DATABASE ARCHITECTURES

The ongoing tremendous market changes that have led to a global economy have forced designers of modern information systems to adopt innovative computing architectures. The service sector of the economy, which includes companies in the financial services, telecommunications, air transportation, retail trade, health care, banking, and insurance, is a heavy user of such information systems (1). For businesses and organizations, the deployed computing systems as well as the used applications and data constitute their lifeline in today's global market. And, as corporations continuously adapt in an ever-changing business world, they become more dependent on their computing infrastructure.

The increasingly complex information needs of modern organizations and corporations with many geographically dispersed branches can only be met by the use of versatile database architectures. These architectures must harness high-performance computing resources and take advantage of much improved and widely available networking options. Such specialized configurations are deployed in order to help reduce system response times, increase productivity, and enhance throughput rates. In this regard main-memory databases (MMDs) have been developed to service the areas of the economy that call for exceedingly good transaction response times. Client–server systems and databases (CSSs/CSDs) have increased productivity through the use of the existing infrastructure in conjunction with internetworking software. Finally parallel databases (PDBs), built around the notion of tightly coupled computing and storage components, have resulted in systems that demonstrate very high throughput features. Earlier implementations of PDBs were called database machines. Here we examine the requirements, review the salient characteristics, and discuss a number of research issues for the above three families of database systems and their underlying data architectures.

Main-memory databases (MMDs) assume that most, if not all, of the operational data remain in volatile memory at all times. Disk–resident database copies are mostly used to recover from either a disaster or an accident (2). There exist a large number of applications in the service sector that call for MMD support in order to function according to predefined tight performance requirements. Environments where such applications are commonplace include securities trading, money market transaction systems, and telecommunication systems. In the financial area, transactions need to complete in real-time, and this can be achieved only if the underlying database system avoids long delays caused by interaction with mechanical parts. Furnishing ultrafast data access and transaction processing in the above environments is only possible if the deployed data architectures avoid interaction with external storage devices (i.e., disks). Accessing main-memory resident data is in the order of nanoseconds, while accessing disk-based data requires possibly tens of milliseconds. Along the same lines a customer of a telephone company desires that an 800-call be completed within acceptable time constraints. The size of the customer base and the volume of companies, organizations, and even individuals who carry such toll-free numbers has become excessively large. Therefore the provision of effective MMDs for the satisfaction of such user requirements is a major concern and a challenging technical task.

There are a number of key differences between MMDs and conventional database systems. In MMDs, access structures can facilitate the retrieval of data items by traversing and checking memory locations, while in disk-based databases, most of the retrieval process is centered around input/output (I/O) operations. In a disk-based system, the costs of disk–access can be amortized by clustering data so that pages can be accessed sequentially, while in MMDs, data are often fetched randomly. Finally memory banks are volatile and cannot maintain their stored information if there is a disruption of power. Although it is possible to use nonvolatile memories, such an option is considered to be not cost-effective.

In client–server computing environments, a number of client processes typically running on small machines (i.e., desktops, laptops) interact with one or more server processes using an underlying interprocess communication system. This interaction is inherently recursive in nature, since a server may become the client of another service site, and it has resulted in integrated systems that allow for distributed access of data, computing, and presentation of results. Windowing systems are often run on the client sites, allowing for easy interface with application packages as well as querying of data. The latter can be done by using standard query languages such as SQL or specialized data-exchange protocols between clients and data sources. Interprocess communication abstractions are used to provide the transport layer among the various sites involved. Once clients have obtained their desired data/results, they can choose to immediately use these data or/and cache them for further analysis and future reuse.

Server processes typically offer services that range from simple file system request handling and provision of CPU-intensive computation to complicated information retrieval and database management. Indeed, a client may independently request services from more than one server at the same time. Servers continuously monitor ("listen" to) the network for incoming requests and respond to those received from clients by providing the required service. Servers attempt to satisfy incoming client requests by creating and executing concurrent tasks. The application programmatic interface of servers hides their internal functionality and organization, as well as the idiosyncrasies of the operating systems and hardware platforms used. Hence servers can not only be providers of services but also repositories of programs, managers of data, and sources for information and knowledge dissemination.

The wide availability of multiple-processor computers offers opportunities for parallel database systems that demonstrate substantially improved throughput rates. Since future databases will have to manage diverse data types that include multimedia such as images, video clips, and sounds, they should be able to efficiently access and manipulate high volumes of data. The projected volume increase of today's databases will only be possible to handle through the use of multiprocessor and parallel database architectures. Such architectures could also be used in conjunction (undertaking the role of specialized servers) with client–server configurations in order to bring to the desktop unmatched CPU and storage capabilities.

Parallel database architectures can partially address the I/O bottleneck problem that ultimately appears in all centralized systems. Instead of having the actual data reside in a few large devices, parallel database architectures advocate an increase in parallel data transfers from many small(er) disks. Working in conjunction with different parallel I/O buses, such disks can help diminish the average access time as long as data requests can be fragmented into smaller ones that can be serviced in a parallel fashion. Two possible mechanisms used to increase performance rates in such systems are intraoperation and interquery parallelism. The former allows for the decomposition of a large job into identifiable smaller pieces that can be carried out by a group of independent processors and/or I/O units; the latter enables the simultaneous execution of multiple queries. Parallel databases can also be classified in terms of their degree of parallelism: coarse or fine granularity. In coarse granularity parallelism, there is a small number of processors per system (often two or four) coupled with a few (less than five) disk units. A fine granularity parallel system may contain tens or even hundreds of processing elements and I/O devices.

In this article we discuss the specific requirements and examine the key features of the above three database architectures. We discuss issues related to data organization and representation, query processing and optimization, caching and concurrency control, transaction handling and recovery. We then discuss main-memory databases, client–server, and parallel databases. The article ends with a summary.

MAIN-MEMORY DATABASES

Main-memory databases (MMDs) feature all the conventional elements that one would expect in a database system, namely data organization, access methods, concurrency and deadlock management, query processing and optimization, commit protocols and recovery. In standard database systems most of these operations and functionalities are designed around the movement of data blocks/pages in the memory hierarchy. In an MMD the fundamental difference is that its components are designed to take advantage of the fact that data do not need to be transferred from disks. Schemes for data organizations in MMDs are of major importance. In this direction, data swizzling is an important step: As soon as a (complex) data item is retrieved from the disk to main-memory, applications can access it through a "direct" pointer. Along the same lines, while conventional query optimizers try to minimize the number of accessed blocks, MMDs attempt to optimize their query processing tasks by reducing the CPU cycles spent on each task. Finally, commit and logging protocols in MMDs have to be designed carefully so that they do not create unnecessary bottleneck points.

The main point of concern for MMDs is that either a crash or an unexpected power outage may disrupt mission critical operations. Unlike disks, memories become oblivious of their contents once power is lost. Therefore it is absolutely critical that frequent backups are taken so that the integrity of data can be guaranteed at all times. Naturally memory banks with uninterruptible power supply (UPS) can be used to keep the memory afloat for some time even after a disruption of power occurs. However, these types of services are not inexpensive, and they may also suffer from overheating. In light of the above, a MMD should be developed in a way that trades off the consistency between the in-core data and the disk–resident data with the overhead required for continuous backups.

If one considers the universality of the 80%–20% rule, then it is evident that the whole database does not need to be in main-memory. Actually only the hot parts of the data can remain in-core, while the less frequently accessed items can be disk-based. The distinction between hot and cold(er) parts of databases is, in a way, natural. For instance, the values of traded securities have to be always maintained in main-memory, whereas background information about corporations and their operations need not.

Organization of MMD Components

Memory Data Representation and Organization. Issues related to MMD data layout and management have been partially addressed in the development of conventional databases, specifically in the development of system catalogs. Objects in such catalogs have to be handled in a very different way than their disk-based counterparts; these subsystems are organized so that optimal times are achieved in terms of access and response times. To maintain this type of fast interaction, their development is centered around variable length structures that use mostly pointers to the memory heap.

Tuples, objects, and many other types of data items when they are disk–resident can be accessed through "object identifiers" (OIDs). The task of a database system is essentially to translate an OID to the address of a block/page. Once the item in discussion is brought into main-memory, accessing is typically facilitated by a hash table that maps the OID to an address in main memory. When an application references an object (in the "shared" database buffer space), a copying operation has to be carried out. This copying operation brings the object into the address space of the application and is carried out with the help of an interprocess communication mechanism. Thus there is a nonnegligible penalty involved in carrying out the above "conversion" in address space every time there is a reference to an object. Instead of performing the above steps, what modern systems tend to do is to "swizzle" database objects (3).

In swizzling, disk-based object layouts, such as tuples of certain constant length and representation, are transformed into strings of variable length. User applications are provided with access to these variable length strings through direct pointers. The key performance question in swizzling is to decide whether it is profitable to convert OID references to objects in main-memory, with direct pointers. Moreover there is a certain cost to be considered when swizzled data have to be stored back on the long-term memory device, since the reverse process has to take place (i.e., objects have to be unswizzled). Unswizzling is done during the save phase of the object access operation.

For operations that involve OIDs and are computationally intensive, there are numerous options that a system designer could pursue. The success of these options depends on the types of operations and the composition of the workloads that the MMD receives. In particular, objects brought into main memory could be simply copied, swizzled in place, or copy-

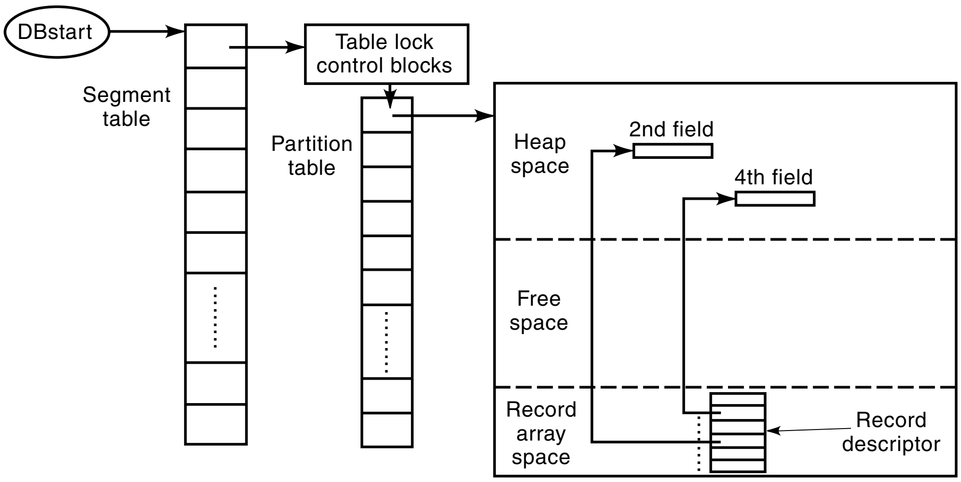

Figure 1. Starburst's main-memory management.

swizzled. Copy-swizzling allows the image of the object in the MMD–buffer to remain intact. In-place swizzling avoids making an extra copy of the object in main-memory and therefore reduces the CPU costs involved. There is a trade-off between the CPU–overhead savings and the overhead required to unswizzle objects before they are flushed into the disk–manager. On the other hand, copy-swizzling may present some savings as only modified objects need to be unswizzled before they are written out to the long-term memory. Also, depending on the way objects are brought into main-memory, swizzling can be either eager or lazy. Although the cost of swizzling may at first appear small, it is evident that if thousands of objects are accessed at the same time, then there might be significant cumulative penalties. On the other hand, if swizzled-pointers to objects are dereferenced more than once, then the benefits of swizzling can be sizable (4).

The organization of MMDs is an area of prime concern because data have to be always accessed in an efficient manner. In Starburst there is a dedicated main-memory database component termed MMM (5), which does not use swizzling and attempts to optimize access to the data using T-trees (6). The key main-memory element of MMM is a partition which is a fixed size unit of memory allocation (Fig. 1). In spirit, MMM partitions are very similar to pages, they are only different in terms of their sizes, which range from 64 K to 256 K. Partitions are dynamically allocated, and they constitute the main unit of recovery. Partitions are clustered together into segments. Segments are areas of memory whose sizes (i.e., sum of partitions) are variable.

Records are identified by record identifiers (RIDs) which consist of three parts: segment number, partition number, and offset within the partition. The fields of a record are heap–resident. They can be addressed through an array of pointers (i.e., the *record descriptor*). The record descriptor provides the means for representing data tuples in the context of a Starburst partition. If the number of attributes of a tuple changes, then a special tail structure is used. This tail structure extends the record representation in the heap.

Accessing a specific record is facilitated by using the corresponding RID to identify both segment and partition within the overall main-memory structure. Once inside the partition, then the offset is used to reach the record's slot. The slot is essentially a descriptor/translation mechanism to get to the various fields of the record in the heap area. Before values of the various fields are used by applications, they have to be copied over into the applications' space. By keeping all the storage structures in main memory, the path length of accessing a data item becomes much shorter as compared to a disk-based database organization.

Continuous additions and modifications of tuple attributes will ultimately require space that is not currently available in the partition. In this case the newly expanded tuple will have to be physically moved into another partition. Such a movement could be easily accommodated as long as there are no references to the augmented record. Tombstones are used in this context in order to avoid undesirable lost references. As expected, tombstones augment the path length of the execution as references go through an additional cycle in order to detect possible encounter of tombstones, and there is some space overhead as well. A possible way to overcome the disadvantages of tombstoning is to assume that field pointers can span across partitions.

The administration of the partition space is done by adopting a scheme where four partition classes are introduced in terms of available capacity: those with available capacity up to 500 bytes, those with 500 or more bytes available, partitions with 2000 or more bytes free, and finally partitions with more than 10,000 bytes of free space. A partition may belong to one or more such classes. Depending on the degree of the expected growth of the record(s), a suitable data partition is selected to place a record in. If there is no space available in the current partition, then a new partition is allocated.

The Dalí main-memory manager (7) exploits the idea of memory mapped I/O. Specifically, most Unix implementations offer the system call *mmap( )*. Memory-mapped I/O allows the system to map disk–resident files in main-memory buffers. Once the mapping has been carried out, reading of bytes from the buffer automatically corresponds to fetching the corresponding data from the disk file. In the same fashion, whenever data are stored/set in this buffer area, the corresponding modified bytes are written back to the disk file. A file can be memory-mapped by many processes. If a file is memory-mapped to a shared virtual memory area, then Dalí multiple-users are provided with access to a file with sequential consistency guarantees. Consequently Dalí advocates that

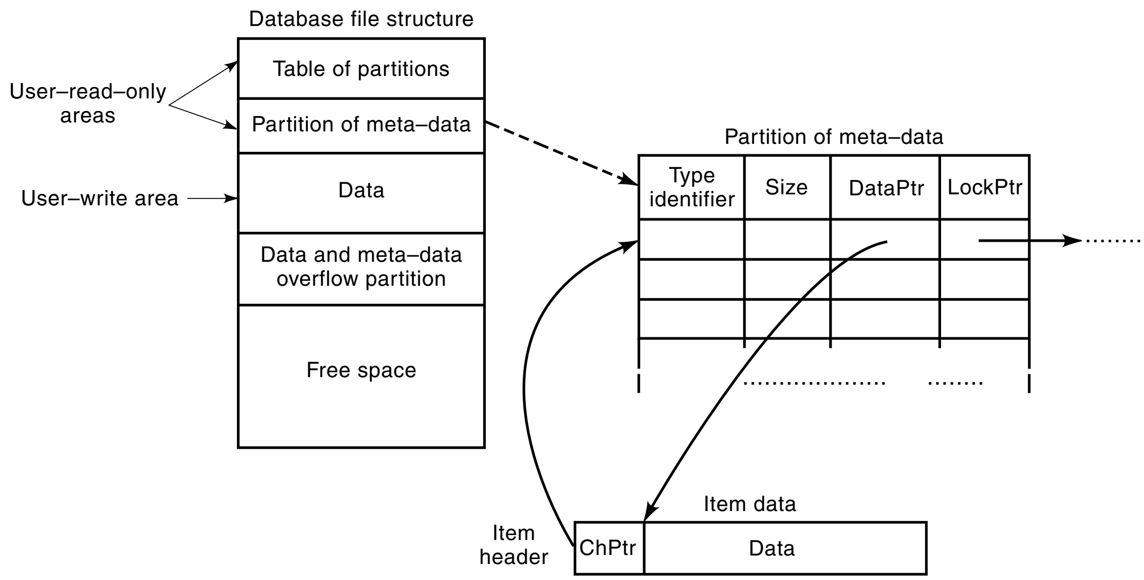

Figure 2. Dalí database file organization.

