Introduction to a System for Distributed Databases (SDD-1)

J. B. ROTHNIE, JR., P. A. BERNSTEIN, S. FOX, N. GOODMAN, M. HAMMER, T. A. LANDERS, C. REEVE, D. W. SHIPMAN, and E. WONG
Computer Corporation of America

The declining cost of computer hardware and the increasing data processing needs of geographically dispersed organizations have led to substantial interest in distributed data management. SDD-1 is a distributed database management system currently being developed by Computer Corporation of America. Users interact with SDD-1 precisely as if it were a nondistributed database system because SDD-1 handles all issues arising from the distribution of data. These issues include distributed concurrency control, distributed query processing, resiliency to component failure, and distributed directory management. This paper presents an overview of the SDD-1 design and its solutions to the above problems.

This paper is the first of a series of companion papers on SDD-1 (Bernstein and Shipman [2], Bernstein et al. [4], and Hammer and Shipman [14]).

Key Words and Phrases: distributed database system, relational data model, concurrency control, query processing, database reliability
CR Categories: 3.5, 4.33

1. INTRODUCTION

SDD-1 is a distributed database management system under development by Computer Corporation of America. SDD-1 is a system for managing databases whose storage is distributed over a network of computers. Functionally, SDD-1 provides the same capabilities that one expects of any modern database management system (DBMS), and users interact with it precisely as if it were not distributed.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract N00039-77-C-0074, ARPA Order No. 3175-6. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Authors' present addresses: J. B. Rothnie, Jr., S. Fox, T. A. Landers, C. Reeve, and D. W. Shipman, Computer Corporation of America, 575 Technology Square, Cambridge, MA 02139; P. A. Bernstein and N. Goodman, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138; M. Hammer, Massachusetts Institute of Technology, Laboratory for Computing Science, 545 Technology Square, Cambridge, MA 02139; E. Wong, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720.

Systems like SDD-1 are appropriate for applications which exhibit two characteristics: First, the activity requires an integrated database. That is, the activity entails access to a single pool of information by multiple persons, organizations, or programs. And second, either the users of the information or its sources are distributed geographically. Many data processing applications have these characteristics, including

(1) inventory control, accounting, personnel information, and similar data processing systems in large companies;
(2) point-of-sale accounting systems, electronic banking, and other consumer-oriented on-line processing systems;
(3) large-scale data resources, e.g., census, climatology, toxicology, and similar databases;
(4) military intelligence databases, and command and control systems;
(5) report-generating systems for businesses with multiple data processing centers; and so forth.

Decentralized processing is desirable in these applications for reasons of performance, reliability, and flexibility of function. *Centralized control* is needed to ensure operation in accordance with overall policy and goals. By meeting both these goals in one system, distributed database management offers unique benefits.

However, distributed database systems pose new technical challenges owing to their inherent requirements for data communication and their inherent potential for parallel processing. The principal bottleneck in these systems is data communication. All economically feasible long-distance communication media incur lengthy delays and/or low bandwidth. Moreover, the cost of moving data through a network is comparable to the cost of storing the data locally for many days. Parallel processing is also an inherent aspect of distributed systems and mitigates to some extent the communication factor. However, it is often difficult to construct algorithms that can exploit parallelism.

For these reasons, the techniques used to implement centralized DBMSs must be reexamined in the distributed DBMS context. We have done this in developing SDD-1, and this paper outlines our main results.

Section 2 describes SDD-1's overall architecture and the flow of events in processing transactions. Sections 3 through 5 then introduce the techniques used by SDD-1 for solving the most difficult problems in distributed data management: concurrency control, query processing, and reliability. Detailed discussions of these techniques are presented in [2–4, 12, 14, 25]. Section 6 explains how these techniques are used to handle the management of system directories. The paper concludes with a brief history of SDD-1 and a summary of its principal contributions to the field.

2. SYSTEM ORGANIZATION

2.1 Data Model

SDD-1 supports a relational data model [8]. Users interact with SDD-1 in a high-level language called Datalanguage [9] which is illustrated in Figure 1. Datalanguage differs from relational languages such as QUEL [15] or SEQUEL [7]

```
Relation:  CUSTOMER (Name, Branch, Acct #, SavBal, ChkBal, LoanBal)
Command:  Update C in CUSTOMER with C.Name = "Adams"
          Begin
              C.SavBal = C.SavBal – 100
              C.ChkBal = C.ChkBal + 100
          End;
```

Fig. 1. A Datalanguage command.

primarily in its use of "declared" variables. This construct and related control structures expand the power of Datalanguage to that of a general-purpose programming language. For purposes of this paper, the differences between Datalanguage and QUEL or SEQUEL are not important, and for pedagogic ease, we adopt QUEL terminology.

Datalanguage may be used as a query language for end-users but is more typically invoked by host programs. Datalanguage is embedded in host programs in essentially the same manner as QUEL or SEQUEL. That is, the host program issues self-contained Datalanguage commands to SDD-1, which processes these commands exactly as if entered by an end-user.

