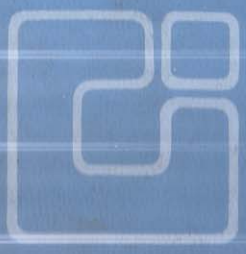
distributed data bases

edited by

h.-j. schneider



north-holland

Second International Symposium on
Distributed Data Bases
September 1-3, 1982

Organized by
Gesellschaft für Informatik (GI)

in cooperation with
AFCET, AICA, IBM, IEEE, IFIP-TC2,
IGDD, INRIA, Nixdorf, Siemens,
TU Berlin, Universität Stuttgart

Symposium Chairman
H.-J. Schneider

Program Commitee
E. J. Neuhold, Chairman (FRG),
Ph. Bernstein (USA), H. Biller (FRG), G. Bracchi (I),
J. Bubenko (S), S. M. Deen (GB), G. Gardarin (F),
G. Le Lann (F), W. Litwin (F), R. Munz (FRG),
F. A. Schreiber (I), P. Selinger (USA),
R. P. van de Riet (NL), H. Weber (FRG)

Tutorial Session Chairman
G. Gardarin (F)

Organizer
S. Florek (FRG)

NORTH-HOLLAND PUBLISHING COMPANY
AMSTERDAM • NEW YORK • OXFORD

DISTRIBUTED DATA BASES

Proceedings of the Second International Symposium on
Distributed Data Bases
Berlin, F.R.G., September 1-3, 1982

edited by

H.-J. SCHNEIDER
Technische Universität Berlin
F.R.G.

1982

NORTH-HOLLAND PUBLISHING COMPANY
AMSTERDAM • NEW YORK • OXFORD

DISTRIBUTED DATA BASES
H.-J. Schneider (editor)
North-Holland Publishing Company

SIRIUS Systems for Distributed Data Management

W. Litwin, J. Boudenant, C. Esculier, A. Ferrier, A. M. Glorieux, J. La Chimia, K. Kabbaj, C. Moulinoux, P. Rolin, C. Stangret

I.N.R.I.A 78153 - Le Chesnay, France

Distributed data management becomes an increasily important field of interest. SIRIUS project investigated several approaches to this goal. We present on the one hand the approach developped in the context of SIRIUS-DELTA system. The characteristic property of this approach is that users manipulate a single database. We also present an approach that we called multidatabase approach. The characteristic property of the corresponding systems is that users manipulate a collection of databases.

1. INTRODUCTION.

SIRIUS project was set up in 1977. The goal of the project was the design of systems allowing to manage distributed databases. SIRIUS was a, so-called, pilot project. This meant that the resources and the objectives of the project were larger than the ones of a typical research project. In particular, it was required from the project :
(1) - to set up research on distributed data management in french universities,
(2) - to spread out the knowledge of the domain within the computer industry and potential users,
(3) - To design prototype(s) of DDBMS(s) that would be sufficiently operational to be qualified of preindustrial.

In order to respond to the objective (1), several research studies have been set up through contracts. These studies gave rise to many theoretical results. The results have been published in virtually all important conferences on databases and distributed systems. The main results obtained prior to 1980 are presented and indexed in [LEB80]. Others are presented below and/or indexed in the references of this paper.

University studies gave also rise to prototypes investigating the design of a DDBMS. The prototypes POLYPHEME (CII - IMAG), FRERES (IRISA), SYSIDOR (USST) and ETOILE (NANCY-1) were in, particular, presented during the International Symposium on Distributed Databases (Paris, March 1980, proc. North Holland). The prototypes MICROBE (IMAG) and TYP-R (NANCY-1) were presented during Journees SIRIUS (Paris, November 1981, proc. Agence de l'Informatique).

The response to the objective (2) consisted in general in a close relationship between the project, industry and users. Seminars and project presentations were frequently organized. Also, industry people were systematically involved in the teams that worked on the main prototypes. Finally, industry or user project were supported to some extend, in particular the projects SCOT [BOG81], SOPHIA [LED81] or PLEXUS [ZUR82]. Most of french computer manufacturers or software houses were in this way involved in SIRIUS project.

Starting from 1979, the theoretical and practical work has been more and more performed in INRIA, by a growing project team. The main, although not the unique, objective assigned to this team was to respond to the objective (3). The main preindustrial prototype is the SIRIUS-DELTA system, intended to be a general purpose DDBMS for enterprise DDBs. Another preindustrial prototype called MESSIDOR [MOU81] allows to manipulate sets of heterogeneous bibliographic DBs, distributed over public networks (EURONET, TRANSPAC,...). Some other, yet less complete, prototypes were also developed : PHLOX [BDV80], MUQUAPOL [KAB81] and MRDSM [GUE81].

These prototypes were based on two different perceptions of distributed data management problematic. The first approach assumes that distributed data constitutes a single database called distributed database (DDB). A DDB is defined by a single conceptual schema, called global schema. Users work with the DDB as they would work with a centralized database defined by the same schema. They are not aware of the existence of several sites. Also, they are not aware of the existence of several DBs if data on a site constitute a DB.

The second approach assumes that distributed data may also constitute a collection of databases called multidatabase. Users of a multidatabase typically know that they face several databases, i. e. that data that they use are logically distributed. The system provides a language for queries to more than one database. It may also allow to define inter-database dependencies, such as integrity dependencies for instance. Users are not aware of the number of sites underlying a multidatabase. The entire multidatabase may, in particular, be on only one site.

SIRIUS-DELTA prototype investigated the first approach. Particular attention was given to DDB data distribution and to distributed processing. Below, we first sum up the objectives that guided the design of this prototype. Then, we describe SIRIUS-DELTA DDB architecture and system architecture. Afterwards, we detail the functional layers. Finally we indicate some results of performance analysis.

The multidatabase approach was investigated within a sub-project called B A BA. In what follows, we will also present this approach. First, we will explain the idea in the multidatabase with respect to the one in the present database approach. Next, we will define formally the concept of a multidatabase and we will investigate some case studies. Then, we will present a general architecture of a multidatabase management system. Afterwards, we will show main features of a relational multidatabase manipulation language. Finally, we will briefly present MRDSM, MUQUAPOL and MESSIDOR prototypes.

The presentation of SIRIUS-DELTA prototype constitutes the section 2. Section 3 discusses the multidatabase approach. Both sections end up with conclusions. The overall conclusions constitute section 4.

2. SIRIUS-DELTA SYSTEM APPROACH

2.1. SIRIUS-DELTA MAIN OBJECTIVES & FEATURES

In this section we describe SIRIUS-DELTA main objectives and features, and we give an introduction to the means used to reach them.

2.1.1. The user is not aware of data distribution :

The DDBMS is responsible for the identification of all data objects involved in the user's request, their localization and selection of the proper copy(ies), if any. In addition, the DDBMS must be able to handle requests from pre-existing query languages on the submitting sites.

This is achieved using first schemas associated to the DDB and related local DB's, next a common internal data manipulation language : the "pivot-language".

2.1.2. Various types of data distribution must be available :

In order to fit with the applications needs, various types of data distribution and duplication must be provided and processed by the DDBMS [SDD78] [NEU77]. This is necessary if one wants to provide some DDB design and processing facilities (local and parallel processing optimization, usage of pre-existing local data for example).

A data distribution description language has been developped to describe the necessary mappings between DDB data objects and local data objects and the localization rules for these local data objects (data objects can be distributed up to an attribute value).

2.1.3. Distributed data must remain consistent in case of conflicting accesses and failures :

When concurrent update requests are submitted on the same or on different sites, the DDBMS must be able to keep consistent the related distributed data.

A transaction is defined in SIRIUS-DELTA as a sequence of one or several inquiries/updates enclosed by a BEG-TRANS and an END-TRANS. Distributed data consistency is achieved thanks to a distributed concurrency system that provides a unique naming of the transaction, maintains transaction atomicity and controls local accesses to data objects using locks.

In case of failures, the transaction is completed or rolled-back according to its current status of completion. Strong consistency is achieved, i.e. all copies are updated or none.

2.1.4. System reliability must be achieved :

In a distributed system the number of components (computers, links...) increases, thus increasing the global fault tolerance.

In SIRIUS-DELTA, system reliability is achieved through a dynamic reconfiguration procedure and local log files that hold distributed checkpoints. This permits hot and cold restart procedures.
In addition, failure protection is achieved during transaction commitment.

2.1.5. Heterogeneity :

Heterogeneity requirements results either from a wish to offer to users flexibility on its configuration components and extensions, or from a wish to make use, as much as possible, of pre-existing components [BOS78], [POL79] :
- data processing units that can be interconnected
- local DBMSs or data managers (DM)
- local DBs or data files.

Heterogeneity is allowed in SIRIUS-DELTA at hardware and software level. At software level, heterogeneity at DBMS/DM level is taken into account using the pivot-langage. Heterogeneity at local DB or data files level is taken into account using the local external views associated to the DDB (see 2.2.1.).

2.2. SIRIUS-DELTA ARCHITECTURE OVERVIEW

2.2.1. DDB architecture :

We are concerned with data description problem-solving when data are distributed, therefore we strongly rely upon schema concept.

The DDB architecture reflects the global/local duality of distributed systems [ADI78], [AFC78], [SPA78] :
- at global level, distributed data and systems are viewed as one logical unit,
- at local level, local components interact with the local DBMS or DM.

In addition local systems participate in the distributed system as local sites, as well as they maintain some local independancy (local processing must be allowed).

This result in a proposition to extend the ANSI/SPARC schema architecture [ANS75], [TSI77] in order to introduce a global level and a local level. At global level the DDB is considered as a database the physical characteristic, of which is data distribution. Thus, the DDB is described using the three-level architecture.

At local level, we want to maintain some independancy from the distributed system. Thus local data are described using the three level architecture.

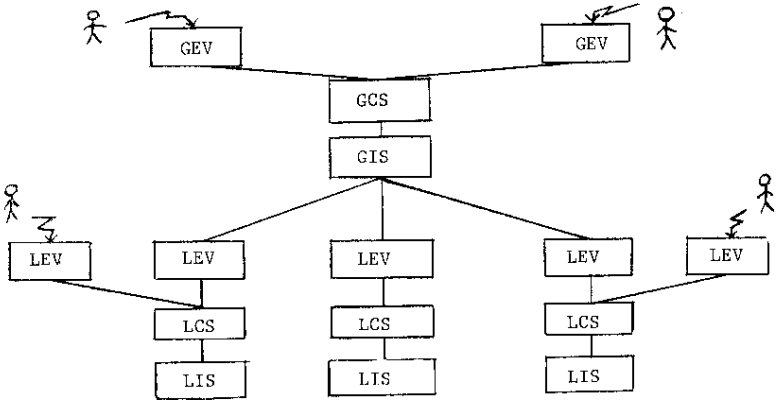


Figure 1. : Schemas

Let's now detail schemata that is hold in each one of these schemas, using the ANSI/SPARC concepts :

a) the user is not aware of data distribution. Thus global external views are defined for DDB users in the same way as in centralized DBMSs.

b) the DDB is considered as ONE logical database. Thus DDB conceptual schema does not include description of the data distribution. As a result, the global conceptual schema is described as a centralized conceptual schema would be.

c) DDB data objects are distributed across sites. This is considered to be the physical characteristic of DDB data. The global internal schema contains the description of the mappings between global and local data, as well as the necessary localization and duplication rules.

d) local data are expected to contribute to the DDB but :
   - not all local data does,
   - from the DDB point of view local, conceptual data may look heterogeneous.

   Thus a local external view is provided to the DDBMS so that heterogeneously maintained local data are homogeneously presented and handled at DDB level.
   The local external view specific to the DDBMS must also be consistent with the local conceptual schema, from an update point of view.

e) local conceptual schema and local internal schema are normal "centralized schemas".

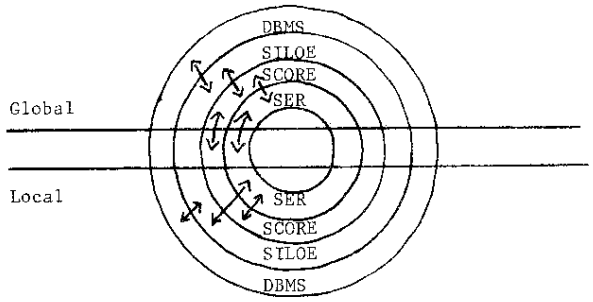
   It may happen that locally configured sites do not host a DBMS and are only provided with a Data Manager. In such a case, the local system must be extended to host the necessary data handling functions expected by the DDBMS :
   - since no schemas are available, the related local external view functionnality must be implemented, in order to provide the DDBMS an homogeneous presentation of local data,
   - since no data manipulation functions are available, they must be developped using, for example, generalized catalogued procedures.

2.2.2. System architecture :

In SIRIUS-DELTA DDBMS we rely on an underlying transport layer. It provides link control between two processes (message and flow control sequencing, message error-free delivery, signal message of site inaccessibility), and an adaptative routing to network topological changes.

Four basic functionnal layers are defined above the transport layer,that provides :
. database classical data management functions ("DBMS" layer)
. distributed data handling functions (SILOE layer)
. distributed concurrency control functions (SCORE layer)
. distributed execution functions (SER layer)

This architecture is depicted in fig.2.



inter-layer arrow : service
intra-layer arrow : protocol

Figure 2.

All these layers reflect the global/local duality of distributed systems [ADI78], [AFC78], [SPA78]. Therefore all of them are divided into two components :
- a global level component, that provides the unified view of a distributed system,
- a local level component, that interacts with the global levels and the local operating system.

A kernel of a distributed executive system responsible for transaction processing is provided by layers SER and SCORE. Their global component is refered as the producer machine, and local component as the consumer machine.

SER is always needed, while SCORE is required if distributed updates are concurrently processed.

On the other hand, SILOE is a DBMS extension so that DDB query processing [CAL78], [NGU79] is allowed : at global level it decomposes the query into a sequence of localized subqueries. At local level, it is the necessary extension of existing software to allow subquery processing in a distributed environment.

Cooperation between the three layers - SILOE, SCORE and SER - is codified via the Distributed Execution Plan (DEP). A DEP is associated to each query and holds its transaction-id, its data allocation requests, and its list of subqueries with a specification of their synchronization.

All SIRIUS-DELTA layers must not necessarily be implemented upon all sites. Different configurations may exist in order to fit with specialized data processing needs and to distribute functionalities (global or local site only, consumer site, etc...).

The sections that follows are devoted to a detailed description of layers SER, SCORE and SILOE.

2.3. SER LAYER

The distributed execution function is as a basic one for distributed data base management systems. Such mechanisms are necessary to provide distributed programs with activation on various computers, synchronization, control (mainly in case of failures) and data exchange.

Many studies about this function are being lead in France, and abroad [ROB77], [FOR77]. In this section, we present the SIRIUS-DELTA distributed execution system (named SER) [SER80], a synthesis of the works conducted within SIRIUS on the subject [DAN77], [AND80].

2.3.1. Objectives :

At present time, most of the distributed applications can be seen as a set of locally predefined actions which are processed in parallel, or are conditionally sequenced. Every application within this class can roughly be schematized by a graph of requests asking for remote catalogued programs execution.

SER is designed to answer the needs of every application in this class. In order to be used on a set of processors, it needs :

- a communication medium between the various machines involved in the application. It must provide a classical transport service (such as layer 4 of ISO Open System Architecture) [ISO82]. We define hereafter a "processor " as the physical entity upon which an occurrence of SER can be run. In most cases a site can be considered identical to a processor.