MMDs be organized in distinct "database" files with each file containing related data.

Figure 2 depicts the organization of a Dalí database file. The space of a file is classified into areas (or partitions) whose functionality is fundamentally different from those of Starburst. The "partition table" indicates the borders of these areas, and it is super-user writable only. The *descriptors* of the various individual database file items are provided by the "meta-data partition." The structure of this partition is shown in the right-hand side of Fig. 2. Data–pointers are used to point to data items that reside in the data partition. The latter is a user-writable area, since individual processes can modify the content of data objects. The free and overflow areas of a file are used when there is need for data and meta-data space, respectively. Naturally the cost of mapping database-pointers—as the ones just mentioned—to virtual memory addresses could be reduced by swizzling. However, Dalí does not provide this feature, since it would complicate the implementation of its concurrency schemes (7).

The internal data representation is not the only core issue in MMDs that needs to be considered. Different indexing schemes have to be used as well. Although the B+-tree structure is one of the most acceptable indexing options for conventional disk-bound operations, it loses some of its appeal when it comes to main-memory resident data. Instead, AVL trees can be used, since they offer elegant rebalancing operations in the light of updates, and logarithmic access times (8). T-trees (6) have been designed for main-memory databases and the utilization of their node space is user-specified. They also exploit pointers to traverse the tree structure fast. Other structures such as BB-trees, skip–lists, and deterministic skip–lists can be used efficiently to access data in memory (9). An additional advantage of all these structures is that the key values do not need to be part of the internal node. Instead, a pointer or a record ID can be used to point to the required key value. Most of the methods above can offer range–queries through minor extensions.

Query Processing. The fact that data are resident in main-memory has ramifications on the way query processing is carried out. While in traditional query processing the dominant cost is that of the involved disk I/O operations, the CPU computation cost becomes a major factor in MMDs. Therefore approaches based on CPU-cost optimization for query processing have been suggested (2,10,11). However, modeling CPU costs is not an always easy task. Costs may vary substantially depending on the hardware platform, the style of programs that carry out the operations, and the overall software design (12). In addition there are interesting trade-offs between the amount of CPU processing required and the memory buffer space reserved for indexing purposes.

In conventional query optimization, there have been numerous efforts to efficiently process queries—and in particular joins—by preprocessing one (or more) of the participating relations. For instance, ordering both relations by their joining attribute offers significant savings. In MMDs such approaches lose most of their appeal, since the traversal of pointers provides very fast access. Sorting relations, before the eventual join is performed, may not be a reasonable option because it can impose additional and unnecessary overheads in terms of CPU-processing and space used. Instead, the outer relation can be traversed sequentially, and the joining attribute value can be used to access the appropriate joining tuples from the inner relation (12). This access is facilitated by the traversal of navigational pointers provided by the MMDs, as mentioned earlier in the context of Dalí and Starburst. Hence the sort–merge approach is not used for join processing in main-memory databases. Further it not only requires extra space to accommodate pointers that denote the sorted order of relations but also CPU time to carry out the actual sorting (10). A number of elegant algorithms used to join relations and/or views by exploiting pointers are discussed in Ref. 13.

A query optimizer that has been specifically developed for a main-memory database was presented in Ref. 10. The approach followed here is geared toward minimizing the number of predicate evaluations. Minimum CPU costs incurred in predicate evaluation determine viable access plans. In addition a branch-and-bound methodology is used to prune the search space during the query processing phase. In trying to build a realistic model, Ref. 10 proposes to identify system bottlenecks that correspond to the pieces of database code that take up most of the CPU processing time in the context of a query. The optimization phase is based on these costs. The costs of such high overhead operations are determined by using profiling techniques and program execution analyzers. In Ref. 10 five specific cost factors have been identified:

1. Cost for evaluating predicate expressions
2. Cost for comparison operations
3. Cost for retrieving a memory-resident tuple
4. Unit cost for creating an index (unit refers to the cost per indexed item)
5. Unit cost for sorting (penalty per sorted item)

Since queries are expressed here in canonical form, these factors are sufficient to model the overall costs required by various materialization plans. Among these five cost factors, Ref. 10 experimentally verified that the first one is the most expensive of all. In fact the first cost is tenfold more expensive than each of the other four factors listed above. This is because the entire predicate tree structure has to be traversed in order to obtain a single evaluation. Since such tree structures can accommodate general forms of predicates, they can lead to expensive evaluation phases.

The query optimizer uses a number of strategies to produce the lowest-cost estimates, namely

1. Evaluation of predicates at the earliest possible opportunity
2. Avoidance of useless predicate or expression evaluation whenever possible
3. Binding of elements as early as possible

The branch-and-bound algorithm used is equivalent to an exhaustive search; however, it prunes subtrees for which there is a strong indication that the optimal solution will not be found even if the search were continued inside these subtrees. This indication can be derived by comparing a continuously maintained global lower bound of the cost with the anticipated cost if a specific subtree is followed.

Concurrency Control. Data items are easily accessible in MMDs, so transactions may have an opportunity to complete much faster, since extreme contention conditions are not expected to develop often. Coarse granularity locking has been suggested as a sufficient option for concurrent MMD operations. However, some long-running transactions may suffer from starvation and/or lengthy delays. Therefore a more flexible technique can be useful here. For instance, a protocol that is capable of adapting from coarse to fine granularity locking whenever necessary could be beneficial.

System designers of MMDs may also avoid overheads by circumventing operations to an independent lock manager. In traditional databases, lock managers are organized around a hashing table. This hash table maintains information about the way that the various data objects are locked at any time. In MMDs this locking mechanism can be adapted and possibly optimized so that the overhead required to access the hashing table is eliminated. This optimization can be

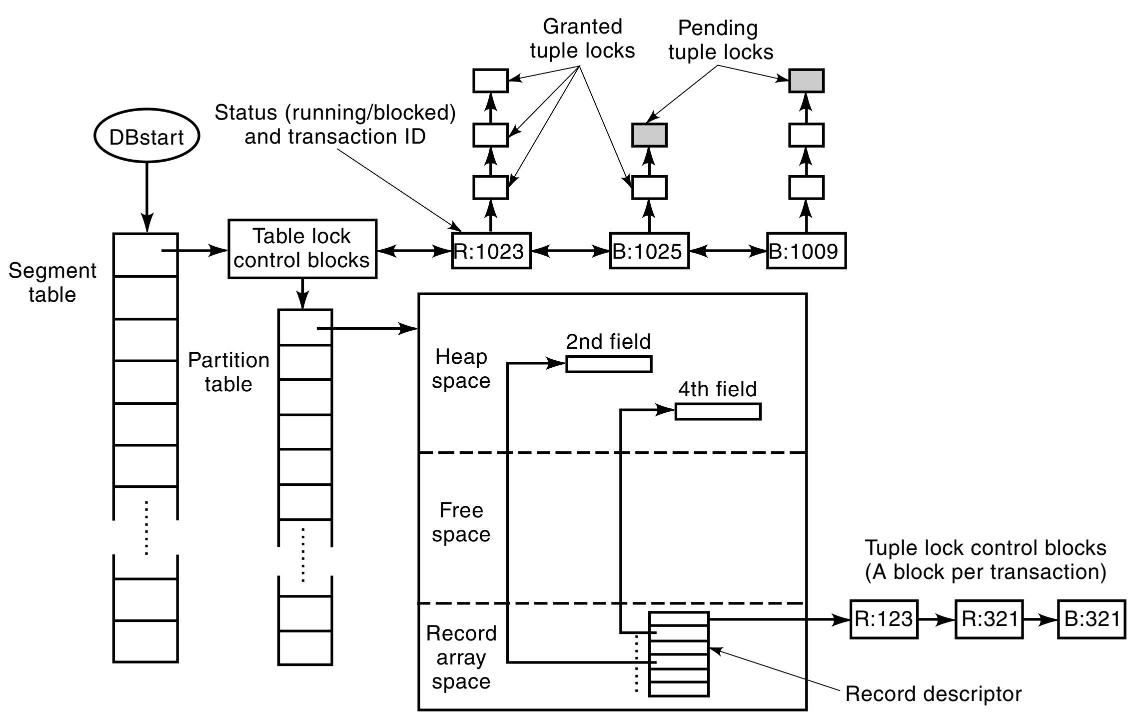

Figure 3. Starburst's main-memory management and concurrency structures.

achieved by attaching the locking information to the actual data.

Both of the above ideas have been implemented in the Starburst main-memory manager (MMM) (14). Figure 3 shows the key data structures used augmented with the supporting locking mechanisms. Each segment maintains a control block that includes the pertinent lock information about the segment in question. Every transaction that attempts to get a lock on the table receives a *table lock control block* that provides the type of lock as well as a list of tuple–locks encountered so far. If tuple–locks are not compatible with the aggregate lock type of the table, then they are kept pending, and the requesting processes are blocked. For instance, Fig. 3 indicates that transaction 1023 has successfully locked the table and is working with three specific tuples. However, transaction 1009, which initially locked the table in a manner compatible to 1023 (and 1025), subsequently requested a non-compatible tuple lock and is currently blocked.

When such contention for data items appears, data tuples can be locked individually. This action will almost certainly increase concurrent sharing. Thus Starburst's MMM is capable of featuring a list of *tuple–lock control blocks* per tuple. Tuple–lock control blocks indicate which processes have accessed specific tuples, and how. In Fig. 3 such a list of control blocks is attached to the descriptor of the record.

A granularity flag is always maintained at the table level (i.e., segment control block) and indicates whether table or tuple locking granularity is in use. Starburst's MMM has the ability to escalate and de-escalate locks so that the level of concurrency can be adjusted. Since table locking is generally inexpensive (carries low overhead), it is the preferred method for low-sharing situations. However, as more transactions accessing the same table become active, the MMM de-escalates the table lock to individual tuple-level ones and the degree of data sharing increases. De-escalation is possible only if the transaction holding the table lock is capable of "remembering" the individual tuple–lock requests up to this point. This is the reason why, besides the locks on segments, the segment control block keeps a record of all the requested (and whether granted or blocked) locks on tuples so far. The tuple–lock control blocks (as shown in Fig. 3) indicate the transactions that have acquired shared access on specific tuples (e.g., transactions with IDs 123 and 312) as well as transactions that are currently blocked (i.e., transaction 231). As soon as de-escalation occurs, the lock-related structure at the segment level is de-activated. Escalation back to table locking occurs when the need for increased data sharing ceases to exist.

In Ref. 12 an alternative way to process exclusive-only concurrent requests is outlined. In this, two bits per object are used to realize concurrency control. If the first bit is set then an object is locked and is unavailable. If an object is locked and the second bit is set as well, it means that one or more transactions are waiting for the object to become available. The set of transaction identifiers waiting for a lock on an object are stored in a hash table. When a finishing transaction resets the first bit, it also checks the status of the second. If the latter is set, then the terminating transaction has to wake up one of the waiting transactions. The last transaction to be waked up needs to clean up the second bit. The benefits of such a scheme rest with the fact that often in MMDs, records are locked for a short period of time and are released soon after the update. If there is no need to access the hash table frequently, this technique presents an acceptable locking alternative. System M (15) features an exclusive/shared locking scheme with conversion capability from shared to exclusive mode at the segment level (set of records).

Logging and Commit Protocols. Logging is mandatory because the MMD should be able to avoid lost data and/or transactions due to media failure. Since logging is the only operation that has to deal with an external device in MDDs, it can become a bottleneck that may adversely affect system throughput. A number of solutions have been suggested to solve this problem; they are based around the concept of a stable main-memory space (2,11,15–17). Whenever a transaction is ready to commit, the transaction writes its changes into stable memory (nonvolatile RAM). Stable memory is often used to "carry" the transaction log and can greatly assist in decoupling persistence from atomicity. Writing to such a stable log is a fast operation, since it is equivalent to a memory-to-memory copy. Once many log entries accumulate, a special process [or processor as in System M (15)] can be used to flush log data to the disk unit. What stable memory really achieves is that it helps keep response times short because transactions do not have to wait long for the log operations to complete. In Ref. 2 it has been suggested that a small amount of stable memory can be as effective as a large one. The rationale is that a small stable buffer space can effectively maintain the tail of the database log at all times.

When stable memory is unavailable, group committing can be used to help relieve the potential log bottleneck (2,15,18). Group commit does not send entries to the disk-based log indiscriminately and on demand as a traditional write-ahead log would normally do. Instead, log records are allowed to accumulate in main-memory buffers. When a page of such log entries is full, it is moved to the log–disk in a single operation. The rationale behind group commit is to diminish the number of disk I/Os required to log committed transactions and amortize the cost of disk I/O over multiple transactions. Precommitting also works in the direction of improving response times because it releases locks as soon as a log entry is made in the main-memory log (2,18). This scheme allows newer transactions to compete for locks and data objects while others are committing.

In Ref. 17 a protocol for commitment is provided that reduces the size of the logging operations by flushing into the disk only redo entries. Undo records are kept in main-memory and are discarded as soon as a transaction has committed successfully to either the disk or a stable area. This action economizes on the log volume and so furnishes a short(er) recovery phase, since the MMD requires only a single log pass. In this scheme the MMD maintains a redo-log on the disk where only the redo entries of committed transactions reside. To achieve this, every active transaction maintains two distinct log areas (for redo and undo entries) in main-memory (Fig. 4). When the commit entry of a transaction ultimately reaches the persistent log (located on either disk or stable RAM), the transaction commits. The novel feature of the commit protocol discussed in Ref. 17 mostly rests with the way that the termination of transactions is handled. There are three distinct phases in the commitment protocol:

1. *Precommit Phase.* A completed transaction T_i is assigned a commit sequence number (*csn*), releases all its

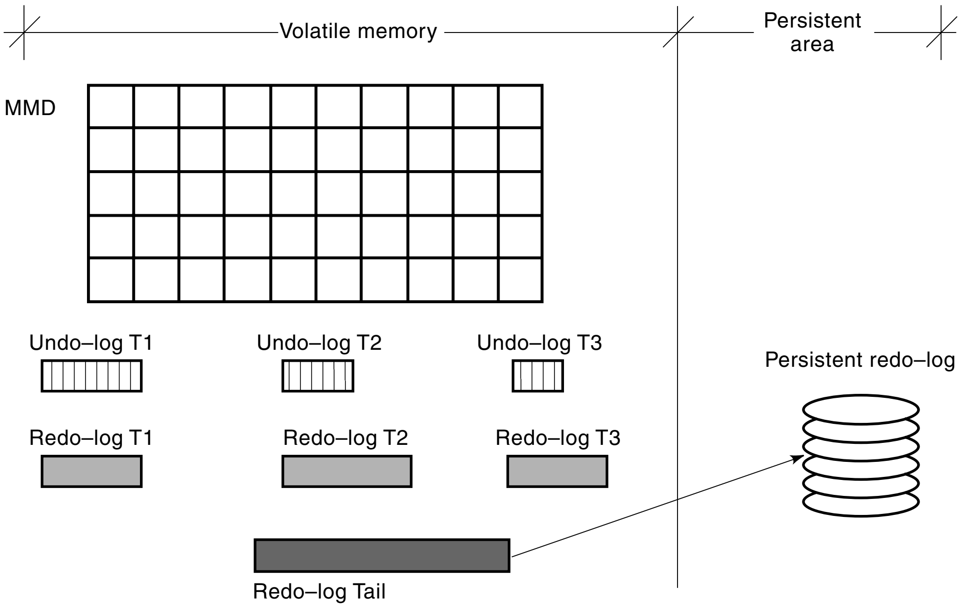

Figure 4. Logs for individual transactions and the global redo–log.

locks and writes an entry $\ll csn, T_i \gg$ to the private redo–log of T_i. This private redo–log is appended to the global redo–log kept by the MMD.