A single Datalanguage command is called a *transaction* (e.g., the command shown in Figure 1 is a transaction). Transactions are the units of atomic interaction between SDD-1 and the external world. This concept of transaction is similar to that of INGRES [15] and System R [1].

An SDD-1 database consists of (logical) relations. Each SDD-1 relation is partitioned into subrelations called *logical fragments*, which are the units of data distribution. Logical fragments are defined in two steps. First, the relation is partitioned horizontally into subsets defined by "simple" restrictions.[1] Then each horizontal subset is partitioned into subrelations defined by projections (see Figures 2 and 3). To reconstruct the logical relation from its fragments, a unique tuple identifier is appended to each tuple and included in *every* fragment [10, 21].

Logical fragments are the units of data distribution, meaning that each may be stored at any one or several sites in the system. Logical fragments are defined and the assignment of fragments to sites is made when the database is designed. A stored copy of a logical fragment is called a *stored fragment.*

Note that user transactions are unaware of data distribution or redundancy. They reference only relations, not fragments. It is SDD-1's responsibility to translate from relations to logical fragments, and then to select the stored fragments to access in processing any given transaction.

2.2 General Architecture

SDD-1 is a collection of three types of *virtual machines* [16]—Transaction Modules (TMs), Data Modules (DMs), and a Reliable Network (RelNet)—configured as in Figure 4.

[1] A simple restriction is a Boolean expression whose clauses are of the form ⟨attribute⟩ ⟨rel-op⟩ ⟨constant⟩, where ⟨rel-op⟩ is =, ≠, >, <, etc.

CUSTOMER	(Name,	Branch,	Acct.#,	SavBal,	ChkBal,	LoanBal)
CUST_1	Washington	1	1234	$100	$200	−$8
CUST_2	Jefferson	2	5678	$200	$300	$30000
CUST_3a	Adams	3	9012	$1000	$0	$20000
CUST_3b	Monroe	3	3456	$100	$50	$0

CUST_1 = CUSTOMER where Branch = 1
CUST_2 = CUSTOMER where Branch = 2
CUST_3a = CUSTOMER where Branch = 3 and LoanBal ≠ 0
CUST_3b = CUSTOMER where Branch = 3 and LoanBal = 0

Fig. 2. Horizontal partitioning.

CUSTOMER	(Name,	Branch,	Acct#,	SavBal,	ChkBal,	LoanBal)
CUST_1	CUST_1.1		CUST_1.2			
CUST_2	CUST_2.1			CUST_2.2		
CUST_3a	CUST_3a.1		CUST_3a.2		CUST_3a.3	
CUST_3b	CUST_3b.1		CUST_3b.2		CUST_3b.3	CUST_3b.4

CUST_1.1 = CUST_1 [Name, Branch]
CUST_1.2 = CUST_1 [Acct#, SavBal, ChkBal, LoanBal]
etc.

Fig. 3. Vertical partitioning. In order to reconstruct CUSTOMER from its fragments, a unique tuple identifier is appended to each tuple and included in *every* fragment [21].

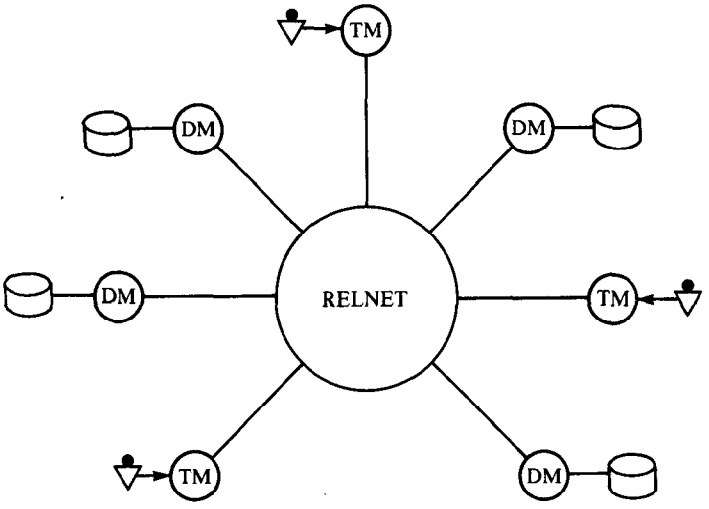


Fig. 4. SDD-1 configuration.

All data managed by SDD-1 are stored by Data Modules (DMs). DMs are, in effect, back-end DBMSs that respond to commands from Transaction Modules. DMs respond to four types of commands: (1) *read* part of the DM's database into a local workspace at that DM; (2) *move* part of a local workspace from this DM to another DM; (3) *manipulate* data in a local workspace at the DM; (4) *write* part of the local workspace into the permanent database stored at the DM.

Transaction Modules (TMs) plan and control the distributed execution of transactions. Each transaction processed by SDD-1 is supervised by some TM which performs the following tasks:

(1) *Fragmentation.* The TM translates queries on relations into queries on logical fragments and decides which instances of stored fragments to access.

(2) *Concurrency control.* The TM synchronizes the transaction with all other active transactions in the system.

(3) *Access planning.* The TM compiles the transaction into a parallel program which can be executed cooperatively by several DMs.

(4) *Distributed query execution.* The TM coordinates execution of the compiled access plan, exploiting parallelism whenever possible.

The third SDD-1 virtual machine is the Reliable Network (RelNet) which interconnects TMs and DMs in a robust fashion. The RelNet provides four services:

(1) *guaranteed delivery*, allowing messages to be delivered even if the recipient is down at the time the message is sent, and even if the sender and receiver are *never* up simultaneously;
(2) *transaction control*, a mechanism for posting updates at multiple DMs, guaranteeing that either *all* DMs post the update *or none* do;
(3) *site monitoring*, to keep track of which sites have failed, and to inform sites impacted by failures;
(4) *network clock*, a virtual clock kept approximately synchronized at all sites.

This architecture divides the distributed DBMS problem into three pieces: database management, management of distributed transactions, and distributed DBMS reliability. By implementing each of these pieces as a self-contained virtual machine, the overall SDD-1 design is substantially simplified.

2.3 Run-Time Structure

Among the functions required to execute a transaction in a distributed DBMS, three are especially difficult: concurrency control, distributed query processing, and reliable posting of updates. SDD-1 handles each of these problems in a distinct processing phase, so that each can be solved independently. Consider transaction T of Figure 1. When T is submitted to SDD-1 for processing, the system invokes a three-phase processing procedure. The phases are called *Read*, *Execute*, and *Write*.

The first phase is the *Read phase* and exists for purposes of concurrency control. The TM that is supervising T analyzes it to determine which portions of the (logical) database it reads, called its *read-set*. In this case the TM would

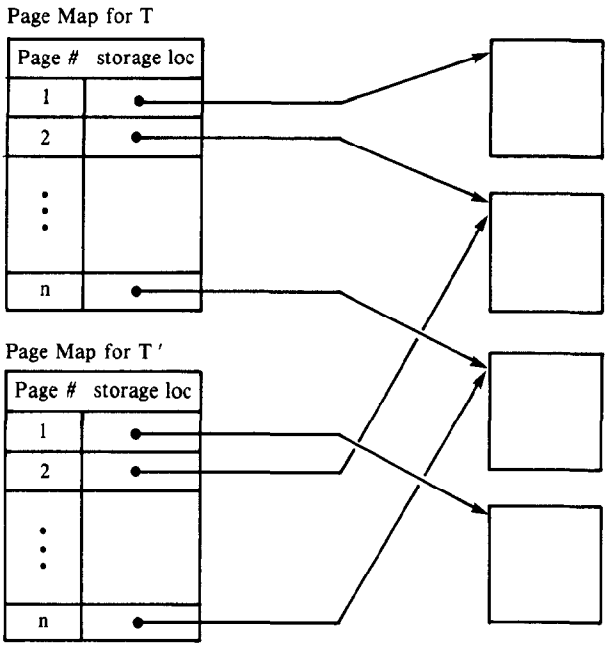


Fig. 5. Version to storage mapping. T and T′ share each page until one transaction or the other modifies the page.

determine that T's read-set is

$$\{\text{C.SavBal, C.ChkBal} \mid \text{C.Name} = \text{"Adams"}\}.$$

In addition, the TM decides which stored fragments to access to obtain that data. Then the TM issues Read commands to the DMs that house those fragments, instructing each DM to set aside a private copy of that fragment for use during subsequent processing phases.

The private copies obtained by the Read phase are guaranteed to be *consistent* even though the copies reside at distributed sites. The techniques for guaranteeing consistency are described in Section 3. Since the data are consistent when read, and since the copies are private, subsequent phases can operate freely on these data without fear of interference from other transactions.

We emphasize that no data are actually transferred between sites during the Read phase. Each DM simply sets aside the specified data in a workspace at the DM. Moreover, in each DM, the private workspace is implemented using a differential file mechanism [22], so data are not actually copied. This mechanism operates as follows. The primary organization of a stored fragment is a *paged file*, much like a UNIX [20] or TENEX [6] file. A *page* is a unit of logical storage; a *page map* is a function that associates a physical storage location with each page (see Figure 5). The "private copy" set aside by a Read command is in reality a page map. Page maps behave like private copies because *pages are never updated in place*; if page P is modified on behalf of transaction T′, say, a new block of secondary storage is allocated, and the modified page is written there. T′ is able to access the modified page because its page map is also modified to reflect

P's new storage location. Other transactions are unaffected because their page maps remain unchanged. A similar mechanism is described in [18].

The second phase, called the *Execute phase*, implements distributed query processing. At this time, the TM compiles T into a distributed program that takes as input the distributed workspace created by the Read phase. This compilation procedure is described in Section 4. The compiled program consists of Move and Manipulate commands which cause the DMs to perform the intent of T in a distributed fashion. The compiled program is supervised by the TM to ensure that commands are sent to DMs in the correct order and to handle run-time errors.

The output of the program is a list of data items to be written into the database (in the case of update transactions) or displayed to the user (in the case of retrievals). In our example, this output list would contain a unique tuple identifier for Adams' tuple, identifiers for the field names SavBal and ChkBal, and the new values for these fields. This output list is produced in a workspace (i.e., temporary file) at one DM, and is not yet installed into the permanent database. Consequently, problems of concurrency control and reliable writing are irrelevant during this phase.