- on every physical processor, a catalogue of independant logical processors which can be used by distributed applications. Those processors (say compilers) are programs, subroutines, software modules, or procedures. They correspond to the use of an existing service or are especially designed for the distributed application. We shall name them "programs".

Using the various catalogues of programs located on interconnected processors, SER supplies applications with the following services :
- activation of remote programs,
- scheduling of local programs that are inter-related according to conditions specified by the distributed application in a distributed execution plan (DEP),
- data transfer between remote programs within a distributed execution plan,
- logical expression and computing of program activating conditions. Those conditions can be related to the results of others programs,
- as far as possible, use of the parallelism provided by the set of interconnected processors,
- detection and signal of failure, and normal or abnormal end of program execution.

2.3.2. Basic concepts :

The DEP is a set of local actions ; each one of them is executed upon one processor.
In SIRIUS-DELTA context, the DEP is automatically generated by global SILOE. However, SER may also be used as a stand-alone Distributed Execution System. In such a case, the DEP must naturally be specified by the application user. The DEP is the main input to SER ; it must point out the various local actions to be performed, their execution processor, and their relationships. Those relationships are data transfers and local execution synchronizations. SER operation is characterized by a fully distributed control relying on three basic entities :
- local action,
- synchronization variable,
- temporary data file.

2.3.2.1. Local action

The local action is the logical execution unit of the DEP. From a physical point of view, it is the execution of a program on a processor.
It can be, for example, a request upon a local database or the execution of a local sort-merge program.
We define a local action within a DEP as an existing program running upon a dedicated processor, as well as its input and output data flows and synchronization conditions within the DEP.

A local action the execution of which is in progress does not perform any synchronization with another local action running in parallel. The only interactions a local action may have on other local actions of a distributed execution program are updating of synchronization variables and the consumption or production of temporary data files.

2.3.2.2. Synchronization variables

These variables are associated with local actions ; they ensure the distributed execution control as well as they define internal synchronization mechanisms. For each local action, an activation condition (defining a determined pattern of associated synchronization variables) permits to define its launching. In the same way, at the launching or at the end of execution, an end expression allows the update of remote synchronization variables, i.e., associated to other local actions. For instance, a synchronization variable will be used so that the launching of a local action is conditioned by the launching or the end of another one. The recursive mechanism of activation condition - end expression enables to program the control of the distributed execution plan.

2.3.2.3. Temporary data file

A local action corresponds to the execution of a program which knows nothing about the distributed environment of its execution. Therefore, this program cannot perform data transfer through the network.

SER ensures those transfers. SER automatically manages a network storage area which is used to store local actions input and output data in the form of temporary data files. Temporary data files appear as sequential files with variable size items.

They are produced without any possibility of roll-back and are automatically destroyed after consumption. SER achieves spooled transfers of temporary data files from producer local actions to consumer local actions. SER permits temporary data files produced by different local actions to be merged and presented with the consumer local action as one temporary data file. It also permits to broadcast a temporary data file towards several consumer local actions.

2.3.3. Principle of operating :

2.3.3.1. Distributed execution control

SER is a distributed system in itself, i.e. on each processor in the network, a portion of SER ensures execution and control of the actions to be run on this processor.

A distributed execution plan, when submitted on a given processor to SER, consists of a set of commands requesting the execution of local actions on several processors. Each command defines the context of one local action : its activation condition, its end expression as well as its temporary data files to be produced or consumed. For each command, the submitter localizes and identifies in a unique manner the local action, the synchronization variables and the files.

2.3.3.2. Global and local SER

A portion of SER, behaving like a customer, named global SER, resides on the processor where the distributed execution plan is submitted. It sends each local action context to the portion of SER, that behaves like a server and is named local SER, where the action is to be executed. When all contexts are set-up, global SER takes no longer part to the execution. It is only concerned with end or abort report and with failures or withdrawals of processor (using services offered by transport station level). Thus, the execution control is entirely decentralized at local SER level. The schedule of local actions within a plan is determined by dynamic computing of activation conditions, the updates of which are performed without any reference to initial global SER.

Figure 3 proposes a presentation of those principles for distributed execution.

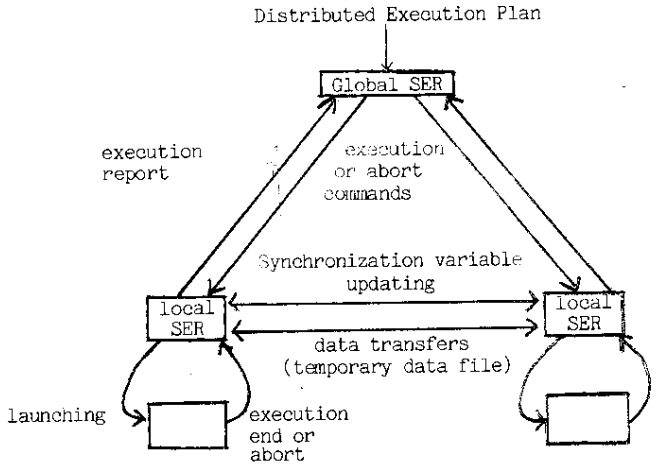


Figure 3. : SER distributed execution structure

2.3.4. Detailed operating of SER :

2.3.4.1. Decomposition of a set-up command

Every local action within a DEP is featured by a local action context which is the logical unit SER deals with. The information held in a context is :

- identification : each context has an identification, that is unique in the whole distributed system. It includes :
  . a unique DEP name within the system
  . a unique local action name within the DEP,

- execution processor : it allows the dispatching of the context towards the processing local SER,

- program to be executed,

- local synchronization variables : the context must include the declaration of the synchronization variables to be used in the activation condition and end expression. They may be boolean, integer or character string. The operations on variables are : allotment or comparison (true or false for booleans ; $=, >, \geq, <, \leq, \neq$ for integers ; $=, \neq$ for strings)

- parameters : this field contains a free character string given as input parameter to the program,

- temporary data file : the context includes declaration and identification of input and output files used by the local action,

- local resources : each context must include the declaration of resources that are needed for the associated execution. Among these resources declarations, we find the lock requests. SER transmits them to SCORE which is responsible for data allocation (see section 2.4),

- activation condition and end expression : the end expression can be evaluated at the launching or at the end of the local action according to the value of a parameter.

2.3.4.2. Dynamic study

Let us consider an execution plan submitted by an application to a global SER. This part of the system analyzes the DEP one command after the other. It transmits to the various local SER the context commands. Those commands must be acknowledged by the receivers so that global SER can give a report on the positionning of the DEP to the application. Then the execution of the DEP can be started in a decentralized manner.

2.3.4.2.1. Synchronization

When receiving a set-up command, local SER stores the context, and evaluates the associated activation condition on each update reception. When the condition becomes true and when the context contains lock requests as soon as data are allocated by SCORE, SER activates the program to which the local action is associated.

When a local action reaches a normal or abnormal end, or when it is launched, SER evaluates the end expression. It sends to the remote SER the updated values of synchronization variables which are declared in other contexts. This mechanism ensures an automatic schedule of the DEP.

2.3.4.2.2. Data exchanges between local actions

These files contain transient data which are only valid for portions of the DEP processing. Data are stored and shared by SER as logical transfer units, called items. The item is a flexible-size packet of data which is produced or consumed in one operation by an action. SER shares a network storage area. Through it, it stores and accesses the items of the various temporary data files. The transfer of data from a producing action to a consuming action is automatically achieved by SER.

2.3.4.2.3. Errors processing

Global SER shares arrays of status variables. There is one array per submitted DEP and one item in the array per local action. Each item is updated on the following supervisory-command receipt from local SER :
. context set-up,
. end of local action,
. abortion of local action (with the reason).

The abortion of an action leads to the abortion of the whole DEP. When aborting a DEP, an abort request is transmitted to every local SER by global SER which destroy all concerned local actions and their environment (network storage area, temporary data files...).

When a site becomes inaccessible during the processing of a local action, the local SER running on this site abort the processing local action. Global SER, informed of this failure by the transport service, aborts the whole plan.

At the end of the plan (normal end or abortion), global SER produces a report of execution to the application.

2.4. SCORE LAYER

SCORE is the name given to the level (and to the software) corresponding to distributed control system (Système de COntrôle REparti) which provides also for a COnsistent and REsilient handling of the SIRIUS-DELTA database.

The consistency of a data base may be threatened for many reasons, in particular :
- interferences between simultaneous accesses to common data items,
- failures of systems components.

SCORE's purpose is to provide mechanisms needed to maintain SIRIUS-DELTA in a consistent state. What we mean is that not only the database should be kept consistent but also the state information used by the distributed executive in order to process user's activities correctly.

To this end, it is necessary to identify all the actions which the executive must associate with any given user transaction so that every "transformated" transaction is actually implemented as an atomic operation.

2.4.1. General principles :

The following model permits to describe SCORE functions. The database is viewed as consisting of a number of logical objects to which access can be gained individually.

Global SILOE produces access requests to objects (i.e. access requests to plots) intended to be processed by local SILOEs. Global SILOE transmits to global SCORE these requests which are inspected by concerned local SCOREs before being entrusted to local SILOEs.

Environment characteristics are assumed to be as follows :
- access requests to objects may be generated at any time during the execution of a transaction (dynamic claiming),
- the ordering of messages exchanged among any two processes is random,
- the communication tool as well as physical components are not fully reliable.

Under these assumptions, we must design decentralized mechanisms which would meet the following constraints :
- automatic detection and automatic recovery from failures,
- no reliance on a single entity (e.g. commit coordinator, commit record), in order to complete a transaction,
- no starvation, either for "reading" transactions or for "writing" transactions,
- fairness among users,
- highly parallel processing,
- no domino effect, except in case of catastrophe,
- construction of consistent recovery lines in such a manner that there would be no interference between transactions and global checkpoints (recovery lines) executing concurrently.

In order to reach this goal, SCORE includes the following three mechanisms :
- a decentralized synchronization mechanism, called Circulating Sequencer [LEL77]. This mechanism is used to give a unique system name to a transaction,
- two-phase locking protocols associated with a deadlock prevention strategy [BOU81]. These protocols are used for data allocation management and assure the synchronization between concurrent and conflicting transactions. A read request must claim a shared or exclusive access, and a write request an exclusive one. The locks are associated to each subtransactions, and set according to the compatibility matrix.

Requested lock \ Object locked in	no lock	read	write
no lock	yes	yes	yes
read	yes	yes	no
write	yes	no	no

The deadlock prevention is upon conflict detection which are solved in accordance with the transaction tickets [ROS78].

- a two-phase commit protocol [GRAY78], [GAR81].

2.4.2. Failures of system components :

This has been extensively described in [BOU81a].

It is not easy for a processor to distinguish between another processor crash and a link failure with this processor, so these two events are treated in the same way.

The used mechanisms guarantee the atomicity of a transaction. For each transaction all or none of the write actions will be performed [LEL80].

In SIRIUS-DELTA a transaction is under the control of one SCORE-producer only (a command executed by a consumer cannot generate other commands).

2.4.2.1. Consumer crash detected by a producer

A producer must only treat the event for transactions which have claimed objects on this consumer.

a) If the transaction is not in commit phase the transaction will be cancelled. The producer broadcasts an "Abort" message to all the consumers involved in the transaction.

b) If the transaction is in commit phase and if the producer has received a PTC (Prepare To Commit) acknowledgment from the crashed site, the commit protocol goes on without any modification.

c) If the transaction is in commit phase and no PTC acknowledgment was received from the crashed site, we don't know if the "PTC" message was received or not by the consumer ; the producer interrupts the commit protocol, an "Abort" message is broadcasted to the consumers involved in the transaction.

2.4.2.2. Producer crash detected by a consumer

A consumer must deal with this event only if local objects are requested by transactions issued from this producer.

a) The transaction is not in commit phase : no "PTC" message received for this transaction. The consumer decides to cancel the transaction.

b) The transaction is in commit phase and the "Commit" message has been received, then the commit protocol goes on without any modification.

c) The transaction is in commit phase and only the "PTC" message has been received, the inquiry protocol is started. The consumer must ask the other consumers involved in the transaction in order to know what decision to make. Some consumers may have committed or cancelled the transaction and SCORE must guarantee that the same thing will be done for all of the involved consumers. The SCORE-consumer broadcasts to all the consumers whose names were given in the "PTC" message a query to know what decision they have made :
   - if at least one has committed, the transaction is committed
   - if at least one has aborted, the transaction is cancelled
   - if all the involved consumers have acknowledged the "PTC" message and have detected a producer crash, the transaction is committed.

2.4.3. Recoveries :

We have chosen to write immediately the updates on the database after having saved the object old value on a "before image file". This choice minimizes the manufacturer software modifications. There is no modification for a read, the writes are trapped to get the "before image" values. During transaction execution, access to modified data doesn't imply any overhead. Of course the "before image file" must be in a stable memory. However, this choice alters the database consistency as soon as a write is performed by a transaction and then the processor recovery mechanism induces the database recovery before inserting a consumer into the distributed system.

For database recovery we can use three elements :
- the database itself,
- the journals,
- the before image files.

Then we find different possibilities for the DB recovery, depending on the availability of these three elements.

2.4.3.1. Only one site involved in the recovery

We can say that there is only one site involved in the recovery if at the end of the local recovery the database, journals and before image file are in a correct state as after a correct running. The processor has just been isolated from all the others processor.

If the site is only a producer there is no problem, the database consistency is not concerned and producers may behave as memory-less processes. We have to repare the damage and locally restart the system ; then the site insertion into the distributed system is automatically started.

If the site is a consumer, the recovery is more complex : the database consistency may be destroyed. In the next sections, to simplify, we assume that we start a new journal when we save local DBs and that all transactions are committed on the DBs.

2.4.3.1.1. The database, journal and "before image" file are available

This is the easier and more rapid local database recovery.

a) Recover all the data base using the "before image" files and release the objects,

b) For each committing transaction we find in the journal with the PTC item the SCORE-consumer context, the list of locks, the modified data, the list of involved sites. Then for these transactions we can recreate their contexts, the "before image" values, put the exclusive locks on all the updated objects (the other locks previously granted to the transaction are no longer needed) and put the updates in the database.

The transaction contexts are now recreated, and the site can be inserted in the DDBMS. The communication level sends to SCORE-consumer the event "link failure with all the other sites", then the inquiry mechanism is initiated for all the transactions in commit phase. There is no loss of committed transaction.

2.4.3.1.2. The database or "before image" file are not available

a) restart from an older DB save,
b) execute sequentially the committed writes from the journal,
c) execute step b of the previous section.

This is still a local recovery without already-committed transaction loss.

2.4.3.2. Several sites may be involved in the recovery

Recovery after a crash with journal partial loss.

We assume, the worst case, that there are no copies of the local data base and journal on other processors. Our recovery protocol leads to a "backward" recovery. The local site has not enough information to come back in the system without danger for the DDB consistency. It is a castastrophic case where the whole DDB has to be rolled-back to a previous consistent state. The distributed database administrator must be informed of this catastrophic error. Other sites may be involved in the recovery and it is the administrator responsability to decide the action to be taken. To make the distributed database recovery easier we must have journal global recovery lines.

2.4.3.3. Global recovery line : transaction termination identifier (TTI)

A way to realize a global recovery line is to initiate a system transaction which locks all the system resources. In some case this is too costly. A modified version of the two-step commit protocol locks is more convenient to write global checkpoints on journals.

