between (1) constructing a new dictionary during reorganization and (2) keeping the existing dictionary. Choice 1 requires time, but it might yield more efficient storage, for example, if the data has changed significantly since the construction of the existing dictionary. Choice 1 also complicates log application in fuzzy reorganization. The log entries use the old dictionary, and the data in the new copy of the area being reorganized uses the new dictionary. Applying such a log entry directly to a record in the new copy is meaningless. If the DBMS stores a *complete* compressed data record in a log entry, then we can decompress the log entry with the old dictionary, compress the result with the new dictionary, and apply the result to a data record in the new copy.

However, if a log entry contains only the changed subset of a compressed data record, then a log entry *alone* might not represent valid data, and the changed subset of the compressed data record in the old copy might not correspond to any contiguous subset of the compressed data record in the new copy. Therefore, the sequence of decompressing and compressing (described above) is meaningless. Ruddy et al. [1999] solve this problem for choice 1 above (when log entries contain only the changed subsets) by performing these steps to apply a log entry to a data record in the new copy of the area being reorganized:

(1) Extract the data record from the new copy of the area and *decompress* it, using the *new* dictionary.
(2) *Compress* the data record, using the *old* dictionary.
(3) Apply the log entry, which uses the *old* dictionary. The subset of data in the log entry corresponds to a contiguous subset of the compressed data record.

(4) *Decompress* the data record, using the *old* dictionary.
(5) *Compress* the data record, using the *new* dictionary, for storing the record in the new copy of the area.

Several people have discussed what we call fuzzy reorganization. Wilson [1979, p. 569] and Stonebraker et al. [1988, p. 329] mention the concept of fuzzy reorganization; Mohan and Narang [1992, pp. 369–370] mention a similar concept. IBM [1997, pp. 2.193–2.232], Sockut et al. [1997], and Friske and Ruddy [1998] restore clustering of data (Section 4.1.3) and reorganize an index (Section 4.2.3) via fuzzy reorganization. BMC [1997; 1998] describes similar restoration of clustering and reorganization of an index. Pereira [2000] and Marshall et al. [2006] also describe similar restoration of clustering. Zou and Salzberg [1996b] use a file that resembles a log during online reorganization of an index, discussed in Section 4.2.3. Bratsberg and Humborstad [2001] and Sun [2005, pp. 77–88] use fuzzy reorganization for online scaling in a parallel or distributed database; see Section 4.3.2. Achyutuni et al. [1996] and Bratsberg et al. [1997] perform fuzzy reorganization while rebalancing parallel data; see Section 4.3.3. O'Toole et al. [1993] use the log during online garbage collection by copying (Section 4.4.5). Troisi [1994; 1996], Englert [1994], and Maier et al. [1997] describe use of the log during online construction of indexes and redefinition of partitions, which we describe in Section 5. Friske et al. [2007] use the log during online construction of indexes; see Section 5.1. Løland and Hvasshovd [2006] use fuzzy reorganization for specific examples of logical redefinition; see Section 6.2.

A strategy with similarities to fuzzy reorganization serves a legacy implementation of the National Crime Information Center (NCIC), which we mentioned in Section 2.1. The indexed sequential files require periodic maintenance. The strategy for maintenance [Sockut and Goldberg 1979, p. 388] has these main steps:

(1) Copy (and reorganize) between disks. This requires at least 3 hours, during which users can query and delete in the old files but cannot insert or modify. Section 2.1 explained the reason for allowing deletions. The system also tracks the deletions.
(2) Replace the old files by the reorganized files. This requires about 15 minutes, during which the files are offline.
(3) Perform the saved deletions in the reorganized files.

About four times per year, the NCIC undergoes logical or physical redefinition, using the same strategy. Examples of redefinition include adding and moving fields, adding and deleting indexes, and changing parameters of indexes.

Several facilities support replication of databases, for example, IBM [1993b], IBM and Integrated Systems Solutions [1994], and Rengarajan et al. [1996, pp. 23–24]. The following strategy for online reorganization, which uses a replication facility, resembles fuzzy reorganization:

(1) Initially, allow users to query and update the primary copy; use the other copy as a backup.
(2) Use the DBMS's normal facility for reorganization to reorganize the backup copy offline. Users can query and update the primary copy.
(3) Use the replication facility to copy recent updates from the primary copy to the backup copy. Users can query and update the primary copy.
(4) After the copying has caught up with the updates of the primary copy, quiesce updates of the primary copy. Users can still query the primary copy.
(5) Use the replication facility to copy any very recent updates from the primary copy to the backup copy. Users can query the primary copy.

(6) Quiesce queries of the primary copy.
(7) Exchange the roles of the primary copy and the backup copy.
(8) Allow users to query and update the primary copy; this copy was originally the backup copy.

If desired, we can then apply the same strategy to reorganize the backup copy; that copy was originally the primary copy. A variation on the above strategy [Mizoguchi and Nishiyama 1994] uses the backup copy to generate a new, reorganized pair of primary copy and backup copy. In a replicated database, Bratsberg et al. [1997] use the log for fuzzy operations. Another approach is to suspend replication, reorganize the backup copy, apply the log of the primary copy to bring the backup copy up to date, copy the backup copy to the primary copy, and resume replication [Kitsuregawa et al. 2008].

Subramaniam and Loaiza [2005] describe reorganization by creating new copies of tables' definitions (and, in some cases, creating some completely new tables), copying data from the old copies to the new (while allowing access to the old), bringing the new copies up to date (e.g., via applying a log), and switching the identities of the old and new copies. The bringing up to date may take place iteratively. The data may be offline during the last iteration of the bringing up to date and during the switching. The authors suggest categories of reorganization to which this strategy can apply, for example, changing physical parameters, changing partitioning, reducing storage fragmentation, adding, dropping, or renaming columns, splitting one table into several, and combining several tables into one. The authors give examples within such categories.

3.3.4. Transitions between Accessing Old and New Copies. We noted earlier that users' accesses during reorganization by copying undergo a transition from the old copy to the new copy. Several alternatives are possible for such a transition, and more research can be done on this topic. Here we will discuss possible transitions.

Suppose that all accesses are via a key, and a *logical-to-physical mapping table* (discussed in Section 3.5.1) maps from key values to records' current addresses [McIver and King 1994]. When we copy a record, we change the address in the record's entry in the mapping table. This simple transition applies to one access at a time. Users access some records in the old area and some in the new area.

Again, suppose that all user accesses are via a key or via a sequence that always matches a key. Also, suppose that the order of processing data instances in the reorganization matches the order of values of the key. Here, another simple transition (again applying to one access at a time) is to direct each access according to whether the reorganization has already reached the key value that the access uses [Omiecinski 1988, 1989; IBM 2005, pp. 370–393]. Again, users access some records in the old area and some in the new area.

Alternatively, suppose that some user accesses are not via the key that the reorganization uses. Examples include sequential access (if it does not always match the key), access via an index of a different key, and access via linked lists (starting from a set of roots). Here, a variation of the simple transition above is to direct each access initially into the old copy and find the old instance, including its key value. If the reorganization has already reached the instance's key value, we then redirect the access into the new copy by using the key value or a forwarding pointer. When the reorganization has finished the last key value, all accesses go directly into the new copy. Of course, this variation applies only if (1) all types of access will operate correctly using the redirection and (2) the applications can tolerate the time for redirection. If reorganization can reorder instances, the last step of this transition (all accesses going directly into the new copy) cannot easily apply to sequential accesses that are *already* in progress.

Another possible transition (for arbitrary user accesses) is to choose one time for switching and to direct user accesses according to when the accesses begin. *All* user accesses that begin before the switching time go into the old copy, and all user accesses that begin after the switching time go into the new copy. This transition involves an issue that is especially important for user activities that take a long time. The issue is whether to quiesce users' old activities (i.e., activities that are already in progress at the switching time) before we allow new activities to begin:

—Quiescing seems to be the simpler choice, since only one copy of each data instance is valid at any time. A disadvantage of quiescing is that it takes time and thus delays new activities, of course. Several of the strategies that we discussed in Sections 3.3.1 and 3.3.3, including our description of fuzzy reorganization, use quiescing. Users initially access all records in the old area and later access all records in the new area.

—Although lack of quiescing avoids a delay of new activities, it requires the DBMS to be able to process two valid copies until the old activities complete. It also requires extra concurrency control, if users can update and the two copies of the area being reorganized sometimes contain valid copies of the same instances. We believe that efficient coordination needs more research.

In yet another possible transition, which O'Toole et al. [1993] describe for garbage collection for persistent linked lists, we again use one time for switching. Here, however, user accesses that are in progress at that time begin in the old copy of the area and continue in the new copy. New accesses also go into the new copy. The switch includes updating the lists' roots, including any that are in processor registers. This strategy of switching the accesses that are in progress applies if the logical order of users' accesses remains valid during reorganization. For example, the strategy applies if all user accesses follow linked lists, whose logical order the reorganization does not change. This strategy avoids delay of new activities and also avoids concurrent processing of two valid copies. Users start in the old area and sometimes follow pointers to the new area. Kolodner [1990; 1992] and Kolodner and Weihl [1993] use a comparable transition for garbage collection for persistent linked lists.

During reorganization by copying, Ghandeharizadeh et al. [2006] direct queries to the old copy and updates to both copies. Eventually, both queries and updates go only to the new copy.

3.4. Use of Differential Files

Another decision involves where to write users' updates of the area being reorganized. One possibility is to write them in the main (primary) area of the database immediately. Another possibility is to save them in a *differential file* (described below) during reorganization and write them in the main area later. Using a differential file can simplify and speed the reorganizer's processing of the main area. However, it adds complexity (in processing *user* access), space overhead, and sometimes time overhead (for users). Another disadvantage of a differential file is that if we suspend reorganization, for example, during a period of peak usage, then the differential file grows.

Severance and Lohman [1976] and Lohman [1977] describe differential files in detail. The main data area contains the data as of a certain time, and a differential file stores any updates that take place later. A filtering scheme (to locate records) uses a set of hashing functions that map record identifiers into addresses in an array of bits in main storage (initially all 0). A *record identifier* (often shortened to *RID*) is a number that identifies a record in a database; most authors use this term to mean a physical-level identifier. On updating a record, the system sets all its mapped bits to 1, and the update goes into the differential file (not into the main area). To read a record, given a RID, the system first tests its mapped bits. If all these bits are 1, the record is *probably* in the differential file, which the system searches. If the search is unsuccessful, then the union of the hashing of one or more *other* records' RIDs must have collided with the hashing of *this* record's RID, so the system then searches the main area. If not all the mapped bits are 1, the system searches *only* the main area, where the record (if it exists) *surely* resides. The system eventually moves the contents of the differential file into the main area and resets all the bits. The authors discuss how the choice of the number of hashing functions affects performance.

Besides the uses that Severance and Lohman discuss, differential files can also assist in online reorganization. Zilio [1998] uses differential files during online rebalancing of parallel data, as we discuss in Section 3.7.3. Also, some strategies for online construction of indexes, which we survey in Section 5.1, use similar files. These include the index strategies of Srinivasan and Carey [1991a; 1992b], Srinivasan [1992], and Mohan and Narang [1992].

Stonebraker [1981] describes a variation using two differential files: one for insertions and one for deletions. A view of the database, expressed via set operations, is: (main data area UNION insertion file) MINUS deletion file. The author describes update operations under this arrangement. Most operations are straightforward, but reinsertion of a deleted row (or modification of a row to have a previous value) requires deletion of an entry from the deletion file. Such deletion eliminates the simple append-only property that the deletion file would otherwise have.

We have described the use of a differential file to save users' updates during reorganization, and Section 3.3.3 described the use of a log for the same purpose. In the context of reorganization, a log and a differential file have several similarities. Each one is a file that is separate from the main area of the database. Each stores users' updates. After reorganization of the main area, we can apply the stored updates to the main area.

However, a log and a differential file also have several differences. Many of the differences result from the fact that a log is designed for recovery, not reorganization, while a differential file can be designed for a different specific purpose, for example, a specific type of reorganization. These are differences:

—Much of the data in a typical log entry is redundant with data in the main area of the database (until possible subsequent updating of that data in the main area). In contrast, data in a differential file does not yet exist in the main area of the database.

—A log typically contains both undo and redo information, but a differential file typically contains just the equivalent of redo.

—User activities typically query a differential file (when necessary) but not a log. Therefore, a DBMS's treatment of a log typically differs from its treatment of the main area. Examples of likely places of difference are elaborate authorization, sophisticated concurrency control, and access mechanisms like indexes, which a log typically lacks. The treatment of a differential file is more likely to resemble the treatment of the main area.

—A log reflects all updates, not just those that are relevant to the reorganization. A specialized differential file need not reflect irrelevant updates [Srinivasan and Carey 1991a, p. 18; Mohan and Narang 1992, p. 369].

—A DBMS typically maintains a log even without online reorganization. However, a DBMS maintains a differential file only if that DBMS has a particular reason for it. Possible reasons include online reorganization and the uses that Severance and Lohman [1976] discuss.

3.5. References to Data that Has Moved

Another issue that arises in some cases is handling of references that point to data records that online reorganization has moved. For example, references might appear in other data records, in an index, in a log, or in the processing of a user activity. If reorganization moves a data record, the DBMS must assure that references to the data record eventually go to the record's new location. This topic arises mainly during maintenance, but it can also apply to redefinition.

Some DBMSs assign a RID to each record. For example, a RID might consist of a page number and an identifier within the page. Indexes and other structures can use RIDs to reference records. During normal usage, when no reorganization takes place, the RID of a record is stable. In some DBMSs, even the compaction of a page (by moving records within the page) does not change any RIDs. However, online reorganization can remove this stability by moving records between pages and thus changing their RIDs; an unchanged reference to a moved record becomes invalid. Weikum [1989] notes that when reorganization moves an object that can be the target of pointers, some choices for changing of pointers include leaving a forwarding pointer at the object's original location, updating all pointers immediately, and updating a distinguished parent while having all other pointers go through that parent.

We will discuss the use of *mapping tables*, which are structures that can support changing of references. Then we will describe work in changing of references, in several contexts: references in indexes for reorganization in place, references in the log for fuzzy reorganization, references in data, and references in users' activities for reorganization in place.

3.5.1. Use of Mapping Tables. We will describe two types of mapping tables (logical-to-physical and physical-to-physical) for changing references.

A *logical-to-physical mapping table* maps from logical identifiers, which do not change during reorganization, to current physical identifiers, which change. We use logical and physical as relative terms. A logical identifier might be the value of a unique key, an object identifier, the identifier of a hash bucket, or a logical block number. A physical identifier might be a RID, a disk number, or the location of a physical block. References outside the mapping table use logical identifiers. In our discussion in this section, a mapping table deals with references to records, although some of the work in later sections uses mapping tables that deal with references to hash buckets or blocks. Typically, a logical-to-physical mapping table contains an entry for each record, even if reorganization has not moved the record. Logical-to-physical mapping tables serve McAuliffe et al. [1998] (discussed in Section 4.1.1), Omiecinski [1985b] (Section 4.1.2), Scheuermann et al. [1989] (Section 4.1.2), McIver and King [1994] (Section 4.1.3), Bratsberg and Humborstad [2001] (Section 4.3.2), Wolczko and Williams [1992] (Section 4.4.2), Yong et al. [1994] (Section 4.4.7), de Jonge et al. [1993] (Section 4.5), Wang et al. [1999] (Section 4.5), and several systems that manage parallel or distributed data (Section 4.3). One of the possible roles of a unique index, besides efficient access and enforcement of uniqueness, is service as a logical-to-physical mapping table [Martin 1975, pp. 357–359; Wolczko and Williams 1992; Srinivasan et al. 2000]. For parallel or distributed data, if several nodes each have a copy of the mapping table, and the table is updated at one node, then the updates must eventually propagate to the other copies [Vingralek et al 1994; 1998].

To achieve the effect of changing all the references when a record moves, the system simply changes the physical identifier in the record's entry in the logical-to-physical mapping table. Parts A and B of Figure 4 show part of the contents of a logical-to-physical mapping table. In this example, reorganization moves a record (which has

LOGICAL IDENTIFIER	CURRENT PHYSICAL IDENTIFIER
...	...
L9	OP1
...	...

A) *Logical*-to-physical mapping table *before* moving record

→

LOGICAL IDENTIFIER	CURRENT PHYSICAL IDENTIFIER
...	...
L9	NP2
...	...

B) *Logical*-to-physical mapping table *after* moving record

OLD PHYSICAL IDENTIFIER	NEW PHYSICAL IDENTIFIER
...	...
(Table has no entry for this record.)	
...	...

C) *Physical*-to-physical mapping table *before* moving record

→

OLD PHYSICAL IDENTIFIER	NEW PHYSICAL IDENTIFIER
...	...
OP1	NP2
...	...

D) *Physical*-to-physical mapping table *after* moving record

Fig. 4. Use of mapping tables.

logical identifier L9) from an old physical location (with physical identifier OP1) to a new physical location (with physical identifier NP2). Part A of the figure represents the mapping table before movement of the record, and part B represents the mapping table after movement. For reorganization by copying, an alternative to changing the physical identifier is to use two mapping tables (old and new). We will explain parts C and D (for a *physical*-to-physical mapping table) shortly.

A logical-to-physical mapping table has advantages and disadvantages. A disadvantage, especially when a table cannot fit in main storage, is a level of indirection in access. However, Srinivasan et al. [2000] describe a system in which secondary indexes contain guesses for physical addresses of records (based on physical addresses that were correct in the past), and an access via the mapping table (primary index) is necessary only if a guess has become wrong. Another disadvantage is space, unless the mapping table is an index that would exist for efficiency or uniqueness even without online reorganization. These disadvantages apply at all times, not just during reorganization. However, changing of references is simple and efficient, and McIver and King [1994] note several other advantages of a mapping table: (1) After reorganization has updated the mapping table's entry for a moved record, the system can safely interrupt reorganization without physically changing references in other objects. (2) Reorganization avoids accessing the pages (in other objects) that contain references. (3) Reorganization can focus on a limited region of the database.

Many systems do not use a logical-to-physical mapping table. Such systems may use a *physical-to-physical mapping table*, which maps from old physical identifiers to new physical identifiers. This differs from the *logical*-to-physical mapping table that we described above, which maps from logical identifiers to current physical identifiers. Again, an implementation can use RIDs or various other identifiers as the physical identifiers. Typically, a physical-to-physical mapping table contains an entry for a record only if reorganization has moved the record. When moving a record, the system inserts an entry into the physical-to-physical mapping table, to support the eventual changing of references. Part C of Figure 4 represents a physical-to-physical mapping table before movement of a record from an old physical location OP1 to a new physical location NP2. Part D represents the mapping table after movement.

For systems that do *not* use *logical*-to-physical mapping tables, the next few sections deal with physical-to-physical mapping tables and other techniques. We include discussions of advantages and disadvantages of physical-to-physical mapping tables.

3.5.2. References in Indexes for Reorganization in Place. For record-moving reorganization in place (approach B in Figure 2), we will discuss several techniques for changing references in indexes (and, if necessary, references in other structures). The techniques are a physical-to-physical mapping table, immediate changing of references, and a hybrid of the two. In addition, some strategies for reorganization of an index in place (Section 4.2.2) include extra pointers to enable correct searching of the index during reorganization. Also, Lee et al. [2000] (discussed in Section 4.3.1) and Achyutuni et al. [1996] (discussed in Section 4.3.3) describe changing of indexes when data moves between parallel disks.

3.5.2.1. Omiecinski et al. The technique of Omiecinski et al. [1992; 1994] and Omiecinski [1996, pp. 27–28] uses a physical-to-physical mapping table. The authors assume that the mapping table is small enough to fit into main storage as a hash table. When a user activity finds a RID in an index, it checks the mapping table (for possible translation) before using the RID. This mapping table is just one part of a strategy for online restoration of clustering, which we discuss in Section 4.1.2.

3.5.2.2. Salzberg and Dimock. For online reorganization that changes RIDs, Salzberg and Dimock [1992] describe several issues, especially changing of references. When a reorganization transaction moves a record, the authors suggest including all required changes (e.g., changes in references) in that one reorganization transaction, for consistency if a system failure occurs. A reorganization transaction write-locks the source and target of the move, moves the record, changes the primary index, updates other references, commits, and unlocks. When making changes, the transaction logs the changes. The authors advocate keeping a reorganization step short, to minimize the delay to users. They also suggest that when users access a record, if the record needs reorganization, the system should append an entry to a recoverable queue of requests for reorganization and should log the queue processing. The authors note that moving a record is better than deleting and inserting, since deleting and inserting might incorrectly trigger actions that the database designer has specified to maintain integrity, for example, deletion of related records. Moving is also more efficient.

Salzberg and Dimock note that existing mechanisms for handling accesses to records that have been deleted might also apply to handling accesses to records that have been moved. Here are possible behaviors:

—A user transaction holds all locks until the end of the transaction; deadlock handling can resolve a conflict, if any. This policy (with page-level locks) avoids multiple reading or skipped reading of a record that the reorganizer would move across a user's position in a scanning of data.

—A user transaction holds locks until it obtains a lock on the identifier of the desired record.

—After obtaining a lock, a user transaction reexamines the access path to check whether the reference has changed.

3.5.2.3. Zou and Salzberg. Another technique is that of Zou and Salzberg [1996a, 1999c pp. 37–39; 1998]. The reorganization moves one record at a time. For simplicity, the authors assume that the *clustering index* (the index that determines the clustering of records) is a unique index. A *boundary value* records the value of the *clustering key* (the key of the clustering index) for the most recently moved record.

For each index that references the record that the reorganization is moving, the timing of the updating of the RID in the index depends on whether the relevant leaf page of the index is currently in a buffer in main storage. If the leaf page is in main storage, the system updates the RID immediately. If the leaf page is not in main storage, the system defers the update. *Lookup tables* in main storage record the deferred updates:

—The *forward address table* is a physical-to-physical mapping table that is indexed by old RID. It stores old RIDs, new RIDs, and the numbers of index references that still need change.

—Each *nonclustering index* (i.e., each index other than the clustering index) has an associated *pending changes table*, which is indexed by page number. Each entry in such a table includes an index value, a page number, and an old RID.

When the lookup tables get full, the system must perform *cleanup* by paging in the index pages and updating them.

The goal of each unit of reorganization is to move one record. In a unit, if any lookup tables lack space, the system performs cleanup. The reorganizer then write-locks the boundary value and the old and new RIDs of the record to be moved. The reorganizer then reads the record and write-locks the key values in each of the nonclustering indexes that require change for this record. If this locking causes a deadlock with users, the reorganizer unlocks all locks and restarts the unit of reorganization. Otherwise, the reorganizer moves the record, logs the move, updates the clustering index, logs the update, and unlocks the old and new RIDs. For each nonclustering index, the reorganizer performs these actions:

—If the relevant leaf page is in main storage, the reorganizer performs and logs the update.

—If the relevant leaf page is not in main storage, the reorganizer inserts an entry in the pending changes table for the index, and it increments the count in the old RID's entry in the forward address table (or creates the entry if it does not already exist).

—The reorganizer unlocks the key value in the nonclustering index.

Finally, after performing or deferring the changes for all nonclustering indexes, the reorganizer changes the boundary value to the clustering key of the moved record, and it unlocks the boundary value.

Later, when the buffer manager (a component of the DBMS) pages in a leaf page of a nonclustering index, it consults the lookup tables, and the system performs the deferred changes (if any) for that page and updates the lookup tables. This changing is incremental reorganization during users' activities.

The system logs the acts of moving a record from a page, moving a record into a page, and changing an index. The authors describe the recovery logic. The system checkpoints the boundary values and the lookup tables. The reorganization is restartable.

Zou and Salzberg discuss two methods for user transactions to scan a data table without reading a moved record twice or skipping a moved record:

(1) Use page-level (or coarser-grained) locks, and hold them until the end of a transaction.

(2) Read-lock the boundary value, and hold the lock until the end of the scanning. Here the reorganization is not completely concurrent with usage. A reorganization step starts when no users are scanning, and no users can start scanning until a reorganization step ends.

A performance analysis reveals that three factors (listed here) each decrease the amount of reorganization input/output, for the reasons shown here: (1) As the time between consecutive reorganization units grows, the lookup tables fill more slowly, requiring less frequent cleanup. (2) As the amount of user input/output between cleanups grows, more of the index changing takes place during user activities. (3) As the size of the lookup tables increases, less cleanup is required, and more of the index changing takes place during user activities.

3.5.2.4. Comparison. The techniques discussed above involve trade-offs. An advantage of a physical-to-physical mapping table is avoidance of immediately accessing all references to a moved record. Advantages of changing references immediately are:

—Avoidance of the time for user transactions to consult a mapping table and read the new RID (if an entry exists), especially if the table does not reside completely in main storage
—Avoidance of space for the mapping table entries, which the system cannot free until all relevant references have changed.

The hybrid technique of Zou and Salzberg (immediately changing the references that are in main storage and deferring the others in a mapping table) avoids (1) immediate access to references that are not in main storage, (2) the time for user transactions to read the new RID in a mapping table (after the first time that a reference is paged in), and (3) the space for some mapping table entries (for references that have already been in main storage since record movement). However, the hybrid technique involves consulting a table when index pages are paged in, and sometimes it involves cleaning up the lookup tables and thus accessing references.

3.5.3. References in the Log for Fuzzy Reorganization. A different context of changing references involves log entries in fuzzy reorganization. In some DBMSs, an entry in the log identifies a record by the record's RID. In fuzzy *dumping*, RIDs do not change. In recovery that uses the result of fuzzy dumping, application of an entry in the log can use the entry's RID to identify the data record to which the log entry should apply. In fuzzy *reorganization*, however, RIDs *can* change. Log entries correspond to users' updates of the old copy of the area being reorganized and thus use the *old* RIDs. To apply a log entry to the new copy, we must change the RID in a copy of the log entry so that the RID identifies the record to which the log entry should apply in the *new* copy of the area. We will discuss two techniques for changing RIDs that the log contains.

One technique is to include a *forwarding field* in each record. When the reorganizer copies a record from old to new, it stores the new RID in this forwarding field in the record in the old copy. Application of the log uses the field to translate RIDs in log entries. O'Toole et al. [1993] use this technique in the context of online garbage collection for persistent data. We describe other aspects of this garbage collection in Section 4.4.5.

Another technique is to create a separate temporary physical-to-physical mapping table, containing an entry for each record in the old copy. When the reorganizer copies a record from old to new, it stores the old and new RIDs in an entry in the mapping table, which can be indexed by old RID. IBM [1997, pp. 2.193–2.232], Sockut et al. [1997], Sockut and Beavin [1998], and Friske and Ruddy [1998] use this technique in the context of restoration of clustering in a relational DBMS. In this implementation, each entry in the mapping table also contains a log position, so that log application can ascertain whether a data record already reflects the operation in a log entry. Application of the log uses the mapping table to translate RIDs in log entries before sorting them by new RID and actually applying them to the new copy. For many of the types of operations in log entries, the application of the log can also update the mapping table. This updating reflects (1) the state of a data record before application of the log entry and (2) the logged operation. For a log entry that represents an insertion, the updating includes the use of an *estimated* new RID (as a basis for sorting of log entries to improve locality of reference) and eventual translation of the estimated RID to an actual new RID. We discuss other aspects of this maintenance in Section 4.1.3. BMC Software [1997] and Pereira [2000] also use a mapping table for fuzzy reorganization.

The choice between a forwarding field and a physical-to-physical mapping table involves trade-offs [Sockut et al. 1997, p. 420]. In a database context, which was not the context for O'Toole et al., a mapping table has several advantages:

The first advantage involves deletion (which does not apply to the environment of O'Toole et al., whose users do not explicitly delete records). Suppose that after the reorganizer has copied a record, a user deletes the record (in the old copy of the area), and the DBMS appends a deletion entry to the log. The reorganizer will eventually find the log entry, translate its RID from old to new, and apply the deletion to the new copy of the area, to delete the record there. After the user's deletion but before the reorganizer's application of the log entry, the DBMS might reuse the space that the deleted record occupied. Therefore, the DBMS cannot safely store the new RID in a forwarding field in the old (deleted) record. A mapping table *can* safely store the mapping of RIDs.

The second advantage involves input/output. A data record can be much larger than an entry in a mapping table, so the set of all data records can require many more pages than the set of all mapping table entries does. Therefore, adding and reading mapping table entries may involve less page input/output than writing and reading the new RIDs in the old data records.

Thirdly, the use of a mapping table requires less locking for each record in the old copy. Both techniques require a read lock while unloading the old record. However, use of a mapping table involves (1) *no* lock while reloading (and adding an entry to the mapping table) and (2) *no* lock while applying the log (and translating the RID). Use of a forwarding field can require (1) a write lock while reloading (to write a new RID in the old record) and (2) a read lock (or perhaps a write lock) while applying the log (to translate the RID and perhaps change an insertion's estimated new RID to the actual new RID). Thus, in a database environment, the use of a mapping table may be faster and may allow more concurrency in the database. However, for reading or writing just the forwarding field, the environment of O'Toole et al. does not require locking.

The fourth advantage is avoidance of extra space (which is permanent) in each data record for the forwarding field. In the most general case, an implementation of the field can require the space, although O'Toole et al. implemented the field in space that already existed but was previously unused.

A forwarding field has these advantages:

(1) It requires only one field for storing the new RID (which involved no new space in the implementation of O'Toole et al.), instead of fields for both the old RID and the new RID (in a temporary mapping table).
(2) It seems simpler; it avoids the extra structure (a mapping table).

The trade-offs between the two techniques illustrate how details of an environment can influence design decisions. The two strategies perform the same function (changing references in log entries during fuzzy reorganization), but differences in factors like allowance of deletions, requirements for locking, and availability of unused space can imply that different techniques are appropriate.

3.5.4. *References in Data.*

A third context of changing of references involves data that contains references to other data. Some of the strategies also handle references in indexes, not just in data.

3.5.4.1. Garbage Collection. We will discuss garbage collection by copying in Section 4.4. When following a reference from a data object X to the old location of another data object Y, several strategies for garbage collection by copying perform these steps:

(1) If Y has not yet been copied, then copy it, and add (to the old copy of Y) a pointer to the new copy of Y.
(2) Change the reference in X.

3.5.4.2. Lakhamraju et al. For an object-oriented database, Lakhamraju et al. [2000; 2002] describe a strategy for changing references. This strategy, which seems applicable to both approaches C and B in Figure 2, finds the parents of moved objects and modifies the references in each parent. The approach has similarities to tracing of references in garbage collection. The database is partitioned, and the authors assume that given a reference to an object, it is easy to find the partition that stores the object. Each partition has a table *ERT* of incoming references from outside the partition; the ERT stores the identifiers of the parents and the identifiers of the children.

Reorganization takes place for one partition at a time. During reorganization, a temporary reference table *TRT* saves all pointer insertions and deletions that occurred after reorganization started. The system can maintain the TRT based on the log. Here is the procedure for reorganization:

(1) Starting at the ERT for the partition, the reorganizer traverses the graph of object references, using latching but not locking. The reorganizer finds the set of referenced objects in the partition and, for each object, an approximate list of parents that reference the object. Such a list is only approximate, because users can update concurrently. The reorganizer then similarly traverses any objects that appear in the TRT but have not yet been traversed.
(2) For each referenced object O, the reorganizer finds and locks an exact set of parents. It does this by finding the parents in the approximate list (omitting former parents that are not current parents) and then by processing the TRT similarly. After finding and locking the parents, the reorganizer copies O, corrects references to O in the parents and in the ERT of this partition, changes the identifier for O in the lists of parents for the children of O in this partition, updates the ERTs of partitions that contain children of O, deletes the old copy of O, and unlocks the parents of O. The authors also discuss an alternative style of locking, which locks O, moves O, and locks one parent at a time to change references.

The authors argue correctness of their algorithms, and they discuss alternatives for recovery and restart.

Lakhamraju et al. describe performance experiments, which use an implementation. The experiments compare three strategies for reorganization. One strategy, called PQR, is *offline* reorganization of one partition; this includes locking other partitions' parents of this partition's objects. IRA is *online* reorganization of one partition, using the original style of locking. The degenerate third strategy, NR, does not perform reorganization. The strategies do not require the database to reside in main storage. However, to concentrate on testing the concurrency, the experiments use a database that resides in main storage. The default values for parameters include 4080 objects per partition, a multiprogramming level of 30, and a probability of 0.5 that a transaction updates.

In the results, IRA always takes longer than PQR to reorganize the partition. With respect to user performance (response time and throughput), NR surpasses IRA (often not by much) in all cases except an update probability of 0.95, and IRA always surpasses PQR (often significantly). The experiments vary one parameter at a time:

—As the multiprogramming level rises from 1 to 60, the throughput rises sharply at first; eventually it falls slowly. The gap in throughput between IRA and PQR exceeds the gap between NR and IRA. As the multiprogramming level rises, both gaps usually narrow, and the user response times increase. At a multiprogramming level of 30, the throughputs for NR, IRA, and PQR are 35, 33.7, and 28 transactions per second, respectively, and the average response times are 819, 861, and 1030 milliseconds.

—As the number of objects per partition rises from 1000 to 8500, the throughputs for NR and IRA decrease slightly (from 36 to 35 for NR and from 34 to 33 for IRA), and the throughput for PQR decreases significantly (from 33 to 21). The response time for NR and IRA varies little; it is around 810 for NR and around 870 for IRA. The response time for PQR rises from around 880 to 1400, since the reorganization locks a larger partition for a longer time and thus blocks users for longer.

—As the probability of update rises from 0 to 0.95, the throughputs for NR and IRA decrease significantly (from about 49 to 26). The throughput for PQR decreases slightly (from 28 to 25) until the probability is 0.9; then it decreases more (to 22). The response times for NR and IRA increase significantly (from about 590 to 1100). The response time for PQR increases by less (from about 1030 to 1190) until the probability is 0.9; then it increases more (to 1260).

The results indicate that user performance with IRA is almost as good as user performance without reorganization.

3.5.4.3. Huras et al. A strategy for restoration of clustering in place via a reorganizer [Huras et al. 2005] (discussed in Section 4.1.2) introduces some new types of records, containing pointers that track reorganization's movement of records; reorganization can change a record from one type to another. Huras et al. describe how user transactions follow or ignore the various pointers. The reorganizer performs these phases to move a record:

MOVE : Move the data, insert a temporary pointer in the old location of the record, log the changes, insert a new pointer in indexes, and mark the old pointer as old in indexes.

CLEAN : Delete the old data (including the temporary pointers), log the changes, and delete the old pointer in indexes.

3.5.4.4. IBM. In one strategy for online reorganization [IBM 2005, pp. 370–393], discussed in Section 4.1.3, suppose that as a result of reorganization, a pointer (from a logical relationship or a secondary index) becomes invalid. When a user activity accesses the pointer, the DBMS can detect the invalidity by finding that the partition identifier or reorganization number stored with the pointer does not match the identifier or number associated with the partition. Then, the DBMS obtains the correct pointer value from the partition's associated information, and it corrects the pointer.

3.5.5. References in Users' Activities for Reorganization in Place. Our final context of changing of references involves users' activities whose processing involves references to data. Suppose that users release some locks before the end of a transaction, to improve concurrency and thus throughput. Here, transactions must be able to tolerate the inaccuracy that results from other transactions' possible concurrent updates of *some* records. However, suppose also that online reorganization includes changing the assignment of records to pages, as in restoration of clustering. If a user transaction scans data, reorganization might move a record from behind the user's position to ahead of the position, or vice versa. A transaction might experience massive inaccuracy that results from multiple reading or skipped reading of *many* records that reorganization moves. Iyer and Sockut [2002] eliminate this massive inaccuracy for queries by maintaining *movement lists* for each transaction during reorganization. Such a list tracks the reorganizer's movement of records across a user's position within a scan of data. The system uses the lists to correct the behavior of user transactions, which thus process each qualifying record just once, not twice or never.

This approach accommodates several possible types of access by user transactions:

—In a *table space scan*, a transaction reads every page of data, and it returns the records that satisfy the query's predicate.

—An *index scan* involves scanning the relevant subset of an index, reading the RIDs in that subset, reading the corresponding records, and returning the records that satisfy the possible part of the predicate that does not involve the indexed columns.

—For each of the several indexes in a *multiple index access*, the transaction scans the relevant subset of the index, copies the RIDs into a list, and merges the list into the composite list from previously scanned indexes. The transaction then reads the corresponding records and returns the records that satisfy the possible part of the predicate that does not involve the indexed columns.

The reorganizer operates by scanning an index that determines clustering, moving records to reflect the desired clustering, and modifying the movement lists to reflect the movement. For each transaction, the reorganizer maintains the types of movement lists that are appropriate for the transaction's type of access:

—For a table space scan, an index scan, or the record-processing phase of a multiple index access, the reorganizer maintains two lists. A *forward ordered list* (*FOL*) contains the new RIDs of records that the reorganizer has moved forward across the user's position. A *backward ordered list* (*BOL*) contains the new RIDs of records that the reorganizer has moved backward.

—For the index-scanning phases of a multiple index access, the reorganizer maintains a *differential list* (*DL*). A DL contains old RIDs of records that the reorganizer has moved forward, new RIDs of records that the reorganizer has moved backward, both the old and new RIDs of records that the reorganizer has moved from one location behind the user's position to another location behind the user's position, and flags to distinguish these three types of movement.

A movement list has similarities to a physical-to-physical mapping table; each entry contains an old RID or a new RID. Except for movement behind the user in a DL, however, a movement list omits a mapping table's explicit *pairing* of old and new RIDs. Iyer and Sockut describe details of maintenance of the movement lists.

The system modifies the normal behavior of user transactions by using the movement lists. If the RID of a scanned record is in a transaction's FOL, the transaction skips the record, which was already processed earlier during the scan. The transaction examines the BOL and processes records whose RIDs are in the BOL, since the transaction does not find such records by scanning. A transaction uses new and old RIDs in the DL to add or delete RIDs in its RID lists.

3.6. General Issues in Performance

Many of the design issues that we have discussed affect performance, of course. Now we concentrate on performance, but here we discuss only issues that are *general*, that is, applicable to a variety of strategies for a variety of specific categories of reorganization. Sections 4, 5, and 6 will include performance issues, models, and measurements that are *specific* to individual strategies for maintenance, physical redefinition, and logical redefinition. Those sections also include more information (on topics other than performance) for some of the work that we cite in this section. Here we will discuss background topics on performance during online reorganization, policies relating to trade-offs between users' performance and reorganization's performance, and criteria for scheduling of online reorganization.

3.6.1. Background. Typically, online reorganization and user transactions both operate more slowly than they would operate if they executed separately (serially). Here are some reasons:

—Of course, users and reorganization compete with each other for resources, for example, input/output paths, processor time, locks, and buffers in main storage.

—Reorganization might move some data away from the first location that a user activity searches.

—Strategies that involve incremental reorganization during users' activities increase the elapsed times both for those activities and for completion of reorganization.

—Similarly, reorganization strategies that involve interpretation during users' activities increase the time to execute those activities.

—Instead of one coarse-grained lock that offline reorganization might use, online reorganization might use many finer-grained locks; this slows reorganization.

—Some strategies involve an extra step of the reorganization's reading and processing of users' updates from a log or differential file.

—Suppose that we use a reorganization strategy in which the reorganization produces data structures to record its activities, for example, movement of data. User transactions or a reorganizer consumes these data structures. Here we might suspend the reorganization activities that produce the structures if the allocation exceeds a threshold and threatens to exhaust storage. After consumption of the structures below a threshold, we resume the reorganization activities.

Some of the slowdown that we noted above is visible in the results of a performance model [Sockut 1997; 1998]. It is an open queuing model that applies to resource contention by users and an online reorganizer. It concentrates on disk activity, and it also considers processing. There can be any number of disks per I/O channel. This model is not specific to one strategy or one category of reorganization (e.g., restoration of clustering). Users have higher priority than reorganization; a step of reorganization can begin only when no user requests are present. The model relates to two policy topics: suspension of reorganization during periods of peak usage and the granularity of data transfer by reorganization. The model predicts the user response time with and without online reorganization. These predictions allow calculation of reorganization's *degradation* of users' performance, which is the ratio of (1) reorganization's increase in the user response time to (2) the user response time without online reorganization. The model also predicts the time to perform a step of reorganization (and hence the time to reorganize the database), for both online and offline reorganization. We will discuss the main parameters that are varied, the results of the model, and the results' implications for policies that we might choose for online reorganization:

Users' *utilization* of disks means the fraction of time that the disks are busy serving users; it is a measure for the user workload. The user utilization varies from 0 to a degenerate value of 1. As the user workload increases, these results occur:

—User response time increases, since users' queues grow. Of course, this well-known result occurs even without online reorganization.

—Reorganization's *degradation* of users' performance *decreases*, since the rarely served reorganizer causes less interference. However, the degradation still exists for all values of users' utilization less than 1.

—Users' degradation of reorganization time increases, since the reorganizer has less service, and users can interfere with the reorganizer's locality of reference.

For a database whose user workload varies, we might reorganize mainly during relatively slack periods; we might suspend reorganization during periods of peak usage. We will discuss this scheduling criterion later. If we combine suspension with a concurrency control granularity of an entire table, we have incremental offline reorganization during slack periods. This combination suffices only if we do not require 24-hour availability.

The granularity of data transfer by reorganization varies from 1/4 of a track to 1 track. As the granularity becomes coarser, user response time increases, since users must wait longer for a reorganization step. However, the time to reorganize the database decreases, since the reorganizer has more efficient accesses, and users interrupt reorganization less frequently. Therefore, if users' performance is more important than reorganization's performance, we might use a fine granularity of data transfer by reorganization.

For a utilization of 0.5 and a granularity of data transfer of 1/4 of a track, a value for the degradation of users' performance (considering just disk activity, not processing) is 30%. With respect to performance, the author considers online reorganization to be feasible if the user workload is not too high.

Batra [1989] modifies the model of Sockut. If, after a user request at the disk completes its service, there are currently no other user requests, then this system pauses instead of serving the reorganizer immediately. If additional user requests arrive during the pause period, the disk serves them. If no more user requests arrive during the pause period, then a step of reorganization begins. The results of this model are similar to the results of Sockut's model (described above).

3.6.2. Trade-Offs between Users and Reorganization. Some of the discussion above dealt with policies that relate to trade-offs between users' performance and reorganization's performance, for cases where we use a reorganizer. Now we will discuss such trade-offs in more detail. If a purpose of reorganization is to benefit user activities that will execute in the near future, the speed of reorganization can be an important consideration, because slowing the reorganization will delay the benefit to those activities. If reorganization is only to benefit user activities that will execute in the distant future, any time of completion of reorganization can be acceptable if it precedes the desired time of introduction of those activities.

Of course, slowing of reorganization not only delays the completion of reorganization but also can lengthen the period when online reorganization degrades users' performance. In the context of online construction of indexes (Section 5.1), Srinivasan [1992] and Srinivasan and Carey [1992b] discuss how the duration of reorganization and the degradation caused by reorganization *both* affect user performance. Their performance model predicts the *loss* in the potential transaction processing in the database system. The loss is the reorganization time multiplied by the decrease in user throughput during reorganization.

To reduce reorganization's degradation of users' performance, a possible policy is to favor users over the reorganizer when allocating resources [Söderlund 1980; 1981a; 1981b; Maheshwari and Liskov 1997]. Here are some specific considerations that deal with favoring users:

—A reorganizer may pause between its steps, to reduce its competition with users [Sockut 1997; 1978; Birrell and Needham 1978; Batra 1989; Smith 1990; Amsaleg et al. 1995; 1999; IBM 2005, pp. 370–393]. Suspension of reorganization during some periods resembles an extreme case of pausing. In a general context of background processing (i.e., not specifically database reorganization), where the completion of each user service begins an interval during which the background process cannot execute, Eggert and Touch [2005] measure the effects of the length of the intervals. Long intervals reduce the impact on users, but short intervals speed the background processing. For online garbage collection for volatile storage, Bacon [2007] discusses limiting the collector's activity, to assure that users are able to execute during at least a specified fraction of the time.

—A reorganizer may use a fine granularity of data transfer or of concurrency control. This policy decreases the amount of data that is unavailable to users and thus can decrease the response time for users [Sockut 1977, 1978; Bastani et al. 1988].

—Similarly, a reorganizer's locking may have a short duration or be preemptable by users. Salzberg and Dimock [1992] suggest keeping a reorganization step short. If the reorganizer executes at a low priority, then acquisition of locks by the reorganizer might lead to a convoy of users waiting for the reorganizer. Possible solutions to this convoy problem include preempting the reorganizer and raising its priority.

—Another important resource is the buffer in main storage. In a simulation model, Omiecinski et al. [1992; 1994] allocate 10–50% of the buffer to reorganization and the rest to users. They found that increasing the reorganization's fraction of the buffer speeds reorganization but decreases user throughput during reorganization. In modeling another strategy, Söderlund [1980; 1981a; 1981b] found that varying the size of the buffer pool (over his range of values) has no major influence on feasibility of online reorganization. Other work Brown et al. [1993; 1996] discusses dynamic adjustment of the allocation of storage buffers to achieve different response time goals of different classes of transactions; this work is not specific to reorganization.

However, a policy of favoring users slows the reorganization, of course. Løland and Hvasshovd [2006] report that reducing the priority of the reorganizer reduces user degradation but increases the reorganization time. For some cases of maintenance, which we survey in more detail later, the slowing of reorganization might even *harm* users' performance:

—An advantage of giving the reorganizer an equal priority instead of a lower priority is shortening of the duration of maintenance, which shortens the period of competition with users and speeds the reduction of structural degradation. The results of Zilio [1998] indicate that under heavy workloads for users or heavy workloads for reorganization, we may minimize the overall degradation of users' performance over time by giving reorganization an equal priority instead of a lower priority.