The final phase, called the *Write phase*, installs data modified by T into the permanent database and/or displays data retrieved by T to the user. For each entry in the output list, the TM determines which DM(s) contain copies of that data item. The TM orders the final DM that holds the output list to send the appropriate entries of the output list to each DM; it then issues Write commands to each of these DMs thereby causing the new values to be installed into the database. Techniques described in Section 5 are used during the Write phase to ensure that partial results are not installed or displayed even if multiple sites or communication links fail in midstream. This is the most difficult aspect of distributed DBMS reliability, and by separating it into a distinct phase, we simplify both it and the other phases.

The three-phase processing of transactions in SDD-1 neatly partitions the key technical problems of distributed database management. The next sections of this paper explain how SDD-1 solves each of these independent problems.

3. CONCURRENCY CONTROL

The problems that arise when multiple users access a shared database are well known. Generically there are two types of problems: (1) If transaction T1 is reading a portion of the database while transaction T2 is updating it, T1 might read inconsistent data (see Figure 6). (2) If transactions T3 and T4 are both updating the database, race conditions can produce erroneous results (see Figure 7). These problems arise in all shared databases—centralized or distributed—and are conventionally solved using *database locking*. However, we have developed a new technique for SDD-1.

3.1 Methodology

SDD-1, like most other DBMSs, adopts *serializability* as its criterion for concurrent correctness. Serializability requires that whenever transactions execute concurrently, their effect must be identical to some serial (i.e., noninterleaved)

Consider the database of Figures 2 and 3, and assume fragments CUST_3a.2, CUST_3a.3, are stored at different DMs:

Let transaction T1 be
  Range of C is CUSTOMER;
  Retrieve C (SavBal + ChkBal) where C.Name = "Adams";
Let transaction T2 be
  Range of C is CUSTOMER;
  Replace C(SavBal = SavBal − $100, ChkBal = ChkBal + $100) where C.Name = "Adams";

And suppose T1 and T2 execute in the following concurrent order
  T1 reads Adams' SavBal (= $1000) from fragment CUST_3a.2
  T2 writes Adams' SavBal (= $900) into fragment CUST_3a.2
  T2 writes Adams' ChkBal (= $100) into fragment CUST_3a.3
  T1 reads Adams' ChkBal (= $100) from fragment CUST_3a.3

T1's output will be $1000 + $100 = $1100, which is incorrect.

Fig. 6. Reading inconsistent data.

Given the database of Figures 2 and 3.

Let transaction T3 be
  Range of C is CUSTOMER;
  Replace C (ChkBal = ChkBal + $100) where C.Name = "Monroe";
Let transaction T4 be
  Range of C is CUSTOMER;
  Replace C (ChkBal = ChkBal − $50) where C.Name = "Monroe";

And suppose T3 and T4 execute in the following concurrent order
  T3 reads Monroe's ChkBal (= $50)
  T4 reads Monroe's ChkBal (= $50)
  T4 writes Monroe's ChkBal (= $0)
  T3 writes Monroe's ChkBal (= $50 + $100 = $150)

The value of ChkBal left in the database is $150, which is incorrect. The final balance should be $50 − $50 + $100 = $100.

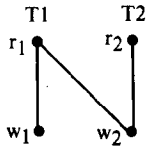
Fig. 7. Race condition producing erroneous update.

execution of those same transactions. This criterion is based on the assumption that each transaction maps a consistent database state into another consistent state. Given this assumption, every serial execution preserves consistency. Since a serializable execution is equivalent to a serial one, it too preserves database consistency.

Most DBMSs ensure serializability through database locking. By locking, we mean a synchronization method in which transactions dynamically reserve data before accessing them [11].

SDD-1 uses two synchronization mechanisms that are distinctly different from locking [5]. The first mechanism, called *conflict graph analysis*, is a technique for analyzing "classes" of transactions to detect those transactions that require little or no synchronization. The second mechanism consists of a set of *synchronization protocols* based on "timestamps," which synchronize those transactions that need it.

Define transactions T1 and T2 as in Figure 6
read-set (T1) = {C.SavBal, C.ChkBal where C.Name = "Adams"}
write-set(T1) = {}
read-set (T2) = read-set(T1)
write-set(T2) = read-set(T1)



Note: nodes r_i and w_i denote the read-set and write-set of transaction T_i

Define transaction T3 and T4 as in Figure 7
read-set (T3) = {C.ChkBal, where C.Name = "Monroe"}
write-set(T3) = read-set(T3)
read-set (T4) = read-set(T3)
write-set(T4) = read-set(T3)

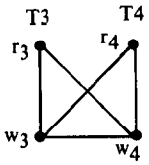


Fig. 8. Conflict graphs.

3.2 Conflict Graph Analysis

The *read-set* of a transaction is the portion of the database it reads and its *write-set* is the portion of the database it updates. Two transactions *conflict* if the read-set or write-set of one intersects the write-set of the other. In a system that uses locking, each transaction locks data before accessing them, so conflicting transactions never run concurrently. However, not all conflicts violate serializability; that is, some conflicting transactions can safely be run concurrently. More concurrency can be attained by checking whether or not a given conflict is troublesome, and only synchronizing those that are. Conflict graph analysis is a technique for doing this.