When a two step commit is executed, a transaction termination identifier (TTI) is computed. Each consumer maintains a private counter Ci. Before sending the message "PTC", a variable TTI is zeroed at the producer site for the transaction. The consumer writes in the journal "PTC" item its current counter value, the current counter is sent in the PTC acknowledgment to the producer. Then the consumer increments its counter. The producer computes TTI as the maximum of the Ci received and sends it with the commit message. The consumer writes this value on the commit item and executes Ci=max(TTI+1,Ci). The Ci computation upon receiving the commit message ensures that a transaction not in commit phase at a consumer will have a greater TTI than all transactions already committed by this consumer. Then if a transaction Tj has dependency relation with Ti (Tj reads a value written by Ti), the read action can be executed only after the commit of Ti had been executed, we have TTI(Tj)>TTI(Ti). The transactions which are already in commit phase may have a lower or equal TTI than the last committed one. This means that they are serializable, there is no dependency relation and no conflict access between these transactions.

- TTI meaning in the global system -

The TTIs assure in some measure a synchronization between the consumer local counters ; but if there is a system partition, the TTIs of one part of the system are without relation with the TTIs of the other part. We can also have a "logical" partitioning in the distributed database : some producers only work on a part of the DDB and others work on another part of the DDB ; if the used parts of the DDB have no common consumer, there is no relation between their TTI. Let us assume than all sites crash at the

same time and all transactions are committed. If we start up all the databases with their last save and if, on each site, we execute their journal committed writes (found in the "commit" items) in their TTI order with TTI $< X$ we are sure that the global database is in a consistent state.

- If X is the greater consumer counter value in the system : no transaction is lost.

- If X is not the maximum of the TTI we have transactions which are in journals and which are discarded. This is the case when we have a journal partial loss on a site with the last available commit item TTI equal X and then all other sites must roll back to this TTI. The discarded transactions may be without dependency-relations with the lost ones. It may be possible to execute some of the discarded transactions without destroying the distributed database consistency.

In [FER81] it can be found how to build an optimized global recovery line in case of catastrophic failure. This paper describes also how to perform a global state saving so as to release local logs.

2.5. SILOE LAYER

In SIRIUS-DELTA architecture, SILOE is the layer which makes data distribution and processing distribution transparent to the users.
SILOE layer stands between the classical DBMS and the distributed transactional sub-system (SCORE/SER).

At global level, the users manipulate a SIRIUS-DELTA database just like a centralized database, without references to any site or local databases. Global SILOE major function is to transform a global query in a set of related and synchronized local queries to perform the required action, taking into account the data distribution.

At the local level, local SILOE major functions are the language translation (pivot ⟶ local external language) and the local functions needed for SCORE (i.e. temporary update, rollback, commit).

Let's outline SILOE main features :
- the data distribution definition which is the basis of SILOE transformation of the queries,

- the query decomposition which is necessary because data involved in the query processing can be located on several sites and databases,

- the query evaluation/optimization which determines an acceptable (if possible optimal) scenario for coordinated local actions,

- the generation of the distributed execution plan corresponding to this scenario, in respect with the formalism used in SCORE, SER and local SILOE,

- the adaptation/translation necessary to interface different local DBMSs, using distinct query languages and producing results in distinct formats,

- the local functions associated to SCORE local resources management which are provided by local SILOE if they don't exist in the local DBMS.

2.5.1. Data distribution capabilities supported by SILOE :

This section describes the various types of data distribution which are supported by SILOE.

2.5.1.1. Overview

SIRIUS-DELTA is aimed to be used either in a top-down approach of data distribution or in a bottom-up approach (where pre-existing databases will act as local databases in a SIRIUS-DELTA DDB).

In both cases we suppose that the DDB conceptual schema is defined. We assume that this conceptual schema (which is called the global conceptual schema) describes relations, the only constraint being the existence of a primary key in every relation definition.

The distribution of data among the local databases is defined on the basis of various distribution units (a distribution unit or DU is a flexible piece of data which is considered as a whole in the distribution). The largest DU is the relation, the smallest one is a tuple consisting of a primary key value.

Data distribution in a SIRIUS-DELTA DDB can be very complex. Localization of a distribution unit can be defined either in respect with some attribute values in it or as the same localization as another related piece of data. In both cases replication is possible.

A data distribution schema provides SILOE with the description of the data distribution. This schema - called the Global Internal Schema (GIS) - is written using the SILOE data distribution definition language (DDDL).

2.5.1.2. Distribution units

A global SIRIUS-DELTA DDB is defined as a set of relations. Several distribution units can be used to define its distribution.

* Relation (no partitioning)

The global relation is entirely located on one or several local databases.

* Tuple (horizontal partitioning of R)

Tuples of R are not transformed when distributed but all tuples are not located on the same local database(s). Localization of every tuple of R is decided upon its value.

* Sub-relation (vertical partioning of R)

All tuples in R are splitted (the primary key value is repeated in every piece) and localization is defined per sub-relation.

* Sub-tuple (horizontal partitioning of a sub-relation)

All the tuples of a sub-relation don't go on the same local database. Localization is decided for every sub-tuple.

2.5.1.3. Localization rules

A localization rule tells on which local database(s) a given distribution unit is or must be.

- Non-dependent localization (for relations or sub-relations)
In this case, the local database (or local databases) is (or are) defined without condition.

ex : DU1 ON local-database-3
DU2 ON local-database-1, local-database-4

- Direct dependent localization (for tuples or sub-tuples)
Localization now depends on the value of one or several attributes in the tuple or sub-tuple.

ex : DU5 ON local-database-1 WHEN att2 xxx, ...
DU7 ON local-database-2, local-database-4 WHEN att1 = ..., ...

- "VIA" function i.e. indirect dependent localization (for tuples or sub-tuples)
This localization rule expresses the fact that a tuple or sub-tuple is located on the same local database(s) than a tuple or sub-tuple of another relation (which contains the value of the primary key attribute(s)).

ex : DU10 VIA att3 FROM DU25
(DU25 has its own localization rule, which can be non-dependent, direct or indirect).

2.5.1.4. SILOE data distribution definition language (DDDL)

Distribution units and localization rules are defined in the Global Internal Schema (GIS) using SILOE-DDDL.

In order to express very complex distributions, we allow the definition of several localization rules for a given DU. We call Homogeneous Distributed Set (HDS) the data set which corresponds to one DU type and one localization rule. The set of data corresponding to a given HDS on a given local DB is called plot.

A GIS consists of three sections :
. the HDS section where the distribution unit types are defined (that is to say the various local relation types) associated to a given localization rule
. the mapping section which defines how the global database can be reconstructed from these HDS
. the localization section where the localization rules are given for each HDS.

The precise definition of the partitioning in several HDS is given in the mapping section where predicates allow to express how the relation is sub-divided.

Then, plots associated to each HDS is described in the localization section. Plot partitioning can only be an horizontal one. It means that the data type of the HDS on every local database where it appears is the same (reason for the term "homogeneous"). Predicates are used to define the horizontal partitioning.

Predicates can be complex expressions using AND, OR ... and comparative operators.

Example :

```
GCS : RESORT (NR, ALTITUDE, STATION) KEY = NR
===   HOTEL  (NH, NR, NB-ROOMS)      KEY = NH

GIS : * HDS SECTION
===
        RESORT1 (NR, ALTITUDE, STATION) KEY = NR
        RESORT2 (NR, ALTITUDE)          KEY = NR
        RESORT3 (NR, STATION)           KEY = NR
        HOTEL1 (NH, NR, NB-ROOMS)       KEY = NH
        HOTEL2 (NH, NR, NB-ROOMS)       KEY = NH

      * MAPPING SECTION

        RESORT = RESORT1 WHEN ALTITUDE > 800
        RESORT = RESORT2 + (*) RESORT3 THRU NR
                         WHEN ALTITUDE ≤ 800
        HOTEL = HOTEL1   WHEN RESORT1
        HOTEL = HOTEL2   WHEN RESORT2

      * LOCALIZATION SECTION

        RESORT1 ON LOCAL-DB-1 WHEN STATION = "LYON",
                ON LOCAL-DB-2 WHEN STATION = "NICE"
                              OR STATION = "GRENOBLE",
                ON LOCAL-DB-3 ELSE
        RESORT2 ON LOCAL-DB-1 WHEN ALTITUDE <500,
                ON LOCAL-DB-5 ELSE
        RESORT3 ON LOCAL-DB-6, LOCAL-DB-7
        HOTEL1 VIA NR FROM RESORT1
        HOTEL2 VIA NR FROM RESORT2
```

2.5.2. Global SILOE functions :

We describe query decomposition algorithm for retrieval queries and updates and the generation of the distributed execution plan to be executed.

2.5.2.1. Query decomposition

Retrieval queries are decomposed into the following steps :
- definition of the conceptual tree (CT)
- production of internal trees (IT)
- reduction of ITs into reduced internal trees (RIT)
- localization of the optimal reduced internal trees
- choice of the final reduced internal tree

2.5.2.1.1. Conceptual tree

The global SILOE-layer receives the user query. The query is translated into a tree call the conceptual tree (CT). The CT is expressed in pivot language which uses relational algebra operators that become the nodes of the tree. In the CT we find all conceptual relations involved by the query.

(*) (+ : join operator)

2.5.2.1.2. Internal tree

In fact the conceptual relations are fragmented into HDS and plots, as specified in the global internal schema where the portioning of the conceptual relations is described. Using informations and the CT we construct an another tree that we call the internal tree (IT).

In fact from one global conceptual tree one can obtain several global internal trees. This, because there are different manners to rebuild the conceptual relation from the plots. For instance the conceptual relation R (figure 4) may be constituted from four plots and may lead to two trees (figure 5.1 and figure 5.2).

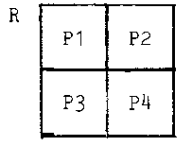


Figure 4

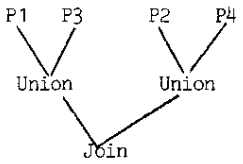


Figure 5.1

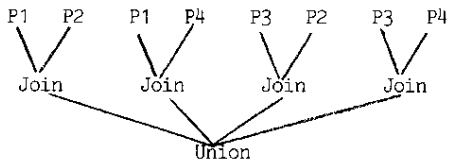


Figure 5.2

2.5.2.1.3. Reduced internal tree

For some queries it is not necessary to consult all corresponding plots The reduction of the global internal tree is the minimal tree which allows to answer to the query. The reduction may be obtained in three different manners :

- through projection : if no attribute of a HDS is requested then we eliminate this HDS
- through restriction : possible only if the localization function of the HDS or of the plots is defined by a predicate. The restriction itself is defined by using predicate. Comparing the localization function predicate with the restriction predicate we deduce whether HDS or plots are to be eliminated or not
- through Via function : if plot i is eliminated and plot k is located by a via function from the plot i then plot k has to be eliminated.

When a plot is suppressed in the global internal tree then one or more internal nodes can disappear. For instance the suppression of the plot P1 decreases the tree from figure 6 to the tree from figure 7.

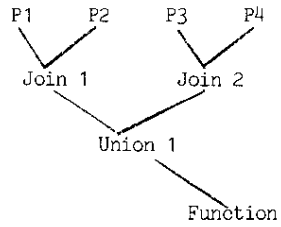


Figure 6

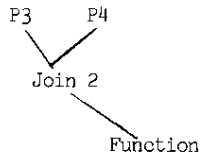


Figure 7

The join between P1 and P2 becomes empty and the join operator "Join 1" disappears. The suppression of the "Join 1" causes the suppression of the union operator "Union 1".

A final tree is called the reduced internal tree (RIT). At this step only the external nodes and the root are localized. They respectively correspond to the different plots located on the different storage sites and to the result site. To all internal nodes correspond relational operators (join, union), their execution localization is the next step of the processing.

2.5.2.1.4. Localization of the optimal reduced internal trees

In order to minimize the network data flow we calculate the communication cost of each RIT. For each node we estimate the size of the intermediate result (in bytes). The sum of the transfer results sizes, gives us the communication cost. The optimal RIT is the one corresponding to the minimal cost.

The estimation of the cardinality relies on [DEM 80], [SMI 75], [YAO 79]. The algorithm is described in detail in [LAC 81].

The localization of external nodes of each RIT is known from GIS since they correspond to plots. The localization of internal nodes is done progressively during the communication cost evaluation of the RIT.

2.5.2.1.5. Choice of final reduced internal tree

The final RIT is the cheapest tree, from the set of the optimal RIT, which has all its sites accessible.

2.5.2.1.6. Data update

The user can update data with CREATE, MODIFY and DELETE commands. Each command is translated into an IT. The retrieval sub-tree is reduced and the optimal RIT is choosen. Then the all tree is constituted by the concatenation of the optimal RIT and the update sub-tree. The final update RIT is the concatenation of the final reduced internal tree and the update sub-tree.

2.5.2.2. Distributed execution plan generation

The internal notation used for the final tree is the post-fixed notation. The functions are the pivot-language operators. The operandes are either an external node (a plot) or a subtree.

From the final tree, global SILOE produces the distributed execution plan (DEP) and transmits it to global SCORE. DEP is made of set of local action contexts. A context defines the schedule of a local action.

A local action is specified by SILOE in the following way : the tree is analyzed from left to right in order to find the site associated to each function and the site of its operande. The operande defines a local action when its site differs from the site of its function. A local action is a mono-site sub-tree.

Then, any reference to that operande in the final tree is replaced by a reference to the temporary data file that will be produced.

Example : Let us assume the following final tree (fig. 8).

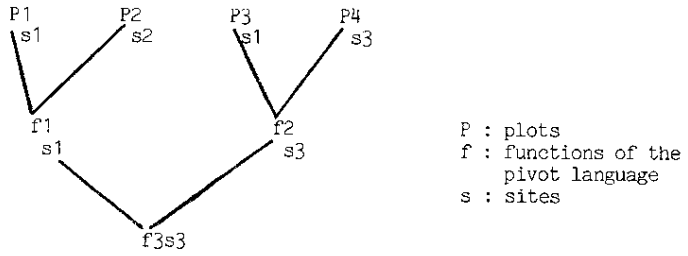


Figure 8

This tree leads to DEP composed of four local actions (fig. 9).

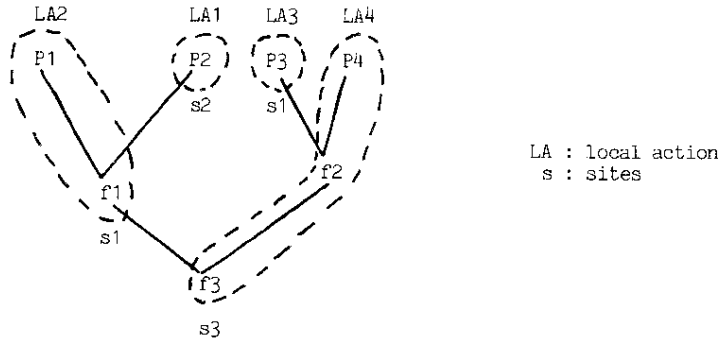


Figure 9

Local actions exchange values associated to synchronized variables and temporary data files in a producer/consumer mode. These specification are stored in the related local action context in a symetrical way. Thus, two phases are required by SILOE to fully generate a DEP : first all consumers, then all producers.