2. *Actual Commitment.* The commit entry of the transaction reaches persistent storage.
3. *Postcommit Phase.* The user–process that instigated the transaction is notified of the completion; the transaction is removed from the list of the active transactions and its volatile undo log is discarded.

The usage of individual redo–logs diminishes the contention for the global log as well as the size of the global log's tail. Transactions that have not completed their commit protocol and need to abort can do so by traversing the undo entries of their own logs in reverse.

Recovery and Efficient Reloading of Data. Check–pointing is often used as the means to reduce the length of recovery once a MMD fails and data have to be restored from the disk–image of the database and the system log. Actually in MMDs, check–pointing and recovery are the only points at which the disk–resident database is accessed. One way to minimize the overheads of check–pointing is to use large-sized blocks so that writing to the external device is more efficient (12).

When a crash takes place, reloading of the database has to be performed. The MMD may experience undesirably long delays if the system is brought up by reloading a large collection of data. Therefore effective reloading techniques are important. In particular, on-demand schemes offer an obvious advantage as transaction processing may restart with the availability of only a small amount of important data in memory. In Refs. 19 and 20, a number of such techniques are introduced, and their behavior is compared (through experimentation) with ordered-reload. Ordered-reload refers to the process of reading data from the archived database sequentially. Its advantage is that the actual reload process lasts for the shortest possible time and presents no additional space and/or CPU overhead later.

More elaborate reloading algorithms attempt to place in main-memory a selected set of pages that will enable the MMD to become operational immediately (19). Such algorithms include reload with prioritization, smart, and frequency reloading. In reload with prioritization, pages are brought into main-memory on-demand according to a predetermined priority scheme, and the MMD resumes normal transaction processing once a prespecified percentage of the database is in place. The smart algorithm is essentially reload with prioritization but uses page prefetching (instead of on-demand paging). In the frequency-reload algorithm, pages are stored in the archival memory according to their frequency of access observed so far. This is facilitated by a specialized disk-based structure that helps classify the various data elements according to their frequency indicators. Using this structure, the frequency-reload algorithm brings pages with higher access frequency counts into memory first. Assuming that frequencies of data page accesses do not change very often, frequency-reload produces good response times and satisfactory reloading times.

CLIENT–SERVER DATABASES

The client–server paradigm has been in use for several years in areas other than database management systems. It is widely used in multitasking operating systems for the provision of various system services such as print spooling. The advent of internetworking has allowed this model to be extended to distributed services such as electronic mail, file transfer, remote login, and even networked file systems (21,22).

In most multiuser computing systems, the data reside at one or more central nodes. With the help of their terminals and/or personal workstations, individual users (clients) access the data from centralized systems (servers) using telephone or other communication lines. When such aggregates involve databases, they are often termed client–server databases (CSDs). In CSDs the interaction among users and data-providing sites occurs mainly in two ways: query–shipping and data–shipping. In pure query–shipping settings, clients dispatch user–queries and updates to the database server(s) and receive the results of their operations. In data–shipping, the client machines request the required set of data objects/pages from the server(s) and perform the necessary processing of the data locally.

In both ways of interaction, there is a straightforward optimization to be found. By storing either data or results received from servers locally, clients may possibly eliminate or reduce the need for future interaction with the server database. The maintenance of such "remote" data is known as data caching. Data caching has been used as a vehicle to achieve scalable performance in CSDs in the presence of large number of clients attached per server. The greatest benefits of data caching are as follows:

- Redundant requests for the same data originating at the same client can be eliminated. This makes such communication between the user machine and the database server unnecessary and significantly improves response times for requests on the cached data.
- Once server-data are locally available, clients can use their own computing resources to process them and furnish the query results to the users. In this manner clients can off-load work from the database server(s). This feature has gained importance as client workstations have become increasingly more powerful.

However, with these benefits come several cost/consistency trade-off issues. Whenever cached data are updated at the owner site, the new value must be propagated to the copies. This propagation cost can be significant. For frequently changing data, the cost of propagating the updated data values to the cache sites can outweigh the gains of caching the data. Another consideration is in the context of client–server databases where the data cached at the clients is updated by transactions. Here the concern is not only with data consistency but with data recovery in case of client– or database server crashes.

Basic Client–Server Database Architecture

Directly applied to databases, the basic CS architecture differs very slightly from that found in operating systems. The principal components of the system are a server, which runs the full database management system; the client, which acts as an interface between applications on a remote processor; and the DBMS. Interaction between the client and server is purely on the basis of queries and results. The client application sends a query to the server as a result of user interaction. This query is transported on a local or wide-area network by some form of message–passing or remote procedure call mechanism to the server. The server receives the query, executes it, and sends the result back to the client using the same communication mechanism. The client application processes the results of the query in a naive fashion, such that should the same data be required again, it must be re-fetched from the server. Figure 5 depicts the configuration of this architecture.

There is little difference between this mode of operation and that used in a time-sharing system, except for the ability of the client application to format the results so that they are better suited for the end-user's consumption. This is the approach taken by the "SQL server" applications commonly available in the market today. Apart from improved presentation capabilities, another more important reason for the adoption of this strategy is that the server is no longer burdened with tasks related to application processing. As a result it is possible to achieve improved performance rates (throughput and response time) than in the basic time-sharing system. The usefulness of a database lies in its ability to store and manage data for future retrieval, functions which inherently make its operations disk-intensive.

Unfortunately, data access times of secondary storage devices lag at least two orders of magnitude behind those of CPU and primary memory, and hence, I/O operations on the server disk remain a major stumbling block in the improvement of overall system performance of a client–server architecture. This was confirmed in Ref. 23. It was also pointed out that although database retrieval operations are not as CPU-intensive as application processing, the basic client–server architecture suffers serious degradation of performance when a large number of "active" clients are attached per server.

A natural extension to the basic architecture, which attempts to overcome the I/O bottleneck, is the use of several disks, accessible in parallel, at the server. A query received by the server is fielded by the disk that holds the relevant data. By this method the response time is improved. Data are distributed among the disks in a manner that ensures that similar loads are imposed on each of them. This can be

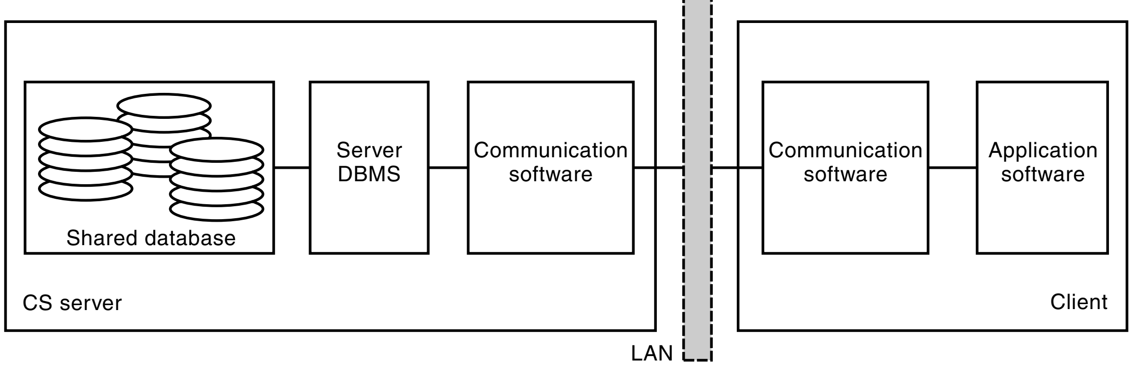

Figure 5. Basic client–server database architecture.

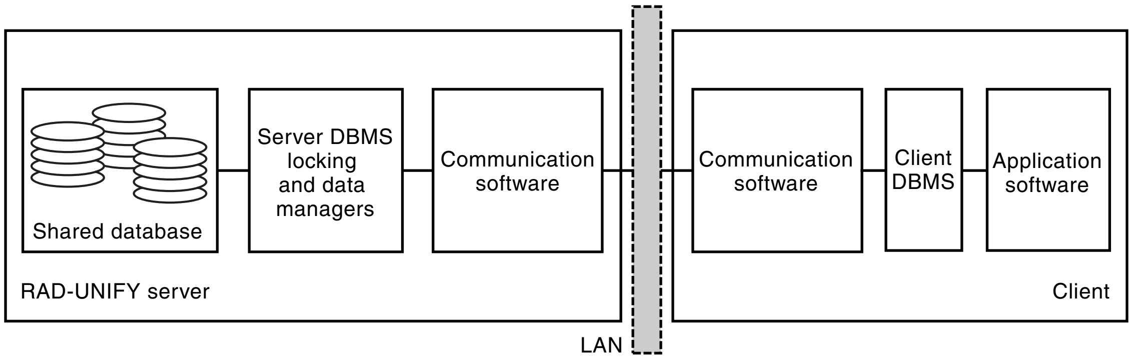

Figure 6. RAD-UNIFY CSD architecture.

achieved by using some load-balancing algorithm, disk striping, or any other scheme similar to those used in distributed database systems (24). Experiments performed on the fully replicated case (23) show this variation to be an improvement on the basic architecture, especially under circumstances where the number of clients is limited. Now the disk that is currently under the least load can field the request for data. However, this architecture still suffers from scalability problems. Other disadvantages in the use of this configuration include the cost of propagation of updates to all the disks. This could be alleviated by the use of a variant of a primary-copy commit mechanism at the cost of reduced concurrency (i.e., all transactions that are interested in a recently updated data item must block until the new data value has been safely forced to secondary storage). The monetary cost of multiple disks is a major concern as well.

RAD-UNIFY Client–Server Database Architecture

Rather than attempting to improve server performance by introducing parallelism, the RAD-UNIFY client–server architecture (25) further reduces demands on the server. This is achieved by moving a significant portion of the database server functionality to the client site. The rationale here is to exploit both the client CPU and primary memory. The client maintains the query-processing and optimization components of the database, while the server retains the data as well as the concurrency control and buffer managers. Interaction between clients and servers takes place at a low level, since only messages and data pages are transported between them. The client "stages" these data pages in its own memory space. Subsequently the query processor running on the local CPU refers to these staged pages to generate the result(s) for the client application/query. The usage of client buffer space to hold a portion of the server database has proved to be a basic yet effective form of caching (25–28). This caching plays a central role in the improvement of performance rates of the architecture (28) as compared to those achieved by the basic CS configuration. Figure 6 shows the functional components of the architecture in discussion.

By allowing the contents of the client memory to remain valid across transactions (intertransaction caching), it is possible to reduce the load on the server on the assumption that data may be held locally. The immediate benefit of this method is that the server may be accessed less frequently if the query patterns are such that locally cached data are relevant to most of a particular client's application requirements. Locality of data access improves response time and the reduction of both I/O and CPU processing demands on the server translates directly into improved system scalability. The RAD-UNIFY model of client–server databases is a popular architecture in the development of object-oriented databases because it simplifies the development of the server.

Enhanced Client–Server Database Architecture

The next step in improving CSD performance is to attempt to increase the locality of data accesses by using the client workstations' disk resources. The obvious approach would be to extend the RAD-UNIFY architecture to use the client disk as an extension of primary memory. While this could be performed automatically as part of the operating system's virtual memory functionality, the DBMS's specialized buffer–management techniques are better suited to the task of maintaining this disk cache. This is the approach taken in the enhanced client–server (ECS) architectures proposed by Refs. 23 and 29. Figure 7 shows the main components of the ECS architecture.

The client site now runs a simplified implementation of the DBMS which features query processing, disk storage, and buffer managers on its own. The use of the disk resource allows a larger amount of data to be staged at the client disk–cache, further increasing the locality of data access and consequently reducing response times. If the disk–caches are large enough and update frequency is low, or conflicting transactions are uncommon, this architecture is shown to improve overall system performance almost linearly with the number of clients attached per server. Once client disk–caches contain the data relevant to the client's work, the server only needs to deal with update requests and their propagation to pertinent sites. Client caches can be built using incremental techniques and maintained by methods of either replacement or merging of data. As the number of updates increases, the degree of conflict increases as well. Therefore the performance of the aggregate system becomes tied to the server's ability to cope with the tasks of maintaining data consistency, update propagation, and concurrency control.

Deppisch and Obermeit (30) propose a checkout system that uses local disks for data storage suitable for environments where most transactions are of a long duration. The proposed architecture involves "multi-level" cooperation between clients and server(s). Large objects are frequently extracted in their entirety from the server database for manipulation on a client workstation. Client queries are exchanged

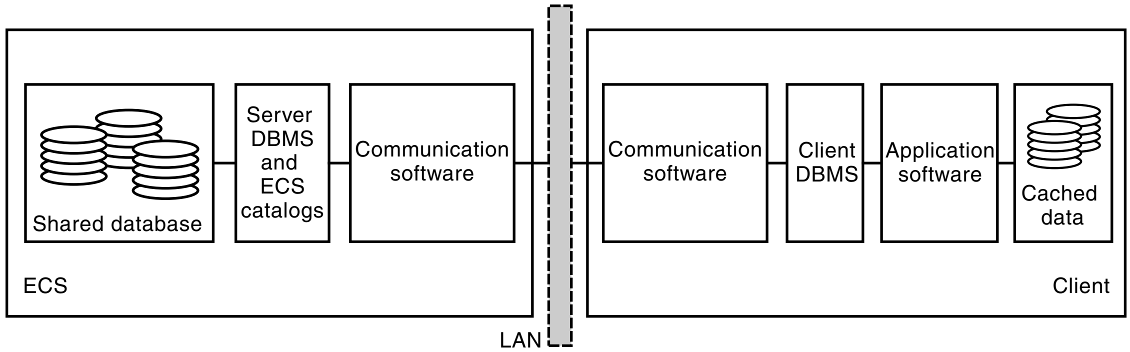

Figure 7. Enhanced client–server architecture.

at the query level to ensure easy constraint checking, but the actual data pages relating to the requested object are shipped back to the server at a low(er) level. By allowing this "dual" interaction, the system offers the consistency maintenance of the query-level interface as well as the performance benefits of low-level transfers. When a modified object is being returned to the server, the data pages are transmitted at page level but the modified access paths and meta-data are submitted at query level. If any consistency constraints are not satisfied by the new data, the injected pages are simply discarded. This avoids the processing of large amounts of data through the higher layers of the database (query processor and complex object manager).

Data Exchange Granularity in CSDs

This section examines CSDs in the light of the interactions of their functional components and the granularity of the data items they exchange. In this regard two broad categories exist, namely query-shipping and data-shipping architectures.

In query-shipping systems, interaction between the client and server takes place as the exchange of queries, submitted in a high-level language such as SQL, and results being returned as matching tuples from a set of data resident on the server. Query-shipping systems are in common use in relational database client–server implementations, particularly those where the level of client interaction is mainly limited to the execution of casual or ad hoc queries. Examples of such systems include "SQL servers," applications that allow PC productivity packages to access enterprise data, and on-line information retrieval systems such as those described by Alonso et al. (31). In Refs. 23 and 32 it was shown that the performance of a properly designed query–server system can be enhanced to the extent where it becomes a viable implementation even for environments that demonstrate high update rates.

Data-shipping systems differ from query-shipping ones in that the unit of data transfer is normally equivalent to the unit of low-level storage. The use of data page transfers allows some of the database functionality to be located at the client site. This allows reduction of the server burden and permits tighter integration between client and server in issues such as concurrency control (27). The scenario used by the enhanced CSDs in Ref. 23 could be viewed as a data-shipping system in which the unit of transfer and client storage is that of data tuple. Such CSDs can therefore be referred to be as tuple-server systems. While the concept of a tuple remains valid in object-oriented databases, their ability to store more complex nested data types and their affiliated methods requires a tighter degree of integration between client and server which can only be offered by low-level data transfers.

Data shipping in client-server architectures has been used for some time in distributed file systems whose principle aims are to increase locality of access and reduce server load. The Andrew File Service (AFS) (33) uses a file–server approach in which files are retrieved from the server when opened, cached at the client while in local use, and finally written back. Caching in AFS is disk-based, which is suitable given that entire files are being transferred at a time and these may exceed the size of primary memory. Sprite (34) and Sun's Network File Server (NFS) use page-shipping approaches to remote file services. Files are opened on the remote server, and pages are fetched as requested by the client. Experiments on the Sprite file system revealed that while client disk caching is definitely beneficial due to the increased locality of access, a large server cache can provide benefits of similar magnitude without the expense of a local disk cache.

