Distributed Database – SCS1613

UNIT 5

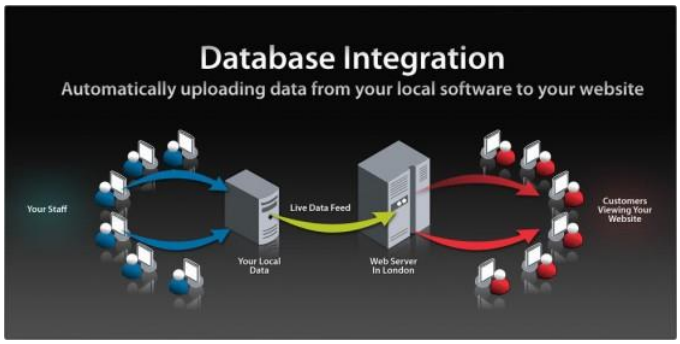
DATABASE INTEGRATION AND MANAGEMENT

Database Integration- Scheme Translation- Scheme Integration- Query Processing Query Processing Layers in Distributed Multi-DBMSs- Query Optimization Issues- Transaction Management Transaction and Computation ModelMultidatabase Concurrency Control- Multidatabase Recovery- Object Orientation And Interoperability- Object Management Architecture - Distributed Component Model

DATABASE INTEGRATION

Database integration means that multiple different applications have their data stored in a specific database – the integration database – so that data is available across all of these different applications. In other words, the data is available between two different parties and therefore, can be easily accessed and implemented into a different application without having to transfer to a different database.

For the database integration to work successfully, it needs to have a plan that allows for all of the client applications to be taken into account. Whether the scheme is more complex, general or both is irrelevant because a separate group controls the database to negotiate between the numerous different applications and the database group. In other words, this plan makes it possible for all the applications to be grouped together into that one database group.



NECESSARY OF DATABASE INTEGRATION

The fundamental reason that database integration is necessary is because it allows for data to be shared throughout an organization without there needing to be another set of integration services on each application. It would be a tremendous waste of resources if each application needed something to convert the data into data it can read. By using database integration, it allows for the information to automatically be integrated so if, at any time the data is needed, it can be pulled up and accessed.

On top of that, it helps when two companies that are merging have their data integrated because when their databases come together, the data can mesh easily. If the data wasn't integrated, an server manager would have to go in and manually integrate everything which can become a hassle and, as previous mentioned, result in a waste of resources. Therefore, integrating before a merger is definitely ideal.

Another application that database integration can be used for is in the scientific community. When collecting data, a scientist might use one application for one bit of data. Then, he'll go to a different application for a different bit of data. By having database integration, the data becomes readily available across the spectrum without thereneeding to be any wasted time. This results in more successful experiments.

All in all, database integration is becoming a technology that more companies are investing in, especially as the quantity and connectivity of data increases. As people need to access more data and share data between departments, companies have realized that have all the data integrated on a database is an incredible time saver.

DATABASE INTEGRATION = TRANSLATION AND SCHEMA INTEGRATION

- **Database integration** is done in most cases in two-steps : **schema translation** (or simply **translation**) and **schema integration**.
- **Scheme translation** means the translation of the participating local schemes into a **common intermediate canonical representation.**

  e.g. if a network and a relational model is used, then an intermediate data model should be chosen, if it is the relational one, the database scheme formulated in the network model is translated into a scheme based on the relational model.

- Scheme transformation is of course only necessary if different data models are involved.
- The scheme integration integrates each intermediate schemes into a **global conceptual scheme**.
- All intermediate schema base on the same data model, the so called **target model**, which is of course the data model for the global conceptual scheme**.**

THE EXAMPLE FOR THE TRANSLATION AND THE SCHEMA INTEGRATION

- We consider the following three local schema. The first one is based on the relational model, the second one on the network model (the CODEASYL network) and the third one on the entity-relationship data model.
- **First** scheme, the **Relational Engineering Database Representation** :

  E(ENO, ENAME, TITLE) *Each Engineer Description*
  J(JNO, CNAME, JNAME, BUDGET, LOC) *Job Description*
  G(ENO, JNO, RESP ,DUR) *Engineer to Job relation description*
  S(TITLE, SALARY) *Salary description*

- **Second** scheme : the **CODEASYL Network Definition of the Employee Database**：

  Two records : DEPARTMENT and EMPLOYEE and their attributes DEP-NAME and so on. One link between the records with →, named employs which links the two corresponding records. It can model only one-to-many relationships. The schema representation looks like :

  DEPARTMENT : DEP-NAME BUDGET MANAGER →(employs) EMPLOYEE : E# NAME ADDRESS TITLE SALARY

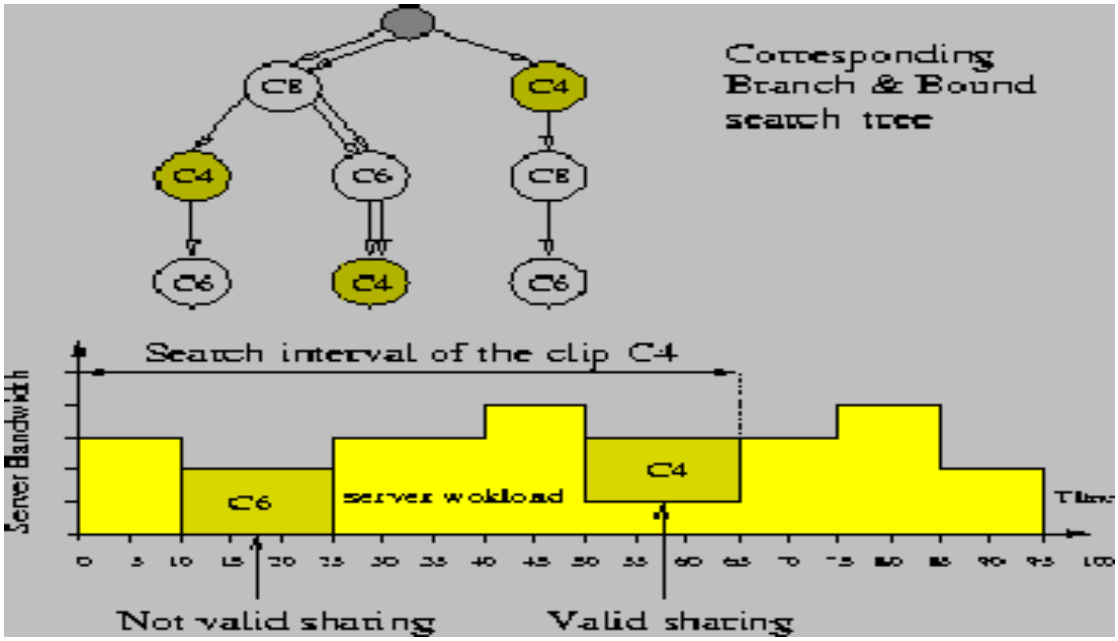
SCHEMA TRANSLATION :

- Schema translation is the task of mapping one schema to another.
- Requires the specification of the **target data model** for the global conceptual schema definition.
- Some rare approaches did merge the translation and integration phase, but increases the complexity of the whole process.

- In the example, the Entity-Relationship model is chosen as the target model.
- The first scheme translation is the CODEASYL network schema to an E-R-scheme one.

SCHEME TRANSLATION 1 : CODEASYL SCHEMA TO E-R SCHEMA

- One entity is created for each record Thus, an EMPLOYEE and one DEPARTMENT entity is created.
- The attributes of the records are taken directly into the E-R scheme.
- Finally, the links employs becomes a many-to-one relationship from the EMPLOYEE entity to the DEPARTMENT entity. The final model looks like :



Remark : Many-to-many relationships modeled in the network by some intermediate records can be represented directly by one many-to-many relationship (→ translation should be optimized).

SCHEME TRANSLATION 2 : RELATIONAL SCHEMA TO E-R SCHEMA

- The example relational model of the engineering database consists of four relations :

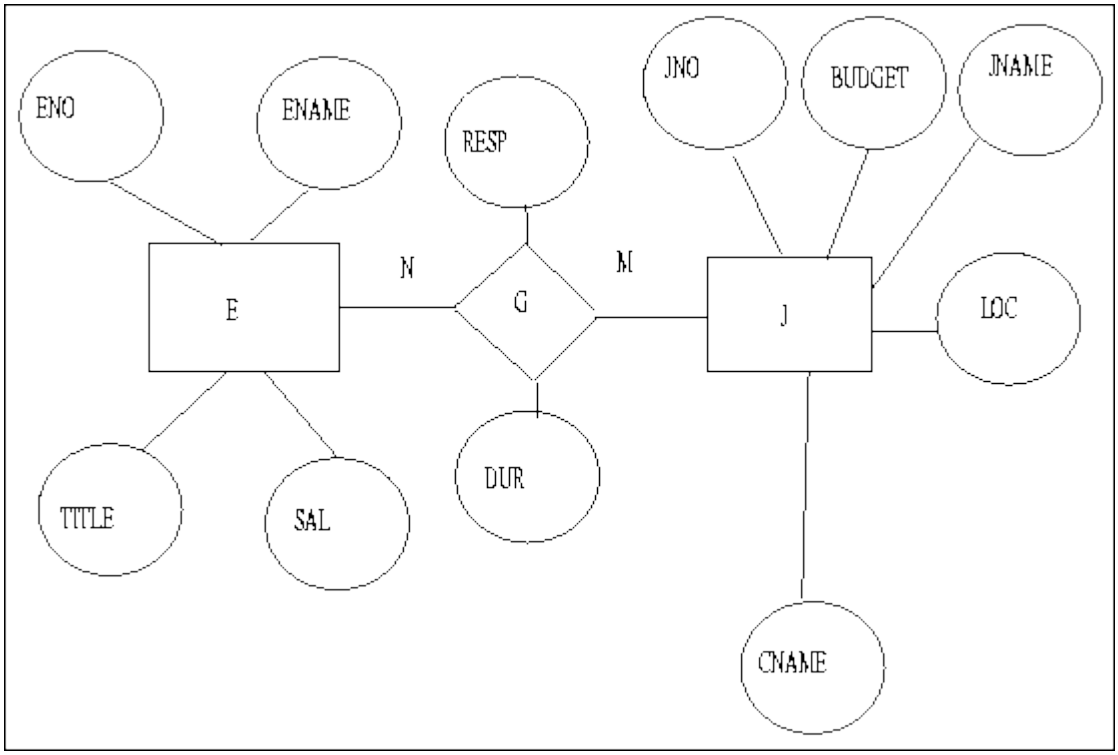
E(ENO, ENAME, TITLE) *Each Engineer Description*
J(JNO, CNAME, JNAME, BUDGET, LOC) *Job Description*
G(ENO, JNO, RESP ,DUR) *Engineer to Job relation description*
S(TITLE, SALARY) *Salary description*

- Identify the base relations : E and J clearly corresponds to an entity.
- Identify the relationships : G corresponds to a relationship, ENO and JNO are foreign keys, thus a relationship between J and E can be identified.
- Handling of S is difficult.

- First it can be a entity. In such a case a relationship between S and E must be established (this would be a many-to-one relationship, e.g. pay between S and E). No relation is specified for this relationship.

  An employee could have only one salary, but a salary can belong to many employees.

- Second salary could be an attribute of E, cleaner, but the relationship between the title and salary is lost.
- See below the result E-R scheme, with SAL as attribute of E.



SCHEME INTEGRATION :

- All local scheme are now translated to an intermediate scheme based on the target model. The task of the schema integration is now to generate the **global conceptual schema (CGS)**, which can be queried by the user of the MDBMS.
- Ozsu defines the schema integration, as the process of **identifying** the components of a database which are related to one another, **selecting** the best representation for the global conceptual schema and finally **integrating** the components of each intermediate schema.
- Integration methodologies are either **binary** or u***nary***

  Binary integration methodologiesinvolves the manipulation of two schema at a time. These occurs either ladder (linear tree !) or purely binary (bushy tree !).

- Binary are either one-shot (integration of all schema) or interactive (integration of 2,3,4 .. at a time). Binary approaches are a special case of the latter.

In general, the one-shot approach is very complex and rarely used, mostly the binary approach is used (Determine the best ordering!).

- Very good graphical tools exists now which help the identification and integration approach.

OVERVIEW OVER THE SCHEMA INTEGRATION :

- **Preintegration** : identify the keys and defines the ordering of the binary processing approach.
- **Comparison** : Identification of naming and structural conflicts.
- **Conformation** : Resolution of the naming and structural conflicts.
- **Restructeration and Merging** of the different intermediate schema to the global conceptual scheme (GCS).
- Interaction with an integrator is absolutely necessary.

Preintegration

- **Preintegration** establishes the rules of the integration process, i.e. the integration method is selected (e.g. binary iterative n-ary) and then the **order of the schema integration** (i.e. which intermediate schema is integrated with which one first).
- **Candidate keys** are determined. Here for each of the entities in all intermediate schemes, the keys are determined.
- Potentially **equivalent domains** of attributes are detected and transformation rules between the domains should be determined (e.g. one scheme defines the attribute temperature in Grad Celsius, the other one in Fahrenheit, transformation rules between the different domains should be prepared for further integration).

Comparison

- The comparison phase detects **naming conflicts**, **relationships between schemes** and **structural conflicts**.
- Naming conflicts are either the **synonym** or the **homonyms** problem.
- Two identical entities which have different names are **synonyms**, and two different entities that have identical names, but are not identical entities, are **homonyms**.
- Example 1 : ENGINEER and E are synonyms and they both refer to an engineer entity.
- Title in the network model refers to an employee and is different from the title related to an engineer, thus these are homonyms**.**

Relationship between the schemes

- The determination of the relationship bases on the recognition of the synonyms as described before.
- There are four possibilities of relationships between schemes
- 1) Equivalent
  2) One is subset of the other
  3) Some components from any may occur in the other
  4) Completely no overlap.

Structural conflicts

- **Type Conflicts** : Type conflict happens if the same object is represented by an attribute in the one intermediate scheme and by an entity in another scheme.

- **Dependency Conflicts** : This conflict occurs, when the different relationship modes (e.g. one-to-one versus many-to-many) are used to represent the same thing in different schemas.
- **Key Conflicts** : This conflict happens, if different candidate keys are available and different primary keys are selected in different scheme.
- **Behavorial Conflicts** are implied by the modeling process. For example deleting the last employee out of the employee record can result in an empty department, as for the engineers this may not be allowed).

Conformation

- **Conformation** is the resolution of the conflicts that are determined at the comparison phase.
- **Naming conflicts** are resolved by simply renaming conflicting ones. In the case of **homonyms**, the identical entities or attributes are extended with the name of the entity and the name of the scheme it belongs to.
- **Structural conflicts** are resolved by transforming entities/attributes or relationships between them.

Resolving structural conflicts

- **Resolving** attribute to represent it.
- **Remark** : Key attributed to Entities require supplemental steps.
- **Dependency Conflicts** will be resolved by choosing the most general relationship.
- The restructuring is virtually an art rather than a science. Semantic knowledge about the all intermediate schemas is repaired, which makes an automatic resolution difficult. There exists many supporting tools.
- structural conflicts means the **restructuring** of some schemes to eliminate the conflicts.
- **Attribute &Entity** : A non-key attribute can be transformed into an entity by creating an intermediate relationship connecting the new entity and a new **Merging and Restructuration**

- All modified and non-conflicting schemes must be **first merged** into a single database schema and **secondly restructured** to create the 'best' (see later) one.
- The **merging** follows the integration order ones fixed in the Preintegration. The merging should be **complete**, i.e. all components of all the intermediate schema should be find their place in the merged one.
- Now a **Restructuration** would take place which searches for the minimal one, thus the redundant relationships are removed.
- Finally, the scheme could be re-transformed to be more **understandable**. This process is in its great parts autonomous and this mechanism ignores all kind of understandability, it is often necessary by the integrator to rebuild or extend some relationships (here the minimalist can be lost) in a way that the user can understand the scheme and thus formulate correct queries.

ACID PROPERTIES : atomicity, consistency, isolation, and durability.

Atomicity

A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.

Consistency

A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state.

Isolation

Even though transactions execute concurrently, it appears to each transaction T, that others executed either before T or after T, but not both.

Durability

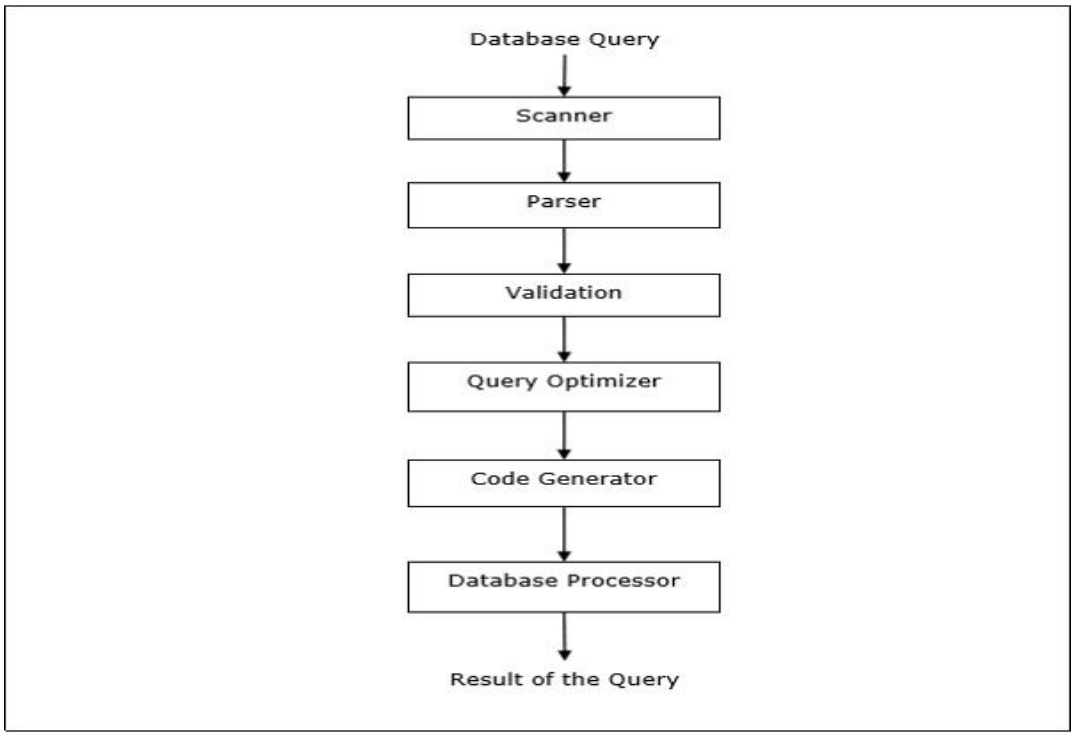
Once a transaction completes successfully (commits), its changes to the database survive failures and retain its changes.