The choice between the two possible end expressions depends on the query and the local action type. For example, the process of an update query requires first a search for the tuple identifiers to be updated, then a test on the result (empty/not empty) that is used to optionally launched the update sub-queries. In such a case, the second end expression is used to prevent the launching of updating local actions while the searching local actions are active. When dealing with inquiries, the first end expression is generally used to allow maximal parallel processing.

The resources allocation requests are produced by the analyze of the local action. If this sub-tree contains a reference to a plot, an allocation request for this plot is produced, in read or write mode, depending on the sub-query type (inquiry or update).

Then, this DEP is submitted to global SCORE.

2.5.3. Local SILOE function :

Local SILOE ensures the interface with the local DBMS. The main module of local SILOE is scheduled by local SER at the launching of the local action. Local SER transmits to local SILOE the tree expressed in the pivot-language associated to this local action.

SILOE is responsible for the translation of this tree in a (or a sequence of) command in the local DBMS data manipulation language. The translation of the local tree uses a local external view where the local relations belonging to the DDB are described using the structure from the global conceptual schema and the global internal schema. This view allows the homogenization of the local relations and the mapping between global and local description of these relations.

If some operators of the pivot language don't exist in the local DML, programs making possible to execute these missing operators are added in local SILOE.

Local SILOE manages the scheduling of DML commands and programs, by using, when necessary, working files in order to store temporary results.
To do so, local SILOE looks at the tree description and isolates sub-trees. These sub-trees are translated into either a command of the local DML or a command of launching of a local SILOE program. This translation uses the local SILOE view.

When all the tree is executed, local SILOE stores the result either in a temporary data file to be transmitted to another local action by local SER, or in an edition file so that it is returned to the user (see figure 10).

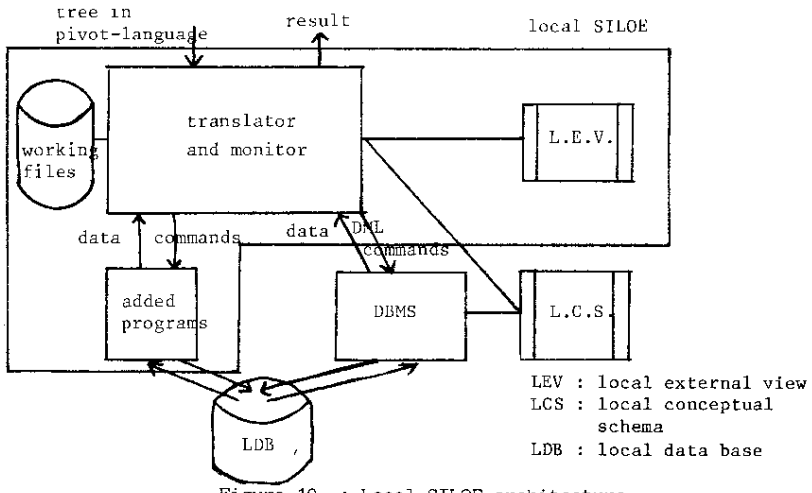


Figure 10. : Local SILOE architecture

2.5.4. User's facilities :

Several user's facilities that exist in SIRIUS-DELTA are pointed out.

2.5.4.1. Catalogue of a query

Since a query may be localization sensitive, or since it may deal with copies that may not all of them be up, then the query should be decomposed at run time. On the other hand, why should the same query be decomposed each time it is submitted ? Today, the fact that query decomposition may be very costly is well know.

We have included an option to catalogue a query at user's request. Several scenarios are stored with their related list of required sites and cost. For each list of required sites only the least expensive scenario is kept.

When a catalogued query is submitted again, SILOE searches for the least expensive scenario among those that match the up-list.

2.5.4.2. DDB design considerations :

Some DDB design thoughts have been involved in our approach to data distribution.

- Although it would have been functionnally feasable, no attempt has been done to allow for a recursive description of data distribution. We deal with a "flat" description of the data distribution since we consider that this is the physical characteristic of the data belonging to the DDB.

- Mappings between global and local data types may become very complex as soon as the DDB is built from a cooperation of pre-existing DBs. The HDS concept permits to describe that mapping at a logical level, without having to be concerned with the problems of "where" and "how" local data is maintained.

- A major DDB change is the addition of a DB in the configuration.

At global level, it results in recompilation of the GIS to define distribution rules for the new tuples on existing relations. If new relations are added, the GCS must also be amended and recompiled.

At the local level, a local external view must be defined, to permit compatibility between local data and their global presentation. Finally global SILOE is expecting a set of operators used by the pivot-language. Therefore if the local DBMS or DM does not contains these features, they must be added.

2.5.5. SILOE status :

At present time SILOE is providing some DDB design facilities such as the DDDL, and data distribution functions, that permit any combination of horizontal and vertical partitioning of the global relation.

Query decomposition does static optimization using an heuristic, and can be requested both at "precompile" and at "run" time. Heterogeneity is allowed at global and local level thanks to the pivot-model and the pivot-language.

Integrity constraints must be included in the transaction design, and are maintained using atomicity of the transaction. A user authentificaiton routine is implemented using pass-word at global level, and some data access control is achieved through the Global External View. Extended data access control that includes access prevention at relation, attribute or attribute value is under testing.

2.6. PHLOX AS A SERVER IN SIRIUS-DELTA

2.6.1. The PHLOX project :

The aim of the PHLOX project is the design and the realization of DBMS packages for micro-computers [BDV80].
Each package is intended for a particular use :
- PHLOX1 for individual micro-computers (mono-user, mono-server)
- PHLOX2 for a dedicated server in a local network (multi-user, mono-server) [FER81a]
- PHLOX3 distributed data bases management system (multi-user, multi-server).

In administration, PHLOX offers the possibility of describing a new data base at three completely separated levels which are those of ANSI-SPARC [ANS75] : an external level and a conceptual level using the relational model [COD70] and an internal level based on the DBTG's network model [COD71].

In manipulation, PHLOX offers two high level languages : a navigational data manipulation language and a relational data manipulation language.

The systems themselves manage all the problems that can be solved transparently for the user : access paths, consistency, recoveries, deadlocks, ...

2.6.2. PHLOX in the SIRIUS-DELTA prototype :

The SIRIUS-DELTA architecture has been designed in order that existing DBMSs can be included in a distributed DBMS. The possibility of building an heterogeneous (in term of DBMSs) DDBMS with SIRIUS-DELTA is being proved by the realization of an heterogeneous prototype using PHLOX as a server.

In this prototype, PHLOX is not an access point to the DDBMS, but it is only a storage point, it only supports the functions of a local DBMS (see figure 11).

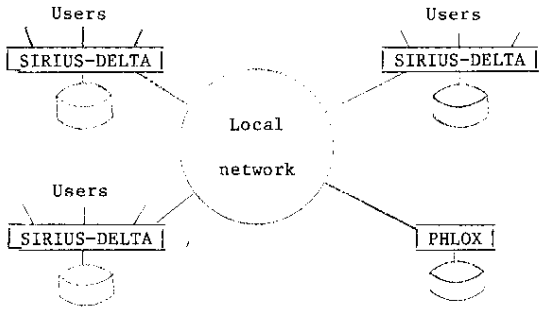


Figure 11. : Heterogeneous prototype of SIRIUS-DELTA with a PHLOX server

The PHLOX server only deals with its own data. In today's version, it cannot take into account partial results sent by other sites. It receives requests and sents results to one (or more) SIRIUS-DELTA site(s) which can exploit them or present them as the result of the global request. PHLOX receives and processes requests of consultation as well as of modification.

In order to cooperate with SIRIUS-DELTA, PHLOX has to participate in the distributed protocols of SIRIUS-DELTA (i.e. DEP local handling) : it has to be able to understand the messages sent by the other sites, to recognize the data base requests and to ensure the consistency of the distributed data base. Some of these functions could already have been assumed by the PHLOX2 prototype and other ones have been added since, as shown in the next section.

2.6.3. Implementation of the SIRIUS-DELTA protocols in PHLOX :

The protocols designed by SIRIUS-DELTA have been implemented in PHLOX using as far as possible the existing software of PHLOX2. Only some functions have been added through a new level called SIRIUS-DELTA interface.
The different levels appear in figure 12 below.

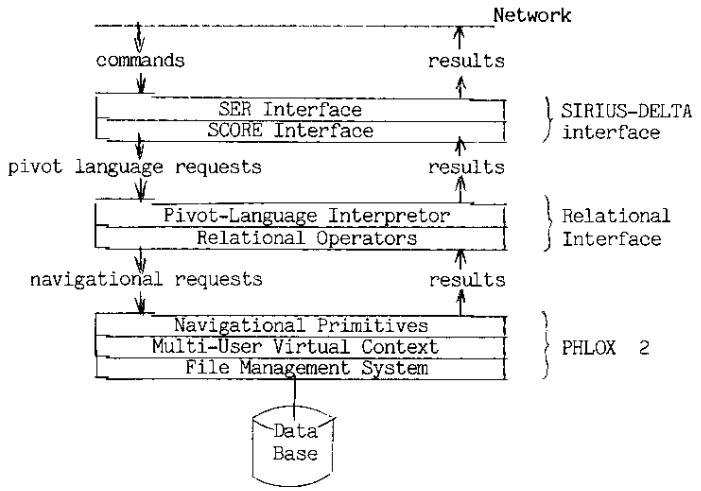


Figure 12. : The three levels of a PHLOX server

2.6.3.1. SIRIUS-DELTA interface

This interface manages the messages which are exchanged between the PHLOX site and the SIRIUS-DELTA sites. It analyses the received messages, it sends the requests to perform to the lower level (relational interface), it treats the system commands of SER (e.g. "send the next records of the result") and of SCORE (e.g. "abort transaction number t") and it sends the messages expected by the other sites (e.g. "local action performed").

2.6.3.2. Relational interface

The aim of this interface is the translation of received pivot-language relational requests into PHLOX navigational requests. This translation requires, moreover, during the administration phase, a mapping from the relational schema of the local database, as viewed by the global DBMSs, into the internal network schema used by PHLOX for accessing data.
This interface has also to prepare and to send the results of the received requests to the upper level.

This relational interface consists of two layers : a pivot language interpretor and relational operators.

2.6.3.2.1. Pivot-language interpretor

The pivot-language has been designed by the SILOE team of SIRIUS-DELTA and the PHLOX team. Thanks to this common language, requests can be sent from the global sites producing the transactions to the local sites which perform the sub-transactions. It is an algebraic language based on the relational model, and it expresses a request by a tree in post-fixed notation.

The interpretor manages the execution and the chaining of the relational operators. It also realises the mapping from the external names of relations and attributes corresponding to the pivot model into the local names corresponding to the conceptual local schema described in the tables of PHLOX.

The "Pivot-language Interpretor" layer has different functions :

- functions of optimization :
  a) reorganization of the sub-request tree,
  b) clustering of relational operators which can be executed simultaneously,
  c) choice of the algorithms to apply in order to use at their best the access paths existing in the network schema of the data base.

- a function of interpretation :
  it consists in managing the chaining of the calls to the relational operators.

2.6.3.2.2. Relational Operators

At this level, we find the three relational operators : project, select and join. The operations on sets are the union, the intersection and the complement. It's possible to execute easily the operations of insertion, suppression and update thanks to the existing primitives.

The project, select and joint operators use the navigational primitives of PHLOX. They have been implemented so that they use the access paths and the operators already implemented in the data base. The inversion index are used to realize the selection in such a way that the system access only to a limited amount of data. The father-son's links of the network schema correspond to already implemented joins the use of which makes possible the decreasing of the reading number.

2.6.3.3. Existing functions in PHLOX2

The part of the distributed protocol already existing in PHLOX2 consists mainly in the consistency and resiliency functions. The mechanisms which ensure data sharing and reliability are integrated into the inner part of PHLOX2, inside the level called multi-user virtual context management.

- Data sharing :
Data sharing is ensured in PHLOX by setting locks on objects with the same principle as in SIRIUS-DELTA. The main difference between both prototypes for data allocation is that SIRIUS-DELTA allocates the data needed by a request before beginning to perform it, whereas PHLOX, because it locks logical pages that it cannot wholly know in advance, sets the locks only as the request proceeds, at the moment when the page becomes really necessary. But both systems have the same external behaviour.

- Data base integrity :
At the end of a transaction, the PHLOX system checks that the modification made by the transaction leave the data base in a consistent state. If, and only if, the modifications are accurate, they are then copied in the data base itself.

- Commitment :
PHLOX manages a two step commitment [BAE81] : in the first step, the pages modified by the committed transaction are saved on a disk and, if the final commit order is given, the second step is initiated and the pages are then copied in the data base and they are unlocked. So, if a breakdown occurs during the commitment, the system restores the data base to a consistent state.

In order that PHLOX can participate to the distributed protocol of consistency and resiliency, it has to converse with the other systems in the network. This is done in the SCORE interface (see figure 12) ; this level receives and sends the abort orders, sends the acknowledgements after each step of the commitment and works with the other sites for the recovery after a breakdown.

2.7. MEASURES ON THE PROTOTYPE

The existence of the prototype rises questions about performances : is the implementation efficient ? What is the cost of the different layers ? What is the cost of the primitives in a layer ? How is parallelism used ? Are the effects significantly profitable or the system cost overheads it ?

We must immediatly point out that it is not possible in a prototype to measure a real application since there is none currently. Also, we have limited our effort on the cost of the major system elements.

In order to give a flavour of what could be the cost of an application we will decompose the cost of a transaction. But before anything we had to find a method to get general enough results. Response time is in fact too much linked to the machine characteristics. Our point of view is that the number of instructions needed to perform a function is more general. It does not depend on the machine speed. It is impacted by the language used to program the prototype but as it is commonly accepted that a compiler has an extension rate, it can be corrected according to the efficiency rate of the compiler. For these reasons we will give most of the result as a number of executed instructions. Sometimes we can use the ellapse time to get a flavour of where are the waiting times.

2.7.1. Overheads :

By overhead we mean all permanent processes needed to schedule, manage the virtual ring, manage the network. This overhead on our very slow machines represent about 21 % of the CPU activity. 14,5 % are due to the network management and 6,5 % to the virtual ring management.It is obvious that this is a prohibitive cost, but in our prototype it is mainly due to our very slow machine (about 40 μs per instruction). A more efficient machine would reduce greatly that owerhead.

2.7.2. Network cost :

The novelties in a distributed system is the use of a network to communicate. Also the efficient of the network impacts deeply the results. We have observed that a complete message exchange cost about 7600 instructions which are devided in 3700 instructions at the emission and 3600 at the reception.

It is obvious that a front end processor in charge of releasing the main processor of the network management reduces widely its load.

2.7.3. Distributed executive cost SER :

The different functions of the distributed executive system are : remote activation of a local action, synchronisation variable activation, temporary file transfer installation and temporary file items file transfer.

2.7.3.1. Local action cost

This cost is divided into two parts, firstly the cost at the global site which receives the DEP and sends each local action to the appropriate site, secondly the cost of the local site which receives that local action and creates a context for it.

On the global site the observed cost for n local actions is : n 37500 + 10 000 instructions. The constant is due to the entry in the program. We can see that for each local action the cost is 37 500 instructions, among them 15 200 are the result of the messages emission and reception cost involved by the protocol. This represent 60% of the cost. The reader can remark that it would be better to send in one message all the local actions dedicated to one site, since the cost of the message is rather important. On the local site since each local action is received alone, the cost is a constant and the observed value is 45 000 instructions. The cost of protocol's messages is 15 200 instructions, about 34% of the global cost. In order to give an overview of the elapsed time cost the fig. 13 shows the elapsed time in the local action installation process. We can observe that message transfer delay is the most significant elapsed time cost.