—For maintenance that makes more storage available to users (as in garbage collection or in cleaning in a log-structured file system), we might favor the reorganizer, to avoid exhaustion of storage [Seltzer et al. 1993].

—Favoring users might even starve the reorganization, causing an increase in structural degradation and thus an increase in users' performance degradation [Bastani et al. 1986]. When the degradation exceeds a threshold, we might give reorganization a higher priority [Breitbart et al. 1996; Vingralek et al. 1998].

The desirability of preferential allocation needs more research.

3.6.3. Scheduling of Online Reorganization. Other policy decisions involve scheduling of reorganization. Some possible criteria for scheduling [Sockut et al. 1997, p. 430] involve expected characteristics of applications:

(1) If the user workload varies, we may reorganize mainly during relatively slack periods, for example, Sockut [1977, 1978], Troisi [1994, p. 19], Zabback et al. [1998], Bratsberg and Humborstad [2001], and Løland and Hvasshovd [2006, p. 420]. We may suspend reorganization during periods of peak usage. Of course, this policy does not apply if the user workload has no significant variation. This policy reduces the number of users whom reorganization delays, and it avoids changing users' performance from bad to worse during peak usage. The policy can also speed *individual steps* of reorganization by reducing competition from users. If reorganization can complete within one slack period, this policy can also reduce reorganization's reading of users' updates from the log (for reorganization strategies that use the log in this way). However, if the reorganization requires longer than one slack period, suspension slows the completion of reorganization, and, for strategies that read the log, suspension greatly *increases* reorganization's total required reading of users' updates from the log.

(2) If the tolerance of delay varies among the applications, we might reorganize mainly during execution of high-tolerance applications. We might suspend reorganization during low-tolerance applications, to avoid irritating users of those applications. For example, a batch program that produces a report at night can have a higher tolerance of delay than interactive query processing during normal business hours.

(3) Execution of a long-running user transaction can increase the time required to quiesce transactions. Therefore, if the reorganization involves quiescing, then scheduling online reorganization when no long-running transactions are expected to execute may speed online reorganization.

These criteria for scheduling can contradict each other. For example, a database might experience most of its interactive use during normal business hours. The second criterion above implies that we should reorganize at night, to avoid irritating the interactive users. However, for the identical reason (avoidance of irritating interactive users), an installation might schedule its heavy batch updates for the night. Then the first criterion above implies that we should *not* reorganize at night. As with many topics in online reorganization, there are trade-offs.

3.7. Activation of Maintenance

One issue that relates closely to performance is activation of maintenance. Maintenance must take place repeatedly, and if the DBMS does not activate maintenance automatically, the DBA must decide whether to activate maintenance. A DBA might activate maintenance when degradation exceeds a threshold, or a DBA might activate maintenance at fixed times (e.g., alternate weekends) without measuring the degradation each time.

Activation, which applies to maintenance and perhaps to some types of physical redefinition, deals with deciding *whether* to reorganize. In contrast, scheduling of reorganization, which we discussed in Section 3.6.3, applies equally well to physical and logical

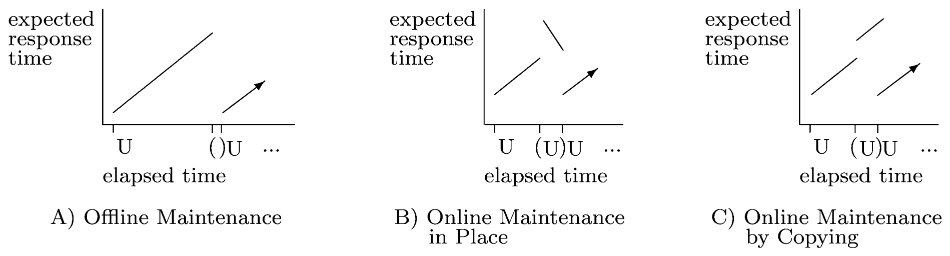

Fig. 5. Expected response time of user transactions vs. elapsed time.

redefinition, not just maintenance. After we have decided to reorganize, scheduling deals with deciding *when* to execute reorganization, for example, perhaps mainly at night. Some of the work on activation also deals with scheduling.

We will sketch issues in activation of offline maintenance, for comparison with online maintenance. Then we will discuss issues and survey policies in activation of online maintenance. Most of the discussions can apply to a variety of specific categories within maintenance, with one caveat. To restore performance or storage allocation, the DBA typically has the flexibility to choose a threshold for the amount of degradation that justifies a reorganization. The caveat is that if the storage allocation (including any maximum "extension" of the original storage area) has degraded to a level that actually prevents insertions, the DBA should reorganize as soon as possible (and arguably should have reorganized earlier). This caveat (especially the advice to reorganize *before* insertions have become impossible) is particularly important for online garbage collection and for cleaning in a log-structured file system.

3.7.1. Activation of Offline Maintenance. We start with *offline* maintenance. Chart A of Figure 5 plots the expected response time of user transactions vs. the elapsed time, for a DBMS that uses offline maintenance. For simplicity, the charts in the figure are stylized; they do not precisely represent one specific DBMS, database, workload, type of structural degradation, or type of maintenance, and they ignore growth in the database. Over time, structural degradation degrades users' performance and storage allocation. Therefore, the plotted response time in chart A has a positive slope during usage of the database (during the periods marked "U"). Alternatively, we could plot throughput, which would have a negative slope during usage. Unlike our simple figure, the degradation of response time in an actual database need not be linear. In chart A, upon satisfaction of a criterion (e.g., structural degradation that exceeds a threshold), we activate offline maintenance, which executes during a period marked in parentheses. After maintenance, the response time should return to a low value, and we resume the cycle of usage and maintenance. Of course, frequent maintenance causes frequent offline periods, and rare maintenance allows high growth in the response time and perhaps in storage requirements. We will discuss charts B and C (for online maintenance) shortly.

Since the activation models that we are about to summarize deal with *offline* maintenance, we will not survey them in detail: Shneiderman [1973], Winslow and Lee [1975], Yao et al. [1976], Maruyama and Smith [1976], Lohman and Muckstadt [1977], Tuel [1978], Mendelson and Yechiali [1981], Ramírez et al. [1982], Batory [1982], Heyman [1982], Teorey and Fry [1982, pp. 392–397] and Park et al. [1989; 1990] present models for deciding whether to activate offline maintenance. Structural degradation and file growth can each increase the cost of user accesses. The models above analyze factors such as increases in access cost caused by structural degradation, increases in access cost caused by file growth, decreases in access cost caused by maintenance, and the cost of performing maintenance. The models differ in their assumptions for factors like linearity of structural degradation, linearity of growth, and uniformity of the lengths of periods of usage. The models lead to recommended criteria for activating maintenance, with a goal of minimizing the total cost of user accesses and maintenance over a period. For an object-oriented database, Cheng and Hurson [1991] also develop formulas to minimize the total cost, and they also mention the possibility of online reorganization. Also, for offline physical deletion of garbage nodes in an index, the goal of Chen and Hassan [1995] and Chen [1995] is to choose the activation interval that minimizes the average response time per user operation; this model takes into account the length of the offline period. Park and Sridhar [1997] develop a performance model to determine a time to activate reorganization of an index, before needing to split a region of storage.

For commercial DBMSs, recommendations typically involve simple criteria for activating offline maintenance. The DBA examines statistics that measure structural degradation or performance degradation. If degradation exceeds a threshold, the DBA activates maintenance [Bracht et al. 1991; Smith 1994]. The threshold can vary with characteristics of the database and with the applications' tolerance of degradation. A DBMS might automate the examination of statistics, comparison with a threshold, and activation of maintenance [Katabami et al. 1997; Friske et al. 2003a]. For deciding which partitions of a database should undergo maintenance, it is possible to rank the partitions, for example, by the amount of fragmentation or the expected amount of improvement in performance [Plow et al. 2008].

We will discuss one research study of activation of offline maintenance [Chen and Banawan 1999], because it compares offline maintenance with incremental maintenance during users' activities. This study involves two strategies:

(1) The storage organization is a single-level sorted file or binary tree, with incremental maintenance during users' activities.
(2) The storage organization is a multilevel data structure, which is a hierarchy of structures, ordered by decreasing search efficiency. Specifically, this organization is a two-level sorted file. The system defers physical deletion and also defers movement of inserted records to their final locations. Maintenance performs the deferred activities. If a server has sufficient idle time, maintenance can execute just during idle periods; here the system aborts a step of maintenance if a user request arrives. In addition, complete maintenance, which is offline, takes place periodically; here users must wait.

The system can limit the overhead of offline maintenance by activating a complete reorganization *only* after execution of enough user operations over which to amortize the cost. The authors study activation points, with a goal of maximizing performance.

Chen and Banawan develop open and closed queuing models. An example involves a data base server, whose steady-state average number of records R is 500. The *degradation level* is the number of records (including logically deleted records) in excess of R. For the two-level strategy, offline maintenance begins when the degradation level exceeds a threshold. In the experiments, the threshold ranges from 1 to 1024. In the results for the open model, for a threshold of 1, the average service rate (for almost all reported arrival rates of user requests) is lower than the average service rate for the single-level strategy with incremental reorganization; reorganizations are too frequent. For a threshold of 8 to 256, the average service rate is higher than the rate for the single-level strategy. For higher thresholds, the service rate depends on the arrival rate:

—For low arrival rates (for which idle-time maintenance is sufficient), the service rate stays high.

—For high arrival rates, the service rate eventually falls below the single-level strategy's rate; reorganizations are too rare. The maximum service rate occurs at a threshold of about 64.

Also, as the arrival rate grows, the response time for the two-level strategy remains finite longer than the response time for the single-level strategy remains finite.

3.7.2. Considerations for Activation of Online Maintenance. We return to *online* maintenance. Chart B of Figure 5 plots the possible behavior of the expected response time of user transactions vs. the elapsed time, for a DBMS that uses online maintenance *in place*. Here, degradation of users' performance has two causes:

—As in the offline case, structural degradation degrades performance and storage allocation. Therefore, the plotted response time has a positive slope when usage but no maintenance takes place (during the periods marked "U" without parentheses).

—During online maintenance and usage (during a period marked "U" with parentheses), the maintenance itself competes with the users and degrades their performance, as we explained in Section 3.6. Of course, frequent maintenance causes frequent competition. We depict the degradation via the jump in the plotted response time when online maintenance begins or ends. The size of the jump (which results from competition from maintenance) does not necessarily equal the size of the gradual decrease in response time *during* maintenance (which results from reduction of structural degradation). Also, the change in the response time when online maintenance begins or ends might be gradual, not a jump. Online maintenance can take longer than offline maintenance, since the users compete with the reorganization.

For online maintenance to succeed, it must decrease *long-term* structural degradation as quickly as users' updates increase structural degradation, as we noted in Section 2.2. We depict this success via the return of the plotted response time to a low value.

We noted earlier that for *offline* maintenance, satisfaction of a criterion can activate maintenance. An example of a criterion is structural degradation beyond a threshold. The maximum degradation of users' performance occurs just before the activation of offline maintenance in chart A. If *online* maintenance uses the *same* value for such a threshold, then the maximum degradation of users' performance *during* online maintenance might exceed the maximum degradation for the offline case, because online maintenance itself degrades users' performance, as we mentioned above. To prevent a higher maximum degradation of users' performance, online maintenance might use a *lower* value for a threshold (if a purpose of maintenance is to improve performance), as we show in chart B. For online maintenance that we discuss in Section 4.2.2 for a commercial DBMS [Smith 1990], the threshold is *not* lower than the threshold for offline maintenance. However, customers of this DBMS have been reorganizing mainly to recover space, not to improve performance.

During online maintenance, users' updates also cause continued structural degradation and thus performance degradation in parts of the database that the reorganization's pass through the database has *already* reached. This structural degradation will remain after completion of this pass. Chart B depicts this degradation via a minimum response time (after completion of maintenance) that slightly exceeds the minimum of chart A. This higher minimum response time and the possible lower threshold for activation (discussed above) may also slightly shorten the time between successive reorganizations.

Chart C plots the possible behavior of the expected response time for a DBMS that uses online maintenance *by copying*. Users' updates during online maintenance might cause structural degradation in the new copy. For example, in fuzzy reorganization, application of the log can cause overflow in the new copy. Therefore, the minimum response time in chart C slightly exceeds the minimum of chart A. Two differences between charts B and C reflect factors that we discussed at the beginning of Section 3.3:

—If maintenance by copying reads but does not write the copy that users access, the maintenance itself might cause less degradation of users' performance, as we depict via a smaller jump in chart C.

—Maintenance by copying benefits users only *after* we begin to direct users' accesses into the new copy (which often means at the end of maintenance); maintenance in place might *begin* to benefit users *immediately*. Therefore, during online maintenance, the slope is still positive in chart C but negative in chart B. Of course, in chart C, the amount of increase in response time *during* maintenance is proportional to the duration of maintenance. However, Section 4.4 includes some copying strategies that direct users' accesses into the new copy at the *start* of maintenance; chart C might be less applicable to those strategies.

Online maintenance also raises a question that does not apply to offline maintenance. After maintenance has completed a pass through the database, should we begin another pass immediately, or should we wait until structural degradation exceeds a threshold? Beginning immediately minimizes the part of users' performance degradation caused by structural degradation, but it perpetuates the part caused by competition from online maintenance. Probably no single comparison of these two causes of performance degradation can apply to all databases and all maintenance strategies, since different databases have different rates of updates (which lead to structural degradation), and different strategies have different amounts of interference with users. Our intuition is that typically, waiting is better. Smith [1990], O'Toole et al. [1993, p. 167], and McIver and King [1994] also wait. However, if the available storage is almost exhausted, and maintenance can make more storage available soon (as in garbage collection or in cleaning in a log-structured file system), then beginning immediately can be better.

In Section 3.6.2, we noted that a reorganizer might pause between its steps, to reduce its interference with users. We briefly compare two combinations of policies:

—After maintenance has completed a pass, we begin another pass immediately. We use a long pause between steps *within* a pass of maintenance.

—After maintenance has completed a pass, we wait until structural degradation exceeds a threshold; then we begin another pass. We use a short pause (or perhaps no pause) between steps of maintenance.

It is possible to select values for the threshold and the length of pauses so that the two combinations result in similar frequencies of activation. However, the first combination can involve more overhead for logging and recovery of reorganization. Also, if the mere presence of the reorganizer (even during pauses, when it is not executing) degrades users' performance, then the first combination causes more degradation of users' performance.

For a partitioned database, the topic of activation (*whether* to reorganize) relates to another topic: choice of the partition to reorganize next (*what* to reorganize). One possible criterion for the choice is the benefit (to choose the partition with the most degradation). Another is the cost (to choose the partition with the lowest cost of reorganization). A third is the ratio of benefit to cost (to choose the partition with the highest rate of reduction in degradation per unit of reorganization work). We will discuss such criteria and modeling of their performance when we survey rebalancing of parallel or distributed data (Section 4.3), policies to choose a partition for garbage collection

(Section 4.4.8), and cleaning (reclamation of space) in a log-structured file system (Section 4.5).

3.7.3. Policies for Activation of Online Maintenance. Continuing our discussion on activation of online maintenance, here we survey work on policies for activation. The models and analyses involve the contexts of physical deletion of logically deleted structures, partitioned garbage collection, priorities of users vs. reorganization, and restoration of clustering.

3.7.3.1. Bastani et al. For use of a reorganizer for maintenance, Bastani et al. [1986] present queuing models and performance results. This work involves aspects of activation and aspects of scheduling. The reorganizer operates at a server. The data structure is a search tree, and an example of a maintenance task is physical deletion of a tree node that is marked as logically deleted. The service time for client requests increases with the number of nodes that are logically but not physically deleted.

In the model for a simple initial policy, the reorganizer executes *only* when there are no client requests. In steady-state, the mean response time is infinite. This result has an intuitive explanation: If the system degrades, then the probability of more degradation increases. If, since the most recent reorganization, the system has served many client requests (some of which inserted and/or logically deleted nodes), then the service time for clients has grown. There is a probability that the system has reached a state in which the mean service time exceeds the mean time between arrivals of client requests.

Another policy is to perform maintenance after each M client requests. Here the mean waiting time for client requests is finite, for a range of values of M. In the results for the model for this policy, as M increases from 1, the mean waiting time for clients initially decreases (because frequent maintenance causes overhead and causes accumulation of waiting clients), and then the time increases (as additional logically deleted nodes accumulate between reorganizations). Thus, it is possible to calculate an optimal value for M. Yet another policy is to perform maintenance after each T seconds if possible (without preempting clients). As T increases, the client waiting time decreases and then increases.

In a final policy, after each client's departure, there is a probability (1-p^d) that the system invokes the reorganizer. The probability increases with the level of degradation d. Each such execution of the reorganizer performs complete maintenance. Also, during idle periods, the reorganizer can perform partial maintenance. In the results of a simulation, as p increases from 0 to 1, the mean waiting time decreases and then increases. The authors implemented this policy for a directory server for a hierarchical file system. The maintenance physically deletes logically deleted slots, and it restores clustering. The authors also implemented an alternative of simple maintenance during client activities at the server. For use of a reorganizer, the measured mean response time increases as the arrival rate increases. For maintenance during client activities, the response time has little variation (for the range of arrival rates that the authors show). The client response time with a reorganizer exceeds the client response time with maintenance during client activities, perhaps due to locking overhead and contention with the reorganizer. For these results, the steady-state size of the directory is 500.

3.7.3.2. Cook et al. For garbage collection (operating on one partition at a time) in an object-oriented database system, Cook et al. [1996] analyze policies for activation. The activation concepts, perhaps expressed more generically, should also apply to other categories of maintenance. The goal is for the DBMS to choose a rate of garbage collection, given a DBA-specified target value for either (1) the fraction of all I/O operations that take place for garbage collection (as opposed to taking place for application programs) or (2) the fraction of all objects that are garbage. The evaluation method uses partition choice policy 2 in the partitioned collection of Cook et al. [1994; 1998] (which we discuss in Section 4.4.8), with copying and compacting within a partition. For each partition, this policy maintains a count of the pointers that referred to objects in this partition and have been updated since the previous garbage collection of this partition. The policy chooses the partition with the largest count. The evaluation method uses simulation, with the I/O buffer size equal to the partition size (12 8-kilobyte pages). The *simulation* uses the assumption that the entire database (3.7 to 7.9 megabytes) is locked during garbage collection. However, the activation *policies* could also be used if the locking applied to only the partition being collected.

When using a target fraction of I/O operations, the authors assume that the number of I/O operations in an execution of garbage collection does not differ significantly from the number in the previous execution. The activation policy uses the history of the counts of I/O operations. In the results of the simulation, the activation policy succeeds in yielding an actual fraction that is very close to the target fraction, which ranges from 2% to 25%.

When using a target fraction of objects, Cook et al. assume that the amount of garbage collected in an execution of garbage collection approximately matches the amount in the previous execution. The activation policy must estimate the amount of garbage in the database. Two heuristics are (1) the number of bytes reclaimed in the most recent collection of a partition multiplied by the number of partitions and (2) the historical amount of garbage per pointer overwrite multiplied by the number of pointer overwrites. The overwriting of a pointer indicates the possible creation of garbage. In the simulation, the target fraction of objects ranges from 5% to 30%. The activation policy succeeds in yielding an actual fraction that is close to the target fraction for the second heuristic but not for the first, because partition choice policy 2 of Cook et al. [1994; 1998] is biased toward finding a partition with a higher-than-average amount of garbage.

3.7.3.3. Zilio. In addition to the basic question of whether to perform maintenance, a second question (for use of a reorganizer) involves whether to give the reorganizer a lower priority than users in allocation of resources, especially processor time. An advantage of giving the reorganizer a lower priority is reduction in reorganization's degradation of users' performance. An advantage of giving the reorganizer an equal priority is shortening of the duration of maintenance, which shortens the period of competition with users and speeds the reduction of structural degradation.

Zilio [1996; 1998] describes a framework for answering both questions above. The framework can also apply to physical redefinition, not just maintenance. This work uses the context of rebalancing of parallel data, but the techniques can also apply to other specific categories of reorganization. Careful placement of data may balance the activity levels among the disks. Over time, users' updates can change the balance of activity by changing the balance of data among the disks. The workload might also change. Zilio describes considerations for both the initial physical design (which we will not discuss) and reorganization. Here the basis for partitioning is hashing of a key. The reorganization improves the balance among disks by changing the assignment of hash buckets to disks, thus moving rows from overpopulated disks to underpopulated disks. During movement, user activities access rows at their original locations, and a differential file holds users' updates.

Zilio presents a cost-benefit analysis for deciding whether to reorganize and deciding the priority of the reorganizer. The benefit of reorganization is a decrease in users' average response time, calculated over a time horizon (the time that the new organization is expected to remain appropriate). The cost is an increase in users' response time during reorganization, calculated over the duration of reorganization. The principle in the decision-making is to maximize the benefit minus the cost, with constraints that the average response time after reorganization is sufficiently lower than the average response time before reorganization, the benefit exceeds the cost, the response time during reorganization is within a limit, and the duration of reorganization is within a limit. It is possible to plot the average response time vs. the elapsed time, with and without reorganization, until the elapsed time reaches the time horizon. The net gain of reorganization is the area under the curve without reorganization minus the area under the curve with reorganization [Baru and Zilio 1993].

The *break-even time* is the time horizon for which the benefit of reorganization equals the cost of reorganization. A low break-even time is desirable, of course. Zilio's queuing model predicts the break-even time under various circumstances. In most cases, the break-even time for a low-priority reorganizer is less than the break-even time for a reorganizer whose priority equals users' priority. However, for high user utilizations or a high amount of reorganization work to perform, an equal-priority reorganizer can have the lower break-even time, because user response times grow so much in these cases (even without the competition from the reorganizer).

3.7.3.4. Martin and Crown. For IMS databases [Strickland et al. 1982], the technique of Martin and Crown [2003b] recommends what part (if any) of a database to reorganize and whether the reorganization should be online. The reorganization deals with restoration of clustering. The technique performs these steps:

(1) For each IMS record in a list of IMS records to be analyzed, ascertain the amount of improvement (in storage or time) that reorganization would produce. If the improvement meets a threshold, add information about the record to an array.

(2) Aggregate the information and produce summary data in another array; for example, calculate the I/O saved per megabyte moved.

(3) Compare the data above with user-specified threshold values to make a recommendation for reorganization.

3.7.3.5. Summary. The investigations by Bastani et al., Cook et al., Zilio, and Martin and Crown involve different aspects of activation of online maintenance. Some of the strategies that we will survey in other sections also include criteria for activation:

—Activate maintenance when finding both (1) a threshold amount of performance degradation and (2) a threshold difference between the desired clustering and the actual clustering [McIver and King 1994; McIver 1994]. The purpose of including the second condition is to avoid the expense of a reorganization that would yield little improvement.

—Activate maintenance when the benefit of maintenance would exceed the cost. Several studies of rebalancing of parallel or distributed data use this criterion; see Section 4.3.

—Activate maintenance when users have added a threshold amount of data [O'Toole et al. 1993, p. 167]. This work involves garbage collection.

—Activate maintenance when the amount of available free space drops to a threshold. Several strategies for cleaning in a log-structured file system (discussed in Section 4.5) use this criterion.

We consider activation of maintenance to be still a research issue.

4. STRATEGIES FOR ONLINE MAINTENANCE

Most of our discussions until now have involved issues that can apply generically to a variety of categories of reorganization, but now we will survey work on strategies that apply to specific categories within *maintenance*. Later sections survey strategies for physical and logical redefinition. Within maintenance, we cover the specific categories of restoration of clustering, reorganization of an index, rebalancing of parallel or distributed data, garbage collection for persistent storage, and cleaning (reclamation of space) in a log-structured file system. These categories have some overlap. For example, garbage collection reclaims space, reorganization of an index can also reclaim space, and reclamation of space can also restore clustering. Restoration of clustering and reorganization of an index both can include removal of overflow. However, we have categorized the strategies according to the activities that they emphasize. Of course, work in additional specific categories within online maintenance might take place eventually. Our discussions in the sections on maintenance, physical redefinition, and logical redefinition implicitly include the analyses and trade-offs that Section 3 presented for the generic issues.

4.1. Restoration of Clustering

Our first category of maintenance is restoration of *clustering,* which is the practice of storing instances near each other if they meet certain criteria. Popular criteria for clustering include (1) linkage between instances and (2) consecutive values of a key. Such clustering should reduce disk input/output for instances that users often access together. Of course, users' updates can decrease the amount of clustering and thus degrade performance. Also, even if a particular clustering was appropriate for users' former patterns of access, it can become inappropriate (and degrade performance) when patterns of access change [Bennett and Franaszek 1977, p. 532; Söderlund 1980, pp. 26–29; McIver and King 1994; McIver 1994]. We will discuss strategies for (1) incremental restoration of clustering in place during users' activities, (2) restoration in place via a reorganizer, and (3) restoration by copying via a reorganizer. Also, for an analysis of the process that performs maintenance (Section 3.2.2), most of the examples from Weikum [1989] deal with clustering.

4.1.1. Incremental Restoration in Place During Users' Activities.

Restoration of clustering may occur in place during users' activities, as in the strategies discussed below. The strategies reorganize in place with movement between pages.

4.1.1.1. Bennett and Franaszek. The concepts in the strategy of Bennett and Franaszek [1977] can apply to any storage hierarchy in which a unit of data transfer between two levels in the hierarchy contains smaller units. In the description of the strategy, a block (which contains pages) is a unit of transfer from secondary storage into main storage. After the transfer of a block into main storage, the block's pages are then managed individually; i.e., they can be paged out separately. The strategy for clustering involves only the blocks that are already in main storage. Each process has an associated clustering task, which executes a sequence of exchanges of the assignments of pages to blocks. Each exchange involves (1) a referenced page and (2) a page in a cluster point. A *cluster point* is the set of unreferenced pages in a block that contains a referenced page. Each process can have several cluster points.

These are the main characteristics of the strategy:

—The system assigns each page to the first process that references the page during its current stay in main storage. This assignment means that the clustering task for that process is responsible for improving the clustering of that page.

—For each process, the first cluster point is the set of unreferenced pages in the block that contains the page that causes the first page fault for the process.

—When assigning pages to a process, the system appends their identifiers to a list associated with the current cluster point.

—After the size of the list grows to the size of the cluster point, the next cluster point for the process is the set of unreferenced pages in the block that contains the page that causes the next page fault. This cluster point will have its own list of page identifiers.

—When a page from a cluster point is about to be paged out, the system exchanges its block assignment with the block assignment of a page in the associated list. Thus the block containing the cluster point accumulates (and clusters) referenced pages.

—When paging out the last page from a cluster point, the system deletes the list for the cluster point.

Bennett and Franaszek report experiments using a trace of 3 days of transactions on a database (6,000,000 database references, with an average of 38 references per transaction). The physical record size (1693 bytes) constitutes 1 page, and a block contains 8 pages. The size of main storage varies from 1024 to 16,384 pages. For low storage sizes, reorganization lowers the miss ratio (the fraction of references that cause page faults) by about 8%. For high sizes, reorganization has little effect. If reorganization ceases after a period, the miss ratio with reorganized data is still less than the miss ratio for unreorganized data for a while, but later it is not less. Thus, the optimal organization varies over time.

4.1.1.2. Chang and Katz. For an object-oriented DBMS, Chang and Katz [1989] use inheritance and structure semantics to influence the clustering. The inputs to clustering can be users' hints, frequencies of inter-object accesses (which are defined when the object *type* is created), and characteristics of inherited attributes. When creating each data instance, the system chooses an initial placement based on the relationship that is most frequently traversed.

Three parameters control the clustering policy:

(1) When seeking a candidate page for placement, the set of pages that the clustering policy can consider might be (1) only the pages that are in the buffer pool, (2) the pages in the buffer pool plus a limited number of pages that are on disks, or (3) any page in the database (with no limit on the number of pages on disks).

(2) When the preferred candidate page is full, the DBMS might split the page (move some data out) to make room for the new object. Alternatively, it might place the object in the next best page. The DBMS should make this choice in a way that minimizes the expected access cost. The policy on splitting might be (1) never split, (2) calculate an approximately optimal splitting, or (3) calculate the exact optimal splitting (which can be expensive to compute).

(3) The policy might (1) consider user hints or (2) not consider them.

The authors also analyze policies for buffer replacement, which we do not discuss.

Chang and Katz collected access patterns from applications that perform computer-aided design. One statistic is the ratio of the number of reads to the number of writes. One application has a ratio of 6000; the others vary from 0.52 to 170. The authors present a simulation model to analyze the clustering policies. The database size is 500 megabytes. The *fan-out* of component objects in retrievals of structures is the number of components retrieved; it can be in any of three ranges: 3 or fewer, 4–9, or 10 or more. The read/write ratio is 5, 10, or 100. The experiments use all the values mentioned earlier for the three parameters that control the clustering policy. For the first parameter, the limit on the number of pages on disk to consider can be 2 or 10, and the authors also model a policy of no clustering.

One set of results applies to the policy of never splitting. In the results, the response time for no clustering always exceeds the response time for clustering within the buffer pool, which exceeds the response time for other clustering. The improvement due to clustering is greatest (up to 200%) when the read/write ratio and/or fan-out are highest. Clustering also reduces the amount of I/O required for logging. When the fan-out is low, the response time with a limit of 2 pages on the disks (when seeking a candidate page) is less than or equal to the response time with no limit. For a low fan-out, the policy of limiting the candidates to the pages that are in the buffer pool performs well, but for a high fan-out, it performs as poorly as no clustering. As the fan-out rises, a transaction needs more objects, and the response time for no clustering or for clustering within the buffer pool rises more sharply than the response time rises for the other clustering.

Page splitting's effects on the response time vary. For a low read/write ratio, splitting usually increases the response time. For a high read/write ratio, splitting usually decreases the response time.

4.1.1.3. McAuliffe et al. Another discussion of clustering for an object-oriented DBMS [McAuliffe et al. 1998] includes a description of two techniques that have appeared in previous work (near hints and fixed-size clusters):

—When creating an object, an application can specify a *near hint* (a request for placement on the same page as a specified existing object). Typically, a DBMS does not record the hint persistently, so its usefulness is limited to the initial placement, not later maintenance of clustering. This technique also does not tell the DBMS that a particular object will be specified in a near hint for a *future* creation of another object, so the DBMS might use nearby space for unrelated objects. However, a DBMS may let the application programmer specify the amount of free space that the DBMS should leave on pages until overridden by a near hint.

—A *fixed-size cluster* represents an integral number of pages to reserve for future insertions; the DBMS will not mix two clusters on one page. This technique can cause fragmentation, and the size of a cluster might be impossible to predict, leading to eventual overflow.

To reduce the problems noted above, the authors developed *Vclusters* (variable-size clusters), whose size can vary over time. Application programmers or an automatic clustering policy can use the clustering information, which the DBMS stores persistently. When creating an object, an application programmer can specify the identifier of an object that already resides in the desired cluster. When *opening* a cluster (to create several new objects in it), a programmer optionally specifies an estimated size, which the DBMS can use in initial allocation.

McAuliffe et al. discuss the mapping of clusters to pages. A *fragment* is the portion of a cluster that resides on any one page. A cluster consists of a set of *dedicated* pages (pages that contain only objects of that cluster) and a sequence of fragments on *mixed* pages (pages that can contain objects of several clusters). The authors describe two algorithms for maintaining Vclusters:

—*Vcl-dense* has a goal of maintaining good space allocation, even if this requires splitting a cluster across many pages. Every page of a cluster can be mixed. The DBMS tries to keep the size of the ith fragment greater than page size * minimum((2^i)/8, 1), to limit the number of fragments.

—*Vcl-minsplit* has a goal of minimizing the splitting of a cluster across many pages, even if space utilization suffers. Each cluster uses 0 or more dedicated pages and at most 1 mixed page.

When either algorithm allocates a mixed page, it chooses an empty page unless this choice would reduce the utilization of the file below a threshold.

For a cluster whose size is less than 1 page, when the size of an object being created exceeds the free space on the cluster's most recent mixed page, the behavior of the DBMS depends on the algorithm:

—With Vcl-dense, if the most recent fragment's size is less than the minimum size defined above, or there is a mixed page whose free space is at least twice as big as the most recent fragment, then the DBMS copies the fragment to a newly allocated mixed page. Otherwise, the DBMS adds a new fragment on a mixed page.

—With Vcl-minsplit, if the page contains more than 1 cluster, the DBMS copies the fragment to another mixed page, and if the page contains only 1 cluster, the new object goes onto a new dedicated page.

When a transaction's deletion of an object from a page shrinks the page's utilization below a threshold, the DBMS adds the object's cluster to the head of a fixed-size list of clusters, maintained for that transaction in least-recently-used order; this heuristic detects shrinking clusters. When a cluster drops off the end of the list, or when an open cluster is closed, the DBMS can reorganize the cluster. Also, when a transaction commits, the DBMS can reorganize all clusters in the list. The reorganization depends on the algorithm:

—With Vcl-dense, among the pages that have a storage utilization below a threshold, the reorganization moves objects from emptier to fuller pages.

—With Vcl-minsplit, the DBMS reorganizes the cluster only if the cluster resides on more than 1 page. Phase 1 of reorganization moves objects from sparsely populated dedicated pages and the mixed page to denser dedicated pages, and it deallocates any pages that become empty. If any remaining dedicated pages have a storage utilization below a threshold, Phase 2 takes place. Phase 2 combines the fragment on the emptiest of those pages with the fragment on the mixed page (if any), placing it on the tightest-fitting mixed page in the file.

A logical-to-physical mapping table handles changes of references. Reorganization can involve pages that have not yet been brought in from disks, although it usually involves just pages that are already in main storage. The DBMS logs moves, and it undoes them if a transaction aborts.

Performance experiments (using an implementation) compare five algorithms: Vcl-minsplit, Vcl-dense, fixed-size clusters, near hints, and no clustering. The objects form 2-level trees. For the two Vclusters algorithms, each tree is in a separate cluster. For near hints, each leaf uses the root as a near hint. For fixed-size clusters, each tree has several clusters, and a cluster size is 1 page. The database size is 32 megabytes, and the buffer pool holds 512 8-kilobyte pages.

McAuliffe et al. describe five experiments. Here are the experiments and results:

(1) This experiment creates the database (6400 clusters). Fixed-size clusters produce the lowest space utilization; near hints and no clustering produce the highest. The three algorithms that actually create clusters (minsplit, dense, and fixed-size clusters) take longer than near hints or no clustering, because minsplit and dense move objects (requiring more I/O), and the poor space utilization of fixed-size clusters

requires writing more pages. Minsplit takes twice as long as near hints or no clustering.

(2) An application repeatedly chooses a random cluster and traverses all nodes of the tree. The elapsed time for near hints or no clustering is approximately 2.5 times the elapsed time for the three algorithms that actually create clusters. The authors say that near hints result in many split clusters and that for fixed-size clusters, the poor space utilization has little effect on the miss rate for the buffer pool.

(3) An application picks a file, visits 1600 clusters in the file, and then repeats this activity for other files. The buffer pool size ranges from 200 pages (1.6 megabytes) to 1152 pages (9 megabytes). For less than 500 pages, near hints and no clustering have elapsed times up to 3 times that of the other three algorithms. For large buffer pools, however, near hints and no clustering are the fastest. The reason is that with near hints and no clustering, clusters are split over many pages, and working sets do not fit in a small buffer pool. These two algorithms have high space utilizations, so they perform well with large buffer pools, in which the working sets fit. Fixed-size clusters are up to 38% slower than minsplit or dense, because their poor space utilization requires more I/O.

(4) An application creates and deletes clusters and objects, and then it executes as in experiment 2. For the updates, minsplit is the slowest, and fixed-size clusters and near hints are the fastest. The degradations of performance (comparing experiment 2 before and after the updates) are 9% for near hints (since the free space from deletions helps), 10% for minsplit, 25% for no clustering, 28% for dense (due to splitting), and 37% for fixed-size clusters (due to cluster overflows). Minsplit, dense, and fixed-size clusters are faster than near hints and no clustering, as in experiment 2. Minsplit and dense produce space utilizations after the updates that are similar to the utilizations before the updates, but the utilizations of the other three algorithms drop by over 10%.

(5) This experiment measures the effects of accurate and inaccurate estimates of the cluster size. For minsplit, accurate estimates reduce the time to create the database by 28%, and inaccurate estimates have little effect. The estimates have a smaller impact on the time to update the database.

In summary, near hints provide good space utilization but poor clustering. Fixed-size clusters provide good clustering but poor utilization. The two Vclusters algorithms give better performance on query-only workloads and worse performance on updates.

4.1.1.4. Comparison. All the strategies above include reorganization that moves data between pages to improve the clustering. Bennett and Franaszek limit the movement to pages that are already in main storage, McAuliffe et al. do not limit the movement (although they usually use just pages that are already in main storage), and Chang and Katz analyze both limiting the movement and not limiting the movement. Bennett and Franaszek cluster on the basis of recent access patterns, Chang and Katz use structural relationships and user hints, and the presentation by McAuliffe et al. uses only user hints (although their techniques can also apply to structural relationships).

4.1.2. Restoration in Place via a Reorganizer. Several strategies are available for using a reorganizer to move records between pages to restore clustering in place. Different strategies emphasize different activities, for example, following links, minimizing the cost to construct a page, and clustering via an index. Much but not all of the work in this section builds upon other work in this section.

4.1.2.1. Söderlund. The work of Söderlund [1980; 1981a; 1981b] deals with CODASYL-style network structures [Taylor and Frank 1976] and their degradation from overflow and from logical deletion without immediate physical deletion. To achieve clustering, the DBMS places an inserted record in a page that contains a logically related record if possible; otherwise the DBMS places the record in an overflow area. Different relationships (represented by different CODASYL set types) can have different priorities in the clustering. Deletion of a record results in marking the record as deleted but does not immediately deallocate the space. A reorganizer follows links of CODASYL sets and performs maintenance. It delinks deleted records from sets, deallocates their space, compacts, and restores clustering. For this online maintenance to be feasible, it must avoid (1) long-term growth in structural degradation (and thus long-term growth in users' response times) and (2) an unacceptable increase in users' response times due to contention with the reorganizer.

Söderlund's description concentrates on a model for users' performance. The model predicts the effects of structural degradation (over time), of contention with an online reorganizer, and of the reorganizer's restoration of the structures. The model considers a mix of three types of query transactions and four types of insertion/deletion transactions. The long-term number of insertions equals the long-term number of deletions. Here are the parameters that are varied and the results of the model:

—The *turnover rate* is the ratio of the number of insertions (or deletions) during a reference period to the number of records at the start of the reference period. For a reference period of 360 days, the turnover rate has values of 20% and 30%. Increasing the turnover rate increases the work that the reorganizer must perform, of course. However, this increase in work has no major influence on the feasibility of online maintenance. The reorganizer succeeds in preventing long-term growth in the average response time. However, Söderlund also partially investigates higher turnover rates (up to 50%). The model predicts that a higher turnover rate does increase the degradation of users' performance over time and thus exceeds the reorganizer's ability. With respect to performance, the author considers online maintenance to be feasible if the turnover rate is not too high.

—The buffer pool is available to users and the reorganizer. Decreasing the number of buffers in the pool (from 30 to 15) has no major influence on the feasibility of online maintenance.

—The number of records stored per page has values of 10 and 5. Decreasing the number of records per page increases response times but has no effect on the feasibility of online maintenance.

4.1.2.2. Omiecinski and Scheuermann. The strategy of Omiecinski and Scheuermann [1984] and Omiecinski [1985b; 1996, pp. 26–27] involves restoration of clustering by exchanging records between pages. The restoration can apply to an entire file or to a subset of a file, e.g., the most frequently accessed pages. Within the file, the reorganizer locks only the pages that are currently in buffers; users can access the rest of the file. A type of logical-to-physical mapping table maps identifiers to page numbers; reorganization of data does not affect indexes. The order of construction of new (clustered) pages affects the amount of page swapping (input/output) during reorganization, since some of the records needed for a page might already be in the buffer, due to construction of previous new pages. Of course, an order that minimizes the swapping is desirable. Omiecinski presents two algorithms (*static_cost_reorganization* and *dynamic_cost_reorganization*) for choosing an order and performing the reorganization. An assumption is that for each new page *j*, the cardinality of delta(*j*) will not exceed the size of the buffer, where *delta*(*j*) is the set of pages that contain records that will eventually reside in the new page *j*.

In the *static_cost_reorganization* algorithm, after constructing a page (*i*), the cost function for any not-yet-constructed new page (*j*) is the fraction of the pages in delta(*j*) that are not in delta(*i*). The next new page (*j*) will be the new page that has the minimum value for the cost function. This algorithm considers only the pages that have been brought into the buffer for construction of the most recent page (*i*). The order of all the new pages is determined statically (i.e., before construction of any of those new pages). For each new page *j*, these steps take place: (1) If necessary, swap some pages out from the buffer, using a lookahead of *k* new pages to determine which pages are not in the deltas of the next few new pages. (2) Bring in the pages that are needed to construct page *j* and that are not already in the buffer. (3) Rearrange records to construct page *j*, and modify the mapping table. (4) Write page *j* to disk.

In the *dynamic_cost_reorganization* algorithm, these steps execute iteratively until all the new pages have been constructed:

(1) The cost function for any not-yet-constructed new page (*j*) is the fraction of the pages in delta(*j*) that are not in the buffer. Choose the current new page (*j*) as the new page that has the minimum value for the cost function. This algorithm considers all the pages that are now in the buffer, even the pages that were brought into the buffer before the most recent construction of a new page. The order of new pages is determined dynamically; i.e., in each iteration, the current new page is determined.
(2) If necessary, swap some pages out from the buffer, choosing the pages that contain the fewest records needed by new pages.
(3) Bring in the pages that are needed to construct page *j* and that are not already in the buffer.
(4) Rearrange records to construct page *j*, and modify the mapping table.
(5) Write page *j* to disk.
(6) Rearrange the records in the buffer into sets according to the new pages in which the records will eventually reside; that is, partially construct those new pages. Order the pages in the buffer by the sizes of those sets of records. Modify the mapping table. The ordering in this step speeds the choosing in Step (2).

Omiecinski calculates and compares the performance of the algorithms, using randomly generated initial assignments of records to pages. The page size is 10 records. The buffer size is initially 10 pages. The file contains 100 pages, and 25 pages need construction. Here are some performance results:

—The dynamic_cost algorithm yields fewer page accesses than the static_cost algorithm does, for example, 246 vs. 367.
—The descriptions of the two algorithms above choose orderings that minimize the values for the cost functions. Using the orderings yields fewer page accesses than simply ordering linearly by new page number, for example, for the dynamic_cost algorithm, 246 vs. 291 accesses.
—For a buffer size of 10, increasing the lookahead from 1 to 24 (a complete lookahead) yields no significant improvement in the static_cost algorithm's performance. Omiecinski notes that the 10 records for a new page could easily reside on 10 different pages, since the initial assignment was random. Hence, the buffer might contain only a few pages beyond the current new page's delta. Hence, the lookahead has little value.
—For a buffer size of 20, increasing the lookahead in the static_cost algorithm reduces the number of page accesses from 318 to 262. Here, the buffer can contain more pages that will be needed in the future. However, the dynamic_cost algorithm is still better, yielding 201 accesses.

4.1.2.3. Scheuermann et al. To restore clustering, another algorithm [Scheuermann et al. 1989; Omiecinski 1996, p. 27] exchanges records between pages. As in the algorithm of Omiecinski [1985b], the restoration can apply to an entire file or to a subset of a file. Again, the order of construction of pages affects the amount of page swapping during reorganization, and an order that minimizes the swapping is desirable. A type of logical-to-physical mapping table maps identifiers to page numbers. An assumption is that the size of the buffer is at least two physical pages. A *physical page* is a page on disk or in the buffer. A *logical page* is a set of records that should reside on the same physical page. The description of the algorithm does not include online operation, but later work (surveyed shortly) analyzes online reorganization based on this algorithm.

The algorithm of Scheuermann et al. initially calculates each logical page's cost of construction. If a physical page already contains exactly the records for one logical page, the algorithm deletes that logical page from the list of logical pages that need reorganization. While some logical pages still need reorganization, these steps execute iteratively:

(1) Select a logical page to construct, based on the cost ranking of the logical pages that need reorganization. The calculation of cost takes into account all the physical pages that are in the buffer, and it tries to minimize the amount of eventual page input/output.
(2) If any physical pages needed to construct the logical page are not already in the buffers, bring them in from disk.
(3) Construct physical pages for the logical page (and perhaps for other logical pages), and write the pages to disk.

When it is necessary to free a page in the buffer, Scheuermann et al. perform these steps:

(1) Try to construct a physical page that contains exactly the records for one logical page.
(2) If the construction in the previous step is impossible, try to construct a physical page that contains only records that belong to no logical page.
(3) If the construction in the previous step is impossible, construct a physical page that contains records for one or more logical pages and possibly records that belong to no logical page. This construction degrades future performance of the reorganization, because this physical page will eventually be read again.
(4) Write the constructed physical page to disk.

Scheuermann et al. describe experiments to test performance, especially to compare performance to that of the dynamic_cost_reorganization algorithm of Omiecinski [1985b]. The experiments involve 25 logical pages to reorganize, a page size of 10 records, a file of 1000 records, and a buffer capacity of 7–20% of the file size (i.e., 7–20 pages). The cost (number of disk accesses) of Scheuermann et al. is always less than or equal to the cost of Omiecinski, and the savings range up to 36%. The algorithm of Scheuermann et al. can operate with fewer buffer pages than the algorithm of Omiecinski, it performs partial construction of logical pages when it must swap out physical pages, and it omits the unnecessary construction of logical pages whose records already reside together.

