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.

NORTH-HOLLAND PUBLISHING COMPANY
AMSTERDAM • NEW YORK • OXFORD
DISTRIBUTED DATABASES
H.J. Schreiber (Editor)
North-Holland Publishing Company
© GI, 1982

SIRIUS Systems
for
Distributed Data Management

W. Litwin, J. Bouillon, C. Basiaux, A. Perrier, A. M. Girault,
J. La Cunza, X. Kasbaj, C. Moulinoux, F. Rolin, C. Stangret

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

Distributed data management becomes an increasingly important field of interest. SIRIUS project investigated several approaches to this goal. In this paper, we present one of them, the approach developed 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 multible-based 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 those 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 prototypes 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 [LEBE]. 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 POLYMER (CII - EMIF), PEGASE (IRISA), SISMOR (USST) and ETOILE (NANCY-1) were in, particular, presented during the International Symposium on Distributed Database (Paris, March 1980, proc. North Holland). The prototypes MICROBE (INRIA) and TPF-R (NANCY-1) were presented during conferences SIRIUS (Paris, November 1981, proc. Agence de l'Informatique).

In response to the objective (3), we succeeded in creating 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 extent, in particular the projects SODI [2346], SOPHIE [2346] and PLEXUS [2346]. Most of French computer manufacturers or software houses were in this way involved in SIRIUS project.

311
Starting from 1973, 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 DBMS for enterprise DBAs. Another preindustrial prototype called MESSIDOR [NO81] allows to manipulate sets of heterogeneous bibliographic DBs, distributed over public networks (EUMUNET, TRANS PAC,...). Some other, yet less complete, prototypes were also developed: FLDDB [BO80], MIAPOLE [K88] and MIRIO [J88].

These prototypes were based on two different perceptions of distributed data management problems: the first approach assumes that distributed data constitutes a single database called distributed database (DB). A DB is defined by a single conceptual schema, called global schema. Users work with the DB as they would work with a centralised 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 multibase database. Users of a multibase database 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 multibase. The entire multibase may, in particular, be on only one site.

SIRIUS-DELTA prototype investigated the first approach. Particular attention was given to DBA 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 DB architecture and system architecture. Afterwards, we detail the functional layers. Finally we indicate some results of performance analysis.

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

The presentation of SIRIUS-DELTA prototype constitutes the section 2. Section 3 discusses the multibase 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 DBMS 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 DBMS must be able to handle requests from pre-existing query languages on the submitting sites.

This is achieved using first schemas associated to the DB and related local DB's, next a common internal data manipulation language: "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 DBMS [S875]. This is necessary if one wants to provide some DB design and processing facilities (local and parallel processing optimization, usage of pre-existing local data for example).

A data distribution description language has been developed to describe the necessary mappings between DB 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 DBMS 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 BEGIN 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 commit.
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 [BOGD78], [FOLT79]:
- 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-language. 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 [AD76], [ASG78], [SI75]:
- 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 independency (local processing must be allowed).

This result in a proposition to extend the ANSI/SPARC schema architecture [ANSI75], [TS77] in order to introduce a global level and a local level. At global level the DBS is considered as a database the physical characteristic of which is data distribution. Thus the DBS is described using the three-level architecture.

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

---

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 DBMS.

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 lock heterogeneous.

Thus a local external view is provided to the DDMS so that heterogeneously maintained local data are homogeneously presented and handled at DDB level.

The local external view specific to the DDMS 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 DDMS:
- since no schemas are available, the related local external view functionality must be implemented, in order to provide the DDMS an homogeneous presentation of local data,
- since no data manipulation functions are available, they must be developed using, for example, generalized catalogued procedures.

2.2.2. System architecture:

In SIRIUS-Delta DDMS 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 adaptive routing to network topological changes.

Four basic functional layers are defined above the transport layer, that provides:
- database classical data management functions ("DBMS" layer)
- distributed data handling functions (SILCE layer)
- distributed concurrency control functions (SCCMS layer)
- distributed execution functions (SBR layer)

This architecture is depicted in fig.2.
Many studies about this function are being lead in France, and abroad [1557], [1558], in this section, we present the SIRIUS-Delta distributed execution system (named SDE) [1559], a synthesis of the works conducted within SIRIUS on the subject [1560], [1561].