2.7.3.2. Synchronization variable

The cost is divided into the cost at the activating site and at the reception. The site which activates the synchronization variable (SV) send. Also the observed cost for n SV is : n 6000 + 32 000 instructions. Notice the important cost for program initialization.

The site receiving the SV activation has to set the SV its new value and then to test if the corresponding process becomes ready. This explains that the observed cost is not linear. The observed curved is given on fig. 14.

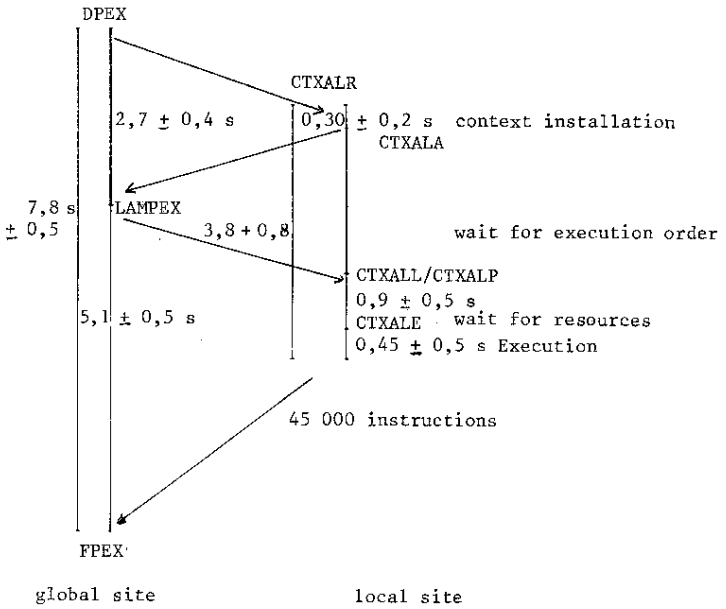


Figure 13. : Elapsed time in the local action installation and execution process (the local action is empty)

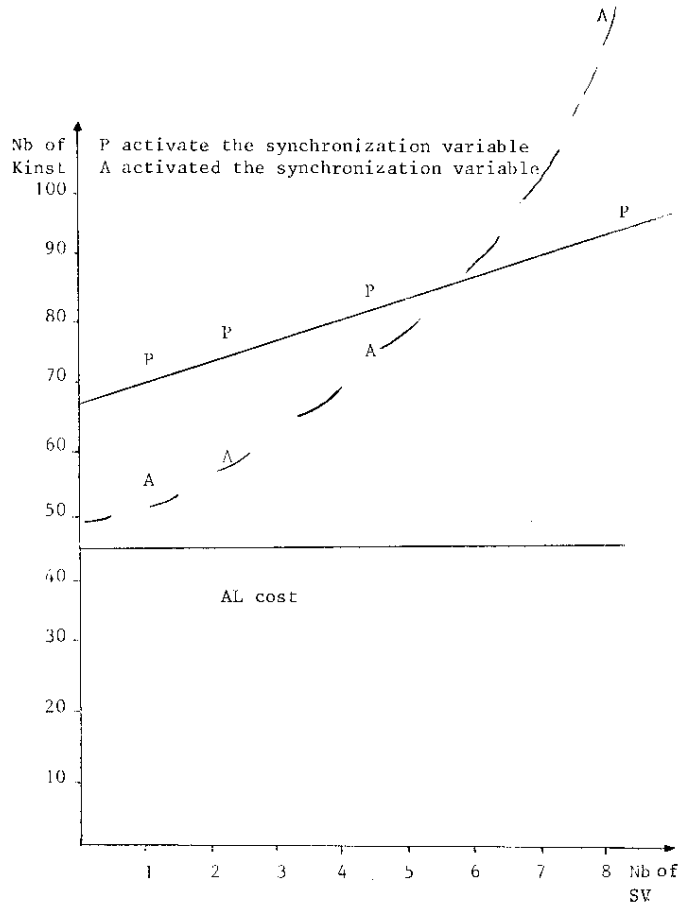


Figure 14. : SV activation cost

2.7.3.3. Temporary data files and data transfer

Creating a temporary data file (TDF) needs a context and so there is an initialization cost, which is not exactly the same at the producer and at the consumer site. Observed result are for the producer of n TDF : n * 25 250 + 37 500 instructions, for the consumer of n TDF : n * 35 000 + 10 000 instructions. The reader will notice that the cost for initiating an TDF is about the same as the one for a local action.

Data transfer is much more important since it is a major function used to execute a distributed query. The first implementation transferred item by item, the transfer cost of n item, at the producer site, item by item is : n * 18 500 + 50 000 instructions and at the consumer site n * 25 850 + 23 000 instructions. Also a single item transfer cost about 44 350 instructions which is quite prohibitive.

Only 7600 are used for the message. This cost is caused by a very unefficient implementation in Basic based on item read and write in a new file ; each time a layer is crossed (SILOE <--- SER <---, Transport). Observing very soon this bottleneck we grouped item before the transfer. Two different blocking values have been measured, 7 and 10. Results are reported on fig. 15. The reader will notice that we have not a linear curve, and that a blocking value of 7 improves significantly the result. The blocking value of 10 gives results quite near the value 7. Of course, the blocking mechanism reduces the parallelism since the consumer site has to wait at least for 7 item before to begin even if the producer had made available items longer before. There is obviously a trade-off between parallelism and efficiency.

2.7.3.4. SCORE cost

The first operation of a transaction is to get a ticket. This includes for SCORE the management of a context at the global site. The cost observed for that primitive is of 110 000 instructions.
The two-step commitment protocol cost is made of :
- a cost at the global site for n involved sites is : n * 12 000 instructions. This cost is mainly due to the messages used by the protocols,
- a cost at each local site of about 90 000 instructions. This cost is important since two items have to be written in the journal. This cost does not depend of the number of data objects accessed.

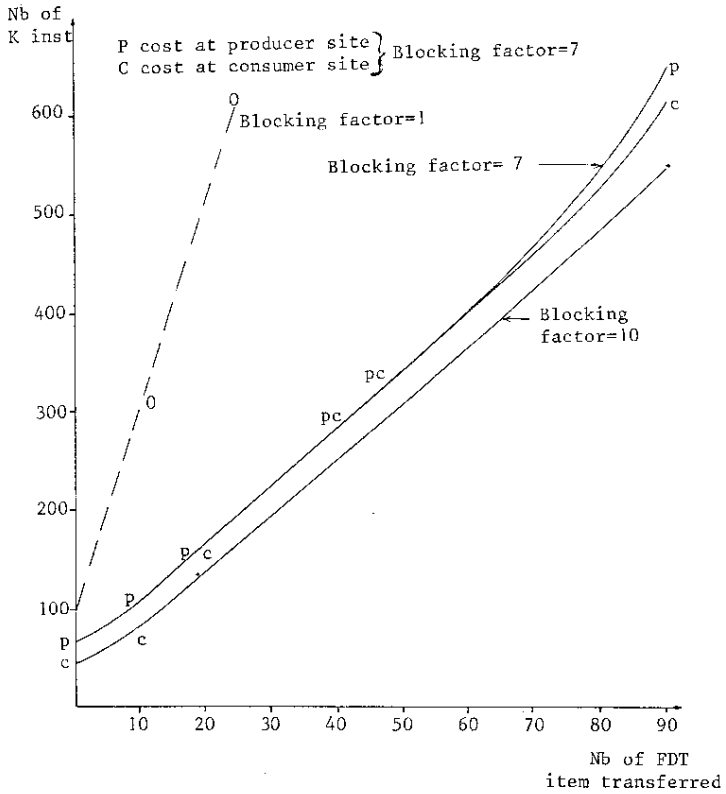


Figure 15. : Item transfer cost for three different blocking factors

Locking object in the measure includes the releasing phase. The observed cost is of 30 000 instructions. This does not include the before-lock process each time an object is modified.
The cost of a single call before-look process is about 2000 instructions with an initialization cost of 30 000 instructions.

The cost of the different aspects of SCORE are rather moderate. Locks are expensive if the granularity is small. We didn't measure the cost of roll-back since it is very difficult to repeat it systematically.

2.7.3.5. SILOE cost

Among the different functions of SILOE we have only measured the main relational operators. The query decomposition cost and DEP generation would be interesting, but it is difficult to define cost factors in them. The cost of JOIN, UNION and PROJECTION has been measured in the distributed context. This is not very specific of a DDBMS but will be useful in the next chapter to evaluate the cost ratio between work useful for the query execution and work involved by the query in the DDBMS.

SILOE exchanges local actions using a pivot-language. This pivot-language must be interpreted at each local site. The observed cost of the interpretor was about 100 000 instructions and seems not depend greatly of the sub-query complexity. This means that most of that cost is due to context set-up.

The observed union cost is about 700 instructions per operation. Join cost depends on the result. If it succeeds then it is about 1560 instructions if it fails then it is 940 instructions. Projection cost is about 2000 instructions. It has not been possible to get the cost of the restriction for implementation reasons.

2.7.3.6. Example and conclusion

We want now study a transaction in order to have an index of where are its costs. For that we have used all the previously described results and reported them on the fig. 17. The transaction is the following :

```
Begin Transaction
     List activity with zone = "mountains" and with activity.type =
     "excursion" describe-activity, fees, correspondant
End Transaction
```

We describe below the transaction graph and give a list of required ressources (fig. 16).

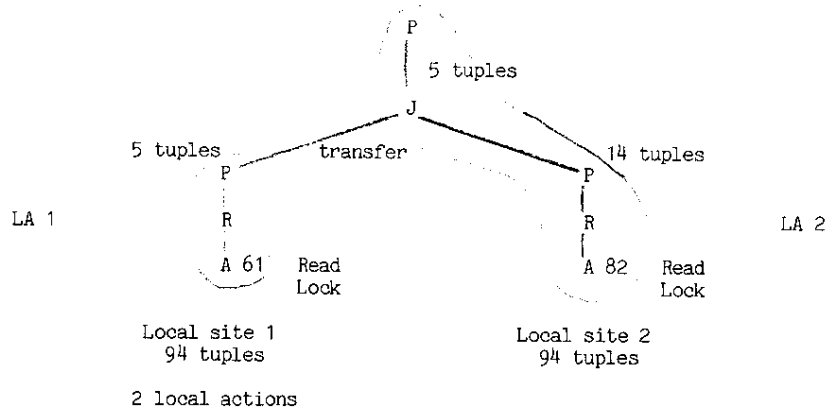


Figure 16. : Transaction graph

Using all these informations we are able to compute the different costs induced by the SER, SCORE and SILOE layers. The results obtained are not detailed here, they are reported on fig. 17.

This figure shows clearly that even with a very simple transaction, only one step, and a simple query in that step working on a little experimental data base, more that 50 % of the executed instructions are dedicated to user work execution.

Of course system cost is rather important and effort must be done to industrialize the prototype. We can also observe that most of the user work is executed in parallel. In fact more that 80 % of this transaction is executed in parallel on the two sites.

As a quick conclusion we can say that the use of a distributed system will offer the possibility to parallelize at an efficient cost transaction that would be to long to execute on a single site. System cost does not overhide parallelism since it uses itself parallelism potentialities. Improvement in the design and realization of the transport function will certainly reduce global cost and message waiting time.

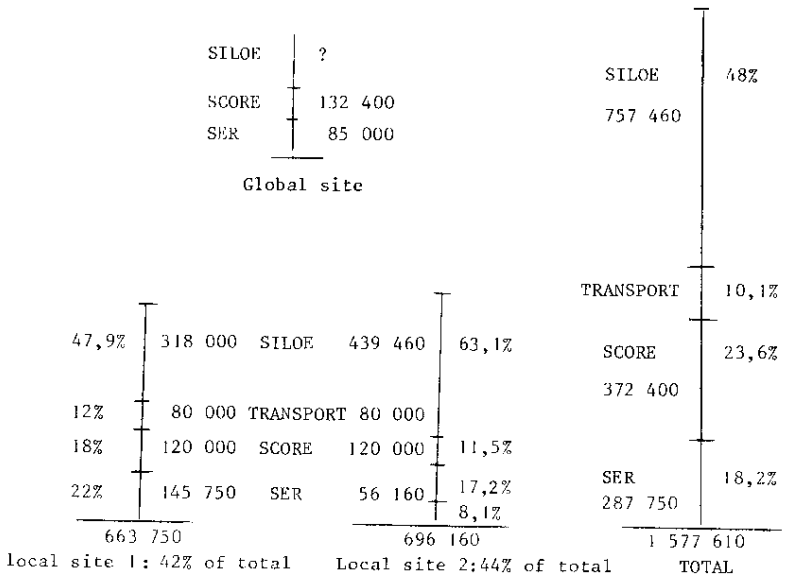


Figure 17. : Costs induced by SER, SCORE and SILOE layers

2.8. CONCLUSION

The prototype SIRIUS-DELTA, initiated in 1979, is now operational. It is a pre-industrial prototype the aim of which was to validate research studies within the SIRIUS project. Focus, in SIRIUS-DELTA, was on data distribution and distributed processing aspects of an heterogeneous Distributed Database Management System.

We think that it has proven feasibility of our approach. Application studies show that systems like SIRIUS-DELTA should be useful for a large class of applications.

3. MULTIDATABASE APPROACH.

3.1. PRINCIPLES OF THE MULTIDATABASE APPROACH.

3.1.1. The idea in the database approach.

Any database (DB) is defined according to some data model. The relational data model [COD71], [COD79] considers that a DB is a set of relations. The other models consider that elements of a DB are record types, or entity types, or components etc. As the names of the elements indicate, the implicit principle of all these data models is that the elements of a DB are not themselves DBs. We will call any DB constituted from such elements a (logically) centralized DB or, shortly, a single DB.

The fundamental principle of the present database approach is that a universe is modelled with a single DB [SMI81], integrated or federated [HAM79] (fig. 18a). There cannot be data that model some universe and are in more than one DB. If such a thing seems to happen, especially when one gains access to a previously unknown DB, it means that the universe revealed wrongly perceived. A single DB corresponding to the updated perception of the universe must be defined, prior to data manipulation.

This DB may, in particular, be a distributed DB (DDB) in the sense used in this article. It is thus defined by a global schema that one assumes therefore always definable !! Users manipulate this (single) DB or its views. They have no knowledge of the underlying DBs.

3.1.2. The idea in the multidatabase approach.

The fundamental principle of our multidatabase approach is that a universe is typically modelled with several databases [LIT80] (fig. 18b). Some of DBs model (sub)universes that are rather distinct : universe of cinemas, universe of restaurants etc (ex. 3.1). Others model differently the same universe (restaurant guides in ex. 3.2). Some DBs may be derived from others, including, in addition, data on their own (a personal DB about the restaurants derived from a public DB, ex. 3.3). Finally, one may discover the existence of DBs relative to his universe in the moment he needs them for the first time (one may discover that his universe includes QUANTAS airline and the corresponding DB only when he needs to fly to Australia).

This variety of DBs that may correspond to a universe means that the corresponding global schema should typically be unknown. The general case should be the one where a user knows that he manipulates several DBs. The general goal of our approach is to render these manipulations easy. The historical evolution of the database approach and the sense of realities, imply that "easy" means for us the following goals :

1. One should dispose of all the possibilities that may result from the concept of a (D)DBMS, extended to the case of a collection of DBs. For instance, one should be able to define a view of a collection of DBs. of several (D)DBs, instead of only one.

2. Multidatabase manipulations should be formally expressible in a data manipulation language. The goal of such a language should be one assertion (command), per manipulation. This was already the goal of assertional languages (ALPHA, QUEL etc) with respect to older navigational languages [COD82].

3. One should be able to formulate constraints preserving integrity and privacy of data that are not within the same DB.

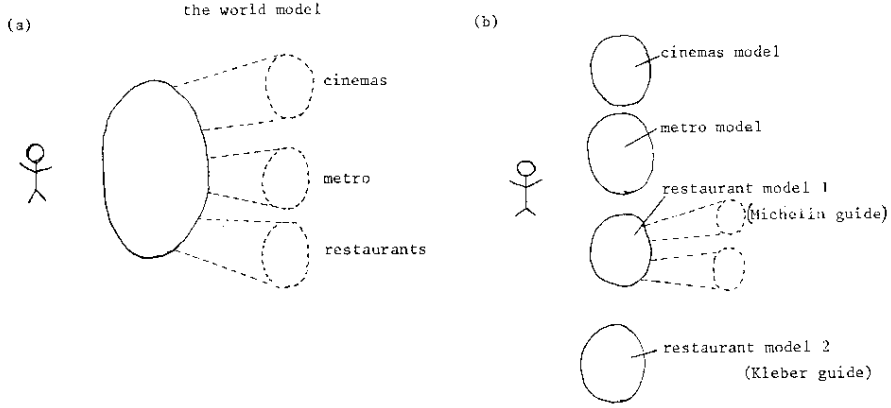


Figure 18 : Database approach (a), Versus multidatabase approach (b), to the reality modelling

we first present the concept of a multidatabase that we introduce in order to satisfy these goals. Then case studies identify user needs. Afterwards we propose the concept of a multidatabase management system and the corresponding general architecture. Finally, a language that we call MALPHA shows a way to satisfy the goal (2). Goal (3) will not be discussed in what follows, proposals relative to this goal may be found in [LIT80] and [ABC82].

Since the goal of the multidatabase approach is the management of several DBs, it must ipso facto include the subgoal that is the management of one DB. Thus our approach is a generalization of the database approach.

3.1.3. The concept of a multidatabase.

We call multidatabase (MDB), a set of databases or of multidatabases presenting the following properties :

(i) - there is a data manipulation language to express manipulations of data that are not within the same DB (multidatabase manipulation language).

(ii) - there is a data definition language to define data within the MDB and, eventually, its structure and the dependencies between DBs or MDBs (multidatabase definition language). The dependencies may be semantic dependencies or integrity dependencies or privacy dependencies etc [LIT80].

Since any MDB includes some single DBs one may consider that the concept of an MDB is a generalization of the present concept of a DB. An MDB is not a DB if one considers that a DB must be a single DB. It is a DB if one considers a DB as a collection of data that is provided with data definition and data manipulation languages, that allow it to be consistent and protected. MDBs become then a yet unknown type of DBs. One may then define the concept of an MDB recursively as "a DB that is a set of DBs" [LIT80], [LEB80]. One may then also consider that an MDB is a logically distributed DB [LIT81].

An MDB may be physically centralized. It may also be physically distributed. DBs of an MDB may in particular be DDBs that means that the schemas of these DBs may be global schemas. The general properties of MDBs are extensively analysed in [LIT81].

A relational MDB is an MDB that is a set of relational DBs or a set of relational MDBs [LIT81]. A relational MDB is thus a collection of relational DBs. It may be a set of sets of relations or a set of sets of sets of relations etc. For obvious reasons, relational MDBs constitute a particularly important class of MDBs.

3.2. CASE STUDY.

The examples that follow define some relational MDBs. They have been chosen in order to determine typical user needs. MDBs are defined and manipulated according to MALPHA language [LIT82]. As its name suggests, MALPHA is a (multidatabase) a generalization of ALPHA language of Codd [COD70].

Example 3.1.

Let us consider the following DBs :

```
DB RESTAURANTS
   R ( R#, RNAME, STREET, TYPE, TEL)      Restaurants
   C ( C#, CNAME, NCAL)                   Courses
   M ( R#, C#, PRICE)                     Menus
ENDDB

DB CINEMAS
   C ( C#, CNAME, STREET, TEL)            Cinemas
   M ( M#, MNAME, KIND)                   Movies
   P ( M#, C#, HOUR, PRICE)               Projections
ENDDB

DB METRO
   M ( M#, MNAME)                         Metro lines
   S ( S#, SNAME, HOPEN, HCLOSE)          Stations
   C ( S#, STREET)                        Streets near a station
   MS ( M#, S#)                           Stations of a line
   P ( S#, S'#, PRICE)                    Station-to-station prices
ENDDB
```

These DBs are single DBs since they are sets of relations. They could be on the same computer managed all by the same relational DBMS: SQL, INGRESS, MRDS or any otner. They could also be on different computers or sites. Moreover, they could be distributed DBs i. e. the above schemas would be the global schemas for SDD-1 or SIRIUS-DELTA or R* or any other DDBMS. For the purpose of the example, we assume however that the system managing these DBs use ALPHA language, instead of the actual languages of these systems.

We now assume that these DBs constitute the set named NIGHTLIFE. Furthermore we assume that NIGHTLIFE is managed by a system providing MALPHA. NIGHTLIFE is then an MDB.

The practical consequences of this fact would be that, on the one hand, users will continue to be able to manipulate each DB as they would do if it was managed by a relational (D)DBMS provided with ALPHA. In other terms, using MALPHA, users continue to be able to formulate ALPHA compatible queries that address relations all witnin the same DB (ex. 3.5). We call such a query elementary query.

On the other hand, the property (i) would allow users to formulate a new type of queries like the following one :

- Retrieve from RESTAURANTS and from CINEMAS the names of all restaurants and of all the cinemas that are on the same street :

```
OPEN NIGHTLIFE                                              (1)
RANGE R X
RANGE CINEMAS.C Y
GET W (X.RNAME, Y.CNAME) : (X.STREET = Y.STREET)
```

This query addresses more than one DB, two in occurrence. We call such queries multidatabase queries. Data obtained using (1) may obviously have importance for someone wishing to see a film and to eat a dinner. Since none of the known (D)DBMSs allows to formulate this query, one may see from this example alone how important new possibilities the multidatabases may bring to users.

The above query has the syntax that is the one of ALPHA query addressing relations that would be named R and CINEMAS.C within a relational DB. In the example, the use of the composed name "database name.relation name" was necessary in order to distinguish between the relation C within CINEMAS DB and the relation C within METRO DB (this was not necessary for R relation). A multidatabase query that acts as an elementary query except that it manipulates relations that are in different DBs, will be called simple multidatabase query (shortly : simple query). Simple queries are obviously practically as easy to formulate as ALPHA queries. On the other hand, for any possible ALPHA query there is a corresponding simple multidatabase query. This means that simple queries alone offer all the power of ALPHA for multidatabase manipulations. Thus if one designs a system allowing even only simple queries, a fundamental extension of present data manipulation possibilities follows.

The examples 3.6 and 3.9 show other mutidatabase queries to NIGHTLIFE.

Example 3.2.

We consider that some editors of restaurant guides wish to put data on computers accessible through a system like TELETEL. We assume that this is the case of famous french guides : Michelin, Kleber and Gault-Millau. Each of these guides describes the restaurants to some extend differently : they do not recommend all the same restaurants, the attribute differ, same meaning may lead to different attribute name etc. Since, in addition, these guides compete for the customers, one may be certain that they would constitute three distinct DBs. This is exactly the case of the guides already available through TELETEL.

Given these premises, we assume for the purpose of the case study that the DBs created by the editors are as follows :

```
DB MICHELIN
   R ( R#, RNAME, STREET, TYPE, STARS, AVPRICE, TEL)      Restaurants
   C ( C#, CNAME)                                         Courses
   M ( R#, C#, PRICE)                                     Menus
ENDDB

DB KLEBER
   REST ( REST#, NAME, STREET, TYPE, FORKS, T#, MEANPRICE, OWNER)
   C ( C#, CNAME, NCAL)
   MENU ( R#, C#, PRICE)
ENDDB

DB GAULT-M
   R ( R#, RNAME, STREET, QUAL, TEL, TYPE, AVPRICE, COMMENT)
   C ( C#, CNAME, NCAL)
   M ( R#, C#, PRICE)
ENDDB
```

In particular, the following differences characterize these DBs :

1 - the quality of a restaurant is measured differently. MICHELIN gives to a restaurant up to three stars (***). KLEBER up to four "forks". GAULT-M apreciation is m/20 ; o =<m =<20. There is no exact translation rule between these measures. Also, the guides may strongly disagree on the quality of a restaurant.

2 - the guides typically disagree also with respect to the value of the average price for a meal and with respect to the number NCAL of calories in a meal. Furthermore, for the purpose of the example, we assume that while MICHELIN and KLEBER indicate the price in french francs (FF), while GAULT-M uses the popular unit called "old franc", (AF), where FF = AF/100.

3. - key values are chosen independently i. e. the same restaurant or course has different key value in different DBs and there is no translation rule between key values. Thus, in order to identify the same restaurant one must a priori use candidate keys, such as phone number or restaurant name and street.

If these DBs are managed each by a (D)DBMS, then one may formulate only the elementary queries. However, users may also have in mind many multidatabase queries. Thus, one may need the query :
- retrieve all the restaurants that MICHELIN or KLEBER considers as chinese (ex. 3.5). Also, one may need the queries from the ex. 3.6 to 3.8.

Such a query either cannot be executed by (D)DBMSs or lead to several elementary queries. In addition, the elementary queries must be formulated differently, despite the same goal. The result would be that, in practice, users would probably never try to use several DBs, as the experience shows already [MOU81], [ROS82].

We now assume that these DBs are elements of an MDB named REST-GUIDES. It might be that REST-GUIDES contains many other guides as well. We will show, as the goal 2 requires, that the above multidatabase queries may be expressed easily, using only one (formal) query.

NIGHTLIFE was composed from DBs that were models of distinct universes. REST-GUIDES is composed from DBs that model differently the same universe. MDBs like REST-GUIDES will be called semantically heteregeneous MDB. The example shows that one will frequently need to deal with such MDBs.

Example 3.3.

We consider that a user of MICHELIN disagrees upon the quality of some restaurants. Furthermore, he is interested typically only in good restaurants, for instance, the ones that are rated at least '**'. Also, he knows some good restaurants on his own. Finally, for the restaurants that interest him particularly, he has in mind additional personal attributes.

It is rather obvious that the owners of a public DB like MICHELIN will not allow this user to modify this DB. The only solution for the user is then to constitute his own personal DB, let it be MY-FAV-REST. The schema of MY-FAV-REST may be as follows :

```
DB MY-FAV-REST
   R ( R#, RNAME, STREET, TYPE, STARS, AVPRICE, TEL)      Restaurants
   C ( C#, CNAME)                                         Courses
   M ( R#, C#, PRICE)                                     Menus
   TEST ( R#, DATE, RESULT)                               User tests
ENDDB
```

The content of MY-FAV-REST may be initially a copy of all data relative in MICHELIN to the restaurants rated '**' or more. R value and, in particular STARS value, may express the user judgment if the corresponding R# value is in TEST, else R may correspond to the original MICHELIN data. Another choice could be to keep in MY-FAV-REST only data about the restaurant that the user has tested.

As he wished, the user may now attribute STARS according to his judgment, add or witndraw restaurants, perform his own control of a restaurant quality etc. He needs multidatabase queries, first, in order to easily create the DB (see ex. 17 of PUT command in [LIT82]). Next, after some time, he will need to add to MY-FAV-REST the restaurants that became rated '**' or more in MICHELIN (ex. 3.9). Also, he may wish to know the restaurants upon quality or price of which MY-FAV-REST and MICHELIN disagree etc. Clearly, first, an MDB is needed also in this case. Next, MDBs mixing public and personal DBs will be obviously of great importance.

Although this subject will not be discussed, one may see that if interdatabase integrity dependencies were handled by the system managing the above DBs, then the updates of MY-FAV-REST due to the evolution of MICHELIN could occur automatically.

3.3. MULTIDATABASE MANAGEMENT SYSTEM ARCHITECTURE.

We call multidatabases management system (MDBMS) a system allowing to manage MDBs. In particular, any MDBMS must allow to formulate simple multidatabase queries. Fig. 19 shows the general architecture that we propose for an MDBMS. We called it three-four level architecture [LIT81].

The MDB is defined at the level that we call multidatabase (conceptual) level. The definition of the MDB is called (conceptual) multischema. A multischema is a collection of the following schemas :

1. - the conceptual schemas of all single DBs.
2. - one or more schemas that define the interdatabase dependencies.

We call the last schema(s) dependencies schema(s) [LIT80], [LIT81].

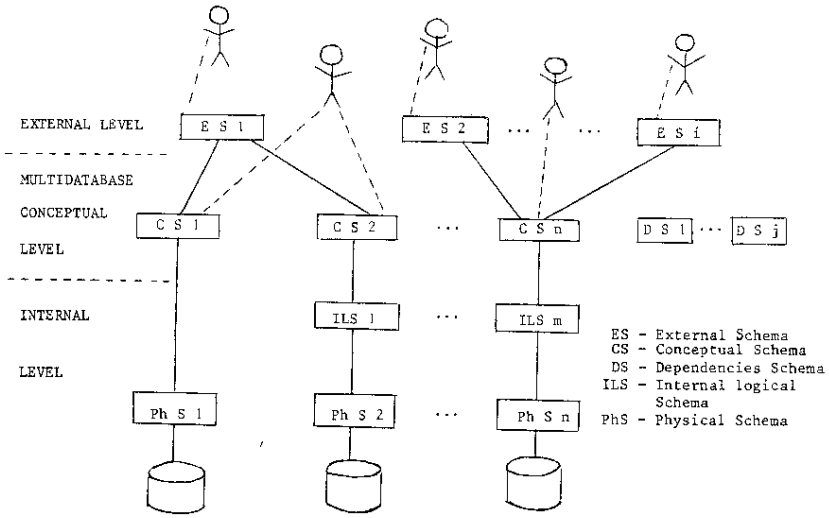


Figure 19 : General architecture of a multidatabase management system

The level above the conceptual one is called external level. At this level, users may define views of the MDB. An external schema may present an MDB as a single DB. Users of such a view see then only one DB. An external schema may also present an MDB as (to some extend different) MDB. Users of such a view see a collection of DBs. We assume of course that one may formulate not only multidatabase queries, but also multiview queries.

The level below the multidatabase level is called internal level. This level may be constituted from two sublevels :

- the optional internal logical level,
- the physical level.

The internal logical level corresponds to the mapping(s) of one or more DBs from the data model(s) within the multischema to some other data models. For instance, from the relational data model to the CODASYL data model [ZAN79]. The latter schemas may be the schemas of some preexisting DBs [MOU81].

The physical level corresponds to the physical implementation of the MDB. The corresponding schemas define the physical data structures of each DB and of the dependencies schema if it exists. They also define the physical distribution if an MDB encompasses several sites. The whole collection of schemas corresponding to the internal level is called internal multischema.

Presently, there is no MDBMS that provides all the possibilities that we define here and in what follows. Four prototype MDBMSs were implemented that provide the main possibility i. e. simple multidatabase queries [GUE81], [KAB81]. [MOU81], [MOU81a]. Two of these systems are relational MDBMSs, two others are bibliographic MDBMSs. An overview of three prototypes is given in the sec. 3.5.

All other presently known systems are not MDBMSs in our sense. The goal of these systems is to manage a single DB defined by a global or a conceptual schema. This is in particular the case of MULTIBASE system of [SMI81]. The same is true for MULTIBASE of CAP-SOGETI software house. Probably the closest to our idea of an MDBMS is R* DDBMS [LIN81]. It would become an MDBMS if one drops the requirement for the global schema, modifies the concept of the site autonomy and introduces to the architecture the concept of interdatabase dependencies.

3.4. MULTIDATABASE MANIPULATION LANGUAGE.

The commands of MALPHA language are designed on the one hand in order to be ALPHA compatible. On the other hand, in order to reach the goal of one command per intentional multidatabase query (goal 2), at least for the types of MDBs revealed by the case study. The commands are as follows :

1. CREATE, ADD, REMOVE DESTROY,
2. OPEN, CLOSE, GET, UPDATE, PUT, DELETE.

The commands (1) allow to create or to restructure an MDB. The commands (2) allow to formulate multidatabase queries. An extensive discussion of all the commands may be found in [LIT82]. Below, we will only show how OPEN, CLOSE, GET and UPDATE commands satisfy the case study, mainly through examples.

3.4.1. Multidatabase queries.

3.4.1.1. Multidatabase opening and closing.

Example 3.4.

The command :

OPEN NIGHTLIFE

opens all DBs that of this MDB no matter how many they are. Vice versa the command :

CLOSE NIGHTLIFE

closes all DBs of NIGHTLIFE. Multidatabase OPEN and CLOSE may thus avoid to formulate even very usual elementary commands.

One may also selectively open or close some DBs of an MDB. In particular, one may indicate in one command a list of DBs or MDBs to be opened or closed.

3.4.1.2. Multidatabase retrieval.

In order to allow multidatabase retrieval, GET command in MALPHA provides the following possibilities :
- one may refer to (multi)database names,
- these names may, in particular, prefix a relation name in order to render it unique, as in the query (1),
- a relation name that is not prefixed is considered as referring to all relations with this name in all DBs that are opened when GET is issued,
- the result of a GET may be, as in ALPHA, a relation called then (elementary) workspace. However, it may also be a set of relations, called workmultispace.

- RANGE declaration may be a semantic RANGE, that we explain further.
- one may use multidatabase standard functions that are a new type of standard functions /COD70/. For instance as the ones in ex. 8 or NAME function in [LIT82].

A GET that leads to a set of relations may be consider as a set of GETs called subqueries. Each subquery results from a substitution of prefixed names to unprefixed names or to variables. A subquery may be :

- irrelevant to DBs that names result from the substitution. Such a query either requires in the result an attribute or relation that is not in the DBs. Or, the query requires it in the predicate, let it be P, and P cannot be put into the disjunctive form :

P = P1 \/ P2

where P1 requires exclusively the names within the DBs. An irrelevant query does not create any result i. e. any workspace.

- relevant to these DBs. This may mean that the subquery is a relational query or a simple multidatabase query. The query may also require in P names that are not within the DBs, but P may then be put into the disjunctive form. P1 is then considered as the predicate of the subquery. Each relevant subquery creates one elementary workspace eventually empty.

Similar principles are valid for UPDATE, PUT and DELETE commands.

Example 3.5.

1. Retrieve in MICHELIN all chinese restaurants.

```
OPEN MICHELIN
GET W (R) : (R.TYPE = 'CHINESE')        (2)
CLOSE MICHELIN
```

R is an unprefixed relation name. However, since only one DB is opened, there is only one relation R. Therefore, there is no substitutions and the query leads to the creation of an elementary workspace. The result of this query is the one of the same ALPHA query. This is true any time one formulates in MALPHA an ALPHA query and one opens only one DB.

2. Retrieve all the restaurants that are chinese for MICHELIN or for GAULT-M.

```
OPEN REST-GUIDES
GET W (R) : (R.TYPE = 'CHINESE')
CLOSE REST-GUIDES
```

This time we nave :

R = (MICHELIN.R, GAULT-M.R)

Despite the same syntax, this GET acts therefore differently from (2). The substitutions lead to two relevant subqueries. GET retrieves then in one command two relations. It thus renders W a set of two elementary workspaces identified respectively as W.MICHELIN and W.GAULT-M.

In this example, one could express the intentional query using two ALPHA queries. MALPHA formulation is however easier in the sense of the goal 2. First, it leads to only one query with the same syntax. Next, this sole query suffices in fact no matter how many DBs refer to the restaurants named R, while ALPHA would necessitate as many queries as there are DBs.@@@

Example 3.6.

Retrieve from RESTAURANTS and from CINEMAS the names and the phone numbers of the restaurants and of the cinemas that are on CHAMPS ELYSEES street.

```
OPEN NIGHTLIFE
RANGE (R.RNAME, C.CNAME) X.Y

GET W (X.Y, X.TEL) : (X.STREET = 'CHAMPS ELYSEES')
```

The RANGE of this query is a semantic RANGE. The values of the variables X and Y are the constants in parentheses. These constants typically are the names of relations and of attributes to which refers an intentional multidatabase query. Our interpretation of unprefixed names allowed to attain the goal (2) when a multidatabase query refers to the same names in different DBs. Semantic RANGEs extend this possibility to different names.

Semantic RANGEs open the way to very many useful multidatabase queries. In the case of NIGHTLIFE one may, for instance, ask for :
- where to go for less than 100 FF,
- is BRETAGNE a cinema or a restaurant,
- etc.

In the case of REST-GUIDES one may, in particular, ask for :
- any restaurant that is chinese for a guide,
- all the appreciations of the quality and of the average price of the restaurant 'LASSERE',
- etc [LIT82].

Example 3.7.

Assume that a restaurant has only one telephone number. Retrieve (i) MICHELIN.RNAME, (ii) MICHELIN.AVPRICE, (iii) GAULT-M.AVPRICE expressed, however, in FF; for any restaurant recommended by MICHELIN and such that MICHELIN.AVPRICE is smaller than GAULT-M.AVPRICE expressed in FF.

```
OPEN REST-GUIDES
X <--\ GAULT-M.R.AVPRICE / 100
RANGE MICHELIN.R Y
RANGE GAULT-M.R Z

GET W (Y.RNAME, Y.AVPRICE, G-AVPRICE <-- X)  :
        ∄ Z (Y.TEL = Z.TEL /\ Y.AVPRICE < Z.X)
```

In this query X is, as we call it, a virtual attribute. The mapping means that the value of X is the one of GAULT-M.R.AVPRICE divided by 100, in order to nave all prices expressed in FF. The symbol '\', means that the values and the name to appear in the result are the ones of X. The name however may be changed further, in this case to G-AVPRICE. One could of course use only the name G-AVPRICE, but, since shorter, X is easier to manipulate. Vice versa, G-AVPRICE is better for the result naming.

Example 3.8.

Retrieve from REST-GUIDES all restaurants that are chinese according to at least one guide. Then, assume that to one value of (RNAME, STREET) corresponds one restaurant for any DB (we recall that the correspondance between the keys of a restaurant in different DBs is assumed to be unknown). Present the result of the query in a manner such that all data about one restaurant constitute only one tuple.

```
OPEN REST-GUIDES
RANGE (R, REST) X
RANGE (RNAME, NAME) (RNAME)
RANGE (TEL, T#) (TEL)
GET W NORM1((RNAME, STREET), X) : (X.TYPE = 'CHINESE').
```

The desired presentation is due to the usage of MALPHA standard function named NORM1. If NORM1 was not applied the result of this GET would be a set of relations where one restaurant would be described with as many tuples as there are guides that recommend it. The formal description of NORM1 may be found in [LIT82]. Shortly speaking, NORM1 produces one tuple from all tuples that have the same values of normalization key, (RNAME, STREET) in this query. If some attributes are functionally dependent on the normalization key and have the same name (even if this name results from a semantic RANGE only, as for TEL and RNAME in this query), then only one attribute is created in the tuple for all of these attributes. In occurrence, the result would be the set of relations as follows :

```
MICHELIN (R#, RNAME, STREET, TYPE, STARS, AVPRICE, TEL)
KLEBER (REST#, RNAME, STREET, TYPE, FORKS, T#, MEANPRICE, OWNER)
GAULT-M (R#, RNAME, STREET, QUAL, TEL, TYPE, AVPRICE, COMMENT)

MICHELIN-KLEBER (R#, RNAME, STREET, MICHELIN.TYPE, STARS, AVPRICE, TEL,
REST#, KLEBER.TYPE, FORKS, MEANPRICE, OWNER)
MICHELIN-GAULT-M (.... )
KLEBER-GAULT-M ( .... )

MICHELIN-KLEBER-GAULT-M (MICHELIN.R#, RNAME, STREET, MICHELIN.TYPE,
STARS, MICHELIN.AVPRICE, TEL, REST#, KLEBER.TYPE, FORKS, MEANPRICE,
OWNER, GAUULT-M.R#, QUAL, GAULT-M.TYPE, GAULT-M.AVPRICE, COMMENT)
```

The first three relations contain the restaurants known only to one of the DBs : MICHELIN, then KLEBER, then GAULT-M. Then three relations contain the restaurants known to a couple of DBs. Finally, the last relation contain the restaurants known to all the DBs. Note that the number of guides that recommend a restaurant is a meaningful information.

One may define a standard function similar to NORM1, let it be NORM2 that would work in the similar manner, except that it would produce one relation. Therefore, typically, this relation would contain null values. One may also think about NORM3 that would also work in the similar manner but would create the attributes corresponding to DB names etc.

3.4.1.3. Multidatabase update.

Example 3.9.

Update in MY-FAV-REST the number of stars of each restaurant that is '**' in MY-FAV-REST and '***' in MICHELIN and which key is not in MY-FAV-REST.TEST relation. (Michelin considers thus that these restaurants improved their quality and the user wishes to update in consequence its personal DB, without however changing his own judgment when it exists).

```
OPEN MICHELIN, MY-FAV-REST
RANGE MICHELIN.R X
RANGE MY-FAV-REST.R Y
RANGE TEST Z

HOLD W (Y.R#, Y.STARS) : (Y.R# = X.R# /\ Y.STARS = '**' /\ X.STARS =
'***') /\ ∄ Z (Z.R# = Y.R#))
W.STARS = '***'
UPDATE W
```


Other, easy to formulate in MALPHA and of obvious practical interest, multidatabase update queries are, for instance, the following queries :
- change in all DBs of REST-GUIDES the phone number 123 to 456 .
- do it for both REST-GUIDES and MY-FAV-REST.
- update STREET = ETOILE to CH-DE-GAULLE in all DBs of NIGHTLIFE, REST-GUIDES and MY-FAV-REST.
- etc [LIT82].

3.5. PROTOTYPES.

3.5.1. MRDSM system.

3.5.1.1. Overview.

MRDSM is a relational MDBMS that generalizes the commercial system MRDS, of HONEYWELL. The aim of MRDSM is the study of easiness of the generalization of a relational DBMS into an MDBMS. The first goal was to provide simple multidatabase queries to DBs that are on the same site. As ex. 3.1 showed, this possibility alone constitute a substantial benefit for users.

The implementation of such MDBMS revealed effectively very easy. It took few month of student work. Same low cost should be expected for the analogous generalization of other existing relational systems. This easiness was mainly due to the concept of work database that will be explained later.

3.5.1.2. Query language.

MRDS language is quite similar to SQL. MRDSM language is a generalization of this language that allows :
- to use multidatabase names and lists of database names in OPEN command,
- to formulate RANGE on database names,
- to prefix relation names with the corresponding variables in a tuple RANGE.

Example 3.10.

The query (1) would be formulated as follows :

```
mrc open nightlife;
-bd(x restaurant) (y cinema)
-range(r1 x.r) (c1 y.c)
-select r1.rname,c1.cname
-where r1.street=c1.street

CMDB

Rose de Tunis
St. Severin
```

3.5.1.3. Query processing.

The algorithm for query processing works as follows :

1. MRDSM determines the names of databases that should be opened. Then, it generates the corresponding OPEN commands of MRDS and calls MRDS.

2. If the query reveals an elementary one, i. e. the user nas formulated an MRDS query, then MRDSM calls MRDS that processes the query as usually.

3. Else MRDSM :
- determines the relations that has to be extracted from each DB, using some elementary (MRDS) queries.
- produces the MRDS schema of a DB that would be a set of these relations.
- asks MRDS to create this DB that, initially, is empty. The DB constitutes the workmultispace called work DB.
- asks MRDS to execute the elementary queries. Then, asks to reinsert each result (a workspace) into the corresponding relation in the work DB.
- asks MRDS to perform the queries to the work DB that complete the execution of the multidatabase query.
- typically, destroys the work DB and closes the other DBs.

The creation of work DB allows to perform by MRDS relational operations that should be otherwise reprogrammed. Such a choice would be a rather bad option, since, first, it would represent much greater programming effort. Next, it would probably be less performant. The reinsertion of tuples into the work DB is in fact not a waste of time, but it allows MRDS to create complex indexes, necessary in order to speed up relational operations such as joins.

3.5.2. MUQUAPOL system.

MUQUAPOL is built on the top of POLYPHEME DDBMS. It allows users to formulate multidatabase queries to DBs managed by the local machines and the global machine of POLYPHEME [ADI80]. The query language is a multidatabase extension of QUEL, called QUELM. Users of QUELM may refer to database names. They may also refer to a kind of predeclared semantic RANGE, called equivalence dependency /KAB81/. The dependency is defined by MDB administrator(s) and is stored in the dependencies schemas. MUQUAPOL translate QUELM queries into relational algebra POLYPHEME queries and, then, manages the query processing.

Example 3.11.

The query "retrieve from REST-GUIDES the names and the average prices of all the restaurants that are chinese for a guide", would be formulated as follows :

```
OPEN REST-GUIDES
RANGE OF X IS (EQUIV REST)

RETRIEVE (X.NAME, X.MEANPRICE)
  WHERE (X.TYPE = 'CHINESE')
```

KLEBER.REST

NAME	MEANPRICE
TONKIN	150
TOUR DE JADE	220

MICHELIN.R

RNAME	AVPRICE
TONKIN	170
SAIGON	100

GAULT-M.R

RNAME	AVPRICE
TOUR DE JADE	240
TONKIN	160
CANARD LAQUE	130

The selection of atribute names is due to the equivalency dependencies, declared for the attributes with the same meaning. All prices are supposed to be in FF. Note the differences between the estimations for a restaurant. Note also that a restaurant is not necessarily chinese for all the guides.

3.5.3. MESSIDOR system.
3.5.3.1. Overview.

MESSIDOR allows multidatabase retrieval from bibliographic databases. The databases may be on different sites, as they may be all on one site (ESA, QUESTEL,...). They may use different data manipulation languages (QUEST, MISTRAL,...). The MESIDOR user may however use a single language called MESSIDOR language. This language is very close to the Common Command Set Language /NEG77/, recommended by European Communities as the standard language.

Presently, hundreds of bibliographic databases are available on the EURONET, TRANSPAC and other network sites. However, the difficulties related to different access procedures, language heterogeneity and the work with several databases on one-by-one basis discourage most of potential users. Systems like MESSIDOR respond to a strong user demand and should overcome the present annoying state-of-the-art. Presently, no system similar to MESSIDOR is known.

MESSIDOR is implemented on MICRAL microcomputer. It is intended to be a personal multidatabase system for users needing various bibliographic data. A site considers the system as its usual terminal which means that no site software modification is required. MESSIDOR may thus potentially be used with any database server site.

3.5.3.2. Messidor language.

A user searches documents during a session. A session is an interactive sequence of MESSIDOR commands. The commands available for users are the following ones :

- BASE for the choice of databases for the session (multidatabase open).
- LIST for multidatabase display of dictionaries of search terms.
- SELECT for multidatabase search terms selection.
- FIND for multidatabase selection of documents using directly typed search terms.
- COMBINE for multidatabase combining of selected sets of documents.
- DISPLAY for multidatabase presentation of details of the selected documents.

- HISTORY for the summary of the previous commands and results of the current search session.
- LOCAL for monodatabase searching using a local language.
- GLOBAL for the return to the MESSIDOR language.
- HELP for explanations about MESSIDOR use.

Example 3.12.

```
BASE INSPEC, PASCAL
     ALL DATABASES ARE OPENED

STEP 1
FIND KW = INFORMATION RETRIEVAL * NATURAL LANGUAGES
     53 DOCUMENTS DISTRIBUTED AS FOLLOWS :
               PASCAL    32
               INSPEC    21

STEP 2
FIND KW = LINGUISTICS
     1534 DOCUMENTS DISTRIBUTED AS FOLLOWS :
               PASCAL 405
               INSPEC 1134

STEP 3
COMBINE 1/2
     45 DOCUMENTS DISTRIBUTED AS FOLLOWS :
               PASCAL 30
               INSPEC 15

STEP 4
DISPLAY STEP = 3
     (the documents are displayed and/or printed)

STEP 4
STOP
     (MESSIDOR closes the databases and disconnects the sites).
```

In this session the user works with databases INSPEC and PASCAL. '*' means 'and'. COMBINE command eliminates from the set resulting the step 1 all documents indexed with the search terms of the step 2.

More extensive discussion of MESSIDOR commands may be found in /MOU81/.

3.5.3.3. System architecture.

MESSIDOR system is built up from the following main components :
(A) - the coordinator that manages all other components, user inteface and higher level interface with the sites.
(B) - the translators to local languages.
(C) - the telecommunication driver .

The components (A) and (B) are written in BASIC. (C) is written in Z80 assembler.

3.6. CONCLUSIONS.

The database approach considers that data about a universe should constitute a single database (DB). The multidatabase approach considers that data may also constitute a collection of DBs. The general goal of the multidatabase approach is to provide tools allowing to manage collections of DBs. One of the main goals are languages allowing to formulate multidatabase queries. Another goal are tools allowing to define interdatabase dependencies.

The multidatabase approach generalizes the database approach. In particular, the concept of a multidatabase generalizes the one of a database. Also, the concept of an MDBMS generalizes the one of a (D)DBMS. The concepts that render MALPHA a multidatabase generalization of ALPHA, lead to analogous generalizations of QUEL, SQL or any relational language. The practical consequence is that one gains new possibilities, without loosing any of those that result, or may result, from the present principles.

The work on the prototypes showed that to implement an MDBMS using existing (D)DBMSs may be an easy task. In our opinion, this means that new systems for data management should henceforward be typically designed as MDBMSs. This design should be particularly useful when one has to manage a great number of DBs.

4. OVERALL CONCLUSIONS.

We have presented two approaches to the distributed data management problem. The SIRIUS-DELTA approach provides tools to constitute a single database. The multidatabase approach provides tools to constitute a multidatabase. The prototypes that have been developped within the project prove the feasibility of the corresponding industrial systems.

Even a glance on the present state-of-the-art shows that such systems will be highly needed. The rapid development of local networks will soon render necessary systems for the management of distributed enterprise data. The Videotex systems such as PRESTEL (UK), TELETEL (France), TELIDON (Canada) or COMPUSERV (USA) offer already access to thousands of databases. Bibliographic database servers like QUESTEL (France), or networks, like EURONET, provide the access to hundreds of heterogeneous bibliographic databases. Finally, the exponential grow of the park of microcomputers, will soon lead to a very large number of personal databases.

The research in the project revealed large domains where the techniques for distributed data management should apply. In particular, one may cite multiprocessor systems such as SABRE [GAR81]. Also, this is the case of real-time systems with strong resiliency constraints [LEL81]. Finally, the Open Systems [ISO82] will particularly need such techniques.

Among the research problems that are still open, one may cite the problem of tools for the efficient management of heterogeneous data. An effort is also required in the area of query decomposition. Finally, one should particularly investigate the techniques for the management of a very large number of databases.

Note.

The project SIRIUS was directed by Mr. J. Le Bihan, until 1982, then by W. Litwin. The coordination team also involved C. Esculier, G. Gardarin and G. Le Lann. The project is supported by Agence de l'Informatique (ADI).

REFERENCES

[ABC82] F. Abchine, M.R.BA project: generalities for external and conceptual models for relational DDB,
These 3 cycle, Toulouse,
February 1982, 218 (in french).

[ADI78] M. Adiba, A relational model and an architecture for distributed databases systems,
PhD thesis, Grenoble university,
September 1978 (in french).

[ADI78a] M. Adiba,J.C. Chupin,R. Demolombe, G. Gardarin, J. Le Bihan, Issues in distributed data base management systems : a technical overview,
Proceedings of 4th VLDB,
September 1978, 89-110.

[ADI80] M. Adiba, J.M. Andrade, P. Decitre,F. Fernandez,G.T. Nguyen POLYPHEME: an experience in distributed database system design and implementation,
international symp. on distributed data bases, C. Delobel and W. Litwin (eds), North-Holland,
March 1980, 67-98.

[AFC78] Groupe AFCET-TTI bases de donnees reparties, Caracterisation d un SGBDR,
Atelier du congres AFCET-TTI, Gif-sur-Yvette,
November 1978 (in french).

[AND80] E. Andre, P. Decitre, Project Polypheme: the DEM distributed execution monitor,
Proceedings of the international symposium on distributed data bases, Versailles, France,
May 1980.

[ANS75] ANSI, ANSI/X3/SPARC study group on data base management systems,
interim report, FDT, bulletin of ACM-SIGMOD, vol.7, no.2,
February 1975.

[BAE81] J.L Baer, G. Gardarin, C. Girault, C. Roucairol,The two-step commitment protocol, modeling, specification and proof methodologie,
5 th international conference on software engeneering,
March 1981.

[BDV80] B. Del Vecchio, P. Penny, The PHLOX project: three databases management systems for micro-computers,
Sigsmall symposium,
September 1980.

[BOG81] G. Bogo, J.C. Chupin, Groupe SCOT CII-HB, Conception des applications transactionnelles réparties,
Actes des journées de présentation des résultats du projet SIRIUS,
November 1981, 309-320 (in french).

[BOS78] P. BOSC, A. Chauffaut, Le systeme FRERES: contribution a la cooperation de bases existantes, interrogation de fichiers sur reseau de calculateurs heterogenes,
PhD thesis, university of Rennes,
1978 (in french).

[BOU81] J. Boudenant, P. Rolin, Modeling of the transaction distribution algorithm in Sirius-Delta,
Symposium on distributed software and database system,
July 1981, 169-176.

[BOU81a] J. Boudenant, Some solutions for distributed database recovery in SIRIUS-DELTA,
Distributed data sharing systems, North-holland publication,
June 1981, 55-65.

[CAL78] J.Y. Caleca, Projet POLYPHEME. L'expression et la décomposition de transactions dans un systeme de bases de données réparties,
PhD thesis, university of Grenoble,
1978 (in french).

[COD70] E.F Codd, A relational model of data for large shared data banks,
CACM, vol.13, no.6,
June 1970, 377-387.

[COD71] CODASYL, Data base task group of the CODASYL,
ACM report, New-York,
April 1971.

[COD79] E.F. Codd, Extending the database relational model to capture more meaning,
CACM, vol.4, no.4,
June 1979.

[COD82] E.F. Codd, Relational database: A practical foundation for productivity,
CACM, vol.25, no.2,
February 1982.

[DAN77] Ng. X. Dang, G. Sergeant, Expression of parallelism and communication in distributed network processing,
Proceedings of the international conference on parallel processing, Bellaire, Michigan,
1977.

[DEM80] R. Demolombe, Estimation of the number of tuples satisfying a query expressed on predicate calculus language,
Proceedings of the VLDB, Montreal
October 1980, 55-63.

[FER81] G. Ferran,Distributed checkpoint in a distributed management system,
Real-time systems symposium,
December 1981.

[FER81a] A. Ferrier,P. Penny, PHLOX2 a server database management system in a distributed environmemt,
Symposium on Reliability in distributed software and database systems,
July 1981.

[FOR77] H. Forsdick, R. Schantz, R. Thomas, Operating systems for computer networks,
2nd distributed processing workshop, Brown University, Rhode Islands,
1977.

[GAR81] G. Gardarin and al., An introduction to SABRE: a multi-micro processor database machine,
6th workshop on computer archictecture for non numerical processing,
June 1981.

[GRA78] J.N. Gray, Notes on database operatingsystems, in operating systems- an advanced course,
R. Bayer and al., Lecture notes in computer science,Springer Verlag 60,
1978, 394-481.

[GUE81] S. Guemara, MRDS: Management system for one relational multi database,
INRIA, SIRIUS report MOD-I-046,
September 1981, 53 (in french).

[HAM79] M. Hammer, D. McLeod, On database management system architecture,
Lab. for comp. science MIT, MIT/LCS/TM-141,
October 1979, 35.

[ISO82] ISO, Information processing system - open systems interconne xion,
ISO/TC97/SC16/N719,
February 1982.

[KAB81] K. Kabbaj, MUQUAPOL: a distributed multidatabases management system,
INRIA, SIRIUS report MOD-I-044,
November 1981, 50 (in french).

[LAC81] J. La Chimia, Cost evaluation of request in SIRIUS-DELTA,
SIRIUS report EVA-I-007,
October 1981.

[LEB80] J. Le Bihan, C. Esculier,G. Le Lann, W. Litwin, G. Gardarin, S. Sedillot, L. Treille, A frenon nationwide project on dist ributed data bases,
Proceedings 6th VLDB, Montreal,
October 1980, 75-85.

[LED81] J.M. Le Dizes, Le projet SOPHIA : une base de donnees repar ties d horaires de transport,
Actes des journées de présentation des résultats du projet SIRIUS,
November 1981, 257-262 (in french).

[LEL77] G. Le Lann, Introduction à l'analyse des systemes multiréfé rentiels,
Thèse doctorat d'état, Université de Rennes,
May 1977 (french).

[LEL80] G. Le Lann, A distributed system for real-time transaction processing,
IEEE computer, vol.14, no.2,
February 81, 43-48.

[LEL81] G. Le Lann, SCORE : real-time distributed computing systems for automated applications,
INRIA report GAL-I-001,
November 1981 (in french) .

[LIN81] B. Lindsay, Object naming and catalog management for distri buted database manager,
2nd int. conf. on distributed computing systems,
April 1981, 31-40.

[LIT80] W. Litwin, A model for a distributed data base,
2nd ACM comp. exp. university of Laffayette,
February 1980.

[LIT81] W. Litwin, Logical model of a distributed data base,
2nd int. sem. on distributed sharing systems.R.P Van de Riet W. Litwin (eds.), North-Holland,
June 1981, 173-207.

[LIT82] W. Litwin, K. Kabbaj, Multidatabase approach to data manage ment,
INRIA research report SIRIUS MOD-I-050,
March 1982, 34.

[MOT81] A. Motro, P. Buneman, Constructing superviews,
Proceedings SIGMOD, Ann Arbor,
May 1981, 56-64.

[MOU81] C. Moulinoux, C. Faure, W. Litwin, MESSIDOR system,
Symposium on small systems, ACM SIGSMALL,
October 1981, 130-135.

[MOU81a] B. Moussaoui, A. Saint Upery, Cooperation between homgenous doctumentary system in network environment,
Proceedings of IDT France,
May 1981, 53-62.

[NEG77] A.E. Negus, Standard commands for retrieval system,
Interim report of European Communities,
May 1977.

[NEU77] E.J. Neuhold, H. Biller,, POREL. A distributed data base on inhomogeneous computer network,
Proceedings of 3th VLDB,
1977.

[NGU79] Nguyen Gia Toan, A dynamic distribution algorithm for rela tional query interpretation over cooperating data bases,
USMG report,
Grenoble 1979.

[POL79] Groupe POLIPHEME, Final report of POLYPHEME project,
SIRIUS internal report, November 1979.

[ROB77] P. Robert, J.P. Verjus, Towards autonomous description of synchronization modules,
Proceedings of the IFIP conference, 1977.

[ROS78] D.J. Rosenkrantz & al., System level concurrency control for distributed database systems,
ACM Tods, vol.3, no.2,
June 1978, 178-198.

[ROS82] J. Rosselin, 2000 available data banks: which to choose,
Temps Reel, no 30,
February 1982 (in french).

[ROT78] J.B. Rothnie, Distributed database management,
Proceedings of the 4th VLDB,
September 1978.

[SDD78] Computer Corporetion of America, SDD-1 technical reports,
Cambridge,
1978.

[SER80] G. Sergeant, L. Treille, SER: a system for distributed execu tion based on decentralized control techniques,
Proceedings of the ICCC, Atlanta,
1980.

[SMI75] J. M. Smith , P. Y. Chang, Optimazing the performance of a relational algebra database interface,
Communication of the ACM, vol.18, no.10,
October 1975, 568-579.

[SMI81] J. M. Smith, P.A. Bernstein, U. Dayal, N. Goodman, T. Lander sK.W.T Lin, E. Wong, Multibase integrating heterogeneous dis tributed database systems,
Proceedings AFIPS Chicago,
May 1981, 487-499.

[SPA78] S. Spaccapietra, Problematical conception of distributed da tabases management systems,
PhD thesis, Paris VI university,
November 1978 (in french).

[STO77] M.A Stonebraker, Distributed database version of INGRES,
2nd Berkley worshop on DDM and comp.networks,
May 1977.

[TAN81] K. Taraka, Y. Kambayashi, Databases into a distributed data base,
Proceedings of 7th VLDB,
September 1981, 131-143.

[THO80] R. Thomas, Process structure alternatives: towards a distri buted INGRES,
Int. symp. on distributed data bases,
C. Delobel and W. Litwin (eds.), North-Holland,
March 1980, 215-227.

[TOT78] K.C. Toth, S.A. Mahmoud, J.S. Riordon, O. Sherif,
The ADD system: an architecture for distributed databases,
Proceedings of 4th VLDB,
September 1978, 462-471.

[TSI77] D.Tsichritzis, A. Klug, The ANSI/SPARC DBMS framework-report of the study group on data base management systems, Technical note 12, CSRG, university of Toronto, July 1977.

[YAO79] S. B. Yao, Optimization of query evaluation algorithms, ACM Tods, vol.4, no.2, June 1979, 133-155.

[ZAN79] C. Zaniolo, Multimodel external schemas for Codasyl database management systems, IFIP TC-2 workshop conf. on data base architecture, June 1979, 157-176.

[ZUR81] G. Zurfluh, Le projet PLEXUS: présentation de l'application et des objectifs, Actes des journées de présentation des résultats du projet SIRIUS, November 1981, 241-256 (in french).