The three main data-shipping classes of CS architectures useful for object-oriented databases are the page–server, object–server, and file–server (26). These differ principally in the granularity of data transfer and caching. The file–server and page–server have their origins in distributed file systems. The following subsections examine each of the above classes in some detail.

File–Server CSDs. While this method is not really a major player in the area of database management, it has some interesting properties that allow rapid system development. It is indeed possible to implement a database system on top of a distributed file system, but there are certain inefficiencies involved. These inefficiencies arise due to the mismatch between file systems and databases, and they make this configuration an inefficient solution for CSDs.

The file–server CSD does not use the notion of a file as the unit of transfer. This would be prohibitively inefficient; in fact, it is common for an entire database to be contained in a single operating system file organized into objects (35). The file–server approach often makes use of a remote-open file service such as Sun NFS or Sprite to perform page-level requests for data. Therefore the architecture would simply consist of simplified client systems sharing a database using a remote file service (26). The clients interact with a single-server process that coordinates client I/O requests, concurrency, and the allocation of new pages in the database.

The key benefit of this architecture is that because the network file system software is normally integrated in the kernel of the operating system (at least with Sun NFS), page read operations are quite fast as compared to the performance that would be achieved by using a remote procedure call (26). Caching of data may be performed explicitly by the client application or by the file system's page cache. The former is probably more beneficial, since the buffer replacement used by the file service may be optimized to take into account access patterns that differ from those encountered in databases. Since network file systems have been in use for a long time, they are fairly stable and reliable products.

The use of remote file services has its costs as well. Because the I/O function is separate from the server process, it is often necessary to make separate requests for tasks that are closely related. For example, reading a page from the database requires one call to the server process to get the lock and another to the network file system to retrieve the actual page. NFS, in particular, is also known for the low speed of executing write operations, which can impact transaction throughput adversely.

Page–Server CSDs. The basic page–server architecture is an instantiation of the RAD-UNIFY architecture that uses pages as the main unit of data transfer (26,36). In this case the server is essentially a large buffer pool with buffer management, I/O access, concurrency, and recovery modules. When the server receives a page request, it locks the page in the appropriate mode, retrieves and transmits it to the requesting client. The client database comprises of an object manager, an access method manager, a page buffer, and, of course, a communication module. The client database system acts as an agent for applications running on the same processor, fulfilling their data requirements either using locally cached data or interacting through the server. The client DBMS may cache only pages (page-to-page system) or both pages and objects (page-to-object system). The benefit of an object cache is that space is not wasted storing objects that have not been referenced. Naturally this is dependent on the relative size of the objects and pages.

Caching of objects is not without costs; it requires that objects be copied from the incoming page buffer before they can be referenced. If an object is modified when its corresponding page in the page buffer has already been replaced by a more recent page request, the client will have to retrieve the page from the server again so that the object can be included on it for transmission back to the server. By using a good clustering scheme, it is possible to ensure that most of the objects contained on a page will be related in some fashion (e.g., clustering all components of a complex objects). By such means the number of requests to the server can be reduced, which in turn has implications on the scalability of the system. Additionally, because retrieval operations on the server only involve locating a particular page and transmitting in its entirety, the overhead on the server is reduced to a minimum. Experiments discussed in Ref. 26 show that the page–server architecture, in the form described above, yields performance superior to both file–server and object–server architectures, provided that a good data clustering scheme is in use.

In the RAD-UNIFY CSD there is no interaction between the clients. In Ref. 37 retrieval of information from other clients' caches is presented as a way to "augment" the local cache. By adding client caches, CSDs follow the trend in building global-memory hierarchy systems (38). This makes the volume of data available in memory buffers (other than in the server's cache) larger, further alleviating the performance bottleneck introduced by the server disk's slower access time. When a client application makes a request to the client DBMS, the presence of the relevant data pages in the client cache is checked. A page miss at the client results in the request being forwarded to the server. The server checks if it has the requested page in its memory. If so, the page is sent to the client as normal. If not, before attempting to retrieve the page from its disk, the server checks if any other client has the page cached and is prepared to ship it to the requester. If so, the server puts the two clients in touch with one another and the page is transferred between them. Only when a page is not cached at any client is the server's disk accessed. A number of algorithms have been developed that allow this method to be used to reduce the server load without affecting data consistency in the database, as well as maximizing the amount of data that is available for retrieval from global memory. As Ref. 37 indicates, this configuration is best suitable for environments where there is low to medium data sharing.

The notion of enhanced CSD and the use of client's disks described earlier can be extended in the page–server environment. A proposal along these lines appears in Ref. 29. There are essentially two choices in designing such an architecture: The first would be to have each client act as the host and server for a portion of the database. This approach gives rise to standard distributed database issues such as fragmentation, replication, and data availability problems. The second alternative is to involve the disk that the operating system's virtual memory uses, thus allowing a large in-memory cache to be held. This technique has the disadvantage that the operating system buffer management and replacement policies may not be in agreement with the database access patterns. An additional problem is that the nature of the virtual memory cache is transient, and thus it does not persist across separate executions of the client DBMS software. These problems are overcome in Ref. 29 by modifying the buffer management system of the client DBMS so that it handles disk storage as a direct extension of main memory.

Applied to object–oriented databases, page-server architectures face a few problems. As the unit of transfer and locking is the page, it is difficult to implement object-level locking. This negatively impacts the concurrency of the system. Since object methods can only be executed on the client, operations on collections or parts thereof may require the transfer of the entire collection to the client, which can be expensive in terms of both server load and communication cost.

Object–Server CSDs. As implied by its name, the unit of exchange between client and server in the object–server architecture is the object (26,36). In this architecture almost all database functionality is replicated between client and server. One glaring disadvantage of the page–server approach is that the server has no understanding of the semantics or contents of the object. In cases where objects are small, the page granularity may not be specific enough to minimize network transmissions. Under situations of poor object clustering page–server performance is affected by multiple page requests for each object required by the client. The same problem arises under circumstances where the cache hit rate is low. As a result the object–server is very sensitive to the client cache size (26). By performing requests for data at the object level, a higher level of specificity is achievable, and the clustering problem can be overcome. Conversely, under situations of high clustering, the object–server offers little benefit. It duplicates the effort in clustering data because it determines relationships between objects navigationally (e.g., based on containment and association relationships).

Retaining DBMS functionality at the server has the benefit of allowing the server to perform consistency and constraint checking before performing potentially expensive data transfers. Query predicates and object methods can be evaluated on the server, reducing the size of results to only relevant data. As Ref. 26 shows, the object–server has better performance when the client cache size is small. The use of objects as the unit of transfer and buffering lends itself to high concurrency, and the object–server is best used under situations of high contention. Several techniques have been proposed in order to increase the page–server's concurrency to similar levels (39).

In Ref. 36 some subtle factors that arise in the choice between an object–server and a page–server are suggested. Since the page–server has no knowledge of the object semantics and methods, it is possible to update data in violation of these conditions. As authorization can only be tied to the data transfer granularity, page–servers are unable to permit fine-granularity authorization constraints. Other considerations relating to application development effort, ability to handle dynamic schema changes, programming language support, and the like, are also difficult to address in the page–server environment.

Consistency Maintenance of Networked Data

When volatile memory or disk caching is in use, consistency and control over updates has to be maintained at all times. There are numerous issues that have been studied in this area, and one could broadly classify them into two categories: concurrency control policies and caching algorithms. These two areas are not completely orthogonal, since concurrency control techniques affect the way caching may work. In the following two subsections we examine the questions addressed by research in these two areas.

Concurrency Control Policies. In Ref. 40 an early form of CSD called ObServer, used mostly for the handling of software–engineering artifacts, is presented. The sole purpose of ObServer is to read from and write to disk chunks of memory (software–engineering applications). The server disk unit is organized in segments that store clustered (related) objects. The rationale is that once a segment is retrieved, all associated data items are selected as well. Both segments and objects maintain unique identifiers. Client sites run the ENCORE database which is able to cache objects and rearrange them so that they can best serve the user–applications. Segments represent the unit of transfer from the server to the clients, while modified objects travel in the other direction. It is up to the server to coordinate, through locking, multiple copies of objects and ultimately streamline update operations on segments. The ObServer lock manager can work in two granularities: segments and objects.

The novel point of the locking scheme used here is that clients issue lock requests in the form of triplets: The first element in a triplet is the type of lock required, the second determines the way the lock is to be communicated to other clients that already have a lock on the object in discussion, and the last designates whether the server is to establish a lock. Read and write modes are differentiated as restrictive (*R*) and nonrestrictive (*NR*). *NR-READ* works as a traditional read lock. *R-READ* disallows processes other than the current to read an object. *R-WRITE* provides a user with exclusive access to an object. *NR-WRITE* disallows other processes from obtaining either *R-WRITE* or *R-READ* but allows reading of an object through the *NR-READ* mode.

The locking scheme uses an additional dimension namely that of communication–mode. This locking–mode refers to the communication among clients as the result of an action of another client. More specifically, any changes in the lock status of a server object should be sent to the clients that maintain a lock on the same object. Five communication modes (and their interaction) are proposed:

- *U-Notify*. Notifies lock holders upon object update.
- *R-Notify*. Notifies lock holders if another client requests the object for reading.
- *W-Notify*. Notifies lock holders if another client requests the object for writing.
- *RW-Notify*. Notifies lock holders if another client requests the object for either reading or writing.
- *N-Notify*. Makes no notification at all.

Deadlock detection is performed in the server using a flexible wait-for graph. This hierarchical locking scheme is capable of operating in a more highly concurrent fashion than its strict two-phase counterpart (40).

Wilkinson and Niemat (41) proposed an extension to the two-phase locking protocol for consistency maintenance of workstation cached data. Their protocol introduces cache–locks (CLs). Such locks indicate that clients have successfully obtained server objects. When a client requests a exclusive lock on an item already cached at another client, the CL at that client becomes a pending-update lock (PL). If an update takes place, the PL is converted to an out-of-date lock (OL); otherwise, it is converted back to a CL lock. CL, PL, and OL track the status of objects that are being modified by a client site and at the same time have already been downloaded to others. The introduced concurrency scheme is compared with the protocol that uses notify-locks (40). Simulation results indicate the following:

- Cache–locks always give a better performance than two-phase locking.
- Notify–locks perform better than cache–locks whenever jobs are not CPU bound.
- Notify–locks are sensitive to CPU utilization and multiprogramming level.

Thus, if the processing in the CSD tends to be CPU-bound, cache–locks should be used; otherwise, notify–locks offer better performance.

In a CSD environment, where clients use portions of their main-memory to cache data pages, Carey et al. (27) examine the performance of a number of concurrency control policies. These techniques are used to achieve consistency between server and client-cached data pages. The proposed algorithms are variations of the two-phase locking (two techniques) and optimistic protocols (three techniques).

The basic two-phase locking scheme (B2PL) disallows intertransaction data caching, and pages can be cached as long as a read–lock has been obtained at the server. A client may request an upgrade to a write–lock and receive it provided that there is no conflict at the server. The server is also responsible for monotoring and resolving deadlocks. Caching two-phase locking (C2PL) allows for intertransaction data caching. All items requested for the first time need to be fetched from the server. Clients read valid data as the server exploits reply–messages to piggyback modified pages. To achieve this, the server compares the log sequence numbers (LSN) of its pages with those maintained locally by clients. The server maintains the pertinent LSN numbers of all client-cached pages.

In the optimistic two-phase locking (O2PL) family of protocols, clients update data pages locally. A committing client will have to ultimately "ship" to the server all modified data pages. This is achieved by sending all the dirty pages to the server (in a precommit logical message). The server will then have to coordinate a prepare-phase for the commitment of updates. This phase entails obtaining update-copy locks at the server and on other client–sites that may have cached images of the pages being updated. Update–locks are similar to exclusive locks, but they are used to assist in early deadlock detection as transactions that conflict at commit time indicate a deadlock. Clients that have already acquired update–locks, may have to obtain new copies of the modified server pages. This can be done in a variety of ways: invalidation (leading to the O2PL-I protocol), update propagation (O2PL-P), and finally, by a combination of the two called dynamic algorithm (O2PL-D).

Since B2PL disallows intertransaction data caching, it demonstrates the poorest performance. The performances of the other four protocols present small variations for a small number of clients, and their throughput rates level out for more than 10 clients. The O2PL-I works well in situations where invalidated pages will not be used soon, while O2PL-D performs satisfactorily when the workload is not known a priori. Finally the O2PL-P is good for "feed" (producer/consumer) settings but does not work well when clients have hot–server pages in their cold sets. For workloads with low or no locality, all algorithms perform similarly.

In a parallel study Wang and Rowe (42) examine the performance of five cache-consistency and/or concurrency control algorithms in a CSD configuration, namely two-phase locking, certification, callback locking, no-wait locking, and no-wait with notification. Callback locking is based on the idea that locks are released at the client sites only when the server requires them to do so for update reasons. Once a write occurs, the server requests that all pertinent clients release their locks on a particular object before it proceeds with the processing of the modification. No-wait locking is based on the idea that a client starts working on a transaction based on the cached data and waits for certification by the server at commit time. In this way, both client and server work independently and in a manner that can help increase the system throughput. Notification is added to the no-wait protocol in order to avoid delays in aborting transactions whose cached data have been invalidated by modifications in other sites (server or clients). Simulation experiments indicate that either a two-phase locking or a certification consistency algorithm offer the best performance in almost all cases. This result is based on the assumption that intertransaction caching is in place and is in accordance to what (27) reports. There are two additional results:

- When the network shows no delays and the server is very fast, then no-wait locking with notification or callback locking perform better.
- Callback locking is better when intertransaction locality is high and there are few writes. Otherwise, no-wait locking with notification performs better.

In a later study Carey et al. (27) show how object-level locking can be supported in a page–server object-oriented DBMS. They compare the two basic granularities for data transfer and concurrency control, namely, object level and page level with three hybrid approaches. In the first hybrid approach, locking and callbacks are considered at the object level only. The second hybrid scheme performs locking at the object level but allows page-level callbacks whenever possible, and the third approach uses adaptive locking as well as callbacks. Client–server data transfers are performed at the page level only. Simulation results showed that the third hybrid scheme outperformed all the other approaches for the range of workloads considered. In Ref. 43 an optimistic concurrency control algorithm is proposed that promises better performance than the schemes presented in Ref. 27 in the presence of low to moderate contention. This algorithm has been described in the context of the Thor object-oriented database (44). Transaction processing in Thor is performed at the clients by allowing data–shipping and intertransaction caching. Instead of using callback locks, Adya et al. (43) propose the use of backward validation (45) to preserve database consistency. Once a client transaction reaches the commit stage, it has to be validated with possibly conflicting transactions at other clients. In order to do this, the validation information for the transaction (identity of each object used along with the type of access) is sent to the server. If there is more than one server, this information is sent to one of the servers that owns some of the objects used by that transaction. The server commits the transaction unilaterally if it owns all the objects in question. Otherwise, it coordinates a two-phase protocol with the other servers. Once a read–write transaction commits, the server sends invalidation messages to clients that are caching objects updated by that transaction. These clients purge all invalid objects from their caches and also abort any transactions that may be using these outdated data. The algorithm takes advantage of the presence of closely, but not exactly, synchronized client clocks in order to serialize globally the order of execution of client transactions.

Caching Schemes. So far caching techniques have been used in numerous instances and in diverse settings. More no-table is their applications in the areas of file systems/servers, retrieval systems and CSDs.

We first present a brief introduction to the issue of caching in OSs. Sprite (34) features a mechanism for caching files among a collection of networked workstations. Sprite guarantees a consistent view of the data when these data are available in more than one site and through a negotiation mechanism (between the main and virtual memory components of the client OS) determines the effective physical client memory for file–caching. Sprite permits sequential as well as concurrent write–sharing. Sequential write–sharing occurs when a file is modified by a client, closed, and then open by another client. If the latter client has an older version of the file in its cache (determined by a version number), then it flushes that file from its cache and obtains a fresh version. Since Sprite uses delayed write-backs, the current data for a file may be with the client that last wrote to it. In this case the server notifies the last writer, waits for it to flush its changes to the server, and then allows the requesting client to access the file. Concurrent write–sharing occurs when a file is open at multiple client sites and at least one of them is writing it. In this situation client caching for that file is disabled, and all reads and writes are undertaken by the server. The file in question becomes cacheable again when it has been closed on all clients. Experiments with file operations indicate that under certain conditions, client caches allow diskless Sprite workstations to perform almost as well as clients with disks. In addition client caching reduces server load by 50% and network traffic by 75%.

In Ref. 46 Korner suggested the use of intelligent methods to improve the effectiveness of caching. Caching algorithms using higher-level knowledge can generate expectations of user process behavior to provide hints to the file system. Using Unix-based generalizations of file usage by programs, depending on the filename, extension, and directory of residence, an expert system was used to generate likely access patterns. Three algorithms were examined, namely LRU, optimal, and "intelligent." The data block that the optimal algorithm selects for replacement is that with the next time of reference farthest away from the present time. The intelligent algorithm makes use of three separate performance enhancements:

1. *Intelligent Caching.* Blocks are cached according to anticipated access patterns. Different cache management policies are used based on these anticipated access patterns.
2. *Cache Preloading/Reloading.* Information of general utility to all processes (i.e., *i-node* tables etc.) is determined and preloaded or reloaded during idle server periods.
3. *Intelligent Background Read–Ahead.* Where sequential access was anticipated, the next block of the sequence is passed with each read request to allow discretionary prefetching.

Of the three performance enhancements used in the intelligent algorithm, cache preloading appears to be always useful, and intelligent caching, too, provides performance increases over the LRU strategy. The cost of the extra processing required by the intelligent cache management algorithm is surprisingly small and is readily amortized by the performance gains it provides.

In Ref. 47 an approach to cache management is proposed for distributed systems (databases, file servers, name servers, etc.). Updates at the server are not automatically propagated to the clients that cache affected data. By looking at the cached data as "hints," rather than consistent replicas of the server data, the problems associated with maintaining strict data consistency can be approached differently. The objective is to maintain a minimum level of cache accuracy. By estimating the lifetime of a cached object and its age, the application could determine the degree of accuracy of the object in discussion. Hints that are highly accurate ensure good performance benefits.

In Ref. 48 the issue of write–caching in distributed systems is examined. Write policies used in traditional file system caches use either write-through or periodic write-back which may result in little benefit in general distributed settings. Here systems with client and server nonvolatile caches are considered. Both a single-level caching system (using the server's memory) and a two-level caching (using client caches as well) settings were examined. The replacement policies used were LRU, WBT (write-back with thresholds which is purging-based) and LRUPT (LRU purge with thresholds). In WBT, a block purge is scheduled whenever the cache occupancy exceeds a given high-limit threshold. LRUPT combines LRU and WBT; cached blocks are maintained in LRU order and purged according to this order. Experimental results suggest that LRU as well as LRUPT perform well in a single-level write–caching environment. In a two-level caching environment, the combination of LRU at the client and WBT at the server results in better performance.

In Ref. 31 Alonso et al. proposed the utilization of individual user's local storage capacity to cache data locally in an information retrieval system. This significantly improves the response time of user queries that can be satisfied by the cached data. The overhead incurred by the system is in maintaining valid copies of the cached data at multiple user sites. In order to reduce this overhead, they introduce the notion of *quasi-copies.* The idea is to allow the copies of the data to diverge from each other in a controlled fashion. Propagation of updates to the users' computers is scheduled at more convenient times, for example, when the system is lightly loaded. The paper discusses several ways in which the decision to add or drop data from the users' cache can be specified by the user. Coherency conditions specify the allowable deviations of the cached image from the data at the server. Several types of coherency conditions are discussed, and analysis shows that quasi-caching can potentially improve performance and availability in most circumstances. Response time problems can arise in systems where a very large fraction of the updates received at the server have to be propagated to the users' computers. Similarly problems arise if the selection and coherency conditions are very complex. In this case the overhead of the bookkeeping may outweigh the savings. The ideas discussed in this paper were further extended and analyzed in Ref. 49.

In Ref. 37 a framework that allows client page requests to be serviced by other clients is proposed. This paper treats the memory available to all the clients as another level in the global memory hierarchy. This available memory is classified into four levels based on the speed of access: The local client–

memory (because it is the fastest to access), server–memory, remote client–memory, and the server–disk (it is the slowest to access). To optimize the page accesses in this context, a number of page replacement techniques have been suggested. In the Forwarding algorithm, a page request can be fulfilled not by the server but by another client that happens to have a copy of the requested page in its own cache. In Forwarding with Hate–Hints, a server page dispatched to a client is marked as its "hated" one. Even if the server page is subsequently removed in the server's buffer, it can be still retrieved from the client that has cached it. In this manner a server disk–access is avoided. If there is only one copy of a page available in the global memory in a nonserver location and the holding client wants to drop the page in question, the server undertakes the task to be its "next" host. This technique is termed Forwarding–Sending–Dropped-Pages. The two last schemes can be combined in a more effective technique called Forwarding–Hate–Hints and Sending–Dropped-Pages. Since the introduced techniques strive to keep pages available in the main-memory areas, they display throughput gains if compared with the conventional callback locking policy.

The idea of distributed-caching as described in Ref. 50 is to off-load data access requests from overburdened data servers to idle nodes. These nodes are called mutual-servers, and they answer query with the help of their own data. This study focuses on the following caching policies: passive sender/passive receiver (PS/PR), active sender/active receiver (AS/AR), and similarly AS/PR and PS/AR:

1. *PS / PR.* The sender does not actively hand over any object. When it needs to throw something away, it simply broadcasts it to the network. If some mutual-server is listening, the object might be picked up if it seems valuable; otherwise, it is dropped. The mutual-servers do not make any active efforts to fill up their buffers either.
2. *AS / PR.* A data server or mutual-server trying to get rid of an object takes the initiative to hand it over to another mutual-server. When an active-sender node perceives itself to be a bottleneck, it broadcasts a message to the network seeking hosts for its most globally valuable objects. From those mutual-servers that respond, the server selects one and hands over the object.
3. *PS / AR.* Idle mutual-servers take the initiative to obtain globally valuable data from data servers and overflowing mutual-servers. As busy servers discover the existence of willing receivers, they hand over their most valuable objects to them.
4. *AS / AR.* In this scenario all nodes are active senders or receivers. When a data server or mutual-server is idle, it volunteers to store other nodes' most valuable objects, and when it becomes a bottleneck, it looks for other nodes to which to off-load its most valuable objects.

In most simulation settings distributed caching policies show superior performance to the pure client–server system. Active-sender policies perform the best under skewed loads.

In Ref. 32 the problem of managing server imposed updates that affect client cached data is examined in the context of the enhanced CSD architecture. Five update propagation techniques are introduced and their behavior is examined through experimentation. The strategies differ mainly in their approaches to server complexity and network bandwidth utilization. The simplest update propagation strategy is the on-demand strategy (ODM) where updates are sent to clients only on demand. The next two strategies are built around the idea of broadcasting server data modifications to clients as soon as they commit. In the first one, updates are sent to all clients indiscriminately as soon as a write operation commits. This strategy requires no extra functional server overhead, and is called broadcasting with no-catalog (BNC) bindings. In the other strategy, the server maintains a catalog of binding information that designates the specific areas of the database that each client has cached. Every time an update job commits, the server sends the updated data only to those clients that require it. This strategy tries to limit the amount of broadcasted data and requires additional server functionality. It is called broadcasting with catalog (BWC) bindings. The two final strategies combine the previous strategies with the idea of periodic update broadcasts. Here client-originated requests are handled in a manner similar to ODM but at regular intervals the server dispatches the updates that have not been seen by clients yet. This can be done in two different ways, indiscriminately [periodic broadcasting with no-catalog bindings (PNC)] or by using a discriminatory strategy based on catalog bindings [periodic broadcasting with catalog bindings (PWC)]. Simulations indicate that the performance of these update propagation techniques depends greatly on the operating conditions of the ECS. For example, the ODM strategy offers the best performance if none of the server resources reaches full utilization, while BNC offers the best performance under high utilization of server resources when the updates have small page selectivities, the number of clients is large, and the number of updates increases linearly with the number of clients in the system.

In Ref. 51 O'Toole and Shrira present a scheme that allows clients to cache objects and pages. Previous studies have shown that when hot data are densely packed on pages, page-based caching performs well, and when hot data are sparsely packed, object-based caching performs better (27). By proposing a hybrid caching scheme, this work tries to reduce the number of I/Os when the server writes client-committed updates into the master database. Such update operations are termed *installation reads.* The server receives commit requests from the clients for whole pages or individual objects. When the commit request provides a page, the server validates the transaction according to the individual object that was modified and then uses the containing page to avoid the read phase of an installation. Commit requests that provide individual objects require the server to perform installation reads. By using an opportunistic log (52), installation–reads are deferred and scheduled along with other object updates on the same pages if possible. Simulation results show that when disk I/O is the system performance bottleneck, the hybrid system can outperform both pure object caching and pure page caching.

Predicate indexing (53) and predicate merging techniques are used to efficiently support examination of cached query results. When a new query partially intersects cached predicates, this query's predicate can be trimmed before submission to the server. This can reduce the time required to materialize a query result at the client. Queries are also augmented at times to make the query result more suitable for caching. Query augmentation can result in simpler cache descriptions, thus in more efficient determination of cache completeness and currency with the potential disadvantages of increasing query results and response times, and wastage of server and client resources in maintaining information that may never be referenced again. By exploiting the above ideas, Keller and Basu (54) introduced predicate-based client-side caching in CSDs. It is assumed that the database is relational and stored entirely at a central server. The key idea is the reuse of locally cached data for associative query execution at the clients. Client queries are executed at the server, and the results are stored in the client cache. The contents of client caches are described by means of predicates. If a client determines from its local cache description that a new query is not computable locally then the query (or a part of it) is sent to the server for execution. Otherwise, the query is executed on the cached local data. Transactions executing at the clients assume that all cached data are current. Predicate descriptions of client caches are also stored by the server. This allows the server to notify clients when their cached data are updated at the server. There are several methods for maintaining the currency of the data cached at a client: automatic refresh by the server, invalidation of cached data and predicates, or refresh upon demand.

Recovery

Since CSDs often stage data in nodes other than the database server(s), the issue of recovery after a failure is of vital importance. Recovery in CSDs has been addressed by introducing variants of the basic ARIES database recovery protocol.

Recovery in the Client–Server EXODUS Storage Manager (ESM-CS) (55) involves two main components. The logging subsystem maintains an append-only log on stable storage, and the recovery subsystem uses the log to provide transaction rollback and crash recovery. Crash recovery is performed by the server in communication with the clients using a modification of the ARIES algorithm (56). ESM-CS uses strict two-phase locking for data pages and non-two-phase locking for index pages. Before each client transaction commits, all the pages modified by it are sent to the server (no intertransaction caching). Before the pages are sent, however, the log records for the transaction are sent to the server and written to stable storage (write-ahead logging). Checkpoints are taken at the server regularly. Each page has a log record counter (pageLRC) that is stored with the page itself. When a page is modified, the pageLRC is updated and copied into the corresponding log record. During crash recovery, the pages that could have possibly been dirty at the time of the crash are identified. This is not as simple as in ARIES, since there may be pages that are dirty at a client but not at the server. The pageLRC is compared with the LRC of the log record to determine whether a particular update has been reflected in the page. Care has to be taken to ensure that the combination of page ID and pageLRC refers to a unique log record.

ARIES/CSA (57) is another modification of the ARIES redo-undo algorithm (56). Adapting ARIES to a CSD environment requires that the log sequence numbers generated throughout the system be unique and monotonically increasing. The log records produced at a client for local updates are sent to the server when dirty pages are sent back or when a transaction commits, whichever happens earlier. Write-ahead logging is used to ensure that log records are sent to the server and written to stable storage before any pages are sent back. The Commit_LSN (58) technique is used to determine whether all the updates on a page were committed. This method uses the LSN of the first log record of the oldest update transaction still executing to infer that all the updates in pages with page_LSN less than Commit_LSN have been committed. Clients as well as the server take checkpoints at regular intervals. This allows for intertransaction caching of data at the clients.

In Ref. 59 Panagos et al. propose the use of local disks for logging and recovery in data–shipping CSD architectures. All updates on cached data items, performed at clients, are logged locally. Concurrency control is based on strict, global two-phase locking. The local logs of the clients need never be merged, and local transaction rollback and crash recovery are handled exclusively by each client. Recovery is based on the write-ahead log protocol and the ARIES redo-undo algorithm (56) is used. The steps taken in the proposed recovery algorithm for recovery from a single node crash are (1) determining the pages that may need recovery, (2) identifying the nodes involved in the recovery, (3) reconstructing lock information, and (4) coordinating the recovery among the involved nodes.

PARALLEL DATABASE SYSTEMS

High-performance computing systems are available today in many flavors and configurations. Such parallel systems already play a vital role in the service sector and are expected to be in the forefront of scientific and engineering computing in the future (60). Parallel database systems (PDSs) offer high performance and high availability by using tightly or loosely connected multiprocessor systems for managing ever-increasing volumes of corporate data. New and data-intensive application areas call for the further development and refinement of PDSs featuring ultra-high CPU processing capacity and aggregate I/O bandwidth.

In today's business world, novel application areas that enjoy tremendous growth include data warehousing, decision support systems (DSS), and data mining. The main characteristics of these applications are the huge volumes of data that they need to handle and the high complexity of the queries involved. Queries in data warehouses and DSSs make heavy use of aggregations, and they are certainly much more complex than their OLTP counterparts (61). In data mining, useful association patterns need to be discovered by scanning large volumes of mostly historical and temporal data (62,63). With the introduction of multimedia and digital libraries, diverse data types have been introduced (i.e., images, video clips, and sounds) that require an order of magnitude higher disk capacity and more complex query processing. Uniprocessor database systems simply cannot handle the capacity or provide the efficiency required by such applications. The goal of a PDS is to provide high performance and availability at a much lower price than an equivalent aggregate of uniprocessor systems (64).

It has been successfully argued and shown that the relational model and its accompanying operators are amenable to parallelization. Hence the relational model has become the natural choice for deployment in PDSs. The power of the model lies in its simplicity and uniformity. Relations consist of sets of tuples, and operators applied on relations produce new relations. In this regard relational queries can be decomposed into distinct and possibly independent relational operators.

A PDS can achieve high performance through parallel implementation of operations such as loading data, building indexes, optimizing and processing of queries, and load balancing (65). A PDS can exploit parallelism by using one of the following approaches:

1. *Pipelined Parallelism.* The PDS can execute a relational query in parallel by streaming the output of one operator into the input of another operator.
2. *Partitioned Parallelism.* The PDS partitions the input data and each processor is assigned to one of these data sets. All processors apply the same operator simultaneously.
3. *Independent Parallelism.* Distinct PDS processors execute different operators on possibly disjoint data sets at the same time. In this type of parallelism, the key assumption is that the input and the output of the parallel operations are not related.

Throughput and average transaction response time are the two performance indicators mostly used in the evaluation of PDSs. A PDS that processes a large number of small transactions can improve throughput by executing as many transactions in parallel as possible. On the other hand, a system that processes large transactions can reduce the response time by performing many different tasks of each transaction in parallel.

There are two possible ways to parallelize a query evaluation process (66): interquery and intraquery parallelism. In interquery parallelism, several different queries are executed simultaneously. The goal of this form of parallelism is to increase transaction throughput by utilizing as many processors as possible at any time. Intraquery parallelism refers to the execution of a single query in parallel on multiple processors and disks. Hence the response time of individual queries is reduced. Interquery parallelism cannot achieve significant response time reduction, since individual tasks assigned per processor are scheduled according to a strict sequential discipline.

Intraquery parallelism can be manifested in two forms: intra- and interoperation parallelism. Intra-operation parallelism executes the same operator on a number of processors with each processor working on a different data set. Interoperation allows the assignment of processors on the various nodes of the query tree on demand. The two types of intraquery parallelism are complementary and can be used simultaneously on a query. Large-scale parallelism of a complex query may introduce significant communication costs. Therefore the PDS must not only consider conventional query optimization and load balancing issues but also take into account the communication overhead involved.

Since critical applications are run on PDSs, high availability is a much desired property for the system in the presence of a failure. The probability of a single processor or disk device failure in a PDS consisting of a large number of processors and disks in significantly higher than in a uniprocessor system. A PDS designed without taking this fact into account will demonstrate very frequent breakdowns of service. For instance, if a component (either CPU or disk unit) has a failure rate of once every five years, then in an aggregate architecture with 100 such components and assuming statistical independence, the mean failure rate is once every 18 days.

For database applications the availability of disk–resident data objects is perhaps the most critical concern (67). One approach to obtain higher availability is to simply replicate data items on separate disks. Thus in the event of a disk failure, the copy of the data may still be available on the backup disk. Unless both disks (the original disk and the backup disk) fail at the same time, the failure of a single disk will be transparent to the users and the PDS will continue to operate properly. However, replication can potentially lead to data inconsistency if a data item gets modified but its copy remains unchanged. To avoid this undesirable effect, a protocol that avoids inconsistencies has to be enforced at all times. A popular such protocol is ROWA (read one, write all) where a logical read operation is converted to a physical read operation of any one of the copies, but a logical write operation is translated into physical writes to all copies.

If disk *A*—which has a (partial) replica of its data on disk *B*—fails, then disk *B* will have to carry not only its own requests but the queries received by the failed disk as well. This "double" work that disk *B* has to accommodate may result in poor response time which could become twice as long. In addition the throughput of the overall system will be effectively reduced. In order to avoid the above phenomena, a scheme that replicates the data on the disks, in a manner more resilient to disk failures, is required. Chained declustering is a technique that allocates data throughout the available disk devices and provides acceptable performance rates in the case of a failure (67). We briefly describe chained declustering in a subsequent section.

Metrics and Design Objectives

The two most important metrics in studying parallelism are speedup and scaleup (65). Speedup indicates how much faster a task can be run by increasing the degree of parallelism. Scaleup refers to the handling a larger task by increasing the degree of parallelism proportional to the size of the task. More specifically, consider a PDS running a database application, and suppose that we enhance the system adding new processors and disks. Let the execution time of the application in the initial system be T_S and that in the enhanced configuration be T_L. Then the speedup given by the larger system is

$$\text{Speedup} = \frac{T_S}{T_L}$$

The speedup is linear if an N-times larger or more expensive system yields a speedup of N. If the speedup is less than N, the PDS demonstrates sublinear speedup. The notion of speedup holds the problem size constant while the PDS grows in terms of available computing resources. However, it is very often the case that we need to increase the "capacity" of the PDS so that it can handle a larger database (problem domain). In this case the effectiveness of the new system is expressed by using the notion of scaleup.

Let us assume that a database task A runs on a parallel database system M and with execution time T_A. Now suppose that we enhance the old system and build a new system L that is N times larger or more expensive than M. In L we run a new database task B that is N times larger than A and the execution time is T_B. Then the scaleup is defined as the ratio

$$\text{Scaleup} = \frac{T_A}{T_B}$$

The PDS demonstrates linear scaleup on task B if the above fraction is equal to one. If $T_B > T_A$ (i.e., scaleup < 1), then the PDS is said to demonstrate sublinear scaleup behavior.

There are two distinct types of scaleup relevant to database systems, depending on the composition of the workload: transactional and batch scaleup. In transactional systems a database task consists of many small independent requests (containing updates as well). For instance, consider an OLTP system that manages deposits, withdrawals, and queries on account balance. In such systems we would like to ideally obtain the same response time despite the increase in the number of user requests and the size of the database. Therefore transactional-scaleup designates not only N-times many requests but also demands that these requests be executed on a shared database that is N-times larger than the original one. Transactional-scaleup is a well-suited indicator for the assessment of a PDS because transactions run concurrently and independently on separate processors, and their execution time is independent of the database size. In batch–scaleup the size of the database increases along with the size (or range) of the submitted query. If a N-times larger (and possibly more complex) transaction runs on a N-times larger database (using a N-times larger PDS) and we still maintain the same levels of response times, then we can say that the PDS presents linear batch–scaleup.

In optimal settings PDSs should demonstrate both linear speedup and scaleup (65). However, a number of restraining factors prevent such systems from achieving this. They are as follows:

1. *Startup Costs.* There exist costs every time a process is initiated in a parallel configuration. If tens or even hundreds of processes must be started, then the startup time can easily dominate the actual computation time, resulting in execution time degradation.
2. *Interference.* A task executed in a PDS may consist of a number of processes executing concurrently that may access shared resources. Whenever there is contention for a shared resource (communication media/buses, disks, locks, etc.) by two or more parallel transactions, a slowdown will inevitably take place. Both speedup and scaleup can be affected by such contention.
3. *Service Time Skew.* A well-designed PDS attempts to break down a single task into a number of equal-sized parallel subtasks. The higher number of subtasks we create, the less the average size of each subtask will be. It is worthwhile to note that the service time of the overall task is the service time of the slowest subtask. When the variance in the service times of the subtasks exceed the average service time, then the partitioning of the task is skewed. In the presence of a skewed partitioning, increasing parallelism improves the execution time only slightly, since there is a subtask with very long service requirements.

Parallel Database Architectures

In Refs. 64 and 68 a taxonomy for such parallel systems and frameworks for their implementation were presented. Depending on the employed hardware configurations and the used software paradigms, various parallel database architectures are feasible. In the following subsections, we discuss four such architectures:

- *Shared-Memory.* All processors share direct access to a common global memory and to all disks.
- *Shared-Disk.* Each processor has a private memory and direct access to all disks through an interconnection network.
- *Shared-Nothing.* Each processor has local main memory and disk space; in addition each site acts as a server for the data resident on the disk or disks in it.
- *Hierarchical or Hybrid.* This model is organized around an interconnection network that allows interoperation of functionally independent sites. Each site is, in its own right, organized according to one of the preceding three models.

Shared-Memory Architecture. In a shared-memory system any processor and disk has direct access to a common global memory. Figure 8 depicts the salient characteristics of this architecture. The advantages of a shared-memory architecture are simplicity in developing database software, efficient communication among processors, and possibility for effective load balancing.

Since every processor shares the database's meta-data and catalog information, migration of a database from a multitasking uniprocessor system to a shared-memory environment is a relatively straightforward task. Simply every process (transaction) that used to run concurrently can be now executed on an individual processor, in parallel with other processes. This represents interquery parallelism which may result in a higher throughput for the overall system. Thus database applications designed for uniprocessors can be run in a shared-memory system with few or no changes. Intraquery parallelism for shared-memory architectures requires more effort to be implemented but remains simple. Un-

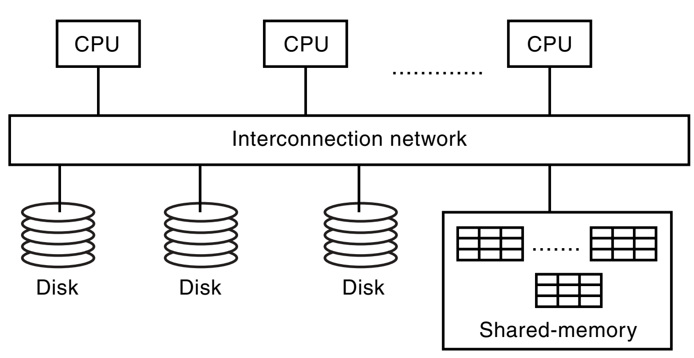

Figure 8. Shared-memory architecture.

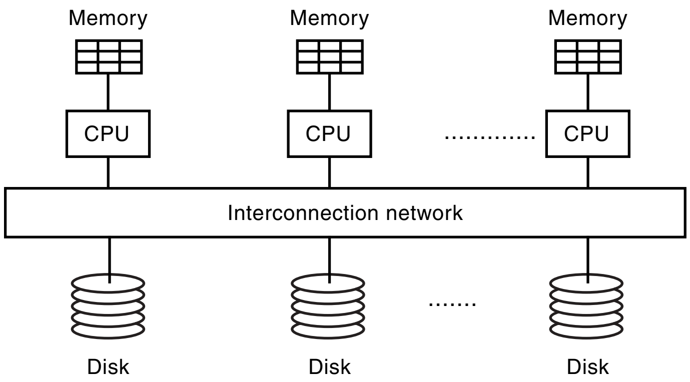

Figure 9. Shared-disk architecture.

fortunately, intraquery parallelism may impose high interference, hurting the response time and the throughput. Most of the contemporary shared-memory commercial PDSs exploit only interquery parallelism.

The communication between processors can be implemented with shared memory segments using only *read* and *write* system calls, which are much faster than message sends and receives. The load balancing is excellent because, every time a processor finishes a task, it can be assigned a new one resulting in an almost perfectly balanced system. On the other hand, shared-memory architectures suffer in cost, scalability, and availability. The interconnection network must be extremely complex to accommodate access of each processor and disk to every memory module. This increases the cost of shared-memory systems when large numbers of participating resources are involved. The interconnection network needs to have a bandwidth equal to the sum of the transfer bandwidths of all the processor and disk components. This makes it impossible to scale such systems beyond some tens of components as the network becomes a bottleneck. Therefore the scalability of a shared-memory system is rather low. Also a memory fault may affect most of the processors when the faulted module is a shared memory space, so reducing the data availability.

Examples of shared-memory PDSs are the XPRS system (69), DBS3 (70), Volcano (71), and Sybase ASE 11.5. In summary, the shared-memory architecture is a satisfactory solution when the PDS maintains coarse granularity parallelism.

Shared-Disk Architecture. In a shared-disk architecture each processor has a private memory and can access all the available disks directly via an interconnection network. Each processor can access database pages on the shared disks and copy them into its own memory space. Subsequently the processor in discussion can work on the data independently, without interfering with anyone else. Thus the memory bus is no longer a bottleneck. To avoid conflicting operations on the same data, the system should incorporate a protocol similar to cache-coherence protocols of the shared-memory systems. Figure 9 depicts this architectural framework.

If the interconnection network can successfully scale up to hundreds of processors and disks, then the shared-disk architecture is ideal for mostly-read databases and for applications that do not create resource contention. The cost of the interconnection network is significantly less than that in the shared-memory model, and the quality of the load balancing can be equally good.

An additional advantage of the shared-disk over the shared-memory organization is that it can provide a higher degree of availability. In case of a processor failure, the other processors can take over its tasks. The disk subsystem can also provide better availability by using a RAID architecture (72). Migrating a system from a uniprocessor system to a shared-disk multiprocessor is straightforward, since the data resident on the disk units need not be reorganized. The shared-disk configuration is capable of exploiting interquery parallelism.

On the other hand, the main drawback of the shared-disk architecture remains its scalability, especially in cases of database applications requiring concurrent read and write operations on shared data. When the database application makes a large number of disk accesses, the interconnection to the disks becomes a bottleneck. Interference among processors is also possible, and control messages among processors due to coherency protocols may further worsen matters.

Shared-Nothing Architecture. In a shared-nothing (SN) system architecture each node of the PDS is a full-fledged computing system consisting of a processor, main-memory buffers, and one or more disks. The sites communicate with each other through a high-speed interconnection network. Such a system can be a parallel multicomputer system or even a number of workstations attached to a high-speed local area network (termed Network of Workstations or NOW). Figure 10 depicts the architecture in question.

The major benefit of a shared-nothing system is its scalability. A shared-nothing architecture can easily scale up to thousands of sites that do not interfere with one another. The interference is reduced by minimizing resource sharing and carefully partitioning data on multiple nodes. It has been shown that shared-nothing architectures can achieve near-linear speedups as well as good scaleups on complex relational queries and on-line transaction processing workloads (28).

As one can easily observe, the previous architectures (Figs. 8 and 9) tend to move large amounts of data through the interconnection network. The shared-nothing, on the other hand, if designed properly, can minimize such data movement. Essentially it can move only requests and answers providing a sound foundation for achieving high scalability. Another advantage of the shared-nothing architecture is that it can make use of commodity computing systems. At the same time, the need for a very expensive interconnection network

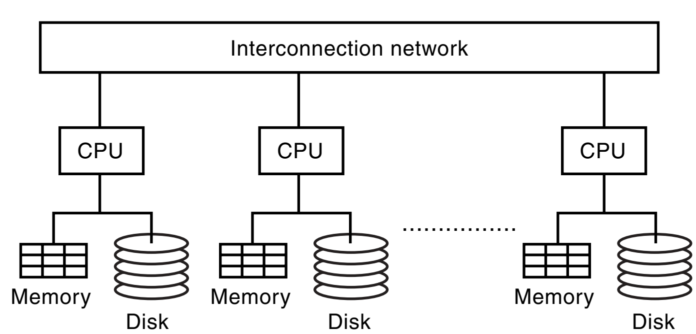

Figure 10. Shared-nothing architecture.

can be avoided. Today's high-performance processors, large memory modules, sizable disk devices, and fast LANs are available at very low costs. Thus the shared-nothing framework can be realized by utilizing "off-the-shelf" components, reducing the cost of the overall architecture tremendously.

The availability of such systems can be increased by replicating data on multiple nodes. Finally, since disk references are serviced by local disks at each node, without going through the network, the I/O bandwidth is high. Under pure-query settings, this I/O bandwidth is equal to the sum of the disk bandwidths of all the nodes involved.

The main drawbacks of the shared-nothing systems lie with the high complexity in the system software layer and the load balancing used. Shared-nothing PDSs require complex software components to efficiently partition the data across nodes and sophisticated query optimizers to avoid sending large volumes of data through the network. Load balancing depends on the effectiveness of the adopted database partitioning schemes and often calls for repartitioning of the data so that query execution is evenly distributed among system nodes. Finally the addition of new nodes will very likely require reorganization of the data to rebalance the load of the system.

The shared-nothing architecture has been adopted by many commercial database systems such as Tandem, Teradata (one of the earliest and most successful commercial database machine), Informix XPS, and BD2 Parallel Edition (73) as well by numerous research prototypes including Gamma (74) and Bubba (75).

Hierarchical-Hybrid Architecture. The hierarchical or hybrid architecture represents a combination of the shared-memory, shared-disk, and shared-nothing architectures (64). The main vehicle of this architecture is an interconnection network that aggregates nodes. These nodes can be organized using the shared-memory model where a few processors are present. This is shown in Fig. 11. Alternatively, every node can be configured as a shared-disk architecture. In this case every processing element could be further organized using the shared-memory model. Thus one may achieve three levels of hierarchy with each one representing a different architecture. Hua et al. (76) proposed a hybrid system where clusters of shared-memory systems are interconnected to form a shared-nothing system.

The case for a hybrid system termed "shared-something" is discussed in Ref. 64. This is a compromise between the shared-memory and shared-disk architectures as CPUs in a shared-disk model work off a global memory space. It is expected that such hybrid architectures will combine the advantages of the previous three models and compensate for their disadvantages (76). Thus hybrid architectures provide high scalability as the outer level employs a shared-nothing design and, at the same time, furnish good load-balancing features by using shared-memory configurations in each node.

Many contemporary commercial PDSs have converged toward some variant of the hierarchical-hybrid model. NCR/Teradata's new version of database machine as well as Tandem's ServerNet-based systems are samples of the hierarchical architecture.

Data Placement

Data placement is one of the most critical issues in PDSs. In the context of the shared-nothing (SN) architecture, it has been studied extensively, and a number of placement algorithms have been proposed. In such systems the effectiveness of the load balancing is largely dependent on proper data placement. In SN architectures data placement determines not only the data distribution but also the distribution of operators that access the data. Thus, if data are not carefully assigned to the nodes, the load might be distributed nonuniformly leading to the creation of bottlenecks. The I/O parallelism in a PDS can be fully exploited only if the data are placed on multiple disks. Thus the data should be horizontally partitioned or "declustered." It has been shown that declustering is useful for shared memory configurations as well, since memory conflicts can be reduced (70).

In data placement there are three major factors to be determined: the degree of declustering, the selection of particular nodes (disks) on which the partitioned data will be stored, and the mapping of data tuples to system nodes (partitioning method). The degree of declustering is the number of nodes (disks) on which a relation is distributed, and its choice is a very important decision as far as the data placement algorithms are concerned. It should be chosen so that the benefit

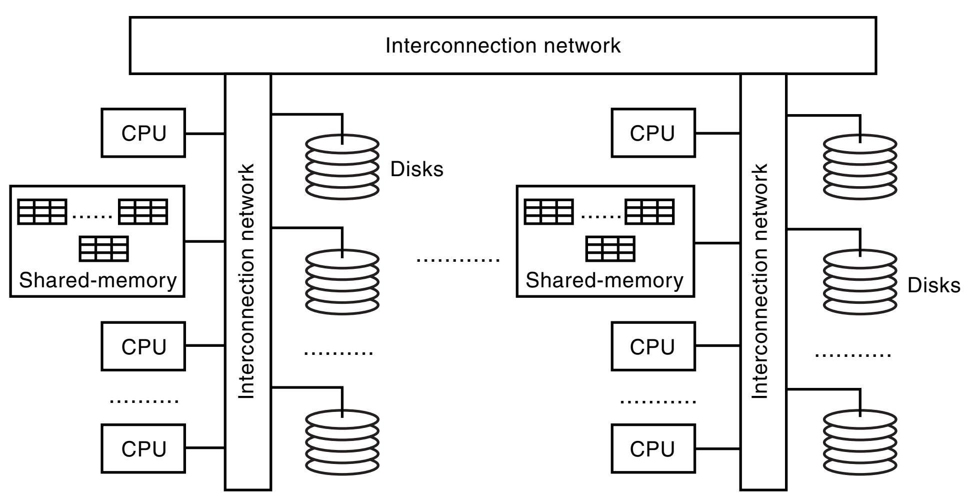

Figure 11. Hierarchical architecture.

of parallelism is higher than the cost of the overheads incurred. A higher degree of declustering indicates higher parallelism for the relational operators. The factors that affect the degree of declustering chosen are startup and termination costs of the operators, communication costs, and data skew. In Ref. 77 an experimental methodology that computes the degree of declustering is discussed. This degree selection is based on the maximization of the system throughput achieved by the PDS. Simulation experiments indicate that for the system parameters used, full declustering is not the best option possible.

As soon as the degree of declustering has been determined, partitioning techniques are used to place tuples into nodes (disks). Some commonly used methods are as follow:

1. *Round-Robin (RR).* The relation is declustered in a round-robin fashion. Thus, if the degree of declustering is M, the ith tuple is placed on the i mod Mth node (disk). The main advantage of this method is its excellent load balancing, since every node (disk) has approximately the same number of tuples. RR is ideal for queries that scan entire relations. On the other hand, all M nodes (disks) must be used for point and range queries, even if the result resides on only one node (disk).
2. *Hash Partitioning (HP).* Here the relation is declustered using a hashing function with range 0 to $(M - 1)$. This function takes as input the partitioning attribute of a tuple and returns the number of the node (disk) where this tuple is to be placed. If the hash function is chosen carefully, and the data are not skewed on the partitioning attribute, the data are declustered almost uniformly. Subsequently queries that scan the entire relation are very efficient, since it takes approximately $1/M$ of the time required to scan the relation on a single-disk system. Point queries on the partitioning attribute are executed very efficiently, since the hash function can directly identify the node (disk) that may contain the target tuples. Range queries have to be materialized by scanning all M nodes.
3. *Range Partitioning (RP).* This method requires from the user to specify a range of attribute values for each node (disk). Such a declustering is described by a "range vector" which consists of the partitioning attribute and the various adopted ranges. The database catalog maintains such range vectors. RP is obviously well suited for point and range queries. As compared to HP, a point query may display some overhead because the range vector has to be looked up before the query is directed to the appropriate node (disk). For range queries, requests are directed only to specific nodes that may have the answer. Depending on the selectivity of the range query, RP can produce the results in either short or long turnaround times. If the selectivity is large, RP will furnish unsatisfactory query turnaround times. In this case the HP or RR are preferable.
4. *Hybrid-Range Partitioning (HPR) (78).* This technique attempts to combine the sequential paradigm of the RP and the load balancing of RR partitioning. To achieve this, the HPR uses the characteristics of the submitted queries. In particular, HPR takes as input the average query CPU execution time, the average query I/O time needed, the average communication time, and the additional costs to initiate and terminate the execution of the query. Then it computes the optimal number of processors (ONP) required to minimize the average response time. Assuming that the average result size of a query is N_{result} tuples, then the fraction of N_{result}/ONP is computed. This fraction represents the maximum number of tuples to be returned by a single node in the case of a range query. This set of tuples is termed "fragment" (78).

   Subsequently the relation is sorted on the partitioning attribute and is chopped into sequential fragments of size N_{result}/ONP. Finally these fragments are distributed among the PDS nodes (disks) through a round-robin technique. The assignment of fragments to nodes is kept in a range table.

Disk	0	1	2	3	4
Primary copy	F0	F1	F2	F3	F4
Backup copy	f4	f0	f1	f2	f3

Figure 12. Disk layout for chained declustering.

In Ref. 79 a simulation study of data placement algorithms is presented for a shared-nothing architecture. Due to the high processing power of contemporary processors and high bandwidth of modern interconnection networks, full declustering is shown to be a viable method for data placement. Full declustering provides the highest degree of parallelism and avoids the penalties of computing either the degree of declustering or the placement of data partitions on the available disks.

Another critical issue in data placement algorithms is the availability of data in the presence of failures. Chained declustering is a technique that redistributes the load in the event of a failure (67). In this technique, system nodes are divided into disjoint groups called clusters. The tuples of a relation are horizontally declustered among the disks of one cluster. Two copies of each relation are maintained, the primary and the backup copy. The tuples in the primary copy are declustered using a partitioning method (from those mentioned earlier) and the ith primary copy partition (Fi) is stored on the i mod Cth disk in the cluster, where C is the cluster size. The backup copy consists of the same partitions as the primary copy and the ith backup partition (fi) is stored on the $(i + 1)$ mod Cth disk. The term chained declustering indicates the fact that any two adjacent disks are "linked" together like a chain. An example with $C = 5$ is shown in Fig. 12.

During normal operation, read operations are directed to primary copies and write operations to both primary and backup copies (i.e., ROWA protocol). If a single-disk failure occurs, chained declustering tries to uniformly distribute the load among the remaining nodes. In this case all primary and backup partitions on the working disks are used. The increase of the load on each disk is $1/(C - 1)$, assuming that the load was distributed uniformly to all disk before the failure occurred. For example, if the disk number 2 fails, the backup copy that resides on disk 3 must be used instead. Now, disk 3 redirects the 3/4th of its own requests to disk 4. Disk 4 will

Disk	0	1	2	3	4
Primary copy	F0	F1	X	F3	F4
Backup copy	f4	f0	X	f2	f3
Load	3/4 FO + 2/4 f4	F1 + 1/4 f0		1/4 F3 + f2	2/4 F4 + 3/4 f3

Figure 13. Disk failure handing in chained declustering.

use the backup partition number 3 (*f3*) to accommodate these requests. In the same manner, disk 4 will send the 2/4th of its own requests to disk 0 and so on (Fig. 13). This *dynamic rebalancing* has, as a direct result, an increase in load of all still functioning disks by 1/4th. The reassignment of the active partitions does not require disk I/O nor data movements across disks. It can be implemented by only changing some bounds in main-memory managed control tables.

Parallel Query Optimization

A vital component for the success of a PDS is the parallel query optimizer (PQO). Given a SQL statement, the objective of the PQO is to identify a parallel query materialization plan that gives the minimum execution time. Since one of the objectives of PDSs is to diminish the query response times in decision–support and warehousing applications, the role of PQO is of paramount importance to the success of such systems (80).

Techniques employed by conventional query optimizers are not adequate for PDSs. More specifically, in the case of multiway joins, a conventional query optimizer considers plans only for the left-linear join tree. In doing so, the optimizer limits the search space and exploits possible auxiliary access structures on the joining operands. This strategy works reasonably well for uniprocessor systems (81). However, the introduction of parallelism in PDS makes the number of possible join trees very high. This means that optimal and even near-optimal solutions may not be included in the search space when it is restricted to linear join trees (82). Additionally the cost function used by the PQO has to take into account the partitioning and communication costs, the placement of the data, and the execution skew. Therefore several algorithms have been introduced for parallel query optimization.

In Ref. 66 opportunities in the parallelism of left-deep (left-linear) and right-deep (right-linear) query trees (Fig. 14) in light of multi-way joins are discussed. For binary join operations the hash join method is used, because it is the best possible choice for parallel execution. This technique consists of two phases: build and probe. In the build phase the inner-join operand is used to create a hash table in main memory. If the hash table exceeds the memory capacity, the overflow tuples are stored to a temporary file on disk. During the probe phase the outer-join operand is used to probe the hash table or the portion of the hash table on the disk. The inner-join operand is called “left operand,” and in the same fashion the outer-join operand is termed “right operand.” In the right-deep query tree, the build phase can be executed in parallel for all join operations, and the probe phases can be executed using extensive pipelining. On the other hand, left-deep trees allow the execution of the probe phase of only one join and the build phase of the next join in the tree at the same time. Hence right-deep query representations are better suited to exploit the parallelism offered by PDSs.

The result above is extended for bushy query trees in Ref. 83. Right-deep trees may suffer from low flexibility of structure, thus implying a limitation on performance. A major problem for pure right-deep trees is that the amount of main-memory available may not be enough to accommodate all the inner relations during the build phase. Hence the right-deep tree has to be decomposed into disjoint segments so that the inner relations of each segment can fit into memory. Bushy trees offer greater flexibility in the generation of query plans at the cost of a larger search space. It has been shown that for sort–merge, the evaluation of a bushy tree can outperform that of the linear trees. However, in the case of hash join, the scheduling of a bushy query tree is much more complex than the corresponding right-deep structure. The problem here is that the execution of join operation should be synchronized in order to fully exploit pipelining. Therefore the use of segmented right-deep trees for the execution of pipelined hash joins is suggested in Ref. 83. A segmented right-deep tree is a bushy tree that consists of right-deep segments. These segments can be evaluated using the approach described in Ref. 66. Each segment is assigned to a set of processors where the size of the set is proportional to the estimated amount of work in the join operations. Thus independent segments can be executed in parallel using sets of disjoint processors.

In Ref. 84 a performance study is provided for four different execution strategies for multi-join queries, using the main-memory PDS PRISMA/DB (85). There are four strategies examined:

- *Sequential Execution Strategy (SP).* This is the simplest way to evaluate a multi-join query using intra-operator, but not interoperator, parallelism. Here join-operators are evaluated one after the other using all available processors. Since there is no pipelining used, the intermediate results have to be stored. In PRISMA these results are kept in main-memory, and this is the main reason for the competitiveness of this strategy.

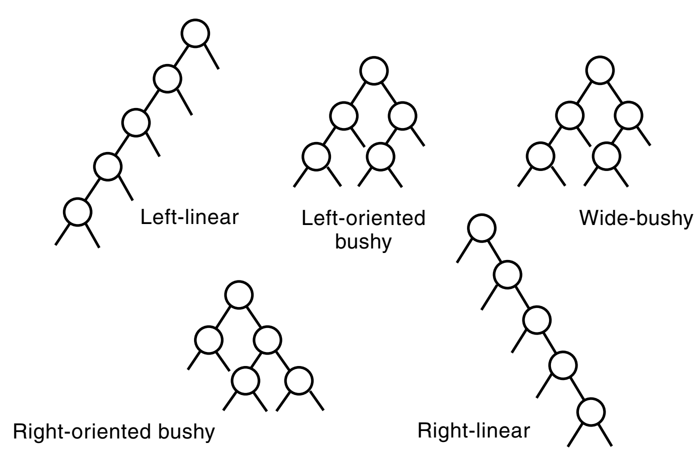

Figure 14. Types of query trees.

- *Synchronous Execution (SE).* The rationale here is to execute independent subtrees of the query tree using independent parallelism.
- *Segmented Right-Deep Execution (RD).* This is the query processing method discussed earlier and was proposed in Ref. 83.
- *Full Parallel Execution (FP).* Both pipelining and independent parallelism are added to partitioned parallelism in the individual join operators. Here each join operator is assigned to a private group of processors, so all join operators are executed in parallel. Depending on the shape of the query tree, pipelining and independent parallelism are used.

All strategies but the first offer imperfect load balancing. The query tree shapes used in the experiments were left-linear, left-oriented bushy, wide-bushy, right-oriented bushy, and right-linear (Fig. 14). The experimental results indicate that for a small number of processors the SP strategy is the cheapest one as intermediate results are buffered. For larger number of processors, the FP strategy outperforms the others. The performances of the SE and RD depend on the shape of the query tree. In particular, RD does not work well for trees with left-deep segments. However, it is possible to transform, with little cost, a query tree to a more right-oriented one. In this case the RD strategy can work very effectively. In terms of memory consumption, the RD appears to be better than the FP. Among the different query-tree shapes, the most competitive seems to be the bushy tree, since it allows for more effective parallelization.

A different approach in PDS query processing is discussed in Ref. 86 where the problem is decomposed into two phases: join ordering and query rewrite (JOQR), and parallelization. The rationale of this approach resembles that followed in the compilation of programming languages where the problem is fragmented into several distinct phases in order to deal effectively with the problem's complexity and provide easy implementation.

The first phase, JOQR, produces an annotated query tree that fixes the order of operators and the join computing methods. This phase is similar to traditional (centralized) query optimization, and a conventional query optimizer can be used. In accordance with the design of traditional optimizers, this phase can be further broken into two steps:

- The first rewrites the submitted query using heuristics (algebraic transformation rules).
- The second arranges the ordering operations and selects the method to compute each operation (e.g., the method to compute the joins).

In JOQR an important issue is the choice of the partitioning attributes in the query tree so that the total sum of communication and computation costs is minimized. In Ref. 86 this problem is reduced to a query tree coloring problem. Here the partitioning attributes are regarded as colors, and the repartitioning cost is saved when adjacent operators have the same color. Subsequently the costs function considers communication and computation costs, access methods expenses, if any, and finally costs for strategies that compute each operator. These algorithms also deal with queries that include grouping, aggregation, and other operations usually contained in DSS and warehousing queries.

The second phase of the approach takes as input the annotated query tree produced and returns a query execution plan:

- The first step translates the annotated query tree to an operator tree by "macroexpansion." The nodes of an operator tree represent operators and the edges represent the flow as well as timing constraints between operators. These operators are considered as atomic pieces of code by the scheduler.
- The second step schedules the operation tree on the parallel machine's nodes, while respecting the precedence constraints and the data placement.

SUMMARY

We have examined three families of database architectures used to satisfy the unique requirements of diverse real-world environments. The architectures optimize database processing by taking advantage of available computing resources and exploiting application characteristics.

To deliver real-time responses and high-throughput rates, main-memory databases have been developed, on the assumption that most of their operational data are available in volatile memory at all times. This is not an unrealistic assumption as only a small fraction of any application's data space is utilized at any given moment. The absence of frequent disk accesses has led to the design of concurrency and transaction processing techniques specifically tuned to perform well in the main-memory environment.

The widespread availability of workstations and high-end PCs coupled with the presence of high-speed networking options have led to the evolution of client–server systems. Empirical observations have indicated that most database users access small and likely disjoint portions of the data. In addition these data portions are accessed with a much greater frequency than the rest of the database. The desire to off-load such localized processing from database servers to the clients' own workstations has led to the development of client–server database architectures. Initial implementations utilized client machines as user-interface points only. However, the increasing processing capabilities of PCs and workstations have allowed clients to not only be able to cache data but also perform database processing. Caching could be of either an ephemeral or long-term nature. In the former, the clients' buffer space is used as a temporary storage area for data. In the latter, the clients' full memory hierarchy is used to store server-originating data not only in main memory but in the disk units as well (i.e., disk-caching).

In the absence of localized database accesses or when the volume of data to be processed is massive, parallel databases offer an appropriate architecture for efficient database processing. Parallel database systems offer high performance and high availability by using tightly or loosely connected multiprocessor systems and I/O devices. The aggregate ultrahigh CPU processing capabilities and the I/O bandwidth of such systems offer numerous opportunities for parallelism in database processing. This parallelism is achieved by first declustering data among the I/O units and then optimizing processing through pipelined, partitioned, and independent parallelism.

Each of the architectures above is radically different from those used in conventional centralized database systems. The advantages offered by each configuration are often traded off with more complex concurrency control and recovery mechanisms. Research efforts in the past few years have aimed at reducing such overheads and, at the same time, concentrated on devising specialized solutions (both software and hardware) to improve their performance characteristics. We have presented a number of key issues involved in the implementation of such database architectures and outlined recent advances.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under Grants NSF IIS-9733642 and NSF IRI-9509527, and by the Center for Advanced Technology in Telecommunications (CATT), Brooklyn, NY.

BIBLIOGRAPHY

1. Committee to Study the Impact of Information Technology on the Performance of Service Activities, *Information Technology in the Service Society, A Twenty-First Century Lever,* Washington, DC: National Academy Press, 1994.
2. M. Eich, A classification and comparison of main-memory database recovery techniques, *Proc. IEEE Int. Conf. Data Eng.,* 1987, pp. 332–339.
3. J. E. B. Moss, Working with persistent objects: To swizzle or not to swizzle, *IEEE Trans. Softw. Eng.,* **18**: 103–139, 1992.
4. S. J. White and D. J. DeWitt, A performance study of alternative object faulting and pointer swizzling strategies, *Proc. 18th Int. Conf. Very Large Data Bases,* Vancouver, BC, Canada, 1992.
5. T. Lehman, E. Shekita, and L. F. Cabrera, An evaluation of Starburst's memory resident storage component, *IEEE Trans. Knowl. Data Eng.,* **4**: 555–566, 1992.
6. T. Lehman and M. Carey, Query processing in main-memory database systems, *Proc. ACM SIGMOD Conf.,* Washington, DC, 1986.
7. H. V. Jagadish et al., Dalí: A high performance main memory storage manager, *Proc. 20th Int. Conf. Very Large Data Bases,* Santiago, Chile, 1994.
8. M. Stonebraker, Managing persistent objects in a multi-level store, *Proc. ACM SIGMOD Conf.,* Denver, CO, 1991.
9. A. Delis and Q. LeViet, Contemporary access structures under mixed-workloads, *Comput. J.,* **40** (4): 183–193, 997.
10. K. Y. Whang and E. Krishnamurthy, Query optimization in a memory-resident domain relational calculus database system, *ACM Trans. Database Syst.,* **15** (1): 67–95, 1990.
11. W. Litwin and T. Rische, Main-memory oriented optimization of OO queries using typed data-log with foreign predicates, *IEEE Trans. Knowl. Data Eng.,* **4**: 517–528, 1992.
12. H. Garcia-Molina and K. Salem, Main memory database systems: An overview, *IEEE Trans. Knowl. Data Eng.,* **4**: 509–516, 1992.
13. N. Roussopoulos, The incremental access method of view cache: Concept, algorithms, and cost analysis, *ACM Trans. Database Syst.,* **16**: 535–563, 1991.
14. V. Gottemukkala and T. Lehman, Locking and latching in a memory-resident database system, *Proc. 18th Int. Conf. Very Large Data Bases,* Vancouver, BC, Canada, 1992.
15. K. Salem and H. Garcia-Molina, System M: A transaction processing testbed for memory resident data, *IEEE Trans. Knowl. Data Eng.,* **2**: 161–172, 1990.
16. X. Li and M. H. Eich, Post-crash log processing for fuzzy checkpointing main-memory databases, *Proc. 9th IEEE Conf. Data Eng.,* Vienna, 1993, pp. 117–124.
17. H. V. Jagadish, A. Silberschatz, and S. Sudarshan, Recovering from main-memory lapses, *Proc. 19th Int. Conf. Very Large Data Bases,* Dublin, Ireland, 1993, pp. 391–404.
18. D. J. DeWitt et al., Implementation techniques for main memory database systems, *Proc. ACM Conf.,* 1984.
19. L. Gruenwald and M. H. Eich, MMDB reload algorithms, *Proc. ACM SIGMOD Conf.,* Denver, CO, 1991.
20. L. Gruenwald and M. H. Eich, MMDB reload concerns, *Inf. Sci.,* **76**: 151–176, 1994.
21. R. Stevens, *Unix Network Programming,* Englewood Cliffs, NJ: Prentice-Hall, 1991.
22. D. Comer and D. Stevens (eds.), *Internetworking with TCP/IP,* Vol. 3, Englewood Cliffs, NJ: Prentice-Hall, 1993.
23. A. Delis and N. Roussopoulos, Performance and scalability of client–server database architectures, *Proc. 18th Int. Conf. Very Large Data Bases,* Vancouver, BC, Canada, 1992.
24. H. Korth and A. Silberschatz, *Database System Concepts,* 2nd ed., New York: McGraw-Hill, 1991.
25. W. Rubenstein, M. Kubicar, and R. Cattell, Benchmarking simple database operations, *ACM SIGMOD Conf. Manage. Data,* Chicago, 1987, pp. 387–394.
26. D. DeWitt et al., A study of three alternative workstation–server architectures for object-oriented database systems, *Proc. 16th Int. Conf. Very Large Data Bases,* 1990, pp. 107–121.
27. M. Carey et al., Data caching tradeoffs in client–server DBMS architecture, *ACM SIGMOD Conf. Manage. Data,* Denver, CO, 1991.
28. A. Delis and N. Roussopoulos, Performance comparison of three modern DBMS architectures, *IEEE Trans. Softw. Eng.,* **19**: 120–138, 1993.
29. M. Franklin, M. Carey, and M. Livny, Local disk caching for client–server database systems, *Proc. 19th Int. Conf. Very Large Data Bases,* Dublin, 1993.
30. U. Deppisch and V. Obermeit, Tight database cooperation in a server–workstation environment, *Proc. 7th IEEE Int. Conf. Distrib. Comput. Syst.,* 1987, pp. 416–423.
31. R. Alonso, D. Barbara, and H. Garcia-Molina, Data caching issues in an information retrieval system, *ACM Trans. Database Syst.,* **15** (3): 359–384, 1990.
32. A. Delis and N. Roussopoulos, Management of updates in the enhanced client–server DBMS, *Proc. 14th IEEE Int. Conf. Distrib. Comput. Syst.,* 1994.
33. J. Howard et al., Scale and performance in a distributed file scale, *ACM Trans. Comput. Syst.,* **6** (1): 51–81, 1988.
34. M. Nelson, B. Welch, and J. Ousterhout, Caching in the Sprite network file system, *ACM Trans. Comput. Syst.,* **6** (1): 134–154, 1988.
35. A. Biliris and J. Orenstein, Object storage management architectures, *NATO ASI Ser., Ser. F,* **130**: 185–200, 1994.
36. I. S. Chu and M. S. Winslett, Choices in database workstation–server architecture, *Proc. 17th Annu. Int. Comput. Softw. Appl. Conf.,* Phoenix, AZ, 1993.
37. M. Franklin, M. Carey, and M. Livny, Global memory management in client–server DBMS architectures, *Proc. 18th Int. Conf. Very Large Data Bases,* Vancouver, BC, Canada, 1992.
38. A. Leff, P. Yu, and J. Wolf, Policies for efficient memory utilization in a remote caching architecture, *Proc. 1st Conf. Parallel Distrib. Inf. Syst.,* Los Alamitos, CA, 1991.

39. M. Carey, M. Franklin, and M. Zaharioudakis, Fine-grained sharing in a page server OODBMS, *Proc. ACM SIGMOD Conf.,* Minneapolis, MN, 1994.
40. M. Hornick and S. Zdonik, A shared, segmented memory system for an object-oriented database, *ACM Trans. Off. Inf. Syst.,* **5** (1): 70–95, 1987.
41. K. Wilkinson and M. A. Niemat, Maintaining consistency of client-cached data, *Proc. 16th Int. Conf. Very Large Data Bases,* Brisbane, Australia, 1990, pp. 122–133.
42. Y. Wang and L. Rowe, Cache consistency and concurrency control in a client/server DBMS architecture, *Proc. ACM SIGMOD Int. Conf.,* Denver, CO, 1991.
43. A. Adya et al., Efficient optimistic concurrency control using loosely synchronized clocks, *Proc. ACM Int. Conf. Manage. Data,* San Jose, CA, 1995.
44. B. Liskov et al., A highly available object repository for use in a heterogenous distributed system, *Proc. 4th Int. Workshop Persistent Object Syst.,* 1990, pp. 255–266.
45. T. Haerder, Observations on optimistic concurrency control schemes, *Inf. Syst.,* **9** (2): 111–120, 1984.
46. K. Korner, Intelligent caching for remote file service, *Proc. 10th IEEE Int. Conf. Distrib. Comput. Syst.,* Paris, 1990.
47. D. Terry, Caching hints in distributed systems, *IEEE Trans. Softw. Eng.,* **SE-13**: 48–54, 1987.
48. K. Chen, R. Bunt, and D. Eager, Write caching in distributed file systems, *Proc. 15th IEEE Int. Conf. Distrib. Comput. Syst.,* 1995, pp. 457–466.
49. Y. Huang, R. Sloan, and O. Wolfson, Divergence caching in client–server architectures, *Proc. 3rd Int. Conf. Parallel Distrib. Syst.,* 1994, pp. 131–139.
50. C. Pu et al., Performance comparison of active-sender and active-receiver policies for distributed caching, *Proc. 1st Int. Symp. High-Perform. Distrib. Comput.,* 1992, pp. 218–227.
51. J. O'Toole and L. Shrira, Shared data management needs adaptive methods, *Proc. 5th Workshop Hot Top. Oper. Syst.,* 1995, pp. 129–135.
52. J. O'Toole and L. Shrira, Opportunistic log: Efficient reads in a reliable object server, *Proc. 1st Conf. Oper. Syst. Des. Implement.,* Tarascon, Provence, France, 1994, pp. 99–114.
53. T. Sellis and C. Lin, *A Study of Predicate Indexing for DBMS Implementations of Production Systems,* Tech. Rep., College Park: University of Maryland, 1991.
54. A. Keller and J. Basu, A predicate-based caching scheme for client-server database architectures, *VLDB J.,* **5** (1): 35–47, 1996.
55. M. Franklin et al., Crash recovery in client–server EXODUS, *Proc. ACM SIGMOD Conf.,* San Diego, CA, 1992.
56. C. Mohan et al., ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging, *ACM—Trans. Database Syst.,* **17** (1): 94–162, 1992.
57. C. Mohan and I. Narang, ARIES/CSA: A method for database recovery in client–server architectures, *Proc. ACM-SIGMOD Int. Conf. Manage. Data,* Minneapolis, MN, 1994, pp. 55–66.
58. C. Mohan, Commit_LSN: A novel and simple method for reducing locking and latching in transaction processing systems, *Proc. 16th Int. Conf. Very Large Data Bases,* Brisbane, Australia, 1990.
59. E. Panagos et al., Client-based logging for high performance distributed architectures, *Proc. 12th Int. Conf. Data Eng.,* New Orleans, LA, 1996, pp. 344–351.
60. D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach,* 2nd ed., San Mateo, CA: Morgan-Kaufman, 1996.
61. T. Johnson and D. Shasha, Some approaches to index design for cube forest, *IEEE Data Eng. Bull.,* **20** (1): 27–35, 1997.
62. R. Agrawal, C. Faloutsos, and A. Swami, Efficient similarity search in sequence databases, *Proc. Int. Conf. Found. Data Organ. Algorithms (FODO),* Chicago, 1993, pp. 69–84.
63. V. J. Tsotras, B. Gopinath, and G. W. Hart, Efficient management of time-evolving databases, *IEEE Trans. Knowl. Data Eng.,* **7**: 591–608, 1995.
64. P. Valduriez, Parallel database systems: The case for shared-something, *Proc. 9th IEEE Int. Conf. Data Eng.,* Vienna, 1993, pp. 460–465.
65. D. DeWitt and J. Gray, Parallel database systems: The future of high performance database systems, *Commun. ACM,* **35** (6): June, 1992.
66. D. Schneider and D. DeWitt, Tradeoffs in processing complex join queries via hashing in multiprocessor database machines, *Proc. Very Large Data Bases Conf.,* Brisbane, Australia, 1990.
67. H. Hsiao and D. DeWitt, Chained declustering: A new availability strategy for multiprocessor database machines, *Proc. 6th Conf. Data Eng.,* Los Angeles, CA, 1990, pp. 456–465.
68. M. Stonebraker, The case for shared nothing, *IEEE Database Eng. Bull.,* **9** (1): 4–9, 1986.
69. M. Stonebraker (ed.), *Readings in Database Systems,* San Mateo, CA: Morgan-Kaufmann, 1988.
70. B. Bergsten, M. Couprie, and P. Valduriez, Prototyping DBS3, a shared-memory parallel database system, *Proc. Int. Conf. Parallel Distrib. Inf. Syst.,* Miami Beach, FL, 1991, pp. 226–234.
71. G. Graefe, Encapsulation of parallelism in the volcano query processing systems, *Proc. ACM SIGMOD Int. Conf.,* Atlantic City, NJ, 1990, pp. 102–111.
72. P. Chen et al., RAID: High-performance, reliable secondary storage, *ACM Comput. Surv.,* **26** (2): 145–186, 1994.
73. C. Baru et al., DB2 parallel edition, *IBM Syst. J.,* **34** (2): 292–322, 1995.
74. D. DeWitt et al., The Gamma database machine project, *IEEE Trans. Data Knowl. Eng.,* **2**: March, 1990.
75. H. Boral et al., Prototyping bubba, a highly parallel database system, *IEEE Trans. Data Knowl. Eng.,* **2**: 4–24, 1990.
76. K. A. Hua, C. Lee, and J.-K. Peir, Interconnecting shared-everything systems for efficient parallel query processing, *Parallel Distrib. Inf. Syst. (PDIS),* Los Alamitos, CA, 1991, pp. 262–270.
77. G. Copeland et al., Data placement in bubba, *Proc. ACM SIGMOD Conf.,* Chicago, 1988.
78. S. Ghandeharizadeh and D. J. DeWitt, Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines, *Proc. 16th Int. Conf. Very Large Data Bases,* Brisbane, Australia, 1990, pp. 481–492.
79. M. Mehta and D. J. DeWitt, Data placement in shared-nothing parallel database system, *VLDB J.,* **6** (1): 53–72, 1997.
80. W. Hasan, D. Florescu, and P. Valduriez, Open issues in parallel query optimization, *ACM SIGMOD Rec.,* **25** (3): 28–33, 1996.
81. P. Selinger et al., Access path selection in a relational data base system, *ACM SIGMOD Conf. Manage. Data,* Boston, 1980.
82. R. Krishnamurthy, H. Boral, and C. Zaniolo, Optimization of nonrecursive queries, *Proc. Very Large Databases Conf.,* Kyoto, Japan, 1986, pp. 128–137.
83. M. S. Chen et al., Using segmented right-deep trees for the execution of pipelined hash joins, *Proc. 18th Int. Conf. Very Large Data Bases,* Vancouver, BC, Canada, 1992.
84. A. N. Wilschut, J. Flokstra, and P. M. G. Apers, Parallel evaluation of multi-join queries, *Proc. ACM SIGMOD Int. Conf. Manage. Data,* San Jose, CA, 1995, pp. 115–126.
85. P. M. G. Apers et al., PRISMA/DB: A parallel main memory relational DBMS, *IEEE Trans. Knowl. Data Eng.,* **4**: 541–554, 1992.

86. W. Hasan and R. Motwani, Coloring away communication in parallel query optimization, *Proc. 21st Int. Conf. Very Large Data Bases,* Zurich, 1995, pp. 239–250.

ALEX DELIS
VINAY KANITKAR
GEORGE KOLLIOS
Polytechnic University