The nodes of a conflict graph represent the read-sets and write-sets of transactions, and edges represent conflicts among these sets. (There is also an edge between the read-set and write-set of each transaction.) Figure 8 shows sample conflict graphs. The important property is that different kinds of edges require different levels of synchronization, and that synchronization as strong as locking is required only for edges that participate in cycles [2]. In Figure 8, for example, transactions T1 and T2 do not require synchronization as strong as locking, whereas T3 and T4 do.

It is impractical to use conflict graph analysis at run-time because too much intersite communication would be required to exchange information about conflicts. Instead, during database design we apply the technique off-line as follows: The database administrator defines *transaction classes*, which are named groups of commonly executed transactions. Each class is defined by its name, a read-set, a write-set, and the TM at which it runs; a transaction is a member of a class if

the transaction's read-set and write-set are contained in the class's read-set and write-set, respectively. Conflict graph analysis is actually performed on these transaction classes, not on individual transactions as in Figure 8. (Notice that transactions from different classes can conflict only if their classes conflict.) The output of the analysis is a table which tells (a) for each class, which other classes it conflicts with, and (b) for each such conflict, how much synchronization (if any) is required to ensure serializability.

It is convenient to assume that each TM is only permitted to supervise transactions from one class, and vice versa.[2] At run-time, when transaction T is submitted, the system determines which class(es) T is a member of, and sends T to the TM that supervises one of these classes. The TM synchronizes T against other transactions *in its class* using a local mechanism similar to locking. To synchronize T against transactions in other classes, the TM uses the synchronization method(s) specified by the conflict graph analysis. These methods are called "protocols" and are described below.

3.3 Timestamp-Based Protocols

To synchronize two transactions that conflict dangerously, one must be run first, and the other delayed until it can safely proceed. In locking systems, the execution order is determined by the order in which transactions request conflicting locks. In SDD-1, the order is determined by a total ordering of transactions induced by *timestamps*. Each transaction submitted to SDD-1 is assigned a globally unique timestamp by its TM. Timestamps are generated by concatenating a TM identifier to the right of the network clock time, so that timestamps from different TMs always differ in their low-order bits. This means of generating unique timestamps was proposed in [23].

The timestamp of a transaction is attached to all Read and Write commands sent to DMs on the behalf of that transaction. In addition, each Read command contains a list of classes that conflict dangerously with the transaction issuing the Read (this list was determined by the conflict graph analysis). When a DM receives a Read command, it defers the command until it has processed *all earlier* Write commands (i.e., those with smaller timestamps) and *no later* Write commands (i.e., those with larger ones) from the TMs for the specified classes. The DM can determine how long to wait because of a DM–TM communication discipline called *piping*.

Piping requires that each TM send its Write commands to DMs in timestamp order. In addition, the Reliable Network guarantees that messages are received in the order sent. Thus when a DM receives a Write from (say) TM_X timestamped (say) TS_X, the DM knows it has received *all* Write commands from TM_X with timestamps less than TS_X. So, to process a Read command with timestamp TS_R, the DM proceeds as follows:

> For each class specified in the Read command, the DM processes all Write commands from that class's TM up to (but not beyond) TS_R. If, however,

[2] This assumption engenders no loss of generality since several TMs can be multiprogrammed at one site, and several classes can be defined with identical read-sets and write-sets.

the DM has already processed a Write command with timestamp beyond TS_R from one of these TMs, the Read is rejected.

To avoid excessive delays in waiting for Write commands, idle TMs periodically send null (empty) timestamped Write commands; also, an impatient DM can explicitly request a null Write from a TM that is slow in sending them.

The synchronization protocol we have just described roughly corresponds to locking and is designed to avoid "race conditions" [5]. However, there are several variations of this protocol, depending on the type of timestamp attached to Read commands and the interpretation of the timestamps by DMs. For example, read-only transactions can use a less expensive protocol in which the *DM* selects the timestamp, thereby avoiding the possibility of rejection and reducing delay. The variety of available synchronization protocols is an important feature of SDD-1's concurrency control.

The SDD-1 concurrency control mechanism is described in greater detail in [3, 4]; its correctness is formally proved in [2].

When all Read commands have been processed, consistent private copies of the read-set have been set aside at all necessary DMs. At this point, the Read phase is complete.

4. DISTRIBUTED QUERY PROCESSING

Having obtained a consistent copy of a transaction's read-set, the next step is to compile the transaction into a parallel program and execute it. The key part of the compilation is *access planning*, an optimization procedure that minimizes the object program's intersite communication needs while maximizing its parallelism. Access planning is discussed in Section 4.1, and execution of compiled transactions is explained in Section 4.2.

4.1 Access Planning

Perhaps the simplest way to execute a distributed transaction T is to move all of T's read-set to a single DM, and then execute T at that DM (see Figure 9). This approach works but suffers two drawbacks: (1) T's read-set might be very large, and moving it between sites could be exorbitantly expensive; (2) little use is made of parallel processing. Access planning overcomes these drawbacks.

The access planner produces object programs with two phases, called *reduction* and *final processing*. The reduction phase eliminates from T's read-set as much data as is economically feasible without changing T's answer. Then, during final processing, the reduced read-set is moved to a designated "final" DM where T is executed. This structure mirrors the simple approach described above but lowers communication cost and increases parallelism via reduction.

Reduction employs the familiar *restriction* and *projection* operators, plus an operator called *semi-join*, defined as follows: let R(A, B) and S(C, D) be relations; the semi-join of R by S on a qualification q (e.g., R.B = S.C) equals the join of R and S on q, projected back onto the attributes of R (see Figure 10). If R and S are stored at different DMs, this semi-join is computed by projecting S onto the attributes of q (i.e., S.C), and moving the result to R's DM.

We define the *cost* of an operation to be the amount of data (e.g., number of

Given the database of Figures 2 and 3.

Let transaction T5 be
Range of C is CUSTOMER;
Replace C (ChkBal = ChkBal + LoanBal) where LoanBal < 0;
(The effect of T5 is to credit loan overpayments to customers' checking accounts.)

Simple strategy
Move every fragment that could potentially contribute to T5's result to a designated site. Process T5 locally at that site.

Fig. 9. Simple execution strategy.

Given:

CUST (Name,	ChkBal,	LoanBal)	AUTO_PAY (Name,	Amount)
Jeff.	$300	$30000	Jeff.	$300
Adams	$100	$20000	Adams	$200
Polk	$250	$20000	Polk	$200
Tyler	$100	$15000	Tyler	$150
Buchanan	$700	$40000	Buchanan	$400
Johnson	$200	$20000	Johnson	$200

Example (i)

The semi-join of
CUST by AUTO_PAY on CUST.ChkBal = AUTO_PAY.Amount
equals the join of
CUST and AUTO_PAY on CUST.ChkBal = AUTO_PAY.Amount

(Name,	ChkBal,	LoanBal,	Name,	Amount)
Jeff.	$300	$30000	Jeff.	$300
Johnson	$200	$20000	Adams	$200
Johnson	$200	$20000	Polk	$200
Johnson	$200	$20000	Johnson	$200

projected onto

(Name,	ChkBal,	LoanBal)
Jeff.	$300	$30000
Johnson	$200	$20000.

Example (ii)

The semi-join of
CUST by AUTO_PAY on CUST.ChkBal < AUTO_PAY.Amount
and CUST.Name = AUTO_PAY.Name
equals:

(Name,	ChkBal,	LoanBal)
Adams	$100	$20000
Tyler	$100	$15000

(These are customers whose balances are insufficient for their automatic loan payments.)

Fig. 10. Semi-join examples.

bytes) that must be sent between sites to execute it, while its *benefit* is the amount by which it reduces the size of its operand.[3] Restrictions and projections can be computed with no intersite data transfer,[4] and always produce monotonically smaller relations. So, under our cost definition, such operations are always

[3] This definition is appropriate because communications is the bottleneck in a distributed DBMS.
[4] We ignore the cost of sending the restriction or projection command from TM to DM. In practice this cost is negligible.

cost beneficial. Semi-joins, on the other hand, require intersite data movement whenever their operands are stored at different sites. Hence the cost effectiveness of a semi-join depends on the database state. The problem of access planning is to construct a program of cost-beneficial semi-joins, given a transaction and a database state.

The procedure we employ uses a hill-climbing discipline, starting from an initial feasible program and iteratively improving it. The initial program is essentially the simple approach described at the beginning of this section. The access planner improves this program by first adding all restrictions and projections permitted by T, and then iteratively incorporating cost-beneficial semi-joins. This process terminates when no further cost-beneficial semi-joins can be found. A final stage reorders the semi-joins to take maximal advantage of their reductive power, since the order in which semi-joins are selected may be suboptimal for execution.

4.2 Distributed Execution

The programs produced by the access planner are nonlooping parallel programs and can be represented as data flowgraphs [17]. To execute the program, the TM issues commands to the DMs involved in each operation as soon as all predecessors of the operation are ready to produce output.

The effect of execution is to create at the final DM a temporary file to be written into the database (if T is an update) or displayed to the user (if T is a retrieval). At this point, the Execute phase has completed.

5. RELIABLE WRITING

To complete transaction processing, the temporary file at the final DM must be installed in the permanent database and/or displayed to the user. Since the database is distributed, the temporary file is first split into a set of temporary files $F_1, \ldots, F_n$ that list the updates to be performed at each of $DM_1, \ldots, DM_n$, respectively; if any results must be displayed to the user, let us treat the user as one of the DMs.

Each of these temporary files is transmitted to the appropriate DM as a Write command. The problem is to ensure that failures cannot cause some DMs to install updates while causing others not to. We must protect against two types of failures: failure of a receiving DM, and failure of the sender; the former is handled by *reliable delivery* and the latter by *transaction control*, described in Sections 5.1 and 5.2, respectively. In addition, it is necessary to ensure that updates from different transactions are installed in the same "effective" order at all DMs. This problem is addressed in Section 5.3.

Ideally one would like 100 percent protection against failures, but this goal is theoretically unattainable [13]. Instead our goal is to attain acceptably high levels of protection and, moreover, to make the level of protection a database design parameter.

5.1 Guaranteed Delivery

Techniques for reliable message delivery are well known in the communication field as long as both sites are up. For example, in ARPANET errors due to duplicate messages, missing messages, and damaged messages are detected and corrected by the network software. SDD-1's guaranteed delivery mechanism

addresses the additional threat that the sender and receiver are not up simultaneously to exchange messages.

To solve this problem, the RelNet employs a mechanism called *spooler*. A spooler is a process with access to secondary storage that serves as a first-in, first-out message queue for a failed site. Any message sent to a failed DM is delivered to its spooler instead. Each spooler manages its secondary storage using conventional DBMS reliability techniques [24] to guarantee the integrity of these messages. In addition, protection against spooler site failures is attained by employing multiple spoolers; as long as one spooler is up and running correctly, messages can be reliably stored.

With the use of reliable delivery, WRITE messages can be sent to failed DMs. When a failed DM recovers, it can receive its (spooled) WRITE messages to bring its database up to date.

5.2 Transaction Control

Transaction control addresses failures of the final DM that occur during the Write phase. Suppose the final DM fails after sending files $F_1, \ldots, F_{k-1}$, but before sending $F_k, \ldots, F_n$. At this point the database is inconsistent because $DM_1, \ldots, DM_{k-1}$ reflects the effects of the transaction, while $DM_k, \ldots, DM_n$ do not. Transaction control ensures that inconsistencies of this type are rectified in a timely fashion.

The basic technique employed is a variant of "two-phase commit" [13]. During phase 1, the final DM transmits $F_1, \ldots, F_n$, but the receiving DMs do *not* install them yet. During phase 2, the final DM sends *Commit messages* to $DM_1, \ldots, DM_n$, whereupon each DM_i does the installation. If some DM, DM_k say, has received F_k, but not a Commit, and the final DM has failed, DM_k consults the other DMs. If any have received a Commit, DM_k does the installation; if none have received Commits, none do the installation, thereby aborting the transaction.

This technique offers complete protection against failures of the final DM but is susceptible to multisite failures. Enhancements that offer arbitrarily high protection against multiple failures are described in [14].

5.3. The Write Rule

If transactions T1 and T2 complete execution at approximately the same time and have intersecting write-sets, a mechanism is needed to ensure that their updates are installed "in the same order" at all DMs. One way to do this is to attach the transaction's timestamp to each Write command and require that DMs process Write commands in timestamp order. This technique introduces unnecessary delays, however. It is possible to do better by timestamping the database as well as Write commands.

Every *physical* data item in the database is timestamped with the time of the most recent transaction that updated it. In addition, each Write command carries the timestamp of the transaction that generated it. When an update is committed at a DM, the following *Write rule* is applied: For each data item, X, in the Write command, the value of X is modified at the DM if and only if X's stored timestamp is less than the timestamp of the Write command. Thus "recent" updates (ones with big timestamps) are never overwritten by "older" ones. The

net effect is the same as processing Write commands in timestamp order. This technique was originally suggested by [23].

A principal objection to this technique is the apparent high cost of storing timestamps for every data item in the database. However this cost is reduced to acceptable levels by caching the timestamps (see [4]).

When updates are installed at all DMs, the Write phase is completed. At this point the transaction has been fully processed.

6. DIRECTORY MANAGEMENT

SDD-1 maintains directories containing relation and fragment definitions, fragment locations, and usage statistics. Since TMs use directories for every transaction, efficient and flexible directory management is important. The main issues in directory management are whether or not to store directories redundantly and whether directory updates should be centralized or decentralized. We have made these issues a matter of database design by treating directories as *ordinary user data.* This approach allows directories to be fragmented, distributed with arbitrary redundancy, and updated from arbitrary TMs.

But there are some problems. First, performance could be degraded by requiring that every directory access incur general transaction overhead, and by requiring that every access to remotely stored directories incur communication delays. We avoid these performance problems by *caching* recently referenced directory fragments at each TM, discarding them if rendered obsolete by directory updates. Since directories are relatively static, this solution is appropriate.

A second problem is that we now need a directory that tells where each directory fragment is stored. This directory is called the *directory locator,* and a copy of it is stored at every DM. This solution is appropriate because directory locators are relatively small and quite static.

7. HISTORY

SDD-1 is the first general-purpose distributed DBMS ever developed. Its design was initiated in 1976 and completed in 1978. The first version of the system, which included distributed query processing, was released in mid-1978; a complete prototype system, including concurrency control and reliable writing, was released in autumn 1979. SDD-1 is implemented for DEC-10 and DEC-20 computers running the TENEX and TOPS-20 operating systems; its communication medium is the ARPA network. SDD-1 is built on top of existing software to the extent possible; most notably it employs an existing DBMS, called Datacomputer [19], to handle all database management issues. The current system is configured with four sites, although the software can support any reasonable number.

The complete SDD-1 software consists of 25,000 lines of BCPL code. The compiled DM has 47K 36-bit words of code; the compiled TM has 120K 36-bit words of code, of which 45K words are "borrowed" from Datacomputer's object code. The design and implementation represents about 10 man-years of effort.

8. CONCLUSION

SDD-1 is a general-purpose distributed DBMS, integrating database management, distributed processing, and reliable communication technologies into a

cohesive system. This integration offers substantial benefits by combining the advantages of distributed processing with the advantages of centralized database management. At the same time it introduces new technical problems, of which the most critical are concurrency control, query processing, and reliable writing. This paper has outlined the SDD-1 solutions to each of these problems. The existence of SDD-1 as a system demonstrates that these problems can be solved in an integrated software system, and that distributed database management is indeed a feasible technology. For in-depth presentations of our techniques we refer the reader to [2–4, 12, 14].

ACKNOWLEDGMENTS

The authors thank Carolyn Carleton, Emy Dickey, and Nancy Wolfe for their editorial assistance in preparing this and the companion SDD-1 papers.

REFERENCES

1. ASTRAHAN, M.M., ET AL. System R: Relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.
2. BERNSTEIN, P.A., AND SHIPMAN, D.W. The correctness of concurrency control mechanisms in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (March 1980), 52–68.
3. BERNSTEIN, P.A., ROTHNIE, J.B., GOODMAN, N., AND PAPADIMITRIOU, C.A. The concurrency control mechanism of SDD-1: A system for distributed databases (the fully redundant case). *IEEE Trans. Software Eng. SE-4*, 3 (May 1978), 154–168.
4. BERNSTEIN, P.A., SHIPMAN, D.W., AND ROTHNIE, J.B. JR. Concurrency control in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (March 1980), 18–51.
5. BERNSTEIN, P.A., SHIPMAN, D.W., AND WONG, W.S. Formal aspects of serializability in database concurrency control. *IEEE Trans. Software Eng. SE-5*, 3 (May 1979), 203–215.
6. BOBROW, D.G., BURCHFIEL, J.D., MURPHY, D.L., AND TOMLINSON, R.S. TENEX, a paged time sharing system for the PDP-10. *Comm. ACM 15*, 3 (March 1972), 135–143.
7. CHAMBERLIN, D.D., ET AL. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM J. Res. and Develop. 20*, 6 (Nov. 1976), 560–575.
8. CODD, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.
9. COMPUTER CORPORATION OF AMERICA. *Datacomputer Version 5 User Manual*, Cambridge, Mass., July 1978.
10. DAYAL, U., AND BERNSTEIN, P.A. The fragmentation problem: Lossless decomposition of relations into files. Tech. Rep. CCA-78-13, Computer Corporation of America, Cambridge, Mass., Nov. 1978.
11. ESWARAN, K.P., GRAY, J.N., LORIE, R.A., AND TRAIGER, I.L. The notions of consistency and predicate locks in a database system. *Comm. ACM 19*, 11 (Nov. 1976), 624–633.
12. GOODMAN, N., BERNSTEIN, P.A., REEVE, C. L., ROTHNIE, J.B., AND WONG, E. Query processing in SDD-1: A system for distributed databases. Submitted for publication.
13. GRAY, J.N. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, Vol. 60 of *Lecture Notes in Computer Science*, Springer-Verlag, 1978, pp. 393–481.
14. HAMMER, M.M., AND SHIPMAN, D.W. The reliability mechanisms of SDD-1: A system for distributed databases. Submitted for publication.
15. HELD, G., STONEBRAKER, M., AND WONG, E. INGRES: A relational data base system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Arlington, Va., pp. 409–416.
16. HORNING, J.J., AND RANDELL, B. Process structuring. *ACM Computing Surveys 5*, 1 (March 1973), 5–30.
17. KARP, R.M., AND MILLER, R.E. Properties of a model for parallel computation: Determinacy, termination, queueing. *SIAM J. Appl. Math. 14* (Nov. 1966), 1390–1411.
18. LORIE, R.A. Physical integrity in a large segmented database. *ACM Trans. Database Syst. 2*, 1 (March 1977), 91–104.

19. MARILL, T., AND STERN, D.H. The datacomputer: A network data utility. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Arlington, Va., pp. 389-395.
20. RITCHIE, D.M., AND THOMPSON, K. The UNIX time-sharing system. *Comm. ACM 17*, 7 (July 1974), 365-375.
21. ROTHNIE, J.B., AND GOODMAN, N. An overview of the preliminary design of SDD-1: A system for distributed databases. Proc. 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Lab., U. of California, Berkeley, Calif., May 1977, 39-57.
22. SEVERANCE, D.G., AND LOHMAN, G.M. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 256-267.
23. THOMAS, R.H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (June 1979), 180-209.
24. VERHOFSTAD, J.S.M. Recovery techniques for database systems. *ACM Computing Surveys 10*, 2 (June 1978), 167-196.
25. WONG, E. Retrieving dispersed data from SDD-1: A system for distributed databases. Proc. 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Lab., U. of California, Berkeley, Calif., May 1977, 217-235.

Received December 1978; revised August 1979