2.3.1. Objectives:

At present time, most of distributed applications can be seen as a set of usually 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.

SDE 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 2 of OSI Open System Architecture) [1562]. We define hereafter a "processor" as the physical entity upon which an occurrence of SDE can be run. In most cases a site can be considered identical to a processor.

- on every physical processor, a catalogue of independent 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, SDE 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 expressions and computing of program activating conditions. Those conditions can be related to the results of others programs,
- as far as possible, use of 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 SIRIUS. However, SDE 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 in the main input to SDE; 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. SDE performance 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 a logical execution unit of the DBF. From a physical view, it is the execution of a program on a processor. We define a local action within a DEP as an existing program running upon a dedicated processor, as well as the input and output data files and synchronization conditions within the DEP.

A local action 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 program are updating of synchronization variables and the consumption or production of temporary data files.

4.5.4.2. Synchronization variables

These variables are associated with local actions; they ensure the distributed execution control as well as the 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 ensures the control of the distributed execution plan.

2.3.2.3. Temporary data files

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 these 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 as a single 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 "local").

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:

```
Distributed Execution Plan

Global SER

execution report

execution or abort commands

Synchronization variable

updating

local SER

data transfer (temporary data file)

local action

launching

execution end or abort

local SER
```

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 SER 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 SER name within the system
  - a unique local action name within the SER,
- 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: assignment or comparison (true or false for booleans; =, >, <, <=, >=, # for integers; =, # 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 positioning 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 to 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 superimposed command received 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 destroys 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 Réparti) 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 "transformed" transaction is actually implemented as an atomic operation.

2.4.1. General principles:

The following model permits to describe SCORE functions. The database is visualized consisting of a number of logical objects to which access can be gained individually.
Global SLOE produces access requests to objects (i.e., access requests to plots) intended to be processed by local SLOEs. Global SLOE transmits to global SCHE these requests which are inspected by concerned local SCORES before being entrusted to local SLOEs.

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 abdication, 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, SCHE includes the following three mechanisms:
- a decentralized synchronization mechanism, called Circulating Sequencer (LELF77). This mechanism is used to give a unique system name to a transaction;
- two-phase locking protocols associated with a deadlock prevention strategy [SHOR81]. 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.

<table>
<thead>
<tr>
<th>Requested Object</th>
<th>Locked</th>
<th>No Lock</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>No Lock</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

The deadlock prevention is upon conflict detection which are solved in accordance with the transaction tickets [MOST81].

- a two-phase commit protocol [GRAY81], [GAR81].

2.4.2. Failures of system components

This has been extensively described in [NOUB81a].

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 [LELF80].

In SIRIUS-DELLA 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 SCORES must guarantee that the same thing will be done for all of the involved consumers. The SCORES-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 does not imply any overhead of course the "before image file" must be preserved. 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 producer may behave as memory-less processes. We have to prepare 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 simple local database recovery.

a) Recover all the database using the "before image" files and release the objects,
b) For each committing transaction we find in the journal the FTC item the SQLEX-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 sites can be inserted in the DEBS. The communication level sends to SQLEX-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 a specially 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 DB consistency. It is a catastrophic case where the whole DB 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 responsibility 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 (ITI)

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 looks more convenient to write global checkpoints on journals.

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

- ITI meaning in the global system -

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

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, SEU, and local SILOE,
- the adaptation/translation necessary to interface different local DBMSes, using distinct query languages and producing results in distinct formats,
- the local functions associated to SCORE local resource 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 DB).

In both cases we suppose that the DDS 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 DB can be very complex. Localization of a distribution unit can be defined 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 DB 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 partitioning of R)

All tuples in R are split into sub-relations. 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.

  - DUE ON local-database-3
  - DUE ON 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.

  - DUE ON local-database-1 WHEN att1 = 1, att2 = 2, ..., att9 = 3
  - DUE ON local-database-2 WHEN att1 = 10, att2 = 11 ...

- "WIA" 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 as another relation (which contains the value of the primary key attribute(s)).

  - DUE2 VIA att3 FROM DUE1

  (DUE2 has its own localization rule, which can be non-dependent, direct or indirect).

### 2.5.1.4. SILOK data distribution definition language (DDDL)

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

In order to express very complex distributions, we allow the definition of several localization rules for a given DB. We call Homogenous Distributed Set (HDS) the data set which corresponds to one DB 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 subdivided.

Then, plots associated to each HDS is described in the localization section. Plot partitioning can only be 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 (NR, NS, NS-ROOMS) KEY = NR

GIS: * HDS SECTION

RESORT (NR, ALTITUDE, STATION) KEY = NR
RESORT2 (NR, ALTITUDE) KEY = NR
RESORT3 (NR, STATION) KEY = NR
HOTEL1 (NR, NS, NS-ROOMS) KEY = NR
HOTEL2 (NR, NS, NS-ROOMS) KEY = NR

* MAPPING SECTION

RESORT1 ON LOCAL-DB-1 WHEN ALTITUDE > 800
RESORT2 ON LOCAL-DB-2 WHEN ALTIMETRY < 800
HOTEL1 ON LOCAL-DB-3 ELSE
HOTEL2 ON LOCAL-DB-4 ELSE