QUERY PROCESSING :

• Query Processing Overview
• Query Optimization
• Distributed Query Processing Steps

QUERY PROCESSING :

Query processing is a set of all activities starting from query placement to displaying the results of the query. The steps are as shown in the following diagram



RELATIONAL ALGEBRA

Relational algebra defines the basic set of operations of relational database model. A sequence of relational algebra operations forms a relational algebra expression. The result of this expression represents the result of a database query.

The basic operations are −

- Projection
- Selection
- Union

- Intersection
- Minus
- Join

Projection

Projection operation displays a subset of fields of a table. This gives a vertical partition of the table.

Syntax in Relational Algebra

π<AttributeList>(<TableName>)π<AttributeList>(<TableName>)

For example, let us consider the following Student database −

STUDENT				
Roll_No	**Name**	**Course**	**Semester**	**Gender**
2	Amit Prasad	BCA	1	Male
4	Varsha Tiwari	BCA	1	Female
5	Asif Ali	MCA	2	Male
6	Joe Wallace	MCA	1	Male
8	Shivani Iyengar	BCA	1	Female

If we want to display the names and courses of all students, we will use the following relational algebra expression −

πName,Course(STUDENT)πName,Course(STUDENT)

Selection

Selection operation displays a subset of tuples of a table that satisfies certain conditions. This gives a horizontal partition of the table.

SYNTAX IN RELATIONAL ALGEBRA

σ<Conditions>(<TableName>)σ<Conditions>(<TableName>)

For example, in the Student table, if we want to display the details of all students who have opted for MCA course, we will use the following relational algebra expression −

σCourse="BCA"(STUDENT)σCourse="BCA"(STUDENT)

Combination of Projection and Selection Operations

For most queries, we need a combination of projection and selection operations. There are two ways to write these expressions −

- Using sequence of projection and selection operations.
- Using rename operation to generate intermediate results.

For example, to display names of all female students of the BCA course −

- Relational algebra expression using sequence of projection and selection operations

πName(σGender="Female"ANDCourse="BCA"(STUDENT))πName(σGender="Female"ANDCourse="BCA"(STUDENT))

- Relational algebra expression using rename operation to generate intermediate results

FemaleBCAStudent←σGender="Female"ANDCourse="BCA"(STUDENT)FemaleBCAStudent←σGender="Female"ANDCourse="BCA"(STUDENT)

Result←πName(FemaleBCAStudent)Result←πName(FemaleBCAStudent)

Union

If P is a result of an operation and Q is a result of another operation, the union of P and Q (pUQpUQ) is the set of all tuples that is either in P or in Q or in both without duplicates.

For example, to display all students who are either in Semester 1 or are in BCA course −

Sem1Student←σSemester=1(STUDENT)Sem1Student←σSemester=1(STUDENT)

BCAStudent←σCourse="BCA"(STUDENT)BCAStudent←σCourse="BCA"(STUDENT)

Result←Sem1StudentUBCAStudentResult←Sem1StudentUBCAStudent

Intersection

If P is a result of an operation and Q is a result of another operation, the intersection of P and Q (p∩Qp∩Q) is the set of all tuples that are in P and Q both.

For example, given the following two schemas −

EMPLOYEE

EmpID	Name	City	Department	Salary

PROJECT

PId	City	Department	Status

To display the names of all cities where a project is located and also an employee resides −

CityEmp←πCity(EMPLOYEE)CityEmp←πCity(EMPLOYEE)

CityProject←πCity(PROJECT)CityProject←πCity(PROJECT)

Result←CityEmp∩CityProjectResult←CityEmp∩CityProject

Minus

If P is a result of an operation and Q is a result of another operation, P - Q is the set of all tuples that are in P and not in Q.

For example, to list all the departments which do not have an ongoing project (projects with status = ongoing) −

AllDept←πDepartment(EMPLOYEE)AllDept←πDepartment(EMPLOYEE)

ProjectDept←πDepartment(σStatus="ongoing"(PROJECT))ProjectDept←πDepartment(σStatus="ongoing"(PROJECT))

Result←AllDept−ProjectDeptResult←AllDept−ProjectDept

Join

Join operation combines related tuples of two different tables (results of queries) into a single table.

For example, consider two schemas, Customer and Branch in a Bank database as follows −

CUSTOMER

CustID	AccNo	TypeOfAc	BranchID	DateOfOpening

BRANCH

BranchID	BranchName	IFSCcode	Address

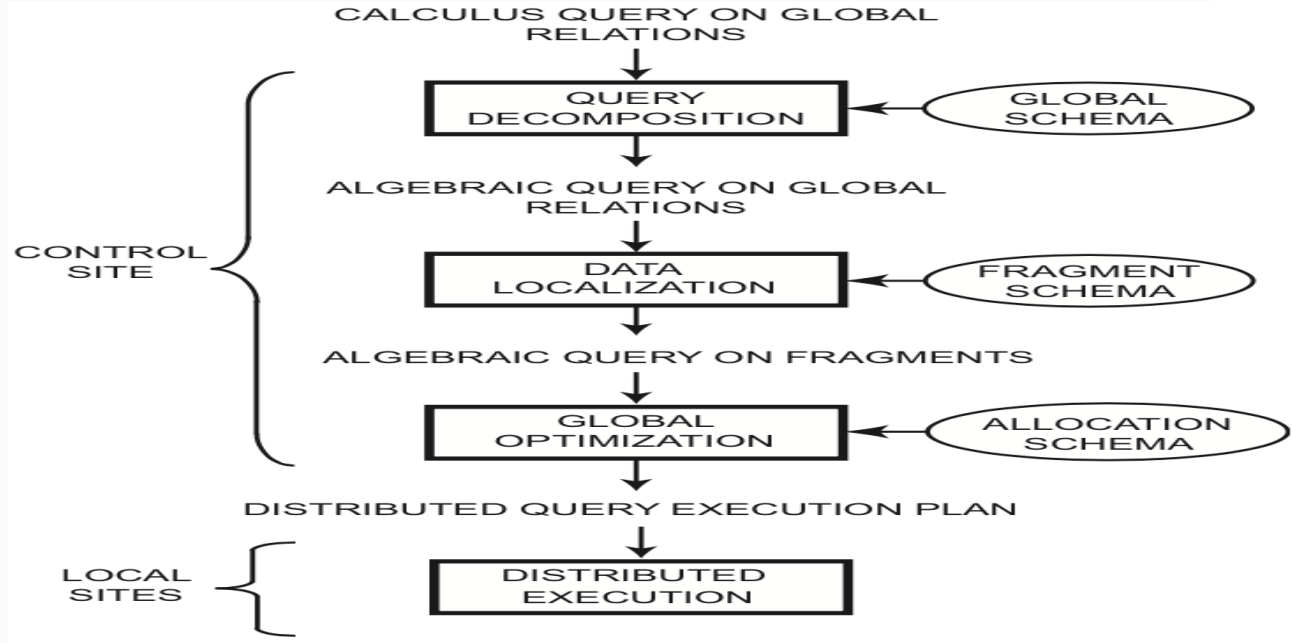
To list the employee details along with branch details −

Result←CUSTOMER⋈Customer.BranchID=Branch.BranchIDBRANCHResult←CUSTOMER⋈Customer.BranchID=Branch.BranchIDBRANCH

LAYERS OF QUERY PROCESSING :

The problem of query processing can itself be decomposed into several subproblems, corresponding tovarious layers. A generic layering scheme for query processing is shown where each layer solves a well-defined subproblem. To simplify the discussion, let us assume a static and semicentralized query processor that does not exploit replicated fragments. The input is a query on global data expressed in relational calculus. This query is posed on global (distributed) relations, meaning that data distribution is hidden. Four main layers are involved in distributed query processing. The first three layers map the input query into an optimized distributed query execution plan. They perform the functions of query decomposition, data localization, and global query optimization. Query decomposition and data localization correspond to query rewriting. The first three layers are performed by a central control site and use schema information stored in the global directory. The fourth layer

performs distributed queryexecution by executing the plan and returns the answer to the query. It is done by the local sites and the control site.



GENERIC LAYERING SCHEME FOR DISTRIBUTED QUERY PROCESSING

QUERY DECOMPOSITION

The first layer decomposes the calculus query into an algebraic query on global relations. The information needed for this transformation is found in the global conceptual schema describing the global relations. However, the information about data distribution is not used here but in the next layer. Thus the techniques used by this layer are those of a centralized DBMS.

Query decomposition can be viewed as four successive steps. First, the calculus query is rewritten in a normalized form that is suitable for subsequent manipulation. Normalization of a query generally involves the manipulation of the query quantifiers and of the query qualification by applying logical operator priority.

Second, the normalized query is analyzed semantically so that incorrect queries are detected and rejected as early as possible. Techniques to detect incorrect queries exist only for a subset of relational calculus. Typically, they use some sort of graph that captures the semantics of the query.

Third, the correct query (still expressed in relational calculus) is simplified. One way to simplify a query is to eliminate redundant predicates. Note that redundant queries are likely to arise when a query is the result of system transformations applied to the user query. Such transformations are used for performing semantic data control (views, protection, and semantic integrity control).

Fourth, the calculus query is restructured as an algebraic query**.** That several algebraic queries can be derived from the same calculus query, and that some algebraic queries are “better” than others. The quality of an algebraic query is defined in terms of expected performance. The traditional way to do this transformation toward a “better” algebraic specification is to start with an initial algebraic query and transform it in order to find a “good” one. The initial algebraic query is derived immediately from the calculus query by translating the predicates and the target statement into relational operators as

they appear in the query. This directly translated algebra query is then restructured through transformation rules. The algebraic query generated by this layer is good in the sense that the worse executions are typically avoided. For instance, a relation will be accessed only once, even if there are several select predicates. However, this query is generally far from providing an optimal execution, since information about data distribution and fragment allocation is not used at this layer.

DATA LOCALIZATION :

The input to the second layer is an algebraic query on global relations. The main role of the second layer is to localize the query's data using data distribution information in the fragment schema. We saw that relations are fragmented and stored in disjoint subsets, called fragments, each being stored at a different site. This layer determines which fragments are involved in the query and transforms the distributed query into a query on fragments. Fragmentation is defined by fragmentation predicates that can be expressed through relational operators. A global relation can be reconstructed by applying the fragmentation rules, and then deriving a program, called a localization program, of relational algebra operators, which then act on fragments. Generating a query on fragments is done in two steps. First, the query is mapped into a fragment query by substituting each relation by its reconstruction program (also called materialization program). Second, the fragment query is simplified and restructured to produce another "good" query. Simplification and restructuring may be done according to the same rules used in the decomposition layer. As in the decomposition layer, the final fragment query is generally far from optimal because information regarding fragments is not utilized.

GLOBAL QUERY OPTIMIZATION :

The input to the third layer is an algebraic query on fragments. The goal of query optimization is to find an execution strategy for the query which is close to optimal. Remember that finding the optimal solution is computationally intractable. An execution strategy for a distributed query can be described with relational algebra operators and *communication primitives* (send/receive operators) for transferring data between sites. The previous layers have already optimized the query, for example, by eliminating redundant expressions. However, this optimization is independent of fragment characteristics such as fragment allocation and cardinalities. In addition, communication operators are not yet specified. By permuting the ordering of operators within one query on fragments, many equivalent queries may be found.

Query optimization consists of finding the "best" ordering of operators in the query, including communication operators that minimize a cost function. The cost function, often defined in terms of time units, refers to computing resources such as disk space, disk I/Os, buffer space, CPU cost, communication cost, and so on. Generally, it is a weighted combination of I/O, CPU, and communication costs. Nevertheless, a typical simplification made by the early distributed DBMSs, as we mentioned before, was to consider communication cost as the most significant factor. This used to be valid for wide area networks, where the limited bandwidth made communication much more costly than local processing. This is not true anymore today and communication cost can be lower than I/O cost. To select the ordering of operators it is necessary to predict execution costs of alternative candidate orderings. Determining execution costs before query execution (i.e., static optimization) is based on fragment statistics and the formulas for estimating the cardinalities of results of relational operators. Thus the optimization decisions depend on the allocation of fragments and available statistics on fragments which are recorder in the allocation schema.

An important aspect of query optimization is join ordering, since permutations of the joins within the query may lead to improvements of orders of magnitude. One basic technique for optimizing a sequence of distributed join operators is through the semijoin operator. The main value of the semijoin in a distributed system is to reduce the size of the join operands and then the communication cost. However, techniques which consider local processing costs as well as communication costs may not use semijoins because they might increase local processing costs. The output of the query optimization layer is a optimized algebraic query with communication operators included on fragments. It is typically represented and saved (for future executions) as a distributed query execution plan.

DISTRIBUTED QUERY EXECUTION :

The last layer is performed by all the sites having fragments involved in the query. Each subquery executing at one site, called a local query, is then optimized using the local schema of the site and executed. At this time, the algorithms to perform the relational operators may be chosen. Local optimization uses the algorithms of centralized systems.

The goal of distributed query processing may be summarized as follows: given a calculus query on a distributed database, find a corresponding execution strategy that minimizes a system cost function that includes I/O, CPU, and communication costs. An execution strategy is specified in terms of relational algebra operators and communication primitives (send/receive) applied to the local databases (i.e., the relation fragments). Therefore, the complexity of relational operators that affect the performance of query execution is of major importance in the design of a query processor.

TRANSACTION AND COMPUTATION MODEL

- Page Model
- Object Model

Page Model

Syntax

Atransactiont is a partial order of steps (actions) of the formr(x) or w(x), where $x \in D$ and reads and writes as well as multiple writes applied to the same object are ordered.

We write t = (op, <),

for transaction t with step set op and partial order <.

Example:r(s) w(s) r(t) w(t)

Semantics

Interpretation of j^{th} step, p_j , of t:

If p_j =r(x), then interpretation is assignment v_j:= x to local variable v_j.

If p_j=w(x), then interpretation is assignment x := $f_j(v_{j1}, ..., v_{jk})$.

with unknown function $f_j$and j_1, ..., j_k denoting t‘s prior read steps.

Object Model

A transaction t is a (finite) tree of labeled nodes with

- the transaction identifier as the label of the root node,
- the names and parameters of invoked operations as labels of inner nodes, and
- page-model read/write operations as labels of leaf nodes, along with a partial order < on the leaf nodes such that for all leaf-node operations p and q with p of the form w(x) and q of the form r(x) or w(x) or vice versa, we have

$$p<q \quad \vee \quad q<p$$

Special case: layered transactions(all leaves have same distance from root)
Derived inner-node ordering: a < b ifall leaf-node descendants of a precede all leaf-node descendants of b
Example: DBS Internal Layers

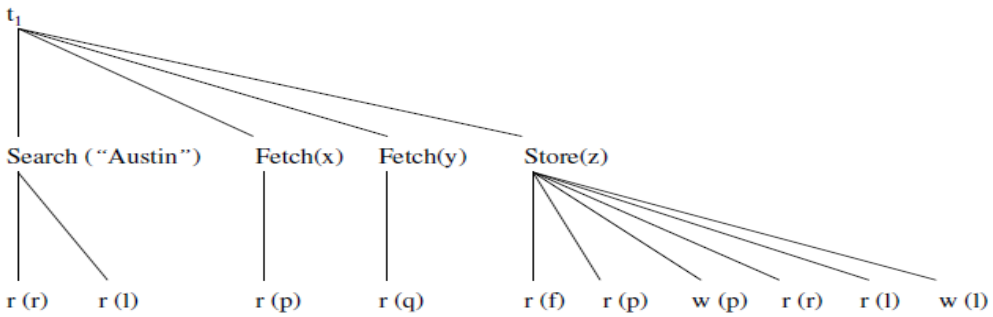


Figure : DBS Internal Layers

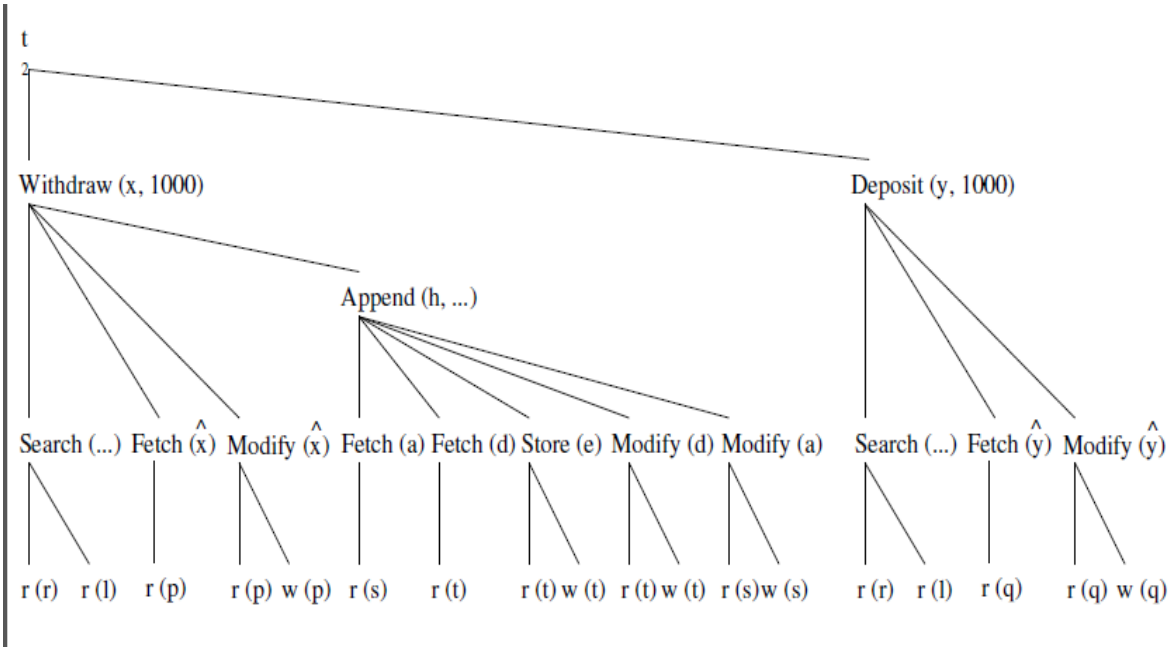


Figure: Business Objects

MULTIDATABASE CONCURRENCY CONTROL :

Concurrency controlin hierarchical MDBSs .In this section, we present a framework for the design of concurrency control mechanisms for hierarchical MDBSs. In a hierarchical MDBS, for the global schedule S to be serializable, the projection of S onto data items in each domain $D \in \Delta$ (that is, S D) must be serializable. However, as illustrated in the following example, ensuring serializability of S D, for each $D \in \Delta$, is not sufficient to ensure global serializability.

For example, in a schedule generated by a serialization-graph-testing (SGT) scheduler, it may not be possible to associate a serialization function with transactions. However, in such schedules, serialization functions can be introduced by forcing direct conflicts between transactions.

Let $\tau' \subseteq \tau$ be a set of transactions in a schedule S. If each transaction in τ' executed a conflicting operation (say a write operation on data item ticket) in S, then the functions that maps a transaction Ti $\in \tau'$ to its write operation on ticket is the serialization function for the transactions in S with respect to the set of transactions τ'. Associating serialization functions with global transactions makes the task of ensuring serializability of S D relatively simple. Since at each local DBMS the order in which transactions that are global with respect to the local DBMSs are serialized is consistent with the order in which their serSk operations execute, serializability of S D can be ensured by simply controlling the execution order of the serSk operations belonging to the transactions global with respect to the local DBMSs. To see how this can be achieved, for a global transaction Ti , let us denote its projection to its serialization function values over the local DBMSs as a transaction T˜ D i .

Formally, T˜ D i is defined as follows.

Definition 1. Let Ti be a transaction and D be a simple domain such that global(Ti , DBk), for some DBk, where child(DBk, D), T˜ D i is a restriction of Ti consisting of all the operations in the set {serSk (Ti) | Ti executes in DBk, and child(DBk, D)} Further, for the global schedule S, we define a schedule S˜D to be the restriction of S consisting of the set of operations belonging to transactions T˜ D i . Thus, S˜D = (τS˜D , $\prec$S˜D), where τS˜D = {T˜ D i | global(Ti , DBk) for some DBk, where child(DBk, D)}, and for all operations oq, or in S˜D, oq $\prec$S˜D or, iff oq $\prec$S or. In the schedule S˜D the conflict between operations is defined as follows:

Definition 2. Let S be a global schedule. Operations Sk (Ti) and Sl (Tj) in schedule S˜D, Ti / Tj , are said to conflict if and only if k = l. It is not too difficult to show that the serializability of the schedule S D can be ensured by ensuring the serializability of the schedule S˜D. Essentially, ensuring serializability of S˜D enforces a total order over global transactions (with respect to the local DBMSs), such that if Ti occurs before Tj in the total order, then serSk operation of Ti occurs before serSk operation of Tj for all sites sk at which they execute in common, thereby ensuring serializability of S D

Notice that operations in the schedule S˜D consist of only global transactions. Thus, since global transactions execute under the control of the MDBS software, the MDBS software can control the execution of the operations in S˜D to ensure its serializability, thereby ensuring serializability of S D. How this can be achieved – that is, how the MDBS software can ensure serializability of S˜D is a topic of the next section. Recall that the above-described mechanism for ensuring serializability of S D has been developed under the assumption that D is a simple domain. In the remainder of this section, we extend the mechanism suitably to ensure serializability of the schedule S D for an arbitrary domain D. One way we can extend the mechanism to arbitrary domains in hierarchical MDBSs is by suitably extending the notion of the serialization function to the set of domains.

Definition 3. Let D be any arbitrary domain in Δ. An extended serialization function is a function sf(Ti , D) that maps a given transaction Ti , and a domain D, to some operation of Ti that executes in D such that the following holds. For all Ti , Tj , if global(Ti , D), global(Tj , D), and Ti $*$ SD Tj , then sf(Ti , D) $\prec$SD sf(Tj , D). We refer to sf(Ti , D) as a serialization function of transaction Ti with respect to the domain D. To see how such a serialization function will aid us in ensuring serializability within a domain, consider a domain D /= DBk, k = 1, 2, . . . , m. To develop the intuition, let us assume that the above-defined serialization function exists for transactions in every child domain of D, that is, for every Dk, where child(Dk, D). If such a serialization function can be associated with the child domains, we can simply use the mechanism developed for simple domains to ensure serializability of S D.

We will, however, have to appropriately extend our definitions of the transaction T˜ D i , and the schedule S˜D with respect to the newly defined serialization function. This is done below.

Definition 4. Let Ti be a transaction and D be a domain such that global(Ti , Dk) for some Dk, where child(Dk, D). T˜ D i is a restriction of Ti consisting of all the operations in the set {sf(Ti , Dk) | Ti

executes in Dk, and child(Dk, D) }. As before, schedule S˜D is simply the schedule consisting of the operations in the transactions T˜ D i . That is, S˜D = (τS˜D , ≺S˜D), where τS˜D = {T˜ D i | global(Ti , Dk) for some Dk, where child(Dk, D)}, 158 and for all operations oq, or in S˜D, oq ≺S˜D or, iff oq ≺S or. Similar to the case of simple domain, two operations in S˜D, where D is an arbitrary domain, conflict if they are both serialization function values of different transactions over the same child domain.

Definition 5.Let S be a global schedule. Operations sf(Ti , Dk) and sf(Tj , Dl) in schedule S˜D, Ti / Tj , are said to conflict if and only if k = l. It it not difficult to see that similar to the case of simple domains, serializability of S D can be ensured, where D is an arbitrary domain, by ensuring the serializability of the schedule S˜D, under the assumption that, for all child domains Dk of D, the schedule S Dk is serializable and further a serialization function sf can be associated with transactions that are global with respect to Dk (see Lemma 1 in the appendix for a formal proof). In fact, this result can be applied recursively over the domain hierarchy to ensure serializability of the schedules S D for arbitrary domains D in hierarchical MDBSs. To see this, consider a hierarchical MDBS shown in Fig. 4. To ensure serializability of S D3 , it suffices to ensure serializability of the schedule S˜D3 , under the assumption that S D1 and S D2 are serializable and further that an appropriate serialization function sf can be associated with transactions that are global with respect to D1 and D2. In turn, serializability of S D1 (S D2) can be ensured by ensuring that the schedule S˜D1 (S˜D2) is serializable, under the assumption that S DB1 and S DB2 (S DB3 and S DB4) are serializable and further that an appropriate serialization function sf can be associated with transactions that are global with respect to DB1 and DB2 (DB3 and DB4). The recursion ends when D is a simple domain, since the child domains are local DBMSs and by assumption the schedule at each local DBMS is serializable. Thus, if we can associate an appropriate serialization function sf with transactions in each domain D ∈ Δ, we can ensure serializability of S D, by ensuring serializability of S˜D for all domains D ∈ Δ. Note that, for a domain D = DBk, the function sf is simply the function serSk introduced earlier. We now define the function sf for an arbitrary domain D ∈ Δ, which is done recursively over the domain ordering relation.

Definition 6. Let D be a domain and Ti be a transaction such that global(Ti , D). The serialization function for transaction Ti in domain D is defined as follows: sf(Ti , D) = serSk (Ti), if for some DBk, D = DBk. serS˜D (T˜ D i), if for all DBk, D /= DBk Let us illustrate the above definition of the serialization function using the following example. Example 3. Consider an MDBS environment consisting of local databases: DBMS1 with data item a, DBMS2 with data item b, DBMS3 with data item c, and DBMS4 with data item d. Let the domain ordering relation be as illustrated in Fig. 4. The set of domains: Δ = {DB1, DB2, DB3, DB4, D1, D2, D3}

MULTIDATABASE RECOVERY :

ReMT - A Recovery Strategy for MDBSs As already mentioned, reliability in MDBSs requires the design of two different types of protocols: commit and recovery protocols. A commit protocol which enforces commit atomicity of global transactions. In this section, we will present a strategy, called ReMT, for recovering multidatabase consistency after failures, without human intervention. In MDBSs, recovering multidatabase consistency has a twofold meaning. First, for global transaction aborts, recovering multidatabase consistency means to undo the effects of locally committed subsequences belonging to the aborted global transactions from a semantic point of view. In addition, the effects of transactions which have accessed objects updated by aborted global transactions should be preserved (recall that, after the last operation of a subsequence, all locks held by the subsequence are released). For the other types of failures, recovering multi database consistency means to restore the most recent global transaction-consistent state. We say that a multi database is in a global transaction-consistent state, if all local DBMSs the effects of locally-committed subsequences. The ReMT strategy consists of a collection of recovery protocols which are distributed among the components of an MDBS. Hence, some of them are performed by the GRM, some by the servers and some are provided by the LDBMSs. We assume that every participating LDBMS provides its own

recovery mechanism. Local recovery mechanisms should be able to restore the most recent transaction-consistent state of local databases after local failures. For each type of failure, we propose a specic recovery scheme. 6.1 Transaction Failures As seen before, we identify different kinds of transaction failures which may occur in a multidatabase environment. Each of them can be dealt with in a different manner. First, a particular global transaction may fail. This can be caused by a decision of the GTM or can be requested by the transaction itself. Second, a given subsequence of a global transaction may fail. In the following, we will propose recovery procedures to cope with failures of global transactions and subsequences.

A global transaction failure may occur for two reasons. The abort can be requested by the transaction or it occurs on behalf of the MDBS. The GTM can identify the reason which has caused the abort. This is because the GTM receives an abort operation from the transaction, whenever the abort is required by the transaction. We have observed that the recovery protocol for global transaction failures can be optimized if the following design decision is used: specific recovery actions should be dined for each situation in which a global transaction abort occurs. Therefore, we have designed recovery actions which should be triggered when the global transaction requires the abort, and recovery actions for coping with aborts which occur on behalf of the MDBS. Aborts Required by Transactions Since we assume that updates of a global transaction Gi may be viewed by other transactions, we cannot restore the database state which existed before the execution Gi, if Gi aborts. This implies that the standard transaction undo action cannot be used in such a situation. However, the effects of a global transaction must be somehow removed from the database, if it aborts. For that reason, we need a more adequate recovery paradigm for such an abort scenario. This new recovery paradigm should primarily focus on the fact that the effects of transactions which have accessed the objects updated by an aborted global transaction Gi and database consistency should be preserved, when removing the effects of Gi from the database. The key to this new recovery paradigm is the notion of compensating transactions. A compensating transaction CT \undoes" the effect of a particular transaction T from a semantic point of view. That means, CT does not restore the physical database state which existed before the execution of the transaction T . The compensation guarantees that a consistent (in the sense that all integrity constraints are preserved) database state is established based on semantic information, which is application-specific.

By definition, a compensating transaction CTi should be associated with a transaction Ti and may only be executed within the context of Ti. That means that the existence of CTi depends on Ti. In other words, CTi may only be executed, if Ti has been executed before. Hence, CTi must be serialized after Ti. We will assume that persistence of compensation is guaranteed, that is, once the compensating action has been started, it is completed successfully. For our purpose the concept of compensation is realized as follows. For a given transaction Gi consisting of subsequences SUBi;1, SUBi;2, :::, SUBi;n, a global compensating transaction CTi is dened, which in turn consists of a collection of local compensating transactions CTi;k, $0 < k$ n. Each local compensating transaction CTi;k is associated to the corresponding subsequence SUBi;k of transaction Gi. Of course, CTi;k must be performed at the same local site as does SUBi;k and must be serialized after SUBi;k. Now, we are in a position to describe the recovery strategy for aborts required by transactions. When the GS receives an abort request from a global transaction Gi, the GS forwards this operation to the GRM. The GRM reads the global log in order to identify which subsequences of Gi are still active. For each active subsequence, the GRM sends a local abort operation to the servers responsible for the execution of the subsequence. The GRM then waits for an acknowledgment from these servers conrming that the subsequences were aborted. After that, the GRM triggers the corresponding local compensating transactions for every subsequence which has already been locally committed. This

information can be retrieved from the global log le. Operations of the compensating transactions are scheduled by the GS. Therefore, the execution of local compensating transactions will undo the effect of committed subsequences from a semantic point of view. Since we have assumed that the LDBMSs implement 2PL to enforce local serializability, the compensation mechanism described above satisfies the following requirement. A particular transaction T (subsequence or local transaction) running at a local system either views a database state reacting the effects of an updating subsequence SUBi;k or it accesses a state produced by the compensating transaction of SUBi;k, namely CTi;k. In other words, T cannot access objects updated by SUBi;k and by CTi;k. Such a constraint is required for preserving local database consistency. Thus far, we have assumed that the effect of any transaction can be removed from the database by means of a compensating transaction. However, not all transactions are compensatable. There are some actions, classified by Gray as real actions ,which present the following property: once they are done, they cannot be undone anymore. For some of these actions, the user does not know how they can be compensated, that is, the semantic of such compensating transactions is unknow. For instance, the action ring a missile cannot be undone. Moreover, the semantic of a compensating transaction for this action cannot be defined. For that reason, we say that transactions involving such real actions are not compensatable. In order to overcome this problem, we propose the following mechanism. The execution of local commit operations for non-compensatable subsequences should be delayed until the GTM receives a commit for the global transaction containing the non-compensatable subsequences. This mechanism requires that the following two conditions are satisfied. First, the user should specify which subsequences of a global transaction are non-compensatable3 . 3When it is not specified that a subsequence is non-compensatable, it is assumed that the subsequence is compensatable. This is a reasonable requirement, since our recovery strategy relies on a compensation mechanism. This latter mechanism presumes that the user defines compensation transactions, when he or she is designing transactions. Hence, the user can identify at this point, which subsequences of a global transaction may not be compensatable. Second, the information identifying which subsequences are non-compensatable should be made available to the GTM. For instance, the GTM can be designed to receive this information as an input parameter of subsequences. The procedure of delaying the execution of local commit operations for non-compensatable subsequences can be realized according to the following protocol: 1. When the GTM receives the rst operation of a particular subsequence, it must identify whether the subsequence is compensatable. If the subsequence is non-compensatable, the GTM saves this information in the log record of the subsequence. The log record should be stored in the global log le. 2. If the GTM receives a local commit operation for a non-compensatable subsequence, it marks the log record of the subsequence stored in the global log with a ag. This ag captures the information that the local commit operation for the subsequence can be processed when the global transaction is to be committed. 3. Whenever the GTM receives a commit operation for a given global transaction Gi, it verifies in the global log if there are local commit operations to be processed for subsequences of Gi. This can be realized by reading the log records of all subsequences belonging to Gi. Following this protocol, we ensure that the
effects of non-compensatable subsequences are reacted in the local databases only when the global transaction is to be committed. This eliminates the possibility of undoing the effect of such subsequences. Unfortunately, this mechanism has the following disadvantage. Locks held by non-compensatable subsequences can only be released when the global transaction completes its execution. Another drawback of the compensation approach is the specification of compensating transactions for interactive transactions as, for instance, design activities. As a solution for overcoming such a problem, we propose the following strategy. When an interactive global transaction G has to be aborted and G has some locally committed subsequences, the GTM reads the global log le in order to identify which subsequences of G were already locally committed. After that,

the GTM notifies the user that the effects of some subsequences of G must be \manually" undone. The GRM informs which subsequences should be undone and what operations these subsequences have executed. Moreover, the GRM informs the user on which objects these operations have been performed. The user then starts another transaction in order to undo the effect of such subsequences. Objects updated by these subsequences may have been viewed by other global transactions. For that reason, the user must know which global transactions have read these objects. With this knowledge the user can notify other designers that the values of the objects x,y,z they have read (the GRM has provided this information) are invalid. Aborts on Behalf of the MDBS Usually, such aborts occur when global transactions are involved in deadlocks. Deadlocks are provoked by transactions trying to access the same objects with connecting locks. Committed subsequences have already released their locks. Besides this, they are not competing for locks anymore. Hence, operations of such subsequences can neither provoke nor be involved in deadlocks. This observation has an important impact in designing recovery actions to cope with transaction aborts required by the MDBS. It is not necessary to abort entire global transactions to resolve deadlock situations. Aborting active subsequences is sufficient. However, we need to replay the execution of the aborted subsequences in order to ensure commit atomicity. This implies that new results may be produced by the resubmission of the subsequences. In such a situation, the user must be noticed that the subsequences were aborted and, for that reason, they must be replayed, which may produce different results from those he/she has already received. With this knowledge, the user can decide to accept the new results or to abort the entire global transaction. Observe that, if the original values read by the failed subsequences were not communicated to other subsequences (those reads may be invalid), the resubmission of the aborted subsequences will produce no inconsistency in the execution of entire global transaction. Such a requirement is reasonable in a multidatabase environment. Based on these observations, we propose the following strategy for dealing with global transaction aborts which occur on behalf of the MDBS. When the GTM (or another component of the MDBS) decides to abort a transaction Gi, the GRM must be informed that Gi has to be aborted. When the GRM receives this signal, it verifies in the global log which subsequences of Gi are still active. For each active subsequence, the GRM sends a local abort operation to the servers (through the GS, of course) responsible for the submission of these subsequences to the local systems. In the meantime, the GRM waits for an acknowledgment from the servers confrming the local aborts of the subsequences. Furthermore, the GRM sends to the user responsible for the execution of Gi the notification informing that some subsequences of Gi have to be aborted and they will be replayed. The GRM is able to inform the user which operations have to be reexecuted. The user can then decide to wait for the resubmission of the aborted subsequences or to abort the entire global transaction. If the user decides to abort the entire global transaction, the process of replaying the subsequences is cancelled and the recovery protocol for global transaction failure requested by the transaction is triggered. Otherwise, the recovery protocol for global transaction aborts which occur on behalf of the MDBS goes on as described below. When the GRM has received the acknowledgments that the subsequences were aborted in the local DBMSs, the GRM starts to replay the execution of each aborted subsequence SUBi;k. For that purpose, the GRM must read from the global log le the log record which contains information about the installation point of each subsequence to be replayed. This record can be identified by the elds SUBID and LRT. Observe that LRT must have the value `IP'.

As mentioned before, a subsequence of a particular global transaction may abort for many reasons. However, there are two situations of subsequence aborts which should be handled in a different manner. The rest situation is when the subsequence is aborted on behalf of the local DBMS. The second situation is when the subsequence decides to abort its execution. In this section, we describe a recovery method to deal with these two subsequence abort situations. Aborts on Behalf of the Local

DBMS Typically, DBMSs decide to abort subsequences, when such subsequences are involved in local deadlocks. After such aborts, the effect of failed subsequences are undone by the LDBMSs. Locks held by the aborted subsequences are released. As soon as the server recognizes that a particular subsequence has been aborted by the local DBMS, the server reads the server log le and retrieves the log records of the aborted subsequence. The server stores a new log record for the subsequence with LRT=`ST' in the server log le. Moreover, the server sends a message to the GTM reporting that the subsequence has been aborted by the LDBMS. The GRM forces a record log of the failed subsequence to the global log le. By doing this, the new state of the subsequence is stored in the global log le as well. After that, the server forces a log record with the new state of the subsequence to the server log and starts the resubmission of the aborted subsequence. As already seen, new results may be produced by such a resubmission. However, we propose a notification mechanism which gives the user the necessary support to decide for accepting the new results or for aborting the entire global transaction. It is important to notice here that a given subsequence SUBi;k belonging to a global transaction Gi may have more than one log record with LRT=`ST' (in each log le) during the execution of Gi. For such a subsequence, only the last record with LRT=`ST' should be considered. Aborts Required by Subsequences When the subsequence identifies some internal error condition (e.g., violation of some integrity constraints or bad input), it aborts its execution. Sometimes the resubmission of the subsequence is sufficient to overcome the error situation. However, we cannot guarantee that the subsequence will be committed after being resubmitted a certain number of times.

This is because the abort is caused when some internal error condition occurs (e.g. division by 0). Hence, it is impossible to predict whether or not the same problem will occur in a repeated execution of the subsequence. In this case, the solution is to abort the complete global transaction. The user or the GTM should be able to make such a decision. Observe that, when an internal error occurs, it is necessary that the subsequence reads new values (new input) and produces new results in order to overcome the internal error condition. Based on this observation, we propose the following actions for dealing with aborts required by subsequences. When the subsequence decides to abort its execution, an explicit abort operation is submitted to the GS, which in turn sends this operation to the GRM. The GRM then writes a new log record with LRT=`ST' for the subsequence in order to re ect its new state. Thereafter, the GRM forwards the abort operation to the server. In turn, the server forces a log record with the new state of the subsequence to the server log and submits the abort operation to the LDBMS. After the subsequence is aborted by the LDBMS, the GTM resubmits the aborted subsequence to the LDBMS. 6.2 Local System Failures Local DBSs reside in heterogeneous and autonomous computer systems (sites). When a system failure occurs at a particular site, we assume that the LDBMS is able to perform recovery actions in order to restore the most recent transaction-consistent state. These actions are executed outside the control of the MDBS. After an LDBMS completes the recovery actions, the interface server assumes the control of the recovery processing. While the server is executing its recovery actions, no local transaction can be submitted to the restarted DBMS. Before describing the strategy for recovering from local system failures, we need to defined states of a subsequence in a given server. A subsequence may present four different states in a server. A subsequence is said to be active, when no termination operation for the subsequence has been submitted to the local DBMS by the server. When the server submits a commit operation, the subsequence enters the to-be-committed state. If the commit operation submitted by the server has been successfully executed by the local DBMS, the subsequences enters the locally-committed state. When the subsequence aborts, it enters the locally-aborted state.

We assume that the GTM can identify when a given server has failed. The protocol for handling server failures is the following:

1.When the GTM recognizes that a server has failed, it aborts the execution of all active subsequences which were being executed in the failed server. Log records (with LRT=`ST') for the aborted subsequences are forced to the global log le in order to store the information that these subsequences have passed from the active to the aborted state. Moreover, the GTM stops submitting operations to that server. In order to decide what kind of recovery actions should be performed for to-be-committed subsequences, the GTM must wait until the server has been restarted, since the GTM must know whether the subsequence was successfully committed by the local DBMS.

2. After the server is restarted, it should trigger the following recovery procedures:

(a) The server log is sequentially read. For each subsequence which was active immediately before the occurrence of the failure, the server sends an abort operation to the local DBMS. If the subsequence was to-be-committed, the server may query the external interface of the local DBMS in order to know whether or not the subsequence was successfully committed by the local DBMS. The server then forwards this information to the GTM.

(b) The server log must be updated. For instance, if a particular to-be-committed subsequence was aborted by the local DBMS before the occurrence of the failure, the server writes a record in the server log le in order to capture this information.

(c) After the server log is read and updated, the server sends a message to the GTM informing that it is in operation. 3. When the GRM receives a message from the server reporting that it is operational, the GRM replays the aborted subsequences. After that, the recovery procedure for server failure is completed.

Communication Failures The components of an MDBS are interconnected via communication links. Typically, communication failures break the communication among some of the components of an MDBS. According to Figure 2, there may be two types of communication links in MDBSs. One type of link, which we call Server-LDBS link, connects servers to local systems. If the interface servers are not integrated with the GTM, that is, each server resides at a different site from the GTM site, the other type of link connects the GTM to servers. Such a communication link is denoted GTM-Server link. We propose different recovery strategies for handling failures in each type of communication link. In order to enable MDBSs to cope with communication failures, the following requirement must be satisfied. Each server in an MDBS must know the timeout period of the local DBMS with which the server is associated. We also assume that each server has its own timeout period and this timeout period is larger than the timeout period of the respective local DBMS. Failures in Server-LDBS links In such failures, the link between a particular local system and a server is broken. The local system and the server will continue to work correctly. Such a situation can lead the local system to abort the execution of some subsequences (which are being executed at the local system) by timeout. For coping with communication failures between a server and a local system, we propose the following strategy. If the communication link is reestablished before the timeout period of the local DBMS is reached, no recovery action is necessary. This is because no subsequence was aborted by timeout. In the case that the communication link is reestablished after the timeout period of the local DBMS is reached, but before the timeout period of the server, the following recovery actions should be performed by the server: 1. The server scans the server log le. During the scan process, the following recovery actions should be performed.

(a) For each subsequence which was active before the occurrence of the failure, the server executes recovery actions, since such subsequences were aborted by the LDBMS by timeout. These recovery

actions are the same as those which should be performed for recovering from subsequence failures required by LDBMSs

(b) If the subsequence was to-be-committed, the server may query the external interface of the LDBMS in order to know whether the subsequence was successfully committed. In this case, the server performs actions to confirm the fact that the subsequence was committed (for instance, log records with LRT=`ST' must be forced to the server log and global log les). Otherwise, it considers the subsequence as locally aborted and performs actions for recovering from subsequence failures required by LDBMSs. If the timeout period of the server is reached before the communication link is reestablished, the server sends a message to the GTM reporting that it cannot process subsequences anymore. After that, the GTM aborts the execution of all subsequences which were being executed in the failed server. Log records for the aborted subsequences are stored in the global log le with their new state (aborted). The GTM stops submitting operations to that server. If the communication link is reestablished before the timeout period of the GTM is reached, recovery actions for recovering from server failures are executed. If the timeout period of the GTM is reached before the Server-LDBS link is reestablished, the global log le is sequentially read. For each global transaction which has submitted a subsequence to the server whose Server-DBMS link is broken, the subsequence's log record with LRT=`ST' is read. If the subsequence is active or to-be-committed, the GRM aborts the global transaction. In this case, recovery actions for global transaction recovery should be triggered. Observe that a subsequence which was submitted to the server with a broken Server-LDBS link and has a to-be-committed state in the global log may have been committed by the local DBMS. In this case, after the link is reestablished, the server must be able to query the external interface of the LDBMS to know whether or not the subsequence was successfully committed. If the subsequence was committed, a compensating transaction for such a subsequence should be executed. Failures in GTM-Server links Of course, such a failure has only to be considered, if it is assumed that the interface servers reside at different sites from the GTM's site. When a failure in the GTM-Server link occurs, the link between the GTM and a server is broken. In order to enable MDBSs to cope with failures in GTM-Server links, we propose the strategy described below. Without loss of generality, consider that the link between the GTM and the server Serverk is broken. Serverk is associated with local system LDBSk. If the communication link is reestablished after the timeout period of LDBSk is reached, but before the timeout period of the server, the following actions are performed: The server log is sequentially read.

1. For each subsequence which was active before the failure, the server executes recovery actions for subsequence failures required by local DBMSs, since such transactions were aborted by the local DBMS (timeout).

2. If the subsequence was to-be-committed, the server may query the external interface of the local DBMS in order to know whether the subsequence was committed. In this case, the server performs the actions to react the fact that the subsequence was locally committed. Otherwise, it performs actions for recovering from subsequence failures required by local DBMSs. If the link is reestablished after the timeout period of Serverk, but before the timeout period of the GTM is reached, actions for recovering from server failures are started. If the timeout period of the GTM is reached before the link is reestablished, the GRM reads the global log in order to identify active global transactions which have submitted a subsequence to Serverk whose link with the GTM is broken. For each global transaction satisfying this condition, the GRM verifies the state of the subsequence submitted to Serverk. If the subsequence was active or to-be-committed, the GRM aborts the global transaction. Recovery actions for global transaction recovery should be triggered. A subsequence which has a to-be-committed state in the global log may have been committed by the local DBMS. In this case, after

the communication link is reestablished, the server must be able to query the external interface of the LDBMS in order to know whether or not the subsequence was successfully committed. If the subsequence was committed, a compensating transaction for such a subsequence should be executed.

OBJECT ORIENTATION AND INTEROPERABILITY:

Interoperating applications are often developed independently of each other in environments that may differ in the following dimensions:

• Locations

• Machine architectures

• Operating systems

• Programming languages

• Models of information. Applications can interoperate along the following dimensions:

• "Horizontal" peer-to-peer sharing of services and information, such as an editor invoking a spreadsheet processor to embed a spreadsheet in a document.

• "Vertical" cascading through levels of implementation. A student registration service may use a database service which in turn uses a file manager which uses a device driver.

• "Time-line" through the life cycle of an application. Enterprise modeling may be done in terms of one set of constructs which are translated into constructs of the application programming language which are compiled into constructs of the run-time environment. Or, a graphical language used to capture a user's conceptual model of a business domain is translated into a computer-executable simulation language, with the results of the simulation then being input either to an analysis tool to allow refinement of the simulation, or to a report generator to produce the final result. Internal Accession Date Only 2

• Others, e.g., the "viewpoints" of the ISO/CCITT Reference Model for Open Distributed Processing (RM-ODP) – Enterprise viewpoint – Information viewpoint – Computational viewpoint – Engineering viewpoint – Technology viewpoint Interoperation is concerned with such things as:

• Application interconnection: – Finding services and information in a distributed environment. – Coping with operational differences between requesters and providers of services, such as interface/communication protocols, synchronization, exception handling, work coordination, resource management, etc.

• Information compatibility.

OMA (OBJECT MANAGEMENT ARCHIRECTURE):

OMA is an architecture developed by the OMG (Object Management Group) that provides an industry standard for developing object-oriented applications to run on distributed networks. The goal of the OMG is to provide a common architectural framework for object-oriented applications based on widely available interface specifications.

The OMA reference model identifies and characterizes components, interfaces, and protocols that comprise the OMA. It consists of components that are grouped into application-oriented interfaces,

industry-specific vertical applications, object services, and ORBs (object request brokers). The ORB defined by the OMG is known more commonly as CORBA (Common Object Request Broker Architecture).

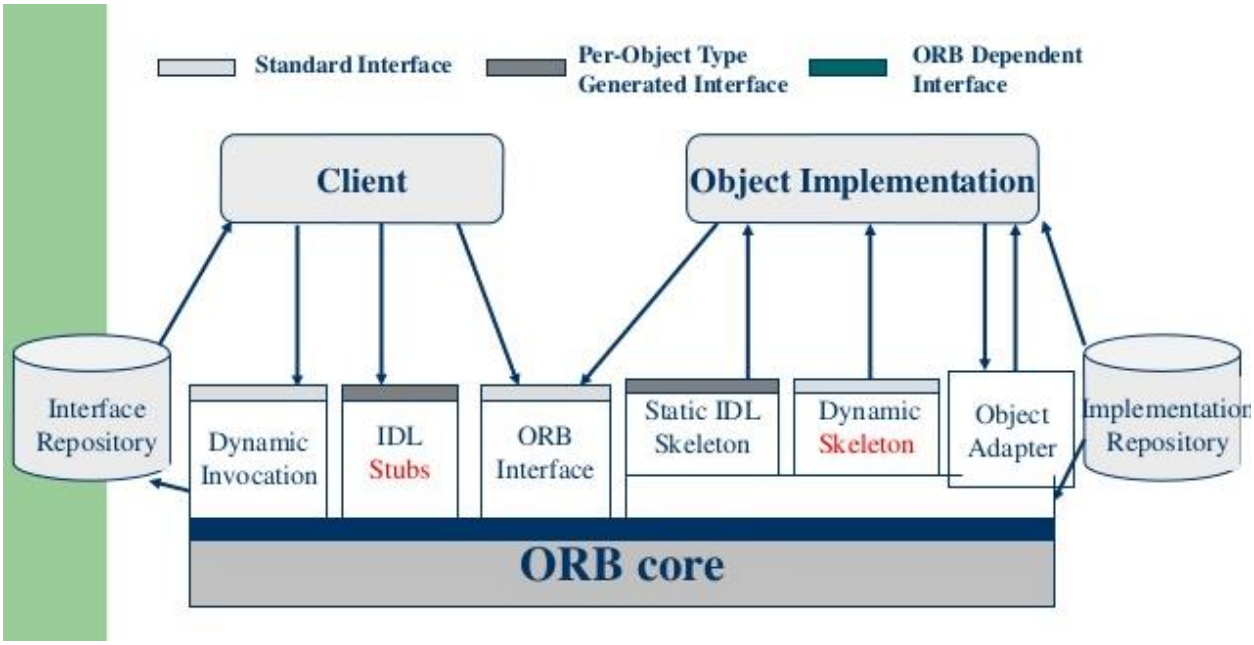
The Common Object Request Broker Architecture (CORBA) is a specification developed by the Object Management Group (OMG). CORBA describes a messaging mechanism by which objects distributed over a network can communicate with each other irrespective of the platform and language used to develop those objects.

There are two basic types of objects in CORBA. The object that includes some functionality and may be used by other objects is called a service provider. The object that requires the services of other objects is called the client. The service provider object and client object communicate with each other independent of the programming language used to design them and independent of the operating system in which they run. Each service provider defines an interface, which provides a description of the services provided by the client.

CORBA enables separate pieces of software written in different languages and running on different computers to work with each other like a single application or set of services. More specifically, CORBA is a mechanism in software for normalizing the method-call semantics between application objects residing either in the same address space (application) or remote address space (same host, or remote host on a network).

CORBA applications are composed of objects that combine data and functions that represent something in the real world. Each object has multiple instances, and each instance is associated with a particular client request. For example, a bank teller object has multiple instances, each of which is specific to an individual customer. Each object indicates all the services it provides, the input essential for each service and the output of a service, if any, in the form of a file in a language known as the Interface Definition Language (IDL). The client object that is seeking to access a specific operation on the object uses the IDL file to see the available services and marshal the arguments appropriately.

The CORBA specification dictates that there will be an object request broker (ORB) through which an application interacts with other objects. In practice, the application simply initializes the ORB, and accesses an internal object adapter, which maintains things like reference counting, object (and reference) instantiation policies, and object lifetime policies. The object adapter is used to register instances of the generated code classes. Generated code classes are the result of compiling the user IDL code, which translates the high-level interface definition into an OS- and language-specific class base to be applied by the user application. This step is necessary in order to enforce CORBA semantics and provide a clean user process for interfacing with the CORBA infrastructure.



DISTRIBUTED COMPONENT MODEL:

DCOM is a programming construct that allows a computer to run programs over the network on a different computer as if the program was running locally. DCOM is an acronym that stands for Distributed Component Object Model. DCOM is a proprietary Microsoft software component that allows COM objects to communicate with each other over the network.

An extension of COM, DCOM solves a few inherent problems with the COM model to better use over a network:

Marshalling: Marshalling solves a need to pass data from one COM object instance to another on a different computer – in programming terms, this is called "passing arguments."

For example, if I wanted Zaphod's last name, I would call the COM Object LastName with the argument of Zaphod. The LastName function would use a Remote Procedure Call (RPC) to ask the other COM object on the target server for the return value for LastName(Zaphod), and then it would send the answer – Beeblebrox – back to the first COM object.

Distributed Garbage Collection: Designed to scale DCOM in order to support high volume internet traffic, Distributed Garbage Collection also addresses away to destroy and reclaim completed or abandoned DCOM objects to avoid blowing up the memory on webservers. In turn, it communicates with the other servers in the transaction chain to let them know they can get rid of the objects related to a transaction.**Using DCE/RPC as the underlying RPC mechanism:** To achieve the previous items and to attempt to scale to support high volume web traffic, Microsoft implemented DCE/RPC as the underlying technology for DCOM – which is where the D in DCOM came from.

How Does DCOM Work?

In order for DCOM to work, the COM object needs to be configured correctly on both computers – in our experience they rarely were, and you had to uninstall and reinstall the objects several times to get them to work.

The Windows Registry contains the DCOM configuration data in 3 identifiers:

CLSID – The Class Identifier (CLSID) is a Global Unique Identifier (GUID). Windows stores a CLSID for each installed class in a program. When you need to run a class, you need the correct CLSID, so Windows knows where to go and find the program.

PROGID – The Programmatic Identifier (PROGID) is an optional identifier a programmer can substitute for the more complicated and strict CLSID. PROGIDs are usually easier to read and understand. A basic PROGID for our previous example could be Hitchiker.LastName. There are no restrictions on how many PROGIDs can have the same name, which causes issues on occasion.

APPID – The Application Identifier (APPID) identifies all of the classes that are part of the same executable and the permissions required to access it. DCOM cannot work if the APPID isn't correct. You will probably get permissions errors trying to create the remote object, in my experience.

A basic DCOM transaction looks like this:

The client computer requests the remote computer to create an object by its CLSID or PROGID. If the client passes the APPID, the remote computer looks up the CLSID using the PROGID.

The remote machine checks the APPID and verifies the client has permissions to create the object.

DCOMLaunch.exe (if an exe) or DLLHOST.exe (if a dll) will create an instance of the class the client computer requested.

Communication is successful!

The Client can now access all functions in the class on the remote computer.

If the APPID isn't configured correctly, or the client doesn't have the correct permissions, or the CLSID is pointing to an old version of the exe or any other number of issues, you will likely get the dreaded "Can't Create Object" message.

DCOM vs. CORBA

Common Object Request Broker Architecture (CORBA) is a JAVA based application and functions basically the same as DCOM. Unlike DCOM, CORBA isn't tied to any particular Operating System (OS), and works on UNIX, Linux, SUN, OS X, and other UNIX-based platforms.

Neither proved secure or scalable enough to become a standard for high volume web traffic. DCOM and CORBA didn't play well with firewalls, so HTTP became the default standard protocol for the internet.