17CAP405D DISTRIBUTED DATABASE MANAGEMENT SYSTEM 4H -4C

Instruction Hours / week: L: 4 T: 0 P: 0 C:4 Marks: Internal: 40 External: 60 Total: 100

End Semester Exam: 3 Hours

Scope: The Scope of this course is to expose the students to the architecture, design, and implementation of massive-scale data systems. The course also discusses foundational concepts of distributed database theory including design and architecture, security, integrity, query processing and optimization, transaction management, concurrency control, and fault tolerance.

Objectives: To impart necessary skills to students

- To design good performing distributed database schemas.
- To create optimized query execution plan.
- To efficiently distribute and manage the data.
- To manage distributed access control
- To know how to make secure the databases.

UNIT - I

Database concepts: Data Models- Database Operations- Database Management-DB Clients, Servers, and Environments. DBE Architecture**:** Services- Components and Subsystems- Sites - Expected Services-Expected Subsystems- Typical DBMS Services– DBE Taxonomy**:** COS Distribution and Deployment- COS Closeness or Openness-Schema and Data Visibility-Schema and Data Control.

UNIT - II

Data Distribution Alternatives: Design Alternatives- Localized Data- Distributed Data. Fragmentation: Vertical Fragmentation- Horizontal Fragmentation. Distribution Transparency: Location Transparency-Fragmentation Transparency-Replication Transparency-Location, Fragmentation, and Replication Transparencies.

UNIT - III

Query Optimization : Sample Database- Query Processing in Centralized Systems: Query Parsing and Translation - Query Optimization- Query Processing in Distributed Systems-Heterogeneous Database Systems - Concurrency Control in Distributed Database Systems.

UNIT - IV

Deadlock Handling: Deadlock Definition- Deadlocks in Centralized Systems- Deadlocks in Distributed Systems- Distributed Deadlock Detection. Replication Control: Replication Control Scenarios. Failure and Commit Protocols: Terminology- Commit Protocols.

UNIT - V

DDBE Security: Cryptography- Securing Data . Traditional DDBE Architectures: Classifying the Traditional DDBMS Architecture- The MDBS Architecture Classifications- Approaches for Developing A DDBE- Deployment of DDBE Software.

SEMESTER-IV

17CAP405D DISTRIBUTED DATABASE MANAGEMENT SYSTEM 4H -4C

SUGGESTED READINGS

1. Saeed K. Rahimi And Frank S. Haug.(2010),“Distributed Database Management Systems :A Practical Approach.”, 1st Edition, A John Wiley & Sons, Inc., Publication.
2. Ceri.(1985),Distributed Databases Principles and Systems , 1st Edition Mchraw Hill Pub.
3. Tamer Ozus M,Patrick Valduriez,S.Sridhar(2006), Principle Of Distributed Database Systems, 1st Edition , Pearson Education.
4. William M.NewMan, Robort F.Sproull (2004), Principles of Interactive Computer Graphics, 1st Edition , Pearson Education.

WEB SITES

1. en.wikipedia.org/wiki/Distributed_computing
2. www.webopedia.com/TERM/D/distributed_computing.html

3.www.tech-faq.com/distributed-computing.shtml

4.http://www.inf.unibz.it/dis/teaching/DDB/ln/ddb01.pdf

Enable | Enlighten | Enrich
KARPAGAM
ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)

Karpagam Academy of Higher Education

(Established Under Section 3 of UGC Act 1956)
Eachanari Post, Coimbatore – 641 021. INDIA
Phone: 0422-2611146, 2611082 Fax No: 0422 -2611043

17CAP405D – DISTRIBUTED DATABASE MANAGEMENT SYSTEM

LECTURE PLAN

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
		UNIT I	
1	1	Database concepts: Data Models Database Operations Database Management	T1: 1 - 3 R1: 41 – 58,W1,J1
2	1	DB Clients, Servers, and Environments.	T1: 3 – 4, W2
3	1	DBE Architecture: Services- Components and Subsystems- Sites	T1: 4 - 6
4	1	Expected Services-Expected Subsystems, Typical DBMS Services	T1: 8 – 11 W1
5	1	DBE Taxonomy COS Distribution and Deployment	T1: 13 - 14, W1
6	1	COS Closeness or Openness Schema and Data Visibility	T1: 15-16,J1
7	1	Schema and Data Control	T1: 17, J1
8	1	Recapitulation and Discussion of important Questions	-
		Total No. of Hours planned for Unit I	**8 hrs**

TEXT BOOK:

T1: Saeed K. Rahimi And Frank S. Haug.(2010),"Distributed Database Management Systems :A Practical Approach.", 1st Edition, A John Wiley & Sons, Inc., Publication

REFERENCE BOOK:

R1: Ceri.(1985),Distributed Databases Principles and Systems , 1st Edition Mchraw Hill Pub

WEBSITES:

W1: http://www.tutorialspoint.com

W2: http://www.studytonight.com

JOURNAL:

J1: Journal of Database Management

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
		UNIT II	
1	1	Data Distribution Alternatives: Design Alternatives	T1: 35 – 38, W1, J1
2	1	Localized Data- Distributed Data	T1: 38 - 39
3	1	Fragmentation: Vertical Fragmentation Horizontal Fragmentation	T1: 39 – 47, R1: 81 – 112,W2
4	1	Distribution Transparency: Location Transparency	T1: 68 - 69
5	1	Fragmentation Transparency Replication Transparency	T1: 68,W1
6	1	Location Transparency	T1: 69
7	1	Replication Transparencies - Location	T1: 70,J1
8	1	Recapitulation and Discussion of important Questions	-
Total No. of Hours planned for Unit II			**8 hrs**

TEXT BOOK:

T1: Saeed K. Rahimi And Frank S. Haug.(2010),"Distributed Database Management Systems :A Practical Approach.", 1st Edition, A John Wiley & Sons, Inc., Publication

REFERENCE BOOK:

R1: Ceri.(1985),Distributed Databases Principles and Systems , 1st Edition Mchraw Hill Pub

WEBSITES:

W1: http://www.tutorialspoint.com

W2: http://www.studytonight.com

JOURNAL:

J1: Journal of Database Management

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
UNIT III			
1	1	Query Optimization : Sample Database Query Processing in Centralized Systems	T1: 111 – 112, R1: 246 - 249 T1: 126 – 127, W1
2	1	Query Parsing and Translation Query Optimization	T1: 127 - 130 J1,W2
3	1	Cost Optimization Plan generation	T1: 131 - 135
4	1	Dynamic Programming Reducing the solution space	T1: 135 – 144 J1
5	1	Query Processing in Distributed Systems	T1: 145 – 146, W2
6	1	Heterogeneous Database Systems	T1: 170 – 172, R1: 307 - 314
7	1	Concurrency Control in Distributed Database Systems	T1: 222 – 237,W1
8	1	Recapitulation and Discussion of important Questions	-
Total No. of Hours planned for Unit III			**8 hrs**

TEXT BOOK:

T1: Saeed K. Rahimi And Frank S. Haug.(2010),"Distributed Database Management Systems :A Practical Approach.", 1st Edition, A John Wiley & Sons, Inc., Publication

REFERENCE BOOK:

R1: Ceri.(1985),Distributed Databases Principles and Systems , 1st Edition Mchraw Hill Pub

WEBSITES:

W1: http://www.tutorialspoint.com

W2: http://www.studytonight.com

JOURNAL:

J1: Journal of Database Management

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
		UNIT IV	
1.	1	Deadlock Handling: Deadlock Definition- Deadlocks in Centralized System	T1: 247 – 252, W2 J1
2.	1	Deadlocks in Distributed Systems Distributed Deadlock Detection	T1: 252 - 257 T1: 260 – 265, R1: 391 – 394, W1
3.	1	Replication Control: Replication Control Scenarios. Failure and Commit Protocols: Terminology-	T1: 275 – 279, T1: 297 – 298
4.	1	Commit Protocols: commit point, Transaction Rollback	T1: 293 – 300
5.	1	Transaction Roll forward, transaction scenario	T1: 300 – 302
6.	1	Database Update modes Transaction Log, Log contents	T1: 302 - 305
7.	1	DBMS storage types	T1: 305 - 308
8.	1	Recapitulation and Discussion of important Questions	-
Total No. of Hours planned for Unit IV			**8 hrs**

TEXT BOOK:

T1: Saeed K. Rahimi And Frank S. Haug.(2010),“Distributed Database Management Systems :A Practical Approach.”, 1st Edition, A John Wiley & Sons, Inc., Publication

REFERENCE BOOK:

R1: Ceri.(1985),Distributed Databases Principles and Systems , 1st Edition Mchraw Hill Pub

WEBSITES:

W1: http://www.tutorialspoint.com

W2: http://www.studytonight.com

JOURNAL:

J1: Journal of Database Management

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
		UNIT V	
1.	1	DDBE Security : Cryptography	T1: 357 – 366, R1: 7 – 15, W2
2.	1	Securing Data : Authentication and authorization, Data Encryption.	T1: 368 – 370,W1,J1
3.	1	Unvalidated Input and SQL Injection, Data Inference, Data Auditing.	T1: 370 – 375,W1
4.	1	Traditional DDBE Architectures:	T1: 453 – 454, W1
5.	1	Classifying the Traditional DDBMS Architecture	T1: 453 – 454,J1
6.	1	The MDBS Architecture Classifications	T1: 457 – 459
7.	1	Approaches for Developing A DDBE	T1: 459 – 460
8.	1	Deployment of DDBE Software.	T1: 461 – 463,W1,J1
9.	1	Recapitulation and Discussion of important Questions	-
10.	1	Discussion of previous year ESE Question paper	-
11.	1	Discussion of previous year ESE Question paper	-
12.	1	Discussion of previous year ESE Question paper	-
Total No. of Hours planned for Unit V			**12 hrs**

TEXT BOOK:

T1: Saeed K. Rahimi And Frank S. Haug.(2010),“Distributed Database Management Systems :A Practical Approach.”, 1st Edition, A John Wiley & Sons, Inc., Publication

REFERENCE BOOK:

R1: Ceri.(1985),Distributed Databases Principles and Systems , 1st Edition Mchraw Hill Pub

WEBSITES:

W1: http://www.tutorialspoint.com

W2: http://www.studytonight.com

JOURNAL:

J1: Journal of Database Management

UNIT-1

SYLLABUS

Database concepts: Data Models- Database Operations- Database Management-DB Clients, Servers, and Environments. DBE Architecture**:** Services- Components and Subsystems- Sites - Expected Services-Expected Subsystems- Typical DBMS Services**–** DBE Taxonomy**:** COS Distribution and Deployment- COS Closeness or Openness-Schema and Data Visibility- Schema and Data Control.

Distributed Database: relating to, or being a computer network in which at least some of the processing is done by the individual workstations and information is shared by and often stored at the workstations.

Database: usually large collection of data organized especially for rapid search and retrieval (as by a computer).

Informally speaking, a database (DB) is simply a collection of data stored on a computer, and the term distributed simply means that more than one computer might cooperate in order to perform some task. Most people working with distributed databases would accept both of the preceding definitions without any reservations or complaints. Unfortunately, achieving this same level of consensus is not as easy for any of the other concepts involved with distributed databases (DDBs). A DDB is not simply “more than one computer cooperating to store a collection of data”—this definition would include situations that are not really distributed databases, such as any machine that contains a DB and also mounts a remote file system from another machine. Similarly, this would be a bad definition because it would not apply to any scenario where we deploy a DDB on a single computer. Even when a DDB is deployed using only one computer, it remains a DDB because it is still possible to deploy it across multiple computers. Often, in order to discuss a particular approach for implementing a DB, we need to use more restrictive and specific definitions. This means that the same terms might have conflicting definitions when we consider more than one DB implementation alternative. This can be very confusing when researching DBs in general and especially confusing when focusing on DDBs.

KAHE

DATABASE CONCEPTS

- Whenever we use the term "DB" in this book, we are always contemplating a collection of persistent data. This means that we "save" the data to some form of secondary storage (the data usually written to some form of hard disk). As long as we shut things down in an orderly fashion (following the correct procedures as opposed to experiencing a power failure or hardware failure), all the data written to secondary storage should still exist when the system comes back online. We can usually think of the data in a DB as being stored in one or more files, Based on the markup, the text would be styled using the appropriate typeface, point size, weight, etc.

possibly spanning several partitions, or even several hard disk drives—even if the data is actually being stored in something more sophisticated than a simple file.

DATA MODELS

Every DB captures data in two interdependent respects; it captures both the data structure and the data content. The term "data content" refers to the values actually stored in the DB, and usually this is what we are referring to when we simply say "data." The term "data structure" refers to all the necessary details that describe how the data is stored. This includes things like the format, length, location details for the data, and further details that identify how the data's internal parts and pieces are interconnected. When we want to talk about the structure of data, we usually refer to it as the data model (DM) (also called the DB's schema, or simply the schema). Often, we will use a special language or programmatic facility to create and modify the DM. When describing this language or facility, authors sometimes refer to the facility or language as "the data model" as well, but if we want to be more precise, this is actually the data modeling language (ML)—even when there is no textual language. The DM captures many details about the data being stored, but the DM does not include the actual data content. We call all of these details in the DM metadata, which is informally defined as "data about data" or "everything about data except the content itself."

DATABASE OPERATIONS

KAHE

Usually, we want to perform several different kinds of operations on DBs. Every DB must at least support the ability to "create" new data content (store new data values in the DB) and the ability to retrieve existing data content. After all, if we could not create new data, then the DB would always be empty! Similarly, if we could not retrieve the data, then the data would serve no purpose. However, these operations do not need to support the same kind of interface; for example, perhaps the data creation facility runs as a batch process but the retrieval facility might support interactive requests from a program or user. We usually expect newer DB software to support much more sophisticated operations than minimum requirements dictate. In particular, we usually want the ability to update and delete existing data content. We call this set of operations CRUD (which stands for "create, retrieve, update, and delete"). Most modern DBs also support similar operations involving the data structures and their constituent parts. Even when the DBs support these additional "schema CRUD" operations, complicated restrictions that are dependent on the ML and sometimes dependent on very idiosyncratic deployment details can prevent some schema operations from succeeding.

Some DBs support operations that are even more powerful than schema and data CRUD operations. For example, many DBs support the concept of a query, which we will define as "a request to retrieve a collection of data that can potentially use complex criteria to broaden or limit the collection of data involved." Likewise, many DBs support the concept of a command, which we will define as "a request to create new data, to update existing data, or to delete existing data—potentially using complex criteria similar to a query." Most modern DBs that support both queries and commands even allow us to use separate queries (called subqueries) to specify the complex criteria for these operations.

Any DB that supports CRUD operations must consider concurrent access and conflicting operations. Anytime two or more requests (any combination of queries and commands) attempt to access overlapping collections of data, we have concurrent access. If all of the operations are only retrieving data (no creation, update, or deletion), then the DB can implement the correct behavior without needing any sophisticated logic. If any one of the operations needs to perform a write (create, update, or delete), then we have conflicting operations on overlapping data. Whenever this happens, there are potential problems—if the DB allows all of the operations to

KAHE

execute, then the execution order might potentially change the results seen by the programs or users making the requests.

DATABASE MANAGEMENT

When DBs are used to capture large amounts of data content, or complex data structures, the potential for errors becomes an important concern—especially when the size and complexity make it difficult for human verification. In order to address these potential errors and other issues (like the conflicting operation scenario that we mentioned earlier), we need to use some specialized software. The DB vendor can deploy this specialized software as a library, as a separate program, or as a collection of separate programs and libraries. Regardless of the deployment, we call this specialized software a database management system (DBMS). Vendors usually deploy a DBMS as a collection of separate programs and libraries.

KAHE

DB CLIENTS, SERVERS, AND ENVIRONMENTS

There is no real standard definition for a DBMS, but when a DBMS is deployed using one or more programs, this collection of programs is usually referred to as the DB Server. Any application program that needs to connect to a DB is usually referred to as the DB-Client. Some authors consider the DB-Server and the DBMS to be equivalent—if there is no DB-Server, then there is no DBMS; so the terms DBMS Server and DBMS-Client are also very common. However, even when there is no DB-Server, the application using the DB is still usually called the DB-Client. Different DBMS implementations have different restrictions. For example, some DBMSs can manage more than one DB, while other implementations require a separate DBMS for each DB.

Because of these differences (and many more that we will not discuss here), it is sometimes difficult to compare different implementations and deployments. Simply using the term “DBMS” can suggest certain features or restrictions in the mind of the reader that the author did not intend. For example, we expect most modern DBMSs to provide certain facilities, such as some mechanism for defining and enforcing integrity constraints—but these facilities are not necessarily required for all situations. If we were to use the term “DBMS” in one of these situations where these “expected” facilities were not required, the reader might incorrectly assume that the “extra” facilities (or restrictions) were a required part of the discussion. Therefore, we introduce a new term, database environment (DBE), which simply means one or more DBs along with any software providing at least the minimum set of required data operations and management facilities. In other words, a DBE focuses on the DB and the desired functionality—it can include a DBMS if that is part of the deployment, but does not have to include a DBMS as long as the necessary functionality is present.

Similarly, the term DBE can be applied to DBs deployed on a single host, as well as DBs deployed over a distributed set of machines. By using this new term, we can ignore the architectural and deployment details when they are not relevant. While this might seem unnecessary, it will prevent the awkward phrasing we would have to use otherwise. (If you prefer, you can substitute a phrase like “one or more DB instances with the optional DBMS applications, libraries, or services needed to implement the expected data operations and management facilities required for this context” whenever you see the term “DBE.”) There are times when we will explicitly use the term DBMS; in those instances, we are emphasizing the use of a traditional DBMS rather than some other facility with more or less capabilities or limitations. For example, we would use the term DBMS when we want to imply that an actual DBMS product such as Oracle, DB2, and so on is being used. If we use the term DBE, we could still be referring to one of these products, but we could also be referring to any other combination of software with greater, lesser, or equal levels of functionality.

DBE ARCHITECTURAL CONCEPTS

When considering the architecture of a complicated system, such as a DBE, there are several different ways we can view the details. For our purposes here, we will merely consider services, components, subsystems, and sites.

SERVICES

KAHE

Regardless of the deployment details, we can create logical collections of related functionality called services. For example, we mentioned earlier that many DBs support queries; we can call the logical grouping of the software that implements this functionality the query service. We can define services like this for both publicly visible functions (such as query) and internal functions (such as query optimization). Services are merely logical collections, which means that they do not necessarily have corresponding structure duplicated in the actual implementation or deployment details. We call any piece of software that uses a service a service consumer, while any piece of software implementing the service is called a service provider. Implicitly, each service has at least one interface (similar to a contractual agreement that defines the inputs, outputs, and protocols used by the service consumers and providers). These interfaces can be very abstract (merely specifying an order of steps to be taken) or they can describe very tangible details such as data types or even textual syntax. Most of these interface details are usually only present in lower-level design diagrams—not the high-level architectural or deployment diagrams.

The same piece of software can be both a service consumer and a service provider and can even consume or provide several different services using many different interfaces—but it is usually better to limit the number of services involved for an individual piece of code. Although we can talk about the services as part of the overall architecture or deployment (like interfaces), we usually do not see them directly represented in architectural or deployment diagrams. Instead, we usually see the components and subsystems implementing the services in these diagrams.

COMPONENTS AND SUBSYSTEMS

For our purposes, a component is simply a deployable bundle that provides a reasonably cohesive set of functionality, and a subsystem is a collection of one or more components that work together toward a common goal. Whenever we want to use the two terms interchangeably, we will use the term COS (component or subsystem). Unlike a service, which is merely a logical grouping, a COS is a physical grouping, which means that it does have a corresponding structure in the implementation. Frequently, we name these COSs after the primary service that they provide. There can be multiple instances of the same COS deployed within the system. These

KAHE

instances are often referred to as servers, although we can also use other terminology. For example, we might see a Query COS defined in the architecture, and several Query Servers (instances of the Query COS) deployed within the environment. Alternatively, we might refer to these COSs or their deployed instances as Query Managers, Query Processors, Query Controllers, or even some other name. Different instances of the same COS can have different implementation and configuration details, as long as the instances still provide all the necessary services using the correct protocols and interfaces. We usually represent a COS in an architectural diagram as a box or oval with its name written inside. Deployment diagrams show each COS instance similarly, but usually the instance name includes some additional hint (such as a number, abbreviation, or other deployment detail) to help differentiate the instances from each other.

SITES

The term site represents a logical location in an architectural diagram or a deployment diagram—typically, this is a real, physical machine, but that is not necessarily true.

For example, we might have a deployment using two sites (named Site-A and SiteB). This means that those two sites could be any two machines as long as all of the necessary requirements are satisfied for the machines and their network connections. In certain circumstances, we could also deploy all of the subsystems located at Site-A and Site-B on the same machine. Remember, a DDB deployed on a single machine is still a DDB. In other words, as long as the deployment plan does not explicitly forbid deploying all the COS instances for that DDB on a single machine, we can deploy them this way and still consider it a DDB. Architectural and deployment diagrams depict sites as container objects (usually with the site name and the deployed COS instances included inside them) when there is more than one site involved; otherwise it is assumed that everything in the diagram is located in a single site, which may or may not be named in the diagram.

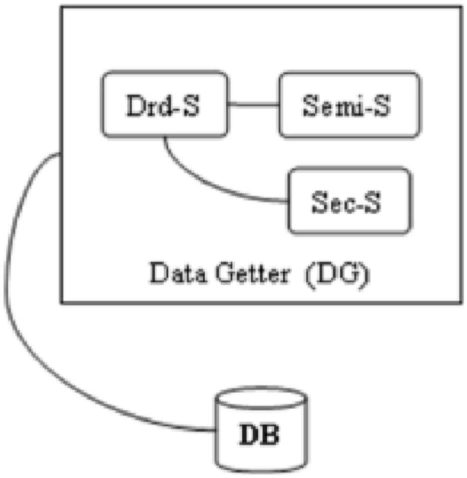
ARCHETYPICAL DBE ARCHITECTURES

When considering a DBE, there are some bare minimum requirements that need to be present—namely, the ability to add new data content and retrieve existing content. Most real-world DBEs provide more than just this minimal level of functionality. In particular, the update and delete operations for data are usually provided. We might see some more sophisticated facilities such as the query service and other services supporting schema operations.

REQUIRED SERVICES

The following figure shows a simplistic architectural diagram for a minimal DBE. The architecture shown is somewhat unrealistic. In this architecture, there is a separate subsystem for each service discussed in this section, and each subsystem is named the same as the service that it provides. These subsystems are contained within a larger subsystem, which we call the Data Getter (DG). This DBE provides (at least) three services—they are named Drd-S, Sec-S, and Semi-S. When reading this diagram, we should recall that it is a DBE and, therefore, the services shown should be considered vitally important, or at least expected—but there might be additional services provided by the environment that are not shown here. For example, this diagram does not show any service providers for query or command operations but that does not necessarily mean we cannot use this diagram to discuss a DBE with those facilities—instead, it merely means that any DBE without those unmentioned facilities is still potentially represented by this diagram.

KAHE



DBE architectural diagram emphasizing the minimum required services.

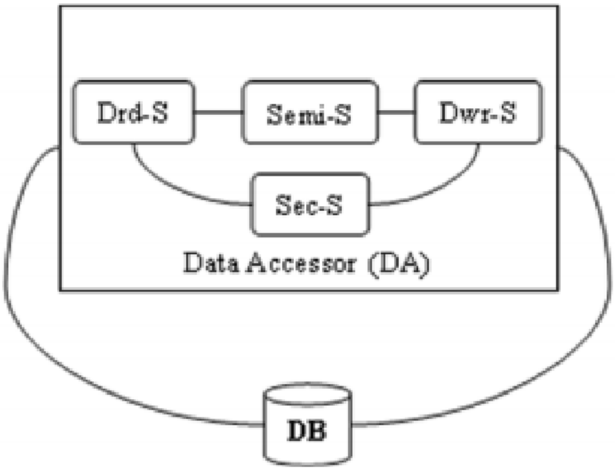
Whenever we use an architectural or deployment diagram for a DBE, we are usually highlighting some requirement or feature of the environment within a specific context; in this case, we are merely showing what a “bare minimum” DBE must provide, and the four services shown here satisfy those minimum requirements. Every DBE must include a service providing the ability to retrieve data from the DB. We will call this the Data Read Service (Drd-S). Since most DBEs also have at least a basic level of privacy or confidentiality, there should always be some form of Security Service (Sec-S). In an effort to be inclusive, we can consider DBEs with “no security” to be implementing a Sec-S that “permits everyone to do everything.” Any real-world DBE should have a Sec-S providing both authentication and authorization. The Drd-S uses the Sec-S to ensure that private data remains unseen by those without proper permissions. There is usually another service providing at least some minimal level of integrity checking or enforcement. This other service is responsible for preventing semantic errors (e.g., data content representing a salary must be greater than zero, otherwise it is a semantic error). Similar to the Sec-S, this service can be less than perfect, or perhaps even implemented using an “allow all modifications” policy if explicit constraints are not defined. We call this service the Semantic Integrity Service (Semi-S). This service can be used by several services, including the Drd-S, which can use it to provide default values for the content it retrieves (among other possibilities).

KAHE

BASIC SERVICES

Every DB must provide some mechanism for populating data (otherwise the DB would always be empty), but we also said that each DB might support different interfaces for the mechanisms they use. Therefore, the ability to write data is not always implemented as a service, or in the very least, it is not always implemented in a way that we can incorporate into our DBE architecture.

For example, if the only data population mechanism provided by a particular DBE was a program that only supported interactive user input (text screens or graphical dialog boxes) then we could not use that program as a service provider in any meaningful sense. However, if the DBE does provide a "program-friendly" mechanism to write data, we can call this the Data Write Service (Dwr-S). Although it is not a required service, and it is not present in all DBEs, it is typically present in most traditional DBMS products, and in many other DBEs we will consider. If there is a Dwr-S, then it uses the Sec-S (to prevent unauthorized data content additions, modifications, and removals) and the Semi-S (to prevent causing any semantic errors when it adds, modifies, or removes any data content). Once again, unless we specify the requirements more explicitly, it is possible for the Sec-S and Semi-S in a particular DBE to provide varying degrees of functionality for these services. The following Figure shows the architectural diagram for a DBE providing the basic services we just discussed. Here, we show all the services used to access the data for read or write operations as a single subsystem called the Data Accessor (DA). We could also have shown the Data Getter subsystem in place of the Drd-S. However, we did not include it here because the Semi-S and Sec-S are used by both the read and the write operations. Similarly, we could consider the combination of the Dwr-S, Semi-S, and Sec-S to be a "Data Setter" subsystem, but these details do not usually add much value to our diagram.



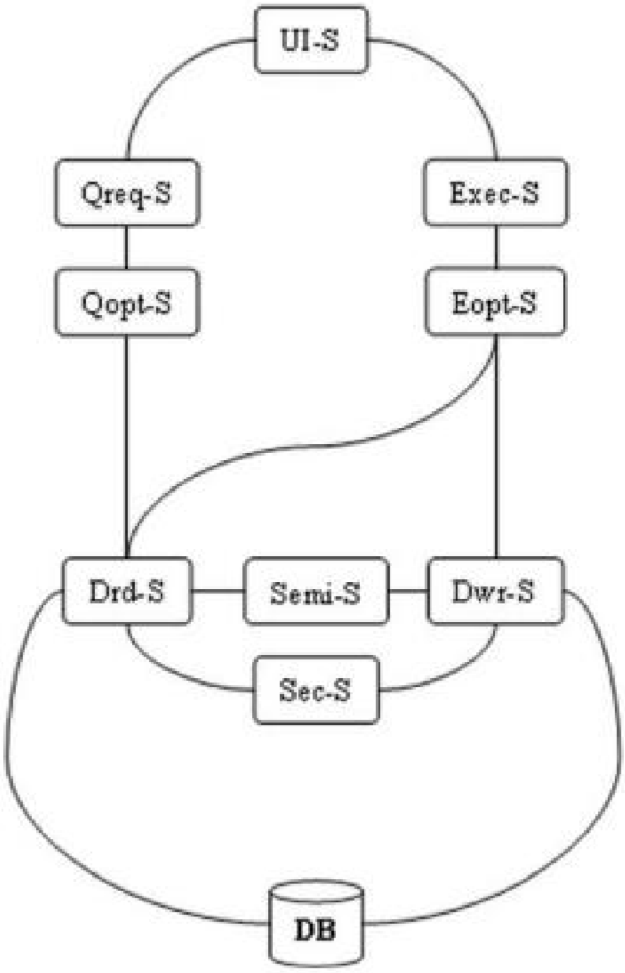
DBE architectural diagram emphasizing the basic services.

EXPECTED SERVICES

Every DBE must supply the functionality contained in the DG, and many DBEs will provide the functionality contained in the DA, but often we expect a DBE to be more powerful than these minimal or basic scenarios. In particular, we mentioned the query service earlier, and here we will call it the Query Request Service (Qreq-S).

Most modern DBMSs should provide this as well as some form of Query Optimization Service (Qopt-S), but neither of these services is a requirement for all DBEs. Typically, the Qreq-S forms a plan for a query and then passes the plan on to the Qopt-S. The Qopt-S optimizes the plan and then uses the Drd-S to retrieve the data matching the query criteria. We will discuss the Qreq-S and Qopt-S further in Chapter 4. We also mentioned that some DBEs have the ability to execute commands (create, update, and delete operations with potentially complex criteria). Therefore, in most DBEs providing DA operations, we would also expect to see an Execution Service (Exec-S) and Execution Optimization Service (Eopt-S) to encapsulate these command operations. Again, these are present in most DBMSs, but not necessarily present in all DBEs. Chapter 3 will explore these services further. Often, there is a “non programmatic” interface provided to users. In particular, many relational DBMSs support a special language (called the SQL) and provide batch and/or interactive facilities that users can employ to pass queries and commands to the DB. We will call the service providing this function the User Interface Service (UI-S). This service is not always present in a DBE and is usually implemented differently, including different syntax, features, and restrictions for the queries and commands. However, we would expect most modern DBMSs (including nonrelational ones) to provide some sort of UI-S.

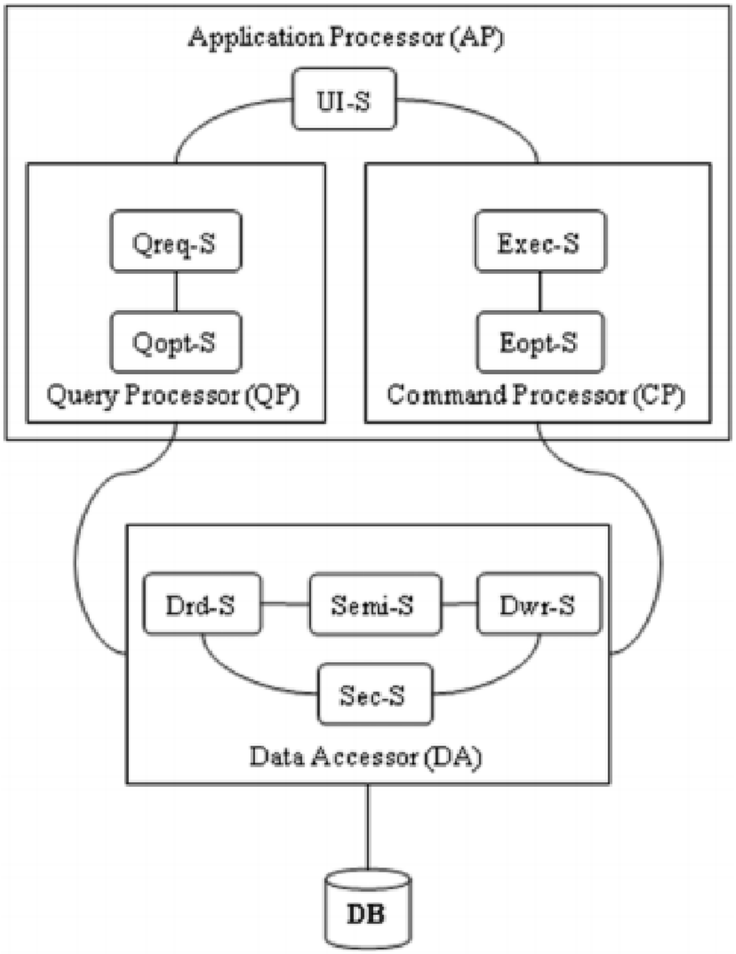
KAHE



DBE architectural diagram emphasizing the expected services.

EXPECTED SUBSYSTEMS

The following figure shows a reasonably realistic DBE set of subsystems for the architecture; it contains all the same services, but we have bundled the services into four subsystems: the application processor (AP), the query processor (QP), the command processor (CP), and the data accessor (DA). Two of the subsystems (QP and CP) are contained within one of the others (AP), while the other two subsystems (AP and DA) are shown as independent packages. Each component has been allocated to one of the subsystems, and the communication links shown only connect subsystems rather than the components inside them. Although the communication links are not quite as detailed, there is no real loss of information when we do this in a diagram. We have placed the Qreq-S and Qopt-S inside the QP. Similarly, we have placed the Exec-S and Eopt-S inside the CP. The AP subsystem contains the combination of the UI-S, QP, and CP subsystems. All the remaining service components have been allocated to the DA.



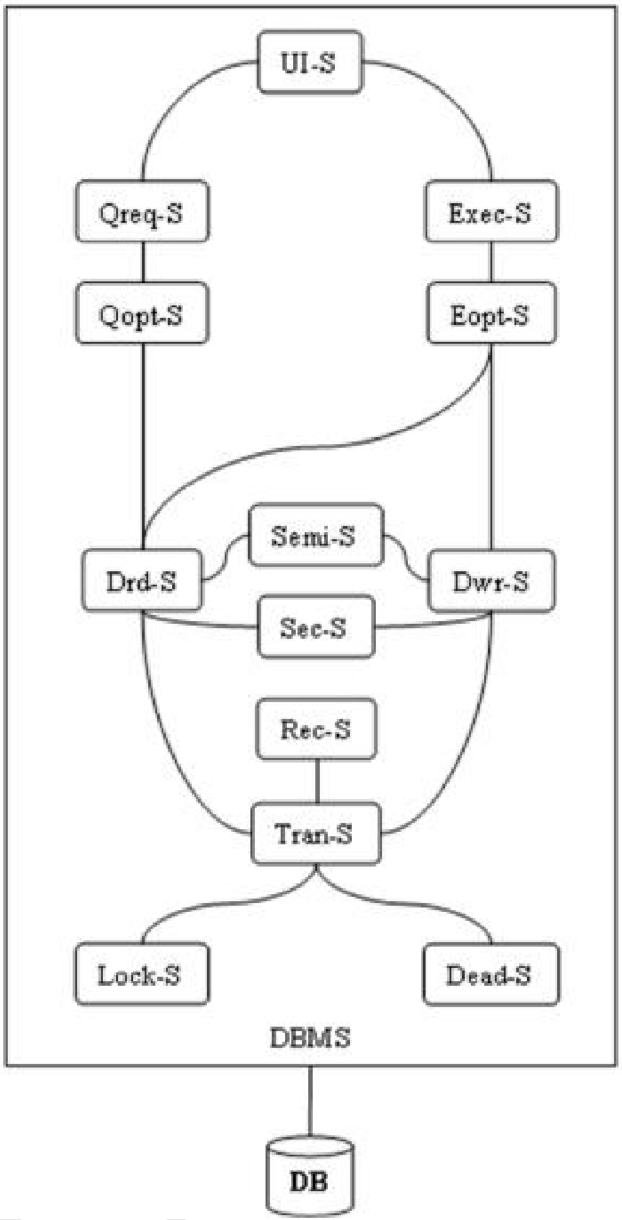
Typical subsystems of simple DBE with the expected services.

TYPICAL DBMS SERVICES

There can be many other services and subsystems in a DBE, but often these additional services and subsystems are highly dependent on other details, specific to the particular DBE being considered. This is especially true when the particular DBE being focused upon is a

DBMS. For example, in a DBE with a Dwr-S, we might include one or more services to handle conflicting operations. Such a DBE might use one or more of the following: a Transaction Management Service (Tran-S), a Locking Service (Lock-S), a Timestamping Service (Time-S), or a Deadlock Handling Service (Dead-S)—all of which will be discussed in Chapters 5 and 6. Similarly, most modern relational DBMSs have a Fallback and Recovery Service (Rec-S), which we will discuss in Chapter 8. The architectural diagram for a DBE like this is shown in Figure 1.5; notice that the services shown are implemented as components inside a single subsystem, called "DBMS" in this diagram. If the DBE is for a DDB, we might even have a Replication Service (Repl-S).

KAHE



DBE architectural diagram for a typical DBMS.

A NEW TAXONOMY

We mentioned earlier that "DBE" is a new term representing several different possible implementations for similar functionality. Because a DBE considers the system at such an abstract level, it can be used to refer to a wide variety of possible architectures and deployments.

In this section, we will present a new taxonomy to be used for classifying all the possible DBE alternatives that we might consider when exploring DB and DDB issues. For our purposes, we will consider a new taxonomy with four levels, presented in order from most abstract to most specific. We hope that this arrangement will reduce the complexity for each successive level and simplify the taxonomic groups (the taxa) containing the environments that we ultimately want to discuss. Like most taxonomies, the extreme cases are most useful for understanding the categorical differences between the taxa, but most real-world DBEs will most likely fall somewhere between the extremes. The four levels of categorizations (from most abstract to most specific) are:

- COS distribution and deployment (COS-DAD)
- COS closedness or openness (COS-COO)
- Schema and data visibility (SAD-VIS)
- Schema and data control (SAD-CON)

COS DISTRIBUTION AND DEPLOYMENT

The first level in our taxonomy is the COS distribution and deployment (COS-DAD) level. This is perhaps the easiest level to understand, and usually this is the first classification that we want to make when evaluating a particular DBE. The two extreme cases that define this level are the completely centralized DBE (CDBE) and the fully distributed DBE (DDBE). In a completely CDBE, we must deploy all the DBs, and COS instances on a single machine. In other words, placing any of the COS instances or DB instances on a second, separate machine is strictly forbidden. This case includes the initial releases of most traditional DBMSs (such as Oracle, Sybase, Informix, etc.) and many other early DBs that did not have a DBMS-Server (such as dBase, Paradox, Clipper, FoxPro, Microsoft Access, etc.). Most modern releases of DBMSs and DBs have moved away from this extreme scenario slightly, since we can often deploy the DB-Clients on a separate machine for many of these systems. Similarly, some modern DBMSs have some ability to distribute their DBs (using techniques such as mirroring, etc.), but they are still essentially a CDBE since the "almost distributed DBs" are not really a "true DB" in the same sense as the original DB that they attempt to duplicate.

If each COS instance and each DB instance is deployed on a separate machine, then we have the other extreme (the fully DDBE). Of course, in the real world, we would probably not go to this extreme—in the very least, it is usually not necessary to separate components within the same subsystem from each other, and also not necessary to separate the lowest-level subsystems (such as the DAs) from the DBs (i.e., the files containing the data structure and content) that they access. Typically, the DDBE will consist of one or more "coordinating" server instances (providing services across the entire DDBE) and a set of DBEs that combine to make the DDBE work. We call each of these DBEs a Sub-DBE (S-DBE), because they combine to form the DDBE in a way that is similar to how subsystems combine to form a system. Each S-DBE is a DBE in its own right, which means that each of them is also subject to categorization too using this taxonomy—in particular, each S-DBE can be either a centralized DBE or another distributed DBE. Most S-DBEs are centralized environments, especially for traditional architectures.

KAHE

COS CLOSEDNESS OR OPENNESS

The second level in our taxonomy is the COS closedness or openness (COS-COO) level. This level considers the software implementation and integration issues for the major subsystems, components, and the DB storage itself. Although we will introduce the two extreme cases for this level (completely open and completely closed), the COSs in most DBEs will occupy a strata (a series of gradations) within this level rather than either extreme scenario.

There is no such thing as a commercial off-the-shelf (COTS) DDBE that we can simply buy and install. Even if there were such a product, we would still probably want to integrate some of our existing COS instances into the DDBE architecture. Conversely, if our goal was to create our own DDBE "from the ground up," it is doubtful that we would write absolutely everything (including the operating systems, file systems, etc.) completely from scratch. Therefore, every DDBE needs to integrate multiple COS instances, some of which we did not write ourselves, regardless of which architectural alternative we choose to implement. When attempting to integrate all of these COS instances, we need to consider the public interface exposed by each COS. Since most of these COSs were not designed with a DDBE specifically in mind, it is quite possible that the interfaces they expose will not provide everything we need. Simply put, there are several DDBE algorithms and services that can only be implemented when we have complete access to the underlying implementation for the COS instances involved—this is especially true if we want to perform our own DDBE research. For example, suppose we wanted to develop a new mechanism for distributed deadlock handling based on a new variation of locking. Obviously, we could only do this if we were able to see (and perhaps even modify) the underlying locking implementation details for each Sub-DBE in the environment.

Subsystems that provide either “unrestrained access” or at least “sufficient access” to the underlying state and implementation details are open to us, while systems that do not are closed to us. While most real subsystems provide some level of access, determining the degree to which a particular subsystem is open or closed depends on the actual interfaces it exposes and the type of functionality we are trying to implement. If all the COS instances are open to us, then we have the first extreme case (completely open). If all the COS instances are closed to us, then we have the other extreme case (completely closed). A completely open DDBE can occur in any of these three scenarios:

- If we write all of the components or service instances ourselves (from scratch)
- If we use free and open source software (FOSS) for all the COS instances that we do not write ourselves
- If we obtain some sort of agreement or arrangement with the COS vendors (for all the COS instances that we do not write ourselves) allowing us the open access we need.

SCHEMA AND DATA VISIBILITY

The third level in our taxonomy is the schema and data visibility (SAD-VIS) level. In this level, we are considering the DB schema and all the data content stored in each DB in the environment. For a CDBE, this is not very interesting because, typically, there is only a single DB to consider or all the DBs use the same ML. However, a DDBE can be a much more complicated consideration. In a DDBE, each DB is actually under S-DBE, which is a DBE in its

KAHE

own right (usually a CDBE). Different DBEs can potentially have different MLs, which means that the ML for the DDBE and the ML for each S-DBE it contains could theoretically be different. In reality, there are not that many different MLs, so it is not likely that there would be that many different ones used in the same environment; but the simple fact that there might be different MLs within the same DDBE is important to consider.

Assuming that there is an appropriate way to combine the different MLs into a single (perhaps different) ML, we can now consider the "combined schema" for all the DBs in the DDBE. If the "most powerful user in the DDBE," which we will call the DDB administrator (DDBA), can see all of the data structure across all the DBs, then we have total schema visibility (TSV). In other words, TSV means that there are no hidden data structures and there is no structure missing from the combined schema but present in one of the S-DBE schemas. Similarly, we can consider the data content for the combined schema and compare it to the data content in each S-DBE. If every data value present in the S-DBE DBs is visible to the DDBA in the DDBE, then we have total data visibility (TDV). In other words, TDV means that there is no hidden data content; there is no data value missing from the DDBE but present in one of the CDBE DBs. When we have both total schema visibility and total data visibility, we have the first extreme case for this level, namely, total visibility (TV).

It should be obvious that the other extreme case is not possible—if the combined schema was empty and all the data content were hidden, then the DDBE would be completely worthless! If we have some hidden schema, then we have partial schema visibility (PSV). If we have some hidden data, then we have partial data visibility (PDV). Having either PSV, PDV, or both PSV and PDV is referred to as partial visibility (PV). PV is a common occurrence and even a requirement for some particular DDBE architectures.

SCHEMA AND DATA CONTROL

The fourth level in our taxonomy is the schema and data control (SAD-CON) level. In this level, we are considering the set of all operations that we are allowed to perform using only the visible schema and the visible data content (we ignore any operation that would attempt to use hidden schema or content, since it would obviously fail). Once again, our primary focus is on the DDBEs. If the DDBA can perform any and every valid schema operation on the visible combined schema structures, then we have total schema control (TSC). Similarly, if the DDBA can perform any and every possible data operation on the visible data content for the DDBE, then we have total data control (TDC). When we have both TSC and TDC, we have the first extreme scenario, which we call total control (TC). If there is at least one valid schema operation that the DDBA does not have permission to perform for some part of the visible combined schema, then we only have partial schema control (PSC). If there is at least one data operation that the DDBA does not have permission to perform on some subset of the visible data content, then we have partial data control (PDC). Having either PSC, PDC, or both PSC and PDC is referred to as partial control (PC).

KAHE

POSSIBLE QUESTIONS

PART B

(EACH QUESTION CARRIES SIX MARKS)

1. Discuss in detail about Database Concepts.
2. Describe about Services, Components, Subsystems and Sites in detail.
3. Discuss in detail about Data Models.
4. Explain in detail about Expected Services and Expected Subsystems.
5. Discuss in detail about Database Operations.
6. Explain in detail about typical DBMS services.
7. Discuss in detail about Database Management.
8. Expound COS distribution and deployment, COS Closedness or Openness in detail.
9. Discuss in detail about Database Clients, Servers and Environments.
10. Explain in detail about Schema and Data Visibility, Schema and Data Control.

KAHE

PART C

(EACH QUESTION CARRIES TEN MARKS)

1. Discuss in detail about Cryptography.
2. Discuss in detail about terminologies of failure and commit protocols.
3. Discuss in detail about Query processing in centralized systems.
4. Explain i)Vertical Fragmentation ii)Horizontal Fragmentation in detail.
5. Discuss in detail about DBE taxonomy.

KARPAGAM ACADEMY OF HIGHER EDUCATION
DEPARTMENT OF COMPUTER APPLICATIONS
DISTRIBUTED DATABASE MANAGEMENT SYSTEM

SUBJECT CODE:17CAP405D **CLASS : II MCA** **SEMESTER: IV**

Unit-1

Questions	Op1	Op2	Op3	Op4	Op5	Op6	Answer
Distributed data base is a collection of multiple ,logically interrelated databases distributed over a ___	Dbms	DDBS	RDBMS	Computer network			Computer network
The data independence are basically __________ types.	One type	Two type	Six Type	Ten type			Two type
Logical data Independence deals with	Physical Structure	Logical Structure	Database Structure	Network Structure			Logical Structure
Physical data independence deals with hiding the details of	Physical Structure	Logical Structure	Database Structure	Network Structure			Network Structure
_______________ is the final form of transparency in DDBMS	Network transparency	Replication Transparency	Data Independence	Fragmentation transparency			Fragmentation transparency
which of the following is commonly done for reasons of performance ,avialability ,and reliablity	fragmentation	replication	network transparecy	data independence			fragmentation
How many types of fragmentation alternatives generally followed	one	two	three	four			two
Horizontal fragmentation has a subset of________ relation	tuples	columns	Rows and columns	vertical			tuples
Vertical fragmenation has a subset of _______________ relation	Rows and coulumns	centrally	logically	columns			columns
The data to be stored in close proximity to use is called_________	Data Localization	Centralization	Conceptual data base	Improved performance			Data Localization
The distributed database design has _____________ basic alternatives to placing a data	2	1	3	4			1
________ database is divided into a number of disjoint partitions	partitioned	fully	replicated	medium,			partitioned
The partition of the database is stored at more than one site is known as__________	partially replicated	fully replicated	no replication	normal			partially replicated
__________processing deals with designing algorithms and convert them into a series of data manipulation operation.	Distributed deadlock	concurrency control	Directory managemen	query processing			query processing
A directory contains information about __________ data items in the database	Descriptional Location	Directory managemen	query processing	DDBS			Descriptional Location
The values of multiple copies of every data item for coverage to the same value is called	concurrency control	mutual consistency	Distributed dbms	Directory management			concurrency control

_______________ synchronizing is the execution of user requests before the execution starts	Optimistic	Pesimistics	Locking	Time stamping			Pesimistics
Executing the request and then checking if the execution compromised the consistency of the database is called ---------------------	Optimistic	Pesimistics	Time stamping	dead lock			Optimistic
mutual exclution of access to data items is known as ______________	security	Database design	Time stamping	Lock			Time stamping
______________ is the potential advantage of distributed systems	Reliability	Confidentiality	Security	locality			Reliability
Developing inidividual module is called_______________	programming in medium	programming in small	programming in normal	programming in large			programming in small
Task of integrating module into a complete system ___________	programming in medium	programming in small	programming in normal	programming in large			programming in medium
Reference model can described into	3 level	2 level	1 level	0 level			3 level
DBMS statandard proposals are prepared as	CCA	ANSI	CCNA	SPARC			CCA
Data is the central resources that a DBMS manages. This approach is referred to as	Data logical approach	OR approach	Exor approcah	functional approcah			Data logical approach
ANSI/SPARC architecture recognizes how many views	1	2	3	4			3
External views deals about the _________	programmer	user	system	programmer and user			programmer and user
which one is the lowest level of architecture	Internal view	External view	Conceptual View	Rear view			Internal view
Architectural models for distributed DBMS divided into ______________ types	2	4	5	3			5
Autonomy refers to the distributIon of	control	data	function	requirement			control
DBMS can execute the transaction that are submitted to it in any way that it wants to is called ____	design autonomy	communication autonomy	execution autonomy	autonomous			execution autonomy
________ system consists of DBMS that can operate independently	semi autonomous	design autonomous	communication autonomous	execution autonomous			semi autonomous
Peer to peer system is called___________	semi distributed	fully distributed	medium distributed	full and half distributed			fully distributed
How many function are involved in client server systems	2	3	0	4			2
Only one server which is accessed by multiple clients is called _________	singel server	one to one	3 tier	multiclient			multiclient
The transparency of data access is provided at the server interface is known as	heavy clients	medium clients	light clients	normal			light clients
Individual Internal schema definition at each site is _________	LIS	GCS	LCS	ES			LIS
_________schema describes the enterprise view of data	ES	GCS	LIS	LCS			GCS
Local conceptual schema describes about_________	data organisation	logical organisation	organisation	data			logical organisation

User application and user access to the database is supported by	LIS	GCS	LCS	ES			ES
________is responsible for interpreting user commands	Distributed execution monitor	Query optimizer	interface handler	semantic data controlloer			interface handler
_____________is responsible for integrity constraints and authorization	Semantic data controller	Data controller	Control	Structure			Semantic data controller
_____________determines an execution strategy to minimize a cost function	semantic data controller	peer to peer	client /server	query optimizer and decomposer			query optimizer and decomposer
Execution monitor is also called	transaction manager	distribution	distributed transaction manager	data controller			distributed transaction manager
Data processor in the distributed DBMS consist of ________ types of elements	3	2	1	4			3
local query optimizer actually acts as_________	path selector	access path selector	optimizer	dataitem			access path selector
The Local database remains consistent even when failure occur is known as	recovery	support processor	optimizer	recovery manager			recovery manager
________physically accesses the database according to physical command	processor	support processor	runtime support processor	time processor			runtime support processor
Run time support processor is the interface to the operating system and contains __________	Data base buffer	cache	Data processor	Database buffer and cache			Database buffer and cache
Different data models and languages both a local database and global data base accessed is known as	unilingual	multilingual	lingual	semilingual			unilingual
________ Is the basic philosophy to permit each user to access the global database	unilingual	multilingual	lingual	semiligual			multilingual
how many layers a model without a global conceputal schema idenitfies ?	2	3	4	5			2
Global directory issues are relevant for________	Distributed DBMS	multi DBMS	DBMS	DDBMS and multi DBMS			DDBMS and multi DBMS

KAHE

UNIT-II

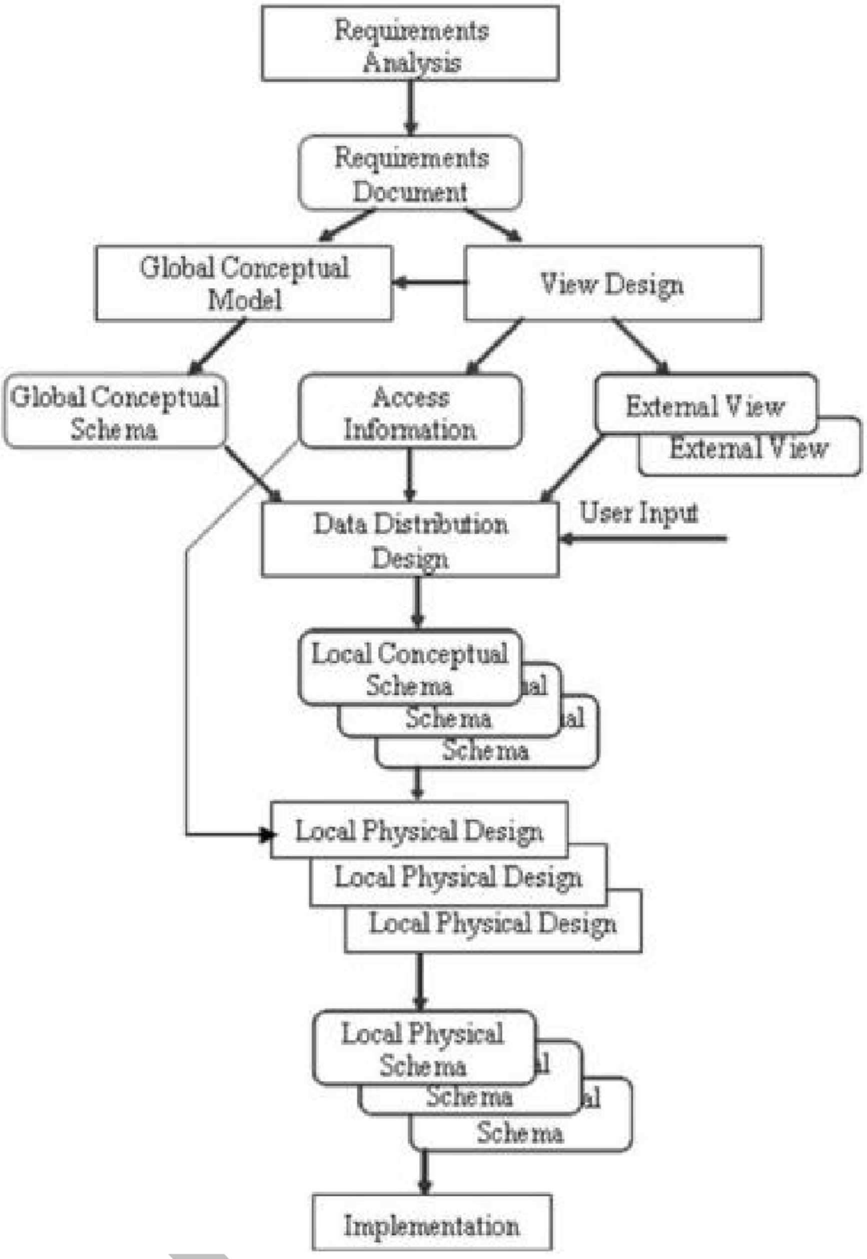
SYLLABUS

Data Distribution Alternatives: Design Alternatives- Localized Data- Distributed Data. Fragmentation: Vertical Fragmentation- Horizontal Fragmentation. Distribution Transparency: Location Transparency-Fragmentation Transparency-Replication Transparency-Location, Fragmentation, and Replication Transparencies.

DATA DISTRIBUTION ALTERNATIVES

In a distributed database management system (DDBMS) data is intentionally distributed to take advantage of all computing resources that are available to the organization. For these systems, the schema design is done top– down. A top–down design approach considers the data requirements of the entire organization and generates a global conceptual model (GCM) of all the information that is required. The GCM is then distributed across all appropriate local DBMS (LDBMS) engines to generate the local conceptual model (LCM) for each participant LDBMS. As a result, DDBMSs always have one and only one GCM and one or more LCM. The following Figure depicts the top–down distribution design approach in a distributed database management system. By contrast, the design of a federated database system is done from the bottom–up.

A bottom–up design approach considers the existing data distributed within an organization and uses a process called schema integration to create at least one unified schema. The unified schema is similar to the GCM, except that there can be more than one unified schema. Schema integration is a process that uses a collection of existing conceptual model elements, which have previously been exported from one or more LCMs, to generate a semantically integrated model (a single, unified schema). Designers of a distributed database (DDB) will decide what distribution alternative is best for a given situation. They may decide to keep every table intact (all rows and all columns of every table are stored in the same DB at the same Site) or to break up some of the tables into smaller chunks of data called fragments or partitions. In a distributed database, the designers may decide to store these fragments locally (localized) or store these fragments across a number of LDBMSs on the network (distributed).



The top–down design process for a distributed database system.

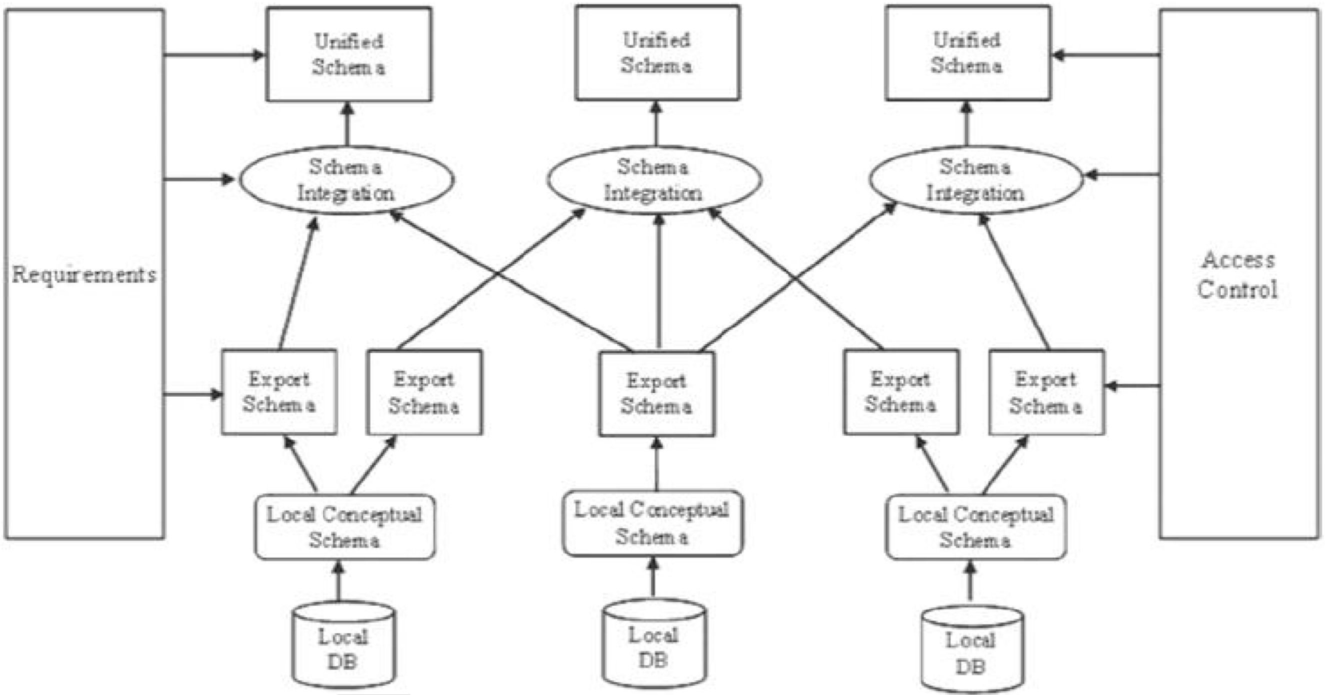
Distributed tables can have one of the following forms:

- Nonreplicated, nonfragmented (nonpartitioned)
- Fully replicated (all tables)
- Fragmented (also known as partitioned)

- Partially replicated (some tables or some fragments)

- Mixed (any combination of the above)

The goal of any data distribution is to provide for increased availability, reliability, and improved query access time. On the other hand, as opposed to query access time, distributed data generally takes more time for modification (update, delete, and insert). It has been proved that for a given distribution ¨ design and a set of applications that query and update distributed data, determining the optimal data allocation strategy for database servers in a distributed system is an NP-complete problem. That is why most designers are not seeking the best data distribution and allocation design but one that minimizes some of the cost elements.



The bottom – up design process for a federated database system using schema integration.

DESIGN ALTERNATIVES

LOCALIZED DATA

This design alternative keeps all data logically belonging to a given DBMS at one site (usually the site where the controlling DBMS runs). This design alternative is sometimes called ―not distributed.‖

DISTRIBUTED DATA

A database is said to be distributed if any of its tables are stored at different sites; one or more of its tables are replicated and their copies are stored at different sites; one or more of its tables are fragmented and the fragments are stored at different sites; and so on. In general, a database is distributed if not all of its data is localized at a single site.

NONREPLICATED, NONFRAGMENTED

This design alternative allows a designer to place different tables of a given database at different sites. The idea is that data should be placed close to (or at the site) where it is needed the most. One benefit of such data placement is the reduction of the communication component of the processing cost. For example, assume a database has two tables called ―EMP‖ and ―DEPT.‖ A designer of DDBMS may decide to place EMP at Site 1 and DEPT at Site 2. Although queries against the EMP table or the DEPT table are processed locally at Site 1 and Site 2, respectively, any queries against both EMP and DEPT together (join queries) will require a distributed query execution. The question that arises here is, ―How would a designer decide on a specific data distribution?‖ The answer depends on the usage pattern for these two tables. This distribution allows for efficient access to each individual table from Sites 1 and 2. This is a good design if we assume there are a high number of queries needing access to the entire EMP table issued at Site 1 and a high number of queries needing access to the entire DEPT table issued at Site 2. Obviously, this design also assumes that the percentage of queries needing to join information across EMP and DEPT is low.

FULLY REPLICATED

This design alternative stores one copy of each database table at every site. Since every local system has a complete copy of the entire database, all queries can be handled locally. This design alternative therefore provides for the best possible query performance. On the other hand,

KAHE

since all copies need to be in sync—show the same values—the update performance is impacted negatively. Designers of a DDBMS must evaluate the percentage of queries versus updates to make sure that deploying a fully replicated database has an overall acceptable performance for both queries and updates.

FRAGMENTED OR PARTITIONED

Fragmentation design approach breaks a table up into two or more pieces called fragments or partitions and allows storage of these pieces in different sites.

There are three alternatives to fragmentation:

- Vertical fragmentation
- Horizontal fragmentation
- Hybrid fragmentation

This distribution alternative is based on the belief that not all the data within a table is required at a given site. In addition, fragmentation provides for increased parallelism, access, disaster recovery, and security/privacy. In this design alternative, there is only one copy of each fragment in the system (non replicated fragments).

PARTIALLY REPLICATED

In this distribution alternative, the designer will make copies of some of the tables (or fragments) in the database and store these copies at different sites. This is based on the belief that the frequency of accessing database tables is not uniform. For example, perhaps Fragment 1 of the EMP table might be accessed more frequently than Fragment 2 of the table. To satisfy this requirement, the designer may decide to store only one copy of Fragment 2, but more than one copy of Fragment 1 in the system. Again, the number of Fragment 2 copies needed depends on how frequently these access queries run and where these access queries are generated.

MIXED DISTRIBUTION

KAHE

In this design alternative, we fragment the database as desired, either horizontally or vertically, and then partially replicate some of the fragments.

FRAGMENTATION

Fragmentation requires a table to be divided into a set of smaller tables called fragments. Fragmentation can be horizontal, vertical, or hybrid (a mix of horizontal and vertical). Horizontal fragmentation can further be classified into two classes: primary horizontal fragmentation (PHF) and derived horizontal fragmentation (DHF). When thinking about fragmentation, designers need to decide on the degree of granularity for each fragment. In other words, how many of the table columns and/or rows should be in a fragment? The range of options is vast. At one end, we can have all the rows and all the columns of the table in one fragment. This obviously gives us a non fragmented table; the grain is too coarse if we were planning to have at least one fragment. At the other end, we can put each data item (a single column value for a single row) in a separate fragment. This grain obviously is too fine: it would be hard to manage and would add too much overhead to processing queries. The answer should be somewhere in between these two extremes. As we will explain later, the optimal solution depends on the type and frequency of queries that applications run against the table. In the rest of this section, we explore each fragmentation type and formalize the fragmentation process.

KAHE

VERTICAL FRAGMENTATION

Vertical fragmentation (VF) will group the columns of a table into fragments. VF must be done in such a way that the original table can be reconstructed from the fragments. This fragmentation requirement is called ―reconstructiveness.‖ This requirement is used to reconstruct the original table when needed. As a result, each VF fragment must contain the primary key column(s) of the table. Because each fragment contains a subset of the total set of columns in the table, VF can be used to enforce security and/or privacy of data. To create a vertical fragment from a table, a select statement is used in which ―Column_list‖ is a list of columns from R that includes the primary key.

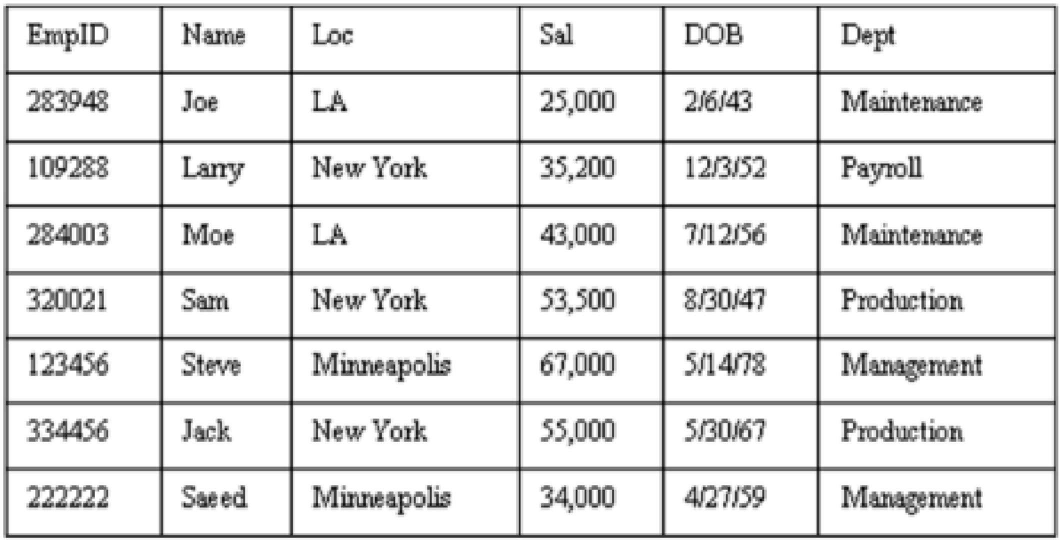
Select Column_list from R;

Example: Consider the EMP table shown in the following Figure. Let‘s assume that for security reasons the salary information for employees needs to be maintained in the company headquarters‘ server, which is located in Minneapolis. To achieve this, the designer will fragment the table vertically into two fragments as follows:

Create table EMP_SAL as

Select EmpID, Sal

From EMP;



EmpID	Name	Loc	Sal	DOB	Dept
283948	Joe	LA	25,000	2/6/43	Maintenance
109288	Larry	New York	35,200	12/3/52	Payroll
284003	Moe	LA	43,000	7/12/56	Maintenance
320021	Sam	New York	53,500	8/30/47	Production
123456	Steve	Minneapolis	67,000	5/14/78	Management
334456	Jack	New York	55,000	5/30/67	Production
222222	Saeed	Minneapolis	34,000	4/27/59	Management

EMP Table

The nonfragmented version of the EMP table.

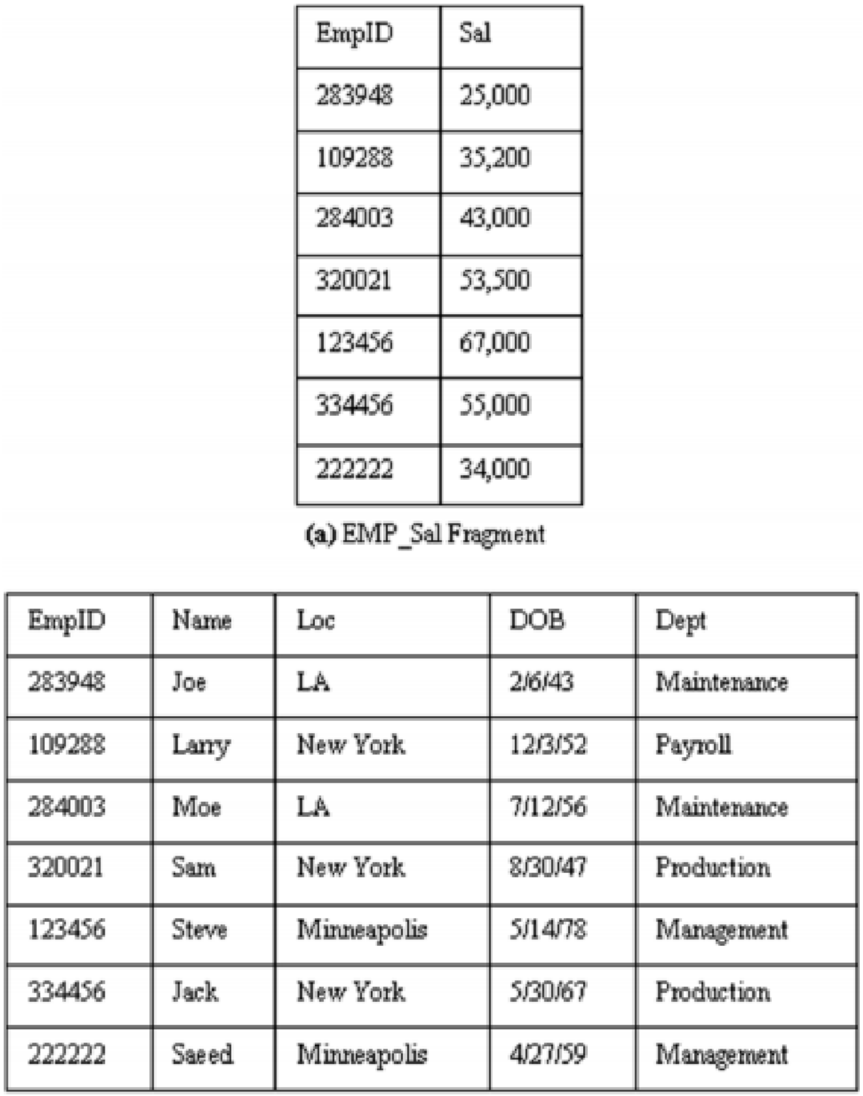
Create table EMP_NON_SAL as

Select EmpID, Name, Loc, DOB, Dept

From EMP;

EMP_SAL contains the salary information for all employees while EMP_NON_SAL contains the nonsensitive information. These statements generate the vertical fragments shown in Figure a, b from the EMP table.

After fragmentation, the EMP table will not be stored physically anywhere. But, to provide for fragmentation transparency—not requiring the users to know that the EMP table is fragmented—we have to be able to reconstruct the EMP table from its VF fragments. This will give the users the illusion that the EMP table is stored intact. To do this, we will use the following join statement anywhere the EMP table is required:



EmpID	Sal
283948	25,000
109288	35,200
284003	43,000
320021	53,500
123456	67,000
334456	55,000
222222	34,000

(a) EMP_Sal Fragment

EmpID	Name	Loc	DOB	Dept
283948	Joe	LA	2/6/43	Maintenance
109288	Larry	New York	12/3/52	Payroll
284003	Moe	LA	7/12/56	Maintenance
320021	Sam	New York	8/30/47	Production
123456	Steve	Minneapolis	5/14/78	Management
334456	Jack	New York	5/30/67	Production
222222	Saeed	Minneapolis	4/27/59	Management

(b) EMP_NON_Sal Fragment

The vertical fragments of the EMP table.

Select EMP_SAL.EmpID, Sal, Name, Loc, DOB, Dept

From EMP_SAL, EMP_NON_SAL

Where EMP_SAL.EmpID = EMP_NON_SAL.EmpID;

This join statement can be used in defining a view called ―EMP‖ and/or can be used as an in-line view in any select statement that uses the virtual (physically non existing) table ―EMP.‖

KAHE

HORIZONTAL FRAGMENTATION

Horizontal fragmentation (HF) can be applied to a base table or to a fragment of a table. Note that a fragment of a table is itself a table. Therefore, in the following discussion when we use the term table, we might refer to a base table or a fragment of the table. HF will group the rows of a table based on the values of one or more columns. Similar to vertical fragmentation, horizontal fragmentation must be done in such a way that the base table can be reconstructed (reconstructiveness). Because each fragment contains a subset of the rows in the table, HF can be used to enforce security and/or privacy of data. Every horizontal fragment must have all columns of the original base table. To create a horizontal fragment from a table, a select statement is used. For example, the following statement selects the row from R satisfying condition C:

Select * from R where C;

As mentioned earlier, there are two approaches to horizontal fragmentation. One is called primary horizontal fragmentation (PHF) and the other is called derived horizontal fragmentation (DHF).

PRIMARY HORIZONTAL FRAGMENTATION

Primary horizontal fragmentation (PHF) partitions a table horizontally based on the values of one or more columns of the table.

The following Example discusses the creation of three PHF fragments from the EMP table based on the values of the Loc column.

Consider the EMP table . Suppose we have three branch offices, with each employee working at only one office. For ease of use, we decide that information for a given employee should be stored in the DBMS server at the branch office where that employee works. Therefore, the EMP table needs to be fragmented horizontally into three fragments based on the value of the Loc column as shown below:

Create table MPLS_EMPS as

Select * From EMP

Where Loc = _Minneapolis‘;

Create table LA_EMPS as

Select * From EMP

Where Loc = _LA‘;

Create table NY_EMPS as

Select * From EMP

Where Loc = _New York‘;

This design generates three fragments, shown in Figure a,b,c. Each fragment can be stored in its corresponding city‘s server. Again, after fragmentation, the EMP table will not be physically stored anywhere. To provide for horizontal fragmentation transparency, we have to be able to reconstruct the EMP table from its HF fragments. This will give the users the illusion that the EMP table is stored intact. To do this, we will use the following union statement anywhere the EMP table is required:

(Select * from MPLS_EMPS

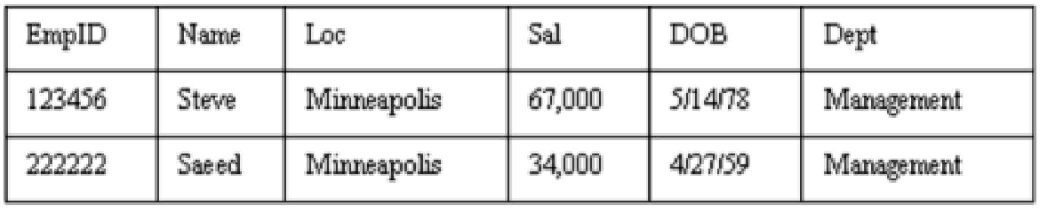
Union

Select * from LA_EMPS)

KAHE

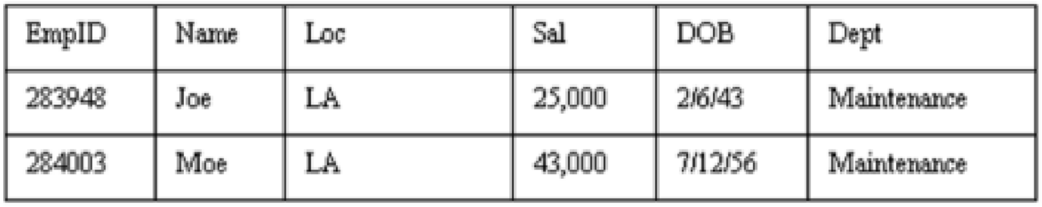
Union

Select * from NY_EMPS;



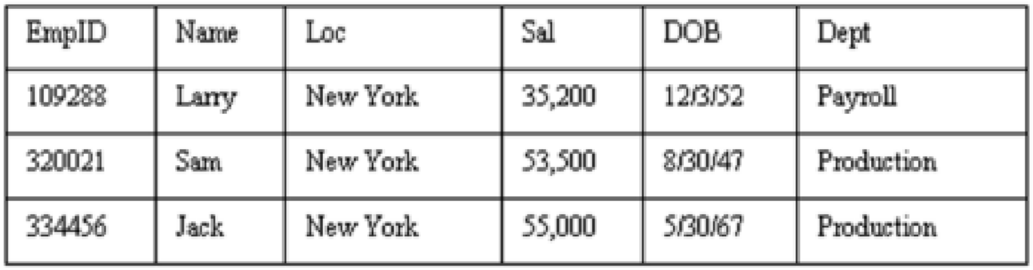
EmpID	Name	Loc	Sal	DOB	Dept
123456	Steve	Minneapolis	67,000	5/14/78	Management
222222	Saeed	Minneapolis	34,000	4/27/59	Management

(a) MPLS_EMPS fragment



EmpID	Name	Loc	Sal	DOB	Dept
283948	Joe	LA	25,000	2/6/43	Maintenance
284003	Moe	LA	43,000	7/12/56	Maintenance

(b) LA_EMPS fragment



EmpID	Name	Loc	Sal	DOB	Dept
109288	Larry	New York	35,200	12/3/52	Payroll
320021	Sam	New York	53,500	8/30/47	Production
334456	Jack	New York	55,000	5/30/67	Production

(c) NY_EMPS fragment

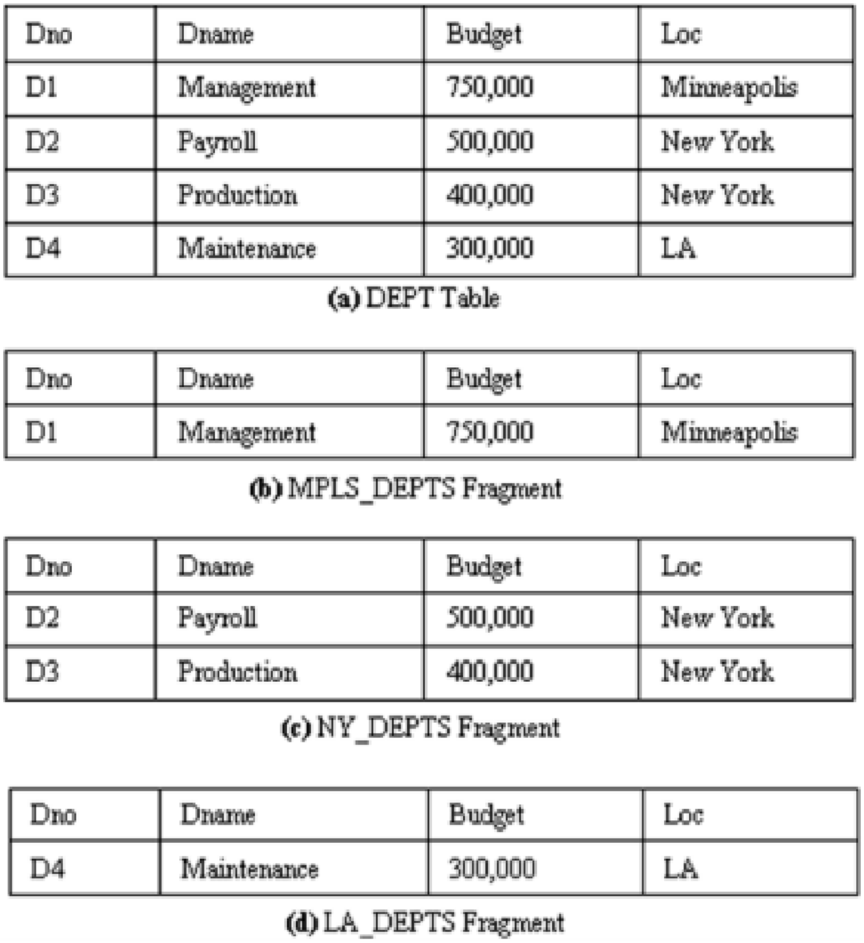
The horizontal fragments of the EMP table with fragments based on Loc.

DERIVED HORIZONTAL FRAGMENTATION

Instead of using PHF, a designer may decide to fragment a table according to the way that another table is fragmented. This type of fragmentation is called derived horizontal fragmentation (DHF). DHF is usually used for two tables that are naturally (and frequently) joined. Therefore, storing corresponding fragments from the two tables at the same site will speed up the join across the two tables. As a result, an implied requirement of this fragmentation design is the presence of a join column across the two tables.

Example: the following Figure shows table ―DEPT(Dno, Dname, Budget, Loc),‖ where Dno is the primary key of the table. Let‗s assume that DEPT is fragmented based on the department‗s city. Applying PHF to the DEPT table generates three horizontal fragments, one for each of the cities in the database, as depicted in Figure b,c,d.

Now, let‗s consider the table ―PROJ,‖ as depicted in Figure a. We can partition the PROJ table based on the values of Dno column in the DEPT table‗s fragments with the following SQL statements. These statements will produce the derived fragments from the PROJ table as shown in Figure b,c. Note that there are no rows in PROJ3, since department ―D4‖ does not manage any project.



Dno	Dname	Budget	Loc
D1	Management	750,000	Minneapolis
D2	Payroll	500,000	New York
D3	Production	400,000	New York
D4	Maintenance	300,000	LA

(a) DEPT Table

Dno	Dname	Budget	Loc
D1	Management	750,000	Minneapolis

(b) MPLS_DEPTS Fragment

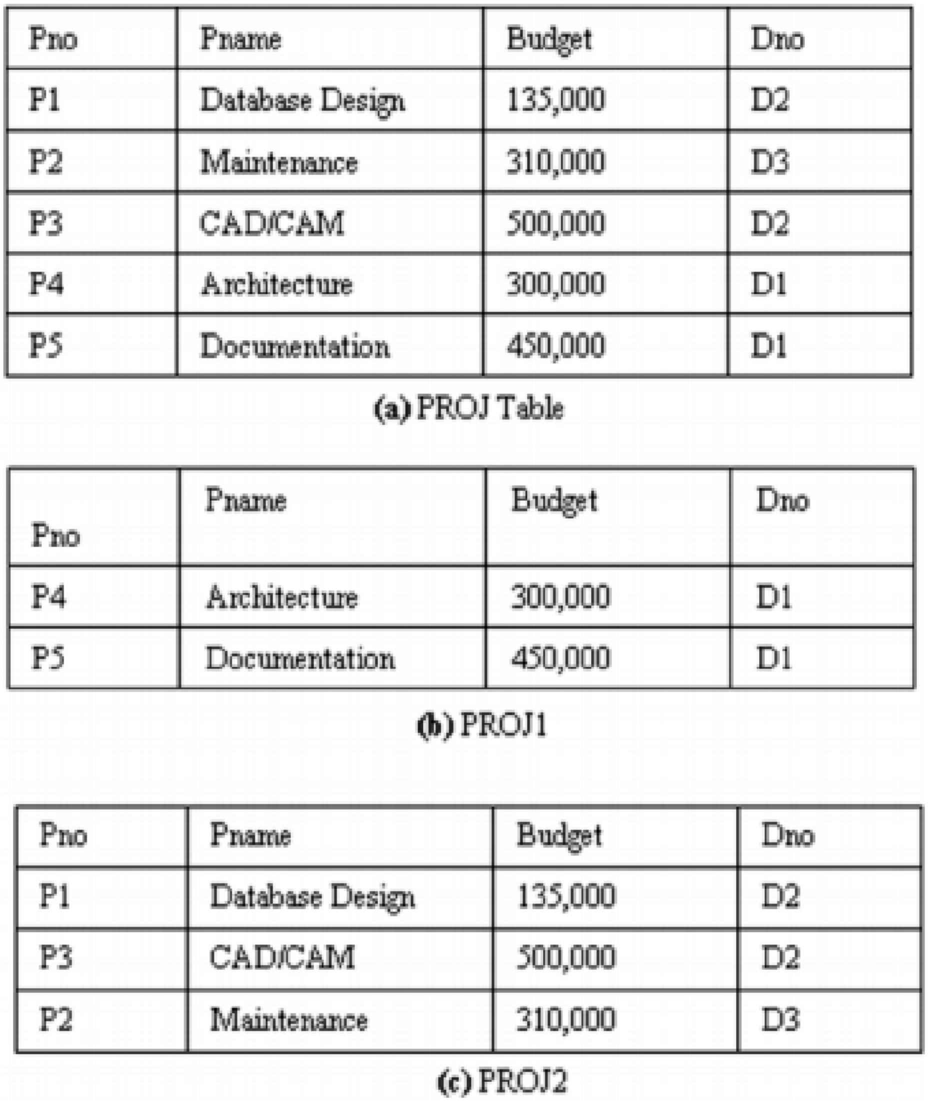
Dno	Dname	Budget	Loc
D2	Payroll	500,000	New York
D3	Production	400,000	New York

(c) NY_DEPTS Fragment

Dno	Dname	Budget	Loc
D4	Maintenance	300,000	LA

(d) LA_DEPTS Fragment

The fragments of the DEPT table with fragments based on Loc.



Pno	Pname	Budget	Dno
P1	Database Design	135,000	D2
P2	Maintenance	310,000	D3
P3	CAD/CAM	500,000	D2
P4	Architecture	300,000	D1
P5	Documentation	450,000	D1

(a) PROJ Table

Pno	Pname	Budget	Dno
P4	Architecture	300,000	D1
P5	Documentation	450,000	D1

(b) PROJ1

Pno	Pname	Budget	Dno
P1	Database Design	135,000	D2
P3	CAD/CAM	500,000	D2
P2	Maintenance	310,000	D3

(c) PROJ2

The PROJ table and its component DHF fragments.

Create table PROJ1 as

Select Pno, Pname, Budget, PROJ.Dno

From PROJ, MPLS_DEPTS

Where PROJ.Dno = MPLS_DEPTS.Dno;

Create table PROJ2 as

Select Pno, Pname, Budget, PROJ.Dno

From PROJ, NY_DEPTS

Where PROJ.Dno = NY_DEPTS.Dno;

Create table PROJ3 as

Select Pno, Pname, Budget, PROJ.Dno

From PROJ, LA_DEPTS

Where PROJ.Dno = LA_DEPTS.Dno;

It should be rather obvious that all the rows in PROJ1 have corresponding rows in the MPLS_DEPTS fragment, and similarly, all the rows in PROJ2 have corresponding rows in the NY_DEPTS fragment. Storing a derived fragment at the same database server where the deriving fragment is, will result in better performance since any join across the two tables‘ fragments will result in a 100% hit ratio (all rows in one fragment have matching rows in the other).

For this example, assume that sometimes we want to find those projects that are managed by the departments that have a budget of less than or equal to 500,000 (department budget, not project budget) and at other times we want to find those projects that are managed by the departments that have a budget of more than 500,000. In order to achieve this, we fragment DEPT based on the budget of the department. All departments with a budget of less than or equal to 500,000 are stored in DEPT4 and other departments are stored in the DEPT5 fragment. Figures a and b show DEPT4 and DEPT5, respectively.

To easily answer the type of questions that we have outlined in this example, we should create two derived horizontal fragments of the PROJ table based on DEPT4 and DEPT5 as shown below.

Create table PROJ5 as

Select Pno, Pname, Budget, PROJ.Dno

KAHE

From PROJ, DEPT4

Where PROJ.Dno = DEPT4.Dno;

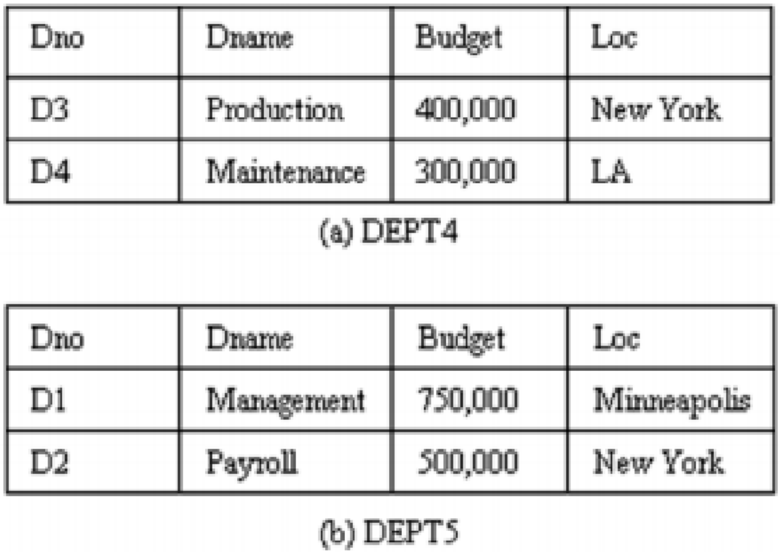
Create table PROJ6 as

Select Pno, Pname, Budget, PROJ.Dno

From PROJ, DEPT5

Where PROJ.Dno = DEPT5.Dno;

The following Figure shows the fragmentation of the PROJ table based on these SQL statements.



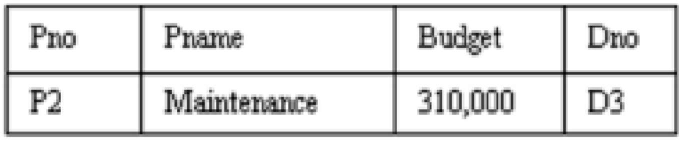
Dno	Dname	Budget	Loc
D3	Production	400,000	New York
D4	Maintenance	300,000	LA

(a) DEPT4

Dno	Dname	Budget	Loc
D1	Management	750,000	Minneapolis
D2	Payroll	500,000	New York

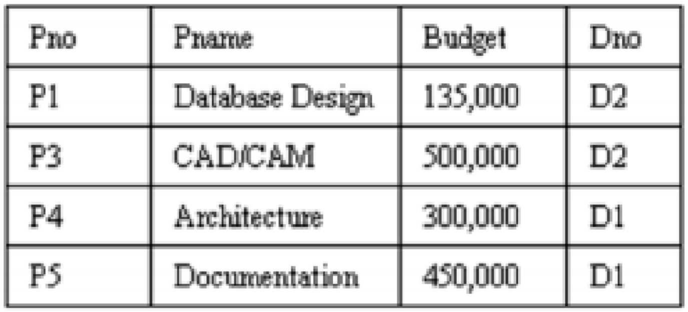
(b) DEPT5

The DEPT table fragmented based on Budget column values.



Pno	Pname	Budget	Dno
P2	Maintenance	310,000	D3

(a) PROJ5



Pno	Pname	Budget	Dno
P1	Database Design	135,000	D2
P3	CAD/CAM	500,000	D2
P4	Architecture	300,000	D1
P5	Documentation	450,000	D1

(b) PROJ6

The derived fragmentation of PROJ table based on the fragmented DEPT table.

DISTRIBUTION TRANSPARENCY

Although a DDBMS designer may fragment and replicate the fragments or the tables of a system, the users of such a system should not be aware of these details. This is what is known as distribution transparency. Distribution transparency is one of the sought after features of a distributed DBE. It is this transparency that makes the system easy to use by hiding the details of distribution from the users. There are three aspects of distribution transparency—location, fragmentation, and replication transparencies.

LOCATION TRANSPARENCY

The fact that a table (or a fragment of table) is stored at a remote site in a distributed system should be hidden from the user. When a table or fragment is stored remotely, the user should not need to know which site it is located at, or even be aware that it is not located locally. This provides for location transparency, which enables the user to query any table (or any fragment) as if it were stored locally.

FRAGMENTATION TRANSPARENCY

The fact that a table is fragmented should be hidden from the user. This provides for fragmentation transparency, which enables the user to query any table as if it were intact and physically stored. This is somewhat analogous to the way that users of a SQL view are often unaware that they are not using an actual table (many views are actually defined as several union and join operations working across several different tables).

REPLICATION TRANSPARENCY

The fact that there might be more than one copy of a table stored in the system should be hidden from the user. This provides for replication transparency, which enables the user to query any table as if there were only one copy of it.

LOCATION, FRAGMENTATION, AND REPLICATION TRANSPARENCIES

The fact that a DDBE designer may fragment a table, make copies of the fragments, and store these copies at remote sites should be hidden from the user. This provides for complete distribution transparency, which enables the user to query the table as if it were physically stored at the local site without being fragmented or replicated.

KAHE

POSSIBLE QUESTIONS

PART B

(EACH QUESTION CARRIES SIX MARKS)

1. Explain about Data Distribution alternatives in detail.
2. Discuss about i)Vertical Fragmentation ii)Horizontal Fragmentation in detail.
3. Describe about Design alternatives in DDBE.
4. Discuss in detail about Distribution Transparency.
5. Describe in detail about Localized Data.
6. Discuss in detail about i) Location Transparency ii) Fragmentation Transparency.
7. Explain in detail about Distributed Data.
8. Describe in detail about Replication Transparency.
9. Discuss in detail about Fragmentation.
10. Describe in detail about Location, Fragmentation and Replication Transparencies.

PART C

(EACH QUESTION CARRIES TEN MARKS)

1. Discuss in detail about Cryptography.
2. Discuss in detail about terminologies of failure and commit protocols.
3. Discuss in detail about Query processing in centralized systems.
4. Explain i)Vertical Fragmentation ii)Horizontal Fragmentation in detail.
5. Discuss in detail about DBE taxonomy.

KAHE

KARPAGAM ACADEMY OF HIGHER EDUCATION
DEPARTMENT OF COMPUTER APPLICATIONS
DISTRIBUTED DATABASE MANAGEMENT SYSTEM

SUBJECT CODE:17CAP405D **CLASS : II MCA** **SEMESTER: IV**

Unit-2

Questions	Op1	Op2	Op3	Op4	Op5	Op6	Answer
How many copies the final issues of replication contains	2	0	1	4			2
multiple copies of database provides more_________	reliability	accessbility	portability	readability			reliability
non distributed multiple dbms always maintains______ copy of the directory	multiple	single	different	big			different
Making decisions on the placement of ___________ across the sites of a computer networks	data	programms	data and programme	functions			data and programme
The organisation of distributed system investigated along _________level of orthogonal dimensions	2	3	1	0			3
level of sharing having how many possibilities	2	1	3	0			3
Programmes are replicated at all sites but data files are not is called	no sharing	data sharing	plus programme sharing	recovery sharing			data sharing
The originate and the necessary datafiles are moved around the network its known to be ?	Data sharing	nosharing	sharing	common sharing			Data sharing
Both data and programmes may be shared is called	Data sharing	No sharing	Data and programmer sharing	sharing			Data and programmer sharing
Reasonably be predicated and not deviate significantly from these predications are called________	information	complete information	partial information	no information			complete information
________ dimension of access pattern behaviour is used to find an identity	2	1	3	4			2
The access pattern of user requests may__________	static	dynamic	static and dynamic	static or dynamic			static and dynamic
There are deviations from predications are called _________	partical information	complete information	no information	information			partical information
How many strategies involved in designing distributed database	1	2	3	4			2
The requirements document is input to______ parallel activites	1	2	3	4			2
_________activity deals with defining the interface for end users	view design	conceptual design	functional design	design			view design
which the enterprises is examined to determine entity types and relationship among these entities	view design	conceptual design	functional design	design			conceptual design

_________ is concerned with determing the entities and their attributes and the relationship among them	entity analysis	functional analysis	relationship analysis	Entitiy relationship			entity analysis
Determining the fundamental functions with which the modeled enterprises involved ________	entity analysis	functional analysis	relationship analysis	ER Diagramme			functional analysis
________ needs to be cross_referenced to get a better understanding of which function deal with entities	functional analysis	requirement analysis	entity analysis	functional analysis and requirement analysis			functional analysis and requirement analysis
__________activity is very important for the conceptual model that should support not only the existing application but also future application	view integration	functional integration	requirement integration	normal			view integration
________ information includes the specification of the frequency of user application	statistical	dynamic	database	server			statistical
The GCS and access pattern information collected as a result of view design that are inputs to the ______ step	distribution	design	distribution design	data			distribution design
Distributing relation is divided into subrelations is called _________	allocation	fragmentation	distribution	database			fragmentation
The distribution of design activity consist of _____ types of activities	fragmentation	allocation	fragmentation and allocation	conceptual			fragmentation and allocation
Which one is the last step in the design process_________	physical design	logical design	distributon design	various design			physical design
_________suitable approach for database already exist and the design task	top down	bottom up	view design	conceptual design			bottom up
__________is the starting point of bottom up design	Individual	local	conceptual	individual,local and conceptual			individual,local and conceptual
______ enviornment exists primarily in the context of heterogeneous data base	topdown	bottom up	viewdesign	conceptual design			bottom up
how many reasons related to fragmentation?	1	2	3	4			2
permitting a number of transactions to execute is called __________	intraquery concurrency	concurrency	schema	none of these			intraquery concurrency
to retrieve data from two fragments use ______	union	join	innerjoin	union and join			union and join
second problem is related to semantic data control specifically to ______ checking	integrity	control	allocation	group			integrity
how many alternatives are generally followed	1	2	3	4			2
Which one is the fragmentation alternatives	horizontal	vertical	horizontal and vertical	horizontal or vertical			horizontal and vertical
the fragmentation of nestings has ______ of different types	horizontal	vertical	hybrid	horizontal or vertical			hybrid
The correctness rules of fragmentation is divided into	1	2	3	4			3

The completeness of which identical to the _________ property of normalization	lossless decompostion	decompostion	composition	fragments			lossless decompostion
The vertical partitioning and disjointness is defined only on the_____ key attributes of relation	primary key	foreign key	non primary key	unique key			non primary key
single copy of replication are reliability and efficiency of ____________	read only queries	sub queries	replication	allocationn			read only queries
_________queries that access the same data items that can be executed in parallel	single copy	multiple copy	read only	write only			read only
A non replicated database commonly called___________	fully replicated	replicated	partially replicated	partitioned database			partitioned database
The database exists in its entirety at each site is _______	fully replicated	replicated	partially replicated	partitioned database			fully replicated
The copies of a fragment may reside in multiple sites is called ?	fully replicated	replicated	partially replicated	partitioned database			partitioned database
The information needed for distribution design that can be divided into _______	1	2	3	4			4
In horizontal partitioning how many versions are involved ?	1	2	3	4			2
_______ relation is performed using predicates that are defined on that relation	primary horizontal fragments	dervied horizontal fragments	horizontal frag	vertical fragements			primary horizontal fragments
___________ Is the partitioning of a relation that results form predicated being defined on another relation	primary horizontal fragments	dervied horizontal fragments	horizontal frag	vertical fragements			horizontal frag
_________information concerns the global conceptual schema	application	network	computer	database			database
_________ information is required for application	qualitative	quantitative	qualitative and quantitative	qualitative or quantitative			qualitative and quantitative
selection operation on the owner relations of a database schema is known as	primary horizontal fragmentation	horizontal	vertical	network			primary horizontal fragmentation
how many type of heuristic approaches exist in vertical fragmentation	3	2	1	4			2
________ was used later in for distributed databases	grouping	spliting	fragmentation	tuples			grouping
_________distributed enviornment introduced in 1984	grouping	spliting	fragmentation	tuples			spliting
which fragmentation minimizes the execution time of user application?	vertical	horizontal	optimal	active			optimal
______sub relation can be identified and placed in faster memory subsystem	vertical	horizontal	optimal	active			active
______of attributes which indicates how closely related the attributes are	affinity	tuple identifier	active	tuples			affinity

hybrid fragmentation also called as ________	mixed fragmentation	nested fragmentation	mixed and nested fragmentation	mixed or nested fragmentation			mixed and nested fragmentation
The rules for semantic data control must be stored in a ________	semantic data control	data control	catalog	distributed			catalog
A view is a___________ defined as the result of a query.	view & data base relation	control	semantic	database			data base relation
A ________ is a dynamic window in the sense that it reflects all updates to the database	view	management	DBMS	centralized			view
The two main schemes in the data encryption are ______ and _______	standard	public key	standard and public key	unique key			standard and public key
Authorization can be uniformly controlled by whom	system administrator	database administrator	systemanalyst	administrator			database administrator
How many actors are involved in authorization control?	1	2	3	4			4
who trigger the execution of application programs	user	programmer	administrator	designer			user

UNIT-III

SYLLABUS

Query Optimization : Sample Database- Query Processing in Centralized Systems: Query Parsing and Translation - Query Optimization- Query Processing in Distributed Systems- Heterogeneous Database Systems - Concurrency Control in Distributed Database Systems.

KAHE

QUERY OPTIMIZATION

We provide an overview of query processing with the emphasis on optimizing queries in centralized and distributed database environments. It is a well documented fact that for a given query there are many evaluation alternatives. The reason for the existence of a large number of alternatives (solution space) is the vast number of factors that affect query evaluation. These factors include the number of relations in the query, the number of operations to be performed, the number of predicates applied, the size of each relation in the query, the order of operations to be performed, the existence of indexes, and the number of alternatives for performing each individual operation—just to name a few. In a distributed system, there are other factors, such as the fragmentation details for the relations, the location of these fragments/tables in the system, and the speed of communication links connecting the sites in the system. The overhead associated with sending messages and the overhead associated with the local processing speed increase exponentially as the number of available alternatives increases. It is therefore generally acceptable to merely try to find a ―good‖ alternative execution plan for a given query, rather than trying to find the ―best‖ alternative.

A query running against a distributed database environment (DDBE) will have to go through two types of optimization. The first type of optimization is done at the global level, where communication cost is a prominent factor. The second type of optimization is done at the local level. This is what each local DBE performs on the fragments that are stored at the local site, where the local CPU and, more importantly, the disk input/output (I/O) time are the main drivers. Almost all global optimization alternatives ignore the local processing time. When these alternatives were being developed, it was believed that the communication cost was a more dominant factor than the local processing cost. Now, it is believed that both the local query cost and the global communication cost are important to query optimization.

Suppose we have two copies of a relation at two different servers, where the first server is a lot faster than the second server, but at the same time, the connection to the first server is a lot slower than the connection to the second server (perhaps we are closer to the second server). An optimization strategy that only considered communication cost would choose the second server to run the local query. This will not necessarily be the best strategy, due to the speed of the chosen (second) server. The overall time to run a query in a distributed system consists of the time it takes to communicate local queries to local DBEs; the time it takes to run local query fragments; the time it takes to assemble the data and generate the final results; and the time it takes to display the results to the user. Therefore, to study distributed query optimization, we need to understand how a query is optimized both locally and globally.

Here, we introduce the architecture of the query processor for a centralized system first. We then analyze how a query is processed optimally, discussing the optimization techniques in a centralized system. The optimization of queries in a distributed system is explained last. We introduce a simple database that we use in our examples.

SAMPLE DATABASE

We will use a small database representing a bank environment for our examples. This database has five relations: Customer, Branch, Account, Loan, and Transaction. In this database, customers, identified by CID, open up accounts and/or loans in different branches of the bank that are located in different cities. This is indicated by the CID and BNAME foreign keys in the Account and Loan relations. Customers also run transactions against their accounts. This is shown in the Transaction relation by the combined foreign key ―(CID, A#).‖ Later in this chapter, when discussing query optimization alternatives, we will specify the statistics for this database. The following shows the relations of our example bank database.

CUSTOMER (CID, CNAME, STREET, CCITY);

BRANCH (BNAME, ASSETS, BCITY);

KAHE

ACCOUNT (A#, CID, BNAME, BAL);

LOAN (L#, CID, BNAME, AMT);

TRANSACTION (TID, CID, A#, Date, AMOUNT);

QUERY PROCESSING IN CENTRALIZED SYSTEMS

The goal of the query processing subsystem of a DBMS should be to minimize the amount of time it takes to return the answer to a user‘s query. Obviously, the response time metric is the most important to the user. However, there are other cost elements that the system is concerned with, which do not necessarily result in the best response time. For example, the system may decide to optimize the amount of resources it uses to get the answer to a given user‘s query or the answers to queries requested by a group of users. In other cases, we may try to maximize the throughput for the entire system. This may translate into a ―reasonable‖ response time for all queries rather than focusing on the response time for a specific query. These goals are sometime contradictory and do not always mean the fastest response time for a given user or group of users.

In a centralized system, the goals of the query processor may include the following: • Minimize the query response time.

• Maximize the parallelism in the system.

• Maximize the system throughput.

• Minimize the total resources used (amount of memory, disk space, cache, etc.).

The system might be unable to realize all of these goals. For example, minimizing the total resource usage may not yield minimum query response time. It is understood that minimizing the amount of memory allocated to sorting relations can have a direct impact on how fast a relation can be sorted. Faster sorting of the relations speeds up the total amount of time needed to join two relations using the sort–merge strategy. The more memory pages (memory frames) allocated to the sort process the faster the sort can be done, but since total physical

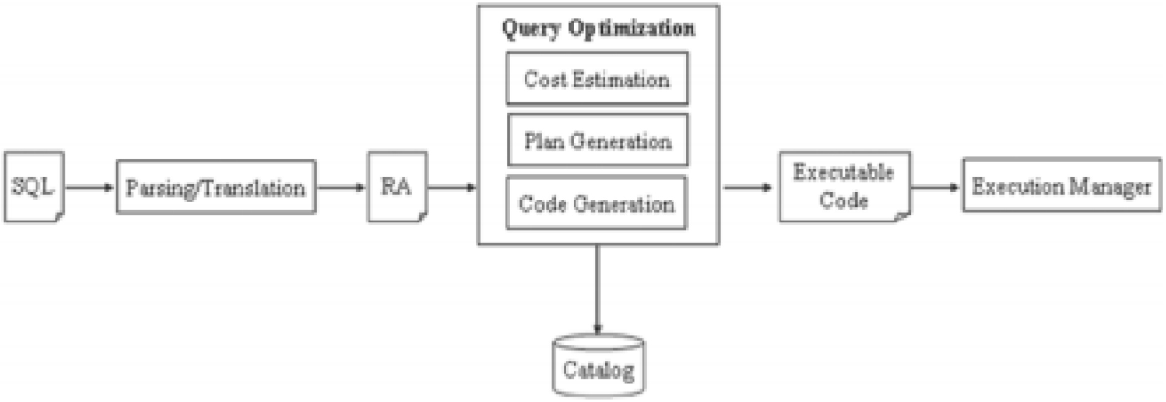
KAHE

memory is limited, increasing the size of sort memory decreases the amount of memory that can be allocated to other data structures, temporary table storage in memory, and other processes. In effect, this may increase the query response time.

The center point of any query processor in a centralized or distributed system is the data dictionary (DD) (the catalog). In a centralized system, the catalog contains dictionary information about tables, indexes, views, and columns associated with each table or index. The catalog also contains statistics about the structures in the database. A system may store the number of pages used by each relation and indexes, the number of rows per page for a given relation, the number of unique values in the key columns of a given relation, the types of keys, the number of leaf index pages, and so on. In a distributed system, the catalog stores additional information that pertains to the distribution of the information in the system. Information on how relations are fragmented, the location of each fragment, the speed of communication links connecting sites, the overhead associated with sending messages, and the local CPU speed are all examples of the details the catalog may contain in a distributed system.

QUERY PARSING AND TRANSLATION

As shown in Figure, the first step in processing a query is parsing and translation. During this step, the query is checked for syntax and correctness of its data types. If this check passes, the query is translated from its SQL representation to an equivalent relational algebra expression.



Query processing architecture of a DBE.

In this Example, Suppose we want to retrieve the name of all customers who have one or more accounts in branches in the city of Edina. We can write the SQL statement for this question as

Select c.Cname From Customer c, Branch b, Account a

Where c.CID = a.CID AND a.Bname = b.Bname AND b.Bcity = ‗Edina‘;

There are two join conditions and one select condition (known as a filter) in this statement. The relational algebra (RA) expression that the parser might generate is shown below:

PJ$_{cname}$ (SL$_{Bcity = \_Edina‘}$ (Customer CP (Account CP Branch)))

The DBMS does not execute this expression as is. The expression must go through a series of transformations and optimization before it is ready to run. The query optimizer is the component responsible for doing that.

QUERY OPTIMIZATION

There are three steps that make up query optimization. These are cost estimation, plan generation, and query plan code generation. In some DBMSs (e.g., DB2), an extra step called ―Query Rewrite‖ is performed before query optimization is undertaken. In query rewrite, the query optimizer rewrites the query by eliminating redundant predicates, expanding operations on views, eliminating redundant sub expressions, and simplifying complex expressions such as nesting. These modifications are carried out regardless of database statistics. Statistics are used in the optimization step to create an optimal plan. Again, an optimal plan may not necessarily be the best plan for the query. The RA expression for Example does not run efficiently, since forming Cartesian products of the three tables involved in the query produces large intermediate relations. Instead, join operators are used and the expression is rewritten as

PJ$_{cname}$ (Customer NJN Account)NJN

(Account NJN (SL$_{Bcity = \_Edina‘}$(Branch)))

KAHE

This expression can be refined further by eliminating the redundant joining of Account relation as

PJ_{cname} (Customer NJN (Account NJN ($SL_{Bcity=\_Edina‘}$ (Branch))))

There are other equivalent expressions that can also be used. All available alternatives are evaluated by the query optimizer to arrive at an optimal query expression.

Cost Estimation Given a query with multiple relational algebra operators, there are usually multiple alternatives that can be used to express the query. These alternatives are generated by applying the associative, commutative, idempotent, distributive, and factorization properties of the basic relational operators. These properties are outlined below (the symbol ―≡‖ stands for equivalence):

• Unary operator (Uop) is commutative:

Uop1(Uop2(R)) ≡ Uop2(Uop1(R))

For example,

$SL_{Bname = \_Main‘}$ ($SL_{Assets > 12000000}$ (Branch) ≡

$SL_{Assets > 12000000}$ ($SL_{Bname = \_Main‘}$ ((Branch))

• Unary operator is idempotent:

Uop((R)) ≡ Uop1(Uop2((R))

For example,

$SL_{Bname = \_Main‘\ AND\ Assets > 12000000}$ (Branch) ≡

$SL_{Bname = \_Main‘}$ ($SL_{Assets > 12000000}$ (Branch)

• Binary operator (Bop) is commutative except for set difference:

KAHE

R Bop1 S ≡ S Bop1 R

For example,

Customer NJN Account ≡ Account NJN Customer

- Binary operator is associative:

R Bop1 (S Bop2 T) ≡ (R Bop1 S) Bop2 T

For example,

Customer NJN (Account NJN Branch) ≡ (Customer NJN Account) NJN Branch

- Unary operator is distributive with respect to some binary operations:

Uop(R Bop S) ≡ (Uop(R)) Bop (Uop(S))

For example,

$SL_{sal > 50000}$ ($PJ_{Cname, sal}$ (Customer)

UN $PJ_{Ename, sal}$ (Employee)) ≡

($SL_{sal > 50000}$ ($PJ_{Cname, sal}$ (Customer))

UN (($SL_{sal > 50000}$ ($PJ_{Ename, sal}$ (Employee)))

- Unary operator can be factored with respect to some binary operation:

(Uop(R)) Bop (Uop(S)) ≡ Uop(R Bop S)

For example,

$SL_{sal > 50000}$ ($PJ_{Cname, sal}$ (Customer))

UN ($SL_{sal > 50000}$ ($PJ_{Ename, sal}$ (Employee))) ≡

$SL_{sal > 50000}$

KAHE

(PJ$_{\text{Cname, sal}}$ (Customer)) UN (PJ$_{\text{Ename, sal}}$ (Employee))

This is the inverse of the distributive property. Applying the above properties to the operators of an RA query can create multiple equivalent expressions. For example, the following eight expressions are equivalent. It is the responsibility of the query optimizer to choose the most optimal alternative.

Alt1: (SL$_{\text{Bname = \_Main‘}}$ (Account)) NJN Branch

Alt2: Branch NJN (SL$_{\text{Bname = \_Main‘}}$ (Account))

Alt3: (SL$_{\text{Bname = \_Main‘}}$(Branch)) NJN Account

Alt4: Account NJN (SL$_{\text{Bname = \_Main‘}}$(Branch))

Alt5: SL$_{\text{Bname = \_Main‘}}$ (Account NJN Branch)

Alt6: SL$_{\text{Bname = \_Main‘}}$ (Branch NJN Account)

Alt7: (SL$_{\text{Bname = \_Main‘}}$ (Account)) NJN (SL$_{\text{Bname = \_Main‘}}$(Branch))

Alt8: (SL$_{\text{Bname = \_Main‘}}$(Branch))NJN(SL$_{\text{Bname = \_Main‘}}$(Account))

For query optimization discussion, it is more convenient to use a query tree instead of the RA expression. A query tree is a tree whose leaves represent the relations and whose nodes represent the query‘s relational operators. Unary operators take one relation as input and produce one relation as output, while binary operators take two relations as input and produce one relation as output. That is why every operator‘s output can be fed into any other operator. Results from one level‘s operators are used by the next level‘s operators in the tree until the final results are gathered at the root. All operators have the same relation-based interface for input and output. This helps with a uniform interface implementation for the query execution manager. This uniformity has given rise to the use of this operator model in most commercial systems such as System R (DB2), Oracle, Informix, and Ingres. In such systems, each operator‘s implementation is based on the concept of an iterator.

KAHE

Each iterator has a set of functions (methods in object-orientated terminology) that can be called when needed. An iterator acts as a producer for the iterator at the next level and as a consumer for the iterator(s) above it— remember in our representation of a query tree the leaves are on the topmost level. An iterator has three functions—to prepare, to produce, and to wrap-up production. These functions are called ―Open,‖ ―Get_Next,‖ and ―Close.‖ For example, to perform a select using a scan access path, the operator opens the file that contains the relation to be scanned and allocates enough input and output memory buffers for the operation. Once opened, the Get_Next function is called repeatedly to process tuples from the input buffer(s), it qualifies them and puts them in the output buffer(s). Once there are no more tuples to process, the Close function is activated to close the file and deallocate the memory buffers. Note that although the Open and Close functions for all iterators perform similar tasks, depending on the operator, the work that Get−Next function has to do may vary tremendously. For instance, a sort–merge to join two relations requires reading the pages for each relation into memory, sorting them, writing them out, bringing them back, and merging them to produce the output, as explained in Section 4.3. One of the advantages of the iterator model is that it naturally supports pipelining, where the output tuples of one operator can be fed as input into another operator without needing to store them on the disk (materializing them).

For a given query expression, the query optimizer analyzes all equivalent query trees that represent the solution space for the query. For the SQL query used in Example, the solution space (not considering the trees that produce Cartesian products) consists of five query trees that can be used to produce the same results as shown in the following Figure.

The optimizer analyzes all these trees in the solution space and selects a tree that is optimal for the query. The first step in the optimization is pruning. In the pruning step, the ―bad‖ alternative trees are eliminated. The optimizer usually uses a small set of rules for pruning. One such rule requires that ―before joining two relations, the relations should be reduced in size by applying the select and the project operations.‖ Considering this rule, alternatives 2 and 5 perform better than the others do, since they perform the select operation before the join operation. In the next step, the optimizer selects trees 2 and 5 and runs them through the cost analysis. In the cost analysis step, statistics are used to determine/estimate the cost of each candidate tree. In this step, the tree with the smallest cost is chosen.

The following are statistics about our database:

• There are 500 customers in the bank.

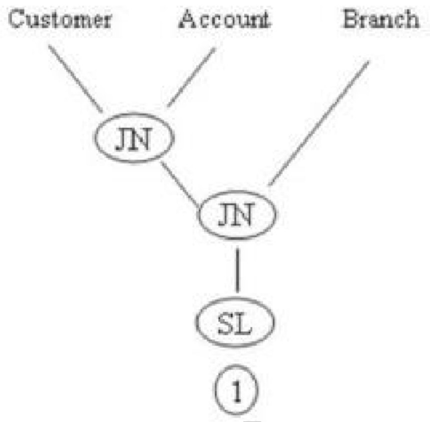
• On average, each customer has two accounts.

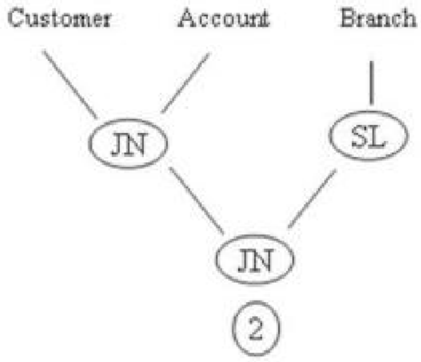
• There are 100 branches in the bank.

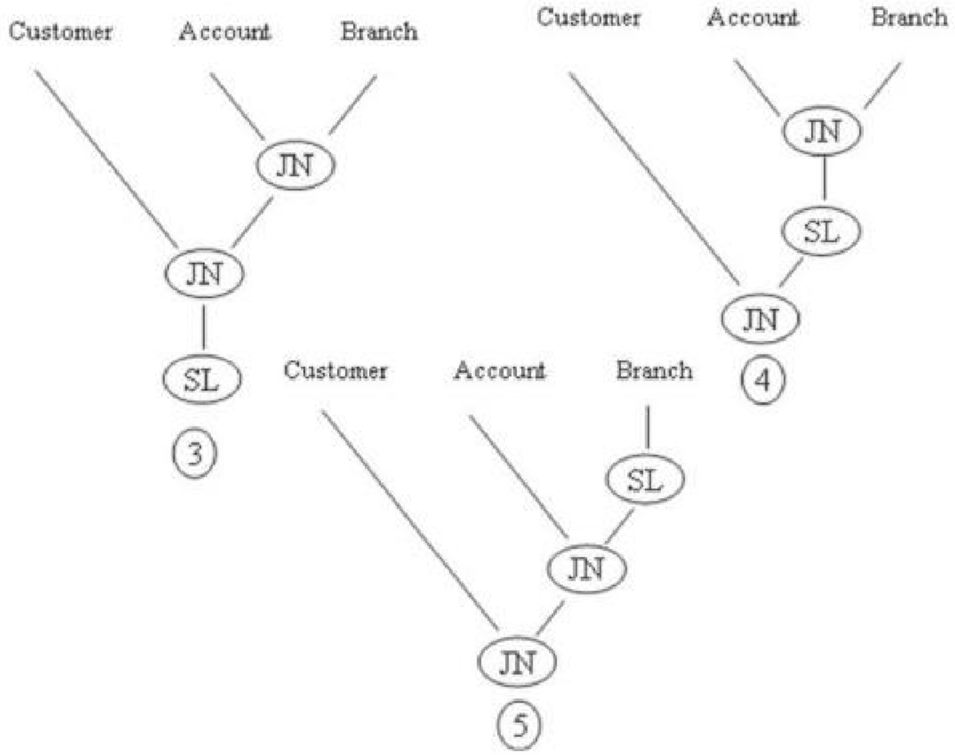
• There are 10 branches in Edina city.

• Ten percent of customers have accounts in the branches in Edina.

Since there are 1000 accounts in the bank, and 50 customers have accounts in the branches in Edina resulting in 100 accounts in that city. Let‘s also assume that it takes ―t‖ units of time to process each tuple of each relation in memory. Using the statistics



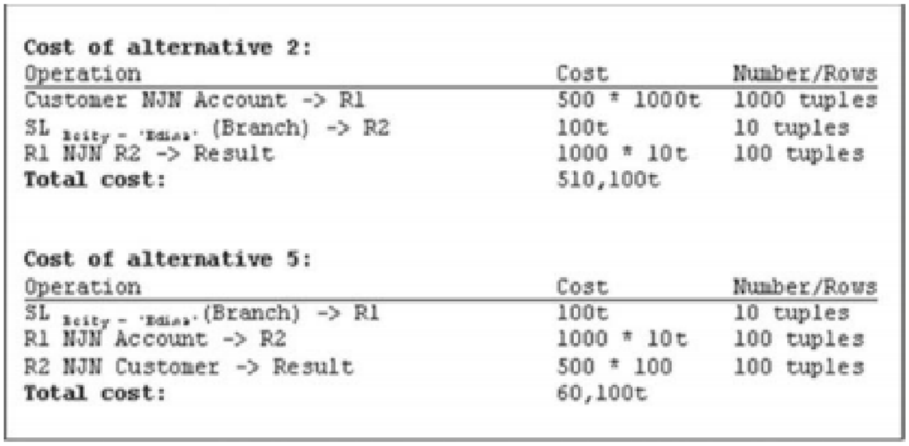




Alternative query tree for the Example

We can calculate the cost of the two chosen alternatives based on the number of rows that each one processes. Figure 4.6 shows the cost analysis for these two alternatives.

Analysis of Alternative 2: The first operation in this alternative is the join between the Customer and the Account relations. The cost of this join is ―500 * 1000t,‖ since there are 500 tuples in the Customer relation and 1000 tuples in the Account relation. The join results in the temporary relation ―R1.‖ R1 has 1000 tuples, since every account must belong to a customer. The second operation selects those branches that are in city



Cost of alternative 2:

Operation	Cost	Number/Rows
Customer NJN Account -> R1	500 * 1000t	1000 tuples
SL $_{Bcity = 'Edina'}$ (Branch) -> R2	100t	10 tuples
R1 NJN R2 -> Result	1000 * 10t	100 tuples
Total cost:	510,100t	

Cost of alternative 5:

Operation	Cost	Number/Rows
SL $_{Bcity = 'Edina'}$ (Branch) -> R1	100t	10 tuples
R1 NJN Account -> R2	1000 * 10t	100 tuples
R2 NJN Customer -> Result	500 * 100	100 tuples
Total cost:	60,100t	

Cost analysis for Example

Edina as temporary relation ―R2.‖ Since there are 100 branches in the bank, the select cost is 100t and R2 has 10 tuples in it. The last operation joins R1 and R2, picking up the customers for accounts in Edina‗s branches. The cost of this join is ―1000 * 10t.‖ Since there are 100 accounts in branches in Edina, it returns 100 tuples. The total cost of this alternative therefore is 510,100t.

Analysis of Alternative 5: The first operation in this alternative is the select with the cost of 100t resulting in relation ―R1‖ with 10 tuples. The second operation joins the Account relation with R1, storing the accounts that are in Edina in the temporary relation ―R2.‖ This operation‗s cost is ―1000 * 10t‖ and it returns 100 tuples. The last operation joins Customer and R2, picking up the customers for the accounts in Edina. The cost of this operation is ―500 * 100t‖ and it returns 100 tuples, which represent the accounts in Edina.

Since query tree 5‗s cost is smaller than query tree 2‗s cost, alternative 5 is chosen for the query. We could have arrived at the same conclusion by noticing that we join the Account and Customer relations, which are the largest relations in the database, first in query tree 2—resulting in a higher cost. As a general rule, most DBMSs postpone joining the larger relations until the select operations and other joins have reduced the number of tuples in the join operands to a more manageable number.

KAHE

KAHE

Plan Generation

In this optimization step, the query plan is generated. A query plan (or simply, a plan, as it is known by almost all DBMSs) is an extended query tree that includes access paths for all operations in the tree. Access paths provide detailed information on how each operation in the tree is to be performed. For example, a join operation can have an access path that indicates the use of the block-nested loop join, hash-join, or sort–merge join, while a select access path can specify the use of an index (a B+Tree index or a hash index) or a table scan.

In addition to the access paths specified for each individual RA operator, the plan also specifies how the intermediate relations should be passed from one operator to the next—materializing temporary tables and/or pipelining can be used. Furthermore, operation combinations may also be specified. For projected out while the tuple is still in memory. This eliminates reprocessing qualified rows in a project operation after a select. Other examples of operator combination include select and join; select and union; project and join; project and union; and select, project, and join.

Query optimization has been researched for many years and is still the focus of research due to its importance in centralized and distributed DBEs. Steinbrunn provides an excellent overview of an alternative approach to query optimization. Among the proposed algorithms, exhaustive search and heuristics-based algorithms are most popular. The difference between these two approaches is in the time and space complexity requirements and superiority of the plans they generate. We will review these two approaches to optimization next.

Exhaustive Search Optimization Algorithms in this class will first form all possible query plans for a given query and then select the ―best‖ plan for the query. Dynamic programming is an example of an algorithm in this class. Since the solution space containing all the possible query execution alternatives using DP is very large—it has an exponential time-and-space complexity. Many different DP algorithms attempting to reduce the solution space and lower the time and/or complexity have been proposed.

Heuristics-Based Optimization Originally, most commercial DBMSs used heuristics based approaches, which are also known as rule-based optimization (RBO) approaches, as their optimization technique. The algorithms in this class have a polynomial time-and-space complexity as compared to the exponential complexity of the exhaustive search-based algorithms, but the heuristics-based algorithms do not always generate the best query plan. In this approach, a small set of rules (heuristics) are used to order the operators in an RA expression without much regard to the database‗s statistics. One such heuristic is based on the observation that if we reduce the size of the input relations to a join, we reduce the overall cost of the operation. This rule is enforced by applying the select and/or project operations before the join. This rule is also known as ―pushing‖ the select and project toward the leaves of the query tree. Another heuristic joins the two smallest tables together first. This is based on the belief that since not all tuples from the relations being joined qualify the join condition, fewer tuples will be used in the next operation of the query tree, resulting in a smaller overall cost for the query. Another popular heuristic is simply to disallow/avoid using the cross-product operation, which is based on the fact that the intermediate result of this operation is usually a huge relation.

These rules (and others like them) are used in many of today‗s commercial systems. In DB2, for example, there is a rule that forbids/avoids the use of an intermediate relation as the right operand of a join operation. In Oracle 10g, the rule-based optimizer (RBO) associates weights with each operation and uses the operation with the smallest weight first. For example, Oracle associates a weight of 5 with a select operation that uses a unique index, while it assigns a weight of 20 to a relation scan. As a result, when given the choice, the RBO will use the index lookup to find a single tuple rather than scanning the table. Oracle‗s RBO generates the query plan by applying this rule repeatedly to all operations in the query tree. One advantage of using a small set of rules like these is that the optimizer can generate a plan quickly. Although Oracle‗s RBO example, select and project operations can be combined. In this case, after a tuple is qualified, unwanted attributes are is fast, because it applies rules statically—without considering available statistics—it does not always generate an optimal plan. For instance, perhaps the RBO only considers nested-loop joins for a particular query, even when a sort–merge join or a hash-join would perform better for the current state of the database.

KAHE

KAHE

Dynamic Programming

Dynamic programming (DP) is an exhaustive search-based algorithm used today by most commercial DBMSs. The approach uses a cost model, such as query response time, to generate a plan with the optimal cost. This approach is also known as cost-based optimization (CBO).

For the sake of simplicity when discussing dynamic programming, let‗s consider the case of joining N relations together in an N-way join, that is, ―R1 JN R2 ... JN RN−1 JN RN.‖ The number of alternatives for joining the N relations is ―(2*(N−1))!/(N−1)!‖ For example, there are 12 alternatives for joining three tables, 120 alternatives for joining four tables, and 1680 alternatives for joining five tables. Dynamic programming uses a cost model to dynamically reduce the solution space and generates an optimal plan by building the plan one step at a time, from the bottom–up. The algorithm iterates over the number of relations that have been joined so far and prunes the alternatives that are inferior (cost more), keeping only the least costly alternatives. In addition to keeping optimal alternatives, the algorithm may also keep plans that generate a sorted intermediate relation. These plans are of an ―interesting order‖ and are kept because having a sorted intermediate relation helps the next join, encouraging it to apply a sort–merge join as discussed before (since the intermediate result is already sorted, the overhead is less than it would be otherwise). To illustrate how DP works, let‗s examine a few steps in the DP processing of this N-way join.

Step 1: Generate All 1-Relation Plans. This step generates the access plans for each of the relations involved in the query. Select and project are the only operations we consider in this step. All other operations are binary operations, which involve more than one relation by definition. When considering 1-relation plans, the access paths for each select and project operation are evaluated and we choose the path with the smallest cost, discarding the others. For instance, we know that a predicate of the form ―attribute = constant‖ for a uniquely valued attribute will return either one tuple or zero tuple. In this case, if a unique index has been created on the attribute, we have an access path based on a unique index lookup. Alternatively, we could scan the relation to find the matching tuple. It should be obvious that between these two access paths, the unique index lookup is faster than the scan, especially for large relations that occupy many disk pages. Therefore, the plan that utilizes the index is chosen for the select operation using this relation. Let‘s assume that the optimal access plans, P1, P2,... , PN, are kept for relations R1, R2, ... , RN correspondingly.

Step 2: Generate All 2-Relation Plans. In this step, the algorithm combines all the 1-relation plans we kept from Step 1 into 2-relation plans and chooses those with the smallest cost for each pair of determined by the different access paths available for the join. The 2-relation plans with the smallest cost are chosen in this step. For example, we know that there are two alternatives for joining R1 and R2, based on the order of operands——R1 JN R2‖ and ―R2 JN R1.‖ For each alternative, it is also possible to join the two relations based on any of the join strategies we discussed in Section 3.4. Let‘s assume that our system only supports nested-loop row-based joins and sort–merge joins. For this system, there will be two ways to join R1 and R2 and two ways to join R2 and R1. Depending on the availability of indexes keyed on the join attributes of the two relations, the nested-loop join can be implemented in two different ways—one using the index and one not using the index. As a result, for each pair of relations, this system has to consider six different join alternatives (each operand ordering alternative can use one of the two nested-loop join alternatives or the sort–merge alternative). Since the sort–merge join produces a sorted intermediate relation, this plan is kept as an interesting order plan, as we discussed earlier. When we consider the two nested-loop join alternatives, we only keep the plan with the smallest cost if its cost is less than the cost of the sort–merge join. Assuming that we only keep one plan for each pair of relations, we can indicate these plans as P12, P13,... , P1N, P21,... , PNN−1, where the two digits in the indexes refer to the relations and the order of their join. In other words, the plan we keep for the join between R1 and R2 is called ―P12,‖ while the plan we keep for the join between R2 and R1 is called ―P21.‖

Step 3: Generate All 3-Relation Plans. In this step, the algorithm combines all the 2- relation plans from Step 2, with the 1-relation plans from Step 1 to form 3-relation plans. For example, assume the query calls for ―R1 JN R2 JN R4.‖ Furthermore, assume that the 2-relation plan P12, which joins R1 and R2 and produces R12, is kept from Step 2. In this case, ―R12 JN R4‖ and ―R4 JN R12‖ must be considered. For each one of these alternatives, we consider all the access paths for the join and keep the optimal plan along with any plans with interesting orders.

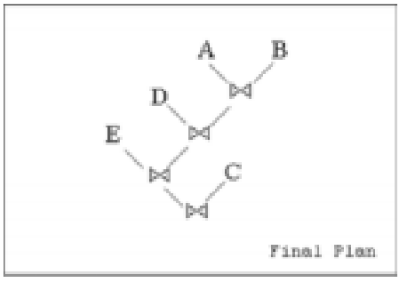
KAHE

Step 4: Generate All 4-Relation Plans. In this step, the algorithm joins the 3-relation plans from Step 3 with the 1-relation plans from Step 1 to form 4-relation plans. The algorithm also joins all the 2-relation plans from Step 2 with each other to form 4-relation plans. Again, for each of these alternatives, we consider all the access paths for each join, keeping the optimal plan and the plans with interesting orders.

Subsequent Steps: The pattern is repeated for each subsequent step, to incrementally create bigger plans until all N relations have been joined.

Reducing the Solution Space

Dynamic programming generates a large solution space even for a small number of relations. Therefore, it is fairly common for the query optimizers to sacrifice some of the query response time in return for faster query plan generation by reducing the solution space. This means the optimizer tries to find an optimal plan and not necessarily the best plan. The optimizer may actually miss some better plans trying to save time to generate the plan.



Dynamic programming final plan for Example

For instance, the optimizer may only consider a left-deep (or a right-deep) query tree instead of a bushy tree when analyzing different access paths as depicted in the above Figure. Although a bushy tree may perform better in some situations, the left-deep plan lends itself better to do the operations on-the-fly. The left-deep plan is the only type of tree that DB2 allows. Using a left-deep query plan, the execution manager can perform the second and third joins in our example on-the-fly. As the tuples from the previous join are produced, they are joined with the relation at the next level. This eliminates the need to materialize the results of the joins that are necessary for the bushy plan. For large relations, the intermediate join results could potentially be large and could not be kept in memory for the next step. Storing these large intermediate relations on disk can take a considerable number of disk I/Os.

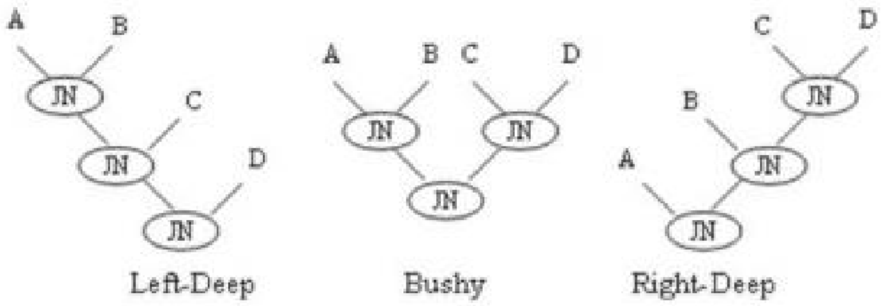
Even considering only one shape for the query tree does not reduce the solution space dramatically. Consider the case of having to join three relations: A, B, and C. With a left-deep type of tree, joining these relations can be done in six different ways, as depicted in Figure.

In this example, we assume that any relation can be joined with any other relation. In reality, this may not be the case; that is, A can be joined with B if there is a common attribute between A and B but not with C if there are no common attributes between A and C. Other alternatives to reduce solution space and the cost-and-time complexity of DP are iterative dynamic programming (IDP) and the greedy approach, both of which we will review next.

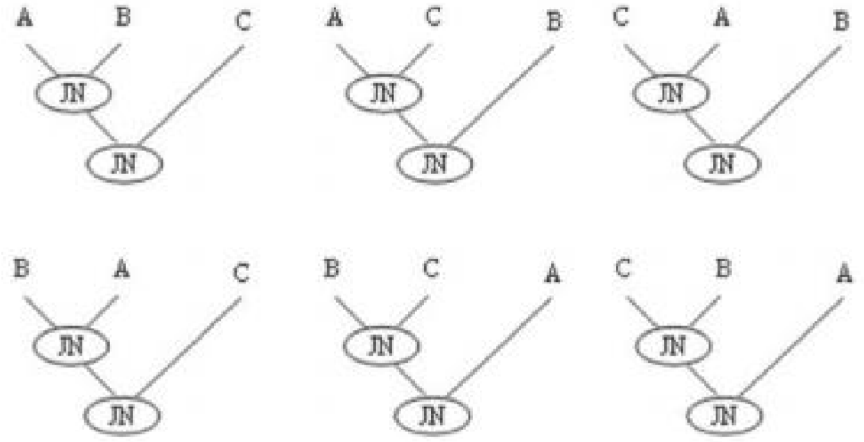
Iterative Dynamic Programming

The main issue with dynamic programming is the space-and-time complexity of the algorithm. DP selects the best possible approach by forming all possible alternatives for joining the relations in a query from the bottom up. The time and space requirements of DP grow exponentially as the number of relations grows. For a large number of relations, it is fairly common for the algorithm to exhaust the available memory since DP keeps all the plans generated throughout the entire process. Thrashing—the constant exchange of information between the memory and the disk—starts when the computer runs out of memory. One alternative that specifically addresses the issue with the size of memory is called iterative dynamic programming (IDP). The IDP algorithm divides the optimization process into a set of iterations. In each iteration, the algorithm only enumerates up to k-relation plans and then stops. Parameter ―k‖ is carefully defined ahead of time to prevent exhausting the available memory based on the number of relations in the query.

KAHE



Left-deep, bushy, and right-deep plan examples.



Left-deep trees to join three relations.

For example, if ―k = 3‖, then in iteration one the algorithm forms only 1-relation, 2-relation, and 3-relation plans. From the first iteration the best 3-relation plan is selected. All 1-relation and 2-relation plans that include one or more of the three relations participating in the chosen 3-relation plan are discarded. Other plans are carried over to the second iteration. Assuming the same five relations that we used in Example, Iteration 1 of IDP generates the same plans as generated by DP for up to 3-relation plans. From these plans, the plans shown in Figure a are kept for Iteration 2. In Iteration 2, the 2-relation plans are formed. Note that although these are called 2-relation plans, in reality they may contain more than two relations. If two base relations are joined the plan is a 2-relation plan. If one 2-relation plan is combined with one 3-

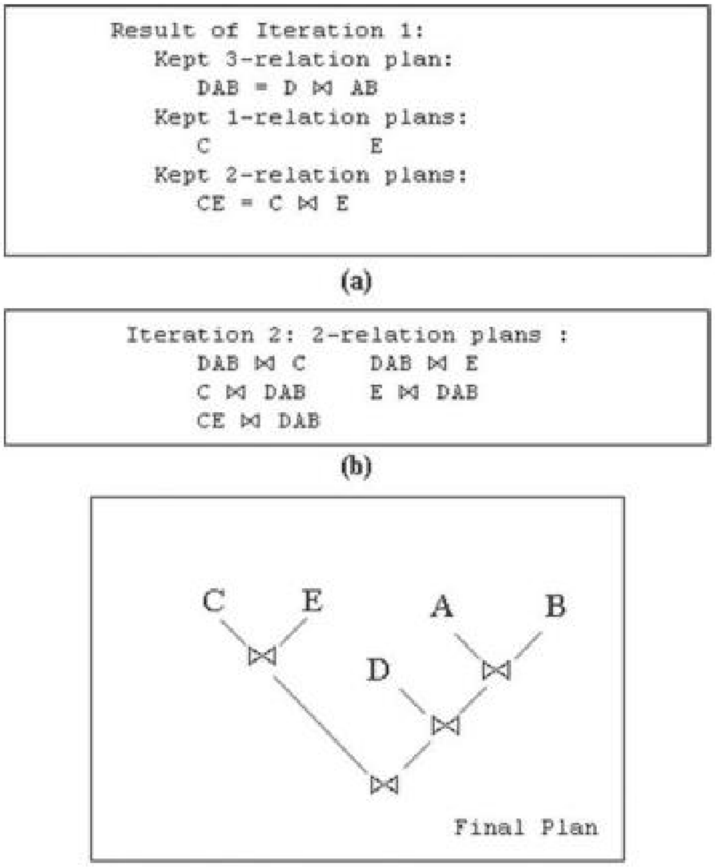
relation plan, the result is a 5-relation plan. Figure b indicates all the 2-relation plans generated from Iteration 2 for our example. As seen from this figure, ―CE JN DAB‖ is actually the 5-relation plan that we are seeking. Figure c shows the final query plan based on this strategy.

The basic strategy behind IDP is this: if we keep the cheapest k-way plan in each step, the final plan costs the least based on the chosen plans. What we must keep in mind is that since we do not consider all possible plans in each step, IDP may actually miss the optimal overall plan.

Greedy Algorithms

Greedy algorithms have a much smaller time-and-space complexity requirement compared to dynamic programming. However, they do not necessarily produce superior plans. One can categorize greedy algorithms as a special case of IDP with k = 2. Greedy algorithms behave exactly the same as IDP except that only 1-relation and 2-relation plans are formed from plans kept from the previous iteration. Let‘s apply the greedy algorithm to joining five relations A, B, C, D, and E. For the first iteration, the greedy algorithm generates the same plans as generated by DP for up to 2-relation.

KAHE



```
Result of Iteration 1:
   Kept 3-relation plan:
      DAB = D ⋈ AB
   Kept 1-relation plans:
      C           E
   Kept 2-relation plans:
      CE = C ⋈ E
```

(a)

```
Iteration 2: 2-relation plans :
     DAB ⋈ C     DAB ⋈ E
     C ⋈ DAB     E ⋈ DAB
     CE ⋈ DAB
```

(b)

(c)

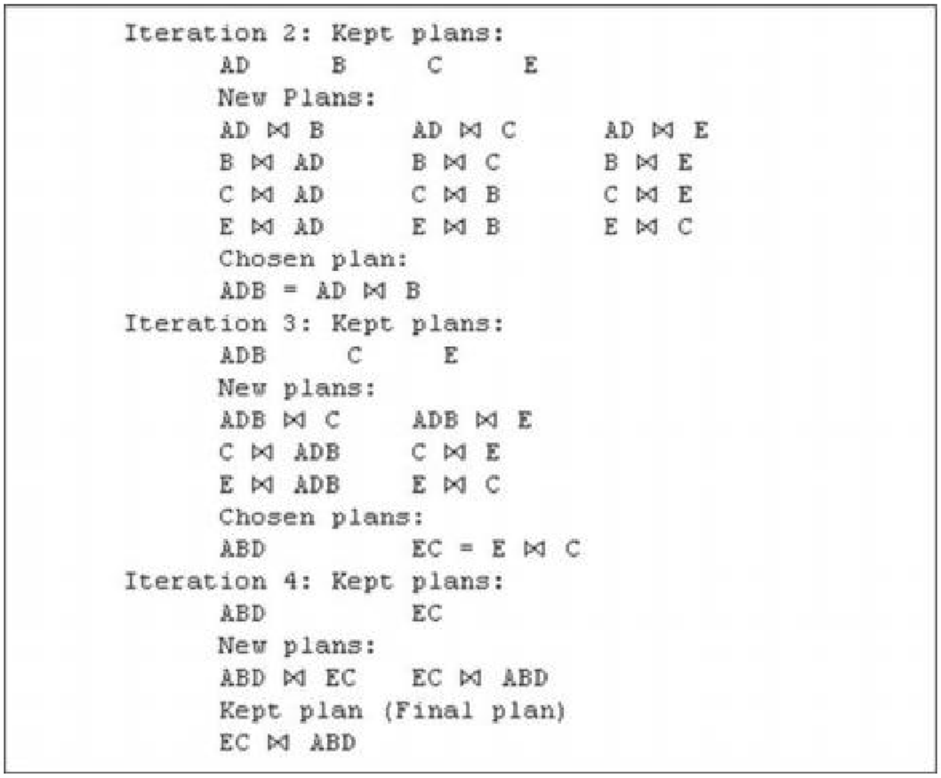
Iterative dynamic programming results for Example

At this point, as shown in Figure, the greedy algorithm breaks and selects the best 2-relation plans and all 1-relation plans that do not include a relation in the chosen 2-relation plan.

The above Figure also shows the 3-relation, 4-relation, and 5-relation plans that the greedy algorithm generates in the second, third, and fourth iterations, respectively. The final plan, which is shown in Figure, is chosen from all the 5-relation plans that we formed.

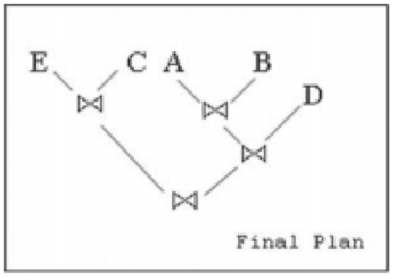
Code Generation

The last step in query optimization is code generation. Code is the final representation of the query and is executed or interpreted depending on the type of operating system or hardware. The query code is turned over to the execution manager (EM) to execute.



```
Iteration 2: Kept plans:
      AD     B     C     E
      New Plans:
      AD ⋈ B      AD ⋈ C      AD ⋈ E
      B ⋈ AD      B ⋈ C       B ⋈ E
      C ⋈ AD      C ⋈ B       C ⋈ E
      E ⋈ AD      E ⋈ B       E ⋈ C
      Chosen plan:
      ADB = AD ⋈ B
Iteration 3: Kept plans:
      ADB     C     E
      New plans:
      ADB ⋈ C     ADB ⋈ E
      C ⋈ ADB     C ⋈ E
      E ⋈ ADB     E ⋈ C
      Chosen plans:
      ABD         EC = E ⋈ C
Iteration 4: Kept plans:
      ABD         EC
      New plans:
      ABD ⋈ EC    EC ⋈ ABD
      Kept plan (Final plan)
      EC ⋈ ABD
```

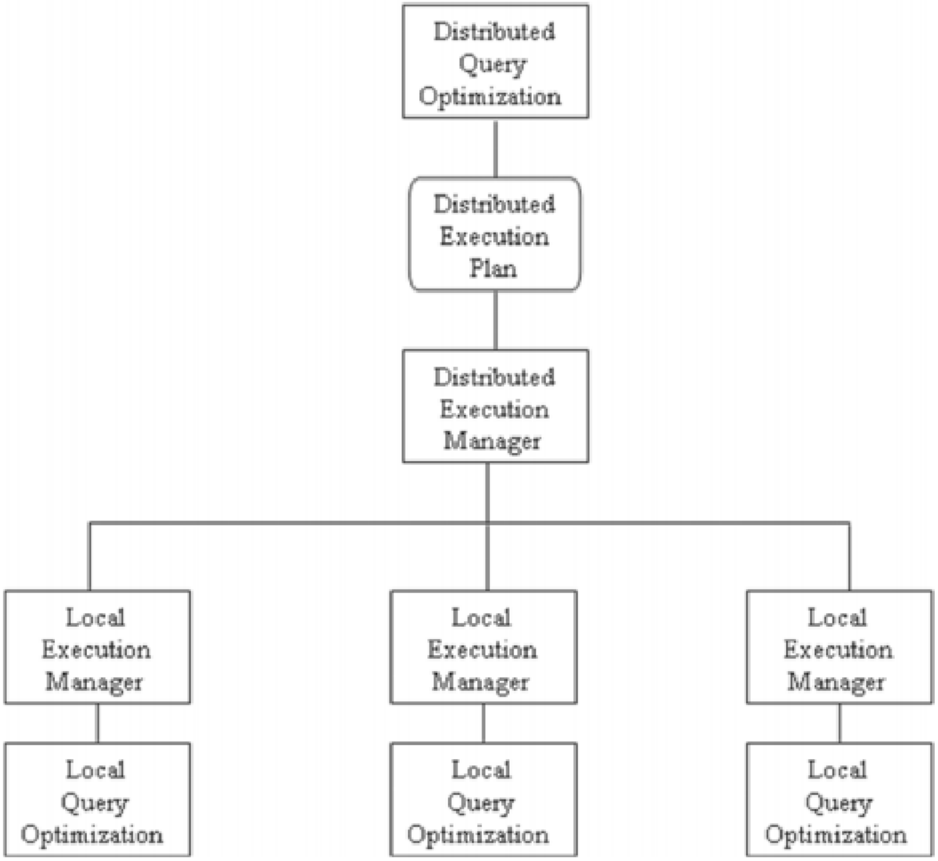
Greedy algorithm applied to Example



Plan generated by greedy algorithm for Example

QUERY PROCESSING IN DISTRIBUTED SYSTEMS

There are two distinct categories of distributed systems: (1) distributed homogeneous DBE (what we called a DDBE) and (2) distributed heterogeneous DBE (what we called a MDB). In both of these systems, processing a query consists of optimization and planning at the global level as well as at the local database environment (LDBE) level. The Figure depicts how queries are processed in a DDBE.



Distributed query processing architecture.

The site where the query enters the system is called the client or controlling site. The client site needs to validate the user or application attempting to access the relations in the query;

to check the query‘s syntax and reject it if it is incorrect; to translate the query to relational algebra; and to globally optimize the query.

Mapping Global Query into Local Queries

The relations used in the global query may be distributed (fragmented and/or replicated) across multiple local DBEs. Each local DBE only works with the local view of the information at its site and is unaware of how the data stored at its site is related to the global view. It is the responsibility of the controlling site to use the global data dictionary (GDD) to determine the distribution information and reconstruct the global view from local physical fragments.

Distributed Query Optimization

Global optimization is greatly impacted by the database distribution design. Just like a local query optimizer, a global query optimizer must evaluate a large number of equivalent query trees, each of which can produce the desired results. In a distributed system, the number of alternatives increases drastically as we apply replication, horizontal fragmentation, and vertical fragmentation to the relations involved in the query.

Utilization of Distributed Resources

In a distributed system, there are many database servers available that can perform the operations within a query. There are three main approaches on how these resources are utilized. The difference among these approaches is based on whether we perform the operation where the data is; send the data to another site to perform the operation; or a combination of the two.

Operation Shipping

In this approach, we run the operation where the data is stored. In order to achieve this, the local DBE must have a database server that can perform the operation. A good example of the kind of operation that lends itself nicely to operation shipping is any unary operation such as relational algebra‘s select (SL) and project (PJ). Since a unary operation works on only one

KAHE

relation, the most logical and most economical in terms of communication cost is to run the operation where the data fragment is stored. Even for binary operations, such as join and union, we prefer operation shipping if the two operands of the operation are stored at the same site. Most of today‘s DBMSs such as Oracle, SQL Server, and DB2 prefer operation shipping. In these systems, an interactive query that enters the system at a client computer does not run at the client but at the database server, where the data is stored. The results are then transferred back to the client. These systems store canned queries such as stored procedure on the database server as well. The database server activates these stored procedures upon request from an application or an interactive user at the server, simulating operation or query shipping.

Data Shipping

Data shipping refers to an alternative that sends the data to where the database server is. This obviously requires shipment of data fragments across the network, which can potentially lead to a high communication cost. Consider the case of having to select some tuples from a relation. We can perform the SL operation where the data is as explained in operation shipping. We can also send the entire relation to another site and perform the SL operation there. One can argue that in a high-speed fiber optic communication network, it might be faster to send the entire relation from a low-speed processor to a high-speed processor and perform the SL operation there.

Data shipping is also used in object-oriented databases as a preferred approach. Pagebased data caching, close to where the use is, will presumably speed up the response time for queries that run frequently when data does not change often. A similar caching approach is proposed for an index lookup by Lomet. An index page fault happens when data required by a query is not in the cache. In this case, the page containing the data is brought into memory. If the cache is full, the least recently used page will be written back to the disk to make room available for the required page.

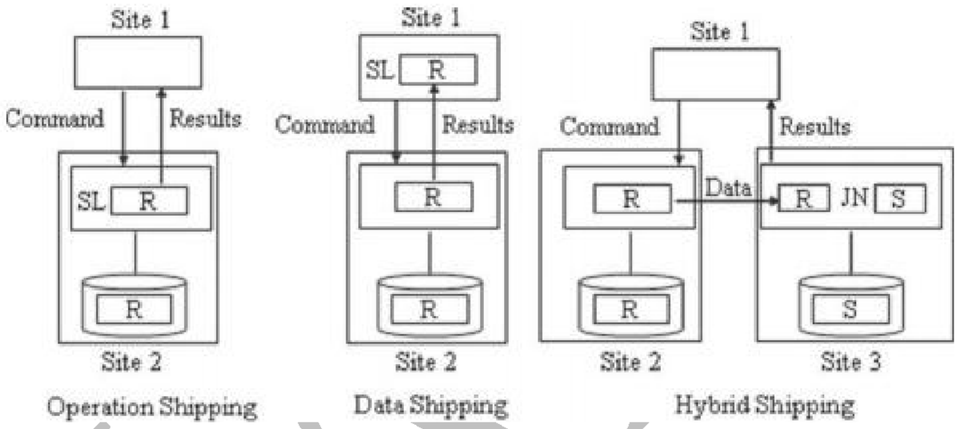
Hybrid Shipping

KAHE

Hybrid shipping combines data and operation shipping. This approach takes advantage of the speed of database servers to perform expensive operations such as joins by transferring the smaller relation to the server where the larger relation resides.

Dynamic Programming in Distributed Systems

The dynamic programming principles we discussed for centralized systems can also be used for query optimization in distributed systems. The following example applies dynamic programming to a distributed database.



Operation, data, and hybrid shipping examples.

Query Trading in Distributed Systems An alternative to dynamic programming that is based on the principles of trading negotiation is called query trading (QT), which was proposed by Pentaris and Ioannidis. In the QT algorithm, the control site for a distributed query is called the buyer and each site where a local query runs is called a seller. The buyer has different alternatives for choosing sites where data fragments are and also different choices for reconstructing the global results. In this algorithm, the buyer tries to achieve the optimal total cost for the query by negotiating services from different local sellers.

For example, assume a relation is replicated at two sites. When the buyer asks each seller about the cost of performing a select operation on a given relation, each seller (local server) bids on the select based on the facilities and services that are locally available. The difference between the prices that two sellers offer for the same select operation on the same relation is due to differences between each local server‘s processing speed, workload, method of local optimization, and available indexes. The buyer decides on which seller‘s offer to choose to lower the overall cost of the query. A cost function is used by the buyer to choose the right seller(s). As an example, assume we need to join relations R1 and R2. Suppose Site 1 has already been chosen to perform a select on R1. In the next step the buyer needs to decide where the join operation is going to be performed. It might be cheaper to join R1 and R2 at Site 1 by sending R2 to this site than sending the results of the select from Site 1 to Site 2 where the join will be performed. The buyer‘s decision in this case depends on the prices that Site 1 and Site 2 offer for the join operation. In subsequent steps, the buyer negotiates with chosen sellers to see if they can perform a larger portion of the query. This process is repeated until the entire query execution is planned.

The query trading algorithm works by gradually assigning subqueries to local sites. A site may be a buyer, a seller, or both a buyer and a seller depending on the data stored at the site and/or the query that is being performed. During the negotiation, the buyer simply puts the plan together from the bottom up. The query is not actually executed until the overall global plan has been determined. The goal of the algorithm is to create the most optimal plan from local optimized query plans that have been proposed by the sellers together with the communication cost of reconstructing the final results.

KAHE

Distributed Query Solution Space Reduction

Both dynamic programming and query trading algorithms have to deal with a large number of alternatives even for simple queries. Attaining the absolute best distributed query plan, therefore, is too expensive to achieve. The following is a summary of the rules that can be applied to a query tree in a distributed system to reduce its solution space.

- Apply Select and Project as Soon as Possible
- Simplify Operations on Horizontal Fragments
- Perform Operations Where Most of the Data Is

- Simplify Join Operations
- Materialize Common Sub expressions Once
- Use Semi-join to Reduce Communication Cost
- Use Bit Vectors to Reduce Communication Cost

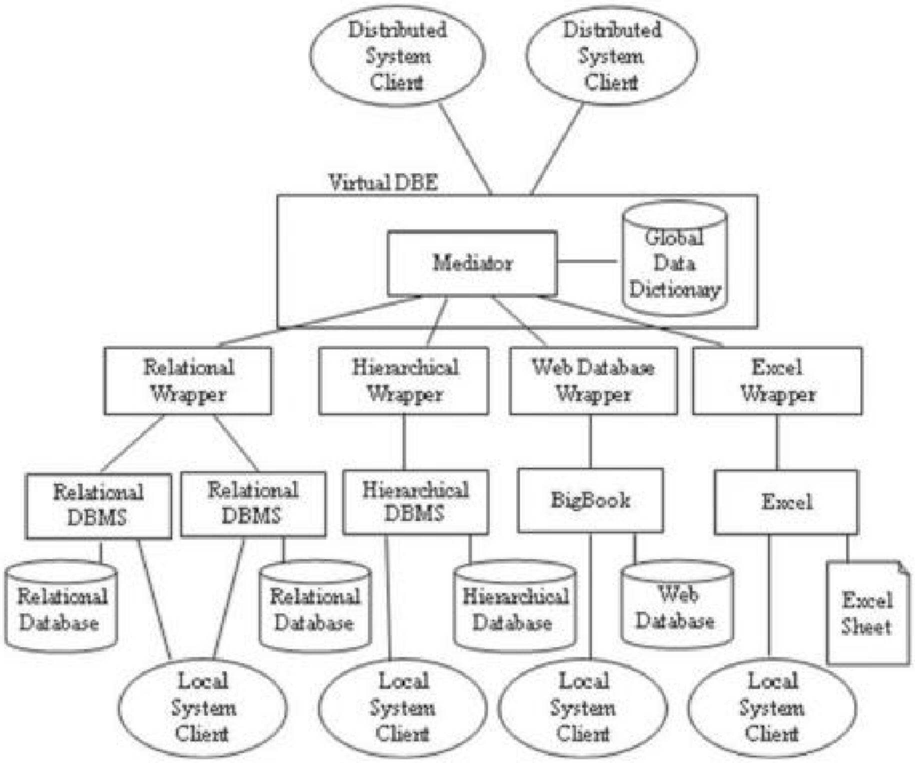
Heterogeneous Database Systems

Heterogeneous distributed database systems behave differently from homogeneous distributed database systems. In a heterogeneous database system, the underlying database systems are different, maintain their autonomy, and may hide the details of how they process queries from the outside world. These systems are also called federated database systems or multidatabase systems. The individual DBEs that make up a heterogeneous database system can support different database models, for example, relational, hierarchical, network, object-oriented, or text databases such as BigBook. DBEs in a heterogeneous database system may also utilize different query processing and optimization approaches. They can be closed or open to the outside world. If open, they provide an application programming interface (API). These differences make processing a distributed query in heterogeneous database systems a lot more difficult than processing queries in a homogeneous distributed database system.

Heterogeneous Database Systems Architecture

The underlying DBEs of a heterogeneous system are different in nature and may provide different interfaces to the outside world. One of the first challenges in integrating heterogeneous DBEs is to hide the difference in the interfaces these systems expose. A wrapper is a software module that uses the open (or the proprietary) interface of an underlying DBE and provides a uniform interface to the outside world based on the capabilities that the DBE provides. Since the de facto standard for query processing in any heterogeneous database system is SQL, a wrapper exposes a relational model and SQL as the interface for the system is wraps.

Depending on the capabilities of the underlying component DBEs, wrappers provide different sets of functionalities. For instance, a relational DBMS wrapper can support all of the capabilities that a relational DBMS provides. An Excel wrapper can provide the ability to enumerate the rows in an Excel worksheet as tuples in a relation. This ability can be utilized to perform a select on the contents of a worksheet very much like performing a select operation on a relation. As a result, we can use an Excel wrapper to join the rows in an Excel worksheet with the rows of a table exposed by a relational DBMS wrapper. Because of limitations of the BigBook database, a wrapper for the BigBook database can only provide capability of selecting limited information such as business category and the city where the business is located. The BigBook wrapper is not able to provide the capability to enumerate rows and, therefore, cannot support joins.



Wrapper-based heterogeneous database system architecture.

Each data source in the system is wrapped by a specific wrapper. Depending on the underlying DBE, a wrapper may be able to provide either tuple level or block level (a set of tuples that are grouped together) access to the information that the database controls. The wrapper may also be able to cache information outside the database for faster access. According to Wiederhold, wrappers are not directly used by the client of the heterogeneous database system, but interact with a layer of software called the mediator. The mediator does not have the capability to interface directly to the underlying DBE. The mediator can access the global data dictionary to find out the schema of the local DBEs and the functionality they provide. The mediator processes the queries that are posted by the global users; determines the location details for each piece of required data for the queries being processed by looking up the details in the GDD; exploits the functionalities that each local DBE provides through its wrapper; and optimizes the queries at the global level.

KAHE

Optimization in Heterogeneous Databases

Query optimization in a heterogeneous DDBE is a much more difficult task than query optimization in a homogeneous DDBE. The reason is that in a homogeneous distributed database system all the DBEs provide the same and uniform capabilities. This provides the global optimizer with the ability to choose any of the DBEs to perform, for example, a join. In a heterogeneous distributed database system, the mediator does not have the same freedom. In a heterogeneous database system, a local DBE may be able to perform a join and the other may not. Lack of join capability by some of the DBEs (or their wrappers) creates a challenge for the mediator that does not exist in a homogeneous distributed database system.

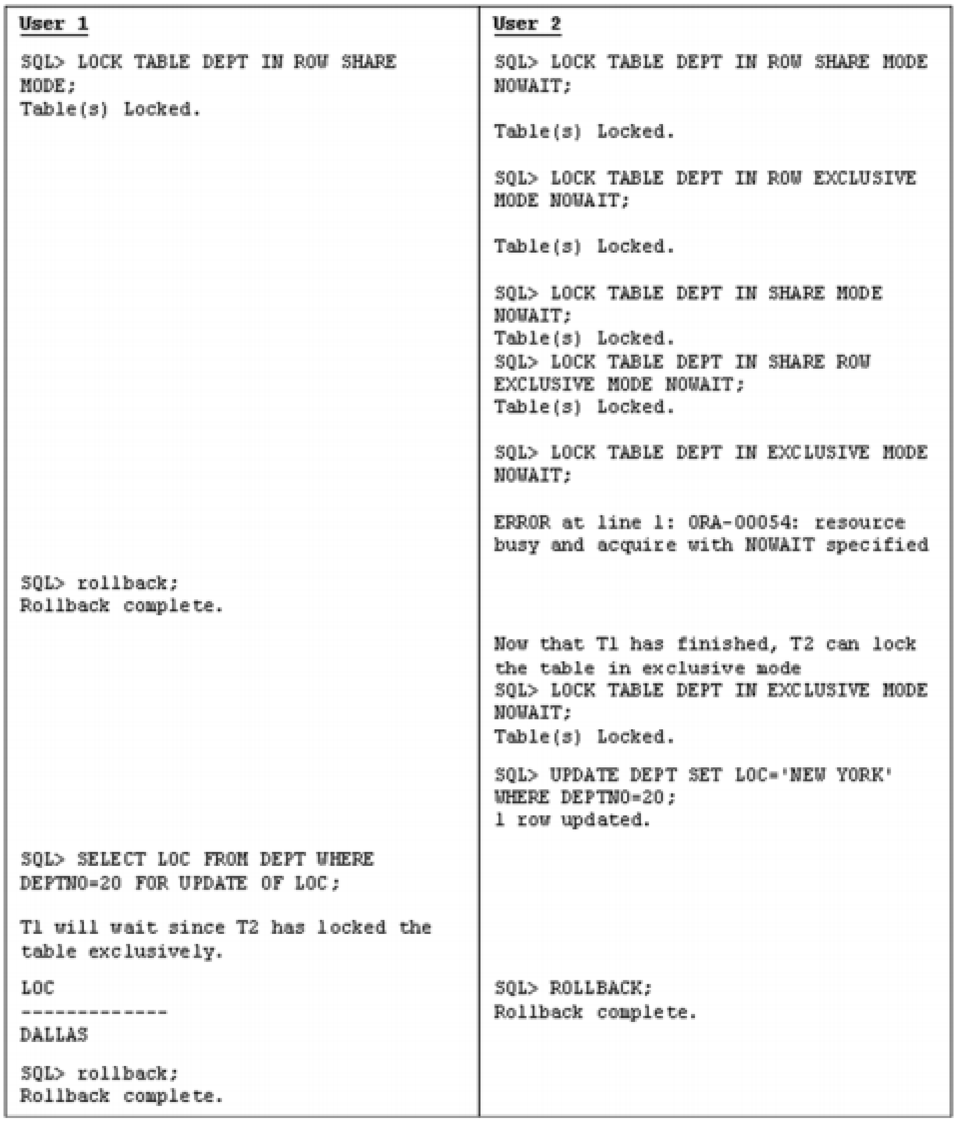
For example, in a homogeneous distributed database system, the global optimizer that needs to join relations ―R‖ and ―S‖ can simply send R from where it resides to the DBE where S is and ask the DBE server there to process the join. In a heterogeneous database system, this may not be easily achieved. Suppose that the wrapper for the DBE that stores S does not have a join capability (such as a BigBook database) and can only select tuples based on a given predicate. In this case, the mediator has to first retrieve all tuples for R from the wrapper that interfaces to the DBE where R is stored. Once all the tuples of the R relation are received, the mediator iterates through all tuples in R and uses the value of the join attribute of R, one-by-one, to find corresponding tuples in S. The lack of a join capability within the S wrapper, in effect, forces a nested-loop join to be carried out by the mediator. This approach is probably a lot more expensive than having the join be performed by the component DBE, as is the case for a homogeneous distributed database system.

Another set of differences that may exist between the DBEs in a heterogeneous database system are called semantic heterogeneity. For example, Wrapper 1 may interface to a DBE that uses letter grades {A, B, C, D, F} while Wrapper 2 interfaces to a DBE that uses scores between 1 and 10. When integrating the results from these two local DBEs or when transferring the grade from one DBE to the other, the mediator has to convert scores of 1 to 10 to letter grades and/or letter grades to scores in the range of 1 to 10. Once the capability differences between the local DBEs have been dealt with, the approach to global optimization for both homogeneous and heterogeneous database systems is very similar.

Once the capabilities of the underlying DBEs are encapsulated by the wrapper and are presented to the mediator as a set of enumeration rules, dynamic programming or iterative dynamic programming can be used by the mediator to optimize global queries in heterogeneous database systems.

CONCURRENCY CONTROL IN DISTRIBUTED DATABASE SYSTEMS

We extend the centralized concurrency control algorithms for a distributed database system. Distributed locking, timestamping, and optimistic concurrency control algorithms are discussed. As discussed before, when extending an algorithm that is designed to work on a centralized system, we have to consider the issue of control. In a centralized system, there is only one site and that site is the center of control. This is the site where transactions enter the system and where the concurrency control module of the DBMS resides. In a distributed system, on the other hand, the control may reside at a site different from the site where a transaction enters the system. The control may be centralized, residing at only one site, or distributed—where multiple sites cooperate to control execution of transactions.

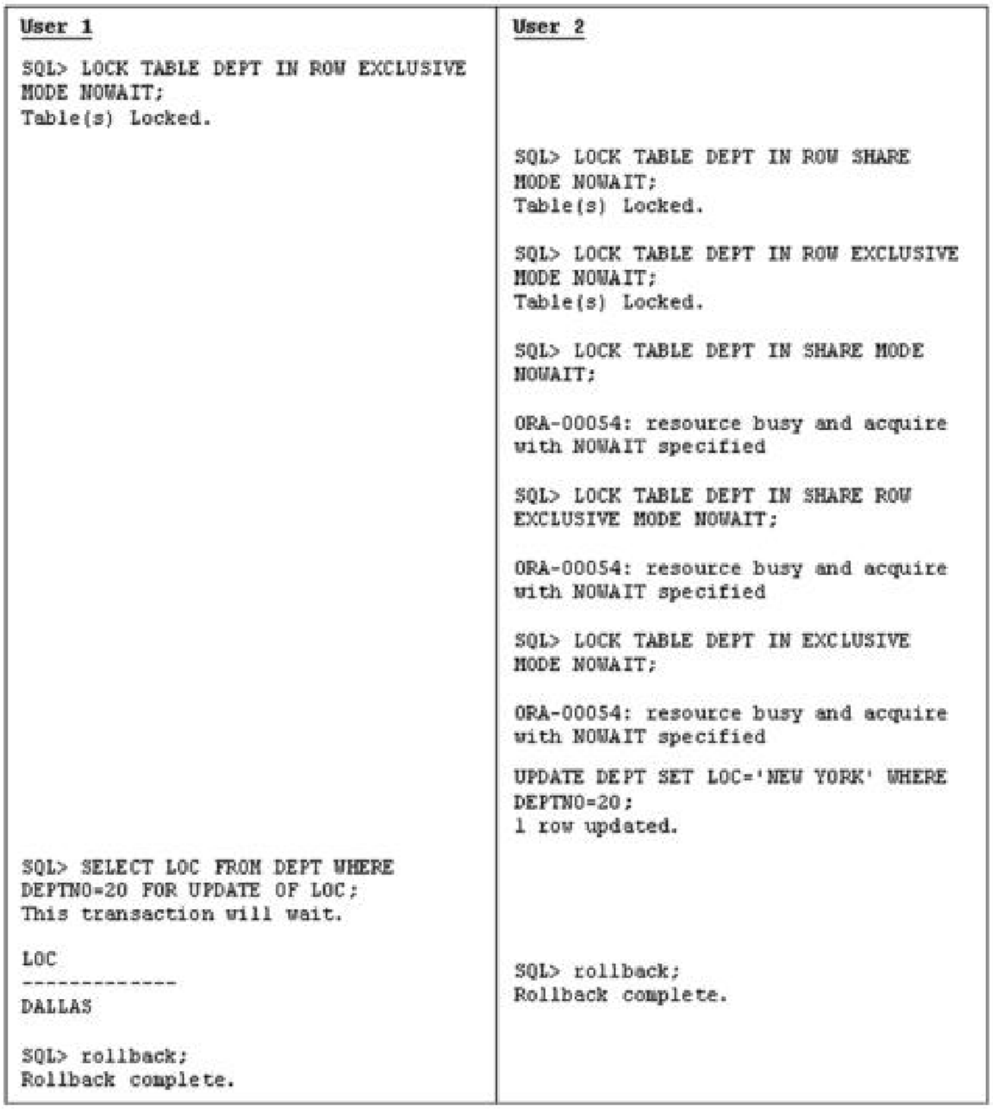


User 1	User 2

SQL> LOCK TABLE DEPT IN ROW SHARE MODE;
Table(s) Locked.
	SQL> LOCK TABLE DEPT IN ROW SHARE MODE NOWAIT;
	Table(s) Locked.
	SQL> LOCK TABLE DEPT IN ROW EXCLUSIVE MODE NOWAIT;
	Table(s) Locked.
	SQL> LOCK TABLE DEPT IN SHARE MODE NOWAIT;
	Table(s) Locked.
	SQL> LOCK TABLE DEPT IN SHARE ROW EXCLUSIVE MODE NOWAIT;
	Table(s) Locked.
	SQL> LOCK TABLE DEPT IN EXCLUSIVE MODE NOWAIT;
	ERROR at line 1: ORA-00054: resource busy and acquire with NOWAIT specified

SQL> rollback; Rollback complete.	
	Now that T1 has finished, T2 can lock the table in exclusive mode SQL> LOCK TABLE DEPT IN EXCLUSIVE MODE NOWAIT; Table(s) Locked. SQL> UPDATE DEPT SET LOC='NEW YORK' WHERE DEPTNO=20; 1 row updated.
SQL> SELECT LOC FROM DEPT WHERE DEPTNO=20 FOR UPDATE OF LOC; T1 will wait since T2 has locked the table exclusively.	
LOC ------------- DALLAS	SQL> ROLLBACK; Rollback complete.
SQL> rollback; Rollback complete.	

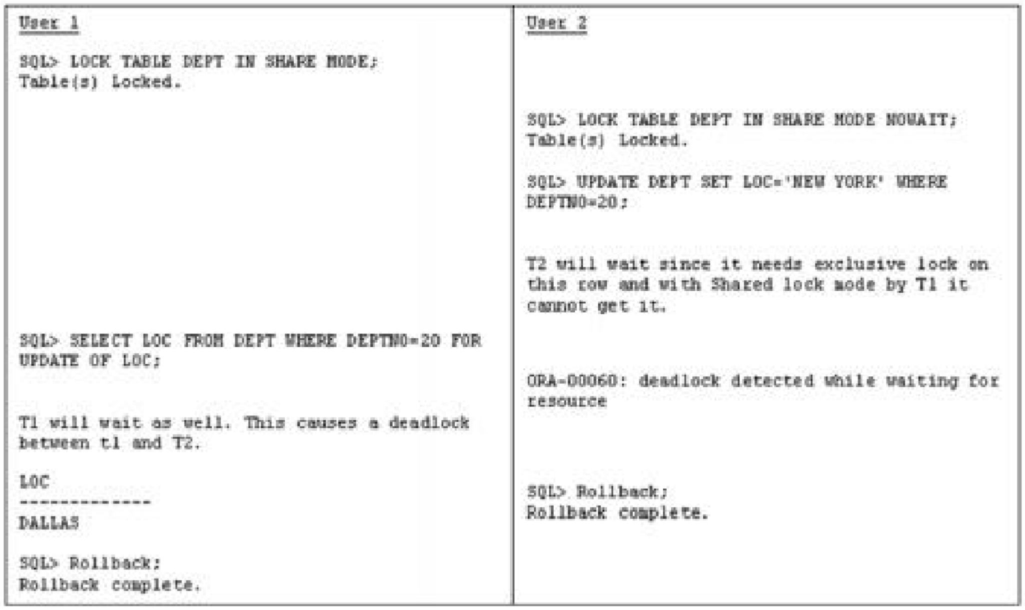
Figure 5.24 Writers using explicit locking block other writers in Oracle.



User 1	User 2
SQL> LOCK TABLE DEPT IN ROW EXCLUSIVE MODE NOWAIT; Table(s) Locked.	
	SQL> LOCK TABLE DEPT IN ROW SHARE MODE NOWAIT; Table(s) Locked.
	SQL> LOCK TABLE DEPT IN ROW EXCLUSIVE MODE NOWAIT; Table(s) Locked.
	SQL> LOCK TABLE DEPT IN SHARE MODE NOWAIT;
	ORA-00054: resource busy and acquire with NOWAIT specified
	SQL> LOCK TABLE DEPT IN SHARE ROW EXCLUSIVE MODE NOWAIT;
	ORA-00054: resource busy and acquire with NOWAIT specified
	SQL> LOCK TABLE DEPT IN EXCLUSIVE MODE NOWAIT;
	ORA-00054: resource busy and acquire with NOWAIT specified
	UPDATE DEPT SET LOC='NEW YORK' WHERE DEPTNO=20; 1 row updated.
SQL> SELECT LOC FROM DEPT WHERE DEPTNO=20 FOR UPDATE OF LOC; This transaction will wait.	
LOC ------------- DALLAS	SQL> rollback; Rollback complete.
SQL> rollback; Rollback complete.	

Writers using RX locks block other writers in Oracle.

We need to generalize the serializability requirements for a distributed DBE. Serializability in a centralized DBE stated that all partial commitment orders (PCOs) imposed by the conflicts between operations in concurrent transactions have to be compatible. As such, if we have the precedence ―Ti → Tj‖ for the commitment order of Ti and Tj, as a result of one conflict, we need to have the same order requirement for all the conflicts between these two transactions. In a distributed system, each transaction may run on multiple sites. As a result, we have to address local serializability as well as global serializability issues.

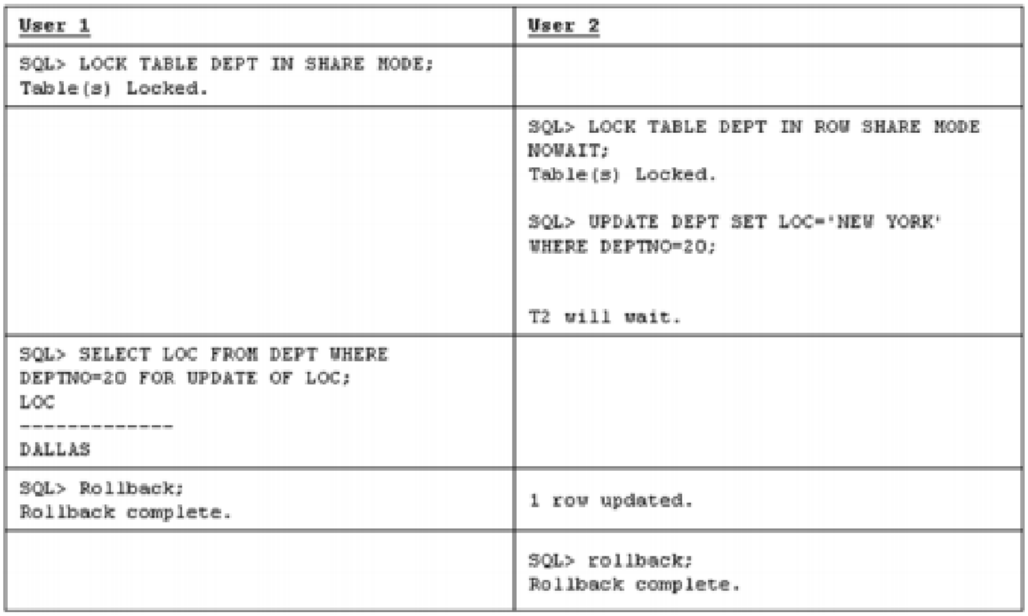


User 1	User 2
SQL> LOCK TABLE DEPT IN SHARE MODE; Table(s) Locked.	

	SQL> LOCK TABLE DEPT IN SHARE MODE NOWAIT;
	Table(s) Locked.
	SQL> UPDATE DEPT SET LOC='NEW YORK' WHERE DEPTNO=20;
	T2 will wait since it needs exclusive lock on this row and with Shared lock mode by T1 it cannot get it.

SQL> SELECT LOC FROM DEPT WHERE DEPTNO=20 FOR UPDATE OF LOC;	ORA-00060: deadlock detected while waiting for resource
T1 will wait as well. This causes a deadlock between t1 and T2. LOC ------------- DALLAS	SQL> Rollback; Rollback complete.
SQL> Rollback; Rollback complete.	

Deadlocks in Oracle.



User 1	User 2
SQL> LOCK TABLE DEPT IN SHARE MODE; Table(s) Locked.	

	SQL> LOCK TABLE DEPT IN ROW SHARE MODE NOWAIT;
	Table(s) Locked.
	SQL> UPDATE DEPT SET LOC='NEW YORK' WHERE DEPTNO=20;
	T2 will wait.

SQL> SELECT LOC FROM DEPT WHERE DEPTNO=20 FOR UPDATE OF LOC; LOC ------------- DALLAS	
SQL> Rollback; Rollback complete.	1 row updated.
	SQL> rollback; Rollback complete.

Row share table lock with the intent to exclusive access.

Two-Phase Locking in Distributed Systems

Implementing 2PL for a centralized system is straightforward. Because a centralized system contains only one site, the lock manager sees all transactions and can control lock acquisition requests from all transaction monitors (TMs). In a distributed system, that is not the case. A transaction may enter the system at Site 1, but request a lock from Site 5. At the same time, a conflicting transaction may enter the system at Site 2 and request a lock from Site 4. If there isn‘t any coordination between the lock managers at Sites 4 and 5, the conflict will not be detected, which may result in an inconsistent database.

To deal with distribution issues, a common requirement for all alternatives for implementing 2PL in a distributed system is that the lock conflicts must be seen by at least one site. In the centralized implementation, the central lock manager sees all lock requests from all sites and enforces 2PL principles. In the distributed implementation, the conflict must be detected by at least one site but may be detected by more than one site. In a distributed system, when the control of 2PL is given to one site, the algorithm is called centralized 2PL. Alsberg and Day first proposed the centralized 2PL in 1976. When the control is shared by multiple sites, the algorithm is called primary copy 2PL and, finally, when the lock responsibility is given to all sites, the algorithm is called distributed 2PL.

Centralized 2PL

Centralized 2PL is an implementation of the two-phase locking approach in a distributed system where one site is designated as the central lock manager (CLM). All the sites in the environment know where the CLM is. Each site is directed to obtain locks, according to 2PL rules, from the CLM. To request a lock, a TM sends a lock request to the CLM instead of its local lock manager. If the data item to be locked is available, the CLM locks the item on behalf of the transaction and sends a ―lock granted‖ message to the requesting TM, thereby granting the lock. If the data item cannot be locked, because it is locked by one or more conflicting transactions, the lock is not granted and the transaction is added to the list of transactions that are waiting for this item.

Upon receiving a ―lock granted‖ message, the requesting TM can continue to run. Note that the data item that is locked on behalf of the transaction may be local to the site where the transaction is or it may be located elsewhere. It is the responsibility of the TM to access the data item wherever it is. If the data item is replicated, then the requesting TM will decide which copy to read. In this case, the requesting TM also makes sure that a write operation is applied to all copies. Once a transaction is done, a TM notifies the CLM that a data item lock is no longer needed by sending a ―lock release‖ message to the CLM. Upon receipt of a ―lock release‖ message, the CLM releases the data item and grants the lock to the transaction in front of the wait queue for the item. If this ―lock release‖ is the first one from a particular TM, the CLM also makes a note that the corresponding TM has entered the second phase of 2PL. This is done to ensure that the TM does not misbehave—that it does not ask for more locks after this point.

Primary Copy 2PL

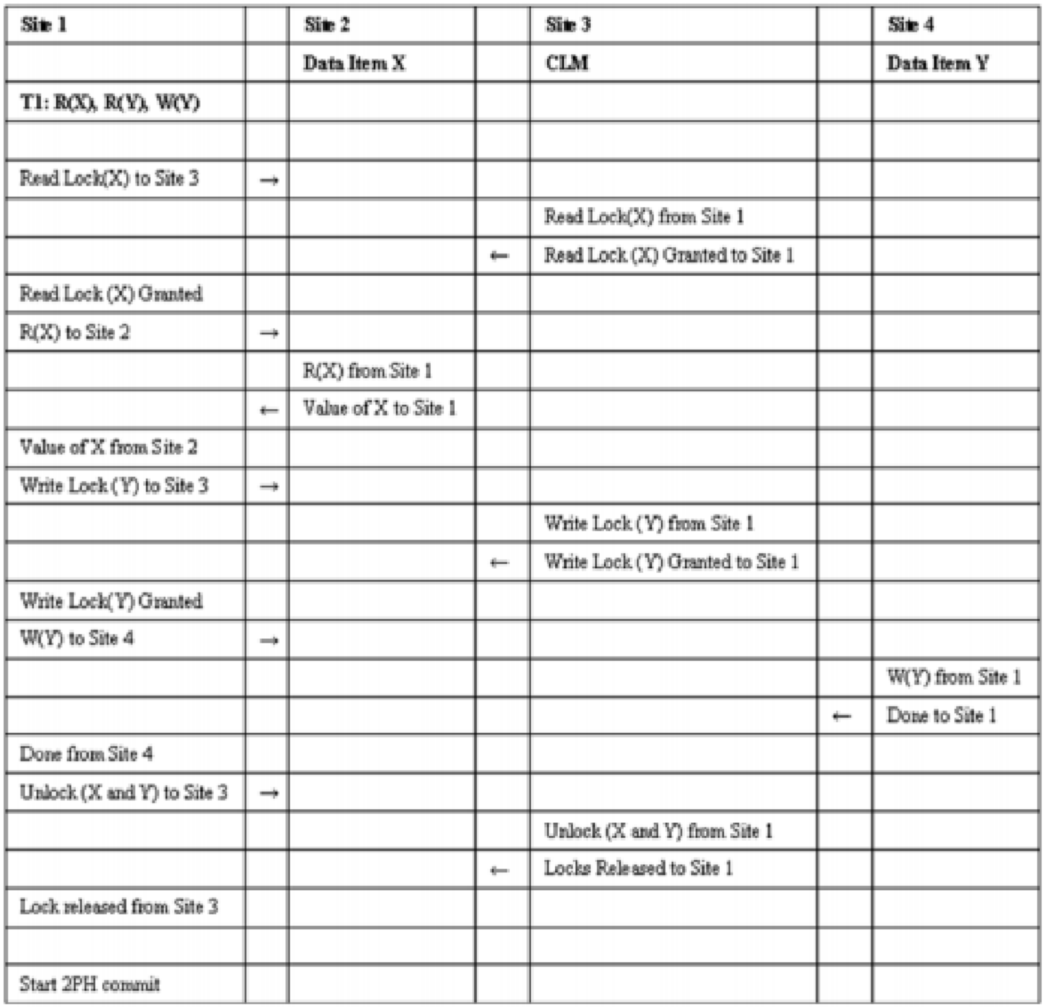
This alternative for implementing 2PL was first proposed by Stonebreaker. In this alternative, a number of sites are designated as control centers. Each one of these sites has the responsibility of managing a number of locks. For example, one site may have the responsibility of locking rows in the EMP table, while another site manages the locks for rows in the DEPT table. Obviously, how EMP and DEPT are distributed and/or replicated is known by all sites—each site knows which site controls EMP and which site controls DEPT. As such, when a TM needs to lock EMP, it directs its lock requests to the primary site responsible for locking EMP and does the same for DEPT when it needs to lock it. Beyond this difference in the distribution of control, the primary copy 2PL is basically the same as centralized 2PL implementation. Similar to the centralized 2PL implementation, if there are multiple copies of the EMP and DEPT tables, the requesting TM decides which copy to read and makes sure that all copies are written to in the case of any write operations.

Distributed 2PL

Variations of this approach have been implemented in IBM‘s system R* by Mohan and Tandem‘s Nonstop SQL engine. Distributed 2PL requires each lock manager (LM) to manage the data items stored at its local site. Where the lock manager resides depends on the data distribution and/or replication. There are three alternatives:

KAHE

• At one end of the spectrum, the distributed database maintains no replicated data items. In this case the site where the only copy of the data item resides acts as the lock manager for the item.



Site 1		Site 2		Site 3		Site 4
		Data Item X		CLM		Data Item Y
T1: R(X), R(Y), W(Y)						
Read Lock(X) to Site 3	→					
				Read Lock(X) from Site 1		
			←	Read Lock (X) Granted to Site 1		
Read Lock (X) Granted						
R(X) to Site 2	→					
		R(X) from Site 1				
	←	Value of X to Site 1				
Value of X from Site 2						
Write Lock (Y) to Site 3	→					
				Write Lock (Y) from Site 1		
			←	Write Lock (Y) Granted to Site 1		
Write Lock(Y) Granted						
W(Y) to Site 4	→					
						W(Y) from Site 1
					←	Done to Site 1
Done from Site 4						
Unlock (X and Y) to Site 3	→					
				Unlock (X and Y) from Site 1		
			←	Locks Released to Site 1		
Lock released from Site 3						
Start 2PH commit						

Example of centralized 2PL implementation.

• At the other end of the spectrum, all data items are replicated. In this case, one of the sites is chosen as the lock manager for each data item. If the responsibility of managing the locks for all items is given to one site, the approach reduces to the centralized 2PL implementation.

• Finally, some of the data items in the system may be replicated while the others are not. For the data items that are not replicated, the site where the data item resides acts as the lock manager for that item. For the replicated data items, one site is chosen as the lock manager for each item. It is the responsibility of the TM to acquire the lock from each data item‘s lock manager. The TM is also responsible for ensuring any updates to a data item are reflected in all the sites where that data item resides.

Distributed Timestamp Concurrency Control

The basic concept of timestamp concurrency control algorithms in centralized systems can easily be extended to a distributed database. Since both the basic and conservative TO algorithms rely on timestamps for resolving conflicts, it is important that timestamps be a good representative of the relative age of each transaction. In a centralized system, the physical clock reading is used as the timestamp. The physical clock obviously produces a true indication of each transaction‘s age—the smaller the timestamp the older the transaction is. As mentioned earlier, we do not actually have to use the physical clock reading of the local system as the timestamp. We can use a logical clock (LC) that is incremented for each event in the system as the transaction timestamp, since it provides for the same relative ordering for a transaction‘s age.

In a distributed system, on the other hand, we cannot use any local physical clock readings or any site‘s logical clock readings as our global timestamps, since they are not globally unique. An indication of the site ID (which is globally unique) needs to be included in the timestamp of a transaction. The issue with using a site‘s physical clocks is called ―drifting‖—when two or more clocks show numbers that are different from each other. To see the impact of clock drifting on the TO algorithms, consider two sites, S1 and S2, that generate the same number of transactions during a period of time. If the clocks at these two sites are relatively close, then the ages of transactions at these sites are also relatively close. But, if the clock at S1 is much faster than the clock at S2, after a period of time, transactions generated at S1 are going to be much younger than the transactions generated at S2. This will cause S1‘s transactions to be given a lower priority when they conflict with S2‘s transactions. To solve this problem, we can either use a system clock with which all sites synchronize their local clocks or we can periodically synchronize all the sites‘ clocks with each other. Using either approach will require

KAHE

a large number of messages. Because of this problem, we can use the logical clock readings at each site as part of the global timestamps.

When using logical clock (LC) readings drifting can still occur. In this case, sites that generate more transactions have larger LC readings. This again will negatively impact the site‘s transactions when conflicts occur. Let‘s assume site S1 generates a lot more transactions than site S2 does within a given period of time. If at time t1 the LCs at sites S1 and S2 are both set to 5, it is possible that, at time t2, S1‘s clock might be 50 while S2‘s clock might only be 20. From this point on, transactions at S1 will get lower priority than the ones at S2. To resolve this issue, we need to synchronize the logical clocks at S1 and S2. Note that it is only important to achieve clock synchronization if transactions at S1 and S2 conflict. If they do, the TMs at these sites will have to communicate with each other. We can use the messages that are sent from one site to the other for synchronization. When one site communicates with another site, it can piggyback its LC reading on the message. The receiving site then examines the LC reading received and compares it to its own. If its clock is slower than the clock at the other site, then it will advance its clock accordingly. Sites do not decrement their LCs since this would cause duplicate clock readings. This idea can be implemented with a simple comparison:

LC at local site = Max(Local LC, LC received with the message)

In the above example, when S2 sends a message to S1, no adjustment is made. On the other hand, when S1 sends a message to S2, site S2‘s clock will be advanced to 50. Implementing the basic TO algorithm in a distributed database is not cost effective due to the substantial number of messages required to enforce the read, pre-write, and commit phases of the algorithm.

Implementation of the conservative TO algorithm proposed by Herman and Verjus in a distributed system is straightforward. The basic idea is to maintain a queue for each TM in the system at each site. For example, for a system with five sites and five TMs, each site‘s scheduler will maintain five queues, one for each TM. Each TM sends the lock request to each site‘s scheduler. The receiving scheduler puts the requests in that site‘s queue, in increasing timestamp order (oldest first). It should be obvious that the queues for a given TM have the operations in

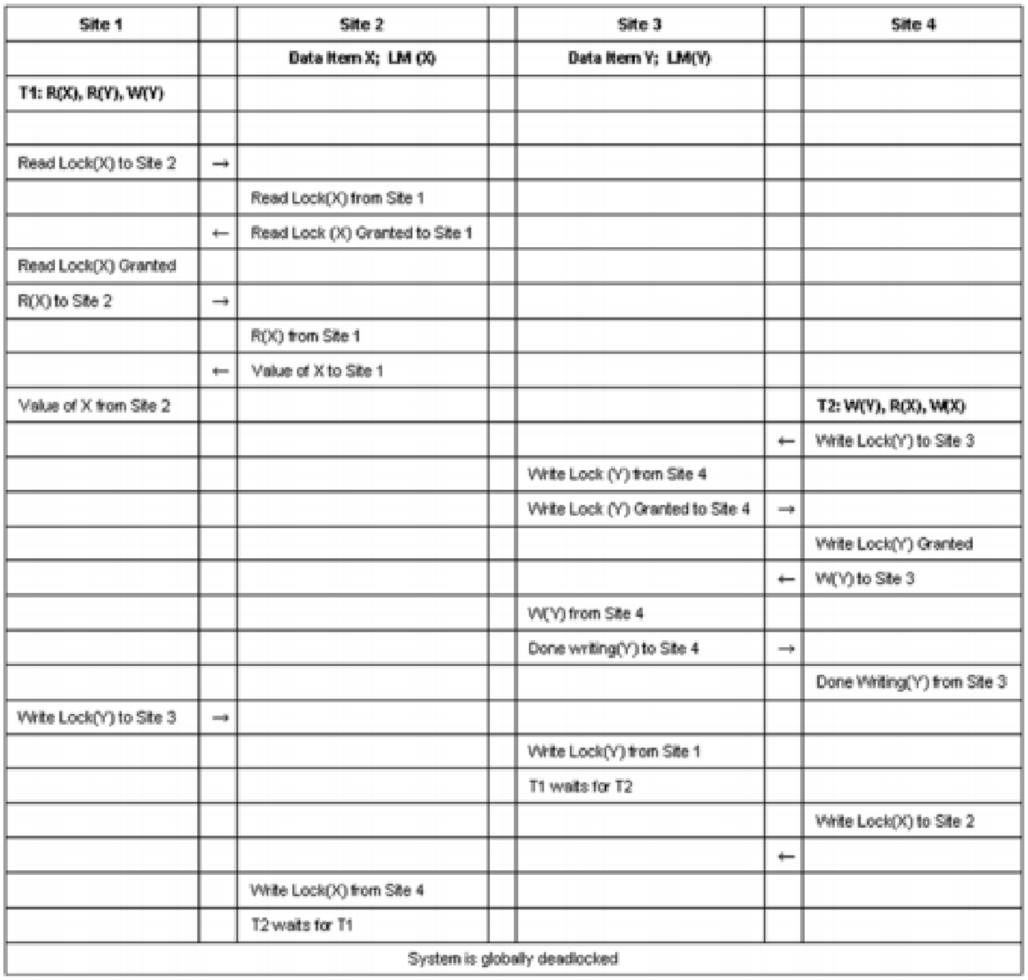
KAHE

the same order at all sites. Requests are then processed from the front of the queues in the order of their age—the oldest first. Although this approach reduces the number of restarts, it does not completely eliminate them. Suppose when processing operations from the front of queues at Site 5, the scheduler finds the queue for Site 3 empty. This indicates a lack of requests from Site 3. It is possible that Site 3 has transactions that are older than the ones in the other queues—processing the requests from younger transactions in the queues for Sites 1, 2, and 4 invalidates all such older transactions initiated at Site 3, resulting in restarts for those transactions.

One way to get around this problem is to force the schedulers to process the queues only when there is at least one request in every queue. There are two issues with this approach. First, this very conservative TO approach forces the system to execute transactions sequentially. Second, the queues for sites that do not have any transaction waiting to be processed will be empty. A site with an empty queue cannot process transactions in the other queues until a request from the corresponding site arrives. To circumvent this problem, we can require each site that does not have any transaction waiting to be processed to send a dummy request periodically in order ensure that it does not delay the processing of transactions from other sites.

Note that using timestamps to order the commitment of transactions according to their age has an advantage over locking—in TO algorithms, local sites will only allow schedules that are age-sensitive (older transactions commit before younger transactions). For example, assume transaction T4 commits before T6 at Site 5. If T4 and T6 also run at Site 7, the local scheduler at Site 7 only allows T4 to commit if it commits before T6, and not the other way around. Consequently, the second requirement we explained in Section 5.4 is automatically enforced by the local schedulers.

KAHE



Site 1		Site 2		Site 3		Site 4
		Data Item X; LM (X)		Data Item Y; LM(Y)		
T1: R(X), R(Y), W(Y)						
Read Lock(X) to Site 2	→					
		Read Lock(X) from Site 1				
	←	Read Lock (X) Granted to Site 1				
Read Lock(X) Granted						
R(X) to Site 2	→					
		R(X) from Site 1				
	←	Value of X to Site 1				
Value of X from Site 2						T2: W(Y), R(X), W(X)
					←	Write Lock(Y) to Site 3
				Write Lock (Y) from Site 4		
				Write Lock (Y) Granted to Site 4	→	
						Write Lock(Y) Granted
					←	W(Y) to Site 3
				W(Y) from Site 4		
				Done writing(Y) to Site 4	→	
						Done Writing(Y) from Site 3
Write Lock(Y) to Site 3	→					
				Write Lock(Y) from Site 1		
				T1 waits for T2		
						Write Lock(X) to Site 2
					←	
		Write Lock(X) from Site 4				
		T2 waits for T1				
System is globally deadlocked						

Example of primary 2PL implementation with a deadlock.

Conflict Graphs and Transaction Classes

SDD-1, a system for distributed databases that was developed at the Computer Corporation of America, uses the conservative distributed timestamp algorithm with some modifications [Bernstein80a]. The following provides an overview of SDD-1's concurrency control implementation. SDD- 1 recognizes different classes of transactions and levels of conflict. Each class contains one set of data items as its read set (RS) and another set of data items as its write set (WS). As we discussed before, two transactions are in conflict if one writes a data item that the other reads or writes. This concept is extended to the definition of conflicts between two transaction classes, TCi and TCj. Two transaction classes are in conflict if one of the following is true:

WS(TCi) ∩ (WS(TCj) U RS(TCj))

= Ø

WS(TCj) ∩ (WS(TCi) U RS(TCi))

= Ø

A given transaction belongs to a particular class if all of the data items in the transaction‘s read set are contained within the class‘ read set and all of the data items in the transaction‘s write set are contained within the class‘ write set. In other words, Ti belongs to transaction class TCj if and only if

RS(Ti) is a subset of RS(TCj), and

WS(Ti) is a subset of WS(TCj)

Each transaction issues its read requests for the data items contained within its read set during its read phase. Each transaction issues its write requests during the second phase, which is called the write phase. Based on the data items currently being read and written, we create a conflict graph for the classes to which the active transactions belong. A conflict graph is a non directed graph that has a set of vertical, horizontal, and diagonal edges. A vertical edge connects two nodes within a class and indicates a conflict between two transactions within the class. A horizontal edge connects two nodes across two classes and indicates a write–write conflict across different classes. A diagonal edge connects two nodes within two different classes and indicates a write– read or a read–write conflict across two classes. The following is an example that shows how a conflict graph is created. Assume the following read sets and write sets for transactions T1, T2, and T3:

T1: (RS1) = {a} and (WS1) = {b}

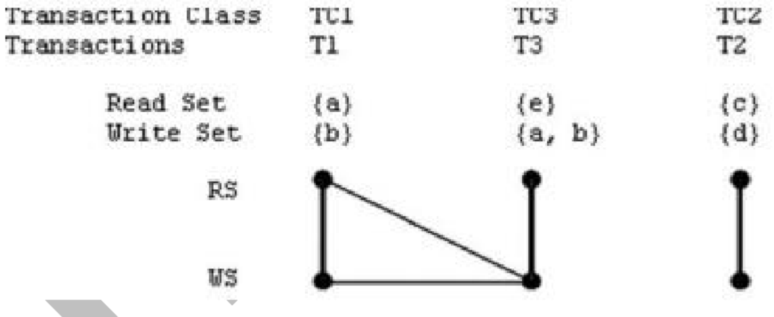
KAHE

T2: (RS2) = {c} and (WS2) = {d}

T3: (RS3) = {e} and (WS3) = {a, b}

Also, assume that transactions T1, T2, and T3 belong to transaction classes TC1, TC2, and TC3, respectively, and that Figure shows the conflict graph for these transactions.

Analysis of the conflict graph determines if two transactions within the same class or across two different classes can be run parallel to each other. In the above example, transactions T1 and T3 cannot run in parallel since their classes are connected (both diagonally and horizontally) in the conflict graph. This means that they have both a write–write conflict and a read–write conflict. On the other hand, transaction T2 can run in parallel to T1 and T3 since there are no edges between TC2 and TC1 and no edges between TC3 and TC2. By knowing to which class a transaction belongs, the scheduler at a given site knows whether the transaction can be safely processed or not. Within a class, conflicting transactions are processed in order, according to their timestamps. Across classes, transactions only conflict with each other if the classes they belong to conflict. This allows SDD-1 to statically define classes that can run in parallel and to statically define classes whose transaction-execution needs to be controlled.



Transaction Class	TC1	TC3	TC2
Transactions	T1	T3	T2
Read Set	{a}	{e}	{c}
Write Set	{b}	{a, b}	{d}

A conflict graph example for SDD-1.

In SDD-1, if a transaction needs to read or write data items at a remote site, it does not start a child process there—instead it issues a remote read or remote write request. The implementation assumes that remote requests arrive at their destination in the order of their timestamp as well. To guarantee this, a younger transaction at a given site (Site K) that wants to issue a remote read or write request to some other site (Site L) must wait until all older transactions at Site K have issued their reads and writes to Site L. This requirement further restricts concurrency among distributed transactions.

Each site maintains two queues (a read queue and a write queue) for each site in the system. Assuming that there are N sites in the system, each site will have 2 * N queues, out of which, two queues are set aside for the local operations from the local transactions. In order to handle read–write conflicts, special processing must be performed when the next read request is taken from the front of a read queue at a given site. In particular, we must examine all the write requests that are at the front of all write queues at this site. If any of these write operations are for an older transaction than the read request we are processing, we must delay the read. If all these write operations are for younger transactions, we may proceed with the read operation. Otherwise, the read is delayed. To take care of write–write conflicts, a write request at front of a write queue at a given site is processed only when all write requests at the front of all write queues at this site are from younger transactions. Otherwise, the write is delayed. To deal with a read–write conflict, similar checks are necessary when a read request in the front of a queue at a site is processed. A careful reader observes that this distributed implementation serializes the transactions based on their age and therefore reduces the amount of parallelism across transactions from different sites.

KAHE

Distributed Optimistic Concurrency Control

To extend the optimistic concurrency control algorithm to a distributed implementation, two rules must be applied:

- The first rule applies to validating transactions locally. Transaction Ti must be locally validated at all the sites where it runs. The local optimistic concurrency control algorithm is used to apply this rule. If Ti is invalid at one or more sites, then it is aborted. Local validation guarantees that Ti is part of a serializable schedule at the sites where it has been run. After Ti has been validated at all the sites where it has run, then it needs to be globally validated.

- The second rule applies to validating transactions globally. When two conflicting transactions run together at more than one site, the global validation requires that these two transactions commit in the same relative order at all the sites they run together. Suppose transactions Ti and Tj run at two sites (Site 1 and Site 2). Ti can globally validate if and only if the relative commit order of Ti and Tj is the same at both Site 1 and Site 2. Otherwise, Ti cannot validate. To apply this rule, Ti‘s commitment at a site is delayed until all conflicting transactions that precede Ti in the serialization order are committed or aborted. Note that this requirement makes the algorithm pessimistic since Ti cannot commit as soon as it is validated at a site.

One way to implement optimistic concurrency control is to require that all local schedulers maintain a total commitment order for all local transactions. These schedulers are required to send their total commitment order graphs to the responsible global TM when they validate a transaction locally. The global TM receiving all such graphs will only commit the transaction globally if the order of committing the transaction in all local graphs is compatible. Otherwise, the global TM aborts the transaction.

Federated/Multidatabase Concurrency Control

The concurrency control methods apply only to centralized databases or homogeneous distributed databases. There are other kinds of distributed database systems, however. In particular, a different class of distributed database systems known as federated or multidatabase also exists—its databases are heterogeneous. For this class of distributed database systems, implementing the concurrency control algorithms that we discussed in this chapter is not practical for various reasons. The first and foremost issue is that federated and multidatabase systems integrate both relational and nonrelational (e.g., flat file based) systems, and these systems do not always expose their transaction management functions (Transaction−Begin, Transaction−End, Commit, Abort, Lock, Unlock, etc.). As a result, implementation of the concurrency control algorithms mentioned above is not possible. Even for newer relational systems, where the distributed database software has access to the transaction management functions, implementing 2PL negatively affects the sites‘ autonomy and performance.

KAHE

POSSIBLE QUESTIONS

PART B

(EACH QUESTION CARRIES SIX MARKS)

1. Explain in detail about Query processing in distributed systems.
2. Discuss in detail about Query parsing and translation.
3. Discuss in detail about Query Translation in DDBS.
4. State the difference between centralized systems and distributed systems.
5. Expound Query parsing in detail.
6. Give an example for query optimization in sample database.
7. Explain about Query processing in centralized systems in detail.
8. Describe Concurrency Control in DDBS in detail.
9. Discuss in detail about Query Optimization.
10. Describe in detail about heterogeneous Database Systems.

PART C

(EACH QUESTION CARRIES TEN MARKS)

1. Discuss in detail about Cryptography.
2. Discuss in detail about terminologies of failure and commit protocols.
3. Discuss in detail about Query processing in centralized systems.
4. Explain i)Vertical Fragmentation ii)Horizontal Fragmentation in detail.
5. Discuss in detail about DBE taxonomy.

KAHE

KARPAGAM ACADEMY OF HIGHER EDUCATION
DEPARTMENT OF COMPUTER APPLICATIONS
DISTRIBUTED DATABASE MANAGEMENT SYSTEM

SUBJECT CODE:17CAP405D **CLASS : II MCA** **SEMESTER: IV**

Unit-3

Questions	Op1	Op2	Op3	Op4	Op5	Op6	Answer
________object in which the operations are performed	database	semantic	security	device			database
In a distributed DBMS is to make view______ efficient	localization	normalization	materialization	distribution			n
query expressed on view into a query expressed on base relation can be done by_______	query optimization	view management	relation	query modification			query modification
_______In a distributed system may be derived from fragmented relations stored at different sites	view	normal	semantic	fragmentaion			view
In a view the name and its retrieval query are stored in _____	catalog	anolog	database	partition			catalog
The view found in a distributed database catalog then merged with query on base relations such modified query is____	distributed	query	distributed query	distributed or query			distributed query
View derivation by maintaining actual versions of the views called	system	relation	optimization	snapshot			snapshot
_______ does not reflect updates to base relation	dynamic	static	database	logic			static
________ is an important function of a database system	concurrency	view	security	system			security
Data security includes how many aspects	1	2	3	4			2
________ Is required to prevent unauthorized users from understanding the physical content of data	authorization	data protection	encryption	decrypted			data protection
The information stored on disk and information exchanged on network is called______	authorization	data protection	data encryption	data decryption			data encryption
Encrypted data can be decrypted only by whom?	authorized user	end user	all	limited user			authorized user
In data security how many schemes are used ?	one scheme	4 schemes	3 schemes	two schemes			two schemes
user authentication is necessary since to be consistent if the database satisfies a set of constraints called _________	remote	reliable	programming	general			programming
semantic integrity constraints are rules that represent the _________ properities of an application	knowledge	database	static	dynamic			knowledge
_______expresses basic semantic properties inherent to a model	behavioural constraints	structural constraints	constraints	concepts			structural constraints
A data base state is said to be consistent if the database satisfies a set of constraints called ________	distributed authorization	database consis	semantic integrity	security			semantic integrity

_____regulates the application behaviour	behavioural constraints	structural constraints	constaints	concepts			behavioural constraints
Integrity constraints should be manipulated by the database administrator using ______ language	high level	lowlevel	structured	conceptual			high level
how many basic methods permits the rejection of in consistent updates	1	2	3	4			2
_______assertion to be expressed in tuple relational calculus	Integrity	individual	set oriented	privacy			Integrity
single relation and single variable assertions is called	individual	set oriented	assertion	taxonomy			individual
multirelation and multivariable constraints such as _______	primary key	foreign key	secondary	unique key			primary key
________ is a special processing because of the cost of evaluating the aggregates	assertion involving	setorinted	individual	group			assertion involving
The user from query optimization a time consuming task is best handled by__________	query processor	optmization	fragments	processor			query processor
calculus query must be decomposed into a sequence of relational operations called_____	algebraic query	query located	fragments	none			algebraic query
The data accessed by the query must be_________ so that the operations on relations are translated to bear on local data	algebraic query	query localized	localized	fragments			localized
high level query to lowlevel query is the main function of__________	calculus query	decomposed	algebraic query	query processor			query processor
A good measure of resource consumption is the ______ that will be incurred in processing the query	cost	total cost	I/o and cpu cost	cpu cost			total cost
Which directly affects their execution time, dictates some principles useful to a query processor	complexiy relational algebra	relational algebra	binary	implementation			complexiy relational algebra
To avoid the high cost of execution search how many methods are proposed?	1	2	3	4			2
popular way of reducing the cost of exhaustive search is the use of ________	heuristics	interactive improvement	simulated annealing	query processor			heuristics
optimization can be done ______ before executing the query or______ as the query is executed	statically & logically	logically & dynamically	static & dynamic	constant			static & dynamic
The main advantage over static query optimization is _____	minimizing cost	optimizing cost	hiring cost	bad choice			optimizing cost
predicated size and actual size of intermediate relations is detected by ?	dynamic qp	static qp	query optimization	constant qp			dynamic qp
The effectiveness of query optimization relies on ________	statistics	dynamics	exploitation	network			statistics
one site makes the major decisions and other sites can make local decisions are also frequent _______	centralized approach	decision	hyrbrid approaches	local information			hyrbrid approaches

The power of the client workstation can be exploited to perform database operations using ________	datamining	data shipping	optimization	query			data shipping
_______ is main function that is used to localize the data involved in the query	replicated	localization	database	semijoins			localization
______ operation has the important property of reducing the size of the operand relation	joint	inner join	semi join	outerjoin			semi join
Which two layers are involved query rewriting ?	query decompostion	data localization	global query optimization	query decomposition and data localization			query decomposition and data localization
In query decompostion global conceptual schema describing the____ relations	conceptual relation	normal	global	local			global
The calculus query is re written in _______ form	conceptual	normalized	global	centralized			normalized
The normalized query is ________ semanticaly so that incorrect queries are detected and rejected as early as possible	analyzed	restructured	simplified	views			analyzed
The calculus query is _______ as an algebraic query	structured	joined	semantic	restructured			restructured
Reconstructed by applying the fragmentation rules and then deriving a pogramme is called	localization program	materialization	normalized	globalized			localization program
query is mapped into a fragment query by substituting each distributed relation by its reconstruction is called ?	localization programmer	materialization	normalized	globalized			materialization
In a distributed query can be described with relational algebra operation and________ for transferring data between sites	communication primitives	send/receive operation	communication primitives and send/receive operation	receive operation			communication primitives and send/receive operation
query decompostion transforms a _________ into _______ query	relational calculus	relational algebra	relational calculus and algebra	normalized			relational calculus and algebra
To transform the query to normalized form to facilitate for their processing is which one of the following goal?	normalization	centralization	decompostion	analysis			normalization
The main reasons for rejection are that the query is type__________	incorrect	semantically incorrect	correct	semantically correct			semantically incorrect
The last step of query decompostion is _______	normalization	analysis	elimination of redundancy	re writing			re writing
sub trees corresponding to the localization programme is called ?	generic query	reducation query	localization	optimization			generic query
The vertical fragmentation function distributes a relation based on_________	projection attributes	reconstruction	decomposition	localization			projection attributes

The goal of hybrid fragmentation is to support,efficiently,queries involving projection, selection and ________	join	inner join	outerjoin	normal			join
Remove useless relation generated by _______ on vertical fragments	contrdicting	projections	isolate	distribute			projections
______ordering of operations for a given query is the main role of the query optimization layer	optimal	optimizer	optimal strategy	execution plan			optimal
optimal strategy also called ________	optimal ordering	query execution	optmizer	optimization			optimal ordering
________ plan consisting of the algebraic query specified on fragments and the communication operations	query execution	optimal strategy	optimizer	query			query execution

UNIT-IV

SYLLABUS

Deadlock Handling: Deadlock Definition- Deadlocks in Centralized Systems- Deadlocks in Distributed Systems- Distributed Deadlock Detection. Replication Control: Replication Control Scenarios. Failure and Commit Protocols: Terminology- Commit Protocols.

DEADLOCK HANDLING

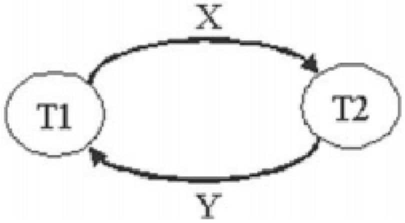
Both centralized and distributed database environments (DBEs) can utilize locks to enforce concurrency control. To access a data item, a transaction must put the right type of lock on it—a read lock to read the item or a write lock to modify the item. Suppose we have a transaction called T1. If T1 wants to lock some particular data item, but that item is already locked by some other transaction such as T2, then T1 must wait for T2 to release the lock before it can proceed. When any transaction waits for another transaction, there is always a potential for a deadlock.

DEADLOCK DEFINITION

Deadlock, or deadly embrace, is a state of the system in which two or more transactions wait forever. This is indicated by a cycle in the wait-for-graph (WFG). A WFG is a directed graph in which the circles indicate transactions and the arcs indicate waits. Sometimes, we put the name of the data item in contention (the item we are waiting for) on the arc, but often, for simplicity, the item name may not be shown. Figure depicts the classical example of a two-transaction deadlock. As seen from this WFG, transaction T1 has locked item Y and needs to lock item X, while T2 has locked item X and needs to lock item Y. In this case, since both transactions are waiting, no transaction can continue and a deadlock is formed.

There are three classical approaches to deadlock handling. These approaches can be used in either a centralized or a distributed database system. These are deadlock prevention, deadlock avoidance, and deadlock detection and removal.

KAHE



UNIT-IV

SYLLABUS

Deadlock Handling: Deadlock Definition- Deadlocks in Centralized Systems- Deadlocks in Distributed Systems- Distributed Deadlock Detection. Replication Control: Replication Control Scenarios. Failure and Commit Protocols: Terminology- Commit Protocols.

KAHE

DEADLOCKS IN CENTRALIZED SYSTEMS

All three approaches mentioned earlier can be used in a centralized system. Most current DBMS vendors, such as Oracle, Microsoft, IBM, and Sybase, utilize deadlock detection and removal in their products.

Deadlock Prevention

Deadlock prevention is an approach that prevents the system from committing to an allocation of locks that will eventually lead to a deadlock. In other words, it is impossible (both theoretically and practically) for a deadlock to occur in a system using a deadlock prevention approach. There are several possible implementations of this approach, but all of them follow the rule that ―when two or more transactions require conflicting locks for the same data item, only one of them will be given the lock.‖ One example of such implementation is the preacquisition of all locks algorithm.

The Preacquisition of All Locks Algorithm

The preacquisition of all locks is an implementation example of deadlock prevention. In this scheme, transactions are required to lock all the data items they need before they are allowed to start their work and are also required to hold onto these locks for the entire duration of the transaction. For example, suppose a transaction needs three data items. This transaction will ask the lock manager to lock all three items before it starts executing any commands inside the transaction. If any data item this transaction needs is locked by another transaction, then this transaction will have to wait. As soon as all the locks needed by the transaction are obtained, this transaction proceeds. Although this transaction may have to wait before it starts executing, the system will never be deadlocked since waiting transactions are not running, and therefore not holding any locks yet. Another way of thinking about this approach is to say that deadlock prevention deals with deadlocks ―ahead of time.‖

Observation 1: It should be obvious that in order for this approach to work, each and every transaction in the system must know all the data items they need to lock before they start executing. If any of this information is missing, the deadlock prevention approach is not going to work. Therefore, this approach is suitable for transactions that know all the data items they need a priori. This is, of course, somewhat limiting with respect to the types of transactions that can be processed. Data driven transactions cannot use this approach since, for them, what the transaction needs to lock is only known at runtime.

Observation 2: It should also be obvious that this system has no overhead for locking. At the same time, this approach may cause some transactions to wait indefinitely. This phenomenon is known as transaction starvation. Transaction starvation usually happens to large/long transactions that need to lock many data items. Because a long transaction needs to lock many data items, it is possible (and likely) that while it is waiting, some other smaller/shorter transactions (requiring fewer data item locks) will succeed in locking the items that it needs. If these other transactions keep locking items required by our long transaction, the long transaction might be forced to wait for a long time.

Deadlock Avoidance

Deadlock avoidance is a deadlock handling approach in which deadlocks are dealt with before they occur. The deadlock avoidance approach uses knowledge about the system in

KAHE

determining whether or not a wait can lead to a deadlock. If deadlock prevention deals with deadlocks ahead of time, deadlock avoidance deals with deadlocks ―just in time.‖

In this scheme, transactions can start executing and can request data items to be locked as needed. If a data item to be locked is available, then the lock manager will allocate that item to the requesting transaction. On the other hand, if the data item is locked by another transaction, then the requesting transaction has to wait. If transactions are made to wait without any control, deadlocks will undoubtedly occur. Therefore, to avoid deadlocks, we must also define a deadlock avoidance approach. When one transaction requests a lock on a data item that has already been locked by another transaction in an incompatible mode, the algorithm decides if the requesting transaction can wait or if one of the transactions needs to abort. Sometimes it is easy to determine when one transaction can safely wait for another transaction. However, it is not easy to see if allowing a transaction to wait will cause a deadlock in future.

The database deadlock avoidance approach proposed by Garcia-Molina uses a variation of the deadlock avoidance approach employed in operating systems. In this approach, data items are ordered in the system and transactions are required to access data items following this predefined order. Implementing this approach in a centralized system is straightforward. Implementing the approach in a distributed database system requires numbering the sites and then numbering the data items within each site as well. For distributed systems, the combination of site number and data item number is unique, and the ordering is defined based on this combination. In a distributed system, transactions are required to request locks on data items determined by the combined site and item number. This is achieved by transactions visiting sites in order identified by their site number, and within the site, transactions access the required data items in order. Garcia-Molina shows that such requirements guarantee that deadlocks never occur.

Two algorithms proposed by Rosenkrantz can be used to address the uncertainty about deadlock creation caused by allowing transactions to wait for each other. These algorithms are known as the wait–die and the wound–wait algorithms.

The Wait–Die Algorithm

KAHE

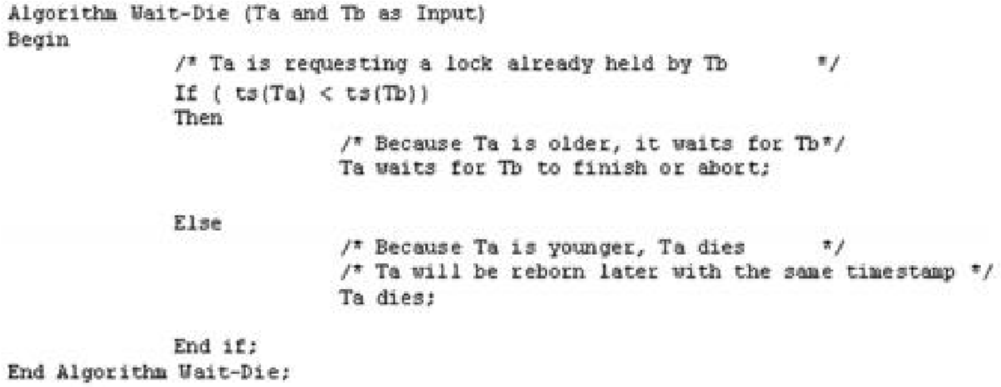
Consider two transactions, called ―Ta‖ and ―Tb.‖ Let's assume that Ta issues a lock request on data item ―X‖ and that Tb holds an exclusive lock on ―X.‖ In this case, we do not want to allow Ta to wait for Tb if this wait will eventually cause a deadlock. To make sure that deadlocks are avoided at all costs, we will utilize each transaction's age as a way of deciding whether we should allow or disallow the wait. Obviously, in order to be able to do this, we need to know the age of each transaction in the system. As a result, we need to keep track of when each transaction entered the system; in other words, we need to maintain a timestamp for every transaction in the system. A timestamp is a unique ID that carries site information as well as the local clock reading at the site where the transaction enters the system. Think of a timestamp as a sort of ―birth date‖ for each transaction. Because the timestamp contains a site component and a time component, we can guarantee uniqueness quite easily, by merely restricting timestamp generation to ―one transaction at a time‖ on each site. Since a transaction can only be ―born‖ at one site and each site can only create one transaction at a given ―local time,‖ we can be sure that the timestamp is unique across all sites in the system.

If we timestamp each transaction when it enters the system, we can use the algorithm depicted in Figure as our wait–die algorithm. Recall that a timestamp is like a birth date. This means that the smaller a timestamp is, the older the transaction is. For example, a person born in 1908 is older than a person born in 2007, and, of course, 1908 is less than 2007. Therefore, if the timestamp of Ta, which is written as ―ts(Ta),‖ is less than the timestamp of Tb, we know that Ta is older than Tb.

This algorithm simply states that when an older transaction (the transaction with the smaller timestamp) needs to lock an item that has already been locked by a younger transaction (the transaction with a larger timestamp), the older transaction waits. If, on the other hand, a younger transaction requests a lock that is held by an older transaction, then the younger transaction will be killed. Because timestamps are unique, it is impossible for two transactions to have the same age. Therefore, because we only allow the older transactions to wait, there is no way that the wait-for-graph can contain a cycle (i.e., a deadlock). The reason the algorithm kills younger transactions, as opposed to killing older transactions, is a simple priority mechanism. The belief is that because older transactions have been in the system for a longer period of time

KAHE

and have done more work, killing them is not advisable. If the younger transaction is killed, it will be ―reborn‖ later, but when this ―rebirth‖ happens, the transaction will retain its current timestamp instead of obtaining a new timestamp.



```
Algorithm Wait-Die (Ta and Tb as Input)
Begin
            /* Ta is requesting a lock already held by Tb       */
            If ( ts(Ta) < ts(Tb))
            Then
                        /* Because Ta is older, it waits for Tb*/
                        Ta waits for Tb to finish or abort;

            Else
                        /* Because Ta is younger, Ta dies       */
                        /* Ta will be reborn later with the same timestamp */
                        Ta dies;

            End if;
End Algorithm Wait-Die;
```

The wait–die algorithm.

The Wound–Wait Algorithm

Because the wait–die algorithm is very straightforward, it does not recognize some special cases. For example, there is no special handling in it to address the scenario where a younger transaction that is a candidate to be killed is very close to being done. Obviously, killing such transactions can add to the overall system overhead and can delay the average response time. Because of this shortcoming, a modified approach to the wait–die algorithm, known as the wound–wait algorithm, can be used instead. This new algorithm has the same assumptions as those we stated for the wait–die algorithm. Let’s consider two transactions, Ta and Tb, and suppose that Ta needs a data item that is locked by Tb. Using the wound–wait algorithm, if Ta is older than Tb, then Ta ―wounds‖ Tb. Notice that we did not ―kill‖ the transaction, we ―wounded‖ it. If instead, Ta were younger than Tb, then Ta would wait for Tb instead of wounding it. Figure depicts the wound–wait algorithm.

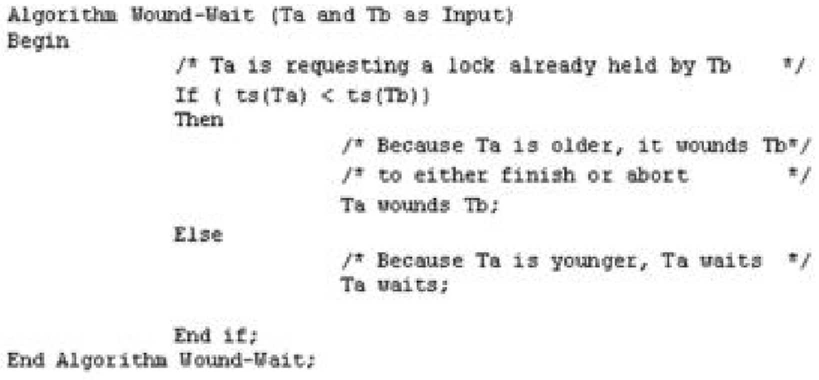
Observation 1: As before, priority is given to older transactions by allowing them to wound younger transactions and thereby access the data items that were locked by the younger transactions. The wound process deserves some explanation. Once the transaction is wounded,

KAHE

the transaction has a small amount of time to finish processing. If the wounded transaction does not finish within the specified period of time, then the transaction is killed. It is worth noticing that, during the wound period, the system may actually ―sort of‖ be in a deadlock. This deadlock, however, is not an indefinite deadlock, and therefore it is not really a deadlock—it is merely an apparent one. As soon as the wounded transaction is finished or killed, the apparent deadlock is broken.

Observation 2: The wound period will give the younger transaction that is holding the data item required by the older transaction a chance to finish. If the younger transaction does finish within this ―grace period,‖ it will not be killed. This obviously eliminates some of the drawbacks of the wait–die algorithm.

Observation 3: Both algorithms are run from the ―second‖ transaction's point of view. In other words, we are always looking from the transaction (Ta) toward the other transaction (Tb)—from the transaction requesting the lock (Ta) toward the transaction that ―got there first‖ and already obtained the lock (Tb) on the data item in conflict. Both names reflect the same processing order (and this ―second‖ transaction's point of view): ―wait–die‖ means ―if Ta is older-then-wait-else-die.‖ ―Wound–wait‖ means ―if Ta is older-then-wound-else-wait.‖ Both algorithms avoid deadlocks by ensuring the ―death‖ of the younger transaction.



```
Algorithm Wound-Wait (Ta and Tb as Input)
Begin
            /* Ta is requesting a lock already held by Tb    */
            If ( ts(Ta) < ts(Tb))
            Then
                        /* Because Ta is older, it wounds Tb*/
                        /* to either finish or abort        */
                        Ta wounds Tb;
            Else
                        /* Because Ta is younger, Ta waits  */
                        Ta waits;

            End if;
End Algorithm Wound-Wait;
```

The wound–wait algorithm.

Deadlock Detection and Removal

In this approach, the lock manager component of the concurrency control subsystem does not check for deadlocks when a transaction requests a lock. The approach is designed for fast response to lock requests. If the data item in question is not locked, the lock manager grants the requested lock to the requesting transaction. If the data item in question is already locked by some other transaction and the lock type is not compatible with the new lock request, the lock manager makes the requesting transaction wait. It is obvious that, without precautions, some of these transactions may end up in a deadlock after awhile. To handle this, the lock manager checks the WFG for cycles (deadlocks) at regular time intervals based on the system load.

Typically, a timer is used to initiate the deadlock detection process. Checking for deadlocks is initiated when the timer goes off. At that time, the lock manager looks for cycles in the WFG. It is possible that there are no cycles and therefore no deadlocks at that time, in which case there is nothing more that needs to be done. If there are any cycles in the graph, the lock manager will have to select a transaction, a victim, that needs to be rolled back to break the deadlock. Sometimes, there might be more than one cycle, meaning that there can be more than one independent (nonoverlapping) deadlock. In this case, one victim from each cycle will have to be rolled back. Sometimes, two deadlocks may overlap (have one or more transactions in common). In this case, an efficient lock manager will choose a victim that is present in both cycles to reduce the overhead of rollback and restart. Once deadlocks are dealt with, the lock manager resets the timer and continues. Victim selection can be based on many parameters.

Possible approaches to choosing the victim include the following:

- Choosing the youngest transaction as the victim
- Choosing the shortest transaction (transaction with fewest required data items) as the victim
- Choosing the transaction with the lowest restart overhead as the victim

DEADLOCKS IN DISTRIBUTED SYSTEMS

Use of timestamps in dealing with deadlocks is basically the same for both centralized and distributed DBMSs. In a distributed system, we will rely on the global timestamps and use a

KAHE

relative order based on age among all transactions in the system. As discussed, this can be achieved using a combination of site ID and local logical clock readings to guarantee global uniqueness as well as age fairness. Therefore, with respect to relative age of transactions, the global timestamps are good indicators and can be used in the wait–die and the wound–wait algorithms.

Because data is distributed in a DDBMS, the transaction processing is also distributed. This means that the same transaction might require processing at multiple sites within the DDBE. Because of this, there are two main issues that we have to deal with when implementing deadlock handling algorithms in a distributed database system.

The first issue deals with where and when a transaction is active at any given point in time. Transactions must be processed at each site that contains one or more of the data items used by the transaction. The amount of processing required at each site is not necessarily divided equally across all the sites. The start and stop times at each site are dependent on the global transaction processing model, as well as site-specific details, such as the amount of work required and the other work being done at that site. This means that processing within a single distributed transaction does not occur at exactly the same time at all the sites. In other words, we can consider the distributed transaction to be active at a given site when work is being done there and inactive when work at the site is either completed, not yet started, or placed on hold for some time. Depending on our implementation, a given transaction might be active at one or more sites at any given time, but we cannot guarantee that it is active at any particular site or at any particular time. As a result, the first issue we have to deal with for a distributed implementation is directly related to the active/inactive status at a given site for a given transaction. When a conflict happens between two transactions at a given site, it is possible that one of the transactions might not be active at that site. This is not an issue in a centralized system since there is only one DBE that contains all active transactions. We call this issue the transaction location issue.

The second issue is related to where the algorithms are actually implemented—in other words, which sites are responsible for carrying out the steps in the deadlock handling scheme.

This is not an issue in a centralized system since there is only one DBE that handles all algorithms. In a distributed system, since there are multiple local DBE servers, we have several alternatives to choose from when deciding where to implement our algorithms. We could designate one server as the center of control, we could have a floating control center that travels from server to server as needed, or we could have a number of servers share the control responsibility. We call this issue the transaction control issue.

The Transaction Location Issue

In order to better understand why there is an issue with the location of the transaction that is in conflict with another transaction, we have to understand the processing model we use in distributed systems. In our processing model, transactions initiate at a single site and get a transaction ID at that site. This ID is unique throughout the entire distributed database management system. Once a transaction has been given an ID, it can start processing. If all the data required by the transaction is locally available, this transaction does not leave its originating site and is therefore called a local transaction. If the transaction needs to access data items on other databases at other sites, then the transaction is called a global transaction or a distributed transaction.

There are two approaches for processing global transactions. In the first approach, the transaction is broadcast to all sites that contain any data items needed by the transaction and these sites can then perform the transaction’s work concurrently. In the second approach, the transaction is not broadcast to all sites but moves from site to site. The difference is that, in the first case, the transaction might be active at multiple sites at any given point in time, while in the second case, the transaction is only active at one site at any given point in time. For simplicity of our discussion, we will only consider the second approach.

In our chosen model—which is also called the daisy-chain model - a transaction carries some needed information with it when it moves from one site to another. As the transaction moves from site to site, it includes details such as the list of databases (and sites) that the transaction has already visited and the number of sites yet to be visited, items the transaction has already locked, and lock types for items locked. When the transaction terminates, either by

KAHE

committing (successful termination) or aborting (unsuccessful termination), it sends a message to all the sites it previously visited informing them of the decision to commit or abort. Figure shows an example of two transactions in a three-site system. In this figure, T1 initiates at Site 1, does some processing at Site 1, moves to Site 2 where it performs more work, and finally moves to Site 3 where it needs to complete its processing. At the same time, T2 initiates at Site 2, performs some work there, and then moves to Site 3 where it is supposed to finish its processing as well. It is plausible that the two transactions running at Site 3 cause a conflict. We will discuss in detail how to handle situations like this, once we explain the details of the distributed wait–die algorithm.

The Transaction Control Issue

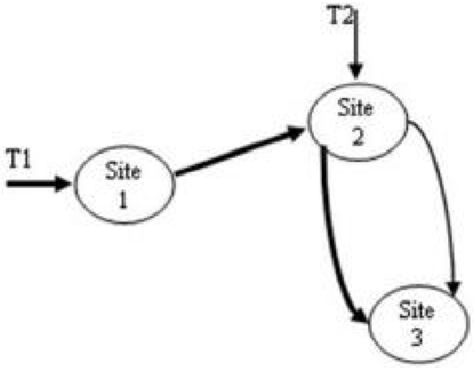
By definition, a distributed system consists of more than one server. This raises an issue that does not exist in a centralized system where there is only one server that controls everything. In a distributed system, we have to decide whether the control is given to one site or more than one site. If the control is given to only one site, we call the control approach ―centralized.‖ If multiple sites share the control, we call the control approach ―distributed.‖ Variations of these approaches are also possible. For example, in the centralized control approach, we can have one fixed site that is always the center of control or the control can be given to a site for awhile and then transferred to another site. Although different sites may be the center of control, at any given point in time there is only one controlling site in the system. In distributed control, we can have all sites be involved in controlling deadlocks or only a subset of the sites.

Distributed Deadlock Prevention

Deadlock prevention in a distributed system is implemented in exactly the same way that it is implemented in a centralized system. All resources needed by the transaction, in this case a global transaction, must be locked before the transaction starts its processing. For this alternative to distributed deadlock handling, the site where the transaction enters the system becomes the controlling site. Since the transaction does not actually start processing until all required locks are secured, deadlocks are prevented. All performance issues mentioned for centralized deadlock prevention are still present in a distributed system. In addition, since the resources that a

KAHE

transaction needs are potentially stored at different sites on different database servers, preacquisition will take a longer amount of time to complete due to communication delays. One side effect of a long wait time is an increased susceptibility to starvation for large/long transactions.



T1 and T2 will cause a conflict at Site 3

An example of two conflicting transactions.

Aside from this issue, the implementation of distributed deadlock prevention is fairly straightforward. The controlling site informs the servers where the items the transaction needs are located, to lock the desired items. The controlling site waits until it receives confirmation that all items have been locked before it starts processing the transaction. If there are any site failures or communication failures, it is possible that one or more of the required sites might not be available. In this case, the transaction cannot start until all failed sites and/or communication lines have been repaired.

Obviously, the controlling site can also fail while waiting and this can be a complex problem for this approach to handle. Consider the case where the controlling site fails after lock acquisition has begun. In this case, all sites that have already locked items must unlock those items but they will not be able to do that since they cannot communicate with the controlling site. These sites will need to wait. This is an example of a larger issue known as blocking. With respect to this issue, if a site failure or communication failure happens, then either the controlling site blocks other sites or other sites block the controlling site.

KAHE

Distributed Deadlock Avoidance

Distributed deadlock avoidance can have one of the two alternatives that we discussed for centralized deadlock avoidance. The distributed wait–die and the distributed wound–wait algorithms essentially work the same as the centralized versions of the algorithms. The major difference, as mentioned earlier, is that we need to address the transaction location issue and the transaction control issue. Because each transaction may perform its work on multiple sites, a given transaction may cause a conflict with some transaction at one site and also cause another conflict with a different transaction at a different site. It is also possible for one or more of the conflicting transactions to be inactive at the site where the conflict happens.

The Distributed Wait–Die Algorithm

The basic wait– die algorithm in a distributed system is essentially the same as it is in the centralized system. When a conflict happens between two transactions, age (based on the global timestamp) is used as a way of deciding which transaction is to be killed. The algorithm states that if a younger transaction requests a lock that is already held by an older transaction, then the younger transaction will die and be reborn later with the same timestamp it has now if the user still wants to run the transaction. On the other hand, if an older transaction needs to lock an item that a younger transaction has already locked, the older transaction will wait for the younger transaction. There are two issues with respect to implementing this algorithm in a distributed system. The Example explains the transaction location issue.

Example: Let’s assume that transaction ―Ta‖ arrives at Site K, where it needs to lock data item ―X.‖ By the time Ta gets to Site K, suppose transaction ―Tb‖ has already locked X exclusively at that site. When the conflict happens, Ta’s age is compared to Tb’s age. The action to be taken depends on which transaction is older and which one is younger. Keep in mind that even though Ta and Tb have a conflict at Site K, Tb might be active at some other site and therefore not be active at Site K.

There are two possible cases:

- Case 1: Ta Is Older than Tb. In this case, Ta has to wait for Tb. Since Ta is only active at Site K, this wait will completely block Ta’s processing. Ta will continue only when Tb has successfully committed, aborted, or decided to die. How does Ta know how long to wait? Ta will wait at Site K until the site receives a message that indicates the termination of Tb—either successful termination or unsuccessful termination. Remember, Site K is in the list of sites that Tb has visited and, therefore, the broadcast from the concurrency control at the site at which Tb terminates will include Site K.

- Case 2: Ta Is Younger than Tb. In this case, Ta will have to die. Again, since Ta is only active at Site K, the concurrency control at Site K kills Ta and sends a message to all of the sites that Ta has already visited, telling them to also kill Ta. Once Ta has successfully died at all sites, the user is notified.

The Distributed Wound–Wait Algorithm

Similar to the centralized wound– wait algorithm, when a conflict happens between two transactions in the distributed wound–wait algorithm, age is used to decide which transaction waits or is wounded. The algorithm states that if a younger transaction requests a lock on an item that is already locked by an older transaction, then the younger transaction will wait for the older transaction. On the other hand, if an older transaction needs to lock an item that a younger transaction already holds, then the older transaction has the right to wound the younger transaction to access the resource. Again, there are two issues with respect to implementing this algorithm in a distributed system.

Example: Continuing with the scenario of Example 6.1, Ta arrives at Site K, where it needs to lock X. Suppose that by the time Ta gets to Site K, Tb has already locked X exclusively. When the conflict happens, Ta’s age is compared to Tb’s age. The action to be taken depends on which transaction is older and which one is younger. Once again, keep in mind that even though Ta and Tb have a conflict at Site K, Tb might be active at some other site and, therefore, not be active at Site K.

Case 1 and Case 2 outline the steps required for implementation of this algorithm.

• Case 1: Ta Is Younger than Tb. According to the rule, Ta has to wait for Tb. Since Ta is only active at Site K, this wait will completely block Ta's processing. Ta will continue only when Tb has successfully committed or aborted.

• Case 2: Ta Is Older than Tb. According to the rule, Ta wounds Tb. In this case, what happens next depends on where Tb is active. There are three possible scenarios: (1) Tb is also active at Site K, (2) Tb has left Site K and is active at some other site, say, Site L, or (3) Tb has left Site K but is waiting (blocked) at some other site, say, Site N, because the resource it needs is locked by another transaction that is older than Tb.

Scenario 1: Tb is active at Site K. The concurrency control at Site K will enforce the wound rule. In this case, Tb is given the wound period of time to finish. If Tb does not finish within the specified period of time, Site K will roll back Tb locally and broadcast the wound-rollback message to all the sites that Tb has already visited. Each of these sites will roll back Tb locally, once they receive this message from Site K.

Scenario 2: Tb has left Site K, so it is not active there, but Tb is active at Site L. In this case, the concurrency control at Site K broadcasts the fact that Tb has been wounded to all sites. When Site L receives the wound message, that site carries out the wound rule. Again, if Tb does not finish within the time period specified by the wound process, then Site L kills Tb locally and broadcasts the wound-rollback message to all sites where Tb had previously done work. If Tb does finish within the wound period, then Site L broadcasts the committed message to all sites where Tb had previously done work.

Scenario 3: Tb has left Site K but Tb is waiting (blocked) at Site N. This scenario is the same as Scenario 2 with one minor difference. Since Tb is blocked at Site N, Site N will carry out the wound rule rather than Site L, and there is very little chance of Tb committing during the wound period.

In general, when a wound message is broadcast for a given transaction, the site at which Tb is active or is blocked has the responsibility of carrying out the wound process. According to our processing model, Tb cannot be both active at one site and blocked at another site. Thus,

KAHE

there is only one site that carries out the wound process and there is no need for further synchronization. Note also that there is a case when Tb might not be active at any site: this transaction is not blocked (not waiting for a locked resource), and this transaction is not active. This transaction is in transit from one site to another. As soon as the transaction arrives at its destination, it becomes active there. This can be handled simply by caching the message that was broadcast from Site K and applying it when the transaction arrives at the site with the newly activated transaction.

Distributed Wound–Wait Control Alternatives

From the control point of view, there are two approaches to implementing deadlock avoidance algorithms in a distributed system. These are, as mentioned earlier, centralized control and distributed control. In the centralized implementation, one site has total control, while in the distributed implementation, two or more sites share the control. Generally, centralized control is simple to implement since all decisions are made by one site. At the same time, centralized control suffers from reliability issues, since the center of control is a weak point in the system. To overcome this issue, distributed control is used to allow the system to operate even when one or more sites fail. The main issue with distributed control is how sites agree on applying the same decision. Distributed control requires additional synchronization steps that sites have to take.

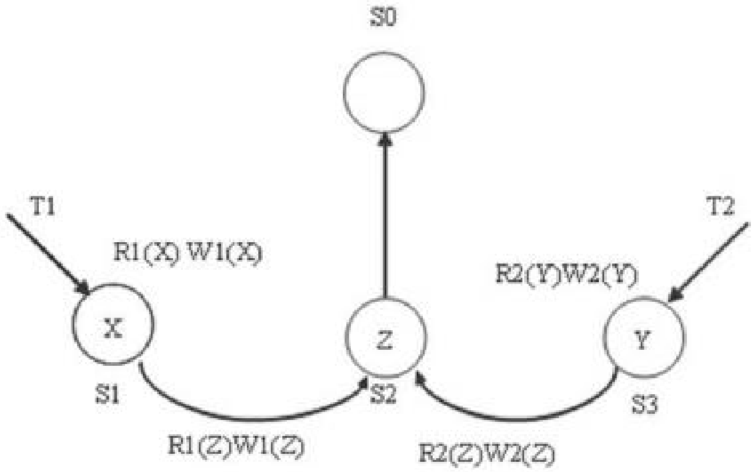
In this example, we describe the centralized control implementation for the distributed deadlock avoidance algorithm. Let's assume there are four sites in the system and they are numbered ―S0,‖ ―S1,‖ ―S2,‖ and ―S3.‖ Furthermore, data item ―X‖ is stored at S1, Y is at S3, Z is at S2, and there are no data items at S0. Let transactions ―T1‖ and ―T2‖ enter the system at roughly the same time and let T1 be older than T2. T1 is issued at S1 while T2 is issued at S3. S0 is the center of control running the wound–wait algorithm. Transaction T1 needs to perform operations ―R(X), W(X), R(Z), W(Z),‖ while T2 wants to perform operations ―R(Y), W(Y), R(Z), W(Z).‖ Obviously, T1 must perform R(X) and W(X) locally and then move to S2 to perform R(Z) and W(Z). Similarly, T2 must perform R(Y) and W(Y) locally and move to S2 to

KAHE

perform R(Z) and W(Z). Figure 6.5 displays data distribution as well as what each transaction does locally and remotely for this example.

To run their operations locally, T1 and T2 send their lock requests to S0. Since X and Y are available, S0 will grant the lock requests to allow T1 and T2 to perform their local reads and writes. Since T1 and T2 enter the system at roughly the same time, the local operations happen in parallel. After T1 and T2 have finished their local operations, they have to move to S2 to work on Z. Now, the questions we have to answer are related to the order in which these transactions arrive at S2 and how the conflict is handled by S0. There are two possible scenarios for T1 and T2 running at S2.

In the first scenario, we assume that T1 arrives at S2 first and write locks Z by sending a request to S0. Since at this point in time Z is not locked, T1's request will be granted and T1 will lock the item. Later on, when T2 arrives at S2, the item is already locked by T1. In this case, T2's request to lock the item will not be granted. The controlling site (S0) will apply the wound–wait rule. Since T2 is younger than T1, T2 will have to wait for T1 to finish or to roll back. In the second scenario, T2 gets to site S2 first and locks item Z. In this case, when T1 requests to lock Z, the controlling site cannot grant the lock to T1 immediately and will have to wound T2.



Data distribution and transaction steps for Example

It is easy to see that the outcome of applying the wound–wait algorithm depends on when transactions request the lock. What happens strongly depends on the timing of transactions and what operations they perform as part of their processing at any given site. This version of the algorithm is straightforward and has the lowest overhead for implementing the wound–wait rules. That is because all of the information necessary to decide whether or not a transaction could wait or be wounded is stored at one site (site S0 in this case). Although centralization helps with simplicity of implementing the algorithm, centralization has a major drawback, namely, having the control site as a potential bottleneck. That is why most distributed database management systems implement a distributed wound–wait deadlock avoidance approach.

In this example, we outline the details of the distributed wound–wait deadlock avoidance algorithm. We need to recall two familiar terms and introduce two new terms:

• Local Transaction. A local transaction is a transaction that does not leave the initiation site (as discussed in Section 6.3.1).

• Global Transaction. A global transaction is a transaction that moves from the initiation site to other sites (as discussed in Section 6.3.1).

• Awaits. We say that ―transaction S awaits transaction T‖ at a particular site if both S and T are local and S is waiting for T at that site, or if both S and T are global and have already visited this site.

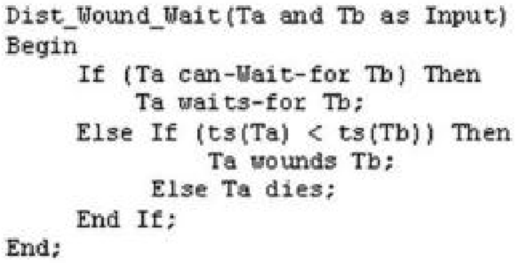
• Can-Wait-for. We say that ―transaction Q can-wait-for transaction P‖ at a particular site if there is no chain of transactions from P to Q in which each process awaits the next at that site.

With these new definitions, we can expand the wound–wait algorithm to incorporate distributed control. The algorithm is shown in the following figure:

In this algorithm, Ta and Tb are both global transactions. As a result, we have to consider not only the local waits but also the global waits (which are denoted as ―awaits‖ in this discussion). Local deadlock detectors can detect deadlocks among local transactions but cannot detect global deadlocks. To make sure that we do not have a global deadlock—a deadlock

KAHE

spanning the sites in the global wait-for-graph—we include the condition ―If (Ta can-wait-for Tb).‖ To demonstrate how this new algorithm works, we apply the distributed wound–wait deadlock avoidance to a three-site system.



```
Dist_Wound_Wait(Ta and Tb as Input)
Begin
     If (Ta can-Wait-for Tb) Then
          Ta waits-for Tb;
     Else If (ts(Ta) < ts(Tb)) Then
               Ta wounds Tb;
            Else Ta dies;
     End If;
End;
```

Distributed wound–wait algorithm.

How a distributed wound–wait algorithm works for this setup depends on the age of these three transactions and the order in which they work on the data items they need at different sites. Let Q be the oldest transaction, P the next oldest, and N the youngest. Figure 6.8 depicts one possible scenario where these three transactions enter the system at almost the same time, do some work locally, and then move to another site to perform the rest of their actions. Note that transaction actions are ordered from the top of the page to the bottom, that is, ―WLQ(b)‖ happens before ―WQ(b),‖ ―WQ(b)‖ happens before ―WLN(c),‖ and so on. Based on the ordering shown, schedule ―WLQ(b),WQ(b),WLN(c),WN(c),WLP(a),WP(a),WLQ(d),WQ(d),RLN(d),WLP(c)‖ will be formed at time 13. At this point in time, Q needs to lock item ―a‖ at Site 1 but the item has already been locked by P. The question that the distributed wound–wait algorithm has to answer is: ―If Q can-wait-for P,‖ can transaction Q wait for transaction P without the probability of a deadlock?

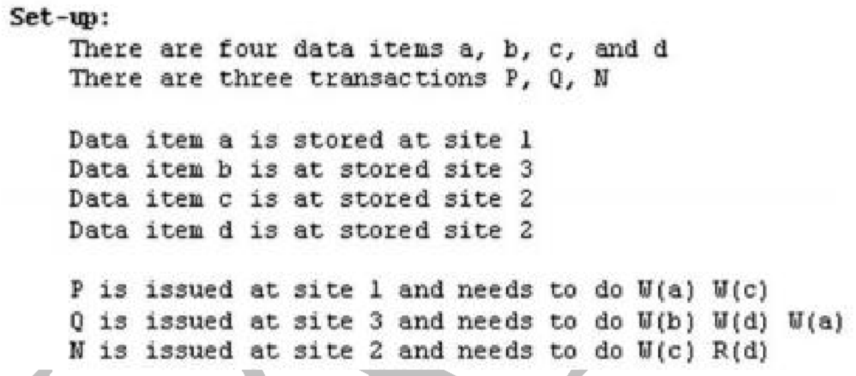
The answer to the question in this instance is, ―No, we cannot allow Q to wait for P.‖ Since P is already waiting for N, and N is already waiting for Q, making Q wait for P would cause a distributed deadlock. Since the answer is no, we have to figure out what to do with Q. That is when we apply the distributed wound–wait rule that says ―if (ts(Q) < ts(P)) then Q wounds P.‖ Because Q is older than P, Q will wound P. Since P is waiting for N at Site 2, it does not finish within the wound period, and it will be killed. In this case, Q is then free to finish its

KAHE

execution, which will then allow N to complete. At this point, P would be free to run if it were executed again.

Distributed Deadlock Detection

In this deadlock-handling approach, deadlocks are allowed to happen and the system periodically checks for them. Deadlock detection in a distributed system works exactly the same as deadlock detection in a centralized system, but it requires the creation of global wait-for-graphs (WFGs). In this approach, when a transaction requests a lock for a data item, it is not checked by the system at the time the lock request is made.

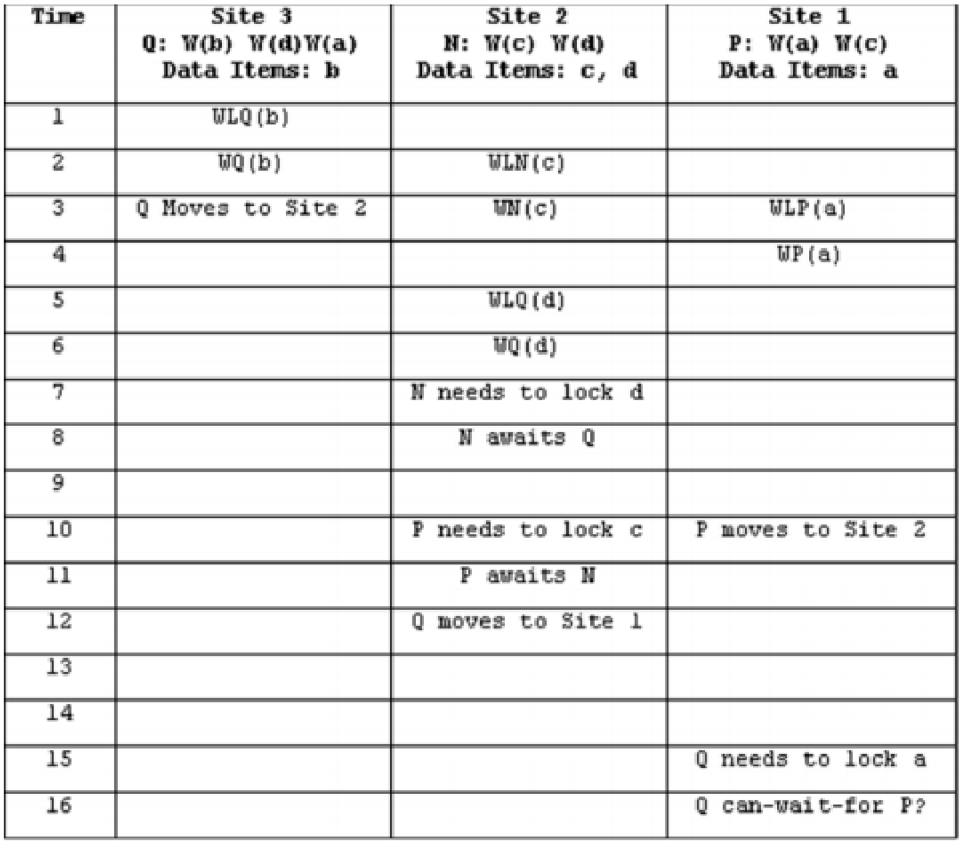


```
Set-up:
    There are four data items a, b, c, and d
    There are three transactions P, Q, N

    Data item a is stored at site 1
    Data item b is at stored site 3
    Data item c is at stored site 2
    Data item d is at stored site 2

    P is issued at site 1 and needs to do W(a) W(c)
    Q is issued at site 3 and needs to do W(b) W(d) W(a)
    N is issued at site 2 and needs to do W(c) R(d)
```

Distributed wound−wait algorithm example.



Time	Site 3 Q: W(b) W(d)W(a) Data Items: b	Site 2 N: W(c) W(d) Data Items: c, d	Site 1 P: W(a) W(c) Data Items: a
1	WLQ(b)		
2	WQ(b)	WLN(c)	
3	Q Moves to Site 2	WN(c)	WLP(a)
4			WP(a)
5		WLQ(d)	
6		WQ(d)	
7		N needs to lock d	
8		N awaits Q	
9			
10		P needs to lock c	P moves to Site 2
11		P awaits N	
12		Q moves to Site 1	
13			
14			
15			Q needs to lock a
16			Q can-wait-for P?

Example of distributed wound–wait algorithm application.

Sometimes, this will result in one or more deadlocks. To make sure that the deadlocks are eventually detected, the system maintains what is known as a global WFG that indicates which transactions are waiting for which other transactions. The existence of a directed cycle in this global WFG indicates a deadlock. It is not too complicated to detect deadlocks that are local to a site. This has been done by all centralized database management systems for many years. In a distributed database management system, however, the problem arises from having transactions wait for resources across the network. In this section, we describe how to extend the approach to local deadlock detection in a centralized system to detect deadlocks in a distributed system.

Deadlock detection algorithms use the concept of a timer. The timer is set to the prespecified amount of time in which a transaction is expected to finish. If the transaction does not finish within a predetermined amount of time and the timer goes off, we suspect that the transaction is in a deadlock and needs our intervention. The other mechanism required for deadlock handling is the concept of a deadlock detector. In a centralized system, there is one deadlock detector that is located with the local database management system. In a distributed system, we can have one or more deadlock detectors. Each deadlock detector is responsible for detecting deadlocks for the site or sites that are under its control. There are three alternatives for deadlock detection in a distributed system. They are centralized, hierarchical, and distributed deadlock detectors.

Centralized Deadlock Detectors

We can designate one site in a distributed system as the centralized deadlock detector. Periodically, each site sends its local WFG (LWFG) to this controlling site. The control site then assembles a global wait-for graph (GWFG) to determine if there are any local or distributed deadlocks. If the controlling site detects any deadlock, it must choose a victim transaction, which will be aborted to break the deadlock. Once the victim transaction is chosen, the controlling site will send information to the site where the transaction was initiated to roll back the transaction.

Although centralized deadlock detectors are easy to implement, they are not very reliable. Often, the controlling site will create a bottleneck, which means that the communication lines to the site are overwhelmed and the site may be overloaded if there are too many transactions in the system and the LWFGs contain long chains. To overcome the bottleneck issues with centralized deadlock detectors, we can implement a hierarchical or a distributed deadlock detection mechanism.

KAHE

Hierarchical Deadlock Detectors

In this approach to deadlock detection, there can be more than one deadlock detector in the system, organized into a hierarchy. Each deadlock detector in the hierarchy is responsible for collecting the wait-for-graph information from all sites that are under its control. In turn, each detector can then send the collected wait-for-graph information to a deadlock detector that is at a higher level in the hierarchy. The site at the top of the hierarchy tree (the root) is responsible for detecting deadlocks across all sites. This approach works well for a system that consists of multiple regions. Each region's deadlock detection responsibilities are assigned to a particular site in that region. Since most deadlocks occur among transactions that only run within a region, regional deadlock detectors work well. If deadlocks occur across regions, then the site that is designated as the root of the hierarchy can detect them by merging the WFGs from each regional deadlock detector involved. It should be obvious that this approach is more reliable than the centralized deadlock detection approach because there is more than one detector in the system at any given point in time.

Distributed Deadlock Detectors

In this approach to deadlock detection, all the sites participate in detecting and resolving potential deadlocks. Ron Obermarck first introduced an approach to distributed deadlock detection that shares the responsibility of detecting distributed deadlocks among all sites in the system. This approach has the highest level of reliability since the system can continue working even when one or more sites have failed. Before we talk about how distributed deadlock detection works, we should describe what makes up a distributed deadlock. In order to understand how transactions can create a distributed deadlock, we have to understand the transaction-processing model we introduced earlier in this chapter. We discussed the fact that in our model a transaction carries out its work by moving from site to site. We now extend that concept to include the concept of subtransaction. We defined a global transaction as one that leaves its initiation site after performing some work locally. To perform work at another site, a global transaction spawns a subtransaction at that site. All spawned subtransactions are called children of the global transaction. Because of the processing model we described earlier in this chapter, a global transaction can be active at only one site at any given point in time, which means that only one subtransaction can be active for a given transaction at any time.

For example, assume that T1 is a transaction initiated at Site 1 and it needs to perform some database access operations at both Site 1 and Site 2. T1 performs its intended work at Site

1 by spawning a subtransaction at Site 1. Once T1 has finished working with the local database at Site 1, it spawns another subtransaction at Site 2 to access the database at that site. At this point in time, the child of T1 at Site 2 is the only child of T1 that is active. In some respects, we can consider T1's child at Site 1 to be waiting for T1's child at Site 2. Figure depicts the WFG for two transactions (Ti and Tj) in a distributed deadlock for a two-site system.

Observation 1: As Figure 6.9 shows, the only time we allow ―waiting‖ to span sites is when the subtransactions are children of the same global transaction. In other words, a distributed (cross-site) wait must be between two children of the same transaction at two different sites. In Figure, this is denoted by the two dashed lines.

Observation 2: In Figure 6.9, we can also see that the only time we allow local ―waiting‖ is when the subtransactions are children of different global transactions. In other words, a local (within-site) wait must be between children of two different transactions at the same site. In Figure, this is denoted by the two solid lines.

Observation 3: In order for a global deadlock to happen, we must have at least two cross-site waits in the global WFG.

Observation 4: Although there are no local deadlocks, Figure clearly indicates that transactions Ti and Tj are in a distributed/global deadlock, because there is a cycle where all the arrows flow in the same direction.

The following Figure depicts an example of a three-site system in which T7, T8, and T9 are globally deadlocked.

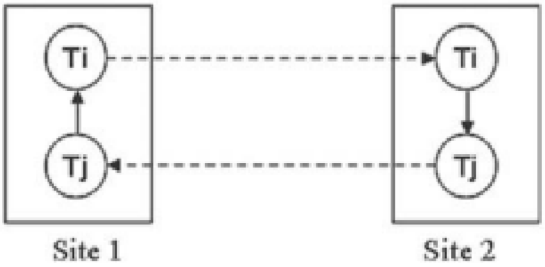
We should recognize that just because there are cross-site waits, it does not mean that there is always a global deadlock. For example, in a two-site system shown in Figure 6.11, there is no distributed deadlock even though we have cross-site waits.

If the existence of cross-site waits does not always indicate the presence of a deadlock, how do we determine if the system is, in fact, in a distributed deadlock state? To examine the issue, we consider the example WFG depicted in Figure. It is clear that locally (at Site 1) there

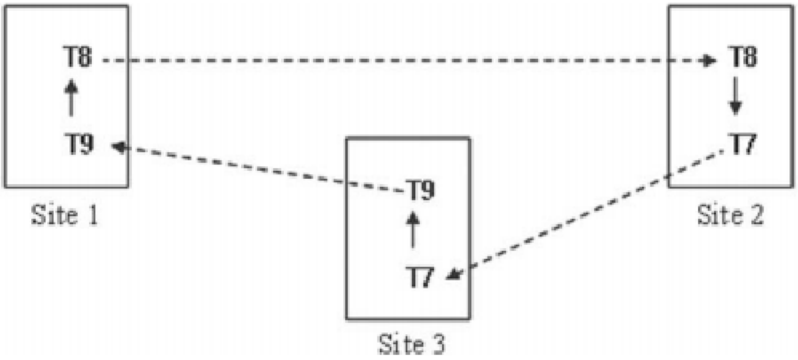
KAHE

are no local deadlocks. How do we know if the transactions on this site participate in a global deadlock?

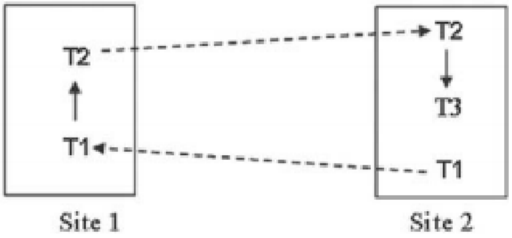
Site 1 knows that Ti's child at Site 1 is waiting for another child of Ti outside Site 1. Site 1 also knows that Tj's child at Site 1 has been spawned from outside Site 1. The rest of transactions on Site 1 are all local transactions and, therefore, Site 1 knows their status completely. Based on the information that Site 1 has, it determines that there is a ―potential‖ for a global deadlock. As far as Site 1 is concerned, if there is a chain of global waits outside Site 1 that transitively makes Ti wait for Tj, then there is a distributed deadlock. The issue here is that Site 1 does not know what is going on with transactions Ti and Tj outside of its control. The approach that realizes the potential for a global deadlock and detects such deadlocks requires the definition of two new terms and two new rules. In what follows, we explain these two terms and rules and then provide the algorithm for distributed deadlock detection.



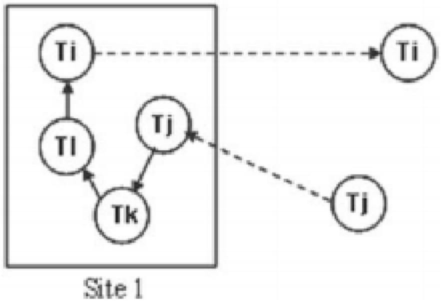
Example of two-site distributed deadlock.



Example of three-site distributed deadlock.



Example of three-site system with no distributed deadlock.



Potential for global deadlock.

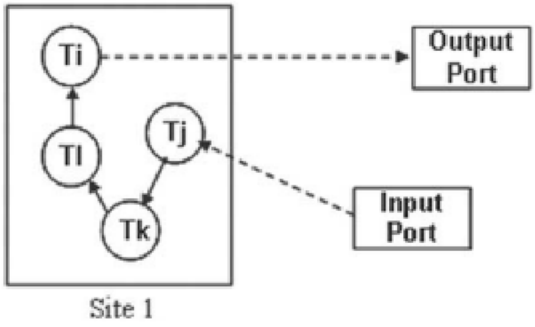
Definition 1: Input and Output Ports. Let's focus on one site. If a transaction at this site spawns a child at some other site (outside this site), we create an output port at this site. If some transaction at some other site (outside this site) spawns a subtransaction on this site, we create an input port at this site. In Figure, transaction Ti has created an output port at Site 1, while transaction Tj has created an input port at Site 1. To distinguish these ports from subtransactions we use a square instead of a circle as depicted in Figure.

As far as Site 1 is concerned, if the two ports are connected by a chain of other transactions, such that the output port is forced to wait for the input port, then there is a global deadlock as depicted in Figure.

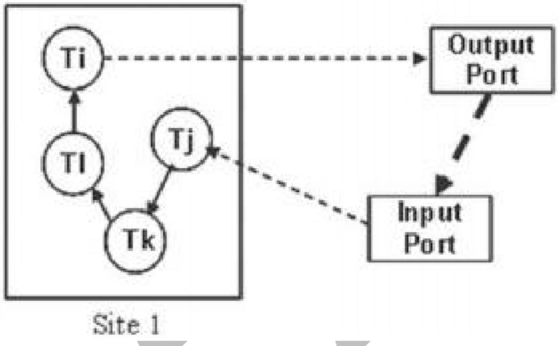
To simplify the process of identifying potential global deadlocks, we can collapse the output ports and input ports into one port called an ―external‖ port, abbreviated as ―Ex.‖

Definition 2: External Port. An external port is a port that represents the world outside a site. Figure 6.15 shows the external port (Ex) for Site 1.

With this definition of an external port, each site can identify the potential for a global deadlock locally if Rule 6.1 as discussed below applies.



Example of input and output ports for a site.



Potential for a global deadlock using input ports and output ports.

Rule 6.1:

Potential for a Global Deadlock There is a potential for a global deadlock at a site if and only if there exists a directed chain of waits that start with Ex and end with Ex at that site. In other words, there is a potential for global deadlock if a site has a directed cycle of waits including Ex. For example, in Figure, the cycle ―Ex→Tj→Tk→Tl→Ti→Ex‖ indicates a potential for a global deadlock including Tj, Tk, Tl, and Ti. Now, we can formally define the algorithm for distributed deadlock detection.

Algorithm 6.1

• All input and output ports at a site are collected into one port called external (Ex).

• A potential global deadlock is detected by the local deadlock detector when there is a cycle in the LWFG that includes Ex.

• When such a LWFG is detected, the site sends the LWFG information (including the Ex) to the other site(s).

• A site combines its local graph with LWFGs received from other sites and checks for cycles in the combined graph.

• Any cycle in the combined graph indicates a global deadlock.

• Upon detecting a global deadlock, the site selects a victim transaction to be rolled back.

• The site rolls back the selected transaction locally and broadcasts a kill message to other sites.

• Every site receiving the kill message also kills the transaction and releases the resources it is holding.

Rule 6.2:

Graph Sending Rule When a site detects a potential for global deadlock, the site sends its graph information if and only if ―the Ex is waiting for a transaction that is younger than the transaction that is waiting for Ex.‖

Applying Rule 6.2 to the above example results in the following sequence of events:

• Both sites detect a potential global deadlock.

• Site 1 detects ―Ex→T2 →T1 →Ex‖ as a potential for distributed deadlock.

• Site 2 detects ―Ex→T1 →T2 →Ex‖ as a potential for distributed deadlock.

• Site 1 sends ―Ex→T2 →T1 →Ex‖ to Site 2.

• Site 2 does not send its LWFG since in its graph T1 is older then T2.

- Site 2 receives information from Site 1, combines the two graphs, and detects the deadlock.
- Site 2 selects either T1 or T2 to be killed (but not both).
- Site 2 informs Site 1 of the decision.

REPLICATION CONTROL

Replication is a technique that only applies to distributed systems. A database is said to be replicated if the entire database or a portion of it (a table, some tables, one or more fragments, etc.) is copied and the copies are stored at different sites. The issue with having more than one copy of a database is maintaining the mutual consistency of the copies—ensuring that all copies have identical schema and data content. Assuming replicas are mutually consistent, replication improves availability since a transaction can read any of the copies. In addition, replication provides for more reliability, minimizes the chance of total data loss, and greatly improves disaster recovery. Although replication gives the system better read performance, it does affect the system negatively when database copies are modified. That is because an update operation, for example, must be applied to all of the copies to maintain the mutual consistency of the replicated items.

This example shows how the mutual consistency of two copies of a database can be lost when the copies are subjected to schedules that are not identical. In this example, the value of data item ―X‖ is 50 at both the LA and NY sites before transactions ―T1‖ and ―T2,‖ as shown below, start.

Begin_Tran T1;	Begin_Tran T2;
R(X);	(R(X);
X = X - 20;	X = X * 1.1;
W(X);	W(X);
End_Tran T1;	End_Tran T2;

KAHE

T1 runs in LA before it moves to NY to run there. T2, on the other hand, runs in NY first and them moves to LA to run there. Based on this ordering of the transactions' operations, the following two schedules are formed at LA and NY:

SLA = R1(X), W1(X), R2(X), W2(X)

SNY = R2(X), W2(X), R1(X), W1(X)

When we apply these schedules to the copies of X in LA and NY, we end up with ―X = 33‖ in LA and ―X = 35‖ in NY as shown below:

At Los Angeles

Initial Value X = 50

T1 Subtracts 20 X = 30

T2 Increases 10% X = 33

At New York

Initial Value X = 50

T2 Increases 10% X = 55

T1 Subtracts 20 X = 35

SLA is a serial schedule as ―T1 → T2‖ and SNY is also a serial schedule as ―T2 → T1.‖ Although both local schedules are serial schedules and maintain the consistency of the local copies, the system has a cycle in its global schedule, which leads to the inconsistency between the copies. To maintain the mutual consistency of the data items in the replicated database, we must enforce the rule that ―two conflicting transactions commit in the same order at every site where they both run‖

REPLICATION CONTROL SCENARIOS

KAHE

It should be clear by now that replication control algorithms must maintain the mutual consistency of the copies of the database. One way to categorize the approaches is based on whether or not the copies are identical at all times. From this perspective, there are two approaches to replication control: synchronous replication control and asynchronous replication control.

In synchronous replication, replicas are kept in sync at all times. In this approach, a transaction can access any copy of the data item with the assurance that the data item it is accessing has the same value as all its other copies. Obviously, it is physically impossible to change the values of all copies of a data item at exactly the same time. Therefore, to provide a consistent view across all copies, while a data item copy is being changed, the replication control algorithm has to hide the values of the other copies that are out of sync with it (e.g., by locking them). In other words, no transaction will ever be able to see different values for different copies of the same data item. In asynchronous replication, as opposed to synchronous replication, the replicas are not kept in sync at all times. Two or more replicas of the same data item can have different values sometimes, and any transaction can see these different values. This happens to be acceptable in some applications, such as the warehouse and point-of-sales database copies.

Synchronous Replication Control Approach

In this approach, all copies of the same data item must show the same value when a transaction accesses them. To ensure this, any transaction that makes one or more changes to any copy is expanded to make the same change(s) to all copies. The twophase commit protocol (see Chapter 8) is used to ensure the atomicity of the modified transaction across the sites that host the replicas. For example, assume we have copied the ―EMP(Eno, Ename, Sal)‖ table in all the sites in a three-site system. Also, assume transaction ―T1‖ has the following operations in it:

Begin T1:

Update EMP

Set Sal = Sal *1.05

```
Where Eno = 100;

End T1;
```

This transaction gives a 5% salary increase to employee 100. To make sure that all copies of the EMP table are updated, we would change T1 to the following:

```
Begin T1:

    Begin T11:

        Update EMP

            Set Sal = Sal *1.05

            Where Eno = 100;

    End T11;

    Begin T12:

        Update EMP

             Set Sal = Sal *1.05

            Where Eno = 100;

    End T12;

    Begin T13:

        Update EMP

        Set Sal = Sal *1.05

        Where Eno = 100;

    End T13;
```

KAHE

End T1;

where T11 is a child of T1 that runs at Site 1, T12 is a child of T1 that runs at Site 2, and T13 is a child of T1 that runs at Site 3. Because of this modification, when T1 commits all three copies will have been updated accordingly. It is important to note that, during the execution of T1, employee 100's salary is locked and therefore no other transaction can see a mutually inconsistent value for this employee's salary.

The lock is removed when T1 commits, at which time T1 reveals the new salary for employee 100 in all copies of the EMP table.

It should be obvious that enforcing synchronous replication control has major performance drawbacks. Another major issue is dealing with site failures. If one of the sites storing a replica of a data item modified by T1 goes down during the execution of T1, T1 is blocked until the failed site is repaired. For these reasons, synchronous replication is only used when one computer is a backup of the other. In this scenario, two computers act as a hot standby for each other. Suppose we have two servers, ―A‖ as the primary and ―B‖ as the backup, that need to be identical at all times. In this setup, when A fails, transaction processing continues on B without interruption. Meanwhile, transactions that run on B are kept in a pending queue for application to A when it is repaired. When the A server restarts, it is not put back in service until it is synchronized with the B server. During synchronization, new transactions that arrive at either site are held in the job queue and are not run. Synchronizing A means that the pending transactions from B are applied to A in the same order that they ran on B. After all of the transactions in the pending queue have been processed against A, both servers are put back in service. At this time, transactions are processed from the front of the job queue and run against both copies.

Asynchronous Replication Control

In this approach, copies do not have to be kept in sync at all times. One or more copies may lag behind the others (be out of date) with respect to the transactions that have run against the copies. These copies need to eventually catch up to the others. In the industry, this process is

KAHE

known as synchronization. How and when the out-of-date copies are synchronized with the others depends on the application. There are multiple approaches to implementing the required synchronization. Most commercial DBMSs support what is known as the primary copy approach. This approach is also known as the store and forward approach. The site that is updated first is called the primary site, while others are known as the secondary sites. Some DBMS vendors call the primary site the publisher and the secondary sites the subscribers. All transactions are run against the primary site first. This site determines a serialization order for the transactions it receives and applies them in that order to preserve the consistency of the primary copy. The transactions are then queued for application to the secondary sites. Secondary sites are updated with the queued transactions using a batch-mode process. This process is known as the rollout. The database designer or the DBA decides on the frequency of rolling out transactions to the secondary sites.

The secondary copies can be kept in sync with the primary copy by either rolling out the transactions that are queued or rolling out a snapshot of the primary copy. To roll out queued transactions, the transactions are run from the front of queue against all secondary sites. In the snapshot rollout, the image of the primary copy is copied to all secondary sites. The advantage of rolling out a snapshot of the primary is speed. Most databases support unloading a database to a file and reloading the database from a file. Snapshot replication can be implemented using these database capabilities to speed up the synchronization of the secondary sites with the primary. Snapshot replication can be done on demand, can be scheduled, or can simply run periodically.

In asynchronous replication, the failure of the primary is troublesome, since the primary is the only copy that is updated in real time. To deal with the failure of the primary, the system may use a hot standby as a backup for the primary. When the primary fails, the standby will continue to act as the primary until the primary is repaired and is synchronized. Instead of having a fixed site as the primary, an approach that allows for a floating primary can also be utilized. In this approach, the responsibility to be the hot standby for the primary is passed from one site to another in a round-robin fashion. As another alternative, sites may also be allowed to compete to be the primary’s hot standby, if the designers choose.

KAHE

Other alternatives to asynchronous replication control can also be implemented. In one approach, instead of having one primary and many secondary sites, a system can have multiple primaries and a single secondary. In this case, transactions are applied to each primary when they arrive at a secondary. Since there is no immediate synchronization between the primaries, the primary copies may diverge. To synchronize all the copies, transactions that are queued at each primary are sent to the single secondary for application. This copy will then generate a serialization order that is applied to the secondary and is then rolled out to all the primaries. In another approach, as opposed to having sites designated as primary and secondary, we can have all sites act as peers. In this approach to replication, we utilize symmetric replication in which all copies are treated the same. Transactions are applied to the local copy of the database as they arrive at a site. This approach potentially causes the divergence of the replicas. Since the copies of the database in this approach may diverge, occasionally the system has to synchronize all the copies. Most DBMS vendors ship the necessary software to compare the contents of the copies and also provide tools for synchronization and identification of differences in the copies. When synchronization is required, tools from the DBMS allow the database designer or a DBA to synchronize the replicas before they are put back in service again.

FAILURE AND COMMIT PROTOCOLS

Any database management system needs to be able to deal with failures. There are many types of failures that a DBMS must handle.

TERMINOLOGY

We define the terms we need here to make sure that the reader’s understanding of these terms is the same as what we have in mind.

Soft Failure

A soft failure, which is also called a system crash, is a type of failure that only causes the loss of data in nonpersistent storage. A soft failure does not cause loss of data in persistent storage or disks. Soft failures can range from the operating system misbehaving, to the DBMS bugs, transaction issues, or any other supporting software issues. We categorize these under the

KAHE

soft failure, since we assume that the medium (the disk) stays intact for these types of failure. A soft failure can also be caused by the loss of power to the computer. In this case, whatever information is stored in the volatile storage of the computer, such as main memory, buffers, or registers, is lost. Again, the assumption we make is that power loss does not cause disk or database loss. This assumption does not mean that we do not deal with disk failure issues but that we treat the disk loss under the hard failure that we discuss next. Note that soft failures can leave the data stored in persistent storage in a state that needs to be addressed; for example, if the software was in the middle of writing information to the disk, and only some data items were written before the failure, then, obviously, the data content might be inconsistent, incomplete, or possibly corrupted. However, the persistent storage did not lose any data. This is true in the sense that the persistent storage still contains everything we wrote to it—it just does not contain all the things we intended to write to it.

KAHE

Hard Failure

A hard failure is a failure that causes the loss of data on nonvolatile storage or the disk. A disk failure caused by a hard failure destroys the information stored on the disk (i.e., the database). A hard failure can be caused by power loss, media faults, IO errors, or corruption of information on the disk. In addition to these two types of failures, in a distributed system network failures can cause serious issues for a distributed DBMS. Network failures can be caused by communication link failure, network congestion, information corruption during transfer, site failures, and network partitioning. There have been many studies on the percentage, frequency, and causes of soft and hard failures in a computer system over the years. We will not repeat these finding here.

As a rule of thumb, the following can be used as the frequency of different types of failures:

- Transaction failures happen frequently—maybe as many as a few times a minute. This is usually for high-volume transaction processing environments like banking and airline reservation systems. The recovery is usually fast and is measured in a fraction of a second.

• System failures (power failure) can happen multiple times a week. The time it takes to recover is usually minutes.

• Disk failures can happen once to maybe twice a year. Recovery is usually short (a few hours), if there is a new, formatted, and ready-to-use disk on reserve. Otherwise, duration includes the time it takes to get a purchase order, buy the disk, and prepare it, which is much longer (could be a number of days to a week or two).

• Communication link failures are usually intermittent and can happen frequently. This includes the communication link going down or the link being congested. Recovery depends on the nature of the failure. For congestion, typically the system state changes over time. For link failure, the link will be bypassed by the routing protocols until after the link is repaired. Sometimes the link failure is caused by the failure of a hub or a router. In this case, the links that are serviced by the hub or the router are disconnected from the rest of the network for the duration of the time the device is being repaired or replaced.

Commit Protocols

A DBMS, whether it is centralized or distributed, needs to be able to provide for atomicity and durability of transactions even when failures are present. A DBMS uses commit protocols to deal with issues that failures raise during the execution of transactions. The main issue that the commit protocols have to deal with is the ability to guarantee the ―all or nothing‖ property of transactions. If a failure happens during the execution of a transaction, chances are that not all of the changes of the transaction have been committed to the database. Commit protocols prevent this from happening either by continuing the transaction to its completion (roll forward or redo) or by removing whatever changes it has made to the database (rollback or undo). The commit protocols guarantee that after a transaction successfully commits, all of its modifications are written to the database and made available to other transactions. Commit protocols also guarantee that all the incomplete changes made by the incomplete transactions are removed from the database by rollback when a failure happens.

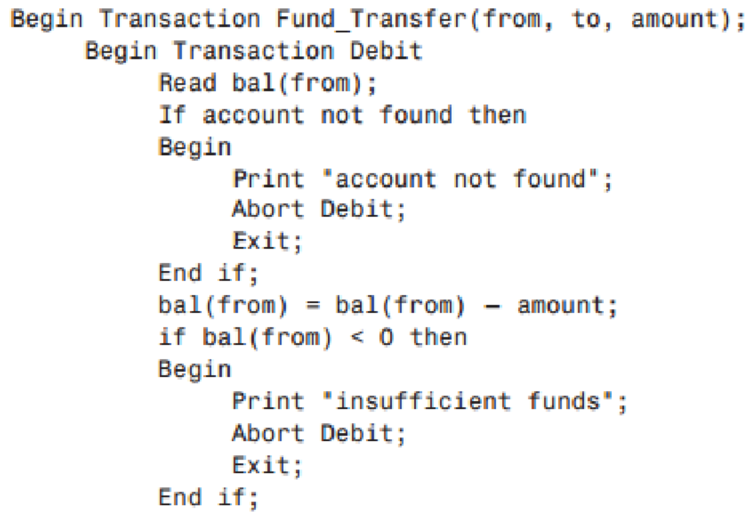
Commit Point

KAHE

A commit point is a point in time when a decision is made to either commit all the changes of a transaction or abort the transaction. The commit point of a transaction is a consistent point for the database. At this point, all other transactions can see a consistent state for the database. The commit point is also a restart point for the transaction. This means that the transaction can be safely undone. Finally, the commit point is a release point for the resources that the transaction has locked.

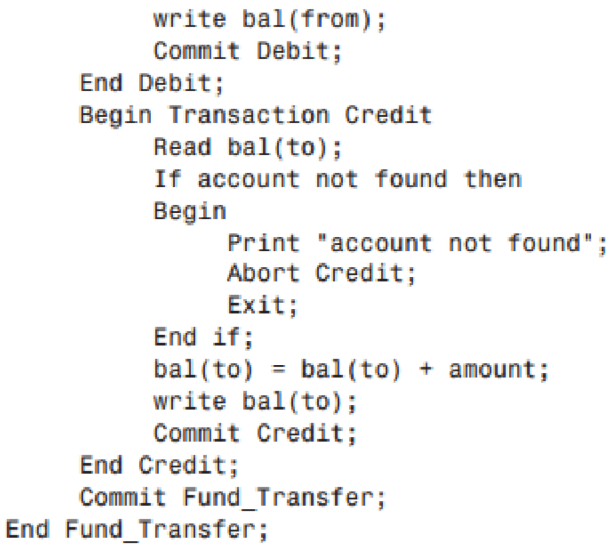
Transaction Rollback (Undo)

Transaction rollback or undo is the process of undoing the changes that a transaction has made to the database. Rollback is applied mostly as the result of a soft failure (see Section 8.1.1). Rollback is also used as a necessary part of transaction semantics. For example, in the fund transfer transaction in a banking system, there are three places from which a transaction should abort. It is necessary to abort this transaction if the first account number is wrong (it does not exist), if there is not enough money in the first account for the transfer, and finally, if the second account number is wrong (it does not exist). The following shows the fund transfer transaction that is written as a nested transaction consisting of two subtransactions—a debit and a credit transaction. This nesting of the two transactions allows the program to run on a distributed DBMS, where accounts are stored on two different servers.



```
Begin Transaction Fund_Transfer(from, to, amount);
     Begin Transaction Debit
          Read bal(from);
          If account not found then
          Begin
               Print "account not found";
               Abort Debit;
               Exit;
          End if;
          bal(from) = bal(from) – amount;
          if bal(from) < 0 then
          Begin
               Print "insufficient funds";
               Abort Debit;
               Exit;
          End if;
```

KAHE



```
                write bal(from);
                Commit Debit;
           End Debit;
           Begin Transaction Credit
                Read bal(to);
                If account not found then
                Begin
                     Print "account not found";
                     Abort Credit;
                     Exit;
                End if;
                bal(to) = bal(to) + amount;
                write bal(to);
                Commit Credit;
           End Credit;
           Commit Fund_Transfer;
      End Fund_Transfer;
```

This transaction transfers the ―amount‖ from the ―from‖ account to the ―to‖ account. The debit subtransaction checks the validity of the account from which funds are to be taken. If that account does not exist, then the transaction is aborted. If the account exists but there is not enough money to transfer—the resulting balance after the debit is less than zero—the transaction has to abort again. The credit subtransaction does not need to worry about the amount of money in the ―to‖ account since money will be added but needs to make sure that the ―to‖ account exists. If the ―to‖ account does not exist, then the transaction aborts again.

If all three abort conditions are false, then the transaction is at the commit point. At this point, as mentioned before, the user needs to make a decision to commit the transaction or roll it back. For an ATM transaction, this point in time is when the ATM machine gives the customer the option to complete the transaction or cancel it. At this point, each account's balance has been updated and everything is ready to be committed. The two database servers—local systems where the accounts are stored—have the accounts' balances still locked. Once the user decides to commit the transaction, the locks are released and the new accounts' balances are available to other transactions.

Transaction Roll Forward (Redo)

Transaction roll forward or redo is the process of reapplying the changes of a transaction to the database. Since the changes are reapplied to the database, typically a transaction redo is applied to a copy of the database created before the transaction started. Redo is mostly necessary for recovery from a hard failure.

Transaction States

A transaction goes through the following set of steps during execution:

1. Start_Transaction

2. Repeat

    2.1 Read a data item's value

    2.2 Compute new values for the data item

    2.3 If abort condition exists then abort and exit

    2.4 Write the new value for the data item

    2.5 If abort condition exists then abort and exit

3. Until no more data items to process

4. Commit

5. End_Transaction

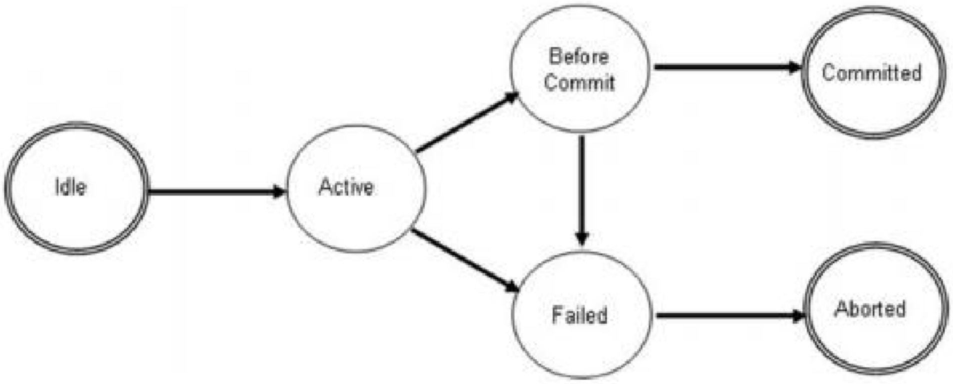
Statement 1 indicates a transaction's activation by a user or the system. Similarly, statement 5 indicates the successful termination of the transaction. The transaction can also terminate unsuccessfully, as shown in statements 2.3 and 2.5, if abort conditions exist. Some DBMSs require explicit ―Start Transaction‖ and ―End Transaction‖ while others use implicit start and end statements. For example, in Oracle a begin transaction is assumed when changes are made to the database. In this context, an implicit ―End Transaction‖ is assumed when a commit is issued. On the other hand, in SQL Server the ―Start Transaction‖ and ―End

KAHE

Transaction‖ are explicit. As seen in the code snippet above, once a transaction starts, it repeatedly reads the data items from the database, calculates new values for them, and writes new values for the data items to the database. Obviously, in this algorithm we assume the operation being performed inside the transaction is an update operation. In case of an insert into the database, no data items need to be read. Conversely, when a delete command is executed, no new values are written to the database.

One formal specification of a transaction uses a finite state automaton (FSA) that consists of a collection of states and transitions. This is typically shown as a state transition diagram (STD). In a STD, states are shown as circles and transitions as arrows, where the tail is connected to the state the program leaves and the tip is connected to the state the program enters. The state of the STD that a transaction is in at the time of a failure, tells the local recovery manager (LRM) what needs to be done for a transaction. Figure depicts the STD for a transaction.

In this diagram, the double-lined circles are terminal states and the single-lined states are transitional. The ―Idle‖ state corresponds to when the transaction is not running.



State transition diagram for a transaction.

Once started, a transaction changes state to the ―Active‖ state. In the ―Active‖ state, the transaction performs its work. In this state, the transaction reads, performs calculations, and writes data items. If any of the abort conditions hold, the transaction issues an abort and changes state to the ―Failed‖ state. Entry into the ―Failed‖ state causes the local recovery manager to undo the transaction. The transaction enters its commit point by transitioning to the ―Before Commit‖ state. As mentioned before, at the commit point a transaction holds a consistent state for the database. It can back out of the transaction by canceling the transaction or it can complete the work by committing. Cancellation is performed by a transition from the ―Before Commit‖ state to the ―Failed‖ state, which causes the LRM to undo the transaction. The successful termination is initiated by transitioning from the ―Before Commit‖ state to the ―Committed‖ state.

It is important to remember the following:

- In the ―Active‖ state, the transaction has not completed all the work necessary—not all changes have been made.
- In the ―Before Commit‖ state, the transaction has completed all the work necessary—all transaction changes have been made.
- In the ―Failed‖ state, the transaction has decided to abort or it has been forced by the DBMS to abort due to concurrency control and/or deadlock issues.
- In the ―Committed‖ state, the transaction has terminated successfully.
- In the ―Aborted‖ state, the transaction has terminated unsuccessfully.

Database Update Modes

A DBMS can use two update modes:

- Immediate update mode—a DBMS is using immediate updates, if it writes the new values to the database as soon as they are produced by a transaction.
- Deferred update mode—a DBMS is using deferred update, if it writes the new values to the database when a transaction is ready to commit.

KAHE

In the case of the immediate update, the old value of a data item being changed is overwritten in the database by the transaction. As a direct result of this, it is not possible to rollback the transaction if the old value of the data item is not stored anywhere else. Keep in mind that maintaining the old value of the data item in memory is not sufficient since the contents of memory will be lost in the case of power loss or system failure. Therefore, we need a safe place where the old values of data items can be stored. A transaction log is used for this purpose.

Transaction Log

A transaction log is a sequential file that stores the values of the data items being worked by transactions. Like the blocks written on a magnetic tape, a transaction log is a sequential device. To reach a certain record within the log, the log must be processed sequentially, either from the beginning or from the end. Typically, log records for a given transaction are linked together for ease of processing.

The transaction log is used for two purposes. The first use of the log is to support commit protocols to commit or abort running transactions. The second use of the log is to allow recovery of the database in case of a failure. In the first case, we use the log information to redo— repeat the changes of a transaction—or undo— remove the changes of a transaction from a database. Redo and undo are sometimes called transaction roll forward and rollback, respectively. When a transaction is rolled forward its changes are redone— rewritten to the database. When a transaction is rolled back, its changes are undone— removed from the database. In the second case, the log information is used to recover a database after a power failure or a disk crash. If power fails, incomplete transactions have to be undone. The log information is used to achieve this. Also, as we will discuss later, if the disk crashes, completed transactions need to be redone. Again, the log information is used to achieve this.

Information that is written to the log is used for recovering the contents of the database to a consistent state after a transaction rollback or after a failure. Without this information, recovery may not be possible. Technologically, today the safest place a log can be stored is on the disk. However, if the disk fails and the log is on it, the log is lost. That is, of course, not too troubling

KAHE

if the database is still intact. Losing the log and the database at the same time, obviously, is disastrous.

To make sure this is not the case, one of the following alternatives can be utilized:

1. Use tapes for the log. Although this separates the database storage (disk) and log storage (tape), this is not typically the choice of DBAs because writing to and reading from the tape is very slow. Some of the DBMSs, such as Oracle, archive the inactive portion of the log—the log portion that pertains to transactions that have been competed—on tape.

2. In smaller systems where the log and the database share the same disk, one can use different disk partitions for the database and the log files. Doing so reduces the probability of loss of the database and the log at the same time when a portion of the disk is damaged. In this case, if the disk completely fails or the disk controller goes bad, both the log and the database will be lost.

To guard against the issue with the second alternative, we can store the log and the database on separate disks. This is typically the case for larger systems where the database may require more than one disk for its storage. In this case, we can use a separate disk for the log files. In very large systems, a RAID (redundant array of independent disks) system may be used to provide for recovery of the disk contents by rebuilding the contents of a failed disk from other disks. Regardless of the alternative used for maintaining the log, the goal is to keep the log safe from power loss and/or disk failure. The storage type that is used for the log is known as the ―stable storage‖ as categorized below.

KAHE

DBMS Storage Types

There are three types of storage in the storage hierarchy of a DBMS:

1. Volatile Storage (Memory). Memory is a fast magnetic device that can store information for the computer to use. It is called volatile storage since loss of electrical power causes it to lose its contents. That is why memory is used for temporary storage of information to

serve the need of the CPU. During normal operations of the system, memory-based log files can be used for database rollback, since during the rollback, the transactions are still running and the memory contents are still intact. On the other hand, the memory cannot serve as a storage medium for logs required for database recovery after a failure since the memory contents of a failed system are lost.

2. Nonvolatile Storage (Disk, Tape). Disk and tape are magnetic devices that can store information in a way that withstands shutdowns and power losses. The information written to a disk or a tape stays intact even when the power is disconnected. Although disks and tapes are nonvolatile with respect to power loss, they are still volatile to medium failure—a disk crash causes the loss of information on the disk. As a result, a log that is stored on the disk is lost when the disk crashes.

3. Stable Storage. Stable storage is a storage type that can withstand power loss and medium failure. The log is written to stable storage so that the information in the log is not lost when the system is subjected to power failure or disk failure. DBMSs usually maintain duplicate or triplicate copies of the log files for resiliency. Lampson and Sturgis assume that replicating the log on the disk (persistent storage) is stable. For the rest of our discussion in this chapter, we also assume a replicated log on the disk and we refer to the disk as stable storage.

Log Contents

The log is a sequential device consisting of a set of records. A log record stores information about a change from a single transaction. In addition to the time of the change, specific information about the transaction ID, the DML operation, the data item involved, and the type of operation are recorded. Specific record contents depend on the mode of update. To represent a log record, we use the notation ―.‖. Obviously, since read operations do not change data item values, they are not tracked in the log. There are many types of log records that support different strategies for updating the database—immediate versus deferred—and recovery alternatives for power and disk failures. We will start by investigating the transaction update modes and their impact on what the log should contain.

KAHE

Deferred Update Mode Log Records

In this mode of update, the DBMS does not write the changes of a transaction to the database until a transaction decides to commit (the commit point). There are two basic approaches used for deferred updates. Ozsu and Valduriez outline the details of these approaches in what they call ¨ ―out-of-place‖ update. We briefly mention the two approaches here.

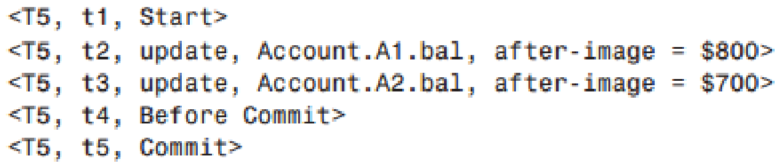
The first approach uses the concept of differential files, which has been used in the operating system for file maintenance for many years. Severance and Lohman proposed to extend the concept to database updates. In this approach, changes are not made to the database directly, but instead, they are accumulated in a separate file as the difference. A file could accumulate changes from a single transaction or from multiple transactions. There are two approaches to merging the changes from the differential files with the database. A transaction's changes can be merged into the database when the transaction commits. Alternatively, changes can be left in the file and be used as the new values for the changed data items. In this case, the DBMS has to keep track of the new values for data items that have been worked on in the files. Once these files become too large and the overhead becomes too high, the files are merged into the database and the differential files are discarded.

The second approach uses the concept of ―shadow paging‖. In this approach, the database page containing the data item being changed is not modified. Instead, the change is made to a copy of the page called the ―shadow page.‖ The actual database page holds the old data item value (used for undo) and the shadow page contains the new data item value (used for redo). If the transaction decides to commit, the shadow page becomes part of the database and the old page is discarded. Logging in conjunction with shadow paging has been used to enable database recovery after a failure in IBM's System R.

As a direct result of deferred updates, for each data item being changed, the old value (also known as the ―before-image‖) is in the database until just before the transaction commits. Therefore, the log only needs to tack the new value (also known as the ―after-image‖) for the data item being changed in the log. Therefore, in the deferred update mode, the old values of the data items being changed by a transaction do not need to be written to the log. To see what is

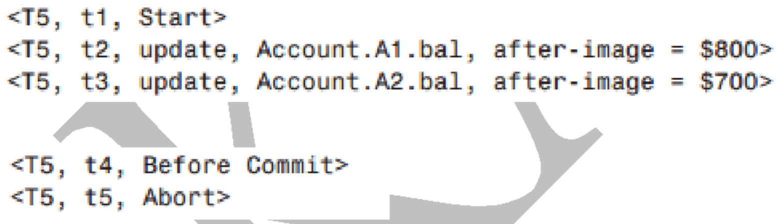
KAHE

stored in the log for a database that uses the deferred update mode, let T5 be a fund transfer transaction that transfers $200 from account A1 with the balance of $1000 to account A2 with the balance of $500. For this example, the records written to the log for T5 are



```
<T5, t1, Start>
<T5, t2, update, Account.A1.bal, after-image = $800>
<T5, t3, update, Account.A2.bal, after-image = $700>
<T5, t4, Before Commit>
<T5, t5, Commit>
```

Notice that every record has a time marker indicating the time of the event. The first record indicates that transaction T5 has started. The second and third records track the writes that T5 makes, the time of these events, and the after-images for the data items being changed. The fourth record indicates that there will be no more changes made by this transaction; that is, the transaction has entered its commit point at time t4. This log shows a successful completion of T5, which is indicated by the existence of the commit record at time t5. During normal operations, transferring the new values for the two data items from the memory buffers is followed by writing a commit record to the log. If, on the other hand, T5 had decided to abort the transaction, the log would have contained the following records:



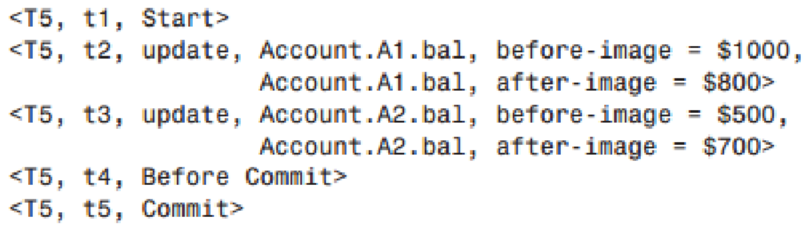
```
<T5, t1, Start>
<T5, t2, update, Account.A1.bal, after-image = $800>
<T5, t3, update, Account.A2.bal, after-image = $700>

<T5, t4, Before Commit>
<T5, t5, Abort>
```

For deferred updates, when T5 aborts there is no need to undo anything. That is because T5 does not write anything to the database until it decides to commit. If a failure happens, depending on the time and type of failure, T5 needs to be undone or redone (see Section 8.4). If T5 needs to be redone, the after-images from the log are used. Again, there is nothing to do to undo T5.

Immediate Update Mode Log Records

In this mode of update, the DBMS writes the changes of a transaction to the database as soon as the values are prepared by the transaction. As a direct result of this, the old values of the data items are overwritten in the database. Therefore, the log not only needs to track the after-images but also the before-images of the data items changed by the transaction. If T5, as discussed above, runs a system with immediate update, the log records will look like the following:



```
<T5, t1, Start>
<T5, t2, update, Account.A1.bal, before-image = $1000,
                 Account.A1.bal, after-image = $800>
<T5, t3, update, Account.A2.bal, before-image = $500,
                 Account.A2.bal, after-image = $700>
<T5, t4, Before Commit>
<T5, t5, Commit>
```

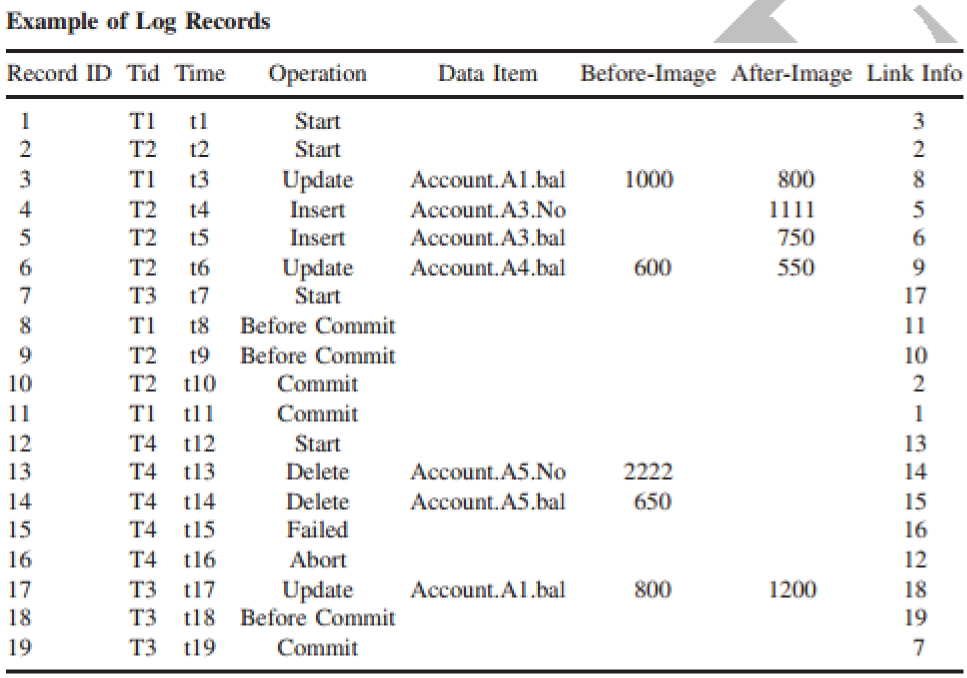
When T5 decides to commit, the new values for the two data items are already in the database. Therefore, the only action required is indicating the successful completion of the transaction by writing a ―Commit‖ record to the log. In immediate update mode, the DBMS has access to the before-and after-images of data items being changed in the log. The DBMS can undo T5 by transferring the before-images from the log to the database and can redo T5 by transferring the after-images of the data items to the database when needed. It should be obvious that in a multitransaction system the records that different transactions write to the log are interleaved.

The log record contents discussed above are generic and do not indicate implementation by any specific DBMS. Most DBMS vendors use two or three additional fields in each record to help traverse the records for a given transaction. A linked list is used to connect together all records that belong to the same transaction. Table 8.1 depicts an example of log records for four transactions ―T1,‖ ―T2,‖ ―T3,‖ and ―T4,‖ including the link fields (Record ID and Link Info in the table). This information is helpful in locating records for a given transaction quickly during recovery from a failure. Another point worth mentioning is that commercial DBMSs do not

KAHE

include ―before commit‖ and ―failed‖ records in the log and therefore eliminate these states. These states are recorded as part of the commit and the abort states.

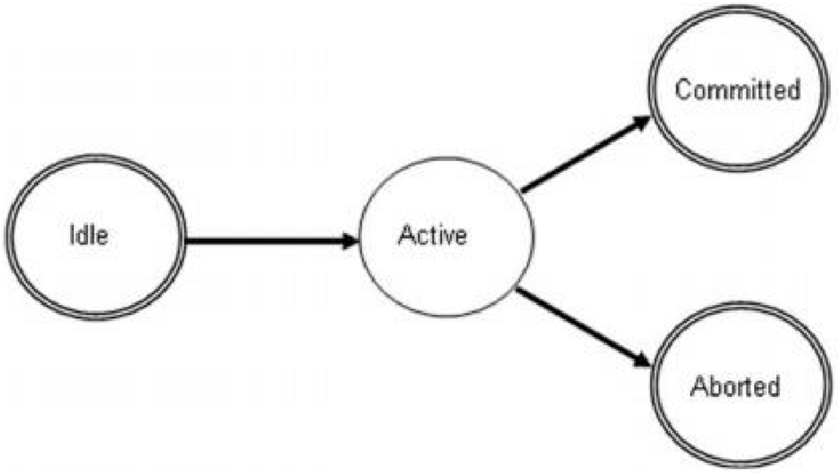
This approach speeds up the log management for committed and/or aborted transactions. The assumption is that most of the time transactions succeed and they are not subjected to failures. As a result, it seems unnecessary to do a lot of recordkeeping for dealing with failures. The drawback to this approach, therefore, is the payback when failures do happen.



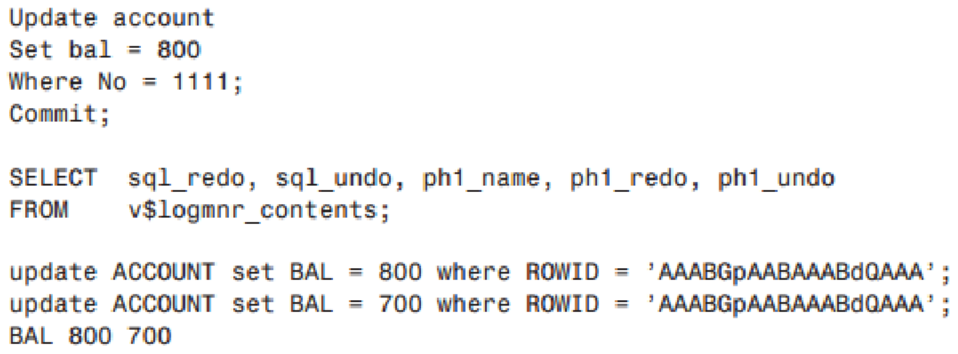
Example of Log Records

Record ID	Tid	Time	Operation	Data Item	Before-Image	After-Image	Link Info
1	T1	t1	Start				3
2	T2	t2	Start				2
3	T1	t3	Update	Account.A1.bal	1000	800	8
4	T2	t4	Insert	Account.A3.No		1111	5
5	T2	t5	Insert	Account.A3.bal		750	6
6	T2	t6	Update	Account.A4.bal	600	550	9
7	T3	t7	Start				17
8	T1	t8	Before Commit				11
9	T2	t9	Before Commit				10
10	T2	t10	Commit				2
11	T1	t11	Commit				1
12	T4	t12	Start				13
13	T4	t13	Delete	Account.A5.No	2222		14
14	T4	t14	Delete	Account.A5.bal	650		15
15	T4	t15	Failed				16
16	T4	t16	Abort				12
17	T3	t17	Update	Account.A1.bal	800	1200	18
18	T3	t18	Before Commit				19
19	T3	t19	Commit				7

As an example, assume a system that does not write the ―Before Commit‖ to the log to indicate that the transaction has finished preparing the new values for the data items. When a failure during the commitment of the transaction happens, since there is no ―Before Commit‖ record in the log, the LRM has to assume that the transaction was active when the failure happened and will have to rollback the transaction. It should be easy to see that the existence of the ―Before Commit‖ record in the log tells the LRM to commit the transaction using the new values of the changed data items from the log. We encourage readers to map the structure discussed above with specific DBMS implementation. For example, the following displays an update statement that changes the balance of account 1111 and the log records that are extracted by Oracle Log Miner from the log running the select statement shown against the ―v$logmnr‒contents‖ dictionary view.



Reduced state transition diagram for a transaction.



```
Update account
Set bal = 800
Where No = 1111;
Commit;

SELECT  sql_redo, sql_undo, ph1_name, ph1_redo, ph1_undo
FROM    v$logmnr_contents;

update ACCOUNT set BAL = 800 where ROWID = 'AAABGpAABAAABdQAAA';
update ACCOUNT set BAL = 700 where ROWID = 'AAABGpAABAAABdQAAA';
BAL 800 700
```

As can be seen from the results of the select statement, Oracle records the actual SQL statement in the log record to be able to undo or redo a column change. In addition, Oracle tracks the operation type, before-image, and after-image for the column being changed. Note that there are many other columns used in this dictionary view in Oracle that we did not print. These columns record transaction ID, transaction name, timestamp, table, tablespace, and so on.

POSSIBLE QUESTIONS

PART B

(EACH QUESTION CARRIES SIX MARKS)

1. Explain in detail about deadlock handling in DDBS.
2. Discuss in detail about replication control.
3. Discuss in detail about Deadlock.
4. Explain in detail about replication control scenarios.
5. Discuss in detail about deadlocks in centralized systems.
6. Describe in detail about replication failures and commit protocols.
7. Explain in detail about deadlocks in distributed systems.
8. Describe about Commit protocols.
9. Describe in detail about distribution deadlock detection techniques.
10. Discuss about the terminologies commit protocols.

PART C

(EACH QUESTION CARRIES TEN MARKS)

1. Discuss in detail about Cryptography.
2. Discuss in detail about terminologies of failure and commit protocols.
3. Discuss in detail about Query processing in centralized systems.
4. Explain i)Vertical Fragmentation ii)Horizontal Fragmentation in detail.
5. Discuss in detail about DBE taxonomy.

KAHE

KARPAGAM ACADEMY OF HIGHER EDUCATION
DEPARTMENT OF COMPUTER APPLICATIONS
DISTRIBUTED DATABASE MANAGEMENT SYSTEM

SUBJECT CODE:17CAP405D **CLASS : II MCA** **SEMESTER: IV**

Unit-4

Questions	Op1	Op2	Op3	Op4	Op5	Op6	Answer
The execution cost is expressed as a weighted combination of I/o_______ and communication cost	SMPS	HTTP	CPU	CMPT			CPU
How many components involved in query optimization ?	1	2	3	4			3
_______ is the set of alternative execution plans to represent the input query	cost model	search starategy	query	search space			search space
_________ predicity of the cost of a given execution plan	search strategy	cost model	query	search spcae			cost model
________ strategey explores the search space and selects the best plan using the cost model	costmodel	query	search space	search strategy			search strategy
_________ is a tree such that at least one operand of each operator node is a base relation	linear tree	busy tree	join tree	tree view			linear tree
______ is more general and may have operators with no base relations as operands	bushy tree	liner tree	join tree	tree view			bushy tree
Deterministic strategies proceed by ____ plan	dynamic	deterministic	building	search strategy			building
_______ strategies allow the optimizer to trade optimization time for execution time	normalized	transformation	optimization	randomized			randomized
query optimization techniques are often extensions of the techniques for _____ system	INGRES	Quel	centralized	decentralized			centralized
_________ Is an important aspect of centralized query optimization	joins	semijoins	union	odering joins			odering joins
The semi join acts as a ________	size reducer	join queries	super join	left join			size reducer
The optimization timing is dynamic for ______	SDD --1	R * algoritham	distributed INGRES	relational			distributed INGRES
which one handle fragments ?	INGRES	R * Algorth	SDD--I	relational			INGRES
Distributed INGRES considers ______	size	local	processing	size, local and processing			size, local and processing
R * co ordinated by a ________	normal site	master site	apprentice site	website			master site
_______ sites are the other sites that have relations involved in the query	normal site	master site	apprentice site	website			apprentice site
SDD 1 is derived from_________	hill climbing	query processing	optmization	normalization			hill climbing

which is described the both highspeed & medium speed network	R*	INGRES	SDD	Relational			R*
The concept of a ________ is used within the database domain as a basic unit of consistent and reliable computing	transaction	transaction consistency	consistent state	reliable			transaction
______ are quite informal	consistent	reliable	consistent and reliable	durability			consistent and reliable
A replicated database is ________ if all the copies of every data item in it have identical values	mutually consistent state	one copy equivalance	resiliency	recover			mutually consistent state
Reliability refers to both the ______________	resiliency	recover	resiliency and recover	consistent			resiliency and recover
The fundamental properties of a transaction__________ as the term is used in database system	atomicity	durability	atomicity and durability	recover			atomicity and durability
The transaction can complete its task successfully is called ?	commits	aborts	update	null			commits
The transaction stops without completing its task is called ?	commit	aborts	update	null			aborts
Actions are updone by returning the database to the state before their execution is called?	commit	aborts	rollback	deadlock			rollback
The data items that a transaction reads are said to constitute is called ?	write set	read set	write and read set	base set			read set
The data item that a transaction writes are said to constitute is called	write set	read set	write and read set	base set			write set
The union of the read set and write set of transaction is known as____	write set	read set	write and read set	base set			base set
dynamic database have to deal with the problem of	phantoms	static	dynamic	tuples			phantoms
Transactions are consist of how many properties ?	1	2	3	4			4
ACID of transation represents atomicity	consistency	Isolation	durability	consistency, isolation and durability			consistency, isolation and durability
Which transaction is treated as a unit of operations	atomicity	consistency	Isolation	durability			atomicity
________ of a transaction is simply its correctness	atomicity	consistency	Isolation	durabiltiy			consistency
Data values that have been updated by a transaction prior to its commitment is called?	dirty	degree	failure	EOT			dirty
Which requires each transaction to see a consistent database at all times ?	atomicity	consistency	durability	Isolation			Isolation
Which ensures that once transaction commits the result cannot be erased from the database ?	atomicty	consistency	durability	isolation			durability
Transaction may be classifed as on line or ________	batch	shortlife	longlife	restricted			batch

_________transactions are characterized by very short execution/response time	online	batch	longlife	online, batch			online, batch
______ transaction take longer to execute and access a larger portions of the database	oneline	batch	shrotlife	long life			batch
________ transaction have a single start point and single termination point	nested transaction	flat transaction	transaction	subtransaction			flat transaction
Transaction include other transaction with their own begin and commit point is known as _________	nested transaction	flat transaction	transaction	subtransaction			nested transaction
The transaction that are embeded in another one are usually called______	nested transaction	flat transaction	transaction	subtransaction			subtransaction
transaction from existing ones simply by inserting the old one inside the new one as ___________ transaction	sub	nested	flat	work flow			sub
A collection of task organised to accomplish some business process is called ?	Human oriented work flow	system oriented work flow	transactional work flow	work flow			work flow
The transaction workflow consist of __________	human oriented	system oriented	both human and system oriented	remote system			both human and system oriented
concurrency control deals with _________ properites	Isolation	consistency	transaction	isolation and consistency			isolation and consistency
_________ is probably the most important parameter in distributed system	Level of concurrecny	distributed	semantic	consistency			Level of concurrecny
In serializability how many conflicts are used ?	1	2	3	4			2
________ algorithm synchronizes the concurrent execution of transaction early in their execution life cycle	pessimistic	ordering	optimistic	locking based			pessimistic
The pessimistic group consist of ________ algorithms	locking based	ordering	optimistic	pessimistic			locking based
one of the copies of each lock unit is designated as the primay copy is known to be ___________	centralised locking	primary copy locking	decentralisedlocking	none of these			primary copy locking
The lock management duty is shared by all the sites of a network is called ?	centralised locking	primary	decentralised locking	copy locking			decentralised locking
________ class involves organising the execution order of transactions	timestamp ordering	hybrid	fragmentation	semantic			timestamp ordering
communication between the transaction manager at the site where the transaction is initiated is called_________	primary site 2PL	centralized 2Pl	2PL lock	co ordinating TM			co ordinating TM
primary copy 2PL was proposed for the prtotype _________ of INGRES	distributed verison	primary copy 2pl	distributed 2pl	centralized 2PL			distributed verison
The transaction implements the ___________ replica control protocol	coloumn & row	rows	colomn	tuples			rows
______ Is only one of the properties of time stamp generation	uniqueness	monotonicity	time stamp	tuple			uniqueness

UNIT-V
SYLLABUS

DDBE Security: Cryptography- Securing Data . Traditional DDBE Architectures: Classifying the Traditional DDBMS Architecture- The MDBS Architecture Classifications- Approaches for Developing A DDBE- Deployment of DDBE Software

DDBE SECURITY

Security is a key facet of any database deployment. It is essential that we authenticate database users (ensuring that they are who they claim to be), allow only authorized users access to information, and maintain overall system data integrity. There are also many subtle security issues, specific to databases, such as SQL injection and inference attacks. Distributed database environments (DDBEs) require communication, so we must ensure not only that the data in databases is secure but that the communication links between users and their data and among the communicating DDBE components are also secure.

CRYPTOGRAPHY

Cryptography is the science of creating secrets. Cryptanalysis is the science of breaking secrets. The related science of hiding secrets is called steganography. To create a secret, we can use either codes, which map whole words to other words, or ciphers, which map individual characters (bytes) to other characters (bytes). The use of codes declined after World War II and today ciphers are most commonly used with digital data. The original (unencrypted) words or characters are referred to as the original message, the unencrypted message, or more specifically as the plaintext, while the characters we map to are referred to as the encrypted message or, more specifically, as the ciphertext.

Specific cryptographic functions include the following:

• Confidentiality keeps messages private between parties even in the face of eavesdroppers attempting to snoop while data is transported over communication networks or while it resides in the database.

KAHE

- Authentication allows message receivers to validate the message source and to ensure the integrity of the message.
- Nonrepudiation validates the message source so strongly that the message sender cannot deny being the message sender (this is a stronger form of authentication).

There are a number of cryptographic building blocks that provide these functions:

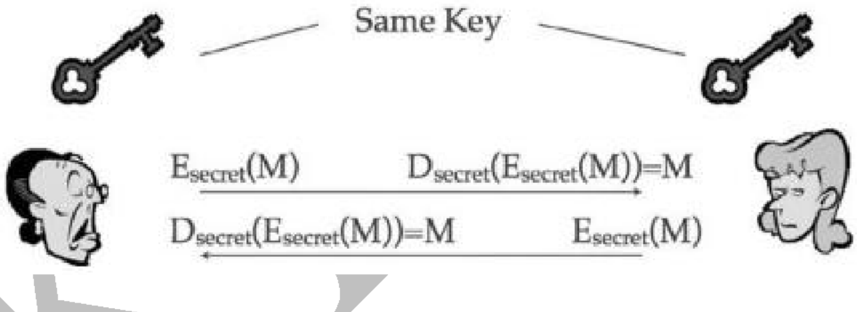
- Conventional cryptography provides confidentiality with previously distributed keys.
- Message digests (MDs) and message authentication codes (MACs) provide authentication.
- Public key cryptography provides confidentiality without prior key distribsution.
- Digital signatures provide nonrepudiation.
- Digital certificates and certificate authorities authenticate public keys.

Conventional Cryptography

Conventional cryptography is a simple concept. Suppose we have two parties who want to communicate securely with each other; let's call them Alice and Bob. Further suppose that Alice and Bob each have their own copy of the same secret key, which is a random string of bits. When Bob wants to send a message to Alice and wants to ensure that no eavesdroppers can read it, he encrypts his message with an encryption algorithm that uses his copy of the secret key before he sends it. After receiving the encrypted message from Bob, Alice uses a corresponding decryption algorithm that uses her copy of the same secret key to decrypt the message so she can read it. If Alice wants to send a message to Bob, she can use exactly the same process (sending an encrypted message to Bob by encrypting her message using her copy of the secret key and then sending the encrypted message to Bob so that he can decrypt the message using his copy of the key). This is shown in Figure, with M representing the message, E representing the encrypt operation, D representing the decrypt operation, and ―secret‖ representing the secret key.

KAHE

There are a variety of algorithms that can provide the encrypt and decrypt operations, but we always assume that the algorithm is known (or at least knowable). The security of these algorithms does not depend on the obscurity of the algorithms themselves; it depends solely on the strength and obscurity of the secret key. Because a key is just a string of bits (typically 128 or 256 bits long) that determines the mapping from plaintext to ciphertext for encryption and from ciphertext to plaintext for decryption, an attacker can always try to break the encryption by trying all of the possible keys. This is called a brute force attack. In order to defend against this attack, we try to use a key space large enough such that the attacker can never, even with large networks of fast computers, try all of the possible keys in any realistic amount of time. Today, a typical key size for conventional cryptography is 128 bits, which yields 2128 or 3.4 × 1038 keys. On average, we need to try half of all the possible keys before we will find the correct one using a brute force attack. If we deployed a million computers, each trying a million keys per second, it would take on average 5.4 × 1018 years to find the key needed to decrypt the message. This is about a billion times the estimated age of the universe!



Alice and Bob use conventional cryptography.

All of the security in conventional cryptography depends on only two things: a sound algorithm (assumed to be known to all) and a long, randomly generated key kept secret between communicating parties. Of course, it is possible that the brute force attack could find the key in a shorter than average time (or take longer than the average time). Since we can never prevent these attacks against the key space, we must use an algorithm that has no weakness that would allow an attacker to find the key or the plaintext more efficiently than brute force.

KAHE

Let us now consider how we can securely transport the shared secret key from Alice to Bob. We cannot encrypt the key—what key would we use to encrypt the key and how would we get that encryption key to the other party? In practice, we could send the key via some alternative communication channel that is hopefully free from eavesdropping, such as email or the telephone or a person with a handcuffed briefcase, but these are cumbersome alternatives. This form of cryptography (and its cumbersome key transfer methods) is known as conventional cryptography because it was the only type of cryptography known before the mid-1970s (and the invention of public key cryptography).

There are two families of conventional cryptographic algorithms: block ciphers and stream ciphers. Block ciphers encrypt a block of bits, typically 128 bits long, at a time and do not maintain state. This means that each resulting ciphertext block does not depend on the encryption of previous blocks (although we will soon see how to add dependency with a concept called modes). Stream ciphers encrypt a bit or a byte at a time while maintaining state from previous encryptions. This means that the same plaintext byte or bit will result in a different ciphertext byte or bit depending on the previous encryption result. These state-dependent mappings help thwart cryptanalysis. Over the years, many conventional cryptographic algorithms have emerged.

With block ciphers (and also, as we will see, with public key cryptography), we need to consider the problem of encrypting data when it does not match the block size of the cipher. The data we have can be smaller or larger than the cipher block size. We use padding to handle the smaller case and modes to handle the larger case. If the data we have is smaller than the cipher block size, we must pad the data, which means that we add bits to the data until it matches the block size. Unfortunately, just adding 0s or 1s is not cryptographically secure, so special padding algorithms, usually matched to specific cipher algorithms, perform this function securely. If the data we have is larger than the cipher block size, we must split it up into chunks that match the block size. The options for doing this are called modes. The obvious way to do this is to just take each block size worth of plaintext and encrypt it independently of any previous encryptions. This is known as electronic code book (ECB) mode. Unfortunately, this approach creates several problems. Since each block of plaintext is encrypted to the same value ciphertext each time, an attacker can build a dictionary of ciphertext values knowing that they are repeats of the same

KAHE

plaintext values. An attacker can also carry out a replay attack by injecting or rearranging ciphertext blocks, causing unexpected results for the receiver.

A better solution is to chain together successive blocks of encryption so that the encryption of one block is dependent on the previous encryptions. This yields ciphertext for a block that is different from the ciphertext generated for the exact same plaintext block elsewhere in the message. This causes any rearrangement of the encrypted ciphertext blocks to result in an incorrect (and incomprehensible) decrypted version of the plaintext because of the broken dependency chain. Many different modes can accomplish this, with different engineering considerations, but the most common one is cipher block chaining (CBC). When using this mode, we have to start the chain with a value called an initialization vector (IV). The IV does not have to be encrypted, but it should be integrity protected and never be reused.

DES/Triple DES

Historically, the most famous algorithm for conventional cryptography was the Data Encryption Standard (DES). IBM submitted an internally developed cipher for a National Institute of Standards and Technology (NIST) standardization effort and it became DES, and reigned from 1976 to 2002. DES uses a 64-bit block size and a 56-bit key length. Unfortunately, it is no longer recommended due to the general vulnerability to brute force attacks (but it has no severe breaks). Several factors contributed to this vulnerability, including the 56-bit key length, the increasing pace of CPU speed improvement, and the ease of harnessing large networks to provide parallel computing capacity. A cipher based on DES as a building block has extended its life. By increasing the key length (to either 112 bits or 168 bits depending on the specific implementation), and encrypting data with DES three times in succession, we get an effective strength of about 256 that of DES alone. This approach is known as Triple DES, 3DES, or DESede. Note that there is no ―Double DES‖ algorithm because an attack called the meet-in-the-middle attack is known. This attack weakens the strength of a Double DES approach—and actually degrades it to the same strength as the original DES. This attack also explains why the strength of Triple DES is only 256 instead of 2112 times as strong as DES alone. While Triple

KAHE

DES is an improvement, it should be viewed as a stopgap solution because of the performance inefficiencies of using DES three times in succession.

AES

In 2002, NIST selected an algorithm called Rijndael as the successor to DES and renamed it. The new name they gave it is the Advanced Encryption Standard (AES). AES has a block size of 128 bits and a choice of three key lengths (128, 192, or 256 bits). The criteria for its selection as the replacement for DES included many factors, such as resistance to all known attacks, good performance when implemented in both hardware and software, efficient resource usage (small memory footprint and small CPU requirements for smart card implementations), and fast key changing times. Currently, best practice is to use the AES algorithm in new applications, typically with a 128-bit key length.

RC4

Ron Rivest of RSA, Inc. created RC4 but never officially released it. Ironically, it was leaked to the public, and once it was leaked, it became commonplace in many applications, ranging from wireless network encryption to browser security via Transport Layer Security (TLS)/Secure Sockets Layer (SSL). RC4 is a stream cipher that encrypts and decrypts one byte at a time. It uses a shared secret key as the seed for a pseudorandom number generator. At the sender side, each successive byte of the random stream is Exclusive-ORed (XORed) with each successive byte of plaintext to produce each successive byte of ciphertext. At the receiver side, each successive byte of the regenerated random number stream (which is the same as the sender's stream because of the common key/seed) is XOR ed with each successive byte of ciphertext to recreate the original plaintext byte.

This approach takes advantage of two properties of XOR. First, if we XOR the same bits twice (in this case, the pseudorandom number stream), the result will be the same as the original

KAHE

data. Second, if we XOR with random bits, then the result (in this case, the ciphertext) will also look like random bits. RC4 can be viewed as a pragmatic implementation of a theoretical cipher known as a one time pad (OTP). When using an OTP, the sender XORs the plaintext with the key (a string of random bits, which is exactly the same length as the plaintext), thereby producing the ciphertext, which is then sent to the receiver. The receiver receives the ciphertext and then XORs it with the same shared key (that string of random bits used by the sender), thereby producing the plaintext. OTPs are impractical because the key lengths need to be as long as the plaintext, the shared key must be perfectly random, and it can never be reused. RC4 makes this idea more practical by generating the key using a pseudorandom number stream. This stream can be exactly as long as needed and uses a shared secret seed value, which can then be viewed as the conventional shared secret key.

Message Digests and Message Authentication Codes

Ensuring data integrity is a fundamental requirement for any database. We must guard against both accidental database modification and malicious attempts to modify the data. Simple error detection and correction algorithms, such as cyclic redundancy codes (CRCs), which were designed to detect errors due to noise and other communication faults, cannot be used for security. If we did use one of these algorithms, an attacker could easily modify both the data and the CRC designed to protect the data in such a way that the receiver would be unaware of the data modification. There are two classes of cryptographic building blocks designed to ensure data integrity, message digests (MDs) and message authentication codes (MACs). MD algorithms take a message of unlimited size as input and produce a fixed sized (often several hundred bits long) output. The output is often referred to as a hash of the input. The algorithm used for MD is designed to quickly process the large number of input bits into the fixed sized output. It is essential that this function is one way, meaning that it is computationally infeasible to reverse engineer the input bit stream from a given output hash value. This means that an attacker cannot modify the data to match the MD value in an attempt to fool the receiver into thinking that the data still has integrity. MACs operate similarly to MDs, except that they use a shared secret key to control the mapping from input message to output hash. Because they use a secret key, MACs can also authenticate data origin.

KAHE

MD5 and SHA MD5 was the most popular MD algorithm. It generates a 128-bit output hash. But, because of weaknesses identified with the algorithm, it is no longer recommended. Despite these weaknesses, it does still exist in many legacy implementations. Its replacement, an algorithm called SHA-1, has a 160-bit output. However, weaknesses were found in SHA-1 as well, so it also is no longer recommended (and it also still exists in many legacy applications). A stronger form of SHA, called SHA-256, is currently recommended while a NIST-sponsored effort is underway to develop a replacement standard MD algorithm. A popular framework for turning an MD algorithm into a MAC algorithm is called HMAC (hash message authentication code). Common versions include HMAC-MD5, HMAC-SHA1, and HMAC-SHA256, which are based on the named MD algorithms.

Public Key Cryptography

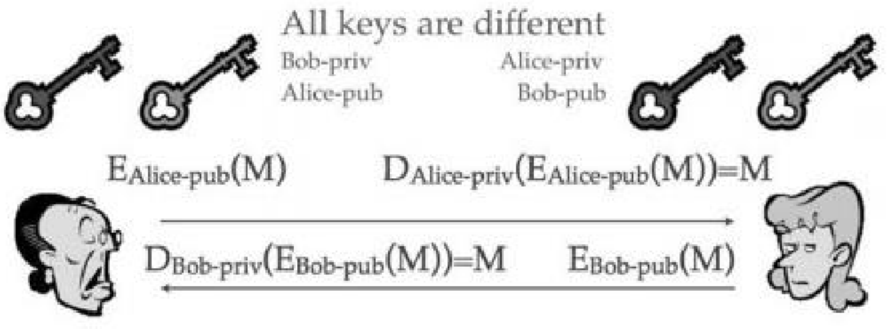
The biggest drawback of conventional cryptography is the requirement for a shared secret key (and the awkward or insecure options for distributing that key). In the mid-1970s, a number of solutions to this problem appeared. All of these solutions extended the notion of a single key to a key pair, consisting of a private key (similar to the secret key of conventional cryptography) and a mathematically related partner called a public key. The public key can safely be revealed to the world. The public key can be sent to any party that wants to send a secret message to the key's owner without any communication channel protection or prior prearranged secret. Consider the example shown in Figure. Whenever Bob wants to send a secret message to Alice using public key cryptography, he first needs to have a copy of Alice's public key. Because this is a public key, Alice can make her public key available to Bob using any mechanism she wants to use (including any nonsecure mechanism). Once Bob has Alice's public key, he can encrypt the plaintext of the message he wants to send using Alice's public key, and then send the resulting ciphertext to Alice. The ciphertext can also be sent using any mechanism, since it is encrypted. Once Alice receives the ciphertext message from Bob, she can decrypt it using her private key. Because Alice is the only one with her private key, she is also the only one who can decrypt messages encrypted using her public key. If she wants to send a message to Bob, she would do the same thing Bob did (obtain a copy of Bob's public key, use it to encrypt the

KAHE

message, and send it to him). With this technology, any two parties can ensure the confidentiality of their communication without previously exchanging any secret information.

Unfortunately, public key cryptography algorithms are much less efficient than conventional cryptography algorithms, typically by several orders of magnitude. This means that it is usually not practical to encrypt or decrypt long messages with public key cryptography. However, we can use it to encrypt a secret key. Then, we can simply send the encrypted secret key using a nonsecure communication channel. Now, we can use conventional cryptography without the awkward secret key distribution issue. By using this hybrid approach, public key cryptography is used for its strength (privately distributing a secret key without prior secret sharing) and conventional cryptography is used for its strength (speed).

We will now look at two algorithms for public key cryptography: the RSA algorithm and the Diffie–Hellman algorithm. There is also a new family of algorithms called elliptic curve cryptography (ECC), which might emerge in the future, but we will not cover them here. Note, as previously discussed, we must pad data that is smaller than a block size and choose a mode for data that is larger than a block size.



Alice and Bob use public key cryptography.

RSA and Diffie–Hellman

Although there are a few different algorithms that can be used for public key encryption, we will only mention two of them. The RSA algorithm is one of the oldest yet still the most popular public key cryptographic algorithm. It is based on the computational difficulty of factoring large numbers into component prime factors. Typically, a user generates a key pair of size 1024 bits (although the best practice is moving to 2048 bits). The RSA block size is typically the same as the key length.

The Diffie–Hellman algorithm does not directly perform encryption. It is an algorithm run between two parties. Each party performs a calculation based on secret information. This secret information is not preshared with the other party. Instead, the two parties exchange some public results of those calculations with each other and then perform another calculation. The algorithm ensures that both parties will have calculated the same result, which can then be treated as a secret key and used with a subsequent conventional cryptographic algorithm such as AES. Diffie–Hellman is an example of a class of algorithms known as key agreement algorithms.

Digital Signatures

Digital signatures (DSs) are an authentication technique based on public key cryptography. A DS can provide authentication, just like a MAC can, but a DS can also provide a more advanced function, namely, nonrepudiation. There is a subtle distinction between authentication and nonrepudiation. Suppose Alice receives a message from Bob, and that message has an associated and appended MAC. Also, suppose that Alice and Bob had previously obtained the secret MAC key. Alice can then take her key and run the message through the MAC algorithm to verify that she generates the same MAC code as the one appended to Bob's message. Because Alice and Bob are the only ones with the shared secret key, Alice knows that only Bob could have originated this message and associated MAC. Therefore, Alice has authenticated Bob as the source of the message (and authenticated the integrity of the message). But, what would happen if Bob denied that he ever sent Alice this message? If Alice took Bob to court, could Alice prove that Bob originated the message? Although Alice knows that Bob is the only one who could have sent this message (because Bob is the only one who had the other copy of the secret key), Bob could claim that since Alice also has a copy of the secret key, she could have composed the message and sent it to herself! Alice would not be able to prove him wrong, so although MACs provide authentication, they do not provide nonrepudiation.

KAHE

With a DS, we can use the public and private key pair and certain public key cryptographic algorithms (RSA is the most popular choice) to perform a completely different function than the encryption function we've already discussed. When Bob wants to send a signed message to Alice, he uses his private key to ―encrypt‖ or sign the message. While this action uses an encryption algorithm, it isn't true encryption because anyone with the corresponding public key can decrypt the message, and anyone can potentially get access to the public key. While the message is not encrypted, it is authenticated, because the message receiver can use Bob's public key to ―decrypt‖ or verify the message. This not only authenticates the message origin to Alice as Bob, but (since Bob is the only holder of Bob's private key) this provides the even stronger notion of nonrepudiation. If Alice takes Bob to court, Alice can prove that Bob is the only one who could have originated the message since he is the only holder of the private key that was used to sign the message. So, Bob cannot completely deny that he originated the message. However, there is still a problem in attaining perfect nonrepudiation from digital signature technology. Bob can claim that his private key was stolen and the thief, not Bob, signed the message. A family of protocols called arbitrated digital signatures can partially address this ―stolen key‖ issue, but we will not discuss them here.

For performance reasons, we typically do not sign an entire message. The sender takes a message and calculates the message digest of the message, and signs (―encrypts‖) that digest with a private key and appends the signed digest along with the plaintext message. The receiver removes the appended signed digest and verifies (―decrypts‖) the digest with the corresponding public key. The receiver then takes the plaintext message and runs it through the same message digest algorithm. If the results of the verified message digest match the results of the message digest calculation, then the receiver knows that the message has integrity and authenticates the message to the sender such that the sender cannot deny message origination. Notice that public key cryptography can be used for two completely different purposes. If we encrypt a message with the receiver's public key, we get an encrypted message, but no authentication. If we sign a message with the sender's private key, we get authentication (and nonrepudiation), but no privacy. We can, however, use both of these functions in sequence.

KAHE

If Bob wants to send a signed, encrypted message to Alice, the following sequence must occur:

- Bob signs the message he wants to send with his private key.

- Bob encrypts his (already signed) message using Alice's public key.

- Bob sends the signed, encrypted message to Alice.

If Alice wants to receive a signed, encrypted message from Bob, the following sequence must occur:

- Alice receives the signed, encrypted message from Bob.

- Alice decrypts the message using her private key.

- Alice verifies the unencrypted message using Bob's public key.

Digital Certificates and Certification Authorities

While public keys are designed to be publicly shared and distributed, they must also be authenticated to their owners. Otherwise, anyone could generate a key pair and then distribute it. The attacker could claim that the public key belonged to someone else, and then use the private key to intercept and decrypt messages meant for the legitimate receiver. The most famous scenario for this attack is called the man-in the-middle attack in which a malicious entity in the middle of a communication path can make two legitimate parties think that they are securely communicating with each other while the malicious entity can decrypt all the traffic.

The most common solution for this problem is to employ a digital certificate. A digital certificate for a user or machine consists of a public key, some identifying information (name, location, etc.), and some dates indicating the certificate validity period. All of this information is unencrypted, but it is signed with the private key of a trusted third party known as a certificate authority (CA), such as Verisign, Inc. For a fee, the CA will verify the identity of a party requesting a certificate and then sign the certificate with their private key. The certificates are in a format defined in a standard called X.509. The CA public key is distributed in browsers and

KAHE

operating systems. Using this technology, users do not send their public keys directly to another party. Instead, they send their digital certificates. Certificate receivers can then verify the CA signature using their own copy of the CA public key (in their browser or operating system). This authenticates the owner of the public key contained inside the certificate, which can then be used for subsequent conventional key distribution, encryption, or digital signature verification. If a security problem with the key or certificate is found, then the certificate serial number is placed on the CA's certificate revocation list (CRL), which should always be checked prior to using any certificate. Most browsers have a setting that automates this checking, but for performance reasons, the default is usually set to not check the CRL.

Digital certificates do not need to be signed by a CA, but can be self-signed or signed by a company for its own use. This works well for internal applications (and there is no external CA cost). External web users, however, must be presented with a CA-signed certificate or they will see a message box in their web browser indicating that the certificate cannot be trusted.

KAHE

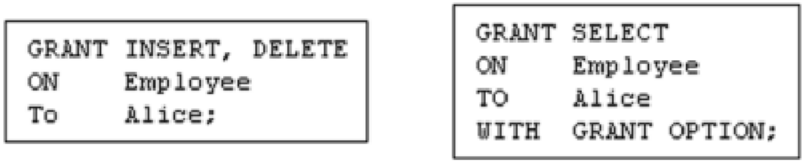
SECURING DATA

While secure communication between users and DDBEs, and between DDBE components themselves, is necessary for overall system security, it is not always sufficient. Sometimes, we must also protect the data residing in the databases from unauthorized viewing and modification. Even when these security measures are in place, we can be vulnerable to various other attacks. In this section, we will begin by examining some techniques for authorizing data access. Then, we will consider techniques for encrypting data. Next, we will look at two classes of database-specific security attacks. The first class of attacks (unvalidated input and SQL injection) depends on deployment and implementation details, while the second (data inference) has more subtle causes. We will close this section with an overview of data auditing.

Authentication and Authorization

Authentication for both DDBE users and components can be implemented using digital certificates, as described above in the TLS protocol using client certificates, or with user ID and password. Security tokens (in the form of a key fob or a credit cardsized device) are increasingly popular extensions to the user ID/password approach. They display a sequence of digits, which randomly change every 30 seconds or so. To authenticate, the user enters his/her user ID, the displayed digit sequence, and his/her PIN, creating a one-time password. This is known as two-factor authentication: something you have (the token) and something you know (the PIN). Biometrics offers a third factor to use for authentication: something you are (such as a fingerprint).

After users are authenticated to the DDBE, they can only access resources if they are authorized to do so. If we model our DDBE security service on the traditional Relational Structured Query Language (SQL) approach, we can offer authorization control via the GRANT statement. If we execute the GRANT command on the left in Figure 9.4, the Alice DDBE Account will be granted the right to insert and delete rows in the Employee table. If we execute the GRANT command on the right in Example 9.1, the Alice User Account will be granted the right to read rows from the Employee table



```
GRANT INSERT, DELETE
ON    Employee
To    Alice;
```

```
GRANT SELECT
ON    Employee
TO    Alice
WITH  GRANT OPTION;
```

Granting the Alice DDBE User privileges on the Employee table.

and her account will also be granted the right to extend this authorization (SELECT) to other DDBE Accounts.

A technology called Lightweight Directory Access Protocol (LDAP) can also support authentication and authorization. Within this context, a directory is a collection of generic information (such as white pages listing people, their phone numbers, addresses, email addresses, etc.) Directories can also contain authentication information such as user ID and password information. Similarly, directories can contain authorization information that specifies what software an authenticated user can or cannot access. By using either a single LDAP server or a

KAHE

collection of cooperating LDAP servers, administrators can use a single facility to control distributed authentication and authorization details. This strategy is called single sign on (SSO). Microsoft Windows Active Directory provides a similar capability. We must secure all the communication links to LDAP servers too, which is often done using TLS.

Firewalls, often specialized for use with databases, can control access at the network level. Administrators can specify parameters such as a list of IP addresses, ports, protocols, dates, and times for which accesses are allowed or denied. This enables the firewall to filter unauthorized network traffic before it arrives at the DDBE components. Intrusion detection systems (IDSs) can monitor network traffic for patterns of malicious traffic (such as viruses, worms, or live attackers) that can potentially compromise a database. The IDS can then alert an administrator to take action. Note that IDSs can be human resource intensive because they typically generate many false positives.

Data Encryption

If the physical hardware systems used by our DDBE components are ever physically compromised, the disks could be removed and scanned for sensitive data. Even if they are never physically compromised, these systems could be attacked across a network from remote machines, from local machines on the network, or even from processes running directly on the host server itself. This means that sensitive data should be encrypted inside the database storage.

Encrypted file systems (EFSs) are becoming increasingly popular for protecting hard disk drives (especially on laptops) containing sensitive data. All the data on the disk is encrypted with a conventional key and accessed via a user password. However, this technology can often create performance problems when used with database systems because database system designers have taken great care to optimize data access assuming specific physical disk layouts that might be disrupted when data is encrypted.

A better approach for encrypting database data is to encrypt the data being stored rather than the underlying file system. There are two basic approaches: internal to the DBE, and external from the DBE. In the first approach, we use components or applications to encrypt the

KAHE

data prior to sending it to the DBE for storage and then use similar components or applications to decrypt the data after it is retrieved from the DBE. Since the DB is storing the encrypted version of the data, the data types used inside our DBs must be modeled to hold the encrypted ciphertext, not the plaintext. This often requires extensive use of binary data types in the database, such as variable-length binary (varbinary) types or binary large objects (BLOBs). In the second approach, because the DBE directly supports encryption, the DDBE users can simply read and write their data normally while the DBE handles the encryption ―under the covers.‖ This is generally the most efficient approach to data encryption for centralized DBMS.

With any type of encryption, we must be careful with key handling, because a compromised key means compromised data. Often, the key is protected by a password. This password must be stronger than the key it is protecting. In this context, stronger has a very specific, technical meaning, called entropy, that refers to more than just the literal password length. A long password consisting of words easily found in a dictionary and using only a subset of possible characters (such as all lowercase letters) has lower entropy than a password that is the same length but uses random, uniformly distributed characters.

Users must never store their passwords directly on a machine unless the passwords are also encrypted using a sufficiently strong encryption mechanism. Software providing password vaulting, a technique where a master password is used to decrypt a list of other passwords, is a reasonably safe way to store passwords. The master password must be at least as strong as the contained passwords. The Mac OS X keychain is an example of a password vault.

We must also be careful when the password is supplied by a machine or process instead of a person. Hard-coding passwords in command scripts can make them vulnerable to both snooping and to system utilities that list the command lines of running processes. Hard-coding passwords in source code can make the passwords vulnerable to string extraction utilities. The best options use operating system provided key protection constructs such as key chains and ACL protected registries. In the future, secure hardware module key storage will probably become increasingly popular.

Unvalidated Input and SQL Injection

KAHE

Unvalidated user input is one of the most common software security flaws. It is responsible for exploits ranging from buffer-overruns, to command injection, to cross-site scripting. It is also responsible for a class of exploits specific to databases known as SQL injection. This attack takes advantage of a lack of user input validation, usually in web-based input fields for forms and in URLs containing post data. The attack uses cleverly crafted input values to sneak in SQL commands that can expose confidential information, allow unauthorized data modification, or even destroy or corrupt the data in the database.

Suppose we have a web-based application that uses web-forms for user input, which it then uses to build SQL commands dynamically. It creates these SQL commands at runtime by concatenating SQL command fragments and the user-supplied input values. Although the actual DDBE authorization and authentication would never be based on this user input, this is an easy example to explore. For each of these scenarios, suppose our web-based application is an online storefront of some kind, and the web-form in the scenario is the customer login and password entry form. Assume we have a relational table named ―CustomerLogin‖ in the DDB that contains a unique ―CustomerId‖ value for each unique pair of ―CustomerName‖ and ―CustomerPassword‖ values. Our web-form must have two user-supplied data fields; let's call them ―inputName‖ and ―inputPass.‖ Now let's consider what happens when this form is attacked using a SQL injection technique.

By passing the Password for Customer Login

Suppose our application dynamically builds a SQL Select statement using the hard-coded strings shown in Figure, and the user input values. Notice that String2 and String3 have a leading space. Both String3 and String4 have a single-quote (apostrophe) at the end, and both String4 and String5 have a single-quote (apostrophe) at the beginning. String5 ends with a semicolon, which is the ―end of command‖ character for SQL. When the customer logging in provides values via the input fields in this form, our application creates a SQL query by concatenating String1, String2, and String3 together, followed by the value of inputName field, then String4, followed by the value of inputPass, and then, finally, String5. This SQL query will return the

KAHE

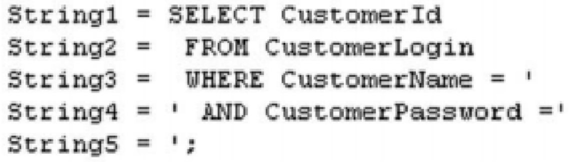
CustomerId value from the relational table where the CustomerName value and CustomerPassword value each match the values specified in the web-form fields.

For an example of what we intended to have happen, if a customer named Alice used this form to enter her name and password (let’s suppose she enters the value ―Alice‖ for inputName and ―Rabbit‖ for inputPass), our application would create the SQL query in Figure. If the specified values (CustomerName equal to ―Alice‖ and CustomerPassword equal to ―Rabbit‖) do not exist for any row in the table, then the SQL query will return an empty set (zero rows returned, a ―SQL-Not-Found‖ error raised, etc.). If this happens, since there is no ―CustomerId‖ value returned for this query, the login attempt will fail. On the other hand, if the SQL query returns a nonempty result set, then it will contain Alice’s CustomerId value, and the login attempt will succeed.

For an example of an unintended consequence, if we have ―unvalidated‖ input for this web-form (if our code does not adequately check for malicious or invalid customer input), it is possible that some attacker could cleverly construct values for these input fields to circumvent both our authentication and authorization mechanisms. Suppose our attacker provides the input values shown in Figure. The value for each field is the same: a single-quote, followed by a space, the word ―OR,‖ and another space, and then two single-quotes followed by an equal-sign and another single-quote. What will happen next?

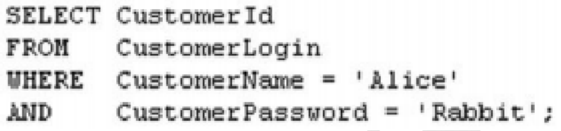
Figure shows the SQL query that our application would create and execute in this scenario. Notice that the single-quotes changed the logic of the WHERE clause. Now, we will return the CustomerId value wherever the CustomerName value is empty OR wherever an empty string equals an empty string. This would effectively return a random CustomerId value and allow the attacker to be logged in as this random customer. By specifying a known customer name for the inputName field, along with this same malicious password value, an attacker can impersonate any customer.

KAHE



```
String1 = SELECT CustomerId
String2 =  FROM CustomerLogin
String3 =  WHERE CustomerName = '
String4 = ' AND CustomerPassword ='
String5 = ';
```

Hard-coded strings used to build SQL query.



```
SELECT CustomerId
FROM   CustomerLogin
WHERE  CustomerName = 'Alice'
AND    CustomerPassword = 'Rabbit';
```

SQL query built for customer "Alice" with password "Rabbit."

fields to circumvent both our authentication and authorization mechanisms. Suppose our attacker provides the input values shown in Figure. The value for each field is the same: a single-quote, followed by a space, the word ―OR,‖ and another space, and then two single-quotes followed by an equal-sign and another single-quote.

Figure shows the SQL query that our application would create and execute in this scenario. Notice that the single-quotes changed the logic of the WHERE clause. Now, we will return the CustomerId value wherever the CustomerName value is empty OR wherever an empty string equals an empty string. This would effectively return a random CustomerId value and allow the attacker to be logged in as this random customer. By specifying a known customer name for the inputName field, along with this same malicious password value, an attacker can impersonate any customer.

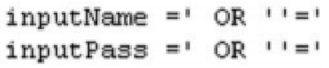
KAHE

By passing the Password for Change Password

This same technique can be used to circumvent other unvalidated web-forms, even when we attempt to enforce security. For example, suppose our attacker has logged in using CustomerName ―Alice‖ and obtained her CustomerId as shown in Figure. Suppose Alice's CustomerId value is ―1234.‖ Now, suppose the attacker wants to change Alice's password—effectively preventing Alice from being able to log in to her own account. Suppose we had another (unvalidated) web-form for changing passwords. Even if this form attempted to provide additional security by requiring the customer to enter the old password value along with the new, the attacker could again use the technique (and values) shown in Figure. Here, the system would provide the CustomerId value, and our attacker would provide the ―old CustomerPassword value‖, and a ―new CustomerPassword value‖ (suppose the new password is ―newValue‖). Figure shows how the exploited SQL Update command might look in this scenario.

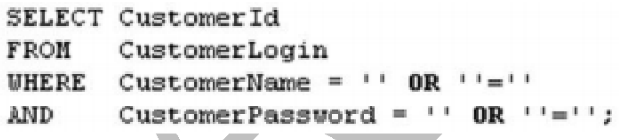
Vandalizing the DDBE

In the previous examples, our attacker was attempting to impersonate one of our customers. Suppose we encounter an attacker who want



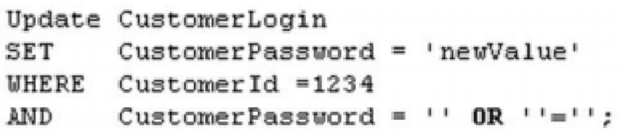
```
inputName =' OR ''='
inputPass =' OR ''='
```

Malicious input values entered by an attacker.



```
SELECT CustomerId
FROM   CustomerLogin
WHERE  CustomerName = '' OR ''=''
AND    CustomerPassword = '' OR ''='';
```

SQL query built from malicious input



```
Update CustomerLogin
SET    CustomerPassword = 'newValue'
WHERE  CustomerId =1234
AND    CustomerPassword = '' OR ''='';
```

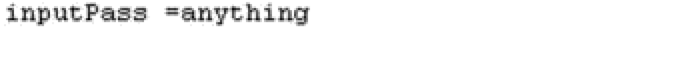
SQL update built from malicious input.

to vandalize our website and DDBE. There are several scenarios where the same SQL injection technique can allow such an attacker to do very damaging things to our system. The issue of unvalidated input again plays a key part in this security breach, but we also have a new class of exposures that increases the potential for harm based on the violation of the least privilege principle, or granting more authority than necessary.

Although there are several different approaches for identifying the right level of authority for a given application or scenario, they are beyond our scope here. Instead, we will focus on a simple example demonstrating what can happen when we are too generous with the authorization policy. When our web application connects to the DDBE, the DDBE authenticates the Account information that the application is using. Whenever the application attempts to execute a query or command, the DDBE verifies that the account has the authority to perform the operations required for request. For example, the account used by our application must have the ability to read the CustomerLogin table if it is going to verify customer names and passwords, and the ability to update the CustomerLogin table if it is going to change customer passwords.

Suppose this account is authorized to do more than simply read and update a single table. In the previous examples, the attacker tricked our application into executing clever variations of the intended commands. In this scenario, the attacker uses the same security vulnerability (unvalidated input) and the same exploit (SQL injection) but, because we have given a tremendous amount of authority to the underlying DDBE account being used by our application, the unintended commands will be more destructive. For example, suppose our attacker uses the same web-form and the same dynamic SQL scenario as we depicted in Figure, but now, the values entered are those in Figure.

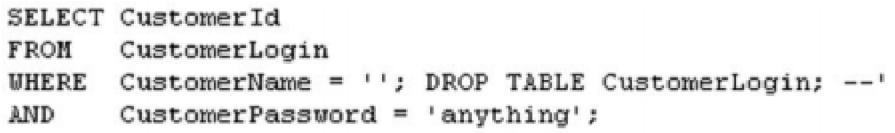
Notice that the inputName starts with a single-quote, and then is followed by a semicolon. Recall that the semicolon terminates a SQL command, which means that the remaining text is a new command. The remaining text in the name value is a command to drop the CustomerLogin table. If the account is authorized to do this command, it will destroy the CustomerLogin table structure and content! The second semicolon in this field is followed by a space and two dashes (hyphens, minus signs). The space followed by two dashes is recognized as a ―begin comment‖ command in most SQL parsers. Figure shows the SQL command generated for this scenario. Everything after the two dashes is ignored. The fact that the SQL query returns no rows is irrelevant, because after the query executes, the entire table will be dropped.



```
inputName ='; DROP TABLE CustomerLogin; --
inputPass =anything
```

KAHE

Even more malicious input values entered by an attacker.



```
SELECT CustomerId
FROM   CustomerLogin
WHERE  CustomerName = ''; DROP TABLE CustomerLogin; --'
AND    CustomerPassword = 'anything';
```

Command built from input values

This means that customer login information will be lost, and nobody will be able to log into the system.

Preventing SQL Injection

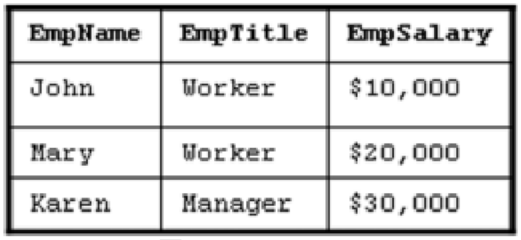
There are several techniques for thwarting this attack. First, we can perform simple input validation, such as ensuring CustomerName values can only contain alphanumeric data (no spaces, punctuation, numbers, etc.). Similarly, verifying that the user-supplied value falls between the minimum and maximum lengths defined for the field can limit the amount of malicious content. Many RDBMS platforms provide a facility known as ―prepared SQL,‖ or ―parameterized SQL;‖ facilities like these can use precompiled type checking as a technique to minimize exposure to attacks such as SQL injections. Unfortunately, there is no such thing as a commercial off-the-shelf (COTS) DDBE product. This means that we must manually create the facilities used to parse and execute commands.

SQL injection is just one example of a broad class of software flaws that can compromise security in applications. In order to guard against these attacks, there are many preventative actions that we need to perform, such as ensuring that software memory buffers on stacks and heaps are not overrun in our components, subsystems, and applications, protecting against injection for operating system commands and DBE commands, and properly handling exceptions and return codes.

Data Inference

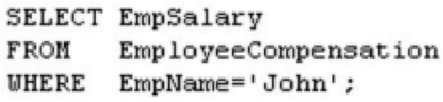
Databases can also suffer from a specific security and privacy vulnerability, known as a data inference attack. This attack is extremely difficult to address, and impossible to completely solve as a general case. When we allow certain, individual queries that are by themselves innocuous, we cannot prevent the aggregation of these results, which then exposes confidential information. It is a specific example of a more general problem: when attackers can create data mining results from disparate sources, each of which is authorized, in order to gain confidential results through the aggregate.

Suppose we have a relational table named EmpCompensation, containing the columns and data needed to capture employee salary. Suppose we have three employees named ―John,‖ ―Mary,‖ and ―Karen‖ (as shown in the figure data), and each employee has an authenticated DDBE account. Employee accounts have the privileges necessary to look up their own salary information, but not the information for other employees. We also allow employees to retrieve statistical information about salaries in the organization (so that they can see where they fall on the salary grid).



EmpName	EmpTitle	EmpSalary
John	Worker	$10,000
Mary	Worker	$20,000
Karen	Manager	$30,000

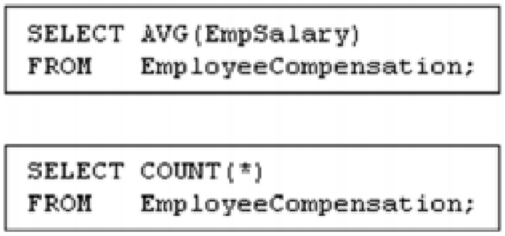
The EmpCompensation table.



```
SELECT EmpSalary
FROM   EmployeeCompensation
WHERE  EmpName='John';
```

Employee John's query.

KAHE



```
SELECT AVG(EmpSalary)
FROM   EmployeeCompensation;
```

```
SELECT COUNT(*)
FROM   EmployeeCompensation;
```

Aggregate queries that any employee can execute.

This means that John could execute the query in Figure, but Mary (or Karen) cannot, but all three employees can execute the query in Figure. Suppose John uses his query to retrieve his salary information and Mary uses a similar query to retrieve her salary information. Each of them can execute aggregate queries on the Employee table, such as those shown in Figure. The first query will return $20,000, while the second query will return three, in this example. Now, if John and Mary collude and share their salaries, the average, and the number of salaries, they can infer their manager Karen’s salary. There are some partial (but no complete) solutions to this problem.

Data Auditing

Data auditing is an essential component of data security. Often, we cannot detect security breaches when they occur, but examining audit logs for evidence can be a very useful post-attack tool. Audit logs can vary in detail, but they can contain useful information such as the date and time for failed access attempts, successful attempts, important modifications made to the system, and other details. We should try to ensure that the audit log is stored separately from the database so that attackers who compromise the database cannot attempt to hide their tracks by modifying the log as well. One approach to accomplish this is to use a separate audit server that can only receive data in a single direction from the database and uses write-only storage.

TRADITIONAL DDBE ARCHITECTURES

In most large organizations, information that is vital to business operations does not reside on a single computer system; instead, this information is scattered across many diverse systems. This distribution of information is more the result of mergers and acquisitions than any corporate strategy or plan. When a large company subsumes several smaller companies, it also

KAHE

inherits the information systems that these smaller companies own. In most cases, these systems do not fit neatly together—often, the database systems are incompatible, the same information is stored in multiple places using different representations, and the level of quality across these systems is inconsistent. The larger company, therefore, is faced with the challenge of application and information integration across these preexisting, incompatible systems that are expected to have inconsistent and duplicated data. If the company in this scenario decides to build a new system in an attempt to address the integration challenges we just described, what kind of system should the company build? This chapter provides some insight into the traditional, architectural alternatives suitable for this task.

There are several ways to categorize data modeling and data storage techniques. Similarly, there are multiple ways to categorize database management systems (DBMSs). There are even different techniques for categorizing the different infrastructures and architectures on which these database environments (DBEs) are built. Before we do that, however, it would be useful to revisit why we prefer to use the term ―DBE‖ in this book, rather than the term ―DBMS.‖ We introduced the concept of a DBE as a way to focus on the similarities between different database implementations, while ignoring any potential differences in their requirements, abilities, or limitations. For example, we said that even though a DBMS is a useful tool, there are times when we do not need all of its power or complexity. Therefore, we developed our own vocabulary. This new term can now safely be used to identify a subsystem within our architecture that would be satisfied by a DBMS, but could also be satisfied by something much less powerful than a DBMS or by something much greater than a mere DBMS. In other words, we introduced this term because the traditional term had too many nonstandard usages and unintended implications and ambiguities.

We introduce our taxonomy-based categorization technique for similar reasons—we think that the nomenclature used by other categorization techniques is unintentionally laden with implications and ambiguities, due in part to the lack of standard definitions for the fundamental terms being used. For example, the term ―distributed database management system‖ has traditionally been used in two distinctly different ways. In the first usage, the term ―DDBMS‖ refers generically to any architecture that supports distributed data. Therefore, it is valid to say

KAHE

that ―a federated database system is one example of a DDBMS, and a multidatabase system is another example of a DDBMS.‖ However, in its second (and still traditional) usage, the term ―DDBMS‖ is interpreted as the name of a specific architectural approach for implementing a distributed database. The architecture to which this second usage refers is radically different from, and incompatible with, many of the other architectural alternatives. Therefore, it is also valid to say that ―the architecture for a federated database system is completely inappropriate and incompatible with any implementation of a DDBMS.‖ It is difficult to discuss the generic architectural issues across these two architectures when both interpretations of ―DDBMS‖ are being used. Similarly, it is potentially confusing to explain how a system with multiple databases is not necessarily the same as a multidatabase system because the former does not meet all of the necessary architectural requirements.

Classifying the Traditional DDBMS Architecture

The DDBMS architecture (meaning the second, traditional interpretation) can be classified in many ways, but we will look at how it fits within our taxonomy first, before we discuss any of the other classification methods. In this architecture, the DDBE consists of several Sub-DBEs (S-DBEs) networked together. The S-DBE's only purpose in this architecture is to serve the DDBE as a whole. In other words, the S-DBEs do not allow independent access from outside the DDBE. For this reason, it could be argued that the S-DBEs are quite different from local database management systems (LDBMSs). Instead, the S-DBEs act more like the Data Accessors or the Data Getters (without any application processor to support outside requests). For example, even if the S-DBEs provided a query processor and a command processor, it would not be LDBMSs. Typically, LDBMSs would provide one or more interfaces to allow both users and applications direct access to the data contained in their databases.

The COS-DAD Level

The first level in the taxonomy is the COS-DAD level. Here, we group all the components and subsystems together (ignoring the differences between them) and focus on how distributed the deployment can be within the architecture. For the traditional DDBMS architecture, we have several databases and several S-DBEs. Each S-DBE is typically deployed

KAHE

at the same site as the database or databases that it manages, but otherwise, all the COSs can be deployed in a highly distributed fashion. The data stored inside the DBE can also be highly distributed (fragmented, replicated, or merely located on different S-DBEs).

The COS-COO Level

The second level in the taxonomy is the COS-COO level. Here, we look at how closed or open the various components and subsystems are. For the traditional DDBMS architecture, we have essentially implemented the system from scratch. Therefore, this architecture is completely open. This is a great advantage, since it means that we can always view and modify the implementation and interface details for any COS in the architecture. However, it also means that in order to implement this architecture, we probably need to write everything ourselves—which can be a monumental task unless we limit the functionality that we will support. For example, we might have restrictions on replication, fragmentation, or even the transactions and write operations—all in an attempt to make the implementation more practical (and possible).

The SAD-VIS Level

The third level in the taxonomy is the SAD-VIS level. Here, we focus on the schema and the data for all things stored in the DDBE—in particular, we will focus on the visibility (ability to access). Since the DBs in this architecture (and S-DBEs) are only used to support the DDBMS as a whole, we can have total visibility (TV). If we have TV, then we have both total schema visibility (TSV) and total data visibility (TDV). In other words, there are no hidden data structures and no hidden data content in this architecture. Every data structure stored in the database must be visible to the distributed database administrator (DDBA), because it is impossible to define any data structures from outside the DDBMS—those operations are not allowed in this architecture. Similarly, all data content must be accessible by the DDBA, because it is impossible to load any data content without using the DDBMS. We mentioned in Chapter 1 that we can artificially mandate partial visibility (PV), which means either partial schema visibility (PSV), partial data visibility (PDV), or both PSV and PDV. This is a common practice, especially in relational DBMS environments—in fact, this is one of the primary purposes of the SQL view, namely, to limit the schema and data visible to particular users or applications. Again,

KAHE

this can be artificially mandated (for several good reasons) for the non-DDBA accounts, but strictly speaking, the architecture requires TV for the DDBA.

The SAD-CON Level

The fourth, and final, level in the taxonomy is the SAD-CON level. Here, we focus on the visible schema and the visible data for all things stored in the DDBE, and we are mainly concerned with the degree of control that the DDBA can exercise. In other words, if the DDBA can perform all possible (valid) schema operations on the visible schema, then we have total schema control (TSC), if the DDBA can do the same for the visible data, then we have total data Control (TDC) and the combination is total control (TC). The partial schema control (PSC), partial data control (PDC), and partial control (PC) are analogous to the previous level's ―partial‖ scenarios. In the traditional DDBMS architecture, we have TC—since partial control would imply that some other environment had control over some piece of schema or data and that is not possible in this architecture. Once again, we could artificially mandate PC for non-DDBA accounts, but the DDBE requires TC for the DDBA.

DDBMS Architectural Summary

The schema for the DDBMS is designed top–down in this architecture and components and subsystems are designed and implemented from scratch. The databases, components, and subsystems can be completely distributed within the environment. Because we write the COSs ourselves, and only access the databases through the DDBMS environment, the system is completely open, and the data and schema are completely visible and under the system's control. We can always artificially impose limits or restrictions on the distribution, visibility, or control—but these are not required by the architecture itself.

THE MDBS ARCHITECTURE CLASSIFICATIONS

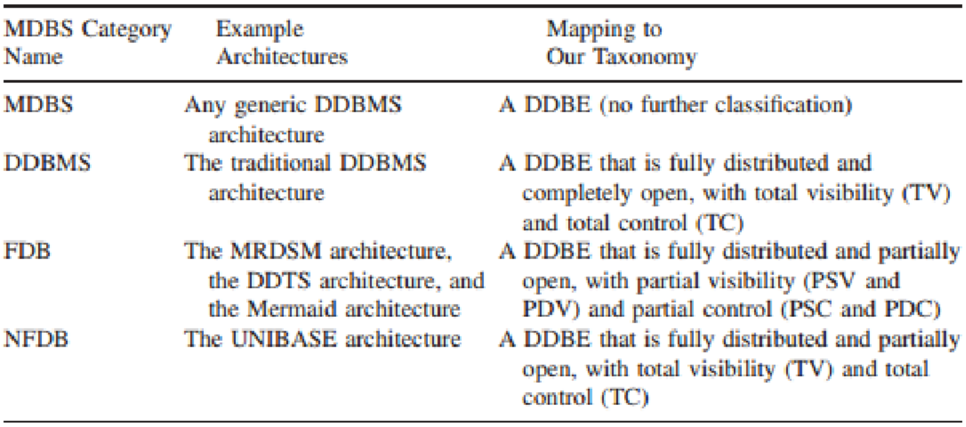
A good representative of other classification techniques for DDBE architectures can be found in a survey by Sheth and Larson. What we call a DDBE in our taxonomy is called a multidatabase system (MDBS) in their classification system. In a MDBS, each S-DBE is called an LDBMS and is actually limited to being a CDBE. This is different from our nomenclature and

KAHE

taxonomy—although all the traditional architectures we have considered so far also have this limitation; the architectures we will discuss in Chapter 13 will use other DDBEs as their S-DBEs. According to Sheth and Larson, a MDBS can be either a nonfederated database (NFDB) system or a federated database (FDB) system. There are several differences between these two subcategories, but the primary distinction is whether the component database systems (S-DBEs) can be used outside the MDBS. In an FDB, the components can freely be used outside the FDB, while component database systems participating in an NFDB cannot be used in this way. This distinction makes the component LDBMS systems in an FDB autonomous while they are nonautonomous in an NFDB. Therefore, the FDB architecture provides partial visibility and partial control while the NFDB MDBS architecture allows total visibility and total control. An example of an NFDB is the UNIBASE architecture. The COSs within FDBs can be coupled to each other loosely or tightly, Sheth and Larson define a loosely coupled FDB as one that provides its users with an imported, global view of the schemas defined by the local systems; while a tightly coupled FDB provides its users access to an integrated, global schema. The difference in coupling is essentially the distinction between an imported view and an integrated view. An example of a loosely coupled FDB is the MRDSM architecture. Tightly coupled FDBs are further divided into single-federation systems, such as DDTS, versus systems with multiple federations, for example, Mermaid. In a single– federation system, all of the DDBE users have access to a single, integrated schema. In a multiple federation FDB, there are multiple, integrated schemas. Other classifications have also been proposed that focus on whether or not database systems and data models are homogeneous or heterogeneous.

KAHE

Mapping the MDBS Classifications into Our Taxonomy



MDBS Category Name	Example Architectures	Mapping to Our Taxonomy
MDBS	Any generic DDBMS architecture	A DDBE (no further classification)
DDBMS	The traditional DDBMS architecture	A DDBE that is fully distributed and completely open, with total visibility (TV) and total control (TC)
FDB	The MRDSM architecture, the DDTS architecture, and the Mermaid architecture	A DDBE that is fully distributed and partially open, with partial visibility (PSV and PDV) and partial control (PSC and PDC)
NFDB	The UNIBASE architecture	A DDBE that is fully distributed and partially open, with total visibility (TV) and total control (TC)

APPROACHES FOR DEVELOPING A DDBE

There are two main approaches for developing a DDBE—the first approach uses a top–down methodology to create the entire system from scratch, while the second uses a bottom–up methodology to integrate the DDBE from its component S-DBEs. The top–down or bottom–up approaches apply to both software and database schema creation.

Before delving into the details of the top–down and bottom–up alternatives, we will review the architecture of a DDBE.

The Top–Down Methodology

In the top–down approach, we develop the entire set of DDBE software components from scratch. In this approach, we design, develop, test, and deploy all required software components such as the user interface, transaction management, recovery and fault tolerance, and optimization services. We can choose to use any programming languages or development environments we like for these tasks. In effect, we act as the DDBE software vendor. Taken in the extreme, this is not a good idea. In practice, we feel that this approach is only acceptable for experimental or research-oriented DDBE projects. This approach is simple to implement in the sense that it avoids the headaches normally associated with integrating new software with existing DBMS software. However, we must be willing to invest the time and money necessary to develop all the required software components, and we must have the necessary skills and experience to implement this DDBE project by ourselves.

KAHE

Creating the GCS Top–Down

When a DDBE is created top–down, the global conceptual schema (GCS) is created first. Next, the requirements for local conceptual schemas (LCSs) are determined, and the global constructs are distributed and deployed into LCSs as needed. Information is loaded into the local databases as specified by the local system's LCS. Therefore, the DDBE has total visibility and total ownership and control over the information in the system.

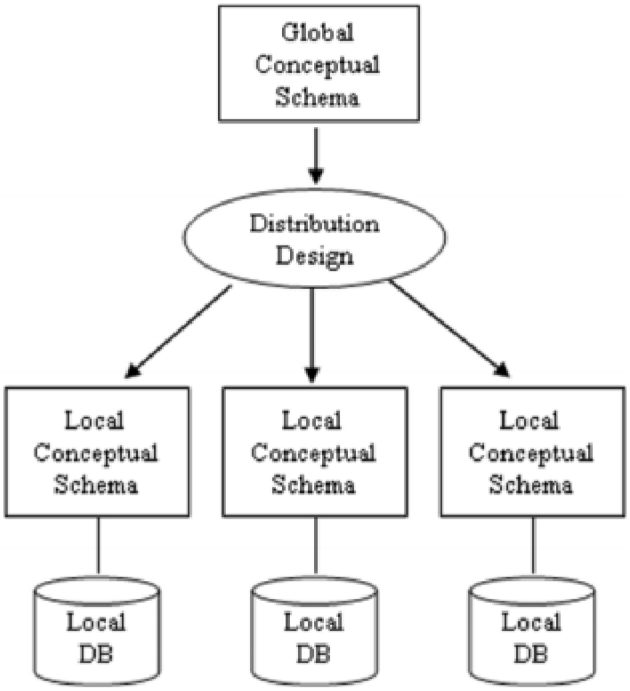
Developing the Software Top–Down

In a top–down approach, we develop all the necessary software components according to a predetermined set of requirements. The top–down strategy is typically used for experimental and research situations where total control is necessary.

Bottom–Up Methodology

In addition to inherent philosophical differences between top–down and bottom–up methodologies, it is important to mention that in a top–down implementation the DDBE ends up with only one GCS—the original GCS that we started with. By contrast, in the bottom–up implementation, the GCS is created by integrating portions of participating LCSs, and, as a result, we have the option of creating more than one GCS. We can create multiple GCSs to satisfy the needs of different groups of users and different applications.

KAHE



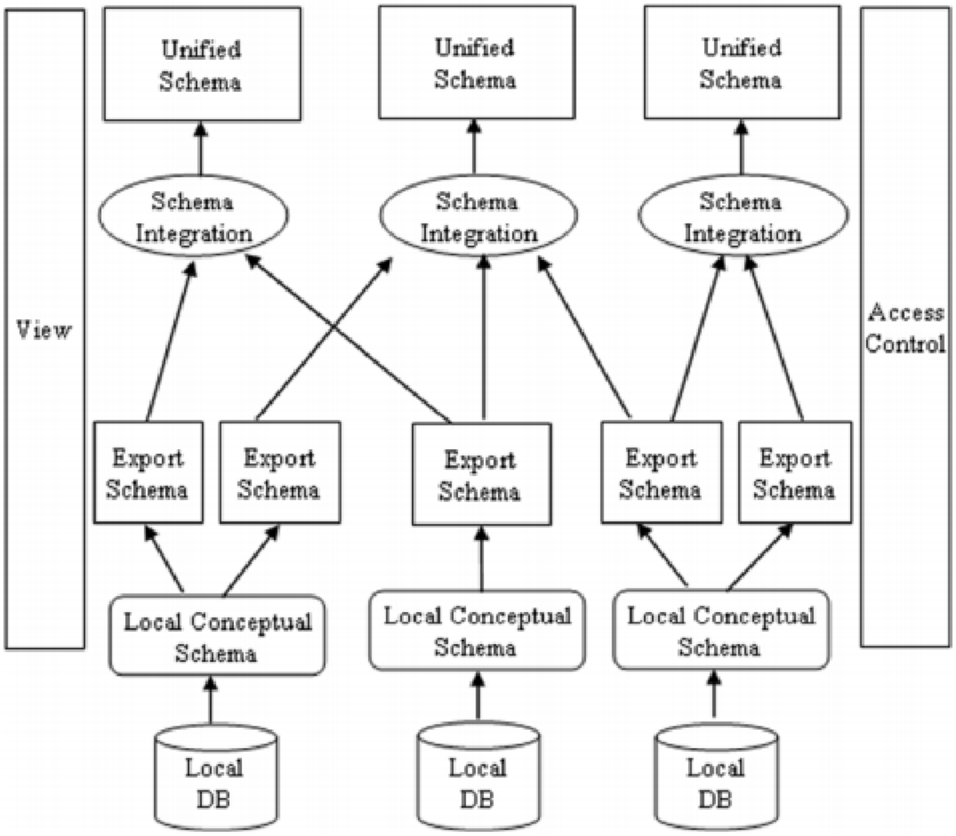
Top–down information architecture design.

Creating the GCS Bottom–Up

When we use the bottom–up approach to implement a GCS, each local database administrator (DBA) decides what degree of visibility and control to allow the DDBE to have. Each DBA defines a subschema for each local database. Each subschema identifies which parts of the schema and data will be made visible to the DDBE. Each subschema (called an export schema is then integrated with the other relevant subschemas from the other local databases in the environment. This integration results in the creation of a single integrated GCS (called the unified schema) as implemented in DDTS or the creation of multiple unified schemas as implemented in Mermaid.

Developing the Software Bottom–Up

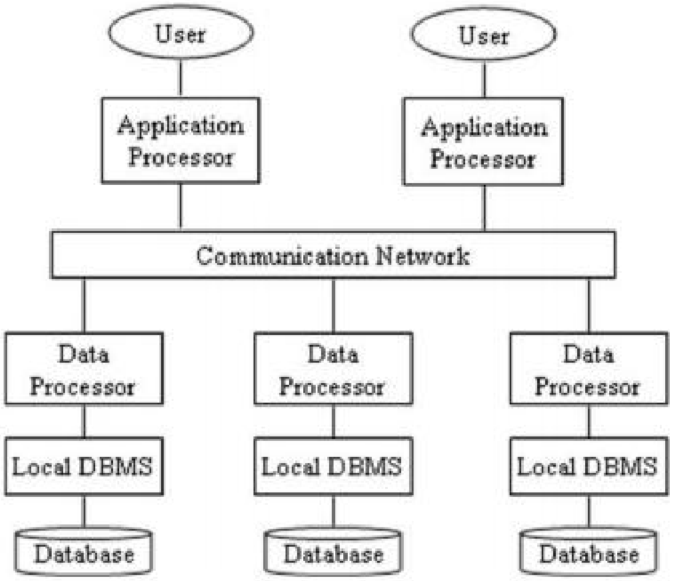
In the bottom–up software development approach, we utilize the database management system functionality of the underlying S-DBEs (usually CDBEs) to develop software capabilities for distributed concurrency control, distributed fault tolerance, distributed commit, distributed query optimization, and distributed semantic integrity control. Any additional software functionality that we need to develop must use, or at least interface with, the capabilities that the underlying S-DBEs provide. In order to accommodate this, the bottom–up architecture typically uses the local DBMS software to manage the local data. The DP for each S-DBE interfaces with the LDBMS software to run the local transactions that were delegated to it by the AP.



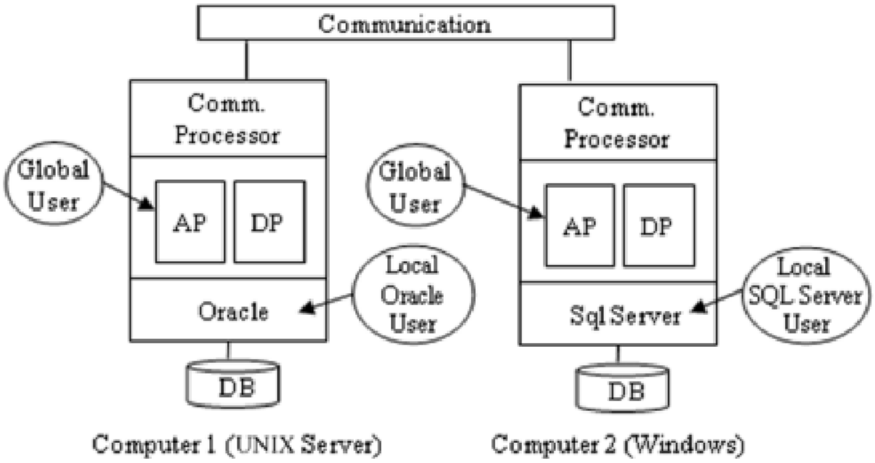
Bottom–up information architecture design.

DEPLOYMENT OF DDBE SOFTWARE

Once written, the software components of a DDBE need to be deployed across the different computer systems within our environment. These computer systems may run different operating systems and support different DBMS software. In order to deploy a software component on one of these target computer systems, we must compile, link, and install it using the facilities provided by the underlying platform. In a well-designed architecture, we can reuse the same source code for several different component implementations and deployments (ideally, across the entire set of target operating systems and DBMSs in our environment). Figure depicts an example deployment, where a CP and an AP have been deployed to both a Windows system and a UNIX-based system. In this example, since the CP has been deployed on both systems, users on either system can access the information contained within the other system (from both the Oracle and SQL Server databases collectively).

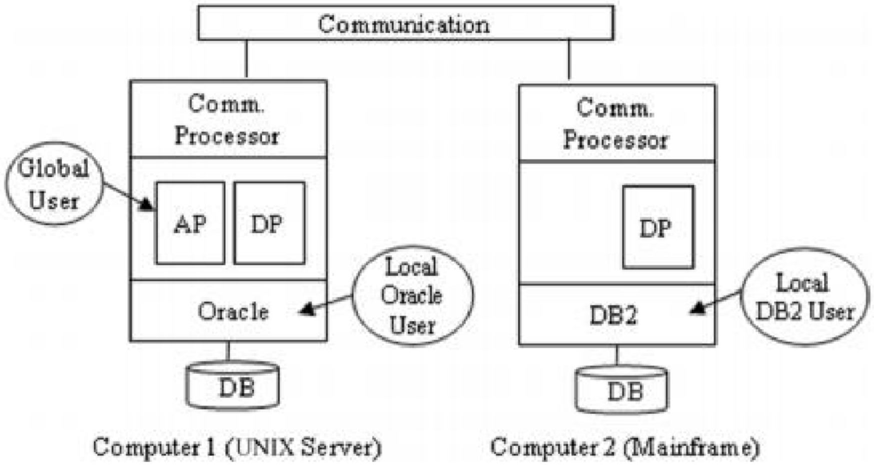


High-level DDBE software architecture for a bottom–up system.



Example of packaging software.

Figure depicts a different deployment, for a different environment, where a DP has been deployed to a mainframe system, but there is no AP deployed within that system. In this case, the users who login to the mainframe do not have access to the data in the Oracle database, but users on the UNIX machine do have access to both the local data within the Oracle database and the data within the DB2 database deployed on the mainframe.



Another example of packaging software.

POSSIBLE QUESTIONS

PART B

(EACH QUESTION CARRIES SIX MARKS)

1. Explain in detail about DDBE Security.
2. Describe in detail about MDBS architecture and classifications.
3. Expound in detail about Cryptography.
4. How to develop a DDBE? Explain the approaches.
5. How to secure data in DDBE? Explain.
6. How to deploy a DDBE software? Explain.
7. Discuss in detail about traditional DDBE architecture.
8. Explain in detail about the cryptography techniques.
9. How to classify the traditional DDBMS architecture? Explain.
10. Describe in detail about the approaches for securing data in DDBE.

PART C

(EACH QUESTION CARRIES TEN MARKS)

1. Discuss in detail about Cryptography.
2. Discuss in detail about terminologies of failure and commit protocols.
3. Discuss in detail about Query processing in centralized systems.
4. Explain i)Vertical Fragmentation ii)Horizontal Fragmentation in detail.
5. Discuss in detail about DBE taxonomy.

KAHE

KARPAGAM ACADEMY OF HIGHER EDUCATION
DEPARTMENT OF COMPUTER APPLICATIONS
DISTRIBUTED DATABASE MANAGEMENT SYSTEM

SUBJECT CODE:17CAP405D **CLASS : II MCA** **SEMESTER: IV**

Unit-5

Questions	Op1	Op2	Op3	Op4	Op5	Op6	Answer
__________ when it is first initiated and does not permit it to proceed if it may cause a deadlock	fragmentation	transaction	deadlock	control			transaction
______________ is the most popular and best studied method	detection	resolution	prevention	deadlock detection and resolution			deadlock detection and resolution
In which approach one site is designated as the deadlock detector for the entire system	deadlock detection	deadlock resolution	centralized deadlock detection	deadlock prevention			centralized deadlock detection
A correctness criterion called________	semantic relative atomicity	atomicity	DBMS	Transaction			semantic relative atomicity
open nested transactions are even more relaxed then their closed nested counter parts are called ______	analog	anarchic	nested transaction	serializability			anarchic
________ state of a system can be defined as the response that system gives to an external stimulus	Internal	reliability	external	aviliabiltiy			external
A permanent fault is also called________	failure	fault	hardfault	erroneous			failure
______ is the expected time between subsequent failures in a system with repair	MTTR	MTBF	MTBR	MTFB			MTBF
______refers to system design approach which recognizes that faults will occur	fault tolerance	fault prevention	fault avoidence	fault detection			fault tolerance
_________ aims at ensuring that the implemented system will not contain any fault	fault tolerance	fault prevention	fault avoidence	fault detection			fault prevention
The terms fault prevention and _____ are used interchangebly	fault tolerance	fault prevention	fault avoidence	fault detection			fault avoidence
more efficient function distribution in which application programs run on workstations is called______	application servers	data base server	parallel database system	database			application servers
Database function are handled by dedicated computer is called_____	application servers	data base server	parallel database system	database			data base server
_________ is used to run the user interface	application server	data base server	parallel server	client server			client server

Distributed database technology can be naturally revised and extended to implement_______	application server	data base server	parallel database system	client server			parallel database system
The main computer executing the application programs was termed the ____	Host computer	data base computer	back end computer	server			Host computer
The dedicated computer was called________	database machine	database computer	back end computer	centralized system			database machine
Data base server fits naturally in a client_server or distributed enviornment has an advantages of_______	database server	application server	parallel server	client server			database server
Any workstation could access the data at any database server through either the_______	local network	local access	local network and access	global access			local network and access
The value of ____ used to provide efficient database management	parallel system	client system	server system	remote system			parallel system
The parallel database system support the _________interface	database function	client server	database function and client server	master slave			database function and client server
______plays the role of a transaction monitor	session manager	request manager	data manager	system manager			session manager
Request manager receives client requests related to query____________	compilation & execution	compilation & run	run & error	data & directory			compilation & execution
Which one provides the lowlevel functions needed to run compiled queries in parallel	session manager	request manager	data manager	system manager			data manager
Parallel system architectures range between two extremes the ______________ architectures	shared_memory	shared nothing	shared memory and shared nothing	shared everything			shared memory and shared nothing
shared _memory has two strong advantages they are______	meta information	control information	load behaviour	simplicity & load behaviour			simplicity & load behaviour
shared memory has a problem of _______	cost	limited extensibility	low aviliabity	consistency			limited extensibility
shared disk suffer from_________ problem	higher complexity	potential	higher complexity and potential	simplicity			higher complexity and potential
In _______ each processor has exclusive access to its mainmemory and disk units	shared disk	shared memory	shared nothing	shared			shared nothing
Hierarchical architecture is a combination of ___________	shared nothing	shared disk	shared memory	shared nothing and memory			shared nothing and memory
Hierarchical architecture is also called ______	cluster	shared	disk	memory			cluster
The term declustering is also used to mean______	partitioning	partition	hash	range			partitioning

_______ is the simplest strategy, it ensures uniform data distribution	round robin partitioning	hash partitioning	range partitioning	hash indexing			round robin partitioning
____ applies a hash function to some attributes which yeilds the partition number	round robin partitioning	hash partitioning	range partitioning	hash indexing			hash partitioning
_______ has distributed tuples based on the value intervals of some attributes	round robin partitioning	hash partitioning	range partitioning	hash indexing			range partitioning
_____ enables the parallel execution of multiple queries generated by concurrent transactions	Inter query	inter operator	intra operation	parallelism			Inter query
_________ are used to decrease response time	Inter query	inter operator parallelism	inter query and parallelism	parallelism			inter query and parallelism
Intra_operator parallelism is based on the decomposition of one operator in a set of independent sub operators called ?	operator instance	inter operator	inter query	query parallelism			operator instance
______ has several operators with a producer_consumer link are executed in parallel	materialized	pipeline parallelism	independent parallelism	processor			pipeline parallelism
_____ Is the set of alternative execution plan to represent the input query	cost model	search strategy	search space	annotations			search space
Operator trees are enriched with_______	cost model	search strategy	search space	annotations			search strategy
Load balancing problem can appear with intra_operator parallelism namley	dataskew	attribute value skew	selectivity sekew	redistribution skew			dataskew
_____ explores the search space and selects the best plan	costmodel	search strategy	search space	annotations			search strategy
_______is the smallest unit of sequential processing that cannot be further partitioned	activation	redistribution sekew	selectivity sekew	product sekew			activation
object represents a ________in the system that is being modeled	real entity	object identity	identical	value			real entity
An ______ datatype is a template for all objects of that type	primary	secondary	primitive	abstract			abstract
The composite object relationship between types can be represented by______	class	collections	composition	subtyping			composition
collection is similar to a_______ in that it groups object	collection	composition	subtyping	class			class
subtyping is based on the ____ relationship among types	specialization	collection	composition	subtyping			specialization
The data allocation problem for object database involves allocation of both________	methods	classes	methods and classes	functions			methods and classes
How many types of client /server architectures have been proposed	2	1	4	5			2
OID represents the______	POID	AID	object identifier management	LOID			object identifier management

The process of converting a disk version of the pointer to a methods version of a pointer is known as _____	pointer Swizzling	Loid mapping	Loid generation	surrogates			pointer Swizzling
A common way of tracking objects is to leave	pointer Swizzling	Loid mapping	Loid generation	surrogates			surrogates
____ object are currently involved in an activity in response to an invocation or a message	ready	active	waiting	suspended			active
_____ object are not currently involved	ready	active	waiting	suspended			ready
______object are temporarily unavailable for invocation	ready	active	waiting	suspended			suspended
Relational query lanaguage operate on very simple type systems consisting of a single type_____	relation	physical storage	path expression	query processing			relation
_______is an alternative general technique to represent and compute path expressions	access support relation	set matching	path indexes	access suppot			access support relation
__________ relation that has been defined to determine serializable histories	access support relation	set matching	path indexes	access suppot			set matching

Reg. No..................................

[12CAP404D]

KARPAGAM UNIVERSITY

(Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2012 onwards)

MCA DEGREE EXAMINATION, APRIL 2014

Fourth Semester

COMPUTER APPLICATIONS

DISTRIBUTED DATABASE MANAGEMENT SYSTEM

Time: 3 hours Maximum : 100 marks

PART – A (15 x 2 = 30 Marks)

Answer ALL the Questions

1. Write short note on traditional file processing.
2. Define DDBS environment.
3. What is a Autonomy? Name the dimensions of autonomy.
4. Define top-down design process?
5. What is a allocation model?
6. How are views managed in semantic data control?
7. What do you mean by equivalent distributed execution strategies?
8. Give an example of operation tree.
9. Write about the semijoin-based algorithms.
10. What is meant by transaction model?
11. Define logging interface.
12. List down all the actions of 3PC protocol.
13. Write about database server approach and distributed database servers through diagrams.
14. Give an example for abstract data types.
15. Differentiate between Horizontal and Vertical class partitioning.

PART B (5 X 14= 70 Marks)

Answer ALL the Questions

16. a) Describe about distributed data processing?

Or

b) Explain peer-to-peer reference architecture with a neat diagram.

17. a) Explain the alternative design strategies in detail.

Or

b) List and explain the differences between three types of fragmentation.

18. a) Discuss about query processing system with examples.

Or

b) Explain distributed query optimization algorithm in detail.

19. a) Elaborate the properties of transactions.

Or

b) Discuss about networking partitioning with examples.

20. a) Explain the following with the help of suitable diagrams:

i) data placement,
ii) different partitioning schemes
iii) query parallelism.

Or

b) Describe about the transaction management through examples.

Reg. No..................................

[14CAP404D]

KARPAGAM UNIVERSITY

Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2014 onwards)

MCA DEGREE EXAMINATION, APRIL 2016

Fourth Semester

COMPUTER APPLICATIONS

DISTRIBUTED DATABASE MANAGEMENT SYSTEM

Time: 3 hours

Maximum : 60 marks

PART – A (20 x 1 = 20 Marks) (30 Minutes)
(Question Nos. 1 to 20 Online Examinations)

PART B (5 x 8 = 40 Marks) (2 ½ Hours)
Answer ALL the Questions

21. a. Explain the technical problems that needs to be addressed to realize the full potential of DDBMS.

Or

b. Explain MDBS architecture.

22. a. Explain the information required to carry out horizontal fragmentation.

Or

b. Explain how the allocation of resources across the nodes is handled in Distributed Database Design

23. a. Explain how cost model is worked out to optimize queries in DDBMS

Or

b. Explain the query optimization techniques for Centralized systems

24. a. Explain the timestamp based concurrency control algorithms.

Or

b. Explain the optimistic concurrency control algorithms:

25. a. Discuss the issues to be addressed in processing Object DBMS.

Or

b. Discuss the issues related to Object storage.

Reg. No....................................

[15CAP405D]

KARPAGAM UNIVERSITY

Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2015 onwards)

MCA DEGREE EXAMINATION, APRIL 2017

Fourth Semester

COMPUTER APPLICATIONS

DISTRIBUTED DATABASE MANAGEMENT SYSTEM

Time: 3 hours

Maximum : 60 marks

PART – A (20 x 1 = 20 Marks) (30 Minutes)
(Question Nos. 1 to 20 Online Examinations)

PART B (5 x 6 = 30 Marks)
Answer ALL the Questions

21. a. Discuss in detail about Database Concepts.

Or

b. Describe about Services, Components, Subsystems and Sites in detail.

22. a. Explain about Data Distribution alternatives in detail.

Or

b. Discuss about i)Vertical Fragmentation ii)Horizontal Fragmentation in detail.

23. a. Discuss in detail about Query Optimization.

Or

b. Describe in detail about heterogeneous Database Systems.

24. a. Explain in detail about deadlock handling in DDBS.

Or

b. Discuss in detail about replication control.

25. a. Explain in detail about DDBE Security.

Or

b. Describe in detail about MDBS architecture and classifications.

PART C (1 x 10 = 10 Marks)
(Compulsory)

26. Discuss in detail about DBE taxonomy.