* LOCALIZATION SECTION

RESORT1 ON LOCAL-DB-1 WHEN STATION = "Lyon"
RESORT2 ON LOCAL-DB-2 WHEN STATION = "Nice"
HOTEL1 VIA NR FROM RESORT1
HOTEL2 VIA NR FROM RESORT2
```

---

**2.5.2. Global SILOK 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 IT's 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 SILOK-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 partitioning 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).

\[
\begin{array}{ccc}
  & P1 & P2 \\
R & & \\
  P3 & P4 \\
\end{array}
\]

Figure 4

\[
\begin{array}{cccc}
  & P1 & P2 & P3 & P4 \\
  & P1 & P2 & P3 & P4 \\
  P1 & P2 & P3 & P4 \\
  \text{Union} & \text{Union} & \text{Join} & \text{Join} \\
  \text{Join} & \text{Join} & \text{Union} & \\
\end{array}
\]

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.

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 \[DEM80\], \[SLM75\], \[YAO79\]. The algorithm is described in detail in \[LAC81\].

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 chosen. 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 operands 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 operand. The operand 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 operand 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](image)

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

![Figure 9](image)

Local actions exchange values associated to synchronized variables and temporary data files in a producer/consumer mode. These specifications are stored in the related local action context in a symmetrical 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 launch 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 DB is 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 DBL, programs making possible to execute these missing operators are added in local SILOE.

Local SILOE manages the scheduling of DBL 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 DBL 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).
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 known.

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 costs. For each list of required sites only the least expensive scenario is kept.

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

2.5.4.2. DBS design considerations:

Some DBS design thoughts have been involved in our approach to data distribution:

- Although it would have been functionally possible, no attempt has been made 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 DBS.

- Mappings between global and local data types may become very complex as soon as the DBS is built from a cooperation of pre-existing DBs. The DBS 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 DBS 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 GIS 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 SILEX is exporting a set of operators used by the pivot-language. Therefore if the local DBMS or DM does not contain these features, they must be added.

2.5.5. SILEX status:

At present time SILEX is providing some DB design facilities such as the DBMS, 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 authentication 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 [0V80]. Each package is intended for a particular use:

- PHLOX for individual micro-computers (mono-user, mono-server)
- PHLOX for a dedicated server in a local network (multi-user, mono-server) [FRA81a]
- PHLOX distributed data bases management system (multi-user, multi-server).

In administration, PHLOX offers the possibility of describing a new database at three completely separated levels which are those of ANGST-SPARC [ANG86]: an external level and a conceptual level using the relational model [CUD70] and an internal level based on the DESY's network model [COB87].

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, recoverability, deadlocks,...
2.6.2. PHILOX 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 DBMS) DBMS with SIRIUS-DELTA is being proved by the realization of an heterogeneous prototype using PHILOX as a server.

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

![Diagram of heterogeneous prototype of SIRIUS-DELTA with a PHILOX server]

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

The PHILOX 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 sends results to one (or more) SIRIUS-DELTA site(s) which can exploit them or present them as the result of the global request. PHILOX receives and processes requests of consultation as well as of modification.

In order to cooperate with SIRIUS-DELTA, PHILOX has to participate in the distributed protocols of SIRIUS-DELTA (i.e. DCP 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 PHILOX2 prototype and others have been added since, as shown in the next section.

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

The protocols designed by SIRIUS-DELTA have been implemented in PHILOX using as far as possible the existing software of PHILOX2. Only some functions have been added through a new level called SIRIUS-DELTA interface. The different levels appear in figure 12 below.

![Diagram of the three levels of a PHILOX server]

Figure 12: The three levels of a PHILOX server

2.6.3.1. SIRIUS-DELTA interface

This interface manages the messages which are exchanged between the PHILOX site and the SIRIUS-DELTA site(s). It analyses the received messages and 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 SOR (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 requests into PHILOX 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 DBMS, into the internal network schema used by PHILOX 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 interpreter and relational operators.

2.6.3.2.1. Pivot-language interpreter

The pivot-language has been designed by the SILO6 team of SIRIUS-DELTA and the PHILOX group. Thanks to this common language, requests can be sent from the global sites producing the transactions to the local sites which performs 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 interpreter manages the execution and the chaining of the relational operators. It also realizes 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 PHILOX.
The "Pivot-language Interpreter" 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 algorithms to apply in order to use at their best the access paths existing in the network schema of the database.

- 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 is 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 database. The inversion index are used to realize the selection in such a way that the system accesses only a limited amount of data. The father-son's links of the network schema correspond to already implemented joins of the use of which makes possible the decreasing of the reading number.

2.6.3.3. Existing functions in PHLOX

The part of the distributed protocol already existing in PHLOX consists mainly in the consistency and resiliency functions. The mechanisms which ensure data sharing and reliability are integrated into the inner part of PHLOX, 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 cannot wholly be known 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 behavior.

- 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.

- Commitement:
  PHLOX manages a two step commitment [BABA91]: 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 commitement, 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 STM interface (see above). At this level, PHLOX 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 raises 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 immediately point out that it is not possible in a prototype to measure with a real application since there is no 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 estimation ratio, 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 elapsed 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 6.8% 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 machines (about 40 μs per instruction). A more efficient machine would reduce greatly this overhead.

2.7.2. Network cost:

The novelties in a distributed system in 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 divided in 3700 instructions at the emission and 3900 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 SIE:

The different functions of the distributed executive system are: remote activation of a local action, synchronization of 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 which 15 200 are the result of messages emission and reception cost involved by the protocol. This represents 40% 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 * 5000 + 32 000 instructions.