4.1.2.4. Omiecinski et al. The topic of Omiecinski et al. [1992; 1994] and Omiecinski [1996, pp. 27–28] is online reorganization based on the algorithm of Scheuermann et al. [1989]. The reorganizer locks only the pages that are in its buffer space. User transactions use page-level locks and hold them until commitment, achieving serializability during online reorganization. During reorganization, users can query records, and they can modify records in ways that do not change the cluster definitions. Users cannot insert or delete records, and they cannot modify records in ways that change the cluster definitions, since the authors assume (for simplicity) that the cluster definitions remain valid for some time. If deadlock arises between the reorganizer and user transactions, the system aborts a user transaction (instead of aborting the reorganizer), to avoid restarting reorganization.

To allow users to use an index, a physical-to-physical mapping table maps from old RIDs to new. Omiecinski et al. assume that the mapping table is small enough to fit into main storage as a hash table. When a user transaction finds a RID in an index, it checks the hash table (for possible translation) before using the RID.

Omiecinski et al. present a simulation model for predicting performance. Parameter values include a database of 1000 pages and a buffer of 100 pages (i.e., 10% of the database). The reorganizer uses 10–50% of the buffer; users use the rest. The probability that a user transaction accesses a cluster of records (as opposed to random records) varies from 0.2 to 0.8. The multiprogramming level varies from 10 to 50. The fraction of the database pages that the reorganizer will access varies from 0.2 to 0.4. The model predicts that increasing the reorganization buffer size (i.e., the fraction of the buffer that the reorganizer uses) speeds reorganization (thus allowing users to exploit the clustering) but reduces user throughput *during* reorganization. For example, increasing the fraction from 10% to 50% can reduce the time to reorganize the database by 56%, and it can raise the maximum decrease in user throughput (caused by competition from the reorganizer) from 26% to 52%. Decreasing the multiprogramming level or decreasing the probability that a user transaction accesses a cluster of records has a similar effect during reorganization.

4.1.2.5. Achyutuni et al. Work by Achyutuni et al. [1992] performs the type of reorganization that Scheuermann et al. and Omiecinski et al. perform (exchanging records between pages to restore clustering). Here the approach involves dividing the pages into groups so that few records (or even no records) need movement between groups; i.e., most necessary movement is within a group. Reorganizing one group at a time uses a small set of pages and exploits locality of reference. A group can also be a granule for recovery and for parallel execution on a multiprocessor. *Within* each group, reorganization can use one of the preceding algorithms, for example, that of Scheuermann et al. or that of Omiecinski et al.

Specifically, Achyutuni et al. construct a graph in which a node is an old or new page. An arc exists between an old page and a new page if one or more records of the old page should be clustered in the new page. Using the graph, the system divides the pages into one or more *components*, where each page in a component connects to the other pages in that component but does not connect to pages in other components. If a component's size exceeds a threshold number of old pages, the system tries to split the component into subcomponents:

(1) Try to find old pages that are articulation points. A node is an *articulation point* if there exist two nodes between which every path passes through the articulation point.

(2) If no old pages are articulation points, the component cannot be split.

(3) If one old page is an articulation point, split the component there; both subcomponents include the articulation point.

(4) If several old pages are articulation points, try to pick some of them to produce subcomponents whose sizes are roughly equal and are all within the threshold.

Achyutuni et al. implemented this reorganization technique on a multiprocessor with shared memory. They consider the performance of offline reorganization with division into subcomponents, division into just components (but not subcomponents), and no division. *N* (the number of pages to reorganize) is 100, and each page contains 5 records. They use 4 components of size 21 pages (each with 3 articulation points), 4 components of size 2, and 8 of size 1. *P* (the number of processors used for reorganization) varies from 1 to 4. *B* (the number of buffer pages for reorganization) varies from 10 to 100.

The authors calculate the *I/O* ratio (the amount of page input/output divided by 2**N*, which is the minimum possible amount). As *B*/*N* (the fraction of the reorganized pages that can fit into the buffer pages for reorganization) increases to 1, the I/O ratio decreases to 1, since all the pages then simultaneously fit in the buffers. For low values of *B*/*N*, division into components improves locality of reference and thus yields a lower I/O ratio than no division. For example, for $P = 1$ and $B/N = 0.2$, division into components reduces the I/O ratio from about 2.5 to 1. For $P = 1$ and *B*/*N* between 0.1 and 0.2, the I/O ratio exceeds 1 even for division into subcomponents, but the I/O ratio for division into subcomponents is less than that for division into components, and both are less than the I/O ratio for no division. For *B*/*N* less than 0.7, the I/O ratio for division rises as *P* rises (since processors compete for the shared buffer space), but it is still less than the I/O ratio for $P = 1$ and no division.

Achyutuni et al. also present a performance model for *online* reorganization. They calculate the degradation of users' response time caused by online reorganization. Users and reorganizers use separate processors. A reorganizer holds locks on a component's pages until the component has been reorganized. The database contains 100,000 pages, and *N* is 20–40% of the database. *B*/*N* varies from 0.1 to 1. For $N = 20\%$ of the database and $B/N = 0.1$, the degradation varies from 31% without division into components to 3% with division. For $B/N = 1$, it varies from 6% to less than 1%. For $N = 40\%$ of the database and $B/N = 0.1$, the degradation varies from an unbounded value (resulting from an unstable system) to 7%. Division into components reduces the response time because a reorganizer can complete its work (and thus release its locks) more quickly.

4.1.2.6. Iyer and Sockut. In the strategy of Iyer and Sockut [2002], one index determines the clustering. The reorganizer scans the clustering index. For each record to which the index refers, the reorganizer calculates the record's desired page and moves the record to the desired page if possible (or to a nearby page). This movement might affect user transactions for which the record is relevant. Therefore, the reorganizer also updates *movement lists*. These lists track records' movement, to let user transactions correct their behavior to accommodate records' possible movement. Section 3.5.5 described the nature, maintenance, and use of the movement lists.

4.1.2.7. Martin and Crown. The strategy of Martin and Crown [2003a] applies to IMS databases [Strickland et al. 1982]. The reorganizer builds a map of storage blocks, indicating which ones contain at least a threshold amount of free space. For each IMS record in a list of IMS records to be analyzed, the reorganizer ascertains whether the record is fragmented and, if so, removes the fragmentation by moving the record into free space and committing. After this reorganization of individual records, the reorganizer ascertains whether the database as a whole needs reorganization. If so, it reorganizes the disorganized records into free space.

4.1.2.8. Huras et al. In another strategy [Huras et al. 2005], the reorganizer repeatedly performs this pair of high-level phases:

(1) Vacate a page range by moving records to higher pages.
(2) Move records into the vacated page range in clustered sequence.

The strategy must assure that user transactions during reorganization access each appropriate record exactly once, not zero or multiple times. This reorganization introduces some new types of records, containing pointers that track reorganization's movement of records; reorganization can change a record from one type to another. Huras et al. describe how user transactions follow or ignore the various pointers. To move a record (within either of the high-level phases above), the reorganizer performs these low-level phases:

MOVE :	Move the data, insert a temporary pointer in the old location of the record, log the changes, insert a new pointer in indexes, and mark the old pointer as old in indexes.
CLEAN :	Delete the old data (including the temporary pointers), log the changes, and delete the old pointer in indexes.

The movement is recoverable.

Two table-level locks (for MOVE and CLEAN) synchronize the actions of users and the reorganizer. At the start of each low-level phase (MOVE or CLEAN), the reorganizer sets an exclusive lock on the relevant lock; it might need to wait for many user transactions to finish. The behavior of a user transaction varies according to the low-level phase. A user's scan that starts during MOVE sets a shared lock on CLEAN, and it locks and scans records at their old positions. A user's scan that starts during CLEAN sets a shared lock on MOVE, and it locks and scans records at their new positions (and also at their old positions if the reorganizer has not yet obtained the CLEAN lock).

4.1.3. Restoration by Copying via a Reorganizer. A reorganizer might restore clustering by copying, as in the strategies that we discuss. The strategies for clustering consider links between objects or values of a clustering key. The authors discuss performance of users and of reorganization.

4.1.3.1. McIver and King. A strategy for restoration of clustering [McIver and King 1994; McIver 1994] applies to object databases. An important concept in the strategy is users' accessing of objects consecutively by following links (in either navigational accessing or set-oriented accessing). The strategy involves three components. The first two components maintain their data outside the database and thus do not lock the database.

The first component of McIver and King executes as a background process. To track patterns of access, this component collects statistics, which user transactions send at commitment. One statistic is the ratio of (1) total consecutive accesses to objects on different pages to (2) total disk accesses. Degradation of performance (specifically, the rise of this ratio beyond a *cluster analysis trigger threshold*) activates the second component (an analysis of clustering).

The second component also executes as a background process. This component analyzes clustering by using a criterion that involves consecutive accesses of objects (not necessarily on different pages). The analysis uses a graph whose nodes represents objects and whose edges represent links. The analysis performs a depth-first traversal for parts of the database that experience mainly navigational accessing; it performs a breadth-first traversal for parts that experience mainly set-oriented accessing. It produces a sequence of object identifiers; the order reflects the relative amounts of consecutive accessing and thus suggests a desired clustering (assignment of objects to pages). Usage patterns might have changed after the previous clustering. The authors specify a formula for measuring dissimilarity between the desired clustering and the previous clustering. Sufficient diminishment of clustering (specifically, dissimilarity beyond a *reorganization trigger threshold*) activates the third component (reorganization).

The third component (reorganization) copies objects to a new set of pages to achieve the desired clustering. A logical-to-physical mapping table maps from object identifiers to addresses on pages, and it reflects the movement. This mapping table obviates the updating of objects that have links to records that have moved, and it allows interleaving of user accesses and movement of records by reorganization. Characteristics of the underlying storage manager required reorganization to lock the entire database. However, reorganization consists of steps (e.g., reclustering of 100 objects per step), and users can execute transactions between steps.

McIver and King measure performance, using one application. In the first stage of their experiment, they use a cluster analysis trigger threshold of 0 (thus always activating analysis of clustering) and an infinite reorganization trigger threshold (thus never activating reorganization), to collect statistics for the probability distribution for the ratio mentioned in the description of the first component above. In the second stage, they use a reorganization trigger threshold of 0 (thus activating reorganization after each analysis of clustering). They use four values for the cluster analysis trigger threshold: 0, the mean minus the standard deviation, the mean, and the mean plus the standard deviation. The measurements collect statistics for the probability distribution for the dissimilarity mentioned in the description of the second component above. In the third stage, for each value for the cluster analysis trigger threshold, they use three values for the reorganization trigger threshold: the mean minus the standard deviation, the mean, and the mean plus half of the standard deviation. For the third stage, measurements show that conservative thresholds (slightly higher than the initially measured mean values) usually yield the lowest ratios of buffer page faults to object accesses. Online reorganization decreases the ratios but causes some delay to users from contention for resources, for example, locks. Measurements also show that breaking reorganization transactions into smaller transactions reduces reorganization's degradation of users' performance by allowing more interleaving of users' transactions with reorganization.

4.1.3.2. Sockut et al. Another copying strategy applies to a relational storage organization in which the key values of an index (the clustering index) determine the clustering. IBM [1997, pp. 2.193–2.232], Sockut et al. [1997], and Friske and Ruddy [1998] describe restoration of clustering (with removal of overflow and distribution of free space evenly) via fuzzy reorganization. After unloading, the DBMS sorts the data by the clustering key and then reloads in the sorted order. During reloading, it also creates a backup copy of the data, as a basis for future recoverability, and it creates new copies of all the indexes. The area being reorganized can be a group of one or more tables or a range of one or more partitions of one table [IBM 2001].

In this storage organization, each data record has a RID, which is a page number and an identifier within the page. For each row, the RID of the record in the old copy of the area being reorganized can differ from the RID in the new copy, since this reorganization moves data between pages to restore clustering. Log entries use RIDs to refer to data records. Therefore, the unloading and reloading create a physical-to-physical mapping table to map between each row's old and new RIDs. We discussed this mapping table in more detail in Section 3.5.3.

The fuzzy reorganization in this strategy uses iterations of log application, as we described in Section 3.3.3. Each iteration includes copying log entries into a buffer, constructing pointers to the buffered log entries, sorting the pointers by the old RIDs, translating old RIDs to new RIDs via the mapping table, sorting the pointers by the new RIDs, actually applying the buffered log entries to the new copy of the area being reorganized, and updating the new copies of the indexes.

Several other facilities use comparable fuzzy reorganization. BMC Software [1997] also uses a physical-to-physical mapping table for the same storage organization. BMC Software [1998] describes a type of fuzzy reorganization (via a specialized file instead of the log) for a hierarchical DBMS. Pereira [2000] also uses a physical-to-physical mapping table. Marshall et al. [2006] describe details of implementation of fuzzy reorganization for a hierarchical DBMS.

4.1.3.3. IMS. For certain databases in the IMS DBMS [Strickland et al. 1982], online reorganization [IBM 2005, pp. 370–393] restores clustering and redistributes space by copying. The reorganizer scans the partition being reorganized in logical order. In each reorganization step, the reorganizer locks a group of data, copies it, commits, logs the copying, and unlocks. A parameter controls the speed of reorganization, as a fraction of the maximum possible speed; the reorganizer pauses between steps. To meet performance goals, the system dynamically adjusts the amount of data per step. User transactions access prior (already-copied) data in the new copy of the partition and later data in the old copy. Section 3.5.4 described correction of pointer values.

4.2. Reorganization of an Index

Another category of maintenance that has received considerable attention is reorganization of an index. Here, we will sketch characteristics of indexes (to explain the need for reorganization) and survey work in online reorganization of indexes.

4.2.1. Background. Many DBMSs support indexes that consist of hierarchies of nodes. *B-trees* and their variations [Bayer and McCreight 1970; 1972; Comer 1979; Teorey and Fry 1982, pp. 305–327] are common forms for such hierarchies. We will sketch the operations of insertion and deletion in an index.

On insertion into an index, if the desired node is already full and thus cannot store the new entry, the system allocates another node and *splits* entries of the original node into the two nodes. The required insertion of an entry into the parent node may cause the splitting to propagate upward, sometimes even adding a level to the index. Allocation of pages, addition of levels, and (in some implementations) loss of physical contiguity of logically contiguous leaf nodes can slow users' future accesses to the index.

In some implementations, on deletion from an index, if a node becomes less than half full and a neighboring node is more than half full, then the system moves some entries between the nodes to balance the nodes. If, instead, no neighboring node is more than half full, then the system *merges* (concatenates) the entries from the node and a neighboring node into one node, and it deallocates the newly empty node. The required deletion of an entry from the parent node may cause the merging to propagate upward. A DBMS that uses an index can split and merge nodes of an index during users' activities, for example, Sockut and Goldberg [1979, pp. 381–388]. In practice, typical commercial implementations do not balance or merge nodes; they deallocate a node only upon a user's deletion of the node's last entry.

Some implementors recommend occasional reorganization of the entire index. Reorganization can reduce the number of allocated pages, reduce the number of levels in the hierarchy, and make the distribution of free space as uniform as possible across the index nodes. These benefits are especially important for implementations that do not balance or merge nodes during deletions. In some implementations, reorganization can also restore the physical contiguity of logically contiguous leaf nodes.

We are aware of more than 40 articles that deal with allowing concurrency in indexes, that is, allowing several users to search, insert, and delete concurrently. These actions include splitting, balancing, and merging of nodes, so we consider them to include a form of online reorganization, namely incremental maintenance during users' activities. However, we consider the specialized topic of index concurrency to exceed the scope of our article and to deserve its own extensive survey article, so we will not survey it. Several articles survey much of the work on index concurrency; these surveys concentrate on comparing the performance of strategies for concurrency. See the comparisons by Johnson and Shasha [1990; 1993], Srinivasan and Carey [1991b; 1993], Srinivasan [1992], and Goyal et al. [1995]. Goodman and Shasha [1985] and Shasha and Goodman [1988] describe a framework for strategies for concurrent access for search structures.

The desire for high performance of user activities can motivate a DBMS designer to move some maintenance activities from user activities to a reorganizer. We will survey *only* the strategies that use a reorganizer for some or all of the maintenance activities. Several of the surveyed articles concentrate on issues in index concurrency (actions during users' activities), and they say very little about actions of the reorganizer, but we concentrate on actions of the reorganizer. The strategies do not all apply to the same variation of B-trees. We will discuss reorganization in place, reorganization by copying, and finally the special topic of merging of indexes.

4.2.2. Reorganization in Place. The strategies here reorganize *without* copying a large number of nodes from an old location to a new location at one time.

Guibas and Sedgewick [1978] and Kessels [1983] describe index rebalancing that does not necessarily use a reorganizer, but they note that the rebalancing can be done by a reorganizer that traverses the hierarchy.

Kung and Lehman [1980] describe concurrency in binary trees. User activities can change the structure of the parts of the index that they access. Nodes that will no longer be in the index go into a garbage queue, but they acquire back pointers, to ensure correctness of searches that are still positioned in those nodes. Periodically, a reorganizer locks the garbage queue, copies it, resets the original garbage queue to empty, unlocks it, waits for all currently executing processes to terminate (and thus to stop using the garbage nodes), and returns the garbage nodes to the list of available storage. This article also describes how to use several concurrent reorganizers, each returning one node at a time, by adding reference counts and waiting for a count to decrease to 0. A *reference count* is an integer that counts the references to a node.

The strategy of Manber and Ladner [1982; 1984] also operates on binary trees. A user activity can logically delete a node and add its identifier to a list of logically deleted nodes. One or more reorganizers can operate. Unlike the strategy of Kung and Lehman, here a reorganizer, not a user activity, adjusts a pointer to a logically deleted node. A user transaction locks at most one node, and a reorganizer locks at most three. A reorganizer's handling of a logically deleted node (which we label B) depends on whether B has two children:

(1) If B has at most one child, then the reorganizer changes the pointer to B (in its parent, which we label A) to point to the child of B (or to null if B has no child), marks B as garbage, adds B to the garbage list, and inserts in B a pointer to A. After a reorganizer has copied the garbage list and has waited until all processes that are now active have terminated, it returns the garbage nodes to the list of available storage.

(2) If B has two children, then the reorganizer finds the node (which we label E) with the maximum key that is less than the key of B, makes a copy of E (which we label E'), sets the two child pointers in E' to point to the two children of B, sets the right

pointer of E to point to E', and changes the pointer to B (in its parent A) to point to E'. The reorganizer locks B, E, and A. A reorganizer eventually deletes B and E similarly to the deletion in case 1 above.

Manber [1984] uses a variation of the strategy of Manber and Ladner [1982; 1984]. Data always resides in leaves. A reorganizer deletes non-leaf nodes that have only one child, based on a deletion list; the reorganizer uses exclusive locking.

For an index in which user activities can delete a key but never delete a node, Salzberg [1985] adds a reorganizer to delete nodes. Operation of the reorganizer requires users to lock nodes as they move down the hierarchy. The reorganizer starts by quiescing user transactions that insert. It does not quiesce transactions that query or delete, but any such transactions that start after the reorganizer started use a new protocol (for locking). The reorganizer waits for completion of both (1) the quiescing of insertions and (2) the queries and deletions that started before the reorganizer started. Then it allows new insertions to start (with a new protocol), and it performs the actual reorganization. The reorganizer scans the leaf nodes' parents, locking one parent at a time. It rebalances (and, if necessary, merges) pairs of leaf nodes (while locking a pair) and adjusts the pointers and keys in a pair's parent. If a parent becomes too sparsely populated, the reorganizer sets a flag to indicate this condition. After finishing the leaves, the reorganizer moves up the hierarchy (if necessary), one level at a time, rebalancing and merging. If the reorganizer is terminated, new queries and deletions can execute immediately, but new insertions must wait for quiescing of existing insertions.

The strategy of Sagiv [1985; 1986] has similarities to the strategy of Salzberg [1985]. A reorganizer, locking three nodes at a time (a node and two adjacent children of that node), handles underflow and releases space. In each step, the reorganizer rearranges the pair of children if necessary (balances the load, perhaps deletes one child, and changes the parent) before moving to the next pair. It scans the leaf nodes' parents, then scans the next higher level, etc. A possible variation is for a user's deletion to enqueue an identifier of an underflowed node, thus obviating a scan; there can be any number of reorganizers. A deleted node returns to the list of available storage after completion of all processes that were active when the node was logically deleted.

Goodman and Shasha [1985, pp. 15–17] and Shasha and Goodman [1988, pp. 73–77] discuss an index that has a desired range for the number of keys per node. Violations of the range can occur; the system maintains a list of nodes that are candidates for splitting or merging. One or more reorganizers split nodes with too many keys and merge nodes with too few keys. A reorganizer locks such a node and its parent. For merging, the emptied node returns to the list of available storage after no processes can still be referencing it. Of course, the splitting or merging of nodes can cause the nodes' parents to become candidates for splitting or merging.

Omiecinski [1985a; 1996, p. 30] describes a strategy for online removal of underflow and overflow in an indexed file organization. An index node contains pairs of a key value and a pointer to a subordinate node (another index node or a leaf node). A leaf node or an overflow node contains pairs of a key value and data. Each leaf node and each pair in an overflow node can point to a pair in an overflow node. An index node or leaf node also contains forward and backward pointers. Here are the main characteristics of the strategy:

—The reorganizer performs a postorder traversal of the hierarchy; a prerequisite for reorganization of any index node is reorganization of all its subordinates. The step of reorganizing a node iterates until the root is reorganized.

—When reorganizing a node that has overflowed, the system allocates another node.

—When reorganizing a node that has underflowed, the system fills the node by obtaining values from successor nodes. The system adds a successor node to a garbage queue if all the values in the successor node have been used to fill its predecessor nodes. A node in the garbage queue contains appropriate forward and backward pointers that point to nodes containing data. Therefore, a user activity that accesses a garbage node from its parent index node during reorganization will eventually locate the desired data.

—In an index node, the system replaces the pairs of a key value and a pointer according to the reorganization of the node's subordinate nodes. The system saved the subordinates' addresses, high keys, and low keys while reorganizing them.

—A process deallocates space for each garbage node that is no longer in use. A node is no longer in use if its time of insertion into the garbage queue preceded the logged start times of all currently active user processes.

—At any time, the reorganizer holds at most one write lock. It holds read locks on the node being reorganized and (for an underflow) on that node's successor nodes. Therefore, the capacity of a node bounds the number of read locks that the reorganizer holds.

In the strategy of Nurmi et al. [1987], when overflow occurs, the system marks the overflowed node as needing reorganization, moves the node's contents into *two* new nodes, and makes the new nodes the children of the overflowed node. This behavior avoids immediate upward propagation of splitting. Also, deletion does not balance or merge immediately. Any number of reorganizers perform the needed splitting, balancing, and merging. The authors discuss three methods for finding nodes that need reorganization: (1) when users' actions necessitate eventual reorganization of a node, add the node to a list, (2) have the reorganizers traverse the hierarchy to find such nodes, or (3) invoke a reorganizer as soon as users' actions necessitate reorganization. Pollari-Malmi et al. [1996] and Kuo et al. [1999] also use such index structures and reorganization.

For key-sequenced files and tables created by one DBMS, maintenance can reduce the number of index levels, restore clustering, and balance the distribution of free space. A facility for maintenance operates online [Smith 1990; Pong 1990]. The reorganizer processes a file in key order; it swaps, merges, and splits blocks of data (and blocks of index records). It includes logging. The index operations involve locking the entire index. This locking waits for completion of existing user operations on the index, and it blocks new user operations, but it does *not* wait for (or block) entire transactions, because a user transaction can perform short operations on the index without holding locks between operations. The reorganizer pauses between its steps, to reduce the degradation of users' performance. The DBA sets a *rate* parameter (with a value up to 100%) to control the speed of reorganization. The length of a pause between steps is ((100 - rate) / rate) times the elapsed time of the previous step. Use of this formula automatically reduces the degradation of users' performance during periods of peak usage. The DBA can also suspend reorganization completely and later resume.

In the strategy of Lomet and Salzberg [1992; 1997], user activities split leaves (which contain all the data), but a reorganizer splits non-leaf nodes. A node being split receives a pointer to its new sibling. Merging (at any level) never needs to be part of a user activity. On splitting or merging, an action is requested (and eventually executed) to perform the required updating of pointers at the next higher level. Of course, this updating of pointers might lead to the requesting of another split or merge. After execution of an action, it is efficient but not essential for any related actions to execute soon. The discussions include locking and recovery:

—Splitting a non-leaf node involves locking just the node to be split. After the splitting, the updating of the pointers in the parent involves locking the parent and the child being split. Merging locks three nodes (two children and the parent). The ordering of locking prevents deadlock. The article discusses details of locking.

—Each reorganization action is atomic, is logged, and changes only one level in the hierarchy. Recovery from a system crash does not need any special measures to reflect the possible incompleteness of reorganization. Instead, if a user transaction, during later ordinary traversal of the hierarchy, finds that a split or merge has taken place but the parent's pointers have not yet been updated (perhaps due to a system crash), the missing action is requested.

For a table whose data can reside in leaves of an index, Srinivasan et al. [2000, p. 290] describe how online reorganization of the index in place merges leaf blocks within a branch, locking a few blocks at a time. An online rebuilding by copying the index is also available.

The various authors emphasize different activities, and the data structures are not identical. However, in most of the strategies, a reorganizer merges underflowed nodes. Maintenance of pointers is an important activity. Some of the descriptions mention a queue of garbage nodes for eventual physical deletion. The authors discuss locking.

4.2.3. Reorganization by Copying. The strategies in this section copy a group of nodes from an old location to a new location. This copying includes leaving the desired amount of free space in each page for future insertions. We also discuss some design decisions.

Zou and Salzberg [1996b; 1999c, pp. 34–37] describe a reorganizer for an index whose leaf nodes contain the data records. A user transaction does not merge sparsely populated nodes. The authors assume that leaf pages and non-leaf pages reside on different disks or on different parts of a disk. Logging of the reorganizer's actions enables forward recoverability. Some actions reorganize in place; some reorganize by copying. The reorganizer makes 3 passes, for (1) compaction of leaves, (2) reordering of leaves, and (3) reorganization of non-leaf nodes:

(1) The reorganizer scans the parents of the leaf nodes, in key order. Within each set of sibling leaf nodes, it compacts the leaf nodes (by moving data from one node to another) and updates the parent. It constructs one new leaf page at a time. If an empty page of storage is available after the highest-keyed page whose compaction is finished and before the leaf page whose data is undergoing reorganization, then the system uses that empty page as the destination of the movement. Otherwise, it uses a partially filled page as the destination. The article describes details of locking. If the reorganizer participates in a deadlock, it gives up its lock.

(2) The reorganizer again scans the parents of the leaf nodes. It moves the nodes so that their physical order matches the logical order of their keys. This pass takes place only if query performance has degraded beyond a threshold.

(3) To reduce the number of levels in the index, the system reorganizes the non-leaves by scanning the parents of leaves and copying to a new area. After building every few pages of the index, it forces the new pages to disk, for durability. If an update by a user is on a page that the reorganizer has not yet read, the update takes place only in the old page; the reorganizer will find it later. If an update is on a page that the reorganizer has already read, the update takes place in the old page, but the system also records the update in a special file. After copying the nodes (and building the index), the system scans the special file and applies the updates to

the new copy. It then write-locks the special file (preventing new updates), switches users from the old index to the new by changing the stored information about the location of the root, write-locks the old index (thus waiting for old queries to finish), and deallocates the old index.

IBM [1997, pp. 2.193–2.232], Sockut et al. [1997], and Friske and Ruddy [1998] concentrate on reorganization of data together with all its indexes, but they also describe reorganization of just an index [Sockut et al. 1997, p. 431]. Typically, not all the data resides in an index. The fuzzy reorganization unloads the old copy of the index, constructs a new copy of the index, applies the log entries that deal with the index, and switches users' future access to the new copy of the index by exchanging the names of the files that underlie the old and new copies. BMC Software [1997] describes similar reorganization.

Ponnekanti and Kodavalla [2000] and Raghuram et al. [2000, p. 118] divide the reorganization of an index into a sequence of transactions. Each transaction reorganizes a group of leaf pages. A transaction contains several actions, each of which consists of these phases:

(1) Write-lock a sequence of leaf pages, copy them to new pages in key sequence, and log all the copying of keys.
(2) Change the entries in the parents of the copied leaf pages; the propagation may continue to higher levels.

The authors describe details of upward propagation and details of locking. At the end of each transaction, the DBMS flushes the new pages to disk and deallocates the old pages.

In designing a strategy that copies, one decision involves the timing of write-locking (at the start of copying or near the end). An advantage of write-locking near the end of copying is availability of the area being copied during the copying. Advantages of write-locking at the start are a simpler implementation and allowance of archiving of the log before the completion of reorganization.

For a strategy that copies leaves, another decision involves the granularity (copying the entire index at one time or incrementally copying subsets of the leaves). Here are advantages of copying the entire index at one time:

—There is less overhead of concurrency control; we lock just once or twice for the index, not n or $2n$ times if there are n subsets.
—The time that we block new user transactions while waiting to lock a subset may exceed $1/n$ times the blocking time while waiting to lock the entire index. Therefore, for incrementally copying subsets, the total blocking time (the number of subsets multiplied by the blocking time for a subset) may exceed the blocking time for copying the entire index at one time.
—Copying the entire index lets the reorganization optimally arrange the non-leaf nodes, not just the leaf nodes.

Here are advantages of incrementally copying subsets of the leaves:

—We need less extra storage for a new copy (a subset instead of the entire index).
—We block less of the index at one time.
—Each period of blockage is shorter.
—Users may gain the performance benefits of reorganization gradually.

4.2.4. Merging of Indexes. Several contexts, e.g., data movement for rebalancing of hash-addressed parallel databases, can motivate the merging of two indexes that have the same key range. Sun [2005, pp. 89–111] and Sun et al. [2005] describe such merging. Because the indexes have the same key range, the merging cannot just attach them to each other.

One index is the main index; the second index will be merged into the main index. User transactions use the main index. The system includes a type of lazy reorganization. When a user transaction accesses a leaf page, if the system has not yet merged the corresponding content of the second index into the leaf page, a system transaction performs the merging and commits, and then the user transaction resumes (and commits). If the merging causes a split, then the user transaction traverses the index again.

The merging procedure locks only the smallest needed number of index pages. Logging enables forward recovery. The system stores the log's current position in an index page when it merges that page. Also, at the beginning of merging, the system records the log's current position. Therefore, the system can detect whether a page has been merged by comparing its stored log position with the beginning log position. If a user transaction is the first to access a page in the main index that has no matched entries in the second index, and the user transaction does not modify the page, the system updates the page's stored log position anyhow (writing an artificial log entry), to prevent future false detections that the page needs merging. The authors discuss details of locking.

Some leaf pages might be unreferenced (and therefore not merged by just the lazy reorganization described above) for an arbitrarily long time. Therefore, a reorganizer also operates when the system is not busy serving users. The reorganizer scans the second index one key at a time, and it searches the main index, triggering the lazy merging described above.

A performance simulation uses one disk, with a disk page size of 4KB. Each index leaf page can hold 100 entries. The main index has 400K distinct keys, and each page is 80% full. The second index has 40K or 120K keys, and each page is 100% full. The buffer size suffices to hold almost all upper-level pages and about 10% of all leaf pages. Half of the user transactions modify a leaf page.

In the results of the simulation, the authors' approach has less total I/O than having all user transactions access both indexes during reorganization by a reorganizer. The response time decreases as the elapsed time progresses, because fewer user transactions trigger merging.

Another result compares the throughput during reorganization with the throughput after completion of reorganization. With 20 concurrent threads, the throughput during reorganization rises from about 1/6 or 1/3 of the post-reorganization throughput to 100% of the post-reorganization throughput.

4.3. Rebalancing of Parallel or Distributed Data

Earlier, we described online restoration of clustering, which can improve performance. Now we will describe how online rebalancing of parallel or distributed data can also improve performance. Rebalancing may apply to *declustering* or *striping*, for which we use these definitions:[2]

—*Declustering* (e.g., Fang et al. [1986], Copeland et al. [1988], and Ghandeharizadeh and DeWitt [1990]), is the practice of interleaving data among parallel disks, assigning *logical* units of data (e.g., a row of a table) per disk. A DBMS is involved, and it might use a round-robin approach or a key. If it uses a key, it might hash the key, or each disk might get a range of key values.

[2]Some authors may use different definitions.

—*Striping* (e.g., Salem and Garcia-Molina [1986], and Kim [1986]), is the practice of interleaving data among parallel disks, assigning *physical* units of data (e.g., *n* bytes) per disk. There is not necessarily a DBMS involved. A round-robin approach is used.

Declustering or striping can permit parallel access to the pieces of data. Careful placement of data can balance the activity levels among the disks and thus improve performance by reducing the probability that queues form at some disks while other disks are idle.

Over time, users' updates (or growth in the network) can change the balance of activity by creating a skew (imbalance) of data among the disks. After such changes, rebalancing can restore the balance of data. Work that we survey below rebalances data online. In addition, Wilkes et al. [1996] mention rebalancing as a background activity. Also, Zilio [1998] analyzes activation of maintenance, as we described in Section 3.7.3; this work uses the context of rebalancing of parallel data. Some of the concerns that arise in the surveyed work are user response times, balance among the nodes, and the cost of reorganization. To limit our scope, we omit work on splitting and merging of hash buckets in distributed hashing, except for work that also includes other movement of data between nodes. Based on the topics that the authors emphasize, we have divided the work into (1) choice of data to migrate (typically without growth in the network), (2) growth in the network, and (3) additional topics. There is some overlap among the topics of the work in the three sections.

4.3.1. Choice of Data to Migrate. This first topic involves criteria for choosing the data to migrate to accomplish the rebalancing. The criteria can include the access frequency of the data and the size of the data, with a goal of avoiding bottlenecks. This work typically does not emphasize growth in the network. Some of the work points out that the importance of rebalancing increases as the workload increases. Much of the work in this section builds upon other work in this section.

4.3.1.1. Weikum et al. If files can be created at any time (not necessarily all at once), file creation can require reorganization. In a striped disk array, decisions for a file include the number of disks across which to stripe and the selection of those disks in a way that balances the load across all disks. Weikum et al. [1991] discuss these issues, with a goal of minimizing the average response time.

The authors describe characteristics of the file system. A *run* is a physically contiguous set of logically consecutive blocks of a file; it is the granularity of striping. An *extent* is a physically contiguous set of one or more runs on a disk. A *region* is a set of extents on different disks. Growth of a file leads to allocation of additional regions. A file is a set of regions. A logical-to-physical mapping table maps from logical block numbers to disk block addresses. The *heat* of an object (extent or disk) is the access frequency (involving I/O, not caching) over a period of time. The *temperature* of an object is the heat divided by the size.

The performance goals are to minimize the access time to consecutive portions of a file, to balance the disk load, and to minimize the cost (extra disk load) of reorganizations. These goals can conflict with each other. When a file is created, the system uses certain formulas for placement. If a disk has enough space for an allocation request but lacks enough *contiguous* space, the system compacts the disk.

If a disk becomes so hot that it causes significant imbalance in the load, the system performs *disk cooling*, which is migration of data from the source (hot disk) to a destination (cooler disk). The granularity of migration is an extent. The destination is usually a disk that does not hold an extent of the same region. A heuristic is to migrate the extent that has the highest temperature. The temperature represents the ratio of benefit (decrease in the disk's heat by removing frequently accessed data) to cost (amount of data to migrate). The system continues cooling a disk until the disk's heat drops below a threshold or until the destination disk would become hotter than the source.

Weikum et al. describe candidate algorithms for picking the disks across which a new file is placed (given that the number of extents w has already been decided):

—*Heat balancing* picks the w coolest disks, regardless of availability of free space on those disks. This algorithm emphasizes the goal of load balancing.

—*Space-restricted heat balancing* picks the w coolest of the disks that have enough space; it then picks additional disks (which lack enough space) if fewer than w disks have enough space. This algorithm and the next should require less disk cooling than the prior one, since they are less likely to pick disks that lack enough space. These algorithms emphasize the goal of low cost of reorganization.

—*Cost minimization* picks the w coolest of the disks that have enough *contiguous* free space; if necessary, it then picks disks that have enough space but not enough contiguous space; if necessary, it finally picks disks that do not have enough space. This algorithm should require less compaction than the prior one, since it is less likely to pick disks that have enough space but not enough contiguous space.

With all the algorithms, if the heat of a selected disk exceeds the average heat of all the disks, and at least one unselected disk is cooler, then disk cooling takes place before allocation of the new extent.

The authors perform simulation experiments. The experiments use an array of 12 disks, an average file size of 400 kilobytes, and a mixture of file creations, deletions, reads, and writes. 90% of the reads and writes access 10% of the files. The simulation compares six algorithms: (1) Allocate extents to balance the space utilization; ignore load balancing. (2) Allocate extents as above, but also, at each creation, if the heat of a disk exceeds 1.1 of the average heat of all the disks, perform disk cooling. (3) Use the heat balancing algorithm described earlier. (4) Use heat balancing, with the possibility of cooling as described above. (5) Use the cost minimization algorithm described earlier. (6) Use cost minimization, with the possibility of cooling as described above.

Weikum et al. discuss the results of the simulation:

—The load balancing of algorithms 4 and 6 suffices to avoid the heat imbalance threshold that activates cooling; these algorithms behave like algorithms 3 and 5.

—Algorithm 2 outperforms 1; here cooling is beneficial. For example, at an arrival rate of 30 requests per second, the average response time of algorithm 1 is 1.66 times the response time of 2.

—At low arrival rates (e.g., less than 28 requests per second), algorithms 1 and 2 perform well, since the load imbalance does not cause queuing. Algorithms 3 through 6 spend some time performing compactions, which do not improve performance if there is no queuing problem.

—At high arrival rates, load imbalance produces bottlenecks, and algorithms 3 through 6 outperform 1 and 2.

4.3.1.2. Scheuermann et al. [1993]. The work by Scheuermann et al. [1993] generalizes the work of Weikum et al. [1991], by including the cost of migration in the activation decision and by considering periodic cycles of access characteristics. The authors note that partitioning determines the degree of parallelism in a single request, the assignment of partitions to disks determines the load balance, and the degree of intra-request parallelism and balancing determine the throughput. Heat can vary over time, as access characteristics change. The system tracks the heat of extents and of disks and the temperature of extents. The authors use temperature as the criterion to select extents to migrate in disk cooling.

The authors discuss the objective function for optimization. If some (not necessarily all) files have predictable cycles of heat, then the optimization time span can be the maximum of their cycle times. Within the time span, changes in heat define the boundaries of time intervals. To distribute the load as evenly as possible, the authors want to minimize *WDHV* (the *weighted disk heat variance* among the disks). WDHV is the sum (over all intervals *j* and over all disks *i*) of ((heat of disk *i* during interval *j*) - (average heat of all disks during interval *j*))^2 * length of interval *j*.

Here is a greedy heuristic algorithm for each cooling action:

(1) Pick the disk with the highest heat.
(2) Pick that disk's extent with the highest temperature.
(3) Pick (as the destination) the lowest-heat disk that does not already hold an extent of the same file.
(4) Estimate when the reading and writing parts of the migration would occur.
(5) Calculate the extra heat variance due to the migration itself.
(6) If (current *WDHV* - predicted new *WDHV* - variance due to migration) exceeds a threshold, then schedule the migration.

This algorithm continues iteratively. To estimate the heat of a block, the authors track the most recent requests to a block and calculate the average time between arrivals. There are also formulas to estimate cooling's contributions to the heat variance. The formulas assume that user requests have higher priority than cooling requests.

Scheuermann et al. simulate two workloads:

—A file system includes files with a total size of 3.5 gigabytes. There are 16 disks, each holding 1 gigabyte. The system initiates cooling when the heat of the hottest disk exceeds the average heat by more than 1%. The *speedup factor*, which determines the arrival rate, is the duration of the *simulation* of all arrivals divided by the duration of the *actual* trace (24 hours). In the results of the simulation, the response time (with or without cooling) rises more than linearly as the speedup factor increases. For speedups up to about 15, cooling has no significant effect. For higher speedups, cooling cuts the response time. For example, for a speedup of 20, cooling decreases the response time by a factor of 2.3. During periods of heavy usage, the system schedules few cooling steps, to reduce degradation caused by cooling.

—A transaction processing workload uses files whose total size of 23 gigabytes. The workload has skewed access frequencies. There are 24 disks. During most periods, cooling has little effect. During peak usage, cooling improves the response time by a factor of 1.7. Again, during periods of heavy usage, the system schedules few cooling steps.

In summary, cooling is most important when the disk array is stressed.

4.3.1.3. Scheuermann et al. [1998]. The strategy of Scheuermann et al. [1998] extends the work of Weikum et al. [1991] and Scheuermann et al. [1993]. Partitioning can be via striping or declustering; the presentation uses striping. This work deals with tuning of the *striping unit* (the number of consecutive bytes stored on a disk), tuning of the allocation on disks, and rebalancing. Two quantities that must be chosen are the striping unit and the *striping width* (the number of disks over which a file is spread). However, we will not discuss the choice of those quantities; our discussion will concentrate on reorganization for balancing.

Disk cooling (described earlier) involves finding the best extent to migrate from the hottest disk, with goals of minimizing data movement and maximizing performance improvement. A reorganizer executes at fixed intervals to perform the cooling. Here is this strategy's version of the basic procedure for cooling:

(1) If the heat of the hottest disk exceeds the average disk heat by a threshold, continue. Otherwise, do not continue.
(2) Find the highest-temperature extent on that source disk such that there is a destination disk that has space and does not already hold an extent of that file. The search for such a destination disk starts from the lowest-heat disk. Select that extent and that destination disk.
(3) If the source disk has no queue (and thus migration actions are unlikely to increase the load imbalance), and the heat of the destination disk after migration would not exceed the heat of the source disk after migration, then migrate the extent and adjust the heats of both disks.

Only a fraction of the executions of the reorganizer result in migration. Refinements of this procedure are possible [Scheuermann et al. 1993]; an example is to consider the heat caused by the reorganization itself.

The system must estimate the request sizes and the request frequencies for various partitions, track the heat and temperature of extents, and track the heat of disks. The tracking of the heat of an area of storage (e.g., an extent) uses a moving average of inter-arrival times of requests that access the area. The system tracks the most recent k requests, where k is in the range 5–50. This policy tracks hot blocks well, but to better track areas whose heat drops, the system also periodically invokes a process that simulates pseudo-requests to all areas. When such a request would lead to a reduction in heat, the system adjusts the heat; otherwise it ignores the pseudo-requests.

Simulation experiments use an array of 32 disks. One synthetic workload involves 10,000 files. There are three classes of files: 1000 files have a mean size of 20 kilobytes, 1000 files have a mean size of 500 kilobytes, and there are files that have a mean size of 1 megabyte but are not accessed in the experiments. We will discuss experiments that use skewed access frequencies; 70% of the accesses use 30% of the files. To study the effects of load balancing, no caching takes place. The experiments use three striping units, each with and without cooling. With cooling, the system attempts a cooling step after every 100 user requests. The unit of migration is 1 extent. The threshold for exceeding the average disk heat is 5%.

In the results, for low rates of arrivals (up to 100 requests per second), the average response time with cooling resembles the response time without cooling. For higher rates, cooling reduces the response time. For example, for one striping unit and an arrival rate of 125 requests per second, cooling cuts the response time in half. For all three striping units, as the arrival rate grows, the response time without cooling increases without bound before the response time with cooling increases without bound. Cooling adds overhead of migration, but it reduces the bottleneck of hot disks, which is important at high loads. In experiments in which the hot fraction shifts from one set of files to another (e.g., due to changes in access patterns), cooling decreases the response time by more than a factor of 5 in some cases. For simulations based on traces (of a server for the world wide web and of a bank's online transaction processing), cooling also improves response time.

4.3.1.4. Scheuermann et al. [1994]. The discussion by Scheuermann et al. [1994] summarizes their work that appears elsewhere [Weikum et al. 1991]; [Scheuermann et al. 1993]; an early version of Scheuermann et al. [1998]. The procedure for disk cooling and the tracking of heat operate as Scheuermann et al. [1998] describe. The authors describe a simulation.

A synthetic workload uses 1000 files. Each file resides on 2 disks, and the striping unit is 1 track. The granularity of each read, write, or migration is an entire file. There is a skew in the choice of file to access (80–20 or 70–30). The heats of the files can shift from one time period to the next. The simulation shows that the average response time with cooling usually varies little over time and is often much less than the average response time without cooling, which can vary greatly.

A trace-based workload applies to applications that mainly deal with design and simulation. It has larger files than the first workload. The trace involves 14,800 files. The striping unit is 1 cylinder (385 blocks). The average response time with cooling does vary over time, and it is lower than the response time without cooling and has less variation. Over the entire simulation period, the average response times are 0.3 seconds with cooling and 1.1 seconds without cooling.

4.3.1.5. Zabback et al. The analysis by Zabback et al. [1998] is oriented toward cases where many of the transactions have a periodic pattern of access; the heat varies from one interval to the next interval. This work is a follow-on to the disk cooling of Scheuermann et al. [1993; 1994]. A queuing model determines when to schedule the read and write phases of migration, given that a load imbalance has been observed. The authors note that it might be good to postpone migration until a low-heat interval, when the heat caused by users plus the heat caused by reorganization is low enough. Reorganization takes place only if its benefit exceeds its cost. The system schedules reading (which executes only if the source disk has no queue) for a low-heat interval; it schedules writing as soon as possible after reading.

