Advances in Data Management
Distributed and Heterogeneous Databases
A.Poulovassilis

1 What is a distributed database system ?

A **distributed database system** (DDB system) consists of several databases stored at different sites of a computer network.

The data at each site is managed by a **database server** running some DBMS software.

These servers can cooperate in executing **global queries** and **global transactions** i.e. queries and transactions whose processing may require access to databases stored at different sites.

A significant cost in global query processing is the communications cost incurred by transmitting data between servers over the network.

An extra level of coordination is needed in order to guarantee the ACID properties of global transactions.

In distributed relational databases, a relation may be *fragmented* across multiple sites of the DDB system, for better query performance:

- fragments of relations can be stored at the sites where they are most frequently accessed;
- **intra-query parallelism** can be supported i.e. a single global query can be translated into multiple local subqueries that can be processed in parallel;

Fragmentation might be **horizontal** or **vertical**:

- Horizontal fragmentation splits a relation R into n disjoint subsets $R_1, R_2, \ldots, R_n$ such that
$$R = R_1 \cup R_2 \cup \ldots \cup R_n$$
- Vertical fragmentation splits a relation R into n projections $\pi_{atts_1} R, \pi_{atts_2} R, \ldots, \pi_{atts_n} R$, such that
$$R = \pi_{atts_1} R \bowtie \pi_{atts_2} R \bowtie \ldots \bowtie \pi_{atts_n} R$$
(this is known as a **loss-less join decomposition** of R).

Relations or fragments of fragmented relations can also be **replicated** on more than one site, with the aim of:

- faster processing of queries (using a local copy rather than a remote one)
- increased reliability (if a system failure makes one copy unavailable, another copy may be available elsewhere)

A disadvantage is the added overhead in maintaining consistency of replicas after an update occurs to one of them.

Thus, decisions regarding whether on not to replicate data involve a trade-off between **query costs** and **update costs**.

There is a Global Catalog at each site of the DDB which holds information about the fragmentation, allocation and replication of relations in the DDB.

2 Autonomy and heterogeneity of DDB systems

DDB systems may be either **homogeneous** or **heterogeneous**:

- Homogeneous DDB systems consist of local databases that are all managed by the same DBMS software.
- Heterogeneous DDB systems consist of local databases each of which may be managed by a different DBMS.

  Thus, the data model, query language and the transaction management protocol may be different for different local databases.

Homogeneous DDB systems are generally **integrated** ones:

- Integrated DDB systems provide one integrated view of the data to users.
- A single database administration authority decides what information is stored in each site of the DDB system, how this information is stored and accessed, and who is able to access it.

Heterogeneous DDB systems are generally **multi-database** or **federated** ones:

- Multi-database/federated DDB systems consist of a set of fully autonomous local database systems.
- An extra level of software — a Multi Database Management System (MDBMS) — runs at each site and manages interaction with the multidatabase e.g. global query processing and global transaction management.
- The **local DBAs** have complete authority over the information in their databases, and what part of that information is made available to global queries and global transactions.
- One or more **global DBAs** control global access to the system, but must accept the access restrictions imposed by the local DBAs.

For simplicity, I will use the term **homogeneous DDB** to mean 'integrated, homogeneous DDB' and I will use the term **heterogeneous DDB** to mean 'multi-database, heterogeneous DDB'.

Homogeneous DDB systems conform to a **4-schema architecture**:

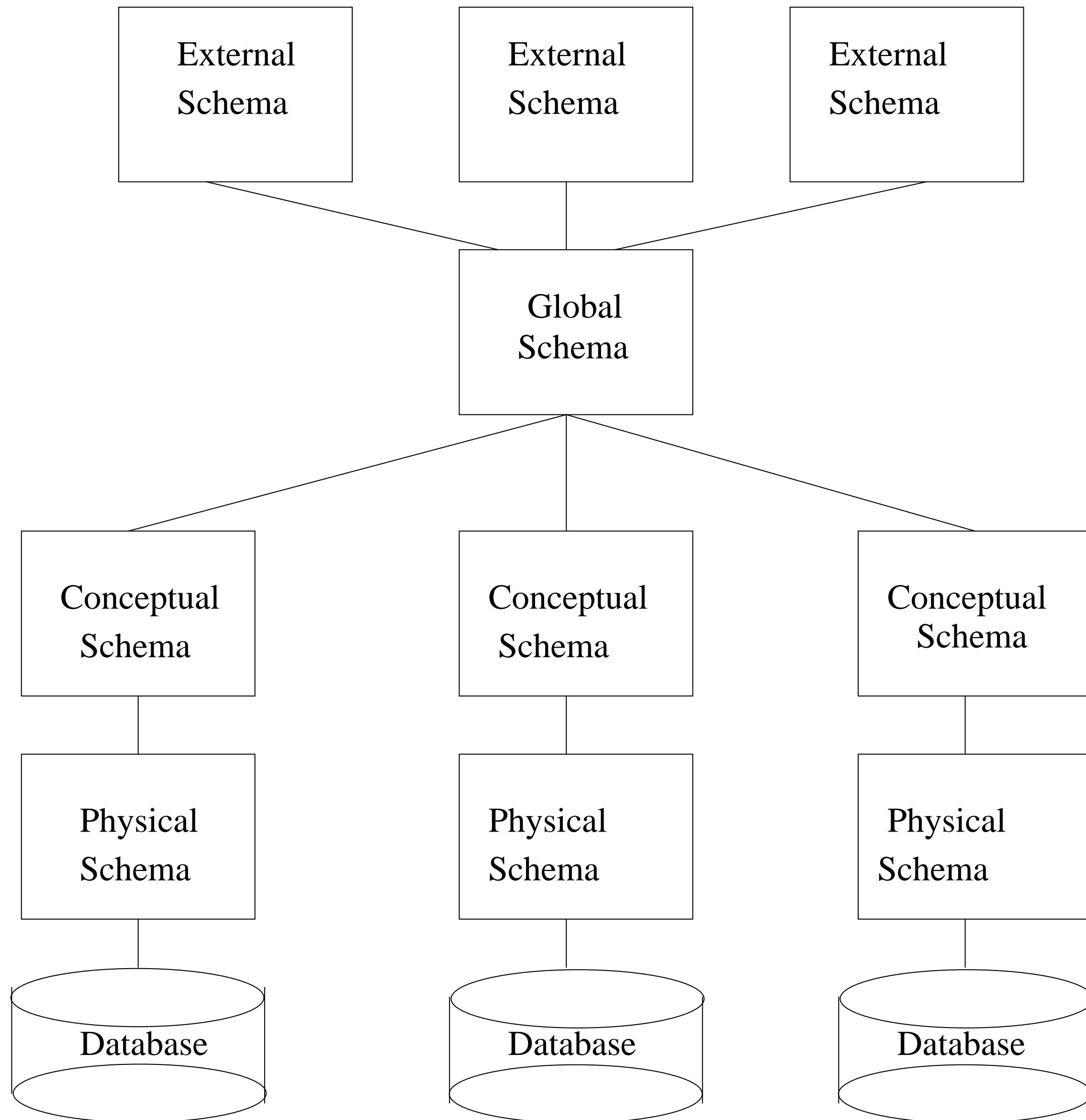
(i) a **local physical schema** for each local database

(ii) a **local conceptual schema** for each local database

(iii) a single **global schema**, which is the union of all the local conceptual schemas

(iv) a number of **global external schemas**, each of which is a view over the global schema.

The following figure illustrates the schema architecture of a homogeneous DDB system:



A **heterogeneous DDB** system differs from this in that

- there is another layer of schemas, the **component schemas**, above the local conceptual schemas;

  this is because local databases may support different data models; so their schemas need to be translated into some **Common Data Model** (CDM) before they can be integrated;

  the component schemas are the translation of the local conceptual schemas into the CDM;

- there is another layer of schemas, the **export schemas**, above the component schemas;

  each export schema defines the part of the local component schema that the local DBA wishes to make accessible to global queries and transactions;