Notice the important cost for program initialization.

The site receiving the SV activation has to get 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 curve is given on fig. 14.

![Figure 13: Elapsed time in the local action installation and execution process (the local action is empty)](image)

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 items, 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.
Looking object in the measure includes the relaying 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-lock 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.

2.7.3.4. 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 commit protocol cost is made of:
- a cost at the global site for n involved sites is: n * 12 000 instructions.
- 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 on the number of data objects accessed.

![Diagram showing item transfer cost for three different blocking factors](image)

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

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 DEP 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 interpreter was about 100 000 instructions and seems not depend greatly of the sub-query complexity. This means that cost 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 1500 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, correspondent
End Transaction

We describe below the transaction graph and give a list of required resources (fig. 16).
Using all these informations we are able to compute the different costs induced by the SIR, SCORE and SILOR 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 database, more than 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 than 60% 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 too long to execute on a single site. System cost does not override 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.

2.8. CONCLUSION

The prototype SIRUS-DELTA, initiated in 1979, is now operational. It is a pre-industrial prototype the aim of which was to validate research studies within the SIRUS project. Focus, in SIRUS-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 SIRUS-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 (Codd, 1971, 1972) 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 those 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 [Spa81], integrated or federated [Lan97] (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 system that one assumes therefore always definable if 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 [LLF89] (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 guide 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 sense 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)DBs, 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, CODI etc) with respect to older navigational languages (CODAS).

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 [LLF90] and [AB92].

Since the goal of the multidatabase approach is the management of several DBs, it may not a priori include the subgoal that is the management of one DB. This our approach is a generalization of the database approach.

3.1.3. The concept of a multidatabase

We call multidatabase (MBD), 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 MBD and, eventually, its structure and the dependencies between DBs or MBDs (multidatabase definition language). The dependencies may be semantic dependencies or integrity dependencies or privacy dependencies etc [LLF90].

Figure 18 : Database approach (a), versus multidatabase approach (b), to the reality modelling.
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 DB. One may then define the concept of an MDB recursively as "a DB that is the set of DBs [MDB1, [MDB2]. One may then also consider that an MDB is a logically distributed DB [MDB1].

An MDB may be physically centralized. It may also be physically distributed. DBs of an MDB may in particular be DBs that mean that the schemas of these DBs may be global schemas. The general properties of MDBs are extensively analyzed in [MDB1].

A relational MDB is an MDB that is a set of relational DBs or a set of relational MDBs [MDB1]. 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 describe some relational MDBs. They have been chosen in order to demonstrate typical user needs. MDBs are defined and manipulated according to MALPHA language [MDB2]. As its name suggests, MALPHA is a (multidatabase) a generalization of ALPHA language of Code [ALPHA].

Example 3.1.

Let us consider the following DBs:

<table>
<thead>
<tr>
<th>DB RESTAURANTS</th>
<th>DB CINEMAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>N ( NANE, STREET, TYPE, IEL)</td>
<td>C ( CF,CNAME, NCAL)</td>
</tr>
<tr>
<td>M ( MP,CQ, PRICE)</td>
<td>H ( MF, CN, KIND)</td>
</tr>
</tbody>
</table>

By definition, Database RESTAURANTS: Restaurant names, addresses, types, and IEL codes. Database CINEMAS: Cinema names, addresses, and kind of films.

Example 3.2.

Let us consider the following DBs:

<table>
<thead>
<tr>
<th>DB METRO</th>
</tr>
</thead>
<tbody>
<tr>
<td>N ( MF, NNAME)</td>
</tr>
<tr>
<td>S ( SF, SNANE, HOPEN, HCLOSE)</td>
</tr>
<tr>
<td>C ( SF, STREET)</td>
</tr>
<tr>
<td>M ( MF, SF)</td>
</tr>
<tr>
<td>P ( SF, SY, FFC, PRICE)</td>
</tr>
</tbody>
</table>

Database METRO: Metro lines, stations, streets near a station, stations of a line, and station-to-station prices.

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, INGRES, MBD or any other. They could also be on different computers or sites. Moreover, they could be distributed DBs i.e. the above schema would be the global schemas for ZDB-1 or SIRIUS-DBLA or R or any other DBMS. 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 thus an MDB.

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

On the other hand, the property (1) would allow users to formulate a new type of queries like the following one:

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

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

The above query has the syntax that is the one of MALPHA 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 MALPHA queries. On the other hand, for any possible MALPHA query there is a corresponding simple multidatabase query. This means that simple queries alone offer all the power of MALPHA 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 multidatabase queries to NIGHTLIFE.

Example 3.3.

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 extent differently: they do not recommend all the same restaurants, the attribute differ, same meaning may lead to different attribute names 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:
In particular, the following differences characterize these DBs:

1. The quality of a restaurant is measured differently. MICHELIN gives a
   restaurant up to three stars (***) and 0.5 stars to four "forks". GAULT-M
   apprication is 0.5 to 4 stars. There is no exact translation rule between
   these measures. Also, the grades may strongly disagree on the quality of a
   restaurant.

2. The grades typically agree also with respect to the value of the average
   price for a meal and with respect to the number kcal of calories in a
   meal. Furthermore, for the purpose of 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" (OF), where FF = OF/100.

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

If these DBs are managed each by a (DD)MS, 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 3 stars
  (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 (DD)MSs or lead to several
elementary queries. In addition, the elementary queries must be formulated differently,
besides the same goal. The result would be that, in practice, users would likely never
try to use several DBs, as the experience shows already [MDSI], [MBM]

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

MICHELIN was composed from DBs that were models of distinct universes.
BEST-GUIDES is composed from DBs that model differently the same universe. MBSs
like BEST-GUIDES will be called semantically heterogeneous MBS. The example shows
that one will frequently need to deal with such MBSs.

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, no 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-FAN-REST. The schema of MY-FAN-REST may be as follows:

```
DB MY-FAN-REST
R ( R#, NAME, STREET, TYPE, STARS, APPRICE, TEL, COMENT)
C ( CR, NAME, NCAL)
M ( R#, CR, AP)
ENDS
```

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

As he wishes, the user may now add further attributes according to his judgment, add
or withdraw restaurants, perform his own control of a restaurant quality etc. He
needs multidatabase queries, first, in order to easily create the DB (see ex. 3.7
of PUT command in [LIT82]). Next, after some time, he will need to add to
MY-FAN-REST the restaurants that become rated *** or more in MICHELIN (ex. 3.9).
Also he may wish to know the restaurants upon quality or price of which
MY-FAN-REST and MICHELIN disagree etc. Clearly, first, an MBS is needed also
in this case. Next, MBSs 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 update of MY-FAN-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
MBS. In particular, any MDBMS must allow to formulate simple multidatabase queries.
Fig. 10 shows the general architecture that we propose for an MDBMS. We called it
three-four level architecture [LIT81].

The MBS is defined at the level that we call multidatabase (conceptual) level. The
definition of the MBS is called (conceptual) multischema. A multischema is a
collection of the following schemes:
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].
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 them only one DB. An external schema may also present an MDB as (to some extend different) MDBs. 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 multischemas to some other data models. For instance, from the relational data model to the CODASYL data model [ZAN73]. The latter schemas may be the schemas of some preexisting DBs [MGB].

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 multidatabases.

Presently, there is no MDDMS that provides all the possibilities that we define here and in what follows. Four prototype MDDMSs were implemented that provide the main possibility: i.e., simple multidatabase queries [GUB85], [KAB81], [MGB], [M3BS]. Two of these systems are relational MDDMSs, two others are bibliographic MDDMSs. An overview of these prototypes is given in the sec. 3.5.

All other presently known systems are not MDDMSs 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 [GUB85]. The same is true for MULTIBASE of CA-SEEGGI software house. Probably the closest to our idea of an MDDMS is the DDBS [LIG01]. It would become an MDDMS 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 [LIG02]. Below we will only show how OPEN, CLOSE, GET and UPDATE commands satisfy the case study, mainly through example.

3.4.1. Multidatabase queries.
3.4.1.1. Multidatabase opening and closing.

Example 3.4.

The command:

```
OPEN NIGNLIFE
```

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

```
CLOSE NIGNLIFE
```

closes all DBs of NIGNLIFE. 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 /DO070/. For instance as the ones in ex. 8 or NAME function in [LIT82].

A GET that leads to a set of relations may be considered 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. Each 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:

\[
P = P_1 \lor P_2
\]

where \( P_1 \) 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. \( P_1 \) 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 5.6.

1. Retrieve in MICHELIN all chinese restaurants.

\[
\text{OPEN MICHELIN}
\]

\[
\text{GET w (R) : (R.TYPE = 'CHINESE')} \quad (2)
\]

\[
\text{CLOSE MICHELIN}
\]

\( R \) 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 \( \text{ALPHA} \) query. This is true any time one formulates in \( \text{ALPHA} \) an \( \text{ALPHA} \) query and one opens only one DB.

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

\[
\text{OPEN REST-GUIDES}
\]

\[
\text{GET w (R) : (R.TYPE = 'CHINESE')} \quad (2)
\]

\[
\text{CLOSE REST-GUIDES}
\]

This time we have:

\[
R = (\text{MICHELIN.R}, \text{GAULT-M.R})
\]

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

In this example, one could express the intentional query using two \( \text{ALPHA} \) queries. \( \text{ALPHA} \) formulation is however easier in the sense of the goal. 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 \( \text{ALPHA} \) would necessitate as many queries as there are DBs.

Example 5.7.

Assume that a restaurant has only one telephone number. Retrieve (1) MICHELIN.NAME, (2) MICHELIN.AVPRICE, (3) 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.

\[
\text{OPEN REST-GUIDES}
\]

\[
X \leftarrow \text{GAULT-M.R.AVPRICE} / 100
\]

\[
\text{RANGE MICHELIN.R Y}
\]

\[
\text{RANGE GAULT-M.R Z}
\]

\[
\text{GET w (Y.NAME, Y.AVPRICE, G.AVPRICE <= X)} : \quad \# 2 \quad (Y.TEL = X.TEL \land Y.AVPRICE < X.AVPRICE)
\]

In this query \( X \) is, as we call it, a virtual attribute. The mapping means that the value of \( X \) is the one of \( \text{GAULT-M.R.AVPRICE} \) divided by 100, in order to have 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 \( \text{G.AVPRICE} \). One could of course use only the name \( \text{G.AVPRICE} \), but, since shorter, \( X \) is easier to manipulate. Vice versa, \( \text{G.AVPRICE} \) is better for the result naming.

Example 5.8.

Retrieve from REST-GUIDES all restaurants that are chinese according to at least one guide. Then, assume that to one value of \( \text{R.NAME, STREET} \) corresponds one restaurant for any DB (we recall that the correspondence 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 (#, REST) X
RANGE (NAME, NAME) (NAME)
RANGE (Tليف, Tليف) (Tليف)
GET M = NORM(((NAME, STREET), X) : (X.TYPE = 'CHINESE').

The desired presentation is due to the usage of MALPHA standard function named NORM. If NORM was not applied to this GET it 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 NORM may be found in [LIT82].

Briefly speaking, NORM produces one tuple from all tuples that have the same values of normalization key, (NAME, 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 RAMUS only, as for Tليف and NAME 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:

MICHHELIN (#, NAME, STREET, TYPE, STARS, AVPRICE, Tليف) KLEBER (REST#) ANAME, STREET, TYPE, FORKS, Tليف, MEANPRICE, OWNER) GAULT-M (#, NAME, STREET, QUAL, Tليف, TYPE, AVPRICE, COMMENT) MICHHELIN-KLEBER (#, RAMLE, STREET, MICHHELIN.TYPE, STARS, AVPRICE, Tليف, REST#, KLEBER.TYPE, FORKS, MEANPRICE, OWNER) MICHHELIN-GAULT-M (#, NAME, STREET, QUAL, Tليف, TYPE, GAULT-M.TYPE, GAULT-M.AVPRICE, COMMENT)

The three first relations contain the restaurants known only to one of the DBs: MICHHELIN, then KLEBER, then GAULT-M. These three relations contain the restaurants known to a couple of DBs. Finally, the last relation contains 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 NORM, let it be NORM 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 NORM 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 MICHHELIN and which key is not in MY-FAV-REST 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 MICHHELIN, MY-FAV-REST
RANGE MICHHELIN.r X
MY-FAV-REST.r y

RANGE TEST Z
MULD W (Y,W, STARS) = (Y.W = $W$ ∧ Y.STARS = ***) ∧ / / X.STARS = ****) ∧ / / Z (G.W = Y.W))
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 = STOOL to CM-DH-GAULE in all DBs of NIGHTLIFE, REST-GUIDES and MY-FAV-REST;
- etc [LIT82].

3.5. PROTOTYPES.
3.5.1. MDRS system.
3.5.1.1. Overview.

MDRS is a relational MDBS that generalizes the commercial system MDS, of HONEYWELL. The aim of MDRS is the study of easiness of the generalization of a relational DBS into an MRS. 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 constitutes a substantial benefit for users.

The implementation of such MDRS revealed very easy. It took few month of student work. Some low cost would 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.

MDRS language is quite similar to SQL. MDRS 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:

```sql
mro open nightlife;
=-dif restaurant,y cinema)
=-range(r1 r2r x)(r1 y, x)
=-select r1.name,r1.country
=-where r1.street=r2.street
CMDS
Haie de Tunis
St. Severin
```
3.5.1.3. **Query processing.**

The algorithms for query processing work as follows:

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

2. If the query reveals an elementary one, i.e., the user has formulated an **MDM** query, then **MDS** calls **MDS** that processes the query as usually.

3. Else **MDS**:
   - determines the relations that have to be extracted from each DB, using some elementary (**MDS**) queries.
   - produces the **MDS** schema of a DB that would be a set of these relations.
   - asks **MDS** to create this DB, if not already, is empty. The DB constitutes the workmultispace called **work DB**.
   - asks **MDS** to execute the elementary queries. Then, asks to reinsert each result (a workspace) into the corresponding relation in the **work DB**.
   - asks **MDS** 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 **MDS** 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. Also, it would probably be less efficient. The reinsertion of tuples into the **work DB** is in fact not a waste of time, but it allows **MDS** to create complex indexes, necessary in order to speed up relational operations such as joins.

3.5.2. **MUNCHAPOL** system.

**MUNCHAPOL** is built on top of **POLYTHEME** DoMMS. It allows users to formulate multidatabase queries to DBs managed by the local machines and the global machine of **POLYTHEME**. The query language is a multidatabase extension of **QUEL** called **QUELON**. Users of **QUEL** may refer to database names. They may also refer to a kind of predefined semantic range, called equivalence dependency /MDM/ or the dependencies defined by **MDS** administrator(3) and is stored in the dependency schemes. **MUNCHAPOL** translates **QUELON** queries into relational algebra **POLYTHEME** queries and, then, manages the query processing.

**Example 3.1.**

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')

RHOST.REST
NAME MEANPRICE
TOKIN 150
```

3.5.3. **MESSIDIOR** system.

3.5.3.1. **Overview.**

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

Currently, hundreds of bibliographic databases are available on the EDINNET, 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 **MESSIDIOR** respond to a strong user demand and should overcome the present annoying state-of-the-art. Presently, no system similar to **MESSIDIOR** is known.

**MESSIDIOR** 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. **MESSIDIOR** may thus potentially be used with any database server site.

3.5.3.2. **MESSIDIOR** language.

A user searches documents during a **session**. A session is an interactive sequence of **MESSIDIOR** 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 term 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.
3.6. CONCLUSIONS.

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

The multibase approach generalizes the database approach. In particular, the concept of a multibase generalizes the one of a database. Also, the concept of an MDBS generalizes the one of a (DBMS). The concepts that render MELMA a multibase generalization of ALPHAS, lead to analogous generalizations of QUEL, SQL or any relational language. The practical consequence is that one gains new possibilities, without losing 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 (UNIX) may be an easy task. In our opinion, this means that new systems for data management should henceforward be typically designed as MDDMSs. 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-MELMA approach provides tools to constitute a single database. The multibase approach provides tools to constitute a multibase. The prototypes that have been developed within the project prove the feasibility of the corresponding industrial systems.

Even a glance at the present state-of-the-art shows that such systems will be highly needed. The rapid development of local networks will soon render necessary the management of distributed enterprise data. The VDASES systems such as FRETEL (UK), TELEFEL (France), VEDION (Canada) or computational systems such as QUEL, (France), or networks, like EUTONET, provide the access to hundreds of heterogeneous data bases. 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 many domains where the techniques for distributed data management apply. In particular, one may cite multiprocessor systems such as SABRE (GARE). Also, this is the case of real-time systems with strong reusability constraints [LESL]. Finally, the Open Systems [ISO 88] 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 M. J. le Bihan, until 1982, then by W. Litwin.
The coordination was also provided by O. Guibaud, G. Sardain and G. le Lann. The project was supported by Agence de l'Informatique (ADI).
REFERENCES


[DEM80]  F. Demolube, Estimation de la durée d'une transaction de mise à jour dans un système de bases de données régionautes, Proceedings of the VLDB, Montreal, October 1980, 89-93.