4.3.1.6. Helal et al. The approach of Helal et al. [1997] starts from the approach of Scheuermann et al. [1994]. Helal et al. emphasize transactional aspects (e.g., locking), and they add replication of fragments to reduce the heat of any one copy. The system declusters rows by round-robin assignment to nodes. The authors describe three replication methods:

—In full replication, each data item exists at each node.
—In half replication, each data item exists at half of the nodes; the assignment is random.
—In chained declustering, each data item exists at two nodes.

In all the methods, a user transaction that reads can read any copy, and a user transaction that writes must write all the copies.

Starting from the migration of Scheuermann et al. [1994], Helal et al. use aborts and restarts to guarantee data consistency. When a user accesses a node, the system updates the node's heat, and, if the heat exceeds a node heat threshold, the node sends a message to an administration node. The system updates data fragments' heat similarly. Periodically, the administration node invokes migration for nodes whose heat exceeds the node heat threshold. For each such hot node, the system migrates the fragments whose heat exceeds the fragment heat threshold (or migrates the warmest fragments if no fragments exceed the fragment heat threshold). Migration consists of locking the catalog, selecting the coolest disk that does not already have a copy, aborting all transactions that access the fragment, moving the data, updating the catalog, and unlocking the catalog. Only a block being moved is offline.

In a simulation model, the system decomposes a transaction into subtransactions (at most one for each node) based on the catalog, and it merges the subtransactions' results before returning the transaction's result to the user. The simulation includes locking. In the simulation, 20% of the data is hot, 60% of operations access hot data, 75% of operations per transactions are reads, the multiprogramming level ranges from 2 to 1000, and there 1, 2, 4, or 8 nodes. The simulation compares performance with and without rebalancing. The simulation applies to the three replication methods, and it applies to two mechanisms for handling the unavailability of a lock: (1) wait and (2) abort and restart. In the results, migration typically improves throughput (for all the replication methods), especially for a multiprogramming level over 400. The effects of the two mechanisms for concurrency control vary.

4.3.1.7. Lee et al. In an index-based strategy for rebalancing [Lee et al. 2000; Mondal et al. 2001; Mondal 2002], the DBMS bulk-loads the migrated data into a tree (specifically, a subset of the primary index) at the destination. Secondary indexes require conventional insertions and deletions. The DBMS maintains height-balance of indexes in the nodes, even if some indexes are fatter or leaner than usual. Data is partitioned by the range of primary key values. The top tier of a 2-tier index directs an access to the node that contains the data. The DBMS replicates this tier across all nodes, to avoid a bottleneck of one master addressing index, and the authors expect this tier to be cached in main storage. The bottom tier is a collection of subtrees, one at each node.

A migration step consists of moving the data by bulk loading and changing of indexes. Each migration moves data to the preceding or following range. The amount of data to move is a subtree of the primary index; this policy enables efficient updating of the index. The subtree's level in the index controls the amount of data to move; the strategy is flexible with respect to the granularity of migration. A migration requires changing the top-tier index; this information will travel to other nodes as part of any future messages that would be sent even without this migration. If a third node misdirects a search to the source of the migration, the source node redirects the search to the target.

The DBMS activates migration when the load, response time, or queue size at a node exceeds a threshold. A control node polls the nodes for their statuses and decides whether to activate migration. An alternative is for each node to compare its load with its neighbors' loads.

Performance measurements involve both simulation and an implementation. The number of nodes ranges from 8 to 64, and the number of records ranges from 500,000 to 5,000,000. The experiments include creating the initial database and issuing 10,000 queries. 40% of the queries go to a hot node; this skew activates migration.

Here are some performance results:

—The average amount of index page I/O per migration for the authors' strategy (updating an entire subtree of the index at one time) is, of course, much smaller than the amount of index page I/O per migration for inserting keys one at a time (fewer than 20 page accesses vs. more than 700).

—Another measurement is the maximum (over all the nodes) of the number of queries to a node. A migration occurs if the load of at least one node differs from the average load of all the nodes by more than 15%. The maximum number of queries to a node decreases from about 430 before any migrations to about 160 after 100 migrations, if the strategy has the flexibility to pick any level in the bottom-tier index as the subtree to migrate. In a restricted form of the strategy, where the subtree to migrate must be a child or grandchild of the root of the bottom-tier index, the maximum decreases by less (to about 220 or 320, respectively).

—For a measurement of response time, migration occurs if at least one node has at least 5 queries waiting to execute. The source of migration is the node with the most waiting queries. As the number of queries progresses to 10,000, the average response time rises and then drops if migrations occur, as migrations reduce the skew, but the time rises by more and continues to rise if no migrations occur. The response time for the hottest node exhibits the same pattern of behavior.

4.3.1.8. Feelifl et al. Another approach [Feelifl et al. 2000] involves data that is partitioned across nodes according to ranges of values of a key. This approach starts from the work of Lee et al. [2000] (including a 2-tier index, as described above), but it also considers the speed of rebalancing. The granularity of migration is one or more branches of an index. Updates of the top-tier index are propagated in any future messages that would be sent even without migration.

The metric for the workload is *heat*, which is the access frequency for a range of values during a period. The system maintains heat statistics for each subtree of the root of each node's lower-tier index, but the approach could use other granularities of statistics. The authors assume a uniform heat distribution at lower levels. The authors note that with range-based partitioning, a disk cooling approach that takes a local view (calculating one movement at a time) can be unstable. For example, suppose that nodes 0 through 3 have heats of 200, 50, 115, and 35, respectively. A migration of 90 from node 0 to node 1 might yield heats of 110 and 140, respectively. A migration of 30 back to node 0 might yield heats of 140 and 110. Such migrations might continue indefinitely. The 115 is never the highest heat, so the system would never migrate to node 3, which would be the best choice. Instead, to decide on the eventual migration, the authors' approach calculates from the ends inward, based on a desired heat per node. This global view of rebalancing can calculate a more balanced result.

There is a trade-off involving the amount of data movement. Quickly doing all of the movement necessary to achieve perfect balance minimizes the performance degradation due to skew, but it maximizes the performance degradation due to reorganization's competition with users. Therefore, two parameters control the reorganization. For the *speed* parameter (which we call s), a value of 0 means to avoid migration, 1 means to perform each step of migration without division into smaller steps, and a value between 0 and 1 means to divide each step into $1/s$ smaller steps. The *distribution* parameter (which we call d) defines the rebalancing goal, namely that the heat at each node should not exceed the average heat by more than $d\%$. The reorganization then moves only from the hot nodes, not necessarily from all nodes.

In a simulation, there are 2.1 million records, occupying 2 megabytes. The hottest node can vary from node 0 to node 15. The speed s ranges from 0 to 1, and the distribution d ranges from 0 to 30%. The number of nodes N is 16, 32, or 64. Initially the data is uniformly distributed, but access is skewed. The rebalancing occurs after every N*100 queries.

The authors plot these results:

—The average response time for the hottest node (plotted against the elapsed time) rises sharply (for any s) and then falls gradually (for any $s > 0$). As s increases, the response time almost always decreases.

—As d increases, the *migration workload* at a node (the time that the node spends in migration) falls, because less migration is necessary. If node 0 is hottest, migrations ripple through the nodes.

—The system migration workload is lowest when the hottest spot is in the middle of the string of nodes; migration is possible in either direction.

—The migration workload increases as the skew increases, since more migration is necessary.

4.3.1.9. Feelifl and Kitsuregawa. For a B-tree indexed database over shared-nothing parallel machines, Feelifl and Kitsuregawa [2001] have a goal of rebalancing with minimal cost of moving data and modifying indexes. Unlike some previous work, migration can wrap between the leftmost and rightmost processing elements. A linear configuration (i.e., without such a ring) has a high cost if a hot spot occurs at an edge. The data is initially range-partitioned across processing elements (*PE*s). The first-tier index, which is replicated in all PEs, directs a search to one PE; the second tier directs the search to pages. To achieve a ring configuration, a PE can hold two trees, not just one.

The algorithm has two parameters ξ and α. Each PE should have no more than $(1+\xi)$ * the average heat. There should be no more than $1/\alpha$ steps required to balance, where α is in the range [0,1]. The balancing algorithm starts from a hot spot. It finds the minimal movement to achieve rebalancing.

A simulation uses 16 or 32 PEs, 1,000,000 records, and a data page size of 4KB. The heat balancing takes place every 100*N queries, where N is the number of PEs. In the results, a linear configuration has a higher migration cost than a ring configuration (3.5 times the ring's cost when $\xi = 0$).

4.3.1.10. Ganesan et al. In a context of range partitioning that distributes a table across parallel disks, Ganesan et al. [2004] have goals of avoiding data skew and minimizing the movement of rows. The *imbalance ratio* σ is the load (number of rows) of the largest partition divided by the load of the smallest. The strategy is designed to keep σ below a tunable constant c, which can be as low as 4.24. Each node manages a range of rows. A central site records the partition boundaries, and it routes each query, insertion, or deletion to the appropriate nodes.

After each insertion or deletion on a node i, the balancing strategy executes on i. The cost of a load balancing strategy is the number of rows moved per insertion or deletion. A node tries to adjust its load if the load increases or decreases by a factor δ. As an example of the strategy, for $\delta = 2$, when an insertion at node i increases the load of i beyond a threshold, here is the behavior:

—If at least one neighbor has a load that is less than or equal to half the load of i, move some data to the neighbor.

—If both neighbors have loads greater than half the load of i, find the globally smallest node k. If load of k is less than one quarter of the load of i, move all the data of k to its smaller neighbor, make node k a neighbor of i, and move half the data of i to k.

—Otherwise, do not move.

Deletion has comparable behavior. The authors prove some properties of the strategy. Load balancing can be broken into small actions that use block-level locks. The authors note that a system can give insertions and deletions a higher priority than load balancing gets. They also discuss networks where the number of nodes can change or where there is no central site for routing.

In a simulation, the number of nodes ranges from 2^4 to 2^{14}. The authors compare the performance using the strategy with performance using periodic offline reorganization. There are three phases: growth (involving 10^6 insertions), a steady phase (when insertions and deletions alternate for a total of 10^6 operations), and shrinkage (involving 10^6 deletions). The authors use several distributions of insertions and deletions. The results achieve the desired σ for a range of values of σ on various workloads. The cost, which is less than the cost of periodic reorganization, decreases as σ increases. Higher values of δ result in higher values of load imbalance; the system waits for more imbalance before rebalancing. The cumulative cost of data movement never grows more than linearly as a function of the cumulative number of insertions and deletions. As the number of nodes increases, the load per node decreases, making it easier to unbalance the system with fewer insertions or deletions, so more rebalancing is needed; the rebalancing cost per insertion or deletion increases.

4.3.2. Growth in the Network. The next topic in rebalancing deals with deciding whether the network should grow (based on meeting performance goals and avoiding unnecessary growth) or with scheduling of migration when the network grows. The system migrates some data to new nodes. Again, much of the work in this section builds upon other work in this section.

4.3.2.1. Vingralek et al. [1994]. For distributed data, Vingralek et al. [1994] have a goal of scalability (specifically, support for growing data and/or a growing workload via a linear increase in the number of servers). The system acquires a new server only if the overall server utilization in the system exceeds a threshold. Hash buckets can migrate from heavily loaded servers to more lightly loaded servers, and hash buckets can also split. The real concern is performance, not storage utilization, but the authors use the data capacity of a server to represent performance capacity. A logical-to-physical mapping table maps from hash buckets to servers; a copy of the table exists at each client and each server. If a server receives a request for a key that another server now owns, it forwards the request to that server and sends its own copy of the mapping table to the requesting client, which can then correct its copy.

The authors define several performance-related quantities. Each server has the same feasible capacity *Cf* (measured in the number of keys); this is a desired maximum workload. Each server also has a higher panic capacity *Cp*. Upon reaching *Cp*, a server cannot accept more keys. The global utilization of the system is the average number of keys per server divided by *Cf*. Vingralek et al. want to achieve these goals simultaneously: (1) The global utilization is at least a threshold, to limit the number of servers necessary to achieve the desired performance. (2) No server exceeds *Cp*.

The *file advisor*, which resides on one dedicated server, tracks (in its mapping table) the number of keys at each server. A server that exceeds *Cf* sends the advisor an overload message (including information about the server's buckets). Such a server sends the message repeatedly, after every *x* insertions, where *x* is typically 10–100. The advisor estimates the number of keys at servers that are not overloaded. When the system acquires a new server, the file advisor picks the server that has the highest number of keys and sends it a message to split its buckets with the new server. If a server *s* reaches *Cp*, but the global utilization would not exceed the threshold, then the advisor tries to find other servers that can accept some of the buckets of *s*. If the advisor succeeds, it tells *s* to migrate buckets out. If it fails, the system acquires a new server, and the advisor tells *s* to split its buckets. Message exchanges between the file advisor and a server can result in correction of the server's copy of the mapping table.

In a simulation model, *Cf* is 10,000 records, *Cp* is 11,00 records, and the global utilization threshold is 0.9. For simplicity, queries do not use caching, but insertions are batched. The experiment involves loading the data via insertion of 1000 records (with a total size of 10 megabytes) by each client, and then executing a mixture of 10% insertions and 90% queries until the file has grown by 10% from its initial size. The authors repeat the experiment for different numbers of clients and thus for different file sizes. As the number of clients ranges from 100 to 1000, the average response time of queries grows from 22.8 to 23.7 milliseconds (a narrow range). The average server utilization stays in another narrow range (0.91–0.93).

4.3.2.2. Breitbart et al. For an environment of distributed file servers, Breitbart et al. [1996] discuss rebalancing, which the system attempts when a server's load reaches the server's capacity. This work builds upon the work of Vingralek et al. [1994] (including a file advisor). Again, the authors want to achieve satisfactory performance with a minimal number of servers. Here, hashing or indexing maps keys to buckets, and a logical-to-physical mapping table maps from buckets to the servers that hold the buckets. A server can hold several buckets. Both splitting and migration of buckets are possible.

In a simulation model, reorganizers perform the splits and migrations. Many of the parameter values match those of Vingralek et al. [1994]. Breitbart et al. extend the earlier work by taking into account the extra message costs of forwarding and reorganization and by varying more parameters, including parameters for tuning. The fraction of insertions (after the initial insertions) ranges from 0% to 100% of the total user requests. The global load threshold ranges from 0.7 to 0.9, the initial number of buckets per server ranges from 1 to 10, and the number of clients ranges from 100 to 800. To expedite the load balancing, each server gives messages from the file advisor higher priority than messages from other servers or from clients.

Here are results of the model:

—As the number of clients grows from 100 to 800 (and thus the number of servers grows), the number of messages per access stays in the range of 2.03-2.05.

—As the file grows, number of reorganization messages divided by the file size is constant.

—If x (the number of insertions between successive overload messages) is in the range of 5–10% of *Cf*, there is a good balance between limiting the number of overload messages and limiting the error of the estimated load. There are 1,177 overload messages during a time period for $x = 1\%$ of *Cf*, vs. 271 messages for $x = 5\%$. For $x \geq 15\%$, the error of estimation exceeds 5%.

—If the initial number of buckets assigned to each new server is less than 5, there is a low precision of global load control, and the fraction of migration attempts that are successful is less than 0.7. If the number of buckets exceeds 20, there are unnecessarily frequent reorganizations without improving the precision of the global load control.

—A slack of over 15% between *Cf* and *Cp* allows too much server overload and thus causes more reorganizations.

4.3.2.3. Vingralek et al. [1998]. For a growing set of clients in a network of workstations, Vingralek et al. [1998] have a goal of scale-up with approximately constant data access time if the network size grows proportionally to the load. To achieve this goal, they investigate migrations across a growing set of workstations. The authors wish to avoid the overload that can result from having too few nodes and the unnecessary communication that can result from having too many nodes. This work is a follow-on to the work of Vingralek et al. [1994] and Breitbart et al. [1996].

A workstation might be both a server and a client. Vingralek et al. allow for the possibility of unequal access frequencies among the workstations and evolution of access patterns, causing change in the access frequency of any part of the database. Tracking of the workload enables automatic choice of the granularity of migration. Coarse-grained migration has the advantage of low overhead of data management, while fine-grained migration has the advantage of enabling fine-tuning of the load distribution.

A *fragment* is a group of records. It can have a variable size. A fragment is the default granularity of migration, but a server can migrate a subset of a fragment.

Each node has its copy of the *addressing scheme*, which is a lookup structure and a logical-to-physical mapping table. If a server receives a request for data that currently resides elsewhere, the server forwards the request to the server that it considers correct, and the acknowledgement to the client includes the server's copy of the addressing scheme.

The *load* of a fragment is the utilization (busy time). This load, calculated over the last 10–50 operations, is recalculated when a request is received (or at fixed time intervals if there have not been requests). The load of a server is the sum of the load of the fragments on that server. A server has a maximal load L (expressed as a utilization, typically 0.5–0.9), beyond which the response time degrades, so the system must migrate some fragments to another server.

The authors discuss two policies for such migration. In one policy (the *local policy*), an overloaded server migrates half of its load to a previously unused server. The server need not consult other used servers.

The other policy is the *global policy*, in which an overloaded server also considers used (but underutilized) servers. The server migrates data to an unused server only if the global load with an additional server would exceed a requested global load G. If the global load with an additional server would not exceed G, then the overloaded server migrates half of its excess load to the least loaded server that is already in use. If that server suddenly becomes more heavily loaded before it accepts the migration, a third server is picked instead. The global policy also enables the release of servers when the least loaded server finds that the global load (with one server removed) would be less than G. Here the server migrates its data to the second least loaded server (if possible) and sends its addressing scheme to other servers. In the global policy, servers should know each other's loads. Therefore, when a server's load changes by more than a threshold, it tells all the other servers its current load. Also, a server s that is a source or destination notifies other servers when its migration begins or ends, so that during that period, the other servers will not select s as the destination of another migration.

Vingralek et al. also discuss selection of fragments to migrate. They try to assure that after migration, the difference between the source server's load and the destination's load will be within a precision e. To try to achieve that precision goal, they sort the source server's fragments in decreasing order of load and then iterate over the list, deciding which fragments to migrate, based on the cumulative loads of the fragments. When that method would not achieve the precision goal, they split the most heavily loaded fragment into smaller fragments and try the method again.

Reorganizers, which ordinarily have lower priority than users, perform the migrations. During a migration, users query and update the original copy. The migration forwards the updates, and the source tracks the keys that have migrated. To prevent starvation of migration, a reorganizer gets a higher priority than users if the server's average access time for accessing blocks on the disk exceeds the time to process the disk accesses that remain to complete the fragment migration. To prevent performance degradation at the destination, the destination caches the frequently accessed records of the migrated segment; the source server has indicated which records were in the source's cache.

In a simulation model, clients only query and insert. 80% of the queries go to 20% of the key range. Each server has a cache. Vingralek et al. discuss several parameters: The workload can be static (if the query key range does not change) or dynamic (if the range periodically shifts). The workload can be monotonic (if the number of clients rises from 1) or non-monotonic (if the number rises from 1 and then decreases back to 1). G is 40%–80%, L is 50%–90%, e is 0.1%–50%, the maximum number of clients is 200, the mean request arrival rate is 10 per second, and the probability that a request is an insertion is 0.1.

In the results, with a monotonic workload, as the number of clients grows, the query response time fluctuates but remains within a range, so the strategy is scalable. With the global policy, the fluctuation increases as G increases from 40% to 70%. The local policy is scalable only under a static monotonic workload. Therefore, later results focus on the global policy. The user response time when using the priority heuristic mentioned earlier is (1) less than the response time when the migration priority is always high (for all reported values of G and L), (2) trivially higher than the response time when the migration priority is always low (for most reported values), and (3) less than the response time when the migration priority is always low (for a few high values of G and L).

The authors discuss the effects of varying the tuning parameters. As G increases from 40% to 70%, the user response time increases. As e increases from 0.1% to 50%, there are increases in the load imbalance, user response time, and frequency of need for migration, but there are decreases in the number of fragments created and in the size of the addressing scheme.

4.3.2.4. Ghandeharizadeh and Kim. The work of Ghandeharizadeh and Kim [1996] is oriented toward an environment of continuous media (audio and video). The system uses striping. The system allocates time slots to various tasks (user requests or reorganization requests). The authors investigate migration that accommodates an increase in the number of nodes.

Ghandeharizadeh and Kim describe two approaches for the process that reorganizes:

—When a user request reads an object that needs reorganization, the system allocates an extra time slot to write (reorganize) the object. If a new user request arrives and no idle time slots are available, the system can suspend a write. The system reorganizes an object *only* when the object is referenced. Therefore, infrequently referenced objects might remain unreorganized for a long time. This first approach can waste idle disk bandwidth, and it has an unpredictable time to reorganize the entire database.

—A different approach is to use two time slots per time period, on a regular basis, regardless of whether user requests arrive.

The authors suggest using both approaches. They note that it is possible to reduce the work of reorganization by loading new nodes with the blocks that would be assigned to them in reorganization; then the reorganization operates only on old nodes. They discuss trade-offs regarding when to delete the original copy of data.

4.3.2.5. Bratsberg and Humborstad. Another approach for online scaling [Bratsberg and Humborstad 2001] uses fuzzy reorganization to redistribute tables to accommodate the addition of nodes. Data is fragmented and replicated in a shared-nothing environment. The fragmentation can use hashing or ranges of key values. Each fragment has a mirrored fragment at another node; flowing of the log keeps the replica up to date. The system identifies data records and log entries by a primary key, so the reorganization needs no physical-to-physical mapping table. The main storage at each node contains a cache of dictionary information, which is a logical-to-physical mapping table that maps primary keys to primary fragments. Scaling consists of write-locking the dictionary, sending the data and log, changing the dictionary to record the new location, and unlocking. During scaling, user activities use an old version of the dictionary. The scaling takes place as a recoverable transaction.

Performance measurements involve three schemes for priorities: *eager* gives priority to reorganization, *lazy* gives priority to users, and *medium* is between those extremes. Each client reads 1 row, updates 1 row, or updates 4 rows. The wait time between a client's transactions is either 0 or 50 msec. Each of the 6 nodes contains 1 client. The table has 900,000 rows, and each node contains 115 megabytes. As the priority scheme varies from eager to medium to lazy, the time to scale increases, except for high update loads, where lazy is fastest. The increase factor for the average response time ranges from 3.4 to 8.0 for eager (where more reorganization interferes with users), from 1.1 to 3.4 for medium, and from 0.8 to 3.1 for lazy. The decrease factor for throughput is 0.2 to 1.0 for eager, 0.4 to 1.0 for medium, and 0.2 to 1.0 for lazy. A high load (a wait time of 0) has much more decrease (0.2 to 0.6) than a low load has (always above 0.9). The authors suggest doing the scaling before a high load occurs.

4.3.2.6. Sun. To move data between nodes to accommodate growth or shrinkage, Sun [2005, pp. 77–88] uses fuzzy reorganization. To reduce the number of log entries sent, the strategy filters out the log entries that do not refer to the copied data. Here are the steps in this fuzzy reorganization:

(1) Copy the data, while users can query and update the old copy. Create a physical-to-physical mapping table at the destination, to accommodate the changes in the records' RIDs. Create new secondary indexes at the destination.
(2) At the destination, apply the log entries, using the mapping table. Users can query and update the old copy.
(3) Quiesce updates while allowing queries, and apply the log again.
(4) Quiesce queries. While users have no access to the copied data, switch future accesses to the new copy by modifying the hash function (used for addressing) at all nodes.
(5) Allow queries and updates at the destination node. Clean up, for example, by deleting the old data.

4.3.3. Additional Topics. Other topics within rebalancing involve high availability, the cost of reorganization, updating of indexes to reflect migration, and heterogeneity of storage devices.

4.3.3.1. Chamberlin and Schmuck. For a replicated parallel database, Chamberlin and Schmuck [1992b; 1992a] emphasize availability, and they also discuss rebalancing; our discussion emphasizes the latter. Migration can take place to balance the load (including cases of adding a node) or to restore replication after a failure of a node. A logical-to-physical mapping table maps each block's identifier to the nodes that store the block. A coordinator node gathers load information, decides when a new mapping is needed, and broadcasts the new mapping. Each node keeps all mappings that still control the storage of some block at some node. When a node receives a new mapping, a reorganizer sends to other nodes the blocks that should be migrated out and then deletes the local copies of the blocks. After a block has migrated, the source node will acknowledge all requests for the block as "block not here." After the node has migrated out all the blocks that should be migrated out, the node notifies the coordinator. When all nodes have notified the coordinator that all migrations for an old mapping have occurred, the coordinator tells all nodes to delete this version of the mapping table. The authors also describe communication techniques to achieve high availability.

4.3.3.2. Achyutuni et al. [1993]. For reorganization to change the degree of allocation (number of disks) of a table, Achyutuni et al. [1993] investigate algorithms for placing hash buckets on nodes. Their goal is a strategy that balances the load, minimizes the cost of reorganization (specifically, the movement of data), and minimizes the *loss* (the number of update transactions that cannot execute during reorganization, due to conflicts with the reorganization).

The *weight* of a bucket can be either the number of rows or the number of accesses per unit of time. The authors describe several algorithms, which use weights:

—The classical Bubba placement algorithm [Copeland et al. 1988] iterates over the buckets from heaviest to lightest. The algorithm assigns each bucket to the disk with the least accumulated weight; it breaks ties randomly. There is a logical-to-physical mapping table.

—In optimization 1 of classical Bubba, if there are n disks, and this is a reorganization (i.e., this is not the first time that the system performs allocation), the system keeps the *n* heaviest buckets where they are, to reduce the cost of reorganization. It performs reallocation (as above) only for the remaining buckets.

—Optimization 2 starts from optimization 1, but it further reduces the cost of reorganization by omitting any exchanges of buckets between disks for which the difference between the buckets' weights is less than a threshold. The authors say that Bubba and its variations are good at balancing but can involve migrating a large amount of data.

—Hash placement uses a round-robin approach. Hashing is poor at balancing (if there is a high variance among the weights of the buckets), but the authors note that in some cases, hashing does not have a high cost of reorganization.

—The *Hubba* algorithm allocates a fraction of the buckets (starting with the heaviest) according to Bubba optimization 1; then it allocates the other buckets via round-robin. This algorithm represents a compromise between balance and low cost of migration.

Achyutuni et al. discuss three metrics for placement algorithms, and they model some of the algorithms:

(1) *Load balance*: For 4–12 disks, all algorithms except hash are effective at keeping the load deviation below 4%.

(2) *Cost of migration*: When the degree of allocation of a table changes from 4 to a higher value, the authors compute the sum of the weights of the hash buckets that must migrate. In classical Bubba, the fraction of the total data that must migrate is always at least 80%; the other algorithms usually have lower fractions. For a highly skewed distribution of accesses, Hubba is always lower than classical Bubba. For a uniform distribution, Hubba is usually lower, but not by as much.

(3) *Loss of update transactions during reorganization* [Srinivasan 1992; Srinivasan and Carey 1992b]: Achyutuni et al. develop a simulation model. The model uses a table size of 100,000 pages, pages of 4 kilobytes, one row per page, 100 hash buckets, 4–16 disks, a multiprogramming level of 40, 4–6 lock requests per transaction, and a probability of 0.25 that a transaction writes. When the cost of migration decreases, the loss and the response time also decrease. Hubba usually but not always has less loss than Bubba optimization 1.

4.3.3.3. Achyutuni et al. [1996]. Migration of data can require updating of indexes to reflect the migration. Achyutuni et al. [1996] and Omiecinski [1996, pp. 30–31] describe two techniques for updating indexes. In this environment, the fragments of a table on different nodes have separate indexes. One technique for updating indexes (OAT) moves one data page at a time, modifying the indexes for the page's data records. The other technique (BULK) copies the entire chunk of data to be moved; it uses fuzzy reorganization to update the indexes.

OAT uses a buffer at the source and a buffer at the destination. The buffer includes a hash table to search for a key in the buffer. It also includes a sorted list of keys extracted from the page in the buffer. For each data page to move, the procedure at the source is to lock the buffer, move the page into the buffer, unlock the buffer, process each index (by extracting the keys from the page, sorting, and deleting from the index), lock the buffer, send the page to the destination, receive an acknowledgement, delete the page from the buffer, and unlock the buffer. For each data page, the procedure at the destination is to receive the page, lock the buffer, move the page into the buffer, unlock the buffer, send an acknowledgement, process each index (by extracting the keys from the page, sorting, and inserting in the index), lock the buffer, store the page on disk, and unlock the buffer. During index update, users can access the data pages in the buffer. At any time, only one copy of a page (source or destination) is available to users.

For OAT, the authors describe update of an index when users update the page in the buffer:

—If a user transaction at the *destination* modifies data in the page, the system examines the position of the reorganizer in the sorted list of keys. If the reorganizer has already inserted the before-value in the index, the system deletes the before-value from the index and inserts the after-value, as if no reorganization were taking place. If the reorganizer has not yet inserted the before-value in the index, the system changes the position of the list entry from the before-value position to the after-value position. Then, if the after-value is before the current position of the reorganizer in the sorted list, the system also inserts the after-value in the index. If a user transaction at the *source* node modifies data in the page, the system does not change the index. The reorganizer will delete the before-value, and the insertion of the after-value will take place at the destination.

—If a user transaction at the destination node inserts data in the page, the system inserts in the index. If a transaction at the source node inserts data in the page, the system does not insert in the index at the source; it will insert in the index at the destination.

—If a user transaction at the destination deletes data from the page, the system deletes either from the index (if the reorganizer has inserted the key in the index) or from the sorted list of keys (otherwise). If a transaction at the source deletes data from the page, the system does not delete from the index.

The authors argue correctness of OAT; a user transaction reads each relevant data item just once, not twice or never.

For BULK, the procedure for the data is to copy the pages to be moved, send them to the destination, update the indexes at the destination, reflect user updates via fuzzy reorganization, redirect future transactions from the source to the destination by changing high-level information (e.g., the range of keys at each node), and delete the source copy. Update of an index consists of extracting the index keys into a bulk file, sorting the keys, and then processing sets of consecutive leaf pages (16 at a time). For each set of leaf pages, the system reads the pages into a buffer (using sequential I/O) and inserts keys from the bulk file into the pages (splitting if necessary). When fetching a leaf page from disk, the system searches the bulk file to find all index entries that can be inserted in the leaf page before the leaf page is released. The authors assume that the bulk file can fit in main storage.

Achyutuni et al. built a parallel database system, with B-tree indexes on fragments of tables. The authors describe experiments in which there are only 2 nodes. This small number clearly measures the effect of online reorganization, since every transaction uses a node that undergoes reorganization. There is one table, and there are up to 5 secondary indexes. Node 1 initially contains 80,000 rows, and node 2 initially contains 40,000. Each node has 1 disk and 128 buffer pages. Each transaction accesses 1 index to locate the row's page, retrieves the pages, accesses the row, and uses the processor. Each transaction is query-only. One parameter in the experiments is the choice between these behaviors: (1) All transactions are in a set of 200 hot pages, all of which are initially on the source node and half of which migrate to the destination. (2) Transactions are random access; again, 100 pages migrate between nodes.

In the results of the experiments, as the number of indexes increases from 0 to 5 in the hot spot case, the average response time for OAT varies little. The response time for BULK drops from about 45% higher than OAT for 0 indexes to about 3% lower than OAT for 5 indexes; the cross-over occurs between 2 and 3 indexes. The reason for the response time behavior is that with OAT, the throughput increases smoothly during reorganization. With BULK, the throughput jumps after completion of insertions in an index. The jump occurs early in reorganization (25% of the total time) for 5 indexes and later (55%) for 1 index. For the experiments with no hot spots, OAT and BULK have similar response times.

The time to reorganize the indexes rises approximately linearly as the number of indexes increases from 0 (for both OAT and BULK), but OAT has a much higher time, for example, about 32 times higher for 5 indexes. The reason is that OAT involves more paging out before reading a page; BULK uses sequential I/O. For the experiments with no hot spots, the time to reorganize is like the hot-spot case.

The authors compare the two techniques. OAT uses little extra disk space; BULK can use a large amount. BULK uses sequential prefetch I/O; OAT does not. BULK is much faster. For tables with 1 or 2 indexes, OAT degrades user performance less than BULK does. For 3–5 indexes, BULK yields slightly better user performance.

4.3.3.4. Bratsberg et al. For an environment of parallel nodes with replication of data, Bratsberg et al. [1997] discuss several aspects of operations and high availability, most of which do not deal specifically with rebalancing. One of these aspects is fuzzy migration of fragments, which a low-priority reorganizer can use for adding nodes. Access to fragments uses a logical-to-physical mapping table.

4.3.3.5. Ghandeharizadeh et al. For a storage area network with heterogeneous disks (having different capacities and different bandwidths), Ghandeharizadeh et al. [2006] want to minimize the average response time. The maintenance migrates fragments of files between disks. This system partitions time into fixed-length slices. In each time slice, maintenance involves these steps:

(1) Calculate the load (bytes retrieved) and average response time of each disk, and calculate the load imposed by each fragment.
(2) Choose which fragments to migrate, using one of the alternative criteria described below.
(3) Using a reorganizer, migrate the chosen fragments. During migration, queries use the source disk, and updates go to both disks.

The authors discuss these alternative criteria for Step (2):

—EVEN: The goal is to distribute the load uniformly across the disks, so that if a disk's bandwidth is $n\%$ of the total of all the disks' bandwidths, then the disk's ideal contribution of consumed bandwidth is $n\%$ of the applications' total consumption of bandwidth. Specifically, identify the disks with the highest positive load imbalance (their contribution of bandwidth minus their ideal contribution) and the disks with the lowest negative load imbalance. Among the fragments on the highest positive disk, find the one whose imposed load is closest to the negative imbalance on the lowest negative disk, and schedule it for migration. Repeat this calculation for all unbalanced disks.

—PYRAMID$_{SP}$: In this and the remaining criteria, the system conceptually sorts the disks by speed, and the goal is to maximize the utilization of the fastest disks by migrating the fragments that impose the greatest load toward the fastest disks. Within a cluster of same-speed disks, these criteria use EVEN. In PYRAMID$_{SP}$, migrate the fragments that impose the highest load to the fastest disks until exhaustion of the disks' storage capacity.

—PYRAMID$_{BW}$: Migrate the fragments that impose the highest load to the fastest disks until estimated exhaustion of the disks' bandwidth.

—PYRAMID$_{BW,C/B}$: This extends PYRAMID$_{BW}$. Sort the candidate migrations by benefit minus cost. The cost of a migration is the time for the source to read the fragment plus the time for the destination to write the fragment. The benefit is the estimated average response time after migration minus the observed average response time. Perform the highest-savings migrations first. As part of this criterion, if two segments are often referenced at the same time, then they should migrate to different disks.

In a simulation, 83% of the accesses reference 10% of the blocks. Of the 9 disks, 3 each have a capacity of 180 GB and a transfer rate of 40 MB/second, 3 have 60 GB and 20 MB/second, and 3 have 20 GB and 4 MB/second. In the results, at the end of the last time slice, PYRAMID$_{BW,C/B}$ has the lowest average response time (about 60% of the next best criterion's response time). The authors note that this criterion uses cost and benefit and that segments that are often referenced at the same time migrate to different disks.

4.4. Garbage Collection for Persistent Storage

Our next category of maintenance is garbage collection. Several languages support linked data structures. The *roots* of such chains might be global variables, static variables, stack variables, and registers. In such languages, one recurring task is *garbage collection*, for example, reclamation (freeing) of storage that is no longer directly or indirectly reachable from roots and thus is no longer used. Many authors use the term *mutator* to mean a user process, which might make storage unreachable. Surveys by Cohen [1981], Wilson [1992], Jones and Lins [1996], Abdullahi and Ringwood [1998], and Detlefs [2004] discuss issues in garbage collection and include many strategies for offline or online garbage collection for *volatile* storage. Besides the work that the surveys cover, additional work in online garbage collection (and related maintenance) for volatile storage includes that of Sockut [1974, pp. 43–45], Bahaa-El-Din et al. [1989], Bastani et al. [1991], Nilsen and Schmidt [1992], Gupta and Fuchs [1993], Chen and Banawan [1993], Schmidt and Nilsen [1994], Domani et al. [2000b], Printezis and Detlefs [2000], Domani et al. [2000a], Salant and Kolodner [2002], Ossia et al. [2002], Kolodner and Petrank [2002], Barabash et al. [2003], Azatchi et al. [2003], Blackburn and McKinley [2003], Blackburn et al. [2004], Detlefs et al. [2004], Bacon et al. [2004], Paz et al. [2005], Kolodner et al. [2005], Barabash et al. [2005], Vechev et al. [2006], Kermany and Petrank [2006], Bacon [2007], Vechev et al. [2007], McCreight et al. [2007], Paz et al. [2007], Kero et al. [2007], Joisha [2007], Pizlo et al. [2007], Stanchina and Meyer [2007], Wegiel and Krintz [2008], Joisha [2008], Siebert [2008], Trancón y Widemann [2008], Pizlo et al. [2008], Garthwaite [2008], Seidl et al. [2008], Puffitsch and Schoeberl [2008], Schoeberl and Puffitsch [2008], Tene and Wolf [2008], Bollella et al. [2009], and probably others. Of course, work in online garbage collection for volatile storage has heavily influenced work in online garbage collection for *persistent* storage, which we will survey.

Here are differences between issues in designing online garbage collection for volatile storage and issues in designing online garbage collection for persistent storage:

—For persistent storage, we are more likely to support recovery and enforce transaction semantics of user actions.

—The size of stored data in persistent storage is likely to exceed the size of stored data in volatile storage. Therefore, the considerations for performance for garbage collection for persistent storage are more likely to include transfer of data between main storage and secondary storage [Franklin et al. 1989, p. 7]. Of course, this performance issue is also important for large volatile storage [Cohen 1981, p. 350]. Transfer of data is a concern for the garbage collection itself, for users during garbage collection, and for users after garbage collection (e.g., as a result of the degree of clustering that the garbage collection has produced).

—For persistent storage, an object's[3] lifetime is likely to be longer. Therefore, the ratio of reachable storage to garbage might be higher than for volatile storage [Franklin et al. 1989, p. 7].

We can collect garbage in place in one data area (without movement between pages) or collect garbage by copying data from one area to another. For garbage collection by copying, some authors use the terms *from-space* and *to-space* for the old and new copies. For a variety of categories of reorganization, the beginning of Section 3.3 discussed trade-offs between online reorganization in place and online reorganization by copying. Besides most of those trade-offs, here are additional trade-offs for garbage collection:

—Clustering can be easier to achieve in a copying strategy. Collection can (but does not necessarily) trace objects in a logical order and cluster them as it writes them in the new area.

—Similarly, a copying strategy can compact the allocated space more easily, reducing the fragmentation of the free space. However, some systems allow compaction within a page even without garbage collection by copying [Amsaleg et al. 1995; 1999].

—If the collector cannot always ascertain whether a value is a pointer, we cannot risk moving the possible object and updating the possible pointer to it, which might not really be a pointer. Therefore, we must perform at least part of the garbage collection in place [Wilson 1992, pp. 18–19].

Some strategies for garbage collection divide the database into partitions and collect one partition at a time. Partitioned collection has several advantages:

—It increases the locality of reference of the collection. In particular, we can reclaim some space without scanning the entire database.

—It can limit the disruption of users, even if a strategy locks an entire partition.

—For collection by copying (discussed in Section 4.4.6), it limits the extra space required for one unit of copying.

Partitioned collection also has disadvantages:

—It requires space and time to maintain lists of references from objects in one partition to objects in another partition.

—If such cross-partition references form a cycle of garbage, collection of just one partition at a time does not reclaim the cycle, since each object in the cycle appears reachable. If we do not supplement partition collection with cross-partition collection, such a cycle will remain uncollected.

We will survey strategies for online garbage collection for persistent storage (in place and by copying). The strategies for persistent storage use techniques from some strategies for offline or online garbage collection for volatile storage. Therefore, for garbage collection in place and then for garbage collection by copying, we will discuss these topics:

(1) We will sketch strategies for volatile storage that are relevant to the strategies for persistent storage. We will not survey all strategies for volatile storage; the earlier

[3]Our discussion of garbage collection uses *object* to mean a unit of data. Unless otherwise indicated, we do not require an object to have "object-oriented" features, e.g., methods and inheritance.

surveys [Cohen 1981; Wilson 1992; Jones and Lins 1996; Abdullahi and Ringwood 1998] describe additional strategies for volatile storage.

(2) We will survey online nonpartitioned strategies for persistent storage.

(3) We will survey online partitioned strategies for persistent storage.

We will also survey comparisons of performance of several aspects of garbage collection for persistent storage.

4.4.1. Garbage Collection in Place for Volatile Storage. We begin by sketching relevant strategies for offline or online garbage collection in place for *volatile* storage.

Suppose that each object includes a *reference count* (a count of the objects and roots that reference the object) [Collins 1960]. When the count decreases to zero, we can conclude that the object is garbage and reclaim the object's storage. This simple strategy is online; i.e., garbage collection usually avoids a long *pause* (period of no access) for users. If the loss of reachability for one object cascades into loss of reachability for many other objects, we might defer the actual reclamation and perform it incrementally [Wilson 1992, p. 6], to limit each pause. Reference counting ordinarily reclaims storage immediately. However, this strategy requires space for the count, and it requires updating the count whenever we add or delete a reference. A more important disadvantage is that the strategy fails to reclaim a directed cycle of objects that are garbage. This failure occurs because each object in the cycle still references its successor, which thus still has a positive count. Therefore, if cycles can form, this strategy alone is inadequate, but some systems combine this strategy (to exploit its advantages) with more expensive but infrequently executed strategies that do reclaim cycles.

Another limitation of reference counting applies if reference counts have a maximum possible value that is less than the maximum number of possible references. After a particular reference count reaches the maximum value, we cannot increment it to represent additional references, so the only safe assumption is that the actual number of references might be arbitrarily higher than the maximum value. Therefore, we can never safely decrement this count to 0. Therefore, we can never reclaim this count's object via reference counting alone, even if all actual references to the object eventually disappear.

Deferred reference counts [Deutsch and Bobrow 1976] use a sequential file and hash tables to achieve the effect of storing reference counts. A sequential file records the transactions that can affect reference counts; this file resembles a log. We maintain a hash table of addresses of objects that have a count that is greater than one and a hash table of addresses of objects that have a count of zero, not counting references from variables on the stack. At convenient intervals, we read the information from the sequential file, and we use this information to bring the hash tables up to date. We can then reclaim the storage of objects that have a count of zero and are not referenced by variables on the stack. Advantages of this strategy involve space and time:

—No object actually contains space for a reference count, and an object that is referenced by exactly one object (and thus effectively has a reference count of one) occupies no space in the hash tables. According to the authors, some statistics suggest that few objects are referenced by more than one object.

—We need not *immediately* take the time to reclaim storage.

The authors supplement their reference count strategy with an infrequently executed copying strategy, which can reclaim unreachable cycles.

Cyclic reference counting [Brownbridge 1985] extends ordinary reference counting to handle cycles.[4] Jones and Lins [1996, pp. 58–62] review some corrections to the original article. In cyclic reference counting, we label each pointer with a strength (*strong* or *weak*). Each object has two reference counts, one for incoming strong references and one for incoming weak references. We maintain a rule that the set of all strong pointers forms a directed acyclic graph that connects all the reachable objects and no unreachable objects. Therefore, strong pointers alone do not form cycles, but weak pointers alone (or a combination of strong and weak pointers) might form cycles. When a program adds a pointer to a preexisting object, a cycle might form, so we label the pointer weak. When creating a new object and a pointer to it, no cycle forms there yet, so we label the pointer strong.

In cyclic reference counting, when deleting a pointer from an object (r) to another object (s), there are 4 cases:

—If the pointer is weak, decrement the weak reference count of s.

—If the pointer is strong, and the weak reference count is 0, decrement the strong reference count. Then, if the strong reference count has become 0, reclaim s and recursively delete any pointers from s, as in ordinary reference counting.

—If the pointer is strong, the weak reference count exceeds 0, and the strong reference count exceeds 1, then decrement the strong reference count.

—If the pointer is strong, the weak reference count exceeds 0, and the strong reference count is 1, then this deletion represents removal of the last strong pointer to s, but there are still weak pointers to s. This is the case in which ordinary reference counting might leave a cycle of garbage. Perform several steps in this case: (1) Decrement the strong reference count. (2) Convert the weak pointers to s into strong pointers. Brownbridge describes an implementation technique that performs this conversion by just flipping one bit in s. (3) Perform a localized mark of the descendants of s, which typically constitute a small subset of the entire set of objects. In the marking, for each object, maintain an extra reference count of pointers from marked objects. (4) Among the descendants, search for each object whose reference count (strong plus weak) exceeds the extra reference count. Such objects are reachable from outside the set of descendants. Unmark these objects and their descendants, and adjust the strengths of pointers where necessary to maintain the rule described earlier. (5) If s is marked now, then it is garbage, so reclaim it and recursively delete any pointers from s.

Besides reference counting, another in-place strategy is *mark and sweep* [McCarthy 1960, pp. 192–193], which typically includes these phases:

(1) Starting at the roots, we trace the reachable objects and mark them as reachable. In a depth-first traversal, if we find an object that we have already marked, we have already traced its successors, so we do not trace them again.
(2) We sweep through storage and reclaim the unmarked objects.

This offline garbage collection can produce a long pause for users.

Dijkstra et al. [1978] describe an online strategy to mark and sweep without any long pause for the user. Instead of using two states of marking (marked as reachable and not marked as reachable), we use three states:

[4]The body of this article by Brownbridge describes one strategy for cyclic reference counting. The appendix of that article defines a second strategy by modifying the strategy of the body, and our description here applies to the second strategy.

White :	We have not marked the object as reachable. This is each object's initial state.
Gray :	We have marked the object as reachable, but we have not yet traced its successors.
Black :	We have marked the object as reachable, and we have traced its successors. Black is each reachable object's final state.

The garbage collector executes as a reorganizer. It first makes the root objects gray. It then processes each gray object by marking it as black after marking its white successors (if any) as gray. The user's addition of a pointer to a white object also marks the object as gray; online operation requires such interaction between users and garbage collection. When no more gray objects exist, the collector's sweep through storage reclaims white objects.

4.4.2. Online Nonpartitioned Garbage Collection in Place for Persistent Storage. Next we discuss strategies for online garbage collection in place for *persistent* storage. The strategies in this section do *not* divide the database into partitions. Later we will discuss partitioned collection.

4.4.2.1. Birrell and Needham. The work of Birrell and Needham [1978] deals with a garbage collector for a file system. A central table stores each object's reference count and object type (directory or segment). The reference count suffices for reclaiming segments, which constitute most of the objects and which have no cycles. To reclaim directories, which *can* have cycles, a reorganizer marks and sweeps. The garbage collection for directories initially scans the central table and sets up a data structure with an entry for each directory, describing its state of marking. The collection strategy and the three possible states resemble the strategy and states of Dijkstra et al. [1978]. The garbage collector is notified of users' creation or retrieval of directories. To reduce the degradation of users' performance, the collector executes slowly. Garnett and Needham [1980] describe a successor with similarities to the garbage collector described above.

4.4.2.2. Almes. For an operating system that supports object-oriented programming, Almes [1980] describes garbage collection for persistent objects. The garbage collection uses both reference counting and a strategy that marks and sweeps. An object might be both *active* (i.e., in primary storage) and *passive* (i.e., in secondary storage). The active symbol table tracks active objects; the passive symbol table tracks passive objects. Each active object has two reference counts. The *total* reference count represents all references; the *active* reference count represents only active references, that is, those that have been converted to physical addresses. When the total reference count becomes zero, the object can be reclaimed. When the active reference count becomes zero, the object becomes just passive and is removed from the active symbol table. Also, a process scans the active symbol table for objects that have not been accessed recently. The process makes each such object's data passive, converts the object's active references to passive form (i.e., names), and reclaims the storage that the data and pointers occupied. The conversion of references decrements the active reference counts for the referenced objects.

In an unreachable cycle in the *active* symbol table that Almes describes, eventually some objects will reach a state of not being accessed recently. As described above, this leads to decrementing of active reference counts for the objects (in the cycle or elsewhere) to which they point. This leads to removal of objects from the cycle when the active reference counts become zero. Thus an unreachable cycle in the active symbol table will eventually disappear; the objects become just passive.

However, reference counting cannot reclaim an unreachable cycle of *passive* objects (for which a lack of recent accessing does *not* cause reclamation). For passive objects, therefore, Almes also uses another online strategy, which resembles the strategy of Dijkstra et al. [1978]. Almes describes the implementation, including these techniques:

—If several objects are gray, the system must select one to process next. To reduce the seeking on disks, Almes selects the object whose data has the lowest disk address.

—After marking a white object as gray, the strategy of Dijkstra et al. will *eventually* trace the object's successors (if any) and then mark the object as black. If such an object contains no pointers (a condition that can be tested efficiently in this operating system), Almes marks it as black *immediately*, instead of marking it as gray and later marking it as black.

—Almes maintains a list of recently traversed (and thus nonwhite) objects. Then, the garbage collection's processing of a pointer to such an object need not check the object for whiteness.

4.4.2.3. Wolczko and Williams. The strategy of Wolczko and Williams [1992] applies to a persistent object system that has three levels of storage: cache, main storage, and disk. All creation of objects takes place in the cache. Each object is indexed by its object identifier; the index (object table) serves as a logical-to-physical mapping table. The index is a hierarchy of specialized objects. Design goals of the garbage collection include (1) reclamation of most (not necessarily all) garbage in each pass of collection and (2) low degradation of user performance, for example, by limiting any pauses.

The goal of performance within the architecture of this virtual storage system influenced design decisions in the garbage collection. Specifically, each pass of collection has three levels, corresponding to the three levels of storage:

(1) An execution of garbage collection in the cache blocks user access, for a time that is proportional to the cache size. It uses a mark and sweep, whose roots are (1) registers and (2) objects that have been referenced from outside the cache. The marking resides in objects' headers. When an object's identifier is written into main storage, the system labels the object's header as being referenced from outside. The authors' simulations indicate that this collection reclaims 90% of all garbage.

(2) The collection for main storage resembles that of Dijkstra et al. [1978]. The field for color resides in the index. Objects referenced from disk are marked gray.

(3) For objects on disk, the collection uses reference counting. The index stores each object's reference count, which counts the references from disk. The collection reclaims an object on disk if its count is 0 and the collection for main storage has not found a reference to the object. As with other reference counting implementations, this reference counting does not reclaim a cycle of garbage on disk, and it does not reclaim an object whose count has reached the maximum possible value, which is 255 in this implementation.

4.4.2.4. Skubiszewski and Valduriez. The work of Skubiszewski and Valduriez [1997] involves garbage collection for a client-server object-oriented database. The garbage collection uses mark and sweep, with three-color marking as in Dijkstra et al. [1978]. The marking executes as a client. Instead of storing marks physically in objects, the marking uses a separate list of reachable objects.

To determine which objects are reachable, the marking uses a set of *virtual* copies of database pages. If the marking must access a database page that has not yet really been copied, the system makes a real copy of the page. If a user transaction writes a page that the system has already really copied, then just before commitment, for each page that this transaction has read or written but that does not yet have a copy, the page is labeled as requiring eventual real copying. If a later transaction writes a page that requires copying, the system then makes a real copy of the page for the marking.

The authors describe an offline implementation. In performance experiments, the marking and the server reside on the same machine. As the database size increases, up to 2.4 gigabytes, the time to mark the database rises slightly more than linearly. As the fraction of objects that are garbage increases, the time to sweep rises slightly less than linearly.

4.4.2.5. Ashwin et al. For an object-oriented database, Ashwin et al. [1997] and Roy et al. [1998] use a version of reference counting based on cyclic reference counting [Brownbridge 1985]. A localized mark and sweep collection supplements the reference counting, to reclaim cycles of garbage. The database can be centralized or client-server.

As a prerequisite for the mark and sweep, log analysis executes whenever a transaction appends a log entry. The analysis could execute during user activities (using the required concurrency control) or as a reorganizer at the server. The analysis maintains the strong and weak reference counts. For adding a pointer to a preexisting object, by default the log analysis labels the pointer as weak; this default behavior avoids a cycle of strong pointers. However, the authors observe that if a link in a schema is *acyclic* (participates in no cycles), then any corresponding pointer (instance) could not participate in a cycle, so it can be labeled strong. Consequently, if all links are acyclic, then all pointers are strong.

The mark and sweep collection executes periodically to reclaim cycles of garbage. For each object that has a strong reference count of 0 and a weak reference count greater than 0, the collection performs a localized mark and sweep. The system has kept track of the objects that have this combination of reference counts. This mark and sweep is based on the mark and sweep of Brownbridge, but it batches the work for several objects, instead of using Brownbridge's approach of marking whenever any object reaches the above combination of reference counts. Objects that are referenced from currently active transactions are not considered garbage. The garbage collection also refrains from following pointers whose schema links are acyclic.

The authors implemented and compared their strategy and a partitioned mark and sweep. They measure performance, using slightly over 100,000 objects, which occupy 4.7 megabytes. The buffer pool contains 500 4-kilobyte pages. For one workload, the overall I/O for their strategy is 61% of the I/O for partitioned mark and sweep. The latter also leaves some garbage.

Roy et al. [2002] describe an extension of the original technique.

4.4.2.6. Comparison. Most of the work that we described in this section uses *both* reference counting (which does not reclaim cycles of garbage) and mark and sweep (which reclaims cycles). The strategies use a variety of techniques to enhance performance: The collector of Birrell and Needham executes slowly, to reduce the degradation of users' performance. The techniques of Almes and of Wolczko and Williams take into account the levels of objects in a storage hierarchy. Almes and Roy et al. maintain lists of certain objects, to enhance performance. Skubiszewski and Valduriez use virtual copies of objects. Roy et al. use strengths of pointers and properties of the schema.

4.4.2.7. Purging of Old Data. Another strategy deals with purging of old data. This type of maintenance resembles a subset of garbage collection, namely sweeping to reclaim objects. Sockut and Goldberg [1979, pp. 388–389] include a case study of this strategy, which is part of a highly available system for airline reservations [Siwiec 1977]. Flights for the next *n* days have a fine index, which enables quick access; flights for later days have a coarse index, which saves space. Each night, the system purges records for flights that have already flown, and it constructs fine indexes for the flights for the *n*th day. The fine granularity of locking limits blockage of users to just a few seconds. The system marks the space for the purged records as deleted, but it does not yet make the space available for reallocation. Another online process scans the records and notes in an allocation table that the space for the purged records is available for reallocation. This process typically requires 3 to 20 hours; it executes at least once a week. In each area of the database, uniformity of the size of records obviates any compaction. At least one other commercial DBMS also defers the physical deletion of record headers until a later transaction [Sockut and Goldberg 1979, p. 386].

4.4.3. Online Partitioned Garbage Collection in Place for Persistent Storage. This section deals with online partitioned collection. The strategies here divide the database into partitions, and they collect one partition at a time.

4.4.3.1. Amsaleg et al. For a client-server object-oriented database system, Amsaleg et al. [1995; 1999] describe server-based online garbage collection. Pages are transferred between clients and the server, a client sends written pages to the server only after sending their log entries, and all client-written pages are copied to the server before commitment. The mark and sweep garbage collection executes at the server. Any crash of the server during collection results in discarding of that pass of collection.

The goals of Amsaleg et al. included avoidance of several problems:

(1) If an uncommitted transaction made an object unreachable (garbage), sent a written page to the server, and rolled back after collection, then the object might be collected inadvertently.
(2) If a reachable object temporarily appears unreachable at the server because a client has sent some but not yet all written pages to the server, then the object might be collected inadvertently.
(3) Suppose that (1) a page has been collected at the server but has not yet been written to disk at the server, (2) a client allocates objects on the collected page and then commits, and (3) crash occurs. Then the subsequent redo operates on an uncollected version of the page and thus might not have enough free space for the redone allocations.

To avoid problems 1 and 2, if a transaction creates an object or breaks a reference to an object, the object cannot be reclaimed in a collection that begins before completion of this transaction. This rule allows collection to execute without locking and to ignore the program state (e.g., program variables) of concurrently executing transactions. To avoid problem 3, reclaimed space can be reused only after its reclamation is reflected on stable storage.

Based on incoming log entries, the garbage collector maintains a data structure (the pruned reference table, PRT); each entry contains (1) the identifier of an object for which at least one incoming reference is broken and (2) the identifier of the transaction that broke the reference. When a transaction terminates, its PRT entries are flagged; they are removed before the start of the next garbage collection. PRT entries are additional roots of persistence, thus implementing the avoidance of problem 1 above. Another structure (the created object table, COT) contains the identifiers of created objects and the identifiers of the transactions that created the objects. The collector creates and prunes the COT similarly to the PRT. Collection refrains from reclaiming objects that the COT references, thus implementing the avoidance of problem 2 above.

With partitioning, each partition has a list of incoming references; this list is a root of persistence. Cycles of garbage that cross partitions are not collected. After collection of a partition, the collector removes untraced outgoing references in the incoming reference lists of other partitions.

Amsaleg et al. implemented the collection. For their performance measurements, the log and the database reside on separate disks. Pages are 8 kilobytes, and each object is 80 bytes. One parameter is the fraction of objects that are garbage. Another parameter is the clustering factor, that is, the fraction of the reachable objects on a page that can be traversed before crossing a page boundary. For various test cases, the overhead to maintain the PRT and COT (without actual collection taking place) ranges from 0.7% to 7.4%. The execution time for offline collection rises approximately linearly as the partition size rises from 500 to 10,000 pages. For online collection, after the sleep time (the delay in scheduling a processor time slice for garbage collection) crosses a threshold, user performance is almost as good as performance without garbage collection. For example, with a single partition, for various test cases, the overhead of collection ranges from 6.6% to 10.7%.

4.4.3.2. Maheshwari and Liskov. For a client-server persistent object store, Maheshwari and Liskov [1997] collect garbage in one partition at a time. Information about inter-partition references resides on disk, for reasons of size and recoverability. Each ordered pair of partitions *i* and *j* has an inter-partition list, which records the references from objects in partition *i* into objects in partition *j*. Each partition has a list of incoming references, which is a set of pointers to inter-partition lists for references into this partition. Similarly, each partition *i* has a list of outgoing references, which is a set of pointers to inter-partition lists for references from partition *i*. The outgoing list is inessential, but it speeds the pruning of other partitions' incoming lists (by obviating the scanning of those incoming lists) during removal of references that were not traced during the garbage collection of partition *i*.

The authors' description is generic with respect to the policy to choose a partition to collect and the collection strategy within a partition. The roots to trace a partition are the persistent root of the heap, roots from applications, and incoming references. Collection can compact within a page; compaction does not change objects' logical names. At the end of collection in a partition, the collection updates the inter-partition lists for outgoing references.

The partitioned collection alone does not collect inter-partition cycles of garbage, so a global marking supplements the partition-based collection. For each partition, the global marking takes place when the partition collection takes place. During the tracing of a partition, any objects that were not marked in the previous pass of global garbage collection are reclaimed. Then, the marking propagates from the partition's roots to the outgoing list's references. Newly created objects are also marked. The marking is complete when marks have propagated through all partitions.

The collection executes at low priority. The performance measurements apply to a database size of 256 megabytes and a partition size of 1 megabyte. As the fraction of references that cross partitions increases from 0 to .15, the overhead due to processing lists increases approximately linearly, the overhead due to disk I/O increases more than linearly, and the overhead due to scanning objects is constant.

4.4.3.3. Comparison. The authors emphasize different features. Amsaleg et al. emphasize correctness and efficiency of collection in a recoverable client-server environment. Maheshwari and Liskov emphasize efficient and recoverable maintenance of inter-partition references for collection that reclaims inter-partition cycles of garbage.

4.4.4. Garbage Collection by Copying for Volatile Storage. Now we turn from collection in place to collection by copying. We sketch relevant strategies for offline or online garbage collection by copying for *volatile* storage. In each such strategy, we *flip*; that is, we exchange the roles of the original area and the new area, so that future accesses by users will apply to the new area. In the online strategies, the flipping occurs when the collection starts.

In a typical *copying* strategy, we copy all reachable objects, producing a *compact* new copy of the data [Cheney 1970]. We start by copying the objects to which the roots point. We then scan the objects in the new area. For each such object, we follow its pointers (if any) to objects in the old area, copy those objects to the new area, and update the pointers. Whenever we copy an object, the old copy acquires a forwarding pointer to the new copy. If the collector later reaches the old copy again (by following a pointer from another object), we will update the pointer that we followed and will avoid copying the object again. After copying all reachable objects, we reclaim the old area and allow users to access the new area. This offline strategy can produce a long pause for users.

Baker [1978] describes *incremental* copying. During each user activity that allocates storage, we copy and compact data incrementally (i.e., we copy a few objects) before actually performing the allocation (from the new area). In each increment of collection, we scan objects in the new area. For each such object, we follow its pointers (if any) to objects in the old area, copy those objects to the new area, insert forwarding pointers in the objects in the old area, and update the pointers in the object being scanned in new area. We track the position of the scanning in the new area, and the next increment will resume at that position. Also, when a user attempts to access an object in the old area, we trap the attempt, copy the object to the new area, and redirect the user's access into the new copy. From the perspective of the user, this redirection achieves the effect of the flipping at the start of garbage collection. Of course, this trapping and copying cause the collector to process such an object sooner than it ordinarily would. After copying all reachable objects, we reclaim the old area. This strategy usually produces a short pause for users. However, the trapping requires checking whether a pointer refers to the old area or the new area. Also, soon after flipping, we have not yet copied most of the objects, so the trapping occurs more frequently. Therefore, the length of pauses soon after the flipping can exceed the length later in the garbage collection [Kolodner and Weihl 1993, p. 180].

In the context of a copying strategy similar to that of Baker [1978], Appel et al. [1988] describe a method to avoid expensive checking of whether a pointer refers to the old area or the new area. The protection mechanism for the virtual storage prevents users from accessing pages in the new area whose objects we have not yet scanned. The handling of a protection trap scans the objects on the page. Appel et al. also execute the garbage collection as a reorganizer, instead of just incrementally during user activities that allocate storage. For each scanned object, we follow its pointers (if any) to objects in the old area, copy those objects to protected pages in the new area, insert forwarding pointers in the objects in the old area, and update the pointers in the object being scanned the new area. After scanning all the objects on a page in the new area, we remove the protection and (if the scanning resulted from trapping) resume the user's execution. Therefore, all the pointers in any unprotected (user-accessible) page refer to the new area. As in the strategy of Baker, the length of pauses soon after flipping can exceed the length later in the garbage collection.

Bartlett [1988] describes a *mostly copying* strategy. Here the collector need not be able to determine whether a value in the user's activation stack (a potential root) is a pointer. We can move a possible referenced object and update the possible pointer only if we can determine that the possible pointer really is a pointer. Therefore, for any page to which the activation stack or the registers might point, collection does not copy the page from the old area to the new. Instead, it marks the page as logically being in the new area. It then copies (from the old area to the new) the objects that are reachable from the pages that it marked as logically being in the new area. Therefore, it copies most of the pages containing reachable objects. This offline strategy can produce a long pause for users.

4.4.5. Online Nonpartitioned Garbage Collection by Copying for Persistent Storage. Next we discuss strategies for online garbage collection by copying for *persistent* storage. These strategies do *not* use partitions. Later we will discuss partitioned collection.

Detlefs [1990] based his collector, which executes as a reorganizer, on the mostly-copying strategy of Bartlett [1988]. The system initially halts users, flips, scans the roots, finds the pages to which the roots might point, marks the pages as logically being in the new area, and then allows users to resume. Users query, allocate, and update in the new area. Use of the method of Appel et al. [1988] traps users' attempts to access pages in the new area whose objects have not yet been scanned. The collector scans pages in the new area. For each such page, the system pins the page in volatile storage (i.e., prevents the page from being paged out) and scans the objects in the page. Upon finding (in the scanned page) a pointer to an object in the old area that has not yet been copied, the system copies the object to the new area, pins its old page in volatile storage, inserts a forwarding pointer in the object in the old page, and changes the pointer in the scanned page (in the new area) that pointed to the object. After the scanning, the system writes a garbage collection entry in the log, and it writes these pages to persistent storage: (1) the pages (in the new area) that hold copied objects, (2) the pages (in the old area) that held the copied objects, and (3) the scanned page in the new area. The system then writes a commitment entry (for the page scan) in the log, and it removes the protection for the scanned page, so that users can access the page. A garbage collection entry in the log refers to the scanned page in the new area and to the pages in the new area that hold copied objects. The granularity of concurrency control is a scan of the roots (at the start) or a scan of a page (after scanning the roots).

The collector of Kolodner [1990; 1992] and Kolodner and Weihl [1993] uses a copying strategy [Baker 1978]. It uses the method of Appel et al. [1988] to trap the user's attempts to access pages in the new area that have not yet been scanned. It initially flips and copies the root objects to the new area. The user queries, allocates, and updates in the new area. As in the strategy of Baker, during each user activity that allocates storage, this strategy copies and compacts data incrementally (i.e., it scans part of the new area) before actually performing the allocation in the new area. Kolodner notes that the strategy can be extended so that collection executes as a reorganizer, along with multiple users. The collector scans objects on pages in the new area. In scanning such a page, the system pins the page in volatile storage, scans it for pointers to the old area, copies referenced objects from the old area to the new area (as discussed next), changes pointers in the page so that they point to the new area, writes a log entry containing the page's address, unpins the page when the log entry is physically in the log, and removes the protection for the page, so that the user can access it. Upon finding (in a scanned object) a pointer to an object in the old area that has not yet been copied, the system pins the not-yet-copied object's old page in volatile storage, copies the object to the new area, inserts a forwarding pointer in the object in the old area, writes a log entry, and (when the log entry is physically in the log) unpins the old page. The log entry contains the object's addresses in the old and new areas and the previous contents of the old copy (before it was overwritten with a forwarding pointer). The garbage collection does not require any extra synchronous input/output to achieve recoverability, but it can encounter the same page faults that any copying collector would encounter as it copies objects.

The garbage collection of O'Toole et al. [1993] executes as a reorganizer and uses fuzzy reorganization. Concurrently, users query, allocate, and update in the *old* area.

The collector scans reachable objects in the old area, copies them to the new area, and inserts forwarding pointers into a reserved field of the objects in the old area. After the copying, the collector uses the log (which records users' updates of the old area) to bring the new area up to date. We described use of the forwarding field for log application in Section 3.5.3. At the *end* of collection, the collector pauses the users, performs any recent updates in the new area, flips, updates users' roots (including any that are in processor registers), and resumes the users' execution. User accesses that are in progress at the time of flipping begin in the old area and continue in the new area. O'Toole et al. describe the implementation. The design can support multiple users, but the implementation supports just one user. Performance measurements compare online and offline collection. For a benchmark that models an engineering application that uses a database of parts, measurements show that the online strategy produces much shorter pauses for users, for example, 1 second vs. 15 seconds. The pause times for online collection are independent of the size of the heap, while the pause times for offline collection increase with the size of the heap. The time to perform a transaction exhibits similar behavior.

Some of the strategies flip at the start of garbage collection, and one of them flips at the end. There are trade-offs between the two times of flipping. Here are advantages of flipping at the end:

—The actions of users do not constrain the order in which the garbage collection copies pages. If we flip at the start, a user's attempt to access an unscanned page in the new area causes the collection to process that page sooner than it ordinarily would, to avoid delaying the user. This early processing might inhibit the collection's ability to place data in optimal locations in persistent storage.

—The implementation is simpler and more portable, since we need no support from the operating system for trapping users' accesses.

—If we are willing to restart garbage collection from its beginning after a failure, we need not log the actions of collection; we only need to log the actions of users. In the strategies that flip at the start, recoverability of the data involves objects in both the old and new areas, and it requires logging the individual actions of garbage collection.

—We never slow a user activity by performing copying as part of the activity.

Here are advantages of flipping at the start:

—During garbage collection, we perform each update only once. Flipping at the end requires performing an update in the first area and then (if the update applied to an object that had already been copied) performing it again in the second area, based on the log. Of course, the importance of this advantage is proportional to the fraction of time during which garbage collection is active.

—We avoid the need to read the log and/or to design the management of the log in a manner that supports more reading than usual. Ordinarily, we read the log only for recovery, but flipping at the end also requires reading the log for garbage collection.

—Some user activities might benefit from improved clustering *before* the completion of garbage collection.

4.4.6. Online Partitioned Garbage Collection by Copying for Persistent Storage. Next, we describe strategies for partitioned collection by copying. In addition, Lakhamraju et al. [2000; 2002] describe changing of references in partitioned reorganization by copying, as we discussed in Section 3.5.4; this technique has similarities to garbage collection.

4.4.6.1. Ferreira and Shapiro. In garbage collection for persistent distributed objects [Ferreira and Shapiro 1994b; 1994a], a design goal is to minimize communication due to collection between nodes. Another goal is preservation of the consistency semantics of the storage system. Here, a segment (set of virtual storage pages) contains objects, and a *bunch* (partition) is a logical group of segments. A partition can be replicated on several nodes. At any time, an object can have one writer or any number of readers. The owner of an object is the node that has (or most recently had) write access to the object. An owner pointer goes from an unowned replica of an object to the owner of the object. Most of the collection activity involves one partition at a time. When a replica of a partition exists at a node, the replica has associated structures that describe references to and from other partitions, owner pointers between replicas, and dependencies (which go from an owner to an unowned replica).

The garbage collection involves three tasks:

(1) The garbage collection of a replica of a partition is based on the strategy of O'Toole et al. [1993], which we described in Section 4.4.5. If a partition has replicas on several nodes, the replicas have separate garbage collections. The roots are program stacks, incoming references, and entering owner pointers. The collection scans and copies locally owned objects. It scans remotely owned objects without copying them; this lack of copying avoids inter-node synchronization. An object header with a forwarding pointer (in the old area of the copying strategy of O'Toole et al.) is deleted only when all references to the object have been updated with the new address. This first task includes updating the structures that describe outgoing pointers. The updates are later sent to other nodes for the next task, by adding the update information to the inter-node messages that would be sent on behalf of applications even without garbage collection.

(2) For a replica of a partition, the information from other partitions (sent from the previous task) is used to update the structures of incoming pointers, that is, roots. This updating will allow recognition that certain objects are no longer reachable from other nodes.

(3) The garbage collection for a node as a whole reclaims inter-partition cycles of garbage for local partitions; it uses the strategy of the first task. Here the roots do *not* include references that originate from the partition being collected. The garbage collection operates on the partitions that are currently in main storage at the node; it does not reclaim cycles that are partially on disk, to save input/output costs.

4.4.6.2. Moss et al. For a persistent object store, Moss et al. [1997] and Munro et al. [1999] describe garbage collection by copying (and then reclaiming the old copy) for one *car* (partition) at a time. Partitions are grouped into sets of partitions. The sets are ordered by time of creation of the sets, and within a set, partitions are ordered by time of creation of the partitions. The authors' description is generic with respect to the granularity of a partition, for example, a page, but a partition must be small enough to fit in a buffer. This garbage collection uses a copying technique, so we classify it as garbage collection by copying, but it does not copy a large area if partitions have a fine granularity. The idea is to copy reachable objects from a set of partitions to younger sets and then to discard the oldest set when it contains no reachable objects. The copying can compact the data. Sometimes the copying from one partition to another can reuse the same area on disk. Cycles of inter-partition garbage can be reclaimed if they can be moved into the same set of partitions. Each partition has a list of incoming references, and each set of partitions has a count of incoming references from outside the set. The authors' description is generic with respect to policies for when to create a partition and when to create a set of partitions.

Each unit of collection copies reachable objects from one partition, in this order:

(1) For each object that is locally reachable from a root, move the object to any partition in a younger set of partitions. An object *i* is *locally reachable* from a particular source (e.g., a root or another partition) if there is a chain of pointers from the source to *i* and each object in the chain (other than the source) is in the same partition as *i*.
(2) For each remaining object that is locally reachable from a younger set of partitions, move the object to any partition in a younger referring set of partitions.
(3) For each remaining object that is locally reachable from another partition in this set of partitions or that is locally reachable only from older sets of partitions, move the object to another partition in this set of partitions.
(4) Reclaim the partition; any objects that remain in this partition are garbage.

Finally, if a set's count of incoming references from outside the set is 0, reclaim this set.

The authors describe mechanisms that enhance performance by allowing updates of pointers and collection's movement of objects to modify only the partitions that are currently paged in. Deferred modifications to the partitions that are not currently paged in are recorded in lists (which could be implemented as B-trees or other structures); these modifications take place later, when the relevant partitions are paged in. Each list is global (i.e., not specific to one partition). One list records any added or deleted references to the partitions that are not paged in. When a partition is paged in, the changes for that partition move from the list to the partition. A second list records movement resulting from garbage collection. When a partition is paged in, the system can update its list of incoming references.

4.4.6.3. Comparison. The descriptions by Ferreira and Shapiro and Moss et al. both include (1) data structures that record references that cross partitions, (2) efficient communication of information between the partitions, and (3) reclamation of cycles of garbage that cross partitions. Ferreira and Shapiro emphasize behavior in a distributed, replicated environment, while Moss et al. emphasize copying between partitions and reclamation of the partitions that have become empty.

4.4.7. Comparison of Performance of Strategies for Garbage Collection. Next we survey work in comparison of performance of strategies for garbage collection for persistent storage. The strategies in the two comparisons that we survey have some overlap but not total overlap with the work that we surveyed above in strategies for online garbage collection for persistent storage. Of the strategies that the comparisons analyze, most but not all operate online.

Butler [1986; 1987] considers several strategies that were designed for garbage collection for volatile storage. She analyzes and compares the performance of the strategies when they operate on persistent objects. She analyzes these strategies:

(1) Mark and sweep offline.
(2) Copy and compact offline.
(3) Copy and compact incrementally [Baker 1978].
(4) Use reference counts. This strategy usually produces a short pause, but the lengths of pauses can vary, since the zeroing of counts can cascade.
(5) Use logged reference counts. Here, we log (but do not immediately apply) the changes to reference counts, and we process the log in small batches. This should reduce the variation in the lengths of pauses.
(6) Use deferred reference counts [Deutsch and Bobrow 1976].

Butler's modeling uses LISP-like lists that reside in a relational database. The cost of an activity is the number of pages that are read plus twice the number of pages that are written. The *average joint cost* per user access is (total cost of user accesses + cost of garbage collection) / (number of user accesses). This reflects both the amount of clustering (which determines the cost of user accesses) and the cost of garbage collection. The average joint costs for the six strategies are 1.3, 0.65, 0.53, 1.7, 1.8, and 1.4, respectively. The model uses 1,000,000 objects. Strategies 2 and 3, which are the only strategies that restore clustering, have the lowest costs. Also, strategies 3, 4, 5, and 6 have shorter pauses than strategies 1 and 2. Butler concludes that overall, strategy 3 (which has short pauses and the lowest average joint cost) is the best of the modeled strategies.

In another comparison, Yong et al. [1994] consider garbage collection that operates on persistent objects that reside at the server in a client-server environment. For that environment, the authors analyze and compare the performance of these strategies:

(1) Use deferred reference counts [Deutsch and Bobrow 1976]. The server maintains hash tables of identifiers of objects that have a reference count of zero or a reference count of two or more. During commitment of a transaction, a client sends to the server a list of the transaction's operations that will affect reference counts. This list might contain several entries for the same object. Periodically, the server sends to all the clients the table of object identifiers that have a count of zero, so that the clients can tell the server which of those objects are locally referenced but not yet committed. The server then reclaims the storage of the remaining zero-count objects.

(2) Use deferred reference counts, as in the previous strategy, but here the client summarizes the changes to each object before sending the list. This summary omits redundant entries for the same object and thus reduces both the communication and the server's processing.

(3) Mark without sweeping. This is the only offline strategy that the authors analyze; it halts all clients. The system integrates garbage collection with disk space management. All objects are initially on the free list; the marking phase removes the reachable objects from the free list. A phase for explicit sweeping is unnecessary, since the marking phase constructs a correct free list.

(4) Mark and sweep online. This strategy resembles the strategy of Dijkstra et al. [1978]. It uses a map of states (black, gray, or white) that resides in main storage at the server. A user's adding of a reference marks the referenced object as gray at the client. During commitment of a transaction, a client sends the gray list to the server, which adjusts its map accordingly. At the end of marking, the server asks the clients to send their unsent gray lists.

(5) Copy and compact online [Baker 1978]. Here, a pointer uses a logical object identifier, which a logical-to-physical mapping table maps into an address. When the system copies an object from the old area to the new, it updates the address in the mapping table; it need not update pointers to the object and leave a forwarding pointer in the old copy. Also, the mapping table obviates the trapping and redirection of a user's attempt to access an object in the old area. During commitment of a transaction, a client sends the relevant changes to the server. At the end of copying, the server asks the clients to send their local references to objects.

(6) Copy and compact online using partitions. Again, a pointer uses a logical object identifier, which a logical-to-physical mapping table maps into an address. Yong et al. physically divide storage into equal-sized partitions, although a partitioned strategy could also apply to logical partitioning, for example, grouping objects of the

same type. The system collects garbage in one partition at a time. It chooses the partition that has experienced the most updates, regardless of whether the updates involved pointers or data. During commitment of a transaction, a client sends to the server the relevant changes to references to objects in the chosen partition. This strategy does not reclaim cycles that cross partition boundaries, but the authors suggest merging of partitions if this becomes a serious problem. This strategy and the previous strategy copy objects and include logging.

Yong et al. use a simulation model of a client-server object-oriented DBMS with page-level locking. The simulation uses several user actions, for example, traversing all the objects in the database. As the size of the database increases from 2.4 to 51 megabytes, the amount of disk input/output for garbage collection is constant for strategy 6 (with a constant partition size), and it increases approximately linearly for strategies 1 through 5. Yong et al. also use the simulation to compare the strategies' abilities to improve users' performance via reclustering after users have replaced all the objects in the database. Strategies 5 and 6, which can recluster, lead to better hit rates than the other strategies, which do not recluster. Strategies 5 and 6 also lead to fewer pages of client-server data communication. The authors conclude that strategy 6 has many desirable characteristics. Thus both Butler (as discussed earlier) and Yong et al. recommend strategies that cluster.

Yong et al. also implemented strategies 2 and 4. For these strategies, the additional client activities for garbage collection (while no actual collection takes place at the server) cause only a small increase in response time, since the normal communication and disk input/output for transactions are more expensive. For strategy 4, if actual collection takes place, the response time increases approximately linearly (from about 57 to about 77 seconds) with the collection rate. The maximum rate described is 1500 objects marked per second, which the authors state "corresponds roughly to continuous execution of the collector." The authors conclude that "unless the garbage collector is run at a significant fraction of the time, it has minimal impact on the client applications."

4.4.8. Comparison of Performance of Policies to Choose a Partition. As we described earlier, garbage collection might use partitions. Here, for each activation of garbage collection, one requirement is a policy for choosing the partition whose garbage will be collected. A policy that chooses the partition with the most garbage would lead to the most efficient collection. Cook et al. [1994; 1998] investigate three such policies, which use the intuition that the values of overwritten pointers provide hints about the locations of garbage. They investigate a fourth policy, which uses the intuition that the partition with the most pages in the I/O buffer can be collected with the least I/O. They analyze the reasons for each policy's performance. They also analyze three more policies, which provide bounds for performance.

Here are the policies:

(1) For each partition, maintain a count of the pointers that are in this partition and have been updated since the previous garbage collection of this partition. Choose the partition with the largest count. The rationale is that the amount of garbage should increase with the amount of updating. This resembles the choice policy in strategy 6 of Yong et al. [1994] (described earlier). However, Yong et al. do not distinguish updates of pointers from updates of data. Cook et al. ignore updates of data, which cannot create garbage.

(2) For each partition, maintain a count of the pointers that referred to objects in this partition and have been updated since the previous garbage collection of this partition. Choose the partition with the largest count. The rationale is that the

amount of garbage should increase with the amount of updating of pointers into this partition. As part of the maintenance of the counts in this policy, when reclaiming an object that pointed into another partition, increment the count for that other partition. The previous policy involves counting for the partition in which a pointer resides; this policy involves counting for the partition that a pointer referenced.

(3) For each partition, maintain a weighted count of the pointers that referred to objects in this partition and have been updated since the previous garbage collection of this partition. The weight is proportional to the referenced object's proximity to the root. Choose the partition with the largest count. The rationale is that if most objects are referenced by only one other object, as in a tree, the amount of garbage caused by deleting a pointer should increase with the proximity to the root. This policy is a refinement of the previous policy. This policy is more expensive, since it involves time and space for maintaining and reading information on objects' proximities to the root.

(4) For each partition, count the pages that are currently in the I/O buffers. Choose the partition with the largest count. Here, the rationale is that the cost of collection should decrease if many pages are already in buffers.

(5) Choose a partition at random. This policy and the next two are intended to provide bounds for performance.

(6) Choose the partition that has the most garbage. This policy is optimal for each garbage collection but not necessarily optimal over all garbage collections. A simulation, but not a practical implementation, can perform this choice.

(7) Do not perform garbage collection. When more space is needed, add a partition.

The authors use simulation to compare the performance of the policies. They use 15–25 partitions. If, on an allocation, there is not enough free space, a new partition is added. The garbage collection, which copies and compacts, takes place after a specified number of pointers (50-400) have been overwritten. The number of collections varies from 15 to 90. The garbage collection itself does not require locking the entire database, but for simplicity, the *simulations* lock the entire database during collection. The simulations use databases with 3–38 megabytes of allocated objects (with 5 megabytes for the results that we will discuss).

Here are results that Cook et al. report: For the seven policies to choose a partition for collection, the mean amounts of page input/output (for the users plus the garbage collection), normalized by dividing by the amount of page input/output for policy 6, are 1.092, 1.011, 1.041, 1.099, 1.053, 1, and 1.073, respectively. The mean maximum amounts of storage required for the database (including reachable objects, uncollected garbage, and fragmented space), similarly normalized, are 1.26, 1.06, 1.18, 1.40, 1.20, 1, and 1.53. The mean efficiencies (numbers of kilobytes of garbage reclaimed divided by numbers of page input/outputs), similarly normalized, are 0.44, 0.82, 0.60, 0.24, 0.56, 1, and 0.

Policy 2 is more efficient than policies 1, 3, and 4. The authors state that reasons for the poor performance of policy 1 include (1) its lack of distinguishing between overwriting of pointers to preexisting objects (which can create garbage) and storage of new pointers during creation of objects (which cannot create garbage) and (2) its ranking of partitions by where the pointers reside instead of where the pointers pointed. Policy 3, which is more expensive than policy 2, performs poorly when many objects are each referenced by more than one other object, unlike the pattern for a tree. According to the authors, a reason for the inefficiency of policy 4 is that recently referenced (and thus paged-in) objects are more likely to be in use and thus *less* likely to be garbage;

also, the overwriting of a root pointer of a subtree can make the entire subtree garbage without paging in the subtree.

4.5. Cleaning (Reclamation of Space) in a Log-Structured File System

Our final category of maintenance is cleaning (reclamation of space) in a *log-structured file system* (*LFS*) [Ousterhout and Douglis 1989; Rosenblum and Ousterhout 1992], which we define below. The authors based their design of an LFS on these assumptions:

—Files can be cached in main storage. With a large cache, many of the requests to read files can find their data in the cache without reaccessing the disk.

—Therefore, most disk input/output will be for writing. Therefore, improving the performance of writing is important.

In an LFS, writing of files takes place by buffering a sequence of writes in the cache and then physically writing to a sequential structure on disk. For logical modification of a preexisting block of data, the system appends a new block of data as if it were writing a log, instead of overwriting the preexisting block. This appending has the effect of invalidating and deallocating the preexisting block. The system also appends blocks of a map that points to certain structures; the structure for a file contains the file's attributes and points to the file's data. The authors state that an LFS can replace many small writes and their seeks by one large write with fewer seeks, an LFS should reduce the requirements for seeks when writing, and an LFS speeds recovery from crashes. Tertiary storage might also use an LFS [Kohl et al. 1993; Ford and Myllymaki 1996].

During usage, the appending and its deallocation of old blocks can fragment the free space. A category of maintenance called *cleaning* restores the availability of adequate *contiguous* free space for future writing. This cleaning merges the *live* (still-valid) data from one or more *segments* (sets of contiguous blocks) into one or more other segments. The cleaning creates contiguous free space where the live data (and its dead neighbors) formerly resided. Of course, the cleaning consumes disk bandwidth and thus degrades system performance. Segments are large enough so that reading or writing an entire segment takes longer than seeking. An LFS does not necessarily serve in a database context, but cleaning has obvious similarities to garbage collection by copying, and cleaning can be viewed as a category of maintenance. Cleaning might execute as a reorganizer, or it might execute when storage is exhausted.

We will survey strategies for cleaning, including the original strategy of Rosenblum and Ousterhout. Much of the cited work also discusses other aspects of an LFS, but we discuss *only* issues related to cleaning. One concern that often arises in this work is *utilization*, which here means the fraction of an area of storage (e.g., a segment or a disk) that contains live data. The strategies use a variety of criteria for choosing segments to clean, criteria for grouping of blocks into segments, techniques for concurrency control, and details of cleaning. Many of the articles compare their performance with one or two other techniques, but we are unaware of a performance comparison that involves a large number of the strategies.

4.5.1. Rosenblum and Ousterhout. In the LFS of Rosenblum and Ousterhout [1992], cleaning operates on heavily fragmented segments. Each execution of cleaning chooses several segments for cleaning, reads the segments into main storage, identifies the live data, writes that data compactly (with no gaps of free space) into a smaller number of clean segments, and marks the chosen segments as clean. Cleaning starts when the number of clean segments drops below a threshold, and it suspends when the number rises past another threshold.

Rosenblum and Ousterhout use a simulation model to compare policies for choosing the segments to clean and for grouping the live data (e.g., sorting it by age) when writing it to new segments. The model uses a fixed number of 4-kilobyte files. Normal operations take place until all clean segments are used. Then the cleaning executes (offline, in the simulation), and it suspends when the number of clean segments reaches a threshold. The authors define the *write cost* of a policy as the average amount of time the disk is busy divided by the time that would be required for just writing the new data (with no seeking, rotational latency, or cleaning). Thus a write cost of 1 is perfect. In an ideal situation (high utilization and low write cost), most segments are nearly full and are not cleaned; a few are nearly empty and are cleaned.

During the simulation of the first policy, each file has an equal probability of being written by users. In this policy (called a *greedy* policy), the cleaning chooses the least-utilized segments (i.e., the ones that would yield the most free space), and it writes out data in the order that it appeared in the segments being cleaned. As the utilization of the disk rises from 0 toward 1, the write cost rises from 1 without bound. At high utilizations, more cleaning work is necessary.

In the simulation of the second policy, files have locality; 90% of user accesses are for 10% of the files. This policy (also called greedy) chooses segments as the first policy does, but when writing data (and arranging it into segments), the cleaning sorts blocks by their ages (the current time minus the times they were last written). The write cost is slightly higher than that of the first policy. The authors explain that the utilizations of rarely written segments tend to take a long time to drop to the threshold for cleaning, so those segments retain dead (but unavailable) blocks for a long time.

In the simulation of the third policy, the files again have locality. Here the cleaning chooses the segments that have the highest ratio of benefit to cost. The *benefit* is the amount of space that can be reclaimed, multiplied by the length of time that the space is predicted to stay free. The predicted time is the age of the youngest block in the segment; the authors assume that old data is likely to remain unchanged for a while. The *cost* is 1 + the utilization of the segment; in this formula, 1 corresponds to reading the segment, and the utilization corresponds to writing the live data into another segment. When writing data and arranging it into segments, the cleaning sorts blocks by their ages. The write cost is lower in this third policy than in the second policy. The simulation shows that the third policy cleans rarely written segments when they drop to about 75% utilization, so their dead blocks are reclaimed more quickly than in the second policy. The policy cleans frequently written segments when they drop to about 15% utilization; these segments lose most of their live data quickly, so cleaning can be delayed until most of their blocks have died.

The implementation uses the third policy above. For five production systems using an LFS, the write costs range from 1.2 to 1.6. The cleaning locks only a few files, but locks tend to propagate and produce convoys of waiting users [Hartman and Ousterhout 1995].

4.5.2. Singhal et al. For an index, log-structured techniques are applicable for replacing a modified block [Singhal et al. 1992]. Since replacing a block implies modifying the pointer that comes from its parent, all the ancestors of the block must also be replaced. These activities take place in the context of a persistent object store. Here the cleaning uses reference counts, not copying or compaction.

4.5.3. Seltzer et al. Another LFS implementation [Seltzer et al. 1993] avoids locking during most of the cleaning procedure. Instead, when the reorganizer is ready to write segments back to disk, the reorganizer checks whether any live data has recently died.

In scheduling, the system gives the reorganizer's writing priority over users' writing, to avoid exhausting free space. Also, normal writing by users suspends when the number of clean segments drops to 2, to ensure that the cleaning can operate. The cleaning uses the benefit/cost policy of Rosenblum and Ousterhout [1992]. The authors report performance measurements. For a software development benchmark, cleaning has negligible cost. For a transaction processing benchmark, however, 60–80% of all write traffic is for cleaning; the cleaning significantly affects system throughput.

4.5.4. de Jonge et al. A *logical disk* [de Jonge et al. 1993] manages a disk, not files. A type of logical-to-physical mapping table maps logical block addresses to physical disk addresses. A file system, at a higher level of abstraction, uses logical addresses, and movement of a data block does not require changing of references that point to the block. A file system can specify ordered lists of blocks, which the logical disk can use to guide physical clustering. A prototype implementation is log-structured. When a user activity finds that storage is exhausted, cleaning produces an empty segment, delaying the user activity. Cleaning can use information from the lists mentioned above to restore clustering. The authors also propose a reorganizer to perform the cleaning and improve the layout.

4.5.5. Hartman and Ousterhout. For an LFS-like file organization that stripes file data across multiple storage servers [Hartman and Ousterhout 1995], each client groups its new data for all files into a sequence, which the system stripes. The cleaning uses the client logs to calculate the utilization of each stripe. It chooses stripes to clean, using the benefit/cost analysis of Rosenblum and Ousterhout [1992]. The reorganizer avoids locking during most of the cleaning procedure, similarly to the strategy of Seltzer et al. [1993]; here the file manager resolves conflicts by assuring that the final pointer for a block reflects users' activities, not the reorganizer's activities. A conflict results in wasted work by the reorganizer, but the authors expect conflict to be rare. The reorganizer periodically checkpoints its state, to avoid complete reprocessing if a crash occurs.

4.5.6. McNutt. The analytic performance model of McNutt [1994] predicts the overhead of cleaning. He organizes blocks into segments by the number of times they have previously been cleaned. Within a group of segments, he cleans the least-utilized segments first. Each group has a threshold; a segment must have a utilization less than the threshold to be a candidate for cleaning. The model predicts that as the utilization of the disk increases toward 1, the number of cleaning moves grows without bound. For utilizations over 0.5, the number of moves per user write is approximately (0.5/(1-utilization))-1. He considers performance good if the utilization is less than 0.8.

4.5.7. Robinson and Franaszek. If cleaning moves only a subset of the data instead of all the data, the reduction in movement might improve performance. Robinson and Franaszek [1994] and Franaszek et al. [1994] consider algorithms that move only the subset that is estimated to have the greatest effect on fragmentation [Franaszek and Considine 1979]. They model the performance of a log structured file system and the performance improvement that results from moving only a subset. Their analytic model predicts how online cleaning affects available disk bandwidth. The model can apply to any criterion for choosing the subset of data to move. In the model, reading or the first writing after reading is random access and thus includes a seek, latency, and data transfer. Writing after prior writing omits the seek, and it also omits the latency if prior writing is still in progress. Parameters to the model include the arrival rate, the probability that an access is reading, the disk storage utilization, and the times for a seek, latency, and data transfer.

Robinson and Franaszek develop formulas for the mean service time and the number of blocks that the reorganizer can move during any idle period. They also calculate the rate of cleaning that is necessary to keep up with a rate of writing. The need for cleaning limits the maximum steady-state arrival rate of users' requests at the disk. The model predicts that the maximum permitted steady-state arrival rate decreases as the disk storage utilization increases, since the increased utilization requires more executions of cleaning. Reduction in the amount of movement required by cleaning increases the maximum permitted steady-state arrival rate.

4.5.8. Mogi and Kitsuregawa [1994]. A *RAID level 5* storage organization [Chen et al. 1994] stripes data and its parity bits across parallel disks. For this organization, Mogi and Kitsuregawa [1994] use two techniques for parity striping (one of which is LFS-based) to improve performance of block updates. To reduce the number of disk accesses for small writes (to recalculate the parity), Mogi and Kitsuregawa buffer several writes and then write the full stripe. They note that if all the blocks in a stripe are updated, then the calculation of parity need not reference the old data blocks and parity blocks on disk. Therefore, if a stripe contains n data blocks, and a new stripe is written for every n block updates, then this saving can take place. The new stripe must be written in a free area.

Mogi and Kitsuregawa describe two implementation techniques:

—In the LFS-based technique, when writes have filled the cache (the capacity of a segment), the system writes the segment. Cleaning reads and writes blocks. The physical locations of data blocks change after cleaning, but the striping linkage between blocks on different disks is fixed according to physical location.

—In the virtual striping (VS) technique, a table, not the physical location, determines which block on each disk is part of a particular stripe. To clean, the system collects n dirty blocks and creates one free stripe. It finds the live stripe (v) with the smallest number of live blocks. For each live block in v, the system modifies the table to exchange the block with a dirty block on the same disk but in another stripe. Stripe v then becomes free. The physical locations of data blocks are fixed, but the striping linkage between blocks on different disks changes after cleaning. Kitsuregawa and Mogi [1993] describe VS in more detail.

A simulation compares the performance of the two techniques. Each disk holds 318 megabytes and has 949 cylinders. There are 8 data disks and 1 parity disk. For the LFS, cleaning starts when the number of free segments shrinks below 3. For the VS, cleaning starts when the number of cylinders with free stripes shrinks below 9. In the results, VS has a lower cost of cleaning, since it accesses fewer blocks, but LFS has a lower cost of writing, since it involves less physical scattering. VS can tolerate a higher workload before response times increase without bound.

4.5.9. Mogi and Kitsuregawa [1995]. Follow-on work by Mogi and Kitsuregawa [1995] investigates the effect of access locality on LFS and VS. The authors use *hot block clustering*, which is clustering of blocks that are hot (referenced frequently). Each disk is divided into two contiguous regions (hot and cold). The strategy considers all updated blocks as hot, and it writes them into the hot area (during ordinary writing and during cleaning). The hot area gets more free space than the cold area; this practice reduces the cost of cleaning by increasing the expected time until the next cleaning. Placement in the hot area can also improve clustering. This strategy counts blocks that move from the cold area to the hot area between two successive cleanings of the hot area; cleaning of the hot area chooses that number of cold blocks from the hot area and moves them to the cold area. This migration is extra work for VS but not for LFS (where the cleaning performs block migrations). For choosing the blocks to migrate from the hot area, this strategy orders the blocks by the elapsed times since their most recent user accesses.

A simulation measures the effect of hot block clustering for LFS and VS. The write cache is part of a disk controller. In the simulation, there is no read cache; the authors want to examine the raw performance of the disk array. Half of the user requests read, and half write. The simulation assumes that 85% of blocks hold valid data; 15% are free. An access request, the block size, the sector size, and the striping unit are all 4 kilobytes. There are 8 disks and 949 cylinders per disk. For LFS, the system activates cleaning when the number of free segments shrinks below 6 for the hot area, 4 for the cold area, or 6 if the system is not using hot block clustering. For VS, the system activates cleaning when the number of cylinders that have free stripes shrinks below 6 for the hot area, 4 for the cold area, or 9 if the system is not using hot block clustering. If every disk has requests for user access or requests for reading by cleaning, the system ordinarily delays making a new request for cleaning; this policy prevents too much degradation by cleaning. However, the system unconditionally performs cleaning if only 1 free segment exists under LFS or fewer than 3 cylinders have free stripes under VS.

In the results of the simulation, as the load (I/O requests per second) grows, the average response time for reading increases without bound, of course. Hot block clustering reduces the response time, and it increases the load limit at which the response time becomes infinite. The authors state that hot block clustering improves performance because cleaning generates a larger free area and clustering reduces the movement of the disk head. For high locality (90% of accesses to 10% of the data), performance is best when the hot area is around 20% of the capacity. The authors vary parameters and show the results. For example, with lower locality (80% of accesses to 20% of the data), hot block clustering improves the performance of LFS (where migrations are combined with cleaning), but it improves the performance of VS (where migrations cause overhead) *only* at high loads.

4.5.10. Blackwell et al. The goal of Blackwell et al. [1995] is to reduce cleaning's degradation of users' performance. When the disk becomes full, on-demand cleaning takes place. When the user workload drops, a reorganizer cleans. Of course, reduction in on-demand cleaning can improve performance. The authors measured data from three file servers (each with .9 utilization) to obtain traces. They simulate an LFS for each server. During the simulation, there are many gaps, that is, periods of no disk activity by users. As might be expected, on a graph whose horizontal axis represents the length of a gap and whose vertical axis represents the number of gaps of each length, the distribution of gaps generally has a negative slope; that is, more gaps are short. Given that a gap is at least 2 seconds long, the probability that it is at least 4 seconds long is over .95. Cleaning one segment takes no more than 1.25 seconds. Using a heuristic of activating cleaning when 2 seconds of idleness have elapsed and ending the cleaning after a user request arrives, all cleaning is via the reorganizer on two servers, and 97% of cleaning is via the reorganizer on the other server, which is the most heavily loaded server. All three servers can service over 70% of all user requests with no queuing delays.

4.5.11. Wilkes et al. As we mentioned earlier, most strategies for cleaning copy the live data from the segments being cleaned and form new segments. In a description of a storage hierarchy, most of which does not deal specifically with an LFS, Wilkes et al. [1996] include the option of *hole-plugging* (copying data from other segments *into* the dead blocks in the segments being cleaned). This technique is appropriate if a segment has a high utilization. Traditional cleaning takes place for segments with low utilizations.

4.5.12. Matthews et al. It has been claimed that an LFS offers good performance if there are frequent small writes, the cache satisfies most of the reads, and there is sufficient idle time. It offers bad performance if there are random writes, high utilization, and little idle time. Matthews et al. [1997] investigate three techniques to improve the write performance of an LFS and one technique to improve the read performance. A simulation predicts the results of using the techniques. In the simulation, cleaning takes place only when there are no empty segments; thus the cleaning delays user activities. However, the techniques could also apply to an environment in which a reorganizer performs the cleaning.

The first technique to improve writing involves choosing the segment size by trading transfer efficiency with cleaning efficiency. An advantage of a large size is that the access time (average seek time + 1/2 the rotation time) is a small fraction of the time to transfer a segment. An advantage of a small size is increased variance (among the segments) in utilizations. With increased variance, the cleaning can choose sparser segments, and more segments are empty and need no cleaning. The simulation shows that for one measured workload, as the segment size increases, the total cost of writing (considering access time, data writing, and cleaning) decreases and then increases. Thus, the lowest total cost occurs at an intermediate size. Of course, disk characteristics affect the cost and the optimal size. For a random update workload, a high utilization, and no idle time, a small size (which minimizes the cleaning cost) can have the lowest total cost.

The second technique deals with modifying the cleaning policy to adapt to changes in the utilization. Traditional cleaning reads the live data from several segments that it cleans, and it writes one segment; it is good at low utilizations. *Hole-plugging* [Wilkes et al. 1996] reads data from one segment, and it writes that data *into* dead blocks in several segments; it is good at high utilizations. For a random-update workload, the break-even point (where traditional cleaning and hole-plugging have the same cost) is 85% utilization. For another workload, traditional cleaning is better through 99% utilization. When cleaning is needed, an adaptive cleaning policy (1) estimates the ratio of cleaning time to free space reclaimed for both traditional cleaning and hole-plugging and (2) chooses the cheaper mechanism.

The third technique to improve writing (by reducing the cost of cleaning) is to use cached data during cleaning. If a segment is completely cached, cleaning can omit the disk read. A policy for selecting segments to clean can take into account the fact that a segment is cached; the simulation shows that for a large enough cache, this policy can reduce costs.

The technique to improve reading is reorganization of data to match read patterns. This technique can help if read patterns do not match write patterns. In this technique, the system builds a graph of block accesses to record sequential accessing, and it prunes the edges in least-recently-used order. The reorganization tries to arrange blocks according to the graph. For a software engineering workload, this technique can reduce the disk time.

4.5.13. Menon and Stockmeyer. The work of Menon and Stockmeyer [1998] deals with criteria for selecting segments to clean. This work focuses on activities in a disk controller, but the activities could alternatively take place in a file system. The authors propose that segments that have recently been filled by users' writing should not become candidates for cleaning until their ages have passed a threshold. Among the segments that have passed the threshold, the authors select the segments that will yield the most free space. The rationale is that the utilization of segments that have reached the threshold probably will not decrease significantly in the near future. A controller performs cleaning as a background task. A segment filled by users' writing gets a timestamp when it is filled. A segment filled by cleaning acquires the largest timestamp of the segments that contributed to it. The age of a segment is defined as the current time minus the timestamp. A threshold near 0 resembles the greedy selection criterion of Rosenblum and Ousterhout [1992]. For a very large threshold, too many low-utilization segments are ineligible for cleaning. In analysis and in a simulation, as the age threshold increases, the average utilization of the cleaned segments decreases and then increases. When this average utilization is low, the cleaning is doing a good job of selecting segments. A simulation predicts that for a range of thresholds, the policy of Menon and Stockmeyer can have a lower average utilization of the cleaned segments than either of the selection criteria of Rosenblum and Ousterhout.

4.5.14. Chee et al. The work of Chee et al. [1998] analyzes statistics to find patterns for reading (sequences of references), which imply guidance for reorganization. The system invokes reorganization when access times exceed a threshold or the number of free segments drops below a threshold. In performance measurements, for sequential access (e.g., in printing a file), the authors report that reorganization can speed disk reads by up to 71%, assuming that reorganization is offline.

4.5.15. Wang et al. The goal of Wang et al. [1999] is to minimize the latency of small synchronous writes to disks. Their technique involves selecting (as the location for writing) a free sector that is near the current location of the disk head. Their analysis assumes that the disk system can contain enough intelligence to make the selection, thus obviating some communication between the disk system and the main processor. In the results of a model of performance, within a disk cylinder, the time to reach the first free sector increases as utilization increases.

Suppose that during idle periods, the disk system can compact data and generate empty tracks. Here, when the utilization of the track used for writing reaches a threshold, the system switches to the next empty track. A performance model shows that as the threshold (fraction of free sectors reserved per track) increases from 0 toward 1, the time to reach a free sector decreases, then is fairly flat, and then increases without bound.

The authors describe *virtual logging*, which uses a type of logical-to-physical mapping table for blocks on the disk. This technique can be used to have the effect of an LFS. A performance evaluation supports the advantages of this technique for writing small files.

4.5.16. Wang and Hu [2001]. For disks whose outer zones (sets of cylinders) have more sectors per track (and therefore a faster transfer rate) than inner zones, Wang and Hu [2001] reorganize data during cleaning and also during idle periods. They put frequently accessed log segments into faster zones and rarely accessed segments into slower zones. A file's *active ratio* is the value of a formula that measures how frequently the file is referenced. A segment's active ratio is the average of its files' active ratios. Each zone's entry in a zone table tracks vacant segments and the minimum and maximum of the segments' active ratios.

The authors describe three policies for reorganization:

Write-optimized: Put migrated segments into slots starting from the slowest zone, in ascending order of their active ratios. Thus, the open area near the outer edge, which is fastest, is available for new writes. If a zone is full, check whether the zone contains a more active segment. If it does, move the more active segment outward. If it does not, place the migrated segment outward.

Read-optimized: Put migrated segments into slots starting at the fastest zone, in descending order of their active ratios. Thus, the area near the outer edge, which is fastest, contains the most frequently accessed data. If a zone is full, check whether the zone contains a less active segment. If it does, move the less active segment inward. If it does not, place the migrated segment inward.

Balanced: The area near the outer edge, which is fastest, is available for large new writes, and the secondary fast zones contain the most frequently accessed data.

To compare the performance of the three policies with the performance of a regular LFS, a simulation uses a disk segment size of 512 KB, an I/O buffer of 8 MB, and a disk of 3.8 GB or 40 GB. The simulation uses several traces, with 200-95,000 MB read, 200-17,000 MB written, and a range of file sizes up to more than 1MB. In the results, all cases of policies 1 and 3 and almost all cases of policy 2 have an improvement in write time, sometimes up to 25%. All cases of policies 2 and 3 and almost all cases of policy 1 have an improvement in read time, sometimes up to 21%. All cases have an improvement in throughput, sometimes up to 32%.

4.5.17. Wang and Hu [2002]. To reduce cleaning overhead, Wang and Hu [2002; 2003] separate active (frequently invalidated, short-lived) data and inactive (rarely invalidated, long-lived) data in two or more segment buffers in main storage, before writing to disk (into different disk segments). The inactive segments should rarely need cleaning, and the active segments should contain less data to copy during cleaning. In main storage buffers, each block has a count of recent accesses, to help the system decide whether the data is active. Cleaning executes when the system is idle or when the disk utilization exceeds a threshold. As with Matthews et al. [1997], cleaning can choose between traditional cleaning and hole-plugging.

A simulation, using real traces and synthetic traces, compares the performance to that of a traditional LFS. It uses two disks (1G and 9G), a 64 MB buffer cache, a disk segment size of 256 KB, and 256 KB per segment buffer. The overall write cost includes both the overhead of cleaning and the actual writing. In the results, the authors' technique always has a lower overall write cost than a traditional LFS, sometimes up to 52% lower. The difference increases as disk utilization increases, because cleaning becomes more important at high utilizations. Varying the number of segment buffers from 2 to 4 has relatively little effect. The read performance is comparable to that of a traditional LFS.

4.5.18. Jambor et al. The cleaning of Jambor et al. [2007] is similar to that of Seltzer et al. [1993]. The authors discuss the implementation of their LFS. For a 10-GB partition, measurements involve disk utilizations of .25, .5, and .75. In an experiment, 8 GB of data are rewritten; the rewritten records are 64 KB. Records have either a uniform probability of overwriting or a skew in which 10% of the records are overwritten with a 90% probability and 90% are overwritten with a 10% probability. With 10–90, the time to complete the experiment with .75 utilization is almost double the time with .25 utilization. With a uniform probability, the time is over 7 times as large.

4.5.19. Chiang and Huang. The technique of Chiang and Huang [2007] deals with placement of the live data. To reduce seek times, they place frequently updated data near the center of a disk and rarely updated data near the edges. They partition the disk space into regions. Update of a data block causes the block's promotion to an inner region. When cleaning a segment, they demote the live data to an outer region. A trace-driven simulation measures the number of blocks copied during cleaning. Of course, the number rises as disk utilization increases. Depending on the utilization, number of disks, and choice of trace, the technique sometimes produces an increase in the number of blocks copied but usually produces a decrease, which, according to the authors, ranges up to 99.9%. Typically, the decrease is greatest as the utilization increases to 90%. Increasing the number of disk regions (from 2 to 3 in their simulation) also decreases the number of blocks copied. The technique also usually decreases disk seek time for the set of requests in the trace, sometimes by 45.45%. In a prototype implementation, the technique again decreases the number of blocks copied, especially when more than 20% of the data is frequently updated.

5. STRATEGIES FOR ONLINE PHYSICAL REDEFINITION

The maintenance activities that we discussed above do not change physical definitions, but now we begin a discussion of changes in physical definitions. Our survey of work in physical redefinition includes construction of indexes, conversion between B^+-trees and linear hash files, and redefinition (e.g., splitting) of partitions.

5.1. Construction of Indexes

During construction of indexes for a table, a traditional DBMS allows users to query but not to update the table. We will discuss algorithms that construct indexes concurrently with users' updates and queries of the table. Also, Stonebraker et al. [1988, p. 329] mention construction of indexes via fuzzy reorganization.

Table I compares the algorithms that we will discuss. The headings of the columns at the right (e.g., the vertical "Sto") represent algorithms. The two classes of differences in the table are (1) the area to save users' updates during construction and (2) the main steps to construct the index and process the saved updates, if any. An asterisk in a cell indicates that the algorithm uses the area or the set of steps.

5.1.1. Stonebraker. Section 3.1 defined a *partial index* [Stonebraker 1989], which references only the records that satisfy a specified condition. A reorganizer for constructing an index (partial or full) can scan the data in page order, locking only a series of one or more pages at a time and constructing the index incrementally from those pages. The reorganizer can mark the index as a partial index whose condition includes "RID < X," where X is the first RID in the next data page [Stonebraker 1989, p. 7]. Thus users' updates of already-scanned pages also update the index. See the column marked "Sto" in Table I. Instead of using a physical condition like "RID < X," an alternative is to use a logical condition like "age < 30." The indexed columns (e.g., salary) need not be the columns in the condition (e.g., age).

If a partial index has a logical condition, preexistence of a different index on the columns in the condition can, of course, greatly speed the construction of the partial index [Olson 1993, p. 11]. Use of a partial index can speed a query whose condition is at least as restrictive as the condition of the index. Also, if a partial index is constructed incrementally (e.g., first for age = 29, then for age = 28, etc.), the DBMS can make the partial index available incrementally for processing relevant queries; this is an advantage of partial indexes. Olson [1993] discusses construction of partial indexes and their use in queries. When a partial index already exists, Olson lets the user extend the index

Table I. Comparison of Algorithms for Online Construction of Indexes

	Sto	LXB	LXS	LXM	IXB	IXM	NSF	SF	Mai	Ron	Fri
(1) *Area to Save Users' Updates During Construction*:											
In sorted list (i.e., leaves of temporary index)					*	*					
In sequential list		*	*	*							
In sequential list; saved only if reorganizer has already scanned relevant data page								*			
In log									*		*
In index being constructed (i.e., no separate saving)	*						*			*	
(2) *Main Steps to Construct Index and Process Saved Updates, if Any*:											
Incrementally, scan data and build index from data.	*									*	
Incrementally, scan data and build index from data; process log entries.									*		
Scan all data; sort data; build index from data; process log entries.											*
Scan all data; sort data; build index from data; process unsorted updates.		*						*			
Scan all data; sort data; build index from data; sort updates; process sorted updates.			*								
Scan all data; sort data; build index from data; process presorted updates.					*						
Scan all data; sort data; sort updates; build index from merger of data and sorted updates.				*							
Scan all data; sort data; build index from merger of data and presorted updates.						*					
Scan all data; sort data; build index from data.							*				

by specifying a condition for extension. The effective condition for the index will then be the disjunction of the existing condition and the extension condition. In his implementation, users have query-only access to the table during construction or extension of a partial index, but a possible alternative implementation (allowing updates) is to use one of the strategies that we discuss for online construction.

The rest of this section discusses only construction of full indexes.

5.1.2. Srinivasan and Carey. We will describe a set of ten algorithms for construction of an index by a reorganizer [Srinivasan and Carey 1991a; Srinivasan 1992]. During some steps of each algorithm, the DBMS uses an update list (instead of the index being constructed) to save users' attempted insertions and deletions in the index. Each algorithm includes the first three operations described here, and some of the algorithms also include the fourth and/or fifth:

(1) The reorganizer scans the data. Specifically, for each page in the table, it acquires a read-lock, appends pairs of key values and RIDs to a heap file, and unlocks.

(2) The reorganizer sorts the data in the heap file, by key and RID. If the file contains two or more entries for a pair of key value and RID, the sorting keeps the last one and discards the others.

(3) The reorganizer builds the index, based on one or two sources. The first source is always the sorted heap file. Some of the ten algorithms also use a second source

(the update list mentioned above). This building includes (1) eliminating duplicate pairs of key values and RIDs and (2) discarding pairs of insertions and deletions that correspond to each other. The reorganizer produces the leaf pages of the index and then creates the higher levels of the hierarchy.

(4) The reorganizer sorts the entries that are in the update list. If the list contains two or more entries for a pair of key value and RID, the sorting keeps the last one and discards the others.

(5) The reorganizer processes the entries that are in the update list. Specifically, for each entry that is a deletion, the reorganizer deletes from the index if the index contains the pair of key value and RID. For an insertion, it inserts in the index if the index does *not* already contain the pair.

Offline construction would read-lock the table, perform the first three operations above (without page-level locking), make the index visible to user transactions, and unlock the table.

Table I includes five of the ten *online* algorithms of Srinivasan and Carey (the columns marked "LXB," "LXS," "LXM," "IXB," and "IXM," which are abbreviations of the names of the algorithms). The simplest algorithm, LXB, has these phases:

(1) The reorganizer scans data in the table, sorts the data, and builds the index from that one source. Each user transaction that updates the data in a way that ordinarily would update the index (if it existed) instead write-locks the update list, appends a set of information (key value, RID, and flag for insertion or deletion) to the update list, and unlocks the update list. The reorganizer write-locks the update list at the end of this phase.

(2) The reorganizer processes the entries that are in the update list. Each user transaction that would update the index waits for completion of this phase. The reorganizer makes the index available to users at the end of this phase. User transactions can always ascertain the current phase of the reorganizer.

All ten algorithms scan and sort the data in Phase 1.

The ten algorithms of Srinivasan and Carey differ from each other in three ways. The first difference involves when to allow users' updates:

—During some steps of the algorithms (scanning data, sorting data, and building an index from just the data), all the algorithms allow users' attempted updates. During other steps (sorting updates, processing updates, and building an index from the merger of data and the update list), the five algorithms in Table I forbid users' additional updates, as we described above for LXB.

—However, five other algorithms (LCB, LCS, LCM, ICB, and ICM) extend the five in Table I to achieve more concurrency by allowing such additional updates (in Phase 2) and including an extra Phase (3) to process those updates. "X" in the middle of an algorithm's name means "exclusive," and "C" means "concurrent." In the C algorithms, after locking the update list in Phase 1, the reorganizer copies, empties, and unlocks the update list. This unlocking permits users' additional updates in Phase 2; users resume their appending to the update list. The reorganizer uses the *copy* of the update list for processing updates or for the second source in building the index. The C algorithms lock the update list again at the end of Phase 2. These algorithms have a third phase, during which the reorganizer processes the update list (which now contains the updates made during Phase 2), and users insert and delete directly in the index. During Phase 3, for coordination between users and the reorganizer, users must sometimes place special insertion or deletion entries in the index, the reorganizer's sorting of the update list handles duplicates in a special way, and the reorganizer's processing of updates must consider the users' special entries.

The second difference among the algorithms deals with the list that saves users' updates. Some algorithms (LXB, LXS, LXM, LCB, LCS, and LCM) save users' updates in a sequential list, and some (IXB, IXM, ICB, and ICM) use a sorted list (i.e., leaves of a temporary index). "L" at the start of an algorithm's name means "list" (sequential), and "I" means "index" (sorted). In the I algorithms, for saving a user's update, if the sorted list already contains an entry for the pair of key value and RID, the system deletes the existing entry before inserting the new entry.

The third difference involves the processing of the saved updates. LXB builds the index from one source in Phase 1, as we described earlier. However, some algorithms (LXM, IXM, LCM, and ICM) build the index in Phase 2 (using the saved updates as a second source in the building) after locking the update list in Phase 1 and (for the L algorithms) sorting the update list in Phase 2. Some algorithms (LXS and LCS) sort and process the saved updates in Phase 2 after the building in Phase 1. Some algorithms (LXB, IXB, LCB, and ICB) process the saved updates in Phase 2 (after the building in Phase 1) without sorting. "M" at the end of an algorithm's name means "merge" (two sources), "S" means "sort," and "B" means "basic" (no sorting).

Srinivasan [1992] and Srinivasan and Carey [1992b] discuss a simulation model and use it to analyze the performance of their ten algorithms. It is a closed queuing model that predicts the time to construct the index and the loss in the potential transaction processing in the database system. The definition of *loss* is the construction time multiplied by the decrease in user throughput during construction. The model uses 100,000 rows in the table initially, 400,000 possible values for the indexed key, and 1 or 8 disks. Each user operation inserts, deletes, or queries on one data page.

The model's results indicate that for a multi-disk system, the loss with offline construction almost always exceeds the loss with online construction. For each of the three ways in which the ten algorithms differ, one choice typically shortens the construction time. Here are those three choices, reasons they shorten the construction time, and the choices' effects on update loss:

(1) Forbidding updates after Phase 1 (in the X algorithms) shortens the construction time, since it reduces both contention from users and the required processing of updates by the reorganizer. However, there is usually a trade-off between construction time and update loss. Forbidding such updates decreases the update throughput during construction, of course; this usually yields more loss than in the C algorithms, despite the shorter construction time.

(2) Saving users' updates in a sorted list (temporary index, in the I algorithms) instead of a sequential list (in the L algorithms) sometimes shortens the construction time. It spares the reorganizer from sorting those updates or accessing the permanent index randomly to process unsorted updates. Again, there is sometimes a trade-off. A user's insertions in a temporary index have less page locality and more overhead than appends to a sequential list, thus decreasing the update throughput. This sometimes yields more loss despite the sometimes-shorter construction time.

(3) Including saved updates in the building of the permanent index (in the M algorithms), instead of building the index without the updates and then processing the updates (in the B and S algorithms), allows the reorganizer to access each permanent index leaf page just once in Phases 1 and 2. Similarly, S is faster than B, since it reduces accesses to index pages. The reduction in construction time in M almost always yields less loss.

Overall, LCM usually has the lowest loss.

5.1.3. Mohan and Narang. Next we discuss two other algorithms for online construction of an index [Mohan and Narang 1992], shown in Table I as "NSF" and "SF." The reorganizer scans the data for key values and RIDs, sorts the keys, and builds the index while periodically checkpointing the highest inserted key. The main difference between the two algorithms lies in their handling of user transactions' insertions and deletions in the index during construction. The algorithms include logging and are restartable. They use a restartable sorting algorithm, which obviates rescanning the data if a failure occurs. The sorting includes checkpointing of the sorted streams.

Mohan and Narang define two problems that each algorithm solves:

—In the *duplicate-key-insert* problem, the reorganizer and a user transaction that inserts data might try to insert the same pair of key value and RID in the index. This problem can arise because the latch on a data page is not held during insertion in the index, to avoid deadlock.

—In the *delete-key* problem, a race condition can cause the reorganizer's activities to straddle a user's activities. Specifically, the reorganizer extracts a key from a data page, then a user transaction deletes the key from the data page (and does not find the key in the index), and finally the reorganizer erroneously inserts the key in the index.

In one algorithm (called *NSF* for "no side file"), users insert and delete directly in the index; the algorithm does not use a separate area to save the insertions and deletions. NSF tolerates interference by users. Both the reorganizer and user transactions write log entries for activities in the index. When the duplicate-key-insert problem occurs, whichever process is second (the reorganizer or a user) refrains from inserting the duplicate. Even if a user transaction refrains from inserting a key, that transaction still writes a log entry for the omitted insertion. This logging enables correctness of possible later rollback of the transaction if a problem occurs. To solve the delete-key problem, a user transaction that tries unsuccessfully to find a key to delete in the index instead inserts a pseudo-deleted key and writes a log entry. The reorganizer will not later insert a key there.

NSF starts by quiescing updates of the index, to assure the absence of uncommitted updates. NSF then creates a descriptor for the index, making it visible to users for insertion and deletion. It then releases the quiesce. Without the quiescing, a rollback of an uncommitted transaction that inserted a record would not delete the record's key from the index. The index is not yet available to users for queries, although it is possible to optimize the later building phase by making the index incrementally available for queries (up through the most recently completed key value). NSF then extracts the keys and RIDs from the data (possibly including uncommitted records), and it sorts the keys.

After this sorting, NSF inserts the keys in the index (while latching index pages) and periodically commits the insertions. It can periodically checkpoint the highest inserted key, to avoid restarting from the beginning if a failure occurs. User transactions log their insertions, even if they refrain from actually inserting due to the duplicate-key-insert problem. The reorganizer logs its insertions only when the insertions succeed. The behavior of a user transaction that deletes is more complex:

—If the index already contains the key, the user writes a log entry and modifies the key to become pseudo-deleted. The user transaction cannot detect whether the reorganizer has already extracted the key from the data page, so the system needs a pseudo-deletion instead of a physical deletion. The presence of the pseudo-deleted key will prevent the reorganizer from inserting the key.

—If the index does not yet contain the key, the user writes a log entry (for deletion) and inserts a pseudo-deleted key in the index. The log entry will assure reinsertion in the index in the event of later rollback of this user transaction. The initial quiescing assures that the deleting transaction will write the log entry.

After inserting all the keys, NSF makes the index available for queries. Optionally, a reorganizer physically deletes any pseudo-deleted keys whose deletions are known to be committed.

In the other algorithm (called *SF* for "side file"), users append insertions and deletions to a side file if the index is visible, that is, if the reorganizer has already scanned the relevant data page. Otherwise, they ignore the index being constructed. User transactions do not interfere with index construction. The reorganizer and user transactions write log entries for the side file, not for the index. The duplicate-key-insert problem does not occur, since a user transaction appends an insertion to the side file only if the reorganizer has already scanned the relevant data page, which did not yet contain the inserted key. The delete-key problem does not occur, since the reorganizer processes the side file (which includes the deletion) *after* the initial construction of the index (which includes insertion of the key). When a user transaction appends a deletion to the side file, the log entry includes the number of visible indexes, to assure correctness of any later rollback.

SF creates a descriptor for the index, scans data, sorts the keys, builds the index from the data (while periodically checkpointing), and performs the side file's insertions and deletions in the index (while periodically checkpointing).

SF is more efficient than NSF, involves less logging, avoids pseudo-deletions, probably yields a more clustered index (since user transactions do not interfere), and need not initially quiesce updates. However, NSF needs no side file and has less sophisticated logging and undo requirements for user transactions.

Mohan [1993a, p. 447] notes that a commercial system allows updates during construction of indexes. The unpublished algorithm is similar to SF and preceded the work of Mohan and Narang.

5.1.4. Maier et al. Another strategy for online construction of indexes [Maier et al. 1997; Troisi 1994; Englert 1994] is similar to fuzzy reorganization. See the column marked "Mai" in Table I. The strategy uses these phases:

(1) Save the current position in the log. For each row of the table, read the row and create an index entry, while users can query and update the table.
(2) Read the relevant log entries, starting from the saved position. For each entry that should change the index, create one or two log entries for the corresponding index modification, and perform "redo" operations to apply the created log entry or entries to the index. Users still can query and update the table.
(3) Lock the table. Read and process the log entries as in the previous step, starting after the previous step's last log entry.
(4) Make the index visible to users and unlock the table.

5.1.5. Ronström. Section 6.2 describes the strategy of Ronström [2000] for redefinition. The author concentrates on examples of logical redefinition but also includes an example of physical redefinition, namely construction of an index (the "Ron" column in Table I). This approach creates the index and scans the table to construct index entries incrementally. In addition, a temporary trigger (defined on the underlying table) assures that for any row that the index already includes, users' updates of the row also update the index. When the index is complete, the DBMS makes the index visible to user transactions.

5.1.6. Friske et al. Another strategy is similar to fuzzy reorganization [Friske et al. 2007]. It records the log position, scans the data, extracts the index keys, pairs the keys with RIDs, sorts by key and RID, builds the index, reads the log entries, and applies the log entries to the index. See the column marked "Fri" in Table I. Some variations are possible in processing the log.

5.1.7. Comparison. An area of differences among the algorithms is in updating of the index to reflect users' updates of the data:

—Some algorithms use a list of users' updates: Srinivasan and Carey save updates in a list, which is sorted or sequential. They read the list and apply the updates to the index. Some of their algorithms also use the list as one of the sources for initial construction of the index. The SF algorithm of Mohan and Narang uses a sequential list (which they read and apply), but only for data pages that the reorganizer has already scanned. Maier et al. and Friske et al. read and apply the log instead of maintaining a specialized list.

—Some algorithms perform users' updates directly in the index: Stonebraker performs updates in the index, but only if the reorganizer has already scanned and processed the data page that the user updates, that is, only if the RID of the updated record satisfies the condition of the partial index. The NSF algorithm of Mohan and Narang performs updates in the index. Ronström creates a trigger that updates the index if (1) the reorganizer has already read and processed the row that the user updates or (2) the update inserts a row in the table.

5.2. Conversion Between B$^+$-Trees and Linear Hash Files

An index, whose construction we described above, might take the form of a B$^+$-tree [Comer 1979]. Another access structure is a linear hash file [Litwin 1980]. For queries that use the indexed or hashed key, a B$^+$-tree can process range queries more efficiently, and a linear hash file may process exact-match queries more efficiently. A change in patterns of usage (from range to exact match, or vice versa) can justify a conversion from one access structure to the other. Omiecinski [1988] describes online conversion from a B$^+$-tree to a linear hash file, and Omiecinski [1989; 1996, pp. 28–30] revises and extends this work, adding online conversion in the opposite direction. For each direction of conversion, it is desired to reuse the space of the original structure, thus requiring a total amount of space (during reorganization) that only slightly exceeds the space of the original structure.

For conversion from a B$^+$-tree, a reorganizer scans the leaves of the B$^+$-tree. For each leaf page, these steps take place:

(1) Set an exclusive lock on the leaf page.
(2) Read the page.
(3) Set a shared lock on the state variables.
(4) Calculate the page addresses in the linear hash file for all the keys in the leaf page.
(5) Insert the data into the linear hash file. For this insertion, the system tests whether the number of distinct page addresses (for insertion in the hash file) exceeds a threshold fraction (e.g., 0.5) of the number of keys. The probability that this condition is true increases as the reorganization proceeds, since a large hash file has a smaller probability that keys will hash to the same page. If the condition is true, the insertion takes place one key at a time, with exclusive locking. This individual insertion locks the state variables briefly. If the condition is false, all the keys belonging to the

same hash file page are inserted in one operation, with exclusive locking. This group insertion saves I/O and processing time but locks the state variables for a long time.

(6) If an overflow has occurred in the hash file, then perform a split.

(7) Release the locks.

For conversion from a linear hash file, the reorganizer proceeds in physical address sequence in the hash file. It reads and locks one primary page (and all its overflow pages, if any) as a unit. For each primary page, it sets an exclusive lock, reads the pages, sorts the keys, inserts the keys in the B^+-tree in order (locking as necessary), and unlocks.

The author assumes that all user accesses are via the indexed or hashed key. During conversion in either direction, the system directs each user access into the hash file or the B^+-tree according to whether the reorganizer has already reached the key value that the access uses.

An analytic performance model and a simulation model predict the breakpoint during reorganization. The *breakpoint* is the time (or number of pages to be converted) when the throughput during online reorganization grows to match what the throughput would be for the original access structure without reorganization. After that point, the throughput during reorganization is better.

For conversion from a B^+-tree, the analytic model's parameters include costs of various activities, characteristics of transactions, and the reorganization *unit* (the number of pages that must be converted in one iteration before the unit is accessible via the hash file). The unit might be large if splitting in the B^+-tree has caused the physical sequence of data pages to differ from the key sequence. The analytic model ignores locking. User transactions use direct access via a single key.

For conversion from a linear hash file, the analytic model's parameters include costs of various activities and characteristics of transactions. Again, the model ignores locking.

The simulation model is a closed queuing model, with queues for input/output, processing, and lock blockage. The multiprogramming level is 10. For conversion from a B^+-tree to a linear hash file, the database size is 143 pages, the page size is 10 records, and the B^+-tree height is 3. For conversion from a linear hash file to a B^+-tree, the database size is 128 pages, and the hash page size is 11 records. User transactions use a mixture of exact match queries, exact match updates, and (for conversion from a hash file to a B^+-tree) range queries.

In the results of the simulation, for conversion from a B^+-tree, the breakpoint grows as the reorganization unit increases. As an example, when the unit is 15 pages (approximately 10% of the database size), the breakpoint is approximately 39% of the time required to complete online reorganization. The analytic and simulation models predict similar breakpoints, differing by about 2% on average. As the unit increases, the maximum number of delayed transactions and the time at which that maximum occurs both increase, but the time to complete online reorganization (which is about 7.7 times the offline reorganization time) does not vary much.

For conversion from a linear hash file, the breakpoint occurs when 26–42% of the hash file has been converted. The breakpoint shrinks as the probability that a query is a range query grows or as the range size grows. This is no surprise, since a B^+-tree supports range queries efficiently. However, when the probability of a range query grows very large (beyond 0.5), the breakpoint shrinkage is less pronounced, perhaps because some queries must access records from both the B^+-tree and the linear hash file. The analytic and simulation models predict similar breakpoints, differing by 4% on average.

Omiecinski considers the results to support the feasibility of this online reorganization.

5.3. Redefinition of Partitions

Our final category of physical redefinition is redefinition of the partitions of a table.

Pong [1990, p. 9] mentions the splitting of a partition into two partitions while allowing concurrent queries of the partition.

Troisi [1994; 1996], Englert [1994], and Maier et al. [1997] describe such splitting while also allowing concurrent updates. In this fuzzy reorganization, the system creates the new partition, records the log's current position, copies the second part (e.g., half) of the original partition into the new partition, applies the log to the new partition (all while allowing updates of the original partition), locks the original partition, applies any remaining log entries to the new partition, modifies the table's descriptors for routing of users' accesses into partitions, and unlocks. The system then physically deletes the copied data from the original partition, concurrently with updates. Partition splitting is recoverable. Recompilation can be necessary for relevant query access plans. In this DBMS, each partition includes information about the ranges of all the partitions. Therefore, after the final application of the log, the system must briefly lock the entire table to modify the range information.

The same strategy of fuzzy reorganization can shift the boundary (ranges of key values) between two adjacent partitions or can move a partition from one disk area to another.

In another strategy, when adding an initially empty partition (e.g., for the forthcoming month's sales), the new partition might be available immediately, with no reorganization of instances needed Friske [2004a].

6. STRATEGIES FOR ONLINE LOGICAL REDEFINITION

Next we move from changes in the physical definition to changes in the logical definition. Logical redefinition may, of course, affect not just data instances (i.e., in persistent storage), but also the physical definition, view definitions, objects' methods, and application programs [Sockut 1985]. However, strategies that we describe below can reduce the impact. Surveys [Roddick 1995; Ram and Shankaranarayanan 2003], an annotated bibliography [Roddick 1992; 1994], and an online bibliography [Rahm and Bernstein 2006] summarize much of the work on logical redefinition.

For some types of logical redefinition, of course, views can shield preexisting application programs from the changes (e.g., Chamberlin et al. [1975] and Sockut [1985, p. 17]). The system provides an unchanged set of interfaces to views while modifying the internal definitions of the views to accommodate the changes. Similarly, several strategies in an object-oriented DBMS can sometimes shield applications from changes:

—Operations in a changed schema can return the same results that those operations returned in the original schema [Osborn 1989].

—We combine methods with a path through the schema graph [Liu et al. 1994b]. The path and the interfaces to the methods omit the irrelevant subset of the schema; thus they hide changes in that subset.

—We change the implementation of an object class without changing the interface to the class [Schwarz and Shoens 1994].

A view can also provide the appearance of a new definition (to applications that need that new definition) without changing the schema [Bertino 1992; Tresch and Scholl 1993; Liu et al. 1994a; Ra and Rundensteiner 1995, 1997; Bellahsene 1996].

Many of the strategies that we discuss below include interpretation of one version of a schema in terms of another version. We can consider such interpretation to be a generalization of the interpretation in ordinary support for views; we map operations and data between two definitions [Sockut 1985, pp. 17–18; Bratsberg 1992; 1993].

This article uses *version* broadly to mean any variation from the original schema. Some authors use *schema versioning* more specifically to include a requirement that users can access data via any version of the schema, including an old version. Such authors use *schema evolution* to denote changing the schema without requiring the ability to use old versions of the schema [Roddick 1995, p. 384].

We will discuss (1) interpretation and incremental reorganization during users' activities, (2) use of a reorganizer, and (3) some additional topics.

6.1. Interpretation and Incremental Reorganization during Users' Activities

Here we discuss strategies for logical redefinition via interpretation and/or incremental reorganization during users' activities. These strategies allow execution of an application program whose schema version differs from the schema version that the accessed data instances currently use. When the database designer specifies redefinition, the DBMS records the change and need not immediately perform any physical change to preexisting data instances.

Gerritsen and Morgan [1976] describe such a strategy. The description uses the CODASYL data model. Each instance indicates the schema version that was in effect when the instance was written. Similarly, each application program indicates its schema version. Any number of versions can exist. A database designer uses a redefinition language to specify changes, e.g., moving a data item between record types. When a program accesses an instance, the DBMS performs these steps:

(1) If the instance uses any one of the old versions of the schema, translate the instance to the most recently defined version. For a query, this translation occurs only in volatile storage; the instance continues to use the old version in persistent storage. For an update, the DBMS will store each updated instance in persistent storage (later in the transaction) using the most recently defined version [Gerritsen 1994]; this achieves incremental reorganization in persistent storage.
(2) If the program uses any one of the old versions, present the instance (to the program) in the form that the old version uses. However, if a program tries to read or modify something that no longer exists, abort the program.

Tan and Katayama [1990] describe a coarser granularity of incremental reorganization. When a program accesses an object instance that uses any one of the old versions, the DBMS reorganizes *all* instances of that object class.

Many strategies have been developed for logical redefinition in object-oriented DBMSs. Surveys [Roddick 1995; Ram and Shankaranarayanan 2003], an annotated bibliography [Roddick 1992; 1994], and an online bibliography [Rahm and Bernstein 2006] cover many of the strategies. Other strategies and related discussions (not in the surveys and bibliographies) include those of Ahlsén et al. [1983; 1984], Zdonik [1986], Versant Object Technology [1991, p. 4.89], Morsi et al. [1991; 1992], Morsi [1992], Bratsberg [1993], Itasca Systems [1993], Odberg [1994], Ferrandina et al. [1994], Fontana and Dennebouy [1995], Peters and Özsu [1995], Malhotra and Munroe [1997], Goralwalla et al. [1998], and probably others. The strategies use interpretation (e.g., by extra actions associated with methods) and/or incremental reorganization during users' activities. The granularity of redefinition is the entire schema in some strategies and an individual object class in some other strategies. Several of the descriptions list repertoires of operations for logical redefinition. For example, Banerjee et al. [1987] list a repertoire of changes to an attribute (e.g., adding, deleting, and renaming an attribute and changing its default value), to a method, to the inheritance, and to a class (adding, deleting, and renaming). Clamen [1994] shows how integration of heterogeneous databases has similarities to support for logical redefinition; the system translates accesses of instances of one definition into accesses of instances of another definition.

Since the surveys and bibliographies cover so many strategies, we do not survey all the strategies. Instead, we have chosen the strategy of Clamen [1994] as an example to illustrate techniques for interpretation and incremental reorganization in object-oriented DBMSs. The rest of this section discusses this strategy in more detail. We will discuss the logical behavior of the strategy and then discuss possible techniques to implement the logical behavior. Choosing the techniques involves performance trade-offs. Clamen's set of implementation techniques includes techniques used in the other strategies; thus this strategy is a good illustrative example.

In the strategy of Clamen, each object instance is logically an instance of each version of its object class and thus logically contains all the attributes of each version. When a user modifies attributes of one version, the modification logically propagates immediately to any relevant attributes of every other version. As we will discuss shortly, an implementation need not physically store all the attributes or physically propagate modifications immediately.

To illustrate the possible relationships between the attributes in two versions of an object class, Clamen uses an example of two versions of an object class called "Undergraduate." The first version contains attributes for the name, class (e.g., "freshman"), and degree program of an undergraduate. The second version contains attributes for the name, identification number, class year (expected year of graduation), and faculty advisor of an undergraduate.

Relationships exist between most attributes in the two versions. The two attributes for name match. The identification number in the second version does not correspond to any attributes in the first version. The four possible values of class correspond to the year of the next June and the three succeeding years. A specific advisor deals with only certain specific degree programs. A relationship can involve a *set* of attributes, as in Cartesian vs. polar coordinates [Bratsberg 1992, p. 429], but for simplicity, this example uses only relationships between single attributes.

The strategy includes the handling of four types of relationships between attributes of two versions. For each of the four types, Table II shows the name of the type of relationship, the definition of that type, example attributes from the Undergraduate object class (with "(1)" and "(2)" appended to denote the versions), and the logical behavior for propagating a user's modification from one version to the other. For example, class and class year are *derived* from each other. The value of either attribute can be calculated from the other. For example, modifying a class to be "senior" propagates to modifying a class year to be the year of the next June.

The relationships between the example attributes from Undergraduate are symmetric; e.g., class is derived from class year, and class year is derived from class. More generally, however, relationships need not be symmetric. For example, it is possible for an attribute *x* to be *derived* from an attribute *y* while *y* is *dependent* on *x*.

For each version and its relationships to other versions, the database designer can choose an implementation for each of several aspects of the logical redefinition. One aspect involves addition and initialization of attributes of a new version. When specifying a new version of an object class, the designer has these options:

—For each instance of this object class, add and initialize the attributes of the new version of the class immediately.

—For each instance of this object class, add and initialize the attributes of the new version only when a user later references that instance via that version. This incremental

Table II. Types of Relationships Between Attributes of Two Versions

Name of Type of Relationship	Definition	Example Attributes	Logical Behavior for Propagation
Shared	Attribute has same meaning in the two versions.	name (1) and name (2)	Copy the value.
Independent	There is no relationship.	identification number (2)	Do nothing.
Derived	Value of attribute can be calculated from attributes in other version.	class (1) and class year (2)	Apply a function; e.g., class = "senior" propagates to class year = year of the next June.
Dependent	Value of attribute is affected by, but cannot always be calculated from, attributes in other version.	degree program (1) and faculty advisor (2)	Apply a function; e.g., degree program = "physics" propagates to current faculty advisor if that advisor deals with physics; null (unknown) otherwise.

reorganization saves space. It saves a large amount of time during specification of the version, but it requires time during some later executions.

A second aspect involves propagation of modifications from the attributes of one version to the attributes of another (if the versions occupy separate spaces). The designer has these options:

—Modification of one version physically propagates immediately to the other version.

—A flag is associated with each version in each instance to indicate whether the attributes of that version are up to date with respect to the other version. When a user modifies one version, the DBMS sets the flag for the other version to indicate "out of date" but does not propagate the values. When a user reads the instance via the second version, the DBMS propagates the values from the first version to the second and resets the flag. This choice speeds writing of any version but slows reading of a version that is out of date.

A third aspect involves the space for shared attributes. Typically, the designer chooses for shared attributes to occupy the same space; this saves both space and time.

A fourth aspect involves the space for mutually derived attributes. The designer can choose between occupancy of the same space (thus saving space, of course) and occupancy of separate spaces. If the attributes occupy the same space, another decision is whether that space stores the first version, the second version, or (for each instance) the version that was more recently written (thus requiring a version identification associated with the space). If the space always stores the first version (or always stores the second), reading or writing of that version is fast, and reading or writing of the other version requires interpretation. If the space stores the version that was more recently written, any writing is fast, reading of the more recently stored version is fast, and reading of the less recently stored version requires interpretation. If the attributes occupy separate spaces, reading and writing are slow if they require immediate propagation; they are fast if they do not require it.

6.2. Use of a Reorganizer

Now we consider strategies that include a reorganizer combined with activities that take place during users' activities.

6.2.1. Wilson. In reviewing approaches and costs for logical redefinition, Wilson [1979] discusses several techniques to avoid long periods of unavailability: a fine granularity, fuzzy reorganization, and interpretation with incremental reorganization during users' activities.

Wilson [1980] also discusses an architecture that uses a reorganizer to perform logical redefinition in place. The discussion uses the CODASYL data model. In this architecture, a transition schema (which includes the old and new schemas and their interrelationships) describes the database during reorganization. For example, CODASYL set types in the transition schema can relate record types in the old schema and record types in the new schema, if the values of the new record instances are derived from the values of the old record instances. The database designer can specify procedures to derive the values. For any application program, the DBMS presents only the old schema or only the new schema. The DBMS maps the application's view onto the transition schema. The author notes that if a mapping from old to new is not reversible, then while some applications are still using the old schema, applications that use the new schema should be forbidden from updating information whose mapping is not reversible. Statements in a control language specify control structures (e.g., iteration over all the record instances of a set instance) to direct the steps of reorganization, for example, creating new record instances.

6.2.2. Ronström. In the approach of Ronström [2000], a reorganizer and users' activities both perform some reorganization work. The strategy can apply to either logical or physical redefinition. Preexisting transactions use the old schema, and new transactions use the new schema. Some cases of redefinition do not require quiescing. The author suggests splitting long-running changes into many small transactions, to avoid long reorganization transactions.

The author defines a *soft* change in a schema as a change that permits concurrent execution of old transactions (using the old schema) and new transactions (using the new schema). A *hard* change does not permit such concurrent execution. An example of a hard change is replacing a pair of an old column for hours worked and an old column for payment per hour by a single new column for total payment; a new transaction's update of the total payment cannot map uniquely into a pair of hours worked and payment per hour.

The approach involves five phases. Many details of the first two phases depend on the specific type of redefinition, and Ronström provides the details for several examples of redefinition. Here are the generic phases:

(1) The reorganizer adds any required columns, tables, indexes, triggers, and foreign keys. Some of the added triggers and foreign keys ensure that the instances of the new columns, tables, and indexes will be up to date at the end of Phase 2 (including effects of old transactions' updates of instances of the old schema). The added columns allow null values, although they forbid null values starting in Phase 5 if the new schema forbids null values. If the new schema omits an old column, table, index, or foreign key, then the actual deletion takes place in Phase 5.

(2) The reorganizer scans the old data and performs any actions that are necessary to make the old and new data consistent, for example, copying rows from an old table to a new table.

(3) The DBA can execute test transactions on the new schema. If the changes are hard, the DBA must first quiesce transactions.

(4) If the tests are successful, new transactions can start, and they use the new schema. Old transactions continue on the old schema.

(5) After all old transactions have finished, the reorganizer can remove any triggers and columns that the reorganizer used, and it removes any old columns, tables, indexes, and foreign keys that the new schema omits.

The author discusses considerations for recovery.

The author's first example of redefinition is adding columns that are derived from old columns (and, if desired, dropping old columns). Phase 1 includes adding the columns, adding a trigger to update the new columns to reflect updates of the old ones, and adding a trigger for the reverse of the first trigger (if there is a reverse mapping and thus the change is soft). Phase 2 scans the table and updates the new columns, based on the old.

An example of *physical* redefinition is construction of an index. In Phase 1, the reorganizer creates the index (which is not yet available to user transactions) and creates (in the underlying table) a column whose presence in a row indicates that the index includes an entry for that row. A trigger assures that an update of a row also updates the index if the index already includes an entry for that row or if the update inserts a row in the table. Phase 2 waits for completion of all user transactions that began before creation of the trigger. The reorganizer then scans the table and constructs entries in the index for rows that do not yet have the column. In Phase 3, after successful testing, the reorganizer makes the index available to user transactions.

Other examples of redefinition include splitting a table (by rows or by columns) and merging tables. Here, triggers assure that the new tables and indexes reflect updates of the original tables and that the original tables and indexes reflect updates of the new tables. In addition, if old foreign keys refer to the original tables, then triggers assure that new foreign keys eventually refer correctly to the new tables. Ronström discusses details of these examples.

6.2.3. Friske. Another online facility [Friske 2004b] applies to some types of logical redefinition (changing data types and lengths) and some types of physical redefinition (e.g., appending columns to indexes and changing indexes between clustering and nonclustering). Redefinition takes effect immediately in the database catalog, but the DBMS does not immediately change data instances in storage. For querying, the DBMS materializes the new format interpretively, and for updating, the DBMS stores updated instances in the new format. During the next maintenance, which may be online (via a reorganizer) or offline, the DBMS stores all instances in the new format.

6.2.4. Oracle. A document [Oracle 2005] shows several examples and describes the DBA's steps (e.g., procedure calls) for several specific categories of reorganization via a reorganizer.

6.2.5. Løland and Hvasshovd. In the work of Løland and Hvasshovd [2006], a reorganizer uses fuzzy reorganization (with iterations of log processing) to perform either of two specific examples of logical redefinition:

—A new table is the full outer join of two old tables.

—Two new tables are the result of splitting one old table.

For each example, the authors describe handling of the various logged operations.

For comparing the performance during reorganization with the performance before reorganization, a prototype implementation keeps all data in main storage. There are 50,000 records in one joined table and 20,000 in the other, and there are 50,000 records in the split table. In the tests, the degradation of user throughput was 2% to 6% during loading and 2% to 11% during log processing. The degradation of user response time was 5% to 30% during loading. Reducing the priority of the reorganizer reduces user degradation but increases the reorganization time.

6.2.6. Comparison. Both Wilson [1980] and Ronström [2000] discuss (1) access via an old schema or a new schema, (2) temporary mechanisms to keep the two versions consistent (set types and procedures; triggers), (3) forbidding updates via a new schema that are concurrent with access via an old schema if a mapping is not reversible, and (4) techniques for initial construction of the new structures (a control language and its steps; scanning and its actions). Friske [2004b] discusses use of the new version during users' activities and eventual change via a reorganizer. Løland and Hvasshovd [2006] discuss fuzzy reorganization for specific examples of logical redefinition. Also, the technique of Subramaniam and Loaiza [2005], described in Section 3.3.3, deals with logical or physical redefinition.

6.3. Additional Topics

We also discuss work that deals with (A) incorporation of time-variation into definitions and (B) addition of columns.

6.3.1. Time-variation. Several studies have investigated the incorporation of the time of validity and/or the time of recording for data instances in a database. A time of validity indicates when a data instance became valid.

Roddick [1991] extends such work by incorporating such time-variation into the database definition. For example, the database catalog can indicate when each attribute became valid, that is, the time when the related redefinition logically took effect. Similarly, any constraint can have a period of validity. Queries can include specification of a time; the query processing uses the database definition that was in effect at that time. If a query requests an attribute that was not valid at that time, the query processing inserts a null value into the result of the query.

In other work in time-varying definitions, Galante et al. [2005] define a model and language in which schema versions have times of validity. The authors define rules for times of validity and for relationships among versions. Queries can specify times and intervals.

6.3.2. Addition of Columns. One straightforward example of interpretive logical redefinition is the appending of columns to a table (if the columns have default values, e.g., null). When a user queries an instance that lacks a new column, the DBMS presents the default. Several DBMSs, including commercial DBMSs, can perform this interpretive redefinition (e.g., Sockut and Goldberg [1979, pp. 384–388]), and the ALTER TABLE statement of SQL [Amer. Natl. Standards Institute 1992] can specify this redefinition.

Takahashi and Takahashi [1990] and Takahashi [1990] describe a strategy for achieving the effect of adding columns to tables without physically changing preexisting instances. For each such table, a special new table contains a row for each nonnull instance of any added column. Each row in the special table contains the key value of a row in the original table, a new column name, the data type and length of that new column, and the value of the instance of the column. Queries and updates operate on a virtual table that the DBMS composes from the original table and the special table. Logically, the composition contains these steps:

(1) Based on the special table, construct an intermediate table containing just the projected columns (and the key) and just the selected rows.
(2) Perform an outer join of the intermediate table and the original table. This produces null values for new column instances that do not appear in the special table.

The cost of this composition can motivate eventual reorganization of the instances in the original table. The reorganization consists of adding columns, copying the column values from the special table, and deleting the rows from the special table.

7. SUMMARY

We have discussed many types of reorganization, and in practice, any database management system can need some type. To avoid taking a highly available or very large database offline for reorganization, a solution is to reorganize online (concurrently with usage, incrementally during users' activities, or interpretively). We have identified requirements for online reorganization. The design of online reorganization involves trade-offs in resolving certain issues, including use of partitions, the process that reorganizes, reorganization by copying, use of differential files, references to data that has moved, performance, and activation of maintenance.

Work has taken place in online maintenance, physical redefinition, and logical redefinition. Much of this work has appeared in research contexts, and much has appeared in commercial contexts. Commercial work in online reorganization and related areas includes the topics of reorganization of a partition, reorganizers, interpretation and incremental reorganization during users' activities, reorganization in place, reorganization by copying, fuzzy dumping, fuzzy reorganization, handling of compression, use of mapping tables (and other reference-correcting techniques), pausing between steps of reorganization, activation of maintenance, restoration of clustering, reorganization of an index, purging of old data (which we described under garbage collection), cleaning in a log-structured file system, construction of indexes, redefinition of partitions (splitting, shifting, or moving), logical redefinition during users' activities, and logical redefinition via a reorganizer.

We have indicated that some topics involve complexity or need more research. These topics include users' and reorganization's tracking of the status of each other's actions on data, transitions between accessing old and new copies in a copying strategy, coordination of concurrent accesses of the old and new copies if the two copies of the area being reorganized sometimes contain valid copies of the same instances, the desirability of giving users preferential allocation of resources, and activation of online maintenance. Of course, we also expect to see more research on other topics within online reorganization.

Not all databases are highly available or very large. However, as dependence on computers and the amount of information in databases both grow, highly available or very large databases will continue to become more common and more important in the world economy. Therefore, the importance of online reorganization will continue to grow.

8. GLOSSARY

Here we briefly restate the definitions of some of the terms and acronyms that we use frequently and a few that we do *not* use frequently but some other authors do. In many cases, we cite sections that discuss the terms in more detail. Some of the terms should be familiar to readers who have studied database management, but we have included those terms for the benefit of readers with little knowledge of database management. Many but not all of our terms are in standard use by other authors. For example, we are unaware of any standard terms for the concepts of *redefinition* or *reorganizer*.

Checkpoint: To record persistently, for example, as a basis for restart.

Cleaning: In a *log-structured file system*, maintenance that merges the live data from one or more *segments* into one or more other segments, thus reclaiming contiguous free space where the data and its deallocated neighbors formerly resided. Section 4.5.

Clustering: The practice of storing instances near each other if they meet certain criteria. Section 4.1.

Declustering: The practice of interleaving data among parallel disks, assigning *logical* units of data (e.g., a row of a table) per disk. A DBMS is involved, and it might use a round-robin approach or a key. If it uses a key, it might hash the key, or each disk might get a range of key values. Section 4.3. Contrast with *striping*. Some authors may use different definitions.

Eager reorganization: Used by some authors to denote either reorganization by a *reorganizer* or offline reorganization. Contrast with *lazy reorganization*. Section 3.2.1.

Flip: To exchange the roles of the original area and the new area, in garbage collection by copying. Section 4.4.

Fuzzy reorganization: Online reorganization by unloading the old (original) copy of the area being reorganized (which users query and update), reloading into a new copy in reorganized form, applying the log to bring the new copy up to date to reflect users' recent updates of the old copy, and changing the logical-to-physical mapping to direct users' accesses into the new copy. Section 3.3.3.

Incremental reorganization during users' activities: Online reorganization that occurs as part of users' activities, typically just for data instances that the activities reference. Section 3.2.1. Contrast with use of a *reorganizer*.

Lazy reorganization: Used by some authors to denote *incremental reorganization during users' activities*. Contrast with *eager reorganization*. Section 3.2.1.

LFS (*log-structured file system*): A file system in which all writing to disk takes place in a sequential structure. To logically modify a preexisting block of data, we append a new block of data, instead of overwriting the preexisting block. This appending has the effect of invalidating and deallocating the preexisting block. Section 4.5.

Maintenance: Reorganization that restores the physical arrangement of data instances without changing the database definition. Sections 1.1 and 4. Contrast with *redefinition*.

Mapping table: A table that maps identifiers for records or for other objects. A *logical-to-physical mapping table* maps from logical identifiers, which do not change during reorganization, to current physical identifiers, which change. An implementation might use *RIDs* as the physical identifiers. Typically, the mapping table contains an entry for each record, even if reorganization has not moved the record. A *physical-to-physical mapping table* maps from old physical identifiers to new physical identifiers. Typically, it contains an entry for a record only if reorganization has moved the record. Section 3.5.1.

Mutator: Used by many authors to denote a user process (which might make some storage unreachable), in the context of a garbage-collected system. Section 4.4.

Performance degradation: An increase in users' response time and/or a decrease in throughput. Contrast with *structural degradation*, which can be a cause of performance degradation. Interference by reorganization can be another cause.

Quiescing of specified activities of users: Blocking of new activities and waiting for preexisting activities to complete.

Redefinition: Reorganization that changes the database definition. Section 1.1. Redefinition can be physical (Section 5) or logical (Section 6). Contrast with *maintenance*.

Reference count: A count of the references to an object. Section 4.4.1.

Reorganization: Changing some aspect of the logical and/or physical arrangement of a database.

Reorganization by copying: Reorganization that can unload *all* the data instances from the area being reorganized, reload all the instances in a new copy of the area in newly allocated storage, and switch users' accesses from the old copy to the new. Section 3.3. Contrast with *reorganization in place*.

Reorganization in place: Reorganization that leaves the set of data instances in the same general area, without necessarily unloading and reloading *all* of them. The reorganization might move groups of one or more data instances between pages, but some instances might not move. Section 3.3. Contrast with *reorganization by copying*.

Reorganizer: A process that executes in the background to sweep through an area and perform online reorganization. Contrast use of a reorganizer with *incremental reorganization during users' activities*. Section 3.2.1.

RID (*record identifier*): A number that identifies a record in a database. Most authors use this term to mean a physical-level identifier.

Scan: To read a range of data sequentially.

Segment: A set of contiguous blocks, in a *log-structured file system*. Section 4.5.

Striping: The practice of interleaving data among parallel disks, assigning *physical* units of data (e.g., *n* bytes) per disk. There is not necessarily a DBMS involved. A round-robin approach is used. Section 4.3. Contrast with *declustering*. Some authors may use different definitions.

Structural degradation: Physical rearrangement of data instances in a way that causes performance degradation or depletion of storage allocation. Contrast with *performance degradation*.

Utilization: (1) In queuing analysis, typically the fraction of time that a resource (e.g., a disk) is busy providing service; it is a measure for the user workload. (2) In analysis of some types of reorganization, the fraction of an area of storage that contains data.

View: A user-visible representation of part of a database, for example, a virtual table that is defined as the result of a query on other views and/or on tables.

ACKNOWLEDGMENTS

A suggestion from Robert Goldberg led to the first author's early work in online reorganization. We thank these people for discussing issues in reorganization: Kiran Achyutuni, S. Ashwin, Tom Beavin, Brian Bennett, Chuck Bonner, Kurt Brown, Mike Carey, Chung Chang, Stewart Clamen, Dick Crus, Greg Davoll, David Detlefs, Susanne Englert, Paolo Ferreira, Peter Franaszek, Mike Franklin, Craig Friske, John Garth, Rob Gerritsen, Rob Goldring, Kevin Gougherty, Don Haderle, Claudia Ho, Andy Johnson, Roger King, Masaru Kitsuregawa, Elliot Kolodner, Rich Lewis, Bruce Lindsay, Rafael Lins, Miron Livny, Guy Lohman, Bob Lyle, Tom Majithia, Jeanna Neefe Matthews, Mark McAuliffe, Bill McIver, Claus Mikkelsen, Roger Miller, Frank Mix, C. Mohan, Magdi Morsi, Eliot Moss, Inderpal Narang, Sham Navathe, Tom Niermeyer, Nels Olson, Ed Omiecinski, Jim O'Toole, John Ousterhout, Franco Putzolu, Gary Ricard, John Robinson, John Roddick, Jim Ruddy, Betty Salzberg, Chris Saul, Dave Schwartz, Peter Schwarz, Margo Seltzer, S. Seshadri, Dennis Shasha, Kalpana Shyam, Gary Smith, Lars Söderlund, V. Srinivasan, Larry Stockmeyer, Junichi Takahashi, Jim Teng, Jim Troisi, Patrick Valduriez, Radek Vingralek, Julie Watts, Steve Watts, Gerhard Weikum, Roy Weise, Paul Wilson, Tom Wilson, Gordon Yamamoto, Danny Zilio, and Ben Zorn. We thank editors-in-chief Gul Agha and Chris Hankin, associate editor Stan Zdonik, and the referees for their help and suggestions.

REFERENCES

ABDULLAHI, S. E. AND RINGWOOD, G. A. 1998. Garbage collecting the Internet: A survey of distributed garbage collection. *ACM Comput. Surv. 30*, 3, 330–373.

ACHYUTUNI, K. J., OMIECINSKI, E., AND NAVATHE, S. B. 1992. HISET: A generalized technique for efficient concurrent reorganization. Tech. rep. GIT-CC-92/57, College of Computing, Georgia Institute of Technology, Atlanta.

ACHYUTUNI, K. J., OMIECINSKI, E., AND NAVATHE, S. B. 1993. The impact of data placement strategies on reorganization costs in parallel disk systems. Tech. rep. GIT-CC-93/58, College of Computing, Georgia Institute of Technology, Atlanta.

ACHYUTUNI, K. J., OMIECINSKI, E., AND NAVATHE, S. B. 1996. Two techniques for on-line index modification in shared nothing parallel databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 125–136. (*SIGMOD Record 25*, 2).

AHLSÉN, M., BJÖRNERSTEDT, A., BRITTS, S., HULTÉN, C., AND SÖDERLUND, L. 1983. Making type changes transparent. In *Proceedings of the IEEE Workshop on Languages for Automation*. 110–117.

AHLSÉN, M., BJÖRNERSTEDT, A., BRITTS, S., HULTÉN, C., AND SÖDERLUND, L. 1984. An architecture for object management in OIS. *ACM Trans. Office Info. Syst. 2*, 3, 173–196. (This includes a summary of the work of Ahlsén et al. [1983]).

ALMES, G. T. 1980. Garbage collection in an object-oriented system. Ph.D. dissertation, Department of Computer Science, Carnegie Mellon University, Pittsburgh. (Also Tech. rep. CMU-CS-80-128.)

Amer. Natl. Standards Institute. 1992. *Database Language SQL*. Amer. National Standards Institute, New York. X3.135-1992.

Ammann, P., Jajodia, S., and Mavuluri, P. 1995. On-the-fly reading of entire databases. *IEEE Trans. Knowl. Data Engin. 7*, 5, 834–838.

Amsaleg, L., Franklin, M. J., and Gruber, O. 1995. Efficient incremental garbage collection for client-server object database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers, San Francisco, 42–53. (For a revision with more details, see Amsaleg et al. [1999].)

Amsaleg, L., Franklin, M. J., and Gruber, O. 1999. Garbage collection for a client-server persistent object store. *ACM Trans. Computer Syst. 17*, 3, 153–201.

Appel, A. W., Ellis, J. R., and Li, K. 1988. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*. ACM, New York, 11–20. (*SIGPLAN Notices 23*, 7, ACM).

Ashwin, S., Roy, P., Seshadri, S., Silberschatz, A., and Sudarshan, S. 1997. Garbage collection in object-oriented databases using transactional cyclic reference counting. In *Proceedings of the 23rd International Conference on Very Large Data Bases*. Morgan Kaufmann, San Francisco, 366–375. (For a revision with more details, see Roy et al. [1998].)

Azatchi, H., Levanoni, Y., Paz, H., and Petrank, E. 2003. An on-the-fly mark and sweep garbage collector based on sliding views. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*. ACM, New York, 269–281. (*SIGPLAN Notices 38*, 11).

Bacon, D. F. 2007. Real-time garbage collection. *Queue 5*, 1, 40–49. ACM, New York.

Bacon, D. F., Cheng, P., and Rajan, V. T. 2004. A unified theory of garbage collection. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*. ACM, New York, 50–68. (*SIGPLAN Notices 39*, 10).

Bahaa-El-Din, W. H., Bastani, F. B., and Teng, J.-E. 1989. Performance analysis of periodic and concurrent data structure maintenance strategies for network servers. *IEEE Trans. Soft. Engin. 15*, 12, 1526–1536.

Baker, Jr., H. G. 1978. List processing in real-time on a serial computer. *Comm. ACM 21*, 4, 280–294.

Banerjee, J., Kim, W., Kim, H.-J., and Korth, H. F. 1987. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD 1987 Annual Conference*. ACM, New York, 311–322. (*SIGMOD Record 16*, 3).

Banzon, A. T., Friske, C. A., Garth, J. M., Ng, K. C., Ruddy, J. A., and Vizconde, B. B. 2006. Method, apparatus and program storage device for managing buffers during online reorganization. U. S. patent application 2006/0173922.

Barabash, K., Ben-Yitzhak, O., Goft, I., Kolodner, E. K., Leikehman, V., Ossia, Y., Owshanko, A., and Petrank, E. 2005. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Prog. Lang. Syst. 27*, 6, 1097–1146.

Barabash, K., Ossia, Y., and Petrank, E. 2003. Mostly concurrent garbage collection revisited. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*. ACM, New York, 255–268. (*SIGPLAN Notices 38*, 11). (For a revision, see Barabash et al. [2005].)

Bartlett, J. F. 1988. Compacting garbage collection with ambiguous roots. Tech. rep. 88/2, Western Research Lab., Digital Equipment Corp., Palo Alto.

Baru, C. and Zilio, D. C. 1993. Data reorganization in parallel database systems. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems* IEEE-CS. IEEE Computer Society Press, Los Alamitos, 102–107. (For later, more extensive analysis, see Zilio [1998].)

Bastani, F. B., Chen, I.-R., and Hilal, W. 1991. A model for the stability analysis of maintenance stragies for linear list. *Comput. J. 34*, 1, 80–87. Oxford University Press, Oxford, UK.

Bastani, F. B., Hilal, W., and Chen, I.-R. 1986. Performance analysis of concurrent maintenance policies for servers in a distributed environment. In *Proceedings of the Fall Joint Computer Conference.* ACM and IEEE-CS, New York, 611–619.

Bastani, F. B., Iyengar, S. S., and Yen, I.-L. 1988. Concurrent maintenance of data structures in a distributed environment. *Comput. J. 31*, 2, 165–174. Oxford University Press, Oxford, UK.

Batory, D. S. 1982. Optimal file designs and reorganization points. *ACM Trans. Datab. Syst. 7*, 1, 60–81.

Batra, D. P. 1989. Response time analysis for computer database reorganisation with concurrent usage and changeover state. *Defence Sci. J. 39*, 1, 33–41. Defence Scientific Info. and Documentation Centre, Ministry of Defence, Delhi, India.

Bayer, R. and McCreight, E. 1970. Organization and maintenance of large ordered indices. In *Proceedings of the ACM SICFIDET Workshop on Data Description and Access*. ACM, New York, 107–141. (For a revision, see Bayer and McCreight [1972].)

Bayer, R. and McCreight, E. 1972. Organization and maintenance of large ordered indexes. *Acta Inf. 1*, 3, 173–189.

Bellahsene, Z. 1996. View mechanism for schema evolution in object-oriented DBMS. In *Advances In Databases*, R. Morrison and J. Kennedy, Eds. Lecture Notes in Computer Science, vol. 1094. Springer-Verlag, New York, 18–35. (Proceedings 14th British National Conference on Databases, BNCOD 14.)

Bennett, B. T. and Franaszek, P. A. 1977. Permutation clustering: An approach to on-line storage reorganization. *IBM J. Res. Develop. 21*, 6, 528–533.

Bernstein, P. A. and Goodman, N. 1981. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2, 185–221.

Bertino, E. 1992. A view mechanism for object-oriented databases. In *Advances in Database Technology – EDBT '92*, A. Pirotte, C. Delobel, and G. Gottlob, Eds. Lecture Notes in Computer Science, vol. 580. Springer-Verlag, New York, 136–151. (Proceedings 3rd International Conference on Extending Database Technology.)

Birrell, A. D. and Needham, R. M. 1978. An asynchronous garbage collector for the CAP filing system. *Oper. Syst. Rev. 12*, 2, 31–33. (ACM SIGOPS.)

Blackburn, S. M., Cheng, P., and McKinley, K. S. 2004. Myths and realities: The performance impact of garbage collection. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2004)*. ACM, New York, 25–36. (*Perform. Evalu. Rev. 32*, 1, SIGMETRICS).

Blackburn, S. M. and McKinley, K. S. 2003. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*. ACM, New York, 344–358. (*SIGPLAN Notices 38*, 11).

Blackwell, T., Harris, J., and Seltzer, M. 1995. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the 1995 USENIX Technical Conference* USENIX Association, Berkeley, 277–288.

BMC Software. 1997. *REORG PLUS for DB2: General Information, Version 5.1*. BMC Software.

BMC Software. 1998. *CONCURRENT REORG and the Extended Performance Utilities: General Information, Version 1.3*. BMC Software.

Bollella, G., Lizzi, C., and Daynes, L. P. 2009. System and method for ensuring non-interfering garbage collection in a real-time multi-threaded environment. U.S. patent 7,484,067.

Bracht, C. J., Malkemus, T. R., and Versteeg, A. 1991. Table reorganization identification. *IBM Tech. Disclosure Bull. 34*, 1, 163–165.

Bratsberg, S. E. 1992. Unified class evolution by object-oriented views. In *Proceedings of the 11th International Conference on the Entity-Relationship Approach – ER '92*, G. Pernul and A. M. Tjoa, Eds. Lecture Notes in Computer Science, vol. 645. Springer-Verlag, New York, 423–439. (For more details, see Bratsberg [1993].)

Bratsberg, S. E. 1993. Evolution and integration of classes in object-oriented databases. Dr.Ing. thesis, Division of Computer Systems and Telematics, Norwegian Institute of Technology, Trondheim, Norway.

Bratsberg, S. E. and Humborstad, R. 2001. Online scaling in a highly available database. In *Proceedings of the 27th International Conference on Very Large Data Bases*. Morgan Kaufmann, San Francisco, 451–460.

Bratsberg, S. E. Hvasshovd, S.-O., and Torbjørnsen, Ø. 1997. Parallel solutions in ClustRa. *Data Engin. 20*, 2, 13–20.

Bratsberg, S. E. and Torbjørnsen, Ø. 2000. Tutorial: Designing an ultra highly available DBMS. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. (*SIGMOD Record 29*, 2).

Breitbart, Y., Vingralek, R., and Weikum, G. 1996. Load control in scalable distributed file structures. *Distrib. Paral. Datab. 4*, 4, 319–354.

Brown, K. P., Carey, M. J., and Livny, M. 1993. Managing memory to meet multi-class workload response time goals. In *Proceedings of the 19th International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 328–341.

Brown, K. P., Carey, M. J., and Livny, M. 1996. Goal-oriented buffer management revisited. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 353–364. (*SIGMOD Record 25*, 2).

Brownbridge, D. R. 1985. Cyclic reference counting for combinator machines. In *Functional Programming Languages and Computer Architecture*, J.-P. Jouannaud, Ed. Lecture Notes in Computer Science, vol. 201. Springer-Verlag, New York, 273–288. (Proceedings IFIP Conference).

Butler, M. H. 1986. Storage reclamation for object oriented database systems: A summary of the expected costs. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*. IEEE Computer Society Press, Los Alamitos, 210–211. (For more details, see Butler [1987].)

Butler, M. H. 1987. Storage reclamation in object oriented database systems. In *Proceedings of the ACM SIGMOD 1987 Annual Conference*. ACM, New York, 410–425. (*SIGMOD Record 16*, 3.)

Chamberlin, D. D., Gray, J. N., and Traiger, I. L. 1975. Views, authorization, and locking in a relational data base system. In *Proceedings of the National Computer Conference*. vol. 44. AFIPS Press, Reston, 425–430.

Chamberlin, D. D. and Schmuck, F. B. 1992a. Dynamic data distribution (D^3) in a shared-nothing multiprocessor data store. In *Proceedings of the 18th International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 163–174. (For more details, see Chamberlin and Schmuck [1992b].)

Chamberlin, D. D. and Schmuck, F. B. 1992b. Dynamic data distribution in a shared-nothing multiprocessor data store. Resear. rep. RJ 8730, IBM T. J. Watson Research Center, Yorktown Heights.

Chang, E. E. and Katz, R. H. 1989. Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented DBMS. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 348–357. (*SIGMOD Record 18*, 2).

Chee, C.-L., Lu, H., Tang, H., and Ramamoorthy, C. V. 1998. Adaptive prefetching and storage reorganization in a log-structured storage system. *IEEE Trans. Knowl. Data Engin. 10*, 5, 824–838.

Chen, I.-R. 1995. A degradable B^{link}-tree with periodic data reorganization. *Comput. J. 38*, 3, 245–252.

Chen, I.-R. and Banawan, S. A. 1993. Modeling and analysis of concurrent maintenance policies for data structures using pointers. *IEEE Trans. Soft. Engin. 19*, 9, 902–911.

Chen, I.-R. and Banawan, S. A. 1999. Performance and stability analysis of multi-level data structures with deferred reorganization. *IEEE Trans. Soft. Engin. 25*, 5, 690–700.

Chen, I.-R. and Hassan, S. 1995. Performance analysis of a periodic data reorganization algorithm for concurrent B^{link}-trees in database systems. In *Proceedings of the 1995 ACM Symposium on Applied Computing*. ACM, New York, 40–45.

Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. 1994. RAID: High-performance, reliable secondary storage. *ACM Comput. Surv. 26*, 2, 145–185.

Cheney, C. J. 1970. A nonrecursive list compacting algorithm. *Comm. ACM 13*, 11, 677–678.

Cheng, J. R. and Hurson, A. R. 1991. Effective clustering of complex objects in object-oriented databases. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 22–31. (*SIGMOD Record 20*, 2).

Chiang, M.-L. and Huang, J.-S. 2007. Improving the performance of log-structured file systems with adaptive block rearrangement. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC)*. 1136–1140.

Clamen, S. M. 1994. Schema evolution and integration. *Distrib. Paral. Datab. 2*, 1, 101–126. Kluwer Academic Publishers, Hingham.

Cohen, J. 1981. Garbage collection of linked data structures. *ACM Comput. Surv. 13*, 3 (Sept.), 341–367.

Collins, G. E. 1960. A method for overlapping and erasure of lists. *Comm. ACM 3*, 12, 655–657.

Comer, D. 1979. The ubiquitous B-tree. *ACM Comput. Surv. 11*, 2, 121–137.

Cook, J. E., Klauser, A. W., Wolf, A. L., and Zorn, B. G. 1996. Semi-automatic, self-adaptive control of garbage collection rates in object databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 377–388. (*SIGMOD Record 25*, 2).

Cook, J. E., Wolf, A. L., and Zorn, B. G. 1994. Partition selection policies in object database garbage collection. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 371–382. (*SIGMOD Record 23*, 2). (For a revision with more details and policies, see Cook et al. [1998].)

Cook, J. E., Wolf, A. L., and Zorn, B. G. 1998. A highly effective partition selection policy for object database garbage collection. *IEEE Trans. Knowl. Data Engin. 10*, 1, 153–172.

Copeland, G., Alexander, W., Boughter, E., and Keller, T. 1988. Data placement in Bubba. In *Proceedings of the SIGMOD International Conference on Management of Data*. ACM, New York, 99–108. (*SIGMOD Record 17*, 3, ACM).

Copeland, G., Franklin, M., and Weikum, G. 1990. Uniform object management. In *Advances in Database Technology – EDBT '90*, F. Bancilhon, C. Thanos, and D. Tsichritzis, Eds. Lecture Notes in Computer Science, vol. 416. Springer-Verlag, New York, 253–268. (Proceedings of the International Conference on Extending Database Technology.)

Crus, R. A. 1984. Data recovery in IBM Database 2. *IBM Syst. J. 23*, 2, 178–188.

DETLEFS, D. L. 1990. Concurrent, atomic garbage collection. Ph.D. dissertation, Department of Computer Science, Carnegie Mellon University, Pittsburgh. (Also Tech. rep. CMU-CS-90-177).

DETLEFS, D. L. 2004. A hard look at hard real-time garbage collection. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*. IEEE Computer Society Press, Los Alamitos, 23–32.

DETLEFS, D. L., FLOOD, C., HELLER, S., AND PRINTEZIS, T. 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM 2004)*. ACM, 37–48.

DEUTSCH, L. P. AND BOBROW, D. G. 1976. An efficient, incremental, automatic garbage collector. *Comm. ACM 19*, 9, 522–526.

DEWITT, D. J. AND GRAY, J. 1992. Parallel database systems: The future of high-performance database systems. *Comm. ACM 35*, 6, 85–98.

DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. 1978. On-the-fly garbage collection: An exercise in cooperation. *Comm. ACM 21*, 11, 966–975.

DOMANI, T., KOLODNER, E. K., LEWIS, E., SALANT, E. E., BARABASH, K., LAHAN, I., LEVANONI, Y., PETRANK, E., AND YANOVER, I. 2000a. Implementing an on-the-fly garbage collector for Java. In *Proceedings of the ISMM 2000, International Symposium on Memory Management*. ACM, New York, 155–166. (*SIGPLAN Notices 36*, 1, 2001).

DOMANI, T., KOLODNER, E. K., AND PETRANK, E. 2000b. A generational on-the-fly garbage collector for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, 274–284. (*SIGPLAN Notices 35*, 5).

EGGERT, L. AND TOUCH, J. D. 2005. Idletime scheduling with preemption intervals. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*. ACM, New York, 249–262. (*Operat. Syst. Rev. 39*, 5, SIGOPS).

ENGLERT, S. 1994. NonStop SQL: Scalability and availability for decision support. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. 491. (*SIGMOD Record 23*, 2).

FANG, M. T., LEE, R. C. T., AND CHANG, C. C. 1986. The idea of de-clustering and its applications. In *Proceedings of the 12th International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 181–188.

FEELIFL, H. AND KITSUREGAWA, M. 2001. RING: A strategy for minimizing the cost of online data placement reorganization for Btree indexed database over shared-nothing parallel machines. In *Proceedings of the 7th International Conference on Database Systems for Advanced Applications.* IEEE Computer Society Press, Los Alamitos, 190–199.

FEELIFL, H., KITSUREGAWA, M., AND OOI, B.-C. 2000. A fast convergence technique for online heat-balancing of Btree indexed database over shared-nothing parallel systems. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, M. T. Ibrahim, J. Küng, and N. Revell, Eds. Lecture Notes in Computer Science, vol. 1873. Springer-Verlag, New York, 846–858.

FERRANDINA, F., MEYER, T., AND ZICARI, R. 1994. Correctness of lazy database updates for object database systems. In *Proceedings of the 6th International Workshop on Persistent Object Systems*, M. P. Atkinson, D. Maier, and V. Benzaken, Eds. Springer-Verlag, New York, 284–301. (For a revision, see Ferrndina et al. [1995].)

FERRANDINA, F., MEYER, T., ZICARI, R., FERRAN, G., AND MADEC, J. 1995. Schema and database evolution in the O_2 object database System. In *Proceedings of the 21st International Conference on Very Large Data Bases.* Morgan-Kaufmann, San Francisco, 170–181.

FERREIRA, P. AND SHAPIRO, M. 1994a. Garbage collection and DSM consistency. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Berkeley, 229–241. (Also *Operat. Syst. Rev.*, ACM SIGOPS.)

FERREIRA, P. AND SHAPIRO, M. 1994b. Garbage collection of persistent objects in distributed shared memory. In *Proceedings of the 6th International Workshop on Persistent Object Systems*, M. P. Atkinson, D. Maier, and V. Benzaken, Eds. Springer-Verlag, New York, 184–199.

FONTANA, E. AND DENNEBOUY, Y. 1995. Time-stamped versions for lazy propagation of schema changes. Tech. rep. 95/100, Department of Informatics, Swiss Federal Institute of Technology, Lausanne, Switzerland.

FORD, D. A. AND MYLLYMAKI, J. 1996. A log-structured organization for tertiary storage. In *Proceedings of the 12th International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, 20–27.

FRANASZEK, P. A. AND CONSIDINE, J. P. 1979. Reduction of storage fragmentation on direct access devices. *IBM J. Res. Devel. 23*, 2, 140–148.

FRANASZEK, P. A., ROBINSON, J. T., AND THOMASIAN, A. 1994. Reorganizing data in log-structured file systems. *IBM Tech. Disclos. Bull. 37*, 6A, 623–624.

Franklin, M., Copeland, G., and Weikum, G. 1989. What's different about garbage collection for persistent programming languages? Tech. rep. ACA-ST-062-89, MCC, Austin.

Friske, C. A. 2004a. DB2 managing large partitioned tables in Version 8. *Proc. SHARE 102*. SHARE, Chicago.

Friske, C. A. 2004b. DB2 online schema changes – what's new in DB2 Version 8. *Proc. SHARE 102*. SHARE, Chicago.

Friske, C. A., Garth, J. M., Lee, C. M., and Ruddy, J. A. 2007. Method and apparatus for building one or more indexes on data concurrent with manipulation of data. U.S. patent 7,308,456.

Friske, C. A., Garth, J. M., and Ruddy, J. A. 2003a. Method of estimating an amount of changed data over plurality of intervals of time measurements. U.S. patent 6,535,870.

Friske, C. A. and Ruddy, J. A. 1998. Online reorganization. *DB2 Mag. 3*, 3. (Online Ed. http://www.db2mag.com.) CMP Media.

Friske, C. A., Ruddy, J. A., and Shibamiya, A. 2003b. Method for estimating the elapsed-time required for a log apply process. U.S. patent 6,535,893.

Galante, R. d. M., dos Santos, C. S., Edelweiss, N., and Moreira, A. F. 2005. Temporal and versioning model for schema evolution in object-oriented databases. *Data Knowl. Engin. 53*, 2, 99–128. North-Holland, Amsterdam, Neth.

Ganesan, P., Bawa, M., and Garcia-Molina, H. 2004. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proceedings of the 30th International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 444–455.

Garnett, N. H. and Needham, R. M. 1980. An asynchronous garbage collector for the Cambridge File Server. *Operat. Syst. Rev. 14*, 4, 36–40.

Garthwaite, A. T. 2008. Concurrent incremental garbage collector with a card table summarizing modified reference locations. U.S. patent 7,412,580.

Gerritsen, R. 1994. Private communication.

Gerritsen, R. and Morgan, H. L. 1976. Dynamic restructuring of databases with generation data structures. In *Proceedings of the ACM Annual Conference*. ACM, New York, 281–286.

Ghandeharizadeh, S. and DeWitt, D. J. 1990. A multiuser performance analysis of alternative declustering strategies. In *Proceedings of the 6th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 466–475.

Ghandeharizadeh, S., Gao, S., Gahagan, C., and Krauss, R. 2006. An on-line reorganization framework for SAN file systems. In *Proceedings of the 10th East European Conference on Advances in Databases and Information Systems*, Y. Manolopoulos, J. Pokorný, and T. K. Sellis, Eds. Lecture Notes in Computer Science, vol. 4152. Springer-Verlag, New York, 399–414.

Ghandeharizadeh, S. and Kim, D. 1996. Online reorganization of data in scalable continuous media servers. In *Proceedings of the 7th International Conference on Database and Expert Systems Applications*, R. Wagner and H. Thoma, Eds. Lecture Notes in Computer Science, vol. 1134. Springer-Verlag, New York, 751–768.

Golding, R., Bosch, P., Staelin, C., Sullivan, T., and Wilkes, J. 1995. Idleness is not sloth. In *Proceedings of the 1995 USENIX Technical Conference.* USENIX Assoc., Berkeley, 201–212.

Goodman, N. and Shasha, D. 1985. Semantically-based concurrency control for search structures. In *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. ACM, New York, 8–19. (For a revision, see Shasha and Goodman [1988].)

Goralwalla, I. A., Szafron, D., Özsu, M. T., and Peters, R. J. 1998. A temporal approach to managing schema evolution in object database systems. *Data Knowl. Engin. 28*, 1, 73–105. North-Holland, Amsterdam, Netherlands.

Goyal, B., Haritsa, J. R., Seshadri, S., and Srinivasan, V. 1995. Index concurrency control in firm real-time DBMS. In *Proceedings of the 21st International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 146–157.

Graefe, G. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv. 25*, 2, 73–170.

Gray, J. 1978. Notes on data base operating systems. In *Operating Systems—An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmüller, Eds. Springer-Verlag, New York, 393–481.

Gray, J. and Reuter, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann, San Francisco.

Guibas, L. J. and Sedgewick, R. 1978. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, 8–21.

GUPTA, A. AND FUCHS, W. K. 1993. Garbage collection in a distributed object-oriented system. *IEEE Trans. Knowl. Data Engin. 5*, 2, 257–265.

HAERDER, T. AND REUTER, A. 1983. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4, 287–317.

HAMADA, Y., ISHIHARA, K., MASUMOTO, T., FUJITA, T., MASATOMI, K., TAKEDA, Y., AND KAWASHIMA, M. 2000. Online database duplication with incorporation of concurrent database updates. U.S. patent 6,023,707.

HARTMAN, J. H. AND OUSTERHOUT, J. K. 1995. The Zebra striped network file system. *ACM Trans. Comput. Syst. 13*, 3, 274–310.

HELAL, A., YUAN, D., AND EL-REWINI, H. 1997. Dynamic data reallocation for skew management in shared-nothing parallel databases. *Distrib. Paral. Datab. 5*, 3, 271–288.

HEYMAN, D. P. 1982. Mathematical models of database degradation. *ACM Trans. Datab. Syst. 7*, 4, 615–631.

HO, F., JAIN, R., AND TROISI, J. 1994. An overview of NonStop SQL/MP. *Tandem Syst. Rev. 10*, 3, 6–17. Hewlett-Packard.

HURAS, M. A., HING, N. H., GOSS, J. J., AND LINDSAY, B. G. 2005. Online database table reorganization. U.S. patent 6,950,834.

IBM. 1986. *IMS/VS Version 1 Utilities Reference Manual*. IBM. SH20-9029-9.

IBM. 1993a. *Implementing Concurrent Copy*. IBM. GG24-3990-00.

IBM. 1993b. *An Introduction to Data Propagator Relational Release 1*. IBM. GC26-3398-01.

IBM. 1997. *DB2 for OS/390 Version 5 Utility Guide and Reference*. IBM. SC26-8967-00.

IBM. 2001. *DB2 Universal Database for OS/390 and z/OS Utility Guide and Reference, Version 7*. IBM. SC26-9945-00.

IBM. 2005. *IMS Administration Guide: Database Manager, Version 9*. IBM. SC18-7806-01.

IBM and Integrated Systems Solutions. 1994. *Replidata/MVS User's Guide*. IBM and Integrated Systems Solutions. BLD-REP-UG-00.

ISIP, JR., A. B. 2007. System and method for reorganizing stored data. U.S. patent 7,225,206.

Itasca Systems. 1993. *ITASCA Distributed Object Database Management System: Technical Summary for Release 2.2.* Itasca Systems.

IYER, B. R. AND SOCKUT, G. H. 2002. Methods for in-place online reorganization of a database. U.S. patent 6,411,964.

IYER, B. R. AND WILHITE, D. 1994. Data compression support in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 695–704.

JAMBOR, M., HRUBY, T., TAUS, J., KRCHAK, K., AND HOLUB, V. 2007. Implementation of a Linux log-structured file system with a garbage collector. *Operat. Syst. Rev. 41*, 1, 24–32.

JOHNSON, T. AND SHASHA, D. 1990. A framework for the performance analysis of concurrent B-tree algorithms. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* ACM, New York, 273–287. (For more details, see Johnson and Shasha [1993].)

JOHNSON, T. AND SHASHA, D. 1993. The performance of concurrent B-tree algorithms. *ACM Trans. Datab. Syst. 18*, 1, 51–101.

JOISHA, P. G. 2007. Overlooking roots: A framework for making nondeferred reference-counting garbage collection fast. In *Proceedings of the 6th International Symposium on Memory Management (ISMM 2007)*. ACM, New York, 141–157.

JOISHA, P. G. 2008. A principled approach to nondeferred reference-counting garbage collection. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, New York, 131–140.

JONES, R. AND LINS, R. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York.

DE JONGE, W., KAASHOEK, M. F., AND HSIEH, W. C. 1993. The logical disk: A new approach to improving file systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM, New York, 15–28. (*Operat. Syst. Rev. 27*, 5, SIGOPS).

KATABAMI, N., ITOH, T., AND TAKAHASHI, Y. 1997. Method and apparatus for reorganizing an on-line database system in accordance with an access time increase. U.S. patent 5,596,747.

KERMANY, H. AND PETRANK, E. 2006. The Compressor: Concurrent, incremental, and parallel compaction. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, 354–363. (*SIGPLAN Notices 41*, 6).

KERO, M., NORDLANDER, J., AND LINDGREN, P. 2007. A correct and useful incremental copying garbage collector. In *Proceedings of the 6th International Symposium on Memory Management (ISMM 2007)*. ACM, New York, 129–140.

KESSELS, J. L. W. 1983. On-the-fly optimization of data structures. *Comm. ACM 26*, 11, 895–901.

KEYES, J. 1992. DBAs face challenge of 24 by 7 availability. *Software Mag. 12*, 11, 58–63.

KIM, M. Y. 1986. Synchronized disk inter-leaving. *IEEE Trans. Comput. C-35*, 11, 978–988.

KITSUREGAWA, M., GODA, K., AND KAWAMURA, N. 2008. Method and system for data processing with database reorganization for the same. U.S. patent 7,421,456.

KITSUREGAWA, M. AND MOGI, K. 1993. Virtual striping: A RAID5 storage management scheme with robustness for the peak access traffic. In *Proceedings of the International Symposium on Next Generation Database Systems and Their Applications*. ACM, New York, 280–287.

KOHL, J. T., STAELIN, C., AND STONEBRAKER, M. 1993. HighLight: Using a log-structured file system for tertiary storage management. In *Proceedings of the Winter 1993 USENIX Conference*. USENIX Assoc., Berkeley, 435–447.

KOLODNER, E. K. 1990. Atomic incremental garbage collection and recovery for a large stable heap. In *Proceedings of the 4th International Workshop on Persistent Object Systems*, A. Dearle, G. M. Shaw, and S. B. Zdonik, Eds. Morgan-Kaufmann, San Francisco, 185–198. (see Kolodner [1992].)

KOLODNER, E. K. 1992. Atomic incremental garbage collection and recovery for a large stable heap. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge. (Also Tech. rep. MIT/LCS/TR-534, Laboratory for Computer Science.)

KOLODNER, E. K., LEWIS, E., AND PETRANK, E. 2005. Trace termination for on-the-fly garbage collection for weakly-consistent computer architecture. U.S. patent 6,920,541.

KOLODNER, E. K. AND PETRANK, E. 2002. On-the-fly garbage collector. U.S. patent 6,490,599.

KOLODNER, E. K. AND WEIHL, W. E. 1993. Atomic incremental garbage collection and recovery for a large stable heap. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. 177–186. (*SIGMOD Record 22*, 2). For more details, see Kolodner [1992].)

KUNG, H. T. AND LEHMAN, P. L. 1980. Concurrent manipulation of binary search trees. *ACM Trans. Datab. Syst. 5*, 3, 354–382.

KUO, T.-W., WEI, C.-H., AND LAM, K.-Y. 1999. Real-time data access control on B-tree index structures. In *Proceedings of the 15th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 458–467.

LAKHAMRAJU, M. K., RASTOGI, R., SESHADRI, S., AND SUDARSHAN, S. 2000. On-line reorganization in object databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 58–69. (*SIGMOD Record 29*, 2). (This is similar to Lakhamraju et al. [2002].)

LAKHAMRAJU, M. K., RASTOGI, R., SESHADRI, S., AND SUDARSHAN, S. 2002. On-line reorganization in object-oriented databases. U.S. patent 6,343,296. (This is similar to Lakhamraju et al. [2000].)

LAM, K.-W. AND LEE, V. C. S. 2006. On consistent reading of entire databases. *IEEE Trans. Knowl. Data Engin. 18*, 4, 569–572.

LAMPSON, B. W. 1984. Hints for computer system design. *Software 1*, 1, 11–28. IEEE-CS.

LANGLEY, J. T. AND MOORE, D. W. 2008. Non-disruptive backup copy in a database online reorganization environment. U.S. patent 7,433,902.

LEE, M.-L., KITSUREGAWA, M., OOI, B. C., TAN, K.-L., AND MONDAL, A. 2000. Towards self-tuning data placement in parallel database systems. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 225–236. May (*SIGMOD Record 29*, 2). (For more details, see Mondal [2002].)

LITWIN, W. 1980. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases*. IEEE Computer Society Press, Los Alamitos, 212–223.

LIU, C.-T., CHRYSANTHIS, P. K., AND CHANG, S.-K. 1994a. Database schema evolution through the specification and maintenance of changes on entities and relationships. In *Proceedings of the 13th International Conference on the Entity-Relationship Approach, ER '94*, P. Loucopoulos, Ed. Lecture Notes in Computer Science, vol. 881. Springer-Verlag, New York, 132–151.

LIU, L., ZICARI, R., HÜRSCH, W., AND LIEBERHERR, K. 1994b. Polymorphic reuse mechanisms for object-oriented database specifications. In *Proceedings of the 10th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 180–189.

LOHMAN, G. M. 1977. Optimal data storage and organization in computerized information processing systems subject to failures. Ph.D. dissertation. School of Operations Research and Industrial Engineering, Cornell University, Ithaca.

LOHMAN, G. M. AND MUCKSTADT, J. A. 1977. Optimal policy for batch operations: Backup, checkpointing, reorganization, and updating. *ACM Trans. Datab. Syst. 2*, 3, 209–222.

LØLAND, J. AND HVASSHOVD, S.-O. 2006. Online, non-blocking relational schema changes. In *Proceedings of the 10th International Conference on Extending Database Technology. Advances in Database*

Technology—EDBT 2006, Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, and C. Böhm, Eds. Lecture Notes in Computer Science, vol. 3896. Springer-Verlag, New York, 405–422.

LOMET, D. B. 2000. High-speed on-line backup when using logical log operations. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 34–45. (*SIGMOD Record 29*, 2).

LOMET, D. B. AND SALZBERG, B. 1992. Access method concurrency with recovery. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 351–360. (*SIGMOD Record 21*, 2). (For more details, see Lomet and Salzberg [1997].)

LOMET, D. B. AND SALZBERG, B. 1997. Concurrency and recovery for index trees. *VLDB J. 6*, 3, 224–240.

LUMB, C. R., SCHINDLER, J., GANGER, G. R., AND NAGLE, D. 2000. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*. USENIX Association, Berkeley, 87–102.

MAHESHWARI, U. AND LISKOV, B. 1997. Partitioned garbage collection of a large object store. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. 313–323. (*SIGMOD Record 26*, 2).

MAIER, D. S., MARTON, R. S., TROISI, J. H., AND CELIS, P. 1997. Relational database system and method with high data availability during table data restructuring. U.S. patent 5,625,815.

MALHOTRA, A. AND MUNROE, S. J. 1997. Schema evolution in persistent object systems. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, R. Connor and S. Nettles, Eds. Morgan-Kaufmann, San Francisco, 194–204.

MANBER, U. 1984. Concurrent maintenance of binary search trees. *IEEE Trans. Softw. Engin. SE-10*, 6, 777–784.

MANBER, U. AND LADNER, R. E. 1982. Concurrency control in a dynamic search structure. In *Proceedings of the ACM Symposium on Principles of Database Systems.* ACM, New York, 268–282. (For a revision, see Manber and Ladner [1984].)

MANBER, U. AND LADNER, R. E. 1984. Concurrency control in a dynamic search structure. *ACM Trans. Datab. Syst. 9*, 3, 439–455.

MARSHALL, B. J., HENDERSON, M. D., BAILEY, M. I., BRUCE, T. R., ROGALA, R. J., WEBSTER, W. A., METZNER, D. F., AND DRES, P. J. 2006. Method and system for online reorganization of databases. U.S. patent 7,117,229.

MARTIN, J. 1975. *Computer Data-Base Organization*. Prentice-Hall, Englewood Cliffs, NJ.

MARTIN, JR., J. L. AND CROWN, JR., G. N. 2003a. IMS on-line reorganization utility. U.S. patent 6,606,631.

MARTIN, JR., J. L. AND CROWN, JR., G. N. 2003b. System and method for analyzing a database for on-line reorganization. U.S. patent 6,633,884.

MARUYAMA, K. AND SMITH, S. E. 1976. Optimal reorganization of distributed space disk files. *Comm. ACM 19*, 11, 634–642.

MATTHEWS, J. N., ROSELLI, D., COSTELLO, A. M., WANG, R. Y., AND ANDERSON, T. E. 1997. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. ACM, New York, 238–251. (*Operat. Syst. Rev. 31*, 5, SIGOPS).

MCAULIFFE, M. L., CAREY, M. J., AND SOLOMON, M. H. 1998. Vclusters: A flexible, fine-grained object clustering mechanism. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 230–243. (*SIGPLAN Notices 33*, 10).

MCCARTHY, J. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Comm. ACM 3*, 4, 184–195.

MCCREIGHT, A., SHAO, Z., LIN, C., AND LI, L. 2007. A general framework for certifying garbage collectors and their mutators. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 468–479. (*SIGPLAN Notices 42*, 6).

MCIVER, JR., W. J. 1994. An approach to self-adaptive, on-line reclustering of complex object data. Ph.D. dessertation. Department of Computer Science, University of Colorado, Boulder.

MCIVER, JR., W. J. AND KING, R. 1994. Self-adaptive, on-line reclustering of complex object data. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 407–418. (*SIGMOD Record 23*, 2). (For more details, see McIver [1994].)

MCNUTT, B. 1994. Background data movement in a log-structured disk subsystem. *IBM J. Resear. Develop. 38*, 1, 47–58.

MELTZER, H. S. 1975. An overview of the administration of data bases. In *Proceedings of the 2nd USA-Japan Computer Conference*. AFIPS Press, Reston, 365–370.

MELTZER, H. S. 1976. Structure and redundancy in the conceptual schema in the administration of very large data bases. In *Proceedings of the 2nd International Conference on Very Large Data*

Bases, P. C. Lockemann and E. J. Neuhold, Eds. North-Holland, Amsterdam, The Netherlands, 13–25.

Mendelson, H. and Yechiali, U. 1981. Optimal policies for data base reorganization. *Oper. Resear. 29*, 1, 23–36.

Menon, J. and Stockmeyer, L. 1998. An age-threshold algorithm for garbage collection in log-structured arrays and file systems. In *Proceedings of the 12th Annual International Symposium on High-Performance Computing Systems and Application*, J. Schaefer, Ed. Kluwer Academic Publishers, Hingham, 119–132. (More details appear in Resear. Rep. RJ 10120, IBM T. J. Watson Research Center, Yorktown Heights.)

Mix, F. 1992. DB2 Reorg and continuous select access. In *Proceedings of the Summer 1992 Meeting / SHARE 79*. SHARE, Chicago, 2577–2614. Vol. II (microfiche). (Mix [1994] describes a revision.)

Mix, F. 1994. DB2 Reorg and continuous select. In *Proceedings of the 6th Annual North American Conference of the International DB2 Users Group*. IDUG, Chicago, 545–563.

Mizoguchi, M. and Nishiyama, K. 1994. Database management system. Japan patent 06-67944.

Mogi, K. and Kitsuregawa, M. 1994. Dynamic parity stripe reorganizations for RAID5 disk arrays. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS 94)*. IEEE Computer Society Press, Los Alamitos, 17–26.

Mogi, K. and Kitsuregawa, M. 1995. Hot block clustering for disk arrays with dynamic striping: Exploitation of accesses locality and its performance analysis. In *Proceedings of the 21st International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 90–99.

Mohan, C. 1993a. IBM's relational DBMS products: Features and technologies. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 445–448. (*SIGMOD Record 22*, 2).

Mohan, C. 1993b. A survey of DBMS research issues in supporting very large tables. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, D. B. Lomet, Ed. Lecture Notes in Computer Science, vol. 730. Springer-Verlag, New York, 279–300.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Datab. Syst. 17*, 1, 94–162.

Mohan, C. and Narang, I. 1992. Algorithms for creating indexes for very large tables without quiescing updates. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 361–370. (*SIGMOD Record 21*, 2).

Mohan, C. and Narang, I. 1993. An efficient and flexible method for archiving a data base. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 139–146. (*SIGMOD Record 22*, 2).

Moitra, A., Iyengar, S. S., Bastani, F. B., and Yen, I.-L. 1988. Multi-level data structures: Models and performance. *IEEE Trans. Softw. Engin. 14*, 6, 858–867.

Mondal, A. 2002. Query-centric online reorganization of indexed data in shared-nothing architectures. Ph.D. dissertation, School of Computing, National University of Singapore.

Mondal, A., Kitsuregawa, M., Ooi, B. C., and Tan, K.-L. 2001. R-tree-based data migration and self-tuning strategies in shared-nothing spatial databases. In *Proceedings of the 9th ACM International Symposium on Advances in Geographic Information Systems*. ACM, New York, 28–33. (For more details, see Mondal [2002].)

Morsi, M. M. A. 1992. Extensible object-oriented databases with dynamic schemas. Ph.D. dissertation, College of Computing, Georgia Institute of Technology, Atlanta.

Morsi, M. M. A., Navathe, S. B., and Kim, H.-J. 1991. A schema management and prototyping interface for an object-oriented database environment. In *Proceedings of the IFIP Working Conference on Object-Oriented Approach in Information Systems*, F. Van Assche, B. Moulin, and C. Rolland, Eds. North-Holland, Amsterdam, The Netherlands, 157–180. (For more details, see Morsi [1992].)

Morsi, M. M. A., Navathe, S. B., and Kim, H.-J. 1992. An extensible object-oriented database testbed. In *Proceedings of the 8th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 150–157. (For more details, see Morsi [1992].)

Moss, J. E. B., Munro, D. S., and Hudson, R. L. 1997. PMOS: A complete and coarse-grained incremental garbage collector for persistent object stores. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, R. Connor and S. Nettles, Eds. Morgan Kaufmann Publishers, San Francisco, 140–150. (Munro et al. [1999] describe an implementation.)

Munro, D. S., Brown, A. L., Morrison, R., and Moss, J. E. B. 1999. Incremental garbage collection of a persistent object store using PMOS. In *Advances in Persistent Object Systems, Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International*

Workshop on Persistence and Java (PJW3), 1998, R. Morrison, M. J. Jordan, and M. P. Atkinson, Eds. Morgan-Kaufmann, San Francisco, 78–91.

NETTLES, S. AND O'TOOLE, J. 1993. Real-time replication garbage collection. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. ACM, New York, 217–226. (*SIGPLAN Notices 28*, 6).

NILSEN, K. D. AND SCHMIDT, W. J. 1992. Cost-effective object space management for hardware-assisted real-time garbage collection. *ACM Lett. Prog. Lang. Syst. 1*, 4, 338–354.

NURMI, O., SOISALON-SOININEN, E., AND WOOD, D. 1987. Concurrency control in database structures with relaxed balance. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, New York, 170–176.

ODBERG, E. 1994. A global perspective of schema modification management for object-oriented data bases. In *Proceedings of the 6th International Workshop on Persistent Object Systems*, M. P. Atkinson, D. Maier, and V. Benzaken, Eds. Springer-Verlag, New York, 479–502.

OLSON, N. E. 1993. Partial indexing in POSTGRES. M.S. dissertation. Department of Electrical Engineering and Computer Science, University of California, Berkeley.

OMIECINSKI, E. 1985a. Concurrency during the reorganization of indexed files. In *Proceedings of the 9th International Conference on Computer Software and Applications (COMPSAC 85)*. IEEE Computer Society Press, Los Alamitos, 482–488.

OMIECINSKI, E. 1985b. Incremental file reorganization schemes. In *Proceedings of the 11th International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 346–357.

OMIECINSKI, E. 1988. Concurrent storage structure conversion: from B+ tree to linear hash file. In *Proceedings of the 4th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 589–596. (For an extended revision, see Omiecinski [1989].)

OMIECINSKI, E. 1989. Concurrent file conversion between B^+-tree and linear hash files. *Inf. Syst. 14*, 5, 371–383.

OMIECINSKI, E. 1996. Concurrent file reorganization: Clustering, conversion and maintenance. *Data Engin. 19*, 2, 25–32. IEEE-CS TC DE.

OMIECINSKI, E., LEE, L., AND SCHEUERMANN, P. 1992. Concurrent file reorganization for record clustering: A performance study. In *Proceedings of the 8th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 265–272. (For more details, see Omiecinski et al. [1994].)

OMIECINSKI, E., LEE, L., AND SCHEUERMANN, P. 1994. Performance analysis of a concurrent file reorganization algorithm for record clustering. *IEEE Trans. Knowl. Data Engin. 6*, 2, 248–257.

OMIECINSKI, E. AND SCHEUERMANN, P. 1984. A global approach to record clustering and file reorganization. In *Proceedings of the 3rd Joint BCS and ACM Symposium on Research and Development in Information Retrieval*, C. J. van Rijsbergen, Ed. Cambridge Univ. Press, Cambridge, 201–219. (For later work, see Omiecinski [1985b].)

ORACLE. 2005. Oracle Database 10g Release 2 online data reorganization & redefinition. Oracle White Paper.

OSBORN, S. L. 1989. The role of polymorphism in schema evolution in an object-oriented database. *IEEE Trans. Knowl. Data Engin. 1*, 3, 310–317.

OSSIA, Y., BEN-YITZHAK, O., GOFT, I., KOLODNER, E. K., LEIKEHMAN, V., AND OWSHANKO, A. 2002. A parallel, incremental and concurrent GC for servers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, 129–140. (*SIGPLAN Notices 37*, 5). (For a revision, see Barabash et al. [2005].)

O'TOOLE, J., NETTLES, S., AND GIFFORD, D. 1993. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM, New York, 161–174. (*Operat. Syst. Rev. 27*, 5, SIGOPS.)

OUSTERHOUT, J. AND DOUGLIS, F. 1989. Beating the I/O bottleneck: A case for log-structured file systems. *Operat. Syst. Rev. 23*, 1, 11–28. (For a revised, more detailed description, see Rosenblum and Ousterhout [1992].)

PARK, J. S., BARTOSZYNSKI, R., DE, P., AND PIRKUL, H. 1990. Optimal reorganization policies for stationary and evolutionary databases. *Manag. Sci. 36*, 5, 613–631.

PARK, J. S., BARTOSZYNSKI, R., AND PIRKUL, H. 1989. Optimal database reorganization policies: A stochastic control approach. In *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*. Vol. 3. IEEE Computer Society Press, Los Alamitos, 752–761. (For a revision, see Park et al. [1990].)

PARK, J. S. AND SRIDHAR, V. 1997. Probabilistic model and optimal reorganization of B+-tree with physical clustering. *IEEE Trans. Knowl. Data Engin. 9*, 5, 826–832.

Paz, H., Bacon, D. F., Kolodner, E. K., Petrank, E., and Rajan, V. T. 2007. An efficient on-the-fly cycle collection. *ACM Trans. Prog. Lang. Syst. 29*, 4. Article 20.

Paz, H., Petrank, E., Bacon, D. F., Kolodner, E. K., and Rajan, V. T. 2005. An efficient on-the-fly cycle collection. In *Compiler Construction: 14th International Conference, CC 2005, Held as Part of Joint European Conferences on Theory and Practice of Software, ETAPS 2005*, R. Bodik, Ed. Lecture Notes in Computer Science, vol. 3443. Springer-Verlag, New York, 156–171. (For a revision, see Paz et al. [2007].)

Pereira, H. M. 2000. Method and apparatus for reorganizing an active DBMS table. U.S. patent 6,122, 640.

Peters, R. J. and Özsu, M. T. 1995. Axiomatization of dynamic schema evolution in objectbases. In *Proceedings of the 11th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 156–164. (For a revision with more details, see Peters and Özsu [1997].)

Peters, R. J. and Özsu, M. T. 1997. An axiomatic model of dynamic schema evolution in objectbase systems. *ACM Trans. Datab. Syst. 22*, 1, 75–114.

Pizlo, F., Frampton, D., Petrank, E., and Steensgaard, B. 2007. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the 6th International Symposium on Memory Management (ISMM 2007)*. ACM, New York, 159–172.

Pizlo, F., Petrank, E., and Steensgaard, B. 2008. A study of concurrent real-time garbage collectors. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 33–44. (*SIGPLAN Notices 43*, 6).

Plow, G. M., Pourmirzaie, F. E., and Shiomi, T. 2008. Method for reorganizing a set of database partitions. U.S. patent 7,454,449.

Pollari-Malmi, K., Soisalon-Soininen, E., and Ylönen, T. 1996. Concurrency control in B-trees with batch updates. *IEEE Trans. Knowl. Data Engin. 8*, 6, 975–984.

Pong, M. 1990. An overview of NonStop SQL Release 2. *Tandem Syst. Rev. 6*, 2, 4–11.

Ponnekanti, N. and Kodavalla, H. 2000. Online index rebuild. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 529–538. May (*SIGMOD Record 29*, 2).

Printezis, T. and Detlefs, D. L. 2000. A generational mostly-concurrent garbage collector. In *Proceedings of the International Symposium on Memory Management (ISMM 2000)*. ACM, New York, 143–154. (*SIGPLAN Notices 36*, 1, ACM 2001).

Pu, C. 1985. On-the-fly, incremental, consistent reading of entire databases. In *Proceedings of the 11th International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 369–375. (For a revision, see Pu [1986].)

Pu, C. 1986. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica 1*, 3, 271–287.

Pu, C., Hong, C. H., and Wha, J. M. 1988. Performance evaluation of global reading of entire databases. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, 167–176.

Puffitsch, W. and Schoeberl, M. 2008. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems.* ACM, 68–76.

Ra, Y.-G. and Rundensteiner, E. A. 1995. A transparent object-oriented schema change approach using view evolution. In *Proceedings of the 11th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 165–172. (For a revision with more details, see Ra and Rundensteiner [1997].)

Ra, Y.-G. and Rundensteiner, E. A. 1997. A transparent schema-evolution system based on object-oriented view technology. *IEEE Trans. Knowl. Data Engin. 9*, 4, 600–624.

Radosevich, L. 1993. Hotel cans mainframes. *Computerworld 27*, 8, 1 and 16.

Raghuram, S., Ranganath, S., Olson, S., and Nandi, S. 2000. Taming the down-time: High availability in Sybase ASE 12. In *Proceedings of the 16th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 111–120. (For more details of index reorganization, see Ponnekanti and Kodavalla [2000].)

Rahm, E. and Bernstein, P. A. 2006. An online bibliography on schema evolution. *SIGMOD Record 35*, 4, 30–31. (For the current bibliography, see http://se-pubs.dbs.uni-leipzig.de.)

Ram, S. and Shankaranarayanan, G. 2003. Research issues in database schema evolution: The road not taken. Working Paper 2003-15, Information Systems Department, Boston University School of Management.

Ramírez, R. J., Tompa, F. W., and Munro, J. I. 1982. Optimum reorganization points for arbitrary database costs. *Acta Informatica 18*, 1, 17–30.

RENGARAJAN, T. K., DIMINO, L., AND CHUNG, D. 1996. Sybase System 11 online capabilities. *Data Engin. 19*, 2, 19–24.

ROBINSON, J. T. AND FRANASZEK, P. A. 1994. Analysis of reorganization overhead in log-structured file systems. In *Proceedings of the 10th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 102–110.

RODDICK, J. F. 1991. Dynamically changing schemas within database models. *Austral. Comput. J. 23*, 3, 105–109.

RODDICK, J. F. 1992. Schema evolution in database systems—An annotated bibliography. *SIGMOD Record 21*, 4, 35–40. (For a revision, see Roddick [1994].)

RODDICK, J. F. 1994. Schema evolution in database systems—An updated bibliography. Tech. rep. CIS-94-012, School of Computer and Information Science, University of South Australia, The Levels, S. Australia.

RODDICK, J. F. 1995. A survey of schema versioning issues for database systems. *Inf. Softw. Tech. 37*, 7, 383–393.

RONSTRÖM, M. 2000. On-line schema update for a telecom database. In *Proceedings of the 16th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 329–338.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Computer Syst. 10*, 1, 26–52.

ROSENKRANTZ, D. J. 1978. Dynamic database dumping. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, 3–8.

ROY, P., SESHADRI, S., SILBERSCHATZ, A., AND SUDARSHAN, S. 2002. Garbage collection in object-oriented databases using transactional cyclic reference counting. U.S. patent 6,363,403.

ROY, P., SESHADRI, S., SILBERSCHATZ, A., SUDARSHAN, S., AND ASHWIN, S. 1998. Garbage collection in object-oriented databases using transactional cyclic reference counting. *VLDB J. 7*, 3, 179–193.

RUDDY, J. A., SHYAM, K., SOCKUT, G. H., AND WATTS, J. A. 1999. Application of log records to data compressed with different encoding scheme. U.S. patent 5,897,641.

SAGIV, Y. 1985. Concurrent operations on B-trees with overtaking. In *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. ACM, New York, 28–37. (For a revision, see Sagiv [1986].)

SAGIV, Y. 1986. Concurrent operations on B*-trees with overtaking. *J. Comput. Syst. Sci. 33*, 2, 275–296.

SALANT, E. E. AND KOLODNER, E. K. 2002. Data structure for keeping track of objects remaining to be traced by concurrent garbage collector. U.S. patent 6,393,440.

SALEM, K. AND GARCIA-MOLINA, H. 1986. Disk striping. In *Proceedings of the International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 336–342.

SALZBERG, B. 1985. Restructuring the Lehman-Yao tree. Tech. rep. BS-85-21, College of Computer Science, Northeastern University, Boston.

SALZBERG, B. AND DIMOCK, A. 1992. Principles of transaction-based on-line reorganization. In *Proceedings of the 18th International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 511–520.

SCHEUERMANN, P., PARK, Y. C., AND OMIECINSKI, E. 1989. A heuristic file reorganization algorithm based on record clustering. *BIT 29*, 3, 428–447.

SCHEUERMANN, P., WEIKUM, G., AND ZABBACK, P. 1993. Adaptive load balancing in disk arrays. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, D. B. Lomet, Ed. Lecture Notes in Computer Science, vol. 730. Springer-Verlag, New York, 345–360. Foundations of Data Organization and Algorithms.

SCHEUERMANN, P., WEIKUM, G., AND ZABBACK, P. 1994. "Disk cooling" in parallel disk systems. *Bull. Tech. Committee Data Engin. 17*, 3, 29–40.

SCHEUERMANN, P., WEIKUM, G., AND ZABBACK, P. 1998. Data partitioning and load balancing in parallel disk systems. *VLDB J. 7*, 1, 48–66.

SCHMIDT, W. J. AND NILSEN, K. D. 1994. Performance of a hardware-assisted real-time garbage collector. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 76–85. (*Operat. Syst. Rev. 28*, 5, ACM SIGOPS).

SCHOEBERL, M. AND PUFFITSCH, W. 2008. Non-blocking object copy for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems*. ACM, New York, 77–84.

SCHUBERT, R. F. 1974. Directions in data base management technology. *Datamation 20*, 9 (Sept), 48–51.

SCHWARZ, P. AND SHOENS, K. 1994. Managing change in the Rufus system. In *Proceedings of the 10th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 170–179.

SEIDL, M. L., WRIGHT, G. M., AND WOLCZKO, M. I. 2008. Method and system for concurrent garbage collection and mutator execution. U.S. patent 7,421,539.

SELINGER, P. G. 1993. Predictions and challenges for database systems in the year 2000. In *Proceedings of the 19th International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 667–675.

SELTZER, M., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. 1993. An implementation of a log-structured file system for UNIX. In *Proceedings of the Winter 1993 USENIX Conference.* USENIX Assoc., Berkeley, 307–326.

SEVERANCE, D. G. AND LOHMAN, G. M. 1976. Differential files: Their application to the maintenance of large databases. *ACM Trans. Datab. Syst. 1*, 3, 256–267. (For more details, see Lohman [1977].)

SHASHA, D. AND GOODMAN, N. 1988. Concurrent search structure algorithms. *ACM Trans. Datab. Syst. 13*, 1, 53–90.

SHNEIDERMAN, B. 1973. Optimum data base reorganization points. *Comm. ACM 16*, 6, 362–365.

SIEBERT, F. 2008. Limits of parallel marking garbage collection. In *Proceedings of the 7th International Symposium on Memory Management (ISMM '08)*. ACM, 21–29.

SILBERSCHATZ, A., STONEBRAKER, M., AND ULLMAN, J. 1991. Database systems: Achievements and opportunities. *Comm. ACM 34*, 10, 110–120.

SINGHAL, V., KAKKAD, S. V., AND WILSON, P. R. 1992. Texas: An efficient, portable persistent store. In *Proceedings of the 5th International Workshop on Persistent Object Systems*, A. Albano and R. Morrison, Eds. Springer-Verlag, New York, 11–33.

SIWIEC, J. E. 1977. A high-performance DB/DC system. *IBM Syst. J. 16*, 2, 169–195.

SKUBISZEWSKI, M. AND VALDURIEZ, P. 1997. Concurrent garbage collection in O_2. In *Proceedings of the 23rd International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 356–365.

SMITH, B. F. 1994. Understanding DB2 RUNSTATS statistics and version 3 enhancements. In *Proceedings of the SHARE 82*. Vol. II (CD-ROM). SHARE, Chicago.

SMITH, G. S. 1990. Online reorganization of key-sequenced tables and files. *Tandem Syst. Rev. 6*, 2, 52–59.

SOCKUT, G. H. 1974. A graphics monitor for animation of dynamic processes. M.S. dissertation., Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge.

SOCKUT, G. H. 1977. Data base performance under concurrent reorganization and usage. Ph.D. dissertation, Division of Applied Sciences, Harvard University, Cambridge. (Also Tech. rep. 12-77, Center for Research in Computing Technology.)

SOCKUT, G. H. 1978. A performance model for computer data-base reorganization performed concurrently with usage. *Oper. Res. 26*, 5, 789–804. (For more details, see Sockut [1977].)

SOCKUT, G. H. 1985. A framework for logical-level changes within data base systems. *Computer 18*, 5, 9–27.

SOCKUT, G. H. AND BEAVIN, T. A. 1998. Interaction between application of a log and maintenance of a table that maps record identifiers during online reorganization of a database. U.S. patent 5,721,915. (Also U.S. Patent 6,026,412, 2000.)

SOCKUT, G. H., BEAVIN, T. A., AND CHANG, C.-C. 1997. A method for on-line reorganization of a database. *IBM Syst. J. 36*, 3, 411–436. Erratum in *37*, 1, 1998, p. 152.

SOCKUT, G. H. AND GOLDBERG, R. P. 1976. Motivation for data base reorganization performed concurrently with usage. Tech. rep. 16-76, Center for Research in Computing Technology, Harvard University, Cambridge, MA. (full article). (Also in Preprints, IEEE-CS Workshop on Operating and Data base Management Systems (*Datab. Engin. 1*, 1, 1977, IEEE-CS Technical Communication on Data Base Engineering), 18-19 (abstract). For more details, see Sockut [1977].)

SOCKUT, G. H. AND GOLDBERG, R. P. 1979. Database reorganization—Principles and practice. *ACM Comput. Surv. 11*, 4, 371–395.

SOCKUT, G. H. AND IYER, B. R. 1996. A survey on online reorganization in IBM products and research. *Data Engin. 19*, 2, 4–11.

SÖDERLUND, L. 1980. A study on concurrent data base reorganization. Ph.D. dissertation. Deparment of Information Processing and Computer Science, University of Stockholm, Sweden.

SÖDERLUND, L. 1981a. Concurrent data base reorganization—Assessment of a powerful technique through modeling. In *Proceedings of the 7th International Conference on Very Large Data Bases*. ACM, New York, 499–509. (For more details, see Söderlund [1980].)

SÖDERLUND, L. 1981b. Evaluation of concurrent physical database reorganization through simulation modeling. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM, New York, 19–32. (*Perform. Eval. Rev. 10*, 3, Fall, SIGMETRICS). (For more details, see Söderlund [1980].)

Srinivasan, J., Das, S., Freiwald, C., Chong, E. I., Jagannath, M., Yalamanchi, A., Krishnan, R., Tran, A.-T., DeFazio, S., and Banerjee, J. 2000. Oracle8i index-organized table and its application to new domains. In *Proceedings of the 26th International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 285–296.

Srinivasan, V. 1992. On-line processing in large-scale transaction systems. Ph.D. dissertation. Computer Sciences Department, University of Wisconsin, Madison. (Also Tech. rep. 1071.)

Srinivasan, V. and Carey, M. J. 1991a. On-line index construction algorithms. Tech. rep. 1008, Computer Sciences Department, University of Wisconsin, Madison. (Presented at 4th International Workshop on High-Performance Transaction Systems; For more details, see Srinivasan [1992].)

Srinivasan, V. and Carey, M. J. 1991b. Performance of B-tree concurrency algorithms. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 416–425. (*SIGMOD Record 20*, 2). (For a revision, see Srinivasan and Carey [1993], For more details, see Srinivasan [1992].)

Srinivasan, V. and Carey, M. J. 1992a. Compensation-based on-line query processing. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 331–340. (*SIGMOD Record 21*, 2). (For more details, see Srinivasan [1992].)

Srinivasan, V. and Carey, M. J. 1992b. Performance of on-line index construction algorithms. In *Proceedings of the 3rd International Conference on Extending Database Technology*, A. Pirotte, C. Delobel, and G. Gottlob, Eds. Lecture Notes in Computer Science, vol. 580. Springer-Verlag, New York, 293–309. (More details appear in Tech. rep. 1047, Computer Sciences Department, University of Wisconsin, Madison, 1991. (For even more details, see Srinivasan [1992].)

Srinivasan, V. and Carey, M. J. 1993. Performance of B^+-tree concurrency control algorithms. *VLDB J. 2*, 4, 361–406. (For more details, see Srinivasan [1992].)

Stanchina, S. and Meyer, M. 2007. Mark-sweep or copying?: A "best of both worlds" algorithm and a hardware-supported real-time implementation. In *Proceedings of the 6th International Symposium on Memory Management (ISMM 2007)*. ACM, New York, 173–182.

Stonebraker, M. 1981. Hypothetical data bases as views. In *Proceedings of the ACM-SIGMOD 1981 International Conference on Management of Data*. ACM, New York, 224–229.

Stonebraker, M. 1989. The case for partial indexes. *SIGMOD Record 18*, 4, 4–11.

Stonebraker, M., Katz, R., Patterson, D., and Ousterhout, J. 1988. The design of XPRS. In *Proceedings of the 14th International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 318–330.

Strickland, J. P., Uhrowczik, P. P., and Watts, V. L. 1982. IMS/VS: An evolving system. *IBM Syst. J. 21*, 4, 490–510.

Subramaniam, M. and Loaiza, J. 2005. Online reorganization and redefinition of relational database tables. U.S. patent 6,965,899.

Sun, X. 2005. Online reorganization in parallel database systems: Scalability, load balancing and high availability. Ph.D. dissertation, College of Computer Science, Northeastern University, Boston.

Sun, X., Wang, R., Salzberg, B., and Zou, C. 2005. Online B-tree merging. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 335–346.

Takahashi, H. and Takahashi, J. 1990. Record-type change method by binary relations. *IBM Tech. Disclosure Bull. 33*, 2, 262–264. (For more details, see Takahashi [1990].)

Takahashi, J. 1990. Hybrid relations for database schema evolution. In *Proceedings of the 14th Annual International Computer Software and Applications Conference (COMPSAC 90)*. IEEE Computer Society Press, Los Alamitos, 465–470.

Tan, L. and Katayama, T. 1990. Meta operations for type management in object-oriented databases: A lazy mechanism for schema evolution. In *Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases*, W. Kim, J.-M. Nicolas, and S. Nishio, Eds. North-Holland, Amsterdam, The Netherlands, 241–258.

Taylor, R. W. and Frank, R. L. 1976. CODASYL data-base management systems. *ACM Comput. Surv. 8*, 1, 67–103.

Tene, G. and Wolf, M. A. 2008. System and method for concurrent compacting self pacing garbage collection using loaded value and access barriers. U.S. patent 7,469,324.

Teng, J. Z. and Todd, J. A. 2002. Method, system, and program for managing file names during the reorganization of a database object. U.S. patent 6,460,048.

Teorey, T. J. and Fry, J. P. 1982. *Design of Database Structures*. Prentice-Hall, Englewood Cliffs.

Trancón y Widemann, B. 2008. A reference-counting garbage collection algorithm for cyclical functional programming. In *Proceedings of the 7th International Symposium on Memory Management (ISMM '08)*. ACM, New York, 71–80.

Tresch, M. and Scholl, M. H. 1993. Schema transformation without database reorganization. *SIGMOD Record 22*, 1, 21–27.

Troisi, J. 1994. Nonstop availability and database configuration operations. *Tandem Syst. Rev. 10*, 3, 18–23.

Troisi, J. 1996. Nonstop SQL/MP availability and database configuration operations. *Data Engin. 19*, 2, 12–18.

Tuel, J. W. G. 1978. Optimum reorganization points for linearly growing files. *ACM Trans. Datab. Syst. 3*, 1, 32–40.

Valduriez, P. 1993. Parallel database systems: Open problems and new issues. *Distrib. Paral. Datab. 1*, 2, 137–165.

Vechev, M. T., Yahav, E., and Bacon, D. F. 2006. Correctness-preserving derivation of concurrent garbage collection algorithms. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. 341–353. (*SIGPLAN Notices 41*, 6).

Vechev, M. T., Yahav, E., Bacon, D. F., and Rinetzky, N. 2007. CGCExplorer: A semi-automated search procedure for provably correct concurrent collectors. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 456–467. (*SIGPLAN Notices 42*, 6).

Versant Object Technology. 1991. *VERSANT System Reference Manual* (for IBM RISC System/6000). Versant Object Technology.

Vingralek, R., Breitbart, Y., and Weikum, G. 1994. Distributed file organization with scalable cost/performance. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 253–264. (*SIGMOD Record 23*, 2). (For a revision, see Breitbart et al. [1996].)

Vingralek, R., Breitbart, Y., and Weikum, G. 1998. SNOWBALL: Scalable storage on networks of workstations with balanced load. *Distrib. Paral. Datab. 6*, 2, 117–156.

Wang, J. and Hu, Y. 2001. PROFS – Performance-oriented data reorganization for log-structured file system on multi-zone disks. In *Proceedings of the 9th International Symposium on Modeling, Analysis, and Simulation of Computers and Telecommunication Systems (MASCOTS 2001)*. IEEE Computer Society Press, Los Alamitos, 285–292.

Wang, J. and Hu, Y. 2002. WOLF – A novel reordering write buffer to boost the performance of log-structured file systems. In *Proceedings of the FAST '02 Conference on File and Storage Technologies*. USENIX Association, Berkeley, 47–60. (For a revision, see Wang and Hu [2003].)

Wang, J. and Hu, Y. 2003. A novel reordering write buffer to improve write performance of log-structured file systems. *IEEE Trans. Comput. 52*, 12, 1559–1572.

Wang, R. Y., Anderson, T. E., and Patterson, D. A. 1999. Virtual log based file systems for a programmable disk. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*. ACM, New York, 29–43. (*Operat. Syst. Rev.*, ACM SIGOPS (For more details, see Tech. rep. UCB/CSD 98/1031, Computer Science Department, University of California, Berkeley, 1998.)

Wegiel, M. and Krintz, C. 2008. The mapping collector: Virtual memory support for generational, parallel, and concurrent compaction. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, 91–102. (*Operat. Syst. Rev. 42*, 2, ACM SIGOPS).

Weikum, G. 1989. Clustering vs. concurrency: A framework and a case study. Tech. rep. ACA-ST-096-89, MCC, Austin.

Weikum, G., Zabback, P., and Scheuermann, P. 1991. Dynamic file allocation in disk arrays. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 406–415. (*SIGMOD Record 20*, 2).

Wiederhold, G. 1983. *Database Design*. McGraw-Hill, New York. 2nd Ed.

Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. 1996. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst. 14*, 1, 108–136.

Wilson, P. R. 1992. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, Y. Bekkers and J. Cohen, Eds. Lecture Notes in Computer Science, vol. 637. Springer-Verlag, New York, 1–42.

Wilson, T. B. 1979. Data base restructuring: Options and obstacles. In *Proceedings of the European Conference on Applied Information Technology (EURO IFIP 79)*, P. A. Samet, Ed. North-Holland, Amsterdam, The Netherlands, 567–573.

Wilson, T. B. 1980. The description and usage of evolving schemas. In *Proceedings of the 4th International Computer Software and Application Conference (COMPSAC '80)*. IEEE Computer Society Press, Los Alamitos, 546–551.

Winslow, L. E. and Lee, J. C. 1975. Optimal choice of data restructuring points. In *Proceedings of the International Conference on Very Large Data Bases*. ACM, New York, 353–363.

Winter. 2005. TopTen program winners. On the web site (http://www.wintercorp.com), search the press releases for the most recent TopTen Program Winners.

Wolczko, M. and Williams, I. 1992. Multi-level garbage collection in a high-performance persistent object system. In *Proceedings of the 5th International Workshop on Persistent Object Systems*, A. Albano and R. Morrison, Eds. Springer-Verlag, New York, 396–418.

Yao, S. B., Das, K. S., and Teorey, T. J. 1976. A dynamic database reorganization algorithm. *ACM Trans. Datab. Syst. 1*, 2, 159–174.

Yong, V.-F., Naughton, J. F., and Yu, J.-B. 1994. Storage reclamation and reorganization in client-server persistent object stores. In *Proceedings of the 10th International Conference on Data Engineering.* IEEE Computer Society Press, Los Alamitos, 120–131.

Zabback, P., Onyuksel, I., Scheuermann, P., and Weikum, G. 1998. Database reorganization in parallel disk arrays with I/O service stealing. *IEEE Trans. Knowl. Data Engin. 10*, 5, 855–858.

Zdonik, S. B. 1986. Maintaining consistency in a database with changing types. *SIGPLAN Notices 21*, 10, 120–127.

Zilio, D. C. 1996. Modeling on-line rebalancing with priorities and executing on parallel database systems. In *Proceedings of the CASCON '96*. IBM Centre for Advanced Studies, Toronto, Ontario, Canada. CD-ROM, (For later, more extensive analysis, see Zilio [1998].)

Zilio, D. C. 1998. Physical database design decision algorithms and concurrent reorganization for parallel database systems. Ph.D. dissertation, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada.

Zou, C. and Salzberg, B. 1996a. Efficiently updating references during on-line reorganization. Tech. rep. NU-CCS-96-08, College of Computer Science, Northeastern University, Boston.

Zou, C. and Salzberg, B. 1996b. On-line reorganization of sparsely-populated B$^+$-trees. In *Proceedings of the 1996 ACM SIGMOD Intl. Conf. Management of Data*. 115–124. (*SIGMOD Record 25*, 2).

Zou, C. and Salzberg, B. 1996c. Towards efficient online database reorganization. *Data Engin. 19*, 2, 33–40.

Zou, C. and Salzberg, B. 1998. Safely and efficiently updating references during on-line reorganization. In *Proceedings of the 24th Annual International Conference on Very Large Data Bases*. Morgan-Kaufmann, San Francisco, 512–522. (For more details, see Zou and Salzberg [1996a].)

Received June 1993; revised July 2000, August 2002, October 2006, July 2007; accepted April 2008