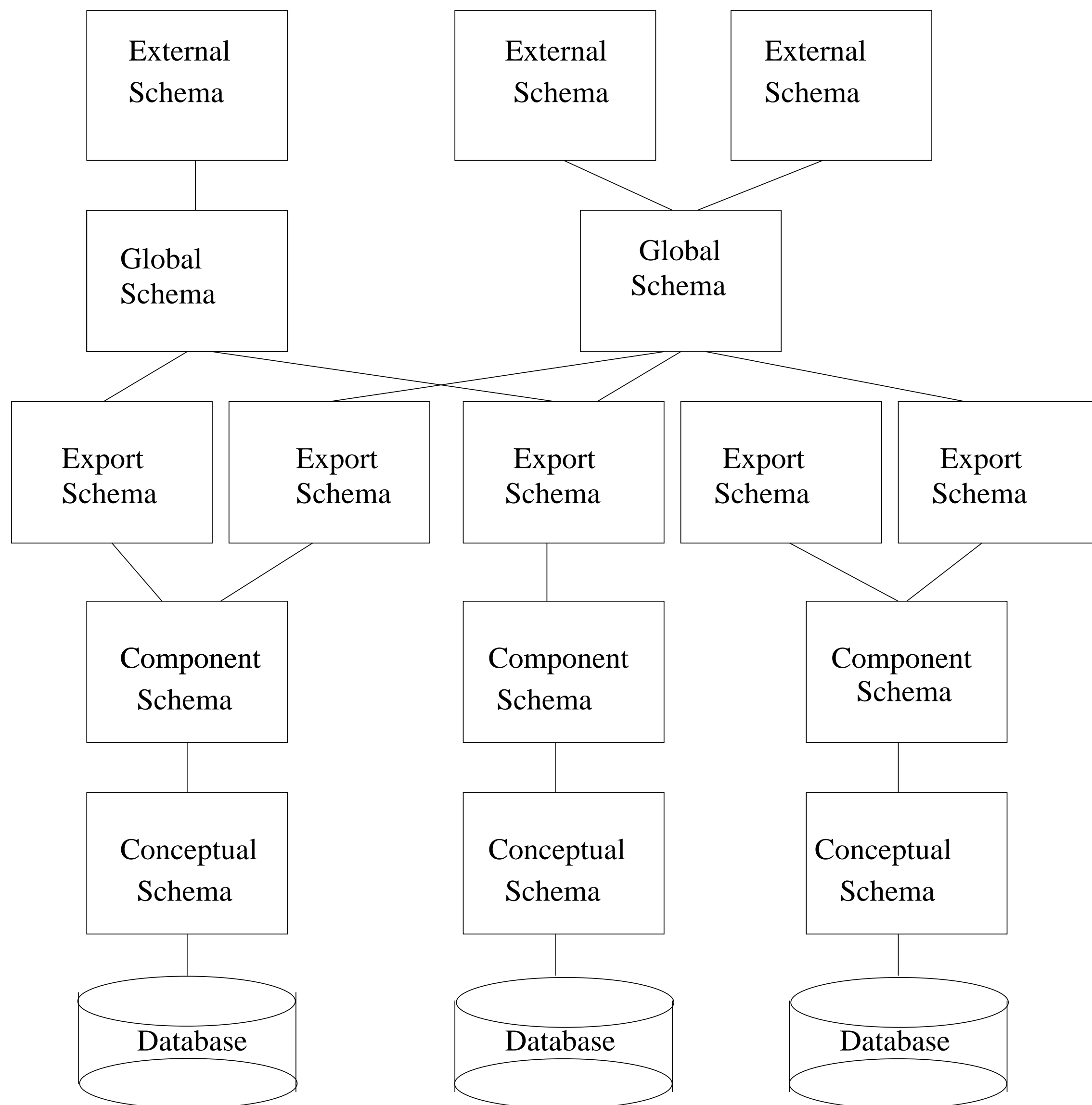
- there may be more than one global schemas, each formed as the union of some of the export schemas;

  this is because it may not be possible, or desirable, to form a single global schema that integrates all of the export schemas;

  the global schemas are all expressed in the CDM;

- **local external schemas** can also exist, due to the autonomy of the local databases.

The following figure illustrates the architecture of a heterogeneous DDB system (the local physical schemas and local external schemas are not shown):



3 Designing Heterogeneous DDBs

The two main steps in designing heterogeneous DDB systems are:

1. **schema translation**, of each local conceptual schema into the component schema; and
2. **schema integration**, of the export schemas into a global schema.

3.1 Schema translation

Each local conceptual schema needs to be translated into a component schema expressed in the CDM. There are standard algorithms for translating a schema expressed in one data model (that of the local database) into an equivalent schema expressed in a different data mdel (the CDM).

The mapping between constructs of the local conceptual schema and constructs of the component schema is stored in the Global Catalog of the MDBMS software managing that site.

This mapping is needed in order to translate queries that are expressed over constructs of the export schema (arising from global queries submitted to a global schema) into queries over the constructs of the local conceptual schema, so that such queries can be submitted to the local DBMS for optimisation and processing by the local query processor.

Example 1. Suppose a heterogeneous database system has access to a relational database and an object-oriented database, and we wish to integrate them using the relational model as the CDM.

The relational database contains five tables and its schema, L_1, is as follows:

$Student(\underline{studentId}, name, address, \mathbf{tutorId})$
$Staff(\underline{tutorId}, name, deptName)$
$Lecturer(\underline{lecturerId}, name, deptName)$
$Course(\underline{courseId}, name, programme)$
$Teaches(\underline{\mathbf{lecturerId}}, \underline{\mathbf{studentId}})$

Here, the underlined attributes are key attributes, and the bold-face attributes are foreign key attributes referencing the key attributes of the same name in other tables.

Since we are going to use the relational model as the CDM, the component schema corresponding this database, C_1, is identical to the above local schema, L_1.

The schema, L_2, of the other OO database contains four classes *Department*, *Course*, *Enrollment* and $Staff$. Each *Department* object has attributes

$$\begin{aligned} \underline{deptName} &: string; \\ deptHead &: string; \end{aligned}$$

where underlinining signifies a key attribute. Each *Course* object has attributes

$$\begin{aligned} \underline{courseId} &: integer; \\ courseName &: string; \\ units &: integer; \end{aligned}$$

Each *Enrollment* object has attributes

$$\begin{aligned} \underline{studentId} &: integer; \\ \underline{year} &: integer; \\ courseEnrollments &: set(Course) \end{aligned}$$

Each *Staff* object has attributes

$$\begin{aligned} \underline{staffId} &: integer; \\ name &: string; \\ worksIn &: Department; \\ teaches &: set(Course) \end{aligned}$$

Here are some general rules for translating an OO schema into a relational schema (N.B. these are a simplification of the full set of translation rules e.g. they ignore subclasses and structured attributes):

1. Each class C translates into a relation C^R.
2. If there is a set of attributes of C that uniquely identifies the instances of C, these are designated as the key attributes of C^R. Otherwise, add an attribute C_id to C^R which will serve at its key.
3. For each other attribute a of C:
   (a) if the type of a is another class, C' say, then a translates into a foreign key in C^R, referencing the key attribute(s) of C'^R;
   (b) if the type of a is $set(C')$ for some class C', then a is translates into a new relation whose attributes are the key attribute(s) of C^R and the key attribute(s) of C'^R.
   (c) otherwise a translates into an attribute of C^R.

Applying these rules, the information in the local OO schema translates into the following relational schema:

$Department(\underline{deptName}, deptHead)$
$Course(\underline{courseId}, courseName, units)$
$Enrollment(\underline{studentId}, \underline{year})$
$Staff(\underline{staffId}, name, \mathbf{deptName})$
$Teaches(\underline{\mathbf{staffId}}, \underline{\mathbf{courseId}})$
$CourseEnrollments(\underline{\mathbf{studentId}}, \underline{\mathbf{year}}, \underline{\mathbf{courseId}})$

The mapping from this component schema to the local OO schema can automatically be generated to be as follows, where L_2 denotes the local OO schema, and C_2 the above component schema[1]:

$$C_2 : Department \quad = \quad \text{SELECT } o.deptName, o.deptHead \text{ FROM } L2 : Department\ o$$

[1]The syntax of the queries on the RHS of these definitions is that of the OO Query language OQL.

$$
\begin{aligned}
C_2 : Course &= \text{SELECT } o.courseId, o.courseName, o.units \text{ FROM } L_2 : Course\ o \\
C_2 : Enrollment &= \text{SELECT } o.studentId, o.year \text{ FROM } L_2 : Enrollment\ o \\
C_2 : Staff &= \text{SELECT } o.staffId, o.name, o.worksIn.deptName \text{ FROM } L_2 : Staff\ o \\
C_2 : Teaches &= \text{SELECT } o.staffId, c.courseId \text{ FROM } L_2 : Staff\ o,\ o.teaches\ c \\
C_2 : CourseEnrollments &= \text{SELECT } o.studentId, o.year, c.courseId \text{ FROM } L_2 : Enrollment\ o, \\
& \quad o.courseEnrollments\ c
\end{aligned}
$$

3.2 Schema integration

After translating the local schema into the component schema, one or more export schemas can be defined.

The set of export schemas contributing to each global schema then need to be **integrated**.

The mappings between the export schemas and the global schemas are stored in each global catalog. They are subsequently used by the global query processor in order to replace each construct of a global schema appearing in a global query by the equivalent constructs from one or more export schemas.

There are three main steps that the global DBA has to undertake in order to integrate a set of export schemas into a global schema:

1. **schema conforming** — making sure that all the export schemas represent the same information using the same schema constructs;

   since the local databases will typically have been designed by different people at different times to serve different application requirements, **conflicts** may exist between export schemas;

   thus, schema conforming requires **conflict detection** and **conflict resolution**, resulting in new versions of the export schemas which represent the same information using the same schema constructs;

2. **schema merging** — identifying common constructs between different export schemas, and producing a single integrated schema;

3. **schema improvement** — improving the quality of the integrated schema by removing any redundant information.

A series of **equivalence-preserving transformations** can be applied in steps 1 and 3:

During step 1, the transformations that are applied include:

- removing **synonyms** and **homonyms**:

  synonyms are schema constructs with different names but representing the same real-world concept;

  homonyms are schema constructs with the same name but representing different real-world concepts;

we need to **rename** schema constructs so that constructs representing different real-world concepts have different names, and constructs representing the same real-world concept have the same name;

- removing **structural conflicts** e.g.
  - converting a relation $R(\underline{k_1}, \ldots, \underline{k_n}, a_1, \ldots, a_m)$ into m relations $R(\underline{k_1}, \ldots, \underline{k_n}, a_1)$, $\ldots$, $R(\underline{k_1}, \ldots, \underline{k_n}, a_m)$;
  - or vice versa;
  - using an attribute of a relation whose domain has n possible values to create n new relations without that attribute (e.g. for a relation People(Name,DoB,Gender) where the attribute Gender has possible values M and F, we can create two relations Male(Name,DoB) and Female(Name,Dob));
  - or vice versa.

During step 3, we can

- add **implied** schema constructs, or
- remove **redundant** schema constructs.

Example 2. To illustrate, consider the two relational component schemas, C_1 and C_2 from Example 1. Suppose that the two export schemas, ES_1 and ES_2, that we wish to integrate into a global schema are identical to C_1 and C_2, respectively.

During schema conforming, the following steps take place:

- there is a structural conflict between ES_1 and ES_2 regarding departments, in that in ES_1 a department is represented by an attribute while in ES_2 it is represented by a relation;

  we can add to ES_1 a new relation $Department(\underline{deptName})$ together with new foreign key constraints on $deptName$ in $Lecturer$ and $Staff$;

- there is a naming conflict between $Lecturer$ and $Staff$ in ES_1 and $Staff$ in ES_2, in that $Staff$ in ES_2 represents the set of course lecturers as does $Lecturer$ in ES_1, whereas $Staff$ in ES_1 represents the set of students' tutors;

  thus, $Staff$ in ES_2 should be remaned to $Lecturer$;

- the attribute $Lecturer.staffId$ in ES_2 should be renamed to $lecturerId$, to conform with $Lecturer.lecturerId$ in ES_1; the attribute $Teachers.staffId$ in ES_2 should similarly be renamed to $lecturerId$;

- the attribute $Course.name$ in ES_1 should be renamed to $courseName$, to conform with $Course.courseName$ in ES_2;

- there is a naming conflict between the relations $Teaches$ in ES_1, where it represents students taught by lecturers, and $Teaches$ in ES_2 where it represents courses taught by lectures;

  one of these relations must be renamed to a different name — let's say $Teaches$ in ES_1 is renamed to $TeachesStudents$.

The schemas have now been conformed, and can be merged on their common relations and attributes, obtaining:

$Student(\underline{studentId}, name, address, \textbf{tutorId})$
$Staff(\underline{tutorId}, name, \textbf{deptName})$
$Lecturer(\underline{lecturerId}, name, \textbf{deptName})$
$Course(\underline{courseId}, courseName, units, programme)$
$TeachesStudents(\underline{\textbf{lecturerId}}, \underline{\textbf{studentId}})$
$Department(\underline{deptName}, deptHead)$
$Enrollment(\underline{studentId}, \underline{year})$
$CourseEnrollments(\underline{\textbf{studentId}}, \underline{\textbf{year}}, \underline{\textbf{courseId}})$
$Teaches(\underline{\textbf{lecturerId}}, \underline{\textbf{courseId}})$

Finally, during schema improvement:

- we can remove the redundant relation *Enrollment* since this information can be derived by projecting on the $studentId, year$ attributes of *CourseEnrollments*; and
- we can remove the redundant relation *TeachesStudents* since this information can be derived by joining *Teaches* and *CourseEnrollments* over their *courseId* attribute.

The final integrated schema after all of the above transformations is as follows:

$Student(\underline{studentId}, name, address, \textbf{tutorId})$
$Staff(\underline{tutorId}, name, \textbf{deptName})$
$Lecturer(\underline{lecturerId}, name, \textbf{deptName})$
$Course(\underline{courseId}, courseName, units, programme)$
$Department(\underline{deptName}, deptHead)$
$CourseEnrollments(\underline{\textbf{studentId}}, \underline{\textbf{year}}, \underline{\textbf{courseId}})$
$Teaches(\underline{\textbf{lecturerId}}, \underline{\textbf{courseId}})$

We will see later how in the AutoMed heterogeneous data integration system the mappings relating these global schema constructs to the constructs of the two component schemas C_1 and C_2 can be generated automatically from a sequence of schema transformations such as those above.

So far I have ignored the possibility that different databases may have different representations for the same information.

For example, for sums of money different currencies, for weights and measures different units of measurement, for numeric amounts different precision etc.

This **representational heterogeneity** also has to be taken into account during the schema integration process, and conversion information has to be incorporated into the mappings between export and global schemas.

I have also ignored the possibility that different local databases may contain conflicting data e.g.

- two Customer databases may contain different addresses for the same person;
- two Income databases may contain different incomes for the same person, etc.

Thus, a **conflict resolution policy** is needed for each of these situations, whereby a selection or aggregation criterion is applied to the conflicting data items. For example,

- one Customer database may be regarded as 'primary' and its information may override that of the other;
- the two incomes from the two Income databases may be added together to form the income data item in the global schema.

AutoMed can handle representational heterogeneity and can also capture conflict resolution policies.

Homework

1. Read the handouts on the AutoMed data integration toolkit, and its IQL query language.

   Writing IQL queries may be required in the exam. However, you won't be required to write queries as complex as the one on the top of page 9 of the IQL handout (which expresses an outerjoin).

2. Read the Appendices to these Notes, and the handout on Oracle's distributed database functionality (for interest only - not examinable).

Appendix I - Distribution Transparency

Ideally, a DDB should make the distribution of data transparent to the user i.e. the distributed data should appear as a single database, and be accessible as if it were stored in a single database.

There are different levels of transparency e.g.

- **fragmentation transparency**: the user does not need to know how data is fragmented, perceives the distributed data as a single database, and uses global names in queries/updates
- **location transparency**: the user does not need to know where data fragments are located in the network, but does need to know the names of the fragments; fragment names need to be used in queries/updates
- no transparency: the user needs to specify both the names and the locations of fragments in queries/updates

Globally unique names are necessary for identifying fragments and replicas in a DDB e.g. names of the form
CreationSiteId.RelationName.FragmentId.CopyId.

Having to use such names directly in queries would result in loss of distribution transparency, so a common solution is to set up local aliases for them in each site's Global Catalog.

Appendix II - Physical database connectivity in heterogeneous DDBs

There are two main approaches to interconnecting different DBMSs — **gateways** and **standard APIs**

A gateway allows one DBMS to exchange information with another DBMS, but different gateway software is needed for each combination of different vendors' DBMSs and thus the approach does not scale very well in highly heterogeneous systems.

This has led to the emergence of standard Application Programming Interfaces (APIs) supported by DBMSs, such as JDBC and ODBC. This allows open, extensible heterogeneous DDB systems to be built, interconnecting any DBMS that supports the API.

Standard APIs are a more generic solution but, since they represent the lowest common denominator, they may allow less information to be shared than via special-purpose gateways.

Use of a standard API involves a driver manager and a driver (for each particular DBMS, Operating System, and Network protocol). Applications communicate with the driver manager to load a particular driver, and then submit requests to the driver.