Rai Technology University

ENGINEERING MINDS

DATABASE MANAGEMENT SYSTEM

Subject: DATABASE MANAGEMENT SYSTEM **Credits: 4**

SYLLABUS

Database System Concept
Evolution, Concepts of redundancy and dependence in flat file systems, Data Models, ER Diagram

Database Technologies
Relational Model, Relational Algebra part -1, Relational Algebra part –II

Normalization for Relational Databases
Informal Design Guidelines for Relational Schemas Functional Dependencies , Normal Forms based on Primary Keys, Second and Third Normal Forms Boyce-Codd Normal Form Multi-valued Dependencies and Fourth Normal Forms Join Dependencies and Fifth Normal Forms

Commercial query language SQL
Data Manipulation - Data Retrieval Features, Simple queries, Queries Involving More than One Relations, Using Built-in Functions, Using the GROUP BY and HAVING Clauses, Advantages of SQL

Client- Server Architecture
Overview of Client-Server Architecture, two and Three Tier Architecture, Multi-Tier Application Design, Three Tier VS Muti tier

Backup and Recovery
Backup and recovery, Use of log files, Concept of Transcaion and log, Recovery and buffer Management

Database Security
Introduction to Database Security Issues, Security mechanism, Database Users and Schemas, Privileges

Suggested Reading:

1. A Silberschatz, H Korth, S Sudarshan, “Database System and Concepts”, fifth Edition McGraw Hill
2. Rob, Coronel, Database Systems, Seventh Edition, Cengage Learning
3. Fundamentals of Database Systems – Elmasri and Navathe, 5th Edition, Addison-Wesley
4. Database Management Systems – Raghu Ramakrishnan and Johannes Gehrke – 3rd Edition, McGraw-Hill,

COURSE OVERVIEW

During the machine age, the measure of power was heavy typified by cannon, locomotives and rolling mills. In the information age, the measure of power is the depth, timeliness and accessibility of knowledge. Communication bandwidth has become more crucial than shop floor capacity. With out the ability to communicate by telephone, e-mail or fax, an organization is deaf and dumb. Without access to the **Database,** an organization is blind. Because every firm is now directly or indirectly reliant on computer software, every decision point in the enterprise is an interface point between workers and information systems.

In an information-intensive business environment, knowledge is power. Competitiveness pivots on how effectively information systems enable management and staff to gain advantages. The important requirements for commercial power is the ability to transform operational data into tactical information, and ultimately into strategic knowledge. The main resource that fuels this power is the **corporate database.**

This course introduces the fundamental concepts necessary for designing, using and implementing the database systems applications. This course assumes no previous knowledge of databases or database technology and the concepts are built from ground up- basic concepts to advanced techniques and technologies.

- After this course a student will be able to
- Describe the components of DBMS
- Define a DBMS strategy for a given application requirement
- Will be able to design, create, and modify databases.
- Could apply the techniques to tune database performance.

DATABASE MANAGEMENT SYSTEM

CONTENT

DATABASE MANAGEMENT SYSTEM

CONTENT

LESSON 1
EVOLUTION

Evolution

Hi! Welcome to the fascinating world of DBMS. The whole Course will deal with the most widely used software system in the modern world, **The Database Management System.** Here in this lecture we are going to discuss about the evolution of DBMS and how it nurtured in to the most wanted software. Let's begin to devise.

The boundaries that have traditionally existed between DBMS and other data sources are increasingly blurring, and there is a great need for an information integration solution that provides a unified view of all of these services. This article proposes a platform that extends a federated database architecture to support both relational and XML as first class data models, and tightly integrates content management services, workflow, messaging, analytics, and other enterprise application services.

A Brief Introduction

- First, DBMSs have proven to be hugely successful in managing the information explosion that occurred in traditional business applications over the past 30 years. DBMSs deal quite naturally with the storage, retrieval, transformation, scalability, reliability, and availability challenges associated with robust data management.
- Secondly, the database industry has shown that it can adapt quickly to accommodate the diversity of data and access patterns introduced by e-business applications over the past 6 years. For example, most enterprise-strength DBMSs have built-in object-relational support, XML capabilities, and support for federated access to external data sources.
- Thirdly, there is a huge worldwide investment in DBMS technology today, including databases, supporting tools, application development environments, and skilled administrators and developers. A platform that exploits and enhances the DBMS architecture at all levels is in the best position to provide robust end-to-end information integration.

This paper is organized as follows

1. We briefly review the evolution of the DBMS architecture.
2. We provide a real-world scenario that illustrates the scope of the information integration problem and sketches out the requirements for a technology platform.
3. We formally call out the requirements for a technology platform.
4. We present a model for an information integration platform that satisfies these requirements and provides an end-to-end solution to the integration problem as the next evolutionary step of the DBMS architecture.

How did database software evolve?

In the early days of computing, computers were mainly used for solving numerical problems. Even in those days, it was observed that some of the tasks were common in many problems that were being solved. It therefore was considered desirable to build special subroutines which perform frequently occurring computing tasks such as computing Sin(x). This lead to the development of mathematical subroutine libraries that are now considered integral part of any computer system.

By the late fifties storage, maintenance and retrieval of non-numeric data had become very important. Again, it was observed that some of the data processing tasks occurred quite frequently e.g. sorting. This lead to the development of *generalized* routines for some of the most common and frequently used tasks.

A general routine by its nature tends to be somewhat less efficient than a routine designed for a specific problem. In the late fifties and early sixties, when the hardware costs were high, use of general routines was not very popular since it became a matter of trade-off between hardware and software costs. In the last two decades, hardware costs have gone down dramatically while the cost of building software has gone up. It is no more a trade-off between hardware and software costs since hardware is relatively cheap and building reliable software is expensive.

Database management systems evolved from generalized routines for file processing as the users demanded more extensive and more flexible facilities for managing data. Database technology has undergone major changes during the 1970's. An interesting history of database systems is presented by Fry and Sibley (1976).

Figure 1 captures the evolution of relational database technology. Relational databases were born out of a need to store, manipulate and manage the integrity of large volumes of data. In the 1960s, network and hierarchical systems such as [CODASYL] and IMS™ were the state-of-the -art technology for automated banking, accounting, and order processing systems enabled by the introduction of commercial mainframe computers. While these systems provided a good basis for the early systems, their basic architecture mixed the physical manipulation of data with its logical manipulation. When the physical location of data changed, such as from one area of a disk to another, applications had to be updated to reference the new location.

A revolutionary paper by Codd in 1970 [CODD] and its commercial implementations changed all that. Codd's relational model introduced the notion of *data independence*, which separated the physical representation of data from the logical representation presented to applications. Data could be moved from one part of the disk to another or stored in a different

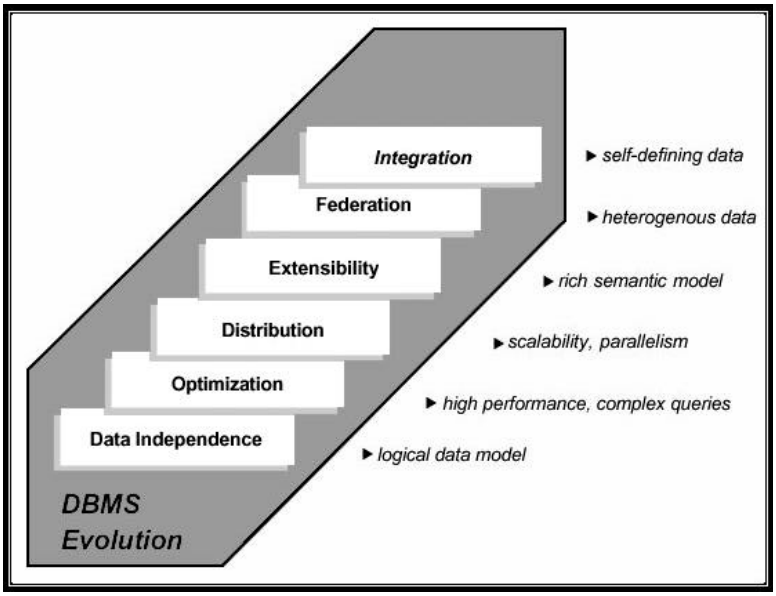
format without causing applications to be rewritten. Application developers were freed from the tedious physical details of data manipulation, and could focus instead on the logical manipulation of data in the context of their specific application.

Not only did the relational model ease the burden of application developers, but it also caused a paradigm shift in the data management industry. The separation between *what* and *how* data is retrieved provided an architecture by which the new database vendors could improve and innovate their products. [SQL] became the standard language for describing what data should be retrieved. New storage schemes, access strategies, and indexing algorithms were developed to speed up how data was stored and retrieved from disk, and advances in concurrency control, logging, and recovery mechanisms further improved data integrity guarantees [GRAY][LIND][ARIES]. Cost-based *optimization*

Techniques [OPT] completed the transition from databases acting as an abstract data management layer to being high-performance, high-volume query processing engines.

As companies globalized and as their data quickly became distributed among their national and international offices, the boundaries of DBMS technology were tested again. Distributed systems such as [R*] and [TANDEM] showed that the basic DBMS architecture could easily be exploited to manage large volumes of *distributed* data. Distributed data led to the introduction of new parallel query processing techniques [PARA], demonstrating the scalability of the DBMS as a high-performance, high-volume query processing engine.

Figure 1. Evolution of DBMS architecture



The lessons learned in extending the DBMS with distributed and parallel algorithms also led to advances in *extensibility*, whereby the monolithic DBMS architecture was deplumed with plug-and-play components [STARBURST]. Such an architecture enabled new abstract data types, access strategies and indexing schemes to be easily introduced as new business needs arose. Database vendors later made these hooks publicly available to customers as Oracle data cartridges, Informix® DataBlades®, and DB2® Extenders™.

Throughout the 1980s, the database market matured and companies attempted to standardize on a single database vendor. However, the reality of doing business generally made such a strategy unrealistic. From independent departmental buying decision to mergers and acquisitions, the scenario of multiple database products and other management systems in a single IT shop became the norm rather than the exception. Businesses sought a way to streamline the administrative and development costs associated with such a heterogeneous environment, and the database industry responded with *federation*. Federated databases [FED] provided a powerful and flexible means for transparent access to heterogeneous, distributed data sources.

We are now in a new revolutionary period enabled by the Internet and fueled by the e-business explosion. Over the past six years, Java™ and XML have become the vehicles for portable code and portable data. To adapt, database vendors have been able to draw on earlier advances in database extensibility and abstract data types to quickly provide object-relational data models [OR], mechanisms to store and retrieve relational data as XML documents [XTABLES] , and XML extensions to SQL [SQLX].

The ease with which complex Internet-based applications can be developed and deployed has dramatically accelerated the pace of automating business processes. The premise of our paper is that the challenge facing businesses today is *information integration*. Enterprise applications require interaction not only with databases, but also content management systems, data warehouses, workflow systems, and other enterprise applications that have developed on a parallel course with relational databases. In the next section, we illustrate the information integration challenge using a scenario drawn from a real-world problem.

Scenario

To meet the needs of its high-end customers and manage high-profile accounts, a financial services company would like to develop a system to automate the process of managing, augmenting and distributing research information as quickly as possible. The company subscribes to several commercial research publications that send data in the Research Information Markup Language (RIXML), an XML vocabulary that combines investment research with a standard format to describe the report's meta data [RIXML]. Reports may be delivered via a variety of mechanisms, such as real-time message feeds, e-mail distribution lists, web downloads and CD ROMs.

Figure 2. Financial services scenario

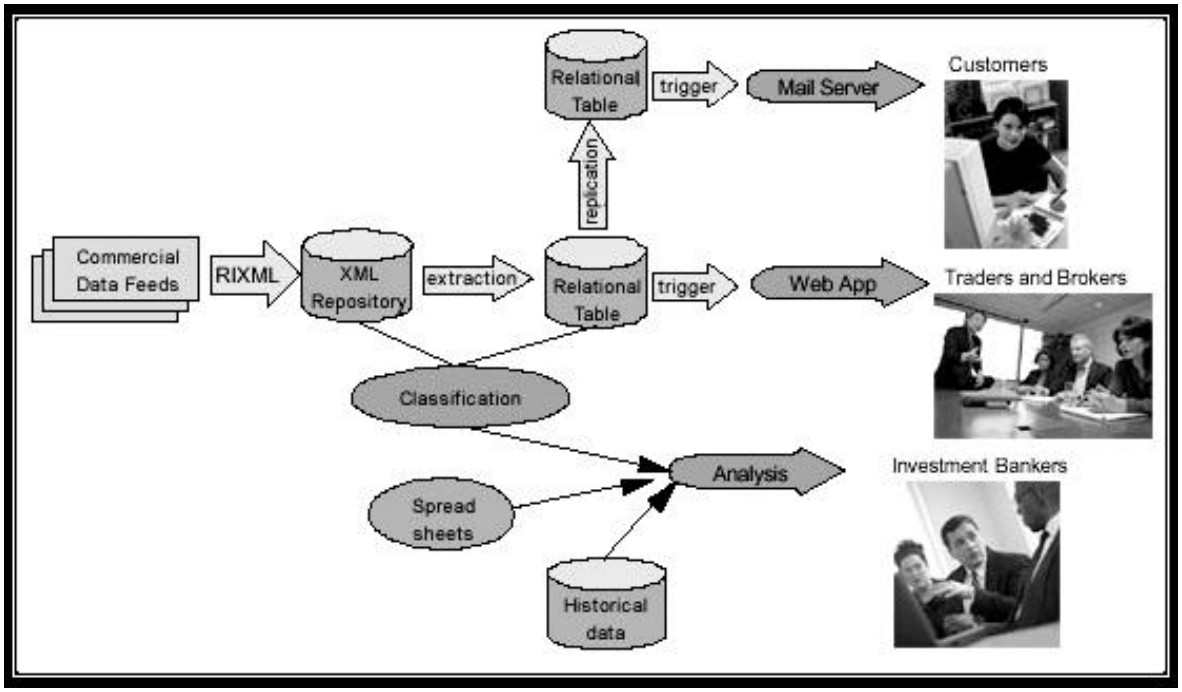


Figure 2 shows how such research information flows through the company.

1. When a research report is received, it is archived in its native XML format.
2. Next, important Meta data such as company name, stock price, earnings estimates, etc., is extracted from the document and stored in relational tables to make it available for real-time and deep analysis.
3. As an example of real-time analysis, the relational table updates may result in database triggers being fired to detect and recommend changes in buy/sell/hold positions, which are quickly sent off to equity and bond traders and brokers. Timeliness is of the essence to this audience and so the information is immediately replicated across multiple sites. The triggers also initiate e-mail notifications to key customers.
4. As an example of deep-analysis, the original document and its extracted Meta data are more thoroughly analyzed, looking for such keywords as "merger", "acquisition" or "bankruptcy" to categorize and summarize the content. The summarized information is combined with historical information made available to the company's market research and investment banking departments.
5. These departments combine the summarized information with financial information stored in spread sheetand other documents to perform trend forecasting, and to identify merger and acquisition opportunities.

Requirements

To build the financial services integration system on today's technology, a company must cobble together a host of management systems and applications that do not naturally coexist with each other. DBMSs, content management systems, data mining packages and workflow systems are commercially available, but the company must develop integration software in-house to integrate them. A database management system can handle the structured data, but XML repositories are just now becoming available on the market. Each time a new data source is added or the information must flow to a new target, the customer's home grown solution must be extended.

The financial services example above and others like it show that the boundaries that have traditionally existed between DBMSs, content management systems, mid-tier caches, and data warehouses are increasingly blurring, and there is a great need for a platform that provides a unified view of all of these services. We believe that a robust information integration platform must meet the following requirements:

- *Seamless integration of structured, semi-structured, and unstructured data from multiple heterogeneous sources*. Data sources include data storage systems such as databases, file systems, real time data feeds, and image and document repositories, as well as data that is tightly integrated with vertical applications such as SAP or Calypso. There must be strong support for standard meta-data interchange, schema mapping, schema-less processing, and support for standard data interchange formats. The integration platform must support both consolidation, in which data is collected from multiple sources and stored in a central repository, and federation, in which data from multiple autonomous sources is accessed as part of a search, but is not moved into the platform itself. As shown in the financial services example, the platform must also provide transparent transformation support to enable data reuse by multiple applications.
- *Robust support for storing, exchanging, and transforming XML data.* For many enterprise information integration problems, a relational data model is too restrictive to be effectively used to represent semi-structured and unstructured data. It is clear that XML is capable of

representing more diverse data formats than relational, and as a result it has become the lingua franca of enterprise integration. Horizontal standards such as [EBXML][SOAP], etc., provide a language for independent processes to exchange data, and vertical standards such as [RIXML] are designed to handle data exchange for a specific industry. As a result, the technology platform must be XML-aware and optimized for XML at all levels. A native XML store is absolutely necessary, along with efficient algorithms for XML data retrieval. Efficient search requires XML query language support such as [SQLX] and [XQuery].

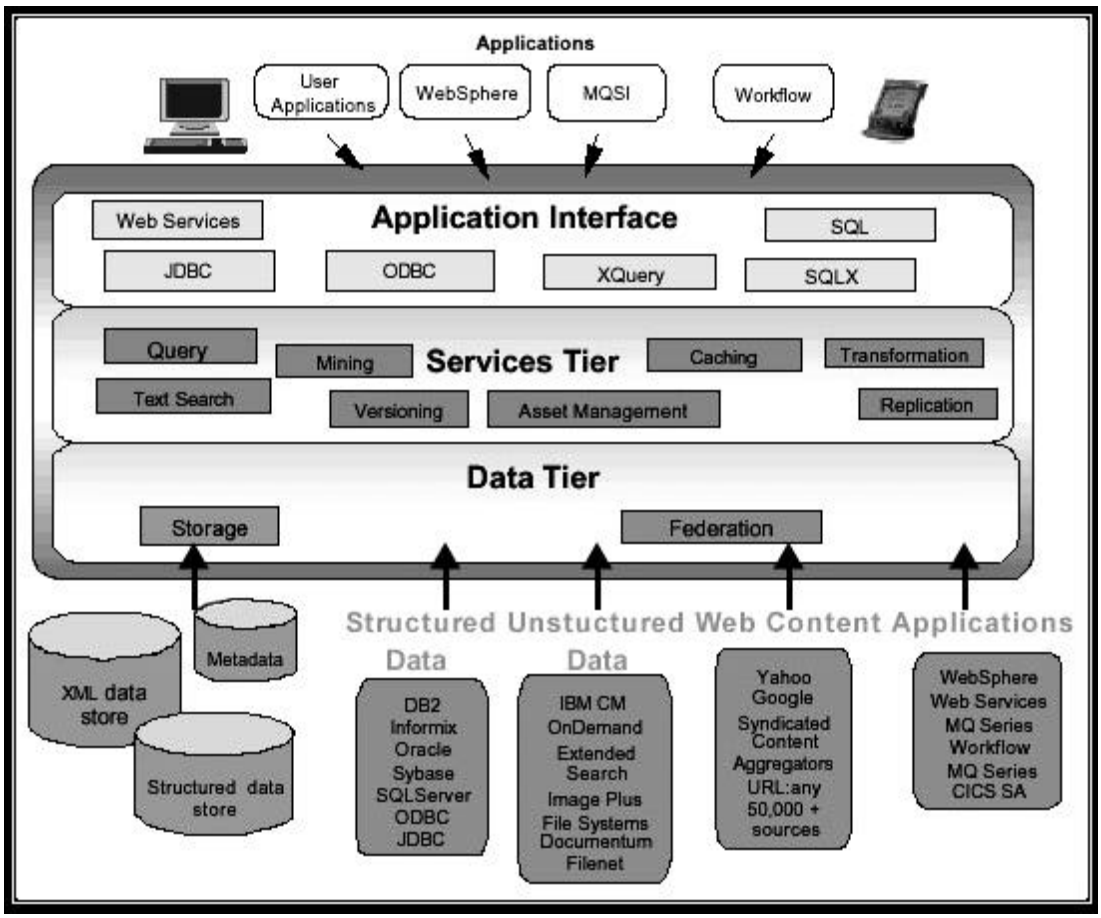
- *Built-in support for advanced search capabilities and analysis over integrated data.* The integration platform must be bilingual. Legacy OLTP and data warehouses speak SQL, yet integration applications have adopted XML. Content management systems employ specialized APIs to manage and query a diverse set of artifacts such as documents, music, images, and videos. An inverse relationship naturally exists between overall system performance and the path length between data transformation operations and the source of the data. As a result, the technology platform must provide efficient access to data regardless of whether it is locally managed or generated by external sources, and whether it is structured or unstructured. Data to be consolidated may require cleansing, transformation and extraction before it can be stored. To support applications that require deep analysis such as the investment banking department in the example above, the platform must provide integrated support for full text search, classification, clustering and summarization algorithms traditionally associated with text search and data mining.
- *Transparently embed information access in business processes*. Enterprises rely heavily on workflow systems to choreograph business processes. The financial services example above is an example of a *macroflow*, a multi-transaction sequence of steps that capture a business process. Each of these steps may in turn be a *microflow*, a sequence of steps executed within a single transaction, such as the insert of extracted data from the research report and the database trigger that fires as a result. A solid integration platform must provide a workflow framework that transparently enables interaction with multiple data sources and applications. Additionally, many business processes are inherently asynchronous. Data sources and applications come up and go down on a regular basis. Data feeds may be interrupted by a hardware or a network failures. Furthermore, end users such as busy stock traders may not want to poll for information, but instead prefer to be notified when events of interest occur. An integration platform must embed messaging, web services and queuing technology to tolerate sporadic availability, latencies and failures in data sources and to enable application asynchrony.
- *Support for standards and multiple platforms*. It goes without saying that an integration platform must run on multiple platforms and support all relevant open standards. The set of data sources and applications generating data will not decrease, and a robust integration platform must be flexible enough to transparently incorporate new sources and applications as they appear. Integration with OLTP systems and data warehouses require strong support for traditional SQL. To be an effective platform for business integration, emerging cross-industry standards such as [SQLX] and [XQuery] as well as standards supporting vertical applications [RIXML].
- *Easy to use and maintain*. Customers today already require integration services and have pieced together in-house solutions to integrate data and applications, and these solutions are costly to develop and maintain. To be effective, a technology platform to replace these in-house solutions must reduce development and administration costs. From both an administrative and development point of view, the technology platform should be as invisible as possible. The platform should include a common data model for all data sources and a consistent programming model. Metadata management and application development tools must be provided to assist administrators, developers, and users in both constructing and exploiting information integration systems.

Architecture

Figure 3 illustrates our proposal for a robust information integration platform.

- The foundation of the platform is the data tier, which provides storage, retrieval and transformation of data from base sources in different formats. We believe that it is crucial to base this foundation layer upon an enhanced full-featured federated DBMS architecture.
- A services tier built on top of the foundation draws from content management systems and enterprise integration applications to provide the infrastructure to transparently embed data access services into enterprise applications and business processes.
- The top tier provides a standards-based programming model and query language to the rich set of services and data provided by the data and services tiers.

Figure 3. An information integration platform



Programming Interface

A foundation based on a DBMS enables full support of traditional programming interfaces such as ODBC and JDBC, easing migration of legacy applications. Such traditional APIs are synchronous and not well-suited to enterprise integration, which is inherently asynchronous. Data sources come and go, multiple applications publish the same services, and complex data retrieval operations may take extended periods of time. To simplify the inherent complexities introduced by such a diverse and data-rich environment, the platform also provides an interface based on Web services ([WSDL] and [SOAP]). In addition, the platform includes asynchronous data retrieval APIs based on message queues and workflow technology [MQ][WORKFLOW]to transparently schedule and manage long running data searches.

Query Language

As with the programming interface, the integration platform enhances standard query languages available for legacy applications with support for XML-enabled applications. [XQuery] is supported as the query language for applications that prefer an XML data model. [SQLX] is supported as the query language for applications that require a mixed data model as well as legacy OLTP-type applications. Regardless of the query language, all applications have access to the federated content enabled by the data tier. An application may issue an XQuery request to transparently join data from the native XML store, a local relational table, and retrieved from an external server. A similar query could be issued in SQLX by another (or the same) application.

Summary

The explosion of information made available to enterprise applications by the broad-based adoption of Internet standards and technologies has introduced a clear need for an information integration platform to help harness that information and make it available to enterprise applications. The challenges for a robust information integration platform are steep. However, the foundation to build such a platform is already on the market. DBMSs have demonstrated over the years a remarkable ability to manage and harness structured data, to scale with business growth, and to quickly adapt to new requirements. We believe that a federated DBMS enhanced with native XML capabilities and tightly coupled enterprise application services, content management services and analytics is the right technology to provide a robust end-to-end solution.

Questions

1. Explain the evolution of the DBMS architecture.
2. Illustrates the scope of the information integration problem and sketches out the requirements for a technology platform.
3. Present a model for an information integration platform that satisfies these requirements and provides an end-to-

end solution to the integration problem as the next evolutionary step of the DBMS architecture.

Selected Bibliography

- [ARIES] C. Mohan, et al.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging., TODS 17(1): 94-162 (1992).
- [CACHE] C. Mohan: Caching Technologies for Web Applications, A Tutorial at the Conference on Very Large Databases (VLDB), Rome, Italy, 2001.
- [CODASYL] ACM: CODASYL Data Base Task Group April 71 Report, New York, 1971.
- [CODD] E. Codd: A Relational Model of Data for Large Shared Data Banks. ACM 13(6):377-387 (1970).
- [EBXML] http://www.ebxml.org.
- [FED] J. Melton, J. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, K. Zeidenstein: SQL and Management of External Data', SIGMOD Record 30(1):70-77, 2001.
- [GRAY] Gray, et al.: Granularity of Locks and Degrees of Consistency in a Shared Database., IFIP Working Conference on Modelling of Database Management Systems, 1-29, AFIPS Press.
- [INFO] P. Lyman, H. Varian, A. Dunn, A. Strygin, K. Swearingen: How Much Information? at http://www.sims.berkeley.edu/research/projects/how-much-info/.
- [LIND] B. Lindsay, et. al: Notes on Distributed Database Systems. IBM Research Report RJ2571, (1979).

Notes:

LESSON 2
DISADVANTAGES OF FILE PROCESSING SYSTEMS

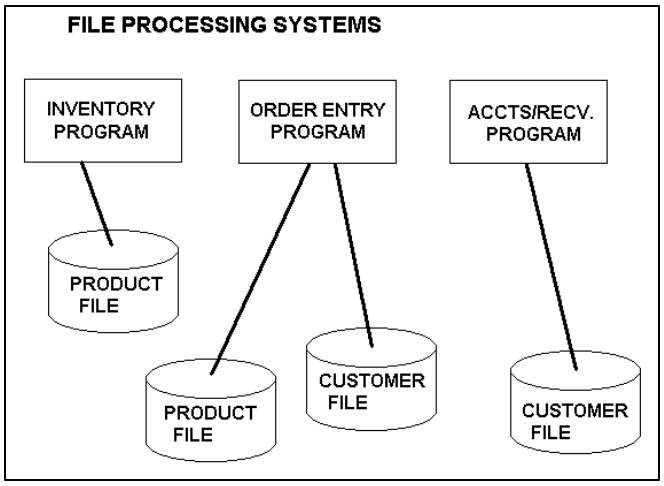
Disadvantages of File processing systems

Hi! The journey has started, today you will learn about the various flaws in the conventional file processing systems. The actual reason for the introduction of DBMS.

Data are stored in files and database in all information systems. Files are collections of similar records. Data storage is build around the corresponding application that uses the files.

File Processing Systems

- Where data are stored to individual files is a very old, but often used approach to system development.
- Each program (system) often had its own unique set of files.



Diagrammatic representation of conventional file systems

Users of file processing systems are almost always at the mercy of the Information Systems department to write programs that manipulate stored data and produce needed information such as printed reports and screen displays.

What is a file, then?

A File is a collection of data about a single entity.

Files are typically designed to meet needs of a particular department or user group.

Files are also typically designed to be part of a particular computer application

Advantages

- Are relatively easy to design and implement since they are normally based on a single application or information system.
- The processing speed is faster than other ways of storing data.

Disadvantages

- Program-data dependence.
- Duplication of data.
- Limited data sharing.
- Lengthy program and system development time.
- Excessive program maintenance when the system changed.
- Duplication of data items in multiple files. Duplication can affect on input, maintenance, storage and possibly data integrity problems.
- Inflexibility and non-scalability. Since the conventional files are designed to support single application, the original file structure cannot support the new requirements.

Today, the trend is in favor of replacing file-based systems and applications with database systems and applications.

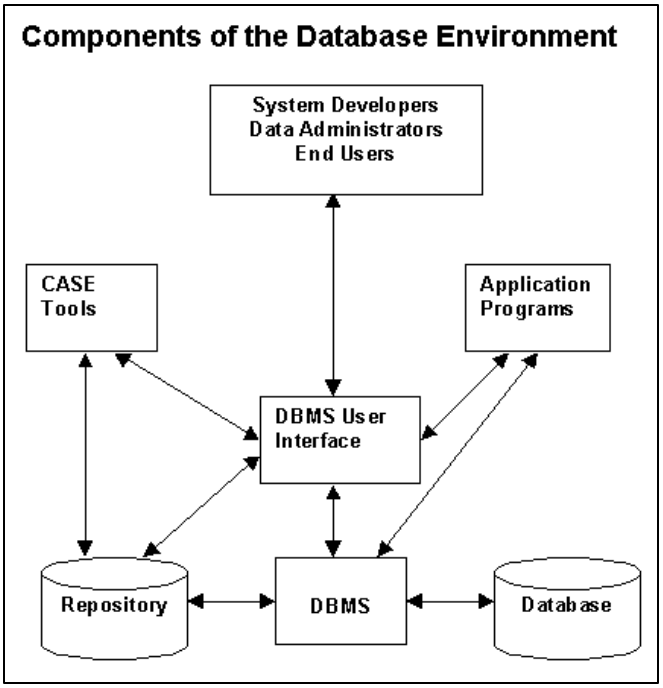
Database Approach

A database is more than a file - it contains information about more than one entity and information about relationships among the entities.

Data about a single entity (e.g., Product, Customer, Customer Order, Department) are each stored to a "table" in the database.

Databases are designed to meet the needs of multiple users and to be used in multiple applications.

One significant development in the more user-friendly relational DBMS products is that users can sometimes get their own answers from the stored data by learning to use data querying methods.



Advantages

- Program-data independence.
- Minimal data redundancy, improved data consistency, enforcement of standards, improved data quality.
- Improved data sharing, improved data accessibility and responsiveness.
- Increased productivity of application development.
- Reduced program maintenance Data can be shared by many applications and systems.
- Data are stored in flexible formats. Data independence. If the data are well designed, the user can access different combinations of same data for query and report purposes.
- Reduce redundancy.

Database Application Size

Personal Computer Database

- Supports a single-user.
- Stand-alone.
- May purchase such an application from a vendor.
- Can't integrate data with other applications.

Workgroup Database

- Example would be a small team using the same set of applications such as in a physician's office.
- Includes numerous workstations and a single server typically.

Department Database

- A functional area (such as production) in a firm.
- Same hardware/software as Workgroup database, but is specialized for the department.

Enterprise Database

- Databases or set of databases to serve an entire organization.
- May be distributed over several different physical locations.
- Requires organizational standards for system development and maintenance.

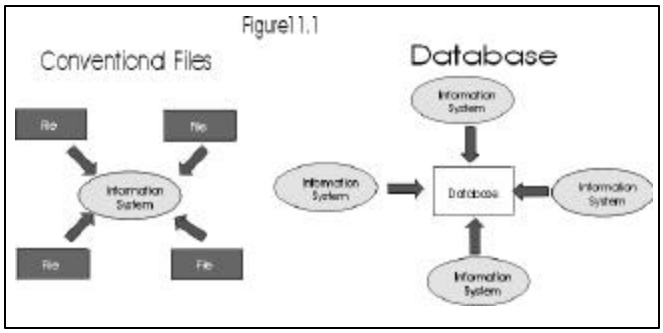


Figure 11.1 Conventional files versus the database

Database Design in Perspective

The focus is on the data from the perspective of the system designer. The output of database design is database schema. Data models were developed during the definition phase. Then, data structures supporting database technology are produced during the database design.

Advantages of Using a DBMS

There are three main features of a database management system that make it attractive to use a DBMS in preference to more conventional software. These features are *centralized data management*, *data independence*, and *systems integration*.

In a database system, the data is managed by the DBMS and all access to the data is through the DBMS providing a key to effective data processing. This contrasts with conventional data processing systems where each application program has direct access to the data it reads or manipulates. In a conventional DP system, an organization is likely to have several files of related data that are processed by several different application programs.

In the conventional data processing application programs, the programs usually are based on a considerable knowledge of data structure and format. In such environment any change of data structure or format would require appropriate changes to the application programs. These changes could be as small as the following:

1. Coding of some field is changed. For example, a null value that was coded as -1 is now coded as -9999.
2. A new field is added to the records.
3. The length of one of the fields is changed. For example, the maximum number of digits in a telephone number field or a postcode field needs to be changed.
4. The field on which the file is sorted is changed.

If some major changes were to be made to the data, the application programs may need to be rewritten. In a database system, the database management system provides the interface between the application programs and the data. When changes are made to the data representation, the metadata maintained by the DBMS is changed but the DBMS continues to provide data to application programs in the previously used way. The DBMS handles the task of transformation of data wherever necessary.

This independence between the programs and the data is called *data independence*. Data independence is important because every time some change needs to be made to the data structure, the programs that were being used before the change would continue to work. To provide a high degree of data independence, a DBMS must include a sophisticated metadata management system.

In DBMS, all files are integrated into one system thus reducing redundancies and making data management more efficient. In addition, DBMS provides centralized control of the operational data. Some of the advantages of data independence, integration and centralized control are:

Redundancies and Inconsistencies can be Reduced

In conventional data systems, an organization often builds a collection of application programs often created by different programmers and requiring different components of the operational data of the organization. The data in conventional data systems is often not centralized. Some applications may require data to be combined from several systems. These several systems could well have data that is redundant as well as

inconsistent (that is, different copies of the same data may have different values). Data inconsistencies are often encountered in everyday life. For example, we have all come across situations when a new address is communicated to an organization that we deal with (e.g. a bank, or Telecom, or a gas company), we find that some of the communications from that organization are received at the new address while others continue to be mailed to the old address. Combining all the data in a database would involve reduction in redundancy as well as inconsistency. It also is likely to reduce the costs for collection, storage and updating of data.

Better Service to the Users

A DBMS is often used to provide better service to the users. In conventional systems, availability of information is often poor since it normally is difficult to obtain information that the existing systems were not designed for. Once several conventional systems are combined to form one centralized data base, the availability of information and its up-to-datedness is likely to improve since the data can now be shared and the DBMS makes it easy to respond to unforeseen information requests.

Centralizing the data in a database also often means that users can obtain new and combined information that would have been impossible to obtain otherwise. Also, use of a DBMS should allow users that do not know programming to interact with the data more easily.

The ability to quickly obtain new and combined information is becoming increasingly important in an environment where various levels of governments are requiring organizations to provide more and more information about their activities. An organization running a conventional data processing system would require new programs to be written (or the information compiled manually) to meet every new demand.

Flexibility of the System is Improved

Changes are often necessary to the contents of data stored in any system. These changes are more easily made in a database than in a conventional system in that these changes do not need to have any impact on application programs.

Cost of Developing and Maintaining Systems is Lower

As noted earlier, it is much easier to respond to unforeseen requests when the data is centralized in a database than when it is stored in conventional file systems. Although the initial cost of setting up of a database can be large, one normally expects the overall cost of setting up a database and developing and maintaining application programs to be lower than for similar service using conventional systems since the productivity of programmers can be substantially higher in using non-procedural languages that have been developed with modern DBMS than using procedural languages.

Standards can be Enforced

Since all access to the database must be through the DBMS, standards are easier to enforce. Standards may relate to the naming of the data, the format of the data, the structure of the data etc.

Security can be Improved

In conventional systems, applications are developed in an ad hoc manner. Often different system of an organization would access different components of the operational data. In such an environment, enforcing security can be quite difficult.

Setting up of a database makes it easier to enforce security restrictions since the data is now centralized. It is easier to control that has access to what parts of the database. However, setting up a database can also make it easier for a determined person to breach security. We will discuss this in the next section.

Integrity can be Improved

Since the data of the organization using a database approach is centralized and would be used by a number of users at a time, it is essential to enforce integrity controls.

Integrity may be compromised in many ways. For example, someone may make a mistake in data input and the salary of a full-time employee may be input as $4,000 rather than $40,000. A student may be shown to have borrowed books but has no enrolment. Salary of a staff member in one department may be coming out of the budget of another department.

If a number of users are allowed to update the same data item at the same time, there is a possibility that the result of the updates is not quite what was intended. For example, in an airline DBMS we could have a situation where the number of bookings made is larger than the capacity of the aircraft that is to be used for the flight. Controls therefore must be introduced to prevent such errors to occur because of concurrent updating activities. However, since all data is stored only once, it is often easier to maintain integrity than in conventional systems.

Enterprise Requirements can be Identified

All enterprises have sections and departments and each of these units often consider the work of their unit as the most important and therefore consider their needs as the most important. Once a database has been set up with entralized control, it will be necessary to identify enterprise requirements and to balance the needs of competing units. It may become necessary to ignore some requests for information if they conflict with higher priority needs of the enterprise.

Data Model must be Developed

Perhaps the most important advantage of setting up a database system is the requirement that an overall data model for the enterprise be built. In conventional systems, it is more likely that files will be designed as needs of particular applications demand. The overall view is often not considered. Building an overall view of the enterprise data, although often an expensive exercise is usually very cost-effective in the long term.

DBMS Architecture

We now discuss a conceptual framework for a DBMS. Several different frameworks have been suggested over the last several years. For example, a framework may be developed based on the functions that the various components of a DBMS must provide to its users. It may also be based on different views of data that are possible within a DBMS. We consider the latter approach.

A commonly used views of data approach is the three-level architecture suggested by ANSI/SPARC (American National Standards Institute/Standards Planning and Requirements Committee). ANSI/SPARC produced an interim report in 1972 followed by a final report in 1977. The reports proposed an architectural framework for databases. Under this approach, a database is considered as containing data about an *enterprise*. The three levels of the architecture are three different views of the data:

1. *External* - individual user view
2. *Conceptual* - community user view
3. *Internal* - physical or storage view

The three level database architecture allows a clear separation of the information meaning (conceptual view) from the external data representation and from the physical data structure layout. A database system that is able to separate the three different views of data is likely to be flexible and adaptable. This flexibility and adaptability is data independence that we have discussed earlier.

We now Briefly Discuss the three Different Views

The external level is the view that the individual user of the database has. This view is often a restricted view of the database and the same database may provide a number of different views for different classes of users. In general, the end users and even the applications programmers are only interested in a subset of the database. For example, a department head may only be interested in the departmental finances and student enrolments but not the library information. The librarian would not be expected to have any interest in the information about academic staff. The payroll office would have no interest in student enrolments.

The conceptual view is the information model of the enterprise and contains the view of the whole enterprise without any concern for the physical implementation. This view is normally more stable than the other two views. In a database, it may be desirable to change the internal view to improve performance while there has been no change in the conceptual view of the database. The conceptual view is the overall community view of the database and it includes all the information that is going to be represented in the database. The conceptual view is defined by the conceptual schema which includes definitions of each of the various types of data.

The internal view is the view about the actual physical storage of data. It tells us what data is stored in the database and how. At least the following aspects are considered at this level:

1. Storage allocation e.g. B-trees, hashing etc.
2. Access paths e.g. specification of primary and secondary keys, indexes and pointers and sequencing.
3. Miscellaneous e.g. data compression and encryption techniques, optimization of the internal structures.

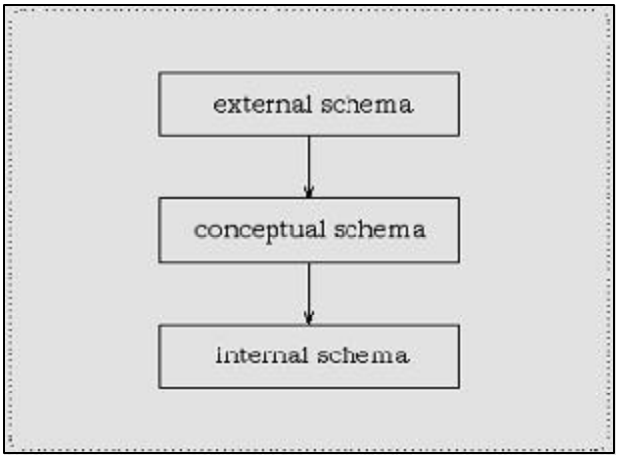
Efficiency considerations are the most important at this level and the data structures are chosen to provide an efficient database. The internal view does not deal with the physical devices directly. Instead it views a physical device as a collection of physical pages and allocates space in terms of logical pages.

The separation of the conceptual view from the internal view enables us to provide a logical description of the database without the need to specify physical structures. This is often called *physical data independence*. Separating the external views from the conceptual view enables us to change the conceptual view without affecting the external views. This separation is sometimes called *logical data independence*.

Assuming the three level view of the database, a number of mappings are needed to enable the users working with one of the external views. For example, the payroll office may have an external view of the database that consists of the following information only:

1. Staff number, name and address.
2. Staff tax information e.g. number of dependents.
3. Staff bank information where salary is deposited.
4. Staff employment status, salary level, leaves information etc.

The conceptual view of the database may contain academic staff, general staff, casual staff etc. A mapping will need to be created where all the staff in the different categories are combined into one category for the payroll office. The conceptual view would include information about each staff's position, the date employment started, full-time or part-time, etc. This will need to be mapped to the salary level for the salary office. Also, if there is some change in the conceptual view, the external view can stay the same if the mapping is changed.



Summary

Now we are coming to the end of this lecture, but before parting we will revise the things. Files are collections of similar records. Data storage is build around the corresponding application that uses the files. Duplication of data items in multiple files. Duplication can affect on input, maintenance, storage and possibly data integrity problems. Inflexibility and non-scalability. Since the conventional files are designed to support single application, the original file structure cannot support the new requirements.

Today, the trend is in favor of replacing file-based systems and applications with database systems and applications.

Questions

1. Explain a file with its advantages and disadvantages
2. Define centralized data management, data independence and systems integration
3. Explain DBMS architecture
4. Explain the 3 different views or architecture of the data

Selected Bibliography

Notes:

LESSON 3
DATA MODELS

Data Models

Hi! In this lecture you are going to learn about the data models used for designing a database.

Before the data available in an enterprise can be put in a DBMS, an overall abstract view of the enterprise data must be developed. The view can then be translated into a form that is acceptable by the DBMS. Although at first sight the task may appear trivial, it is often a very complex process. The complexity of mapping the enterprise data to a database management system can be reduced by dividing the process into two phases. The first phase as noted above involves building an overall view (or model) of the *real world* which is the enterprise (often called the *logical database design*). The objective of the model is to represent, as accurately as possible, the information structures of the enterprise that are of interest. This model is sometimes called the *enterprise conceptual schema* and the process of developing the model may be considered as the requirements analysis of the database development life cycle. This model is then mapped in the second phase to the user schema appropriate for the database system to be used. The logical design process is an abstraction process which captures the meaning of the enterprise data without getting involved in the details of individual data values. Figure 2.1 shows this two step process.

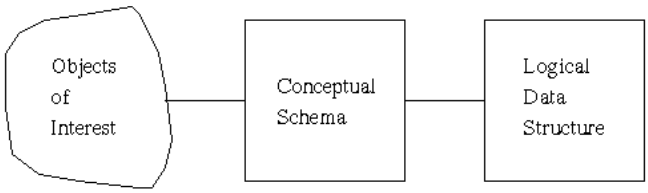


Figure 2.1

The process is somewhat similar to modeling in the physical sciences. In logical database design, similar to modeling in the physical sciences, a model is built to enhance understanding and abstracting details. A model cannot be expected to provide complete knowledge (except for very simple situations) but a good model should provide a reasonable interpretation of the real-life situation. For example, a model of employee data in an enterprise may not be able to capture the knowledge that employees May Adams and John Adams are married to each other. We accept such imperfections of the model since we want to keep the model as simple as possible. It is clearly impossible (and possibly undesirable) to record every available piece of information that is available in an enterprise. We only desire that the meaning captured by a data model should be adequate for the purpose that the data is to be used for.

The person organizing the data to set up the database not only has to model the enterprise but also has to consider the efficiency and constraints of the DBMS although the two-phase process separates those two tasks. Nevertheless, we only consider data models that can be implemented on computers.

When the database is complex, the logical database design phase may be very difficult. A number of techniques are available for modeling data but we will discuss only one such technique that is, we believe, easy to understand. The technique uses a convenient representation that enables the designer to view the data from the point of view of the whole enterprise. This well known technique is called the *entity-relationship model*. We discuss it now.

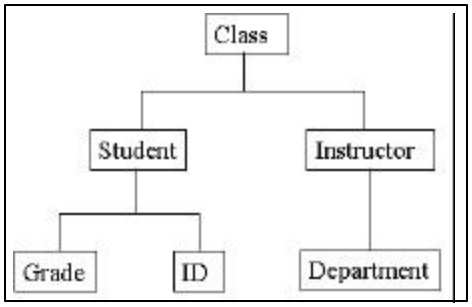
Types of Database Management Systems

A DBMS can take any one of the several approaches to manage data. Each approach constitutes a database model. A data model is a collection of descriptions of data structures and their contained fields, together with the operations or functions that manipulate them. A data model is a comprehensive scheme for describing how data is to be represented for manipulation by humans or computer programs. A thorough representation details the types of data, the topological arrangements of data, spatial and temporal maps onto which data can be projected, and the operations and structures that can be invoked to handle data and its maps. The various Database Models are the following:-

- *Relational* - data model based on tables.
- *Network* - data model based on graphs with records as nodes and relationships between records as edges.
- *Hierarchical* - data model based on trees.
- *Object-Oriented* - data model based on the object-oriented programming paradigm.

Hierarchical Model

In a Hierarchical model you could create links between these record types; the hierarchical model uses Parent Child Relationships. These are a 1: N mapping between record types. This is done by using trees, like set theory used in the relational model, “borrowed” from maths. For example, an organization might store information about an employee, such as name, employee number, department, salary. The organization might also store information about an employee’s children, such as name and date of birth. The employee and children data forms a hierarchy, where the employee data represents the parent segment and the children data represents the child segment. If an employee has three children, then there would be three child segments associated with one employee segment. In a hierarchical database the parent-child relationship is one to many. This restricts a child segment to having only one parent segment. Hierarchical DBMSs were popular from the late 1960s, with the introduction of IBM’s Information Management System (IMS) DBMS, through the 1970s.

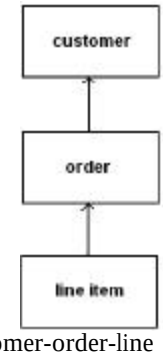


Advantages

- Simplicity
- Data Security and Data Integrity
- Efficiency

Disadvantages

- Implementation Complexity
- Lack of structural independence
- Programming complexity



The figure on the right hand side is a Customer-order-line item database:

There are three data types (record types) in the database: customers, orders, and line items. For each customer, there may be several orders, but for each order, there is just one customer. Likewise, for each order, there may be many line items, but each line item occurs in just one order. (This is the schema for the database.) So, each customer record is the root of a tree, with the orders as children. The children of the orders are the line items. Note: Instead of keeping separate files of Customers, Orders, and Line Items, the DBMS can store orders immediately after customers. If this is done, it can result in very efficient processing.

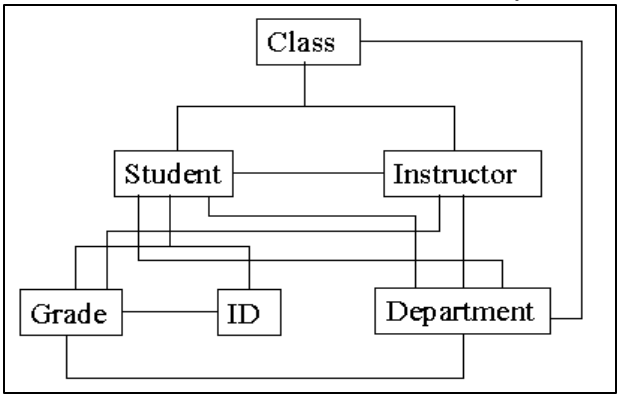
Problem: What if we also want to maintain Product information in the database, and keep track of the orders for each product?

Now there is a relationship between orders and line items (each of which refers to a single product), and between products and line items. We no longer have a tree structure, but a directed graph, in which a node can have more than one parent.

In a hierarchical DBMS, this problem is solved by introducing pointers. All line items for a given product can be linked on a linked list. Line items become "logical children" of products. In an IMS database, there may be logical child pointers, parent pointers, and physical child pointers

Network Data Model

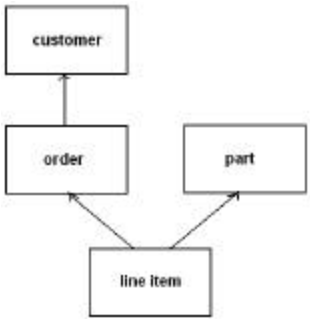
A member record type in the Network Model can have that role in more than one set; hence the multivalent concept is supported. An owner record type can also be a member or owner in another set. The data model is a simple network, and link and intersection record types (called junction records by IDMS) may exist, as well as sets between them . Thus, the complete network of relationships is represented by several pair wise sets; in each set some (one) record type is owner (at the tail of the network arrow) and one or more record types are members (at the head of the relationship arrow). Usually, a set defines a 1:M relationship, although 1:1 is permitted. The CODASYL network model is based on mathematical set theory.



Network Data Model

Advantages

- Conceptual Simplicity
- Ease of data access
- Data Integrity and capability to handle more relationship types
- Data independence
- Database standards
- Disadvantages
- System complexity
- Absence of structural independence



Instead of trees, schemas may be acyclic directed graphs.

In the network model, there are two main abstractions: *records* (record types) and *sets.* A set represents a one-to-many relationship between record types. The database diagrammed above would be implemented using four records (customer, order, part, and line item) and three sets (customer-order, order-line item, and part-line item). This would be written in a *schema* for the database in the network DDL.

Network database systems use linked lists to represent one-to-many relationships. For example, if a customer has several orders, then the customer record will contain a pointer to the head of a linked list containing all of those orders.

The network model allows any number of one-to-many relationships to be represented, but there is still a problem with many-to-many relationships. Consider, for example, a database of students and courses. Each student may be taking several courses. Each course enrolls many students. So the linked list method of implementation breaks down (Why?)

The way this is handled in the network model is to decompose the many-to-many relationship into two one-to-many relationships by introducing an additional record type called an "interesection record". In this case, we would have one intersection record for each instance of a student enrolled in a course.

This gives a somewhat better tool for designing databases. The database can be designed by creating a diagram showing all the record types and the relationships between them. If necessary, intersection record types may be added. (In the hierarchical

model, the designer must explicitly indicate the extra pointer fields needed to represent "out of tree" relationships.)

In general, these products were very successful, and were considered the state of the art throughout the 1970s and 1980s. They are still in use today.

- IMS (hierarchical) IBM
- IDMS (network) Computer Associates
- CODASYL DBMS (network) Oracle

But - there are still some problems.

There is an insufficient level of data abstraction. Database designers and programmers must still be cognizant of the underlying physical structure.

Pointers are embedded in the records themselves. That makes it more difficult to change the logical structure of the database. Processing is "one record at a time". Application programs must "navigate" their way through the database. This leads to complexity in the applications. The result is inflexibility and difficulty of use.

Performing a query to extract information from a database requires writing a new application program. There is no user-oriented query language. Because of the embedded pointers, modifying a schema requires modification of the physical structure of the database, which means rebuilding the database, which is costly.

Relational Model

A database model that organizes data logically in tables. A formal theory of data consisting of three major components: (a) A structural aspect, meaning that data in the database is perceived as tables, and only tables, (b) An integrity aspect, meaning that those tables satisfy certain integrity constraints, and (c) A manipulative aspect, meaning that the tables can be operated upon by means of operators which derive tables from tables. Here each table corresponds to an application entity and each row represents an instance of that entity. (RDBMS - relational database management system) A database based on the relational model was developed by E.F. Codd. A relational database allows the definition of data structures, storage and retrieval operations and integrity constraints. In such a database the data and relations between them are organized in tables. A table is a collection of records and each record in a table contains the same fields.

Properties of Relational Tables:

- Values Are Atomic
- Each Row is Unique
- Column Values Are of the Same Kind
- The Sequence of Columns is Insignificant
- The Sequence of Rows is Insignificant
- Each Column Has a Unique Name

Certain fields may be designated as keys, which mean that searches for specific values of that field will use indexing to speed them up. Often, but not always, the fields will have the same name in both tables. For example, an "orders" table might contain (customer-ID, product-code) pairs and a "products" table might contain (product-code, price) pairs so to calculate a given customer's bill you would sum the prices of all products ordered by that customer by joining on the product-code fields of the two tables. This can be extended to joining multiple tables on multiple fields. Because these relationships are only specified at retrieval time, relational databases are classed as dynamic database management system. The RELATIONAL database model is based on the Relational Algebra.

Advantages

- Structural Independence
- Conceptual Simplicity
- Ease of design, implementation, maintenance and usage.
- Ad hoc query capability
- Disadvantages
- Hardware Overheads
- Ease of design can lead to bad design

The relational model is the most important in today's world, so we will spend most of our time studying it. Some people today question whether the relational model is not too simple, that it is insufficiently rich to express complex data types.

Note: Database theory evolved from file processing. So ideas that were driving programming languages and software engineering were not part of it.

Example A table of data showing this semester's computer science courses

course number	section number	title	room	time	instructor
150	01	Principles of Computer Science	King 221	MWF 10-10:50	Bilar
150	02	Principles of Computer Science	King 135	T 1:30-4:30	Geitz
151	01	Principles of Computer Science	King 243	MWF 9-9:50	Bilar
151	02	Principles of Computer Science	King 201	M 1:30-4:30	Bilar
151	03	Principles of Computer Science	King 201	T 1:30-4:30	Donaldson
210	01	Computer Organization	King 221	MWF 3:30-4:20	Donaldson
280	01	Abstractions and Data Structures	King 221	MWF 11-11:50	Geitz
299	01	Mind and Machine	King 235	W 7-9:30	Borroni
311	01	Database Systems	King 106	MWF 10-10:50	Donaldson
383	01	Theory of Computer Science	King 227	MWF 2:30-3:30	Geitz

Codd's idea was that this is the way that humans normally think of data. It's simpler and more natural than pointer-based hierarchies or networks. But is it as expressive?

Object Oriented Data Models

Object DBMSs add database functionality to object programming languages. They bring much more than persistent storage of programming language objects. Object DBMSs extend the semantics of the C++, Smalltalk and Java object programming languages to provide full-featured database programming capability, while retaining native language compatibility. A major benefit of this approach is the unification of the application and database development into a seamless data model and language environment. As a result, applications require less code, use more natural data modeling, and code bases are easier to maintain. Object developers can write complete database applications with a modest amount of additional effort.

In contrast to a relational DBMS where a complex data structure must be flattened out to fit into tables or joined together from those tables to form the in-memory structure, object DBMSs have no performance overhead to store or retrieve a web or hierarchy of interrelated objects. This one-to-one mapping of object programming language objects to database objects has two benefits over other storage approaches: it provides higher performance management of objects, and it enables better management of the complex interrelationships between objects. This makes object DBMSs better suited to support applications such as financial portfolio risk analysis systems, telecommunications service applications, World Wide Web document structures, design and manufacturing systems, and hospital patient record systems, which have complex relationships between data.

Advantages

- Capability to handle large number of different data types
- Marriage of object-oriented programming and database technology
- Data access
- Disadvantages
- Difficult to maintain
- Not suited for all applications

Some Current And Future Trends

1. Object-oriented Databases

Can we apply object-oriented theory to databases? Some object-oriented commercial systems have been developed, among them O2, Objectivity, and POET. None of these is dominant or particularly successful. There is no standard for O-O databases. One trend is toward "object-relational" systems (like Oracle 8i, 9i). Oracle has added object-oriented features to their existing relational system.

2. Multimedia Data

Traditional DBMSs handled records with fields that were numbers or character strings, a few hundred characters long at most. Now, DBMSs must handle picture and sound files, HTML documents, etc.

3. Data Warehousing and Data Mining

Maintain storage of large amounts of data, often historical or on legacy systems, to support planning and analysis. Searching for trends and other information in existing data.

Bottom line: We expect a DBMS to support:

1. Data definition. (schema)
2. Data manipulation. (queries and updates)
3. Persistence. (long-term storage)
4. Multi-user access.

Summary

A DBMS can take any one of the several approaches to manage data. Each approach constitutes a database model

The various Database Models are the following:-

Relational - Data model based on tables.

Network - Data model based on graphs with records as nodes and relationships between records as edges.

Hierarchical - Data model based on trees. Object-Oriented -Data model based on the object-oriented programming paradigm

Questions

1. Explain the network data model?
2. Explain the Hierarchical data model?
3. Explain relational data model?
4. Explain the new trends in DBMS

References

http://www.tc.cornell.edu/services/edu

http://www.unixspace.com/context/databases.html

Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000

Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld.

Notes:

LESSON 4
E-R MODEL

E-R Model

Hi! Here in this lecture we are going to discuss about the E-R Model.

What is Entity-Relationship Model?

The entity-relationship model is useful because, as we will soon see, it facilitates communication between the database designer and the end user during the requirements analysis. To facilitate such communication the model has to be simple and readable for the database designer as well as the end user. It enables an abstract global view (that is, the *enterprise conceptual schema*) of the enterprise to be developed without any consideration of efficiency of the ultimate database.

The entity-relationship model views an enterprise as being composed of *entities* that have *relationships* between them. To develop a database model based on the E-R technique involves identifying the entities and relationships of interest. These are often displayed in an E-R diagram. Before discussing these, we need to present some definitions.

Entities and Entity Sets

An *entity* is a person, place, or a thing or an object or an event which can be distinctly identified and is of interest. A specific student, for example, John Smith with student number 84002345 or a subject Database Management Systems with subject number CP3010 or an institution called James Cook University are examples of entities. Although entities are often concrete like a company, an entity may be abstract like an idea, concept or convenience, for example, a research project P1234 - Evaluation of Database Systems or a Sydney to Melbourne flight number TN123.

Entities are classified into different *entity sets* (or entity types). An entity set is a set of entity instances or entity occurrences of the same type. For example, all employees of a company may constitute an entity set *employee* and all departments may belong to an entity set *Dept*. An entity may belong to one or more entity sets. For example, some company employees may belong to *employee* as well as other entity sets, for example, *managers*. Entity sets therefore are not always disjoint.

We remind the reader that an entity set or entity type is a collection of entity instances of the same type and must be distinguished from an entity instance.

The real world does not just consist of entities. As noted in the last chapter, a database is a collection of interrelated information about an enterprise. A database therefore consists of a collection of entity sets; it also includes information about relationships between the entity sets. We discuss the relationships now.

Relationships, Roles and Relationship Sets

Relationships are associations or connections that exist between entities and may be looked at as mappings between two or more entity sets. A relation therefore is a subset of the cross products of the entity sets involved in the relationship. The associated entities may be of the same type or of different types. For example, *working - for* is a relationship between an employee and a company. *Supervising* is a relationship between two entities belonging to the same entity set (*employee*).

A *relationship set* R_i is a set of relations of the same type. It may be looked at as a mathematical relation among *n* entities each taken from an entity set, not necessarily distinct:

$$R_i = [e_1, e_2, \ldots, e_n] \mid e_1 \subset E_1, e_2 \subset E_2, \ldots, e_n \subset E_n$$

where each e_i is an entity instance in entity set E_i. Each tuple [e_1, e_2,..., e_n] is a relationship instance. Each entity set in a relationship has a *role* or a function that the entity plays in the relationship. For example, a person entity in a relationship *works-for* has a role of an *employee* while the entity set company has a role of an *employer*.

Often role names are verbs and entity names are nouns which enables one to read the entity and relationships for example as “employee works-for employer”.

Although we allow relationships among any number of entity sets, the most common cases are binary relationships between two entity sets. Most relationships discussed in this chapter are binary. The *degree* of a relationship is the number of entities associated in the relationship. Unary, binary and ternary relationships therefore have degree 1, 2 and 3 respectively.

We consider a binary relation R between two entity types E_1 and E_2. The relationship may be considered as two mappings E_1 -> E_2 and E_2 -> E_1. It is important to consider the constraints on these two mappings. It may be that one object from E_1 may be associated with exactly one object in E_2 or any number of objects in E_1 may be associated with any number of objects in E_2. Often the relationships are classified as one-to-one, one-to-many, or many-to-many. If every entity in E_1 is associated with at most one entity in E_2 and every entity in E_2 is associated with no more than one entity in entity set E_1, the relationship is one-to-one. For example, marriage is a one-to-one relationship (in most countries!) between an entity set *person* to itself. If an entity in E_1 may be associated with any number of entities in E_2, but an entity in E_2 can be associated with at most one entity in E_1, the relationship is called one-to-many. In a many-to-many relationship, an entity instance from one entity set may be related with any number of entity instances in the other entity set. We consider an example of entity sets employees and offices.

1. If for each employee there is at most one office and for each office there is at most one employee, the relationship is one-to-one.
2. If an office may accommodate more than one employee but an employee has at most one office, the relationship between office and employees is now one-to-many.

3. If an office may accommodate more than one staff and a staff member may be assigned more than one office the relationship is many-to-many. For example, an engineer may have one office in the workshop and another in the design office. Also each design office may accommodate more than one staff. The relationship is therefore many-to-many.

These three relationships are shown in Figures 2.2a, 2.2b and 2.2c. Figure 2.2a shows a one-to-one relationship between employees and offices.

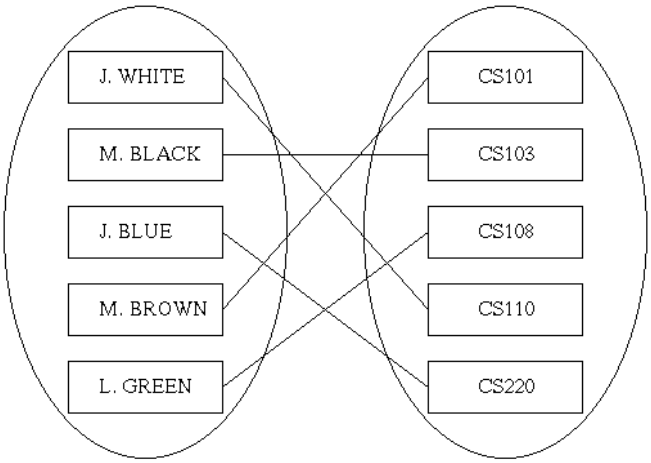


Figure 2.2a. One-to-one relationship Figure 2.2b shows a many-to-one relationship between employees and offices.

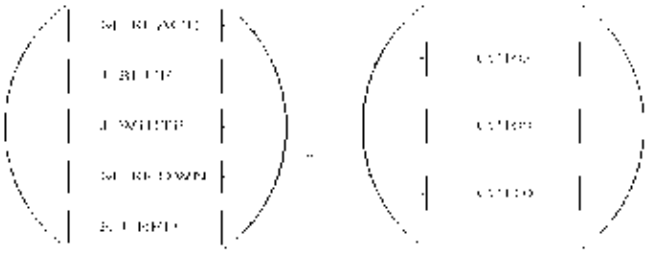


Figure 2.2b. Many-to-one relationship

Figure 2.2c shows a many-to-many relationship between employees and offices.

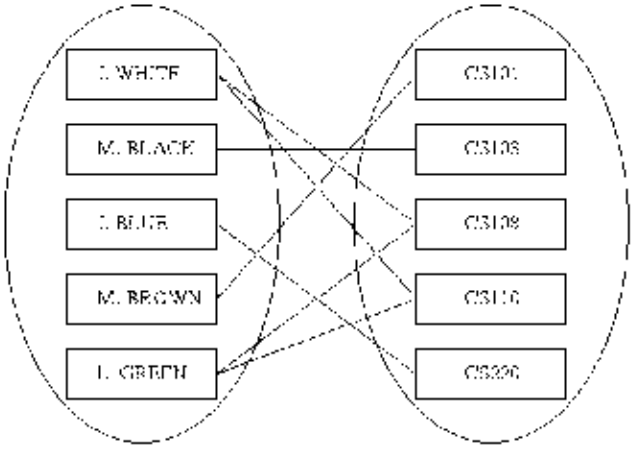


Figure 2.2c. Many-to-many relationship

As noted earlier, relationships are often binary but more than two entity types may participate in a relationship type. For example, a relationship may exist between entity type's *employee*, *project* and *company*.

Attributes, Values and Value Sets

As we have noted earlier, our database model consists of entities and relationships. Since these are the objects of interest, we must have some information about them that is of interest. The information of interest for each object is likely to be

Object name or identifier
object properties
values of the properties
time

Object name or identifier enables us to identify each object uniquely. Each object is described by properties and their values at some given time. Often however we are only interested in the current values of the properties and it is then convenient to drop time from the information being collected.

In some applications however, time is very important. For example, a personnel department would wish to have access to salary history and status history of all employees of the enterprise. A number of ways are available for storing such temporal data. Further details are beyond the scope of these notes.

Each entity instance in an entity set is described by a set of *attributes* that describe their qualities, characteristics or properties that are relevant and the values of the attributes. Relationships may also have relevant information about them. An employee in an *employee* entity set is likely to be described by its attributes like *employee number*, *name*, *date of birth*, etc. A relationship like *enrolment* might have attributes like the *date of enrolment*. A relationship *shipment* between suppliers and parts may have *date of shipment* and *quantity shipped* as attributes.

For each attribute there are a number of legal or permitted values. These set of legal values are sometimes called *value sets* or *domain* of that attribute. For example, employee number may have legal values only between 10000 to 99999 while the year value in date of birth may have admissible values only between year 1900 and 1975. A number of attributes may share a common domain of values. For example, employee number, salary and department number may all be five digit integers.

An attribute may be considered as a function that maps from an entity set or a relationship set into a value set. Sometime an attribute may map into a Cartesian product of value sets. For example, the attribute *name* may map into value sets *first name*, *middle initial* and *last name*. The set of (attribute, data value) pairs, one pair for each attribute, define each entity instance. For example, an entity instance of entity set Employee may be described by

1. (employee number, 101122)
2. (employee name, John Smith)
3. (employee address, 2 Main Street Townsville Q4812)
4. (telephone, 456789)
5. (salary, 44,567)

In this chapter, we will discuss the following entity sets:

1. *EMPLOYEE* - the set of all employees at a company. The attributes of Employees are: name, employee number, address, telephone number, salary.
2. *DEPT* - the set of all departments at the company. The attributes are: department number, department name, supervisor number.
3. *PROJECTS* - the set of all projects at the company. The attributes are: project number, project name, project manager number.

Representing Entities and Relationships

It is essential that we be able to identify each entity instance in every entity set and each relationship instance in every relationship set. Since entities are represented by the values of their attribute set, it is necessary that the set of attribute values be different for each entity instance. Sometimes an artificial attribute may need to be included in the attribute set to simplify entity identification (for example, although each instance of the entity set *student* can be identified by the values of its attributes *student name*, *address* and *date of birth*, it is convenient to include an artificial attribute student number to make identification of each entity instance easier)

A group of attributes (possibly one) used for identifying entities in an entity set is called an *entity key*. For example, for the entity set *student*, *student number* is an entity key and so is (*student name, address, date of birth*). Of course, if k is a key then so must be each superset of k. To reduce the number of possible keys, it is usually required that a key be *minimal* in that no subset of the key should be key. For example, *student name* by itself could not be considered an entity key since two students could have the same name. When several minimal keys exist (such keys are often called *candidate keys*), any semantically meaningful key is chosen as the *entity primary key*.

Similarly each relationship instance in a relationship set needs to be identified. This identification is always based on the primary keys of the entity sets involved in the relationship set. The primary key of a relationship set is the combination of the attributes that form the primary keys of the entity sets involved. In addition to the entity identifiers, the relationship key also (perhaps implicitly) identifies the role that each entity plays in the relationship.

Let *employee number* be the primary key of an entity set EMPLOYEE and *company name* be the primary key of COMPANY. The primary key of relationship set WORKS-FOR is then (*employee number*, *company name*). The role of the two entities is implied by the order in which the two primary keys are specified.

In certain cases, the entities in an entity set cannot be uniquely identified by the values of their own attributes. For example, children of an employee in a company may have names that are not unique since a child of one employee is quite likely to have name that is the same as the name of a child of some other employee. One solution to this problem is to assign unique numbers to all children of employees in the company. Another, more natural, solution is to identify each child by his/her name *and* the primary key of the parent who is an employee of the company. We expect names of the children of an employee to be unique. Such attribute(s) that discriminates between all the entities that are dependent on the same parent entity is sometime called a *discriminator*; it cannot be called a key since it does not uniquely identify an entity without the use of the relationship with the employee. Similarly history of employment (Position, Department) would not have a primary key without the support of the employee primary key. Such entities that require a relationship to be used in identifying them are called *weak entities*. Entities that have primary keys are called *strong or regular entities*. Similarly, a relationship may be weak or strong (or regular). A *strong relationship* is between entities each of which is strong; otherwise the relationship is a *weak relationship*. For example, any relationship between the children of employees and the schools they attend would be a weak relationship. A relationship between employee entity set and the employer entity set is a strong relationship.

A weak entity is also called *subordinate* entity since its existence depends on another entity (called the *dominant* entity). This is called *existence dependence*. A weak entity may also be *ID dependent* on the dominant entity, although existence dependency does not imply ID dependency. If a weak entity is ID dependent, the primary key of the weak entity is the primary key of its dominant entity plus a set of attributes of the weak entity that can act as a discriminator within the weak entity set. This method of identifying entities by relationships with other entities can be applied recursively until a strong entity is reached although the relationship between dominant entity and a weak entity is usually one-to-one, in some situations the relationship may be many-to-many. For example, a company may employ both parents of some children. The dependence is then many-to-many.

We note several terms that are useful:

1. Weak entity relation - a relation that is used for identifying entities, for example, relationship between employees and their dependents.
2. Regular entity relation - a relation not used for identifying entities.
3. Similarly regular relationship relation and weak relationship relations.

The ER Diagrams

As noted earlier, one of the aims of building an entity-relationship diagram is to facilitate communication between the database designer and the end user during the requirements analysis. To facilitate such communication, the designer needs adequate communication tools. Entity-Relationship diagrams are such tools that enable the database designer to display the overall database view of the enterprise (the enterprise conceptual schema).

An E-R diagram naturally consists of a collection of entity sets and relationship sets and their associations. A diagram may also show the attributes and value sets that are needed to describe the entity sets and the relationship sets in the ERD. In an ERD, as shown in Figure 2.4, entity sets are represented by rectangular shaped boxes. Relationships are represented by diamond shaped boxes.

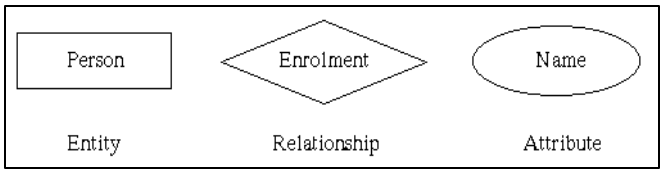


Figure 2.3

Ellipses are used to represent attributes and lines are used to link attributes to entity sets and entity sets to relationship sets.

Consider the following E-R diagram.

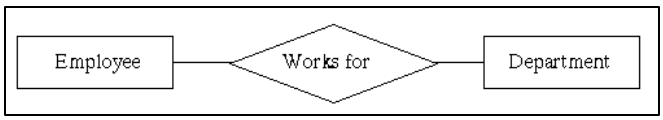


Figure 2.4a A simple E-R diagram. The following E-R diagram gives the attributes as well.

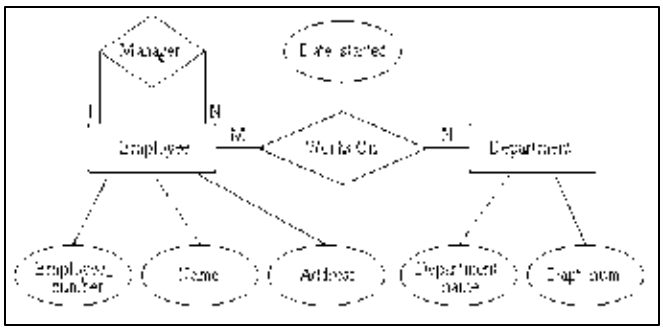


Figure 2.4b An E-R diagram with attributes.

The following E-R diagram represents a more complex model that includes a weak entity.

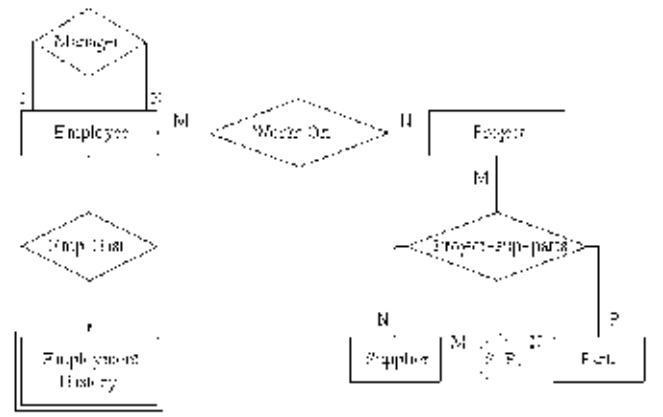


Figure 2.4c An E-R diagram showing a weak entity. The following E-R diagram represents more than one relationship between the same two entities.

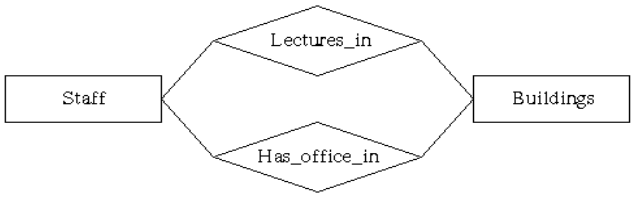


Figure 2.4d More than one relationship between two entities. The diagram below shows unary relationships.

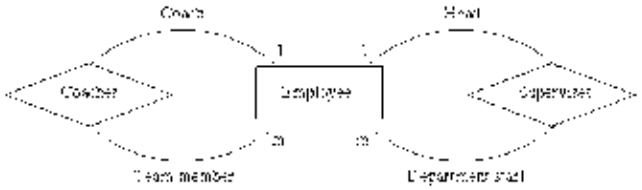


Figure 2.4e Two unary relationships.

Figures 2.4a to 2.4e show a variety of E-R diagrams. Figure 2.4a is an E-R diagram that only shows two entities and one relationship and not the attributes. Figure 2.4b shows the same two entities but shows an additional relationship between employees and their manager. This additional relationship is between the entity set *Employee* to itself. In addition, the E-R diagram in Figure 2.4b shows the attributes as well as the type of relationships (1:1, m:n or 1:m). The type of relationship is shown by placing a label of 1, m or n on each end of every arc. The label 1 indicates that only one of those entity instances may participate in a given relationship. A letter label (often m or n) indicates that a number of these entity instances may participate in a given relationship. The E-R diagrams may also show roles as labels on the arcs.

The E-R diagram in Figure 2.4c shows a number of entities and a number of relationships. Also note that the entity employment history that is shown as double box. A double box indicates that *Employment History* is a weak entity and that its existence depends on existence of corresponding employee entity. The existence dependency of employment history entity on employee entity is indicated by an arrow. A relationship between three entities is shown. Note that each entity may participate in several relationships.

The E-R diagram in Figure 2.4d shows two entities and two different relationships between the same two entities. Figure 2.4e shows two relationships also but these both relationships are between the entity *Employee* to itself.

Entity Type Hierarchy

Although entity instances of one entity type are supposed to be objects of the same type, it often happens that objects of one entity type do have some differences. For example, a company might have vehicles that could be considered an entity type. The vehicles could however include cars, trucks and buses and it may then be necessary to include capacity of the bus for buses and the load capacity of the trucks for trucks, information that is not relevant for cars. In such situations, it may be necessary to deal with the subsets separately while maintaining the view that all cars, trucks and buses are vehicles and share a lot of information.

Entity hierarchies are known by a number of different names. At least the following names are used in the literature:

1. Supertypes and subtypes
2. Generalization hierarchies
3. ISA hierarchies

We will not discuss entity hierarchies any further although use of hierarchies is now recognized to be important in conceptual modeling.

Guidelines for Building an ERM

We have so far discussed the basics of the E-R models and the representation of the model as an E-R diagram. Although the E-R model approach is often simple, a number of problems can arise. We discuss some of these problems and provide guidelines that should assist in the modeling process.

Choosing Entities

As noted earlier, an entity is an object that is of interest. It is however not always easy to decide when a given thing should be considered an entity. For example, in a supplier-part database, one may have the following information about each supplier

Supplier number
supplier name
supplier rating
supplier location (i.e. city)

It is clear that supplier is an entity but one must now make a decision whether city is an entity or an attribute of entity supplier. The rule of thumb that should be used in such situations is to ask the question "Is city as an object of interest to us?". If the answer is yes, we must have some more information about each city than just the city name and then city should be considered an entity. If however we are only interested in the city name, city should be considered an attribute of supplier.

As a guideline therefore, each entity should contain information about its properties. If an object has no information other than its identifier that interests us, the object should be an attribute.

Multivalued Attributes

If an attribute of an entity can have more than one value, the attribute should be considered an entity. Although conceptually multi-value attributes create no difficulties, problems arise when the E-R model is translated for implementation using a DBMS. Although we have indicated above that an entity should normally contain information about its properties, a multi-value attribute that has no properties other than its value should be considered an entity.

Database Design Process

When using the E-R model approach to database design, one possible approach is to follow the major steps that are listed below:

1. Study the description of the application.
2. Identify entity sets that are of interest to the application.
3. Identify relationship sets that are of interest to the application. Determine whether relationships are 1:1, 1: n or m: n.
4. Draw an entity-relationship diagram for the application.
5. Identify value sets and attributes for the entity sets and the relationship sets.
6. Identify primary keys for entity sets.
7. Check that ERD conforms to the description of the application.
8. Translate the ERD to the database model used by the DBMS.

It should be noted that database design is an iterative process.

Rigorous definition of Entity

Although we have discussed the concept of an entity, we have not presented a rigorous definition of an entity. A simple rigorous definition is not possible since there is no absolute distinction between entity types and attributes. Usually an attribute exists only as related to an entity type but in some contexts, an attribute can be viewed as an entity.

Further complications arise because an entity must finally be translated to relation. Since relations have a somewhat inflexible structure, an entity itself must satisfy several artificial constraints. For example, problems arise when things that we consider entities have attributes that are mutli-valued. The attribute then needs to be declared an entity to avoid problems that will appear in mapping an entity with multi-values attribute to the relational database.

Consider for example, an entity called *vendor*. The vendor has several attributes including vendor number, vendor name, location, telephone number etc. Normally the location will be considered an attribute but should we need to cater for a vendor with several branches, the location would need to be made an entity and a new relationship *located-in* would be needed.

To overcome some of the above problems, rule of thumbs like the following should be followed:

1. Entities have descriptive information; identifying attributes do not.
2. Multivalued attributes should be classed as entities.
3. Make an attribute that has a many-to-one relationship with an entity.
4. Attach attributes to entities that they describe most directly.
5. Avoid composite identifiers as much as possible.

Also there is no absolute distinction between an entity type and a relationship type although a relationship is usually regarded as unable to exist on its own. For example, an enrolment cannot be expected to exist on its own without the entities students and subjects.

A careful reader would have noticed our reluctance to discuss time in our data models. We have assumed that only current attribute values are of interest. For example, when an employee's salary changes, the last salary value disappears for ever. In many applications this situation would not be satisfactory. One way to overcome such problem would be to have an entity for *salary history* but this is not always the most satisfactory solution. Temporal data models deal with the problem of modeling time dependent data.

The Facts-based View (of W.Kent) !!

If we study what we usually store in records we find that fields are character strings that usually represent facts. Any field by itself conveys little useful information. For example, a field may convey a name or a fact like department number but most useful information is conveyed only when the interconnections between fields are also conveyed. Information therefore is

expressed not only by facts represented in fields but more importantly by relations among fields. Records tie information together; they represent aggregation of facts. Records may be considered to have three components:

1. What is each field about i.e. the entity type.
2. How each field refers to the entity representation
3. What information each field conveys i.e. a relationship or a fact about the entity.

The facts based view is based on the premise that rather than choosing entities as clustering points and aggregating facts around them we use a design based on aggregating single-valued related facts together.

Kent suggests the following as basis for facts based view of database design:

1. Single-valued facts about things are maintained in records having that thing's identifier.
2. Several single-valued facts about the same thing can be maintained in the same record.
3. All other kinds of facts (e.g. multi-valued) are maintained in separate records, one record for each fact.

The following methodology may be followed for fact-based database design:

1. Identify the facts to be maintained. Identify which are single-valued.
2. Generate a pseudo-record for each fact.
3. Identify "pseudo-keys" for each pseudo-record, based in single-valuedness of facts.
4. Merge pseudo-records having compatible keys
5. Assign representations
6. Consider alternative designs
7. Name the fields and record types

Extended E-R Model

When the E-R model is used as a conceptual schema representation, difficulties may arise because of certain inadequacies of the initial E-R model constructs. For example, view integration often requires abstraction concepts such as generalization. Also data integrity involving null attribute values requires specification of structural constraints on relationships. To deal with these problems the extended E-R model includes the following extensions:

1. A concept of "Category" is introduced to represent generalization hierarchies and subclasses
2. Structural constraints on relationships are used to specify how entities may participate in relationships.

A category is a subset of entities from an entity set. For example, a manager, secretary and technician may be categories of entity set Employee.

Employee

Manager Secretary Technician

The categories share most attributes of the entity type whose subset they are, some attributes may not be shared. For example, department may apply to Manager and Secretary and not to technician.

Further extensions have been suggested. For example, Chen has suggested that a concept of "composite entity" be added to the E-R model. A composite entity is an entity formed by other entities like a relationship. It differs from relationship in that a relationship cannot have attributes (descriptors) while a composite entity can.

Summary

The entity-relationship model is useful because, as we will soon see, it facilitates communication between the database designer and the end user during the requirements analysis. An E-R diagram naturally consists of a collection of entity sets and relationship sets and their associations. An entity is a person, place, or a thing or an object or an event which can be distinctly identified and is of interest.

Relationships are associations or connections that exist between entities and may be looked at as mappings between two or more entity sets.

Questions

1. What do you mean by entity relationship model?
2. Explain Entity set and relationship set?
3. Explain entity type?
4. Explain extended E-R Model?

Activities

An Example

Consider building a data model for a University. The information available includes:

1. Students with associated information (student number, name, address, telephone number)
2. Academic staff with associated information (staff number, name, address, telephone number, salary, department, office number)
3. Courses with associated information (course number, course name, department name, semester offered, lecture times, tutorial times, room numbers, etc.)
4. Students take courses and are awarded marks in courses completed.
5. Academic staffs are associated with courses. It changes from year to year.

References

1. P. P. Chen (1976), "The entity-relationship model: Towards a unified view of data", ACM Trans. Database Systems, Vol. 1, pp. 9-36.
2. P. P. Chen (1977), "Entity Relationship Approach to Logical Database Design", Monograph Series, QED Information Sciences Inc., Wellesley, Ma.
3. P. P. Chen (1980), Ed., "Entity-Relationship to Systems Analysis and Design", North-Holland, New York/ Amsterdam.

4. W. Kent (1986), "The Realities of Data: Basic Properties of Data Reconsidered", in *Database Semantics (DS-1)*, T.B.Steel and R.Meersman, Eds., North-Holland, pp. 175-188.
5. Ng, P. (1981), "Further Analysis of the Entity-Relationship Approach to Database Design", IEEE Trans. on Soft. Engg., Vol. SE-7, No. 1, 1981, pp. 85-98.
6. T. J. Theorey, D. Yang and J. P. Fry (1986), "Logical Design Methodology for Relational Databases", ACM Computing Surveys, June 1986, pp. 197-222.
7. Yao, S. B. (1985), "Principles of Database Design, Vol. 1, Logical Organizations", Prentice-Hall.

Notes:

LESSON 5
RELATIONAL MODEL

Hi! Welcome to the fascinating world of DBMS. Here in this lecture we are going to discuss about the Relational Model.

We have discussed the E-R Model, a technique for building a logical model of an enterprise. Logical models are high level abstract views of an enterprise data. In building these models little thought is given to representing or retrieving data in the computer. At the next lower level, a number of data models are available that provide a mapping for a logical data model like the E-R model. These models specify a conceptual view of the data as well as a high level view of the implementation of the database on the computer. They do not concern themselves with the detailed bits and bytes view of the data.

What is relational model?

The relational model was proposed by E. F. Codd in 1970. It deals with database management from an abstract point of view. The model provides specifications of an abstract database management system. The precision is derived from solid theoretical foundation that includes predicate calculus and theory of relations. To use the database management systems based on the relational model however, users do not need to master the theoretical foundations. Codd defined the model as consisting of the following three components:

Data Structure - a collection of data structure types for building the database.

Data Manipulation - a collection of operators that may be used to retrieve, derive or modify data stored in the data structures.

Data Integrity - a collection of rules that implicitly or explicitly define a consistent database state or changes of states.

Data Structure

Often the information that an organization wishes to store in a computer and process is complex and unstructured. For example, we may know that a department in a university has 200 students, most are full-time with an average age of 22 years, and most are females. Since natural language is not a good language for machine processing, the information must be structured for efficient processing. In the relational model the information is structures in a very simple way.

The beauty of the relational model is its simplicity of structure. Its fundamental property is that all information about the entities and their attributes as well as the relationships is presented to the user as tables (called *relations*) and nothing but tables. The rows of the tables may be considered records and the columns as fields. Each row therefore consists of an entity occurrence or a relationship occurrence. Each column refers to an attribute. The model is called relational and not tabular because tables are a lower level of abstractions than the mathematical concept of relation. Tables give the impression that positions of rows and columns are important. In the relational model, it is assumed that no ordering of rows and columns is defined.

We consider the following database to illustrate the basic concepts of the relational data model.

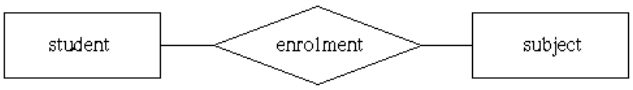


Figure 3.1 A simple student database

We have not included the attributes in the above E-R diagram. The attributes are student id, student name and student address for the entity set student and subject number, subject name and department for the entity set subject.

The above database could be mapped into the following relational schema which consists of three relation *schemes*. Each relation scheme presents the structure of a relation by specifying its name and the names of its attributes enclosed in parenthesis. Often the primary key of a relation is marked by underlining.

student(student_id, student_name, address)
enrolment(student_id, subject_id)
subject(subject_id, subject_name, department)

An example of a database based on the above relational model is:

student id	student name	address
8656789	Peta Williams	9, Davis Hall
8700071	John Smith	9, Davis Hall
8800020	Arun Kumar	90, Second Hall
8801234	Peter Chew	88, Long Hall
8654321	Reena Rani	88, Long Hall
8712371	Kathy Garcia	88, Long Hall
8612345	Chris Watanabe	11, Main Street

Table 1. The relation student

student id	subject id
8700074	CP302
8900020	CP302
8900020	CP304
8700071	MA111
8801234	CP302
8801234	CH001

Table 2. The relation enrolment

subject_id	subject_name	department
CP302	Database Management	Comp. Science
CP304	Software Engineering	Comp. Science
CH001	Introduction to Chemistry	Chemistry
PH101	Physics	Physics
MA111	Pure Mathematics	Mathematics

Table 3. The relation subject

We list a number of properties of relations:

Each relation contains only one record type.

Each relation has a fixed number of columns that are explicitly named. Each attribute name within a relation is unique.

No two rows in a relation are the same.

Each item or element in the relation is atomic, that is, in each row, every attribute has only one value that cannot be decomposed and therefore no repeating groups are allowed.

Rows have no ordering associated with them.

Columns have no ordering associated with them (although most commercially available systems do).

The above properties are simple and based on practical considerations. The first property ensures that only one type of information is stored in each relation. The second property involves naming each column uniquely. This has several benefits. The names can be chosen to convey what each column is and the names enable one to distinguish between the column and its domain. Furthermore, the names are much easier to remember than the position of the position of each column if the number of columns is large.

The third property of not having duplicate rows appears obvious but is not always accepted by all users and designers of DBMS. The property is essential since no sensible context free meaning can be assigned to a number of rows that are exactly the same.

The next property requires that each element in each relation be atomic that cannot be decomposed into smaller pieces. In the relation model, the only composite or compound type (data that can be decomposed into smaller pieces) is a relation. This simplicity of structure leads to relatively simple query and manipulative languages.

A relation is a set of tuples. As in other types of sets, there is no ordering associated with the tuples in a relation. Just like the second property, the ordering of the rows should not be significant since in a large database no user should be expected to have any understanding of the positions of the rows of a relation. Also, in some situations, the DBMS should be permitted to reorganize the data and change row orderings if that is necessary for efficiency reasons. The columns are identified by their names. The tuple on the other hand are identified by their contents, possibly the attributes that are unique for each tuple.

The relation is a set of tuples and is closely related to the concept of relation in mathematics. Each row in a relation may be viewed as an assertion. For example, the relation *student* asserts that a student by the name of *Reena Rani* has *student_id 8654321* and lives at *88, Long Hall*. Similarly the relation *subject* asserts that one of the subjects offered by the Department of Computer Science is *CP302 Database Management.*

Each row also may be viewed as a point in a n-dimensional space (assuming n attributes). For example, the relation *enrolment* may be considered a 2-dimensional space with one dimension being the *student_id* and the other being *subject_id*. Each tuple may then be looked at as a point in this 2-dimensional space.

In the relational model, a relation is the only compound data structure since relation does not allow repeating groups or pointers.

We now define the relational terminology:

Relation - essentially a table
Tuple - a row in the relation
Attribute - a column in the relation
Degree of a relation - number of attributes in the relation
Cardinality of a relation - number of tuples in the relation
N-ary relations - a relation with degree N
Domain - a set of values that an attribute is permitted to take. Same domain may be used by a number of different attributes.
Primary key - as discussed in the last chapter, each relation must have an attribute (or a set of attributes) that uniquely identifies each tuple.

Each such attribute (or a set of attributes) is called a *candidate key* of the relation if it satisfies the following properties:

a. The attribute or the set of attributes uniquely identifies each tuple in the relation (called *uniqueness*), and

b. If the key is a set of attributes then no subset of these attributes has property (a) (called *minimality*).

There may be several distinct set of attributes that may serve as candidate keys. One of the candidate keys is arbitrarily chosen as the *primary key* of the relation.

The three relations above *student*, *enrolment* and *subject* have degree 3, 2 and 3 respectively and cardinality 4, 6 and 5 respectively. The primary key of the relation *student* is *student_id,* of relation *enrolment* is *(student_id, subject_id)*, and finally the primary key of relation *subject* is *subject_id*. The relation *student* probably has another candidate key. If we can assume the names to be unique than the *student_name* is a candidate key. If the names are not unique but the names and address together are unique, then the two attributes *(student_id, address)* is a candidate key. Note that both *student_id* and *(student_id, address)* cannot be candidate keys, only one can. Similarly, for the relation *subject*, *subject name* would be a candidate key if the subject names are unique.

The definition of a relation presented above is not precise. A more precise definition is now presented. Let $D_1, D_2, \ldots, D_n$ be n atomic domains, not necessarily distinct. R is then a degree n relation on these domains, if it is a subset of the cartesian product of the domains, that is, $D_1 X D_2 X \ldots X D_n$. If we apply this definition to the relation *student*, we note that *student* is a relation on domains *student_id* (that is, all permitted *student_id*'s), *student_name* (all permitted student names) and *address* (all permitted values of addresses). Each instance of the relation *student* is then a subset of the product of these three domains.

We will often use symbols like *R* or *S* to denote relations and *r* and *s* to denote tuples. When we write $r \subset R$, we assert that tuple *r* is in relation *R*.

In the relational model, information about entities and relationships is represented in the same way i.e. by relations. Since the structure of all information is the same, the same operators may be applied to them. We should note that not all

relationship types need to be represented by separate relations. For 1:1 and 1:m relationships, key propagation may be used. Key propagation involves placing the primary key of one entity within the relation of another entity. Many-to-many relationships however do require a separate relation to represent the association.

We illustrate the use of key propagation by considering the following E-R model:

Figure 3.2 a simple database

Let the relationship *occupies* be many-to-one, that is, one staff may occupy only one room but a room may be occupied by more than one staff. Let the attributes of the entity *staff* be staff number (*s_num*), staff name (*name*) and *status*. Attributes of the entity *room* are room number (*r_num*), *capacity*, *building*. *occupies* has no attributes. *s_num* and *r_num* are the primary keys of the two entities *staff* and *room.*

One way to represent the above database is to have the following three relations:

staff(s_num, name, status)
occupies(s_num, r_num)
room(r_num, capacity, building)

There is of course another possibility. Since the relationship *occupies* is many-to-one, that is, each staff occupies only one room, we can represent the relationship by including the primary key of relation *room* in the entity *staff*. This is propagation of the key of the entity *room* to entity *staff*. We then obtain the following database:

staff(s_num, name, status, r_num)
room(r_num, capacity, building)

We should note that propagating the key of the entity *staff* to the entity *room* would not be appropriate since some rooms may be associated with more than one staff member. Also note that key propagation would not work if the relationship *occupies* is many-to-many.

Data Manipulation

The manipulative part of relational model makes set processing (or relational processing) facilities available to the user. Since relational operators are able to manipulate relations, the user does not need to use loops in the application programs. Avoiding loops can result in significant increase in the productivity of application programmers.

The primary purpose of a database in an enterprise is to be able to provide information to the various users in the enterprise. The process of querying a relational database is in essence a way of manipulating the relations that are the database. For example, in the student database presented in Tables 1-3, one may wish to know names of all students enrolled in CP302, or names of all subjects taken by John Smith.

We will show how such queries may be answered by using some of several data manipulation languages that are available for relational databases. We first discuss the two formal languages that were presented by Codd. These are called the *relational algebra* and *relational calculus*. These will be followed by a discussion of a commercial query language SQL.

Questions

1. What do you mean by relational model?
2. Explain what is a data structure?
3. Explain what do you mean by data manipulation?

References

1. Codd, E. F. (1974), “A Relational Model of Data for Large Shared Data Banks”, Comm. ACM, Vol. 13, No. 6, pp. 377-387.
2. Codd, E. F. (1974), “Recent Investigations in Relational Data Base Systems”, in Information Processing 74, North Holland.
3. Codd, E. F. (1982), “Relational Database: A Practical Foundation for Productivity”, in Comm. ACM, Vol. 25, No. 2, pp. 109-117.
4. Codd, E. F. (1981), “The Capabilities of Relational Database Management Systems”, IBM Report RJ3132.

Notes:

LESSON 6
RELATIONAL ALGEBRA PART - I

Relational Algebra-Part I

Hi! I would like to discuss with you about the relational algebra, the basis of SQL language.

A brief introduction

- Relational algebra and relational calculus are formal languages associated with the relational model.
- Informally, relational algebra is a (high-level) procedural language and relational calculus a non-procedural language.
- However, formally both are equivalent to one another.
- A language that produces a relation that can be derived using relational calculus is relationally complete.
- Relational algebra operations work on one or more relations to define another relation without changing the original relations.
- Both operands and results are relations, so output from one operation can become input to another operation.
- Allows expressions to be nested, just as in arithmetic. This property is called closure.

What? Why?

- Similar to normal algebra (as in 2+3*x-y), except we use relations as values instead of numbers.
- Not used as a query language in actual DBMSs. (SQL instead.)
- The inner, lower-level operations of a relational DBMS are, or are similar to, relational algebra operations. We need to know about relational algebra to understand query execution and optimization in a relational DBMS.
- Some advanced SQL queries requires explicit relational algebra operations, most commonly *outer join*.
- SQL is *declarative*, which means that you tell the DBMS *what* you want, but not *how* it is to be calculated. A C++ or Java program is *procedural*, which means that you have to state, step by step, exactly how the result should be calculated. Relational algebra is (more) procedural than SQL. (Actually, relational algebra is mathematical expressions.)
- It provides a formal foundation for operations on relations.
- It is used as a basis for implementing and optimizing queries in DBMS software.
- DBMS programs add more operations which cannot be expressed in the relational algebra.
- Relational calculus (tuple and domain calculus systems) also provides a foundation, but is more difficult to use. We'll skip these for now.

The Basic Operations in Relational Algebra

- Basic Operations:
  - Selection (s): choose a subset of rows.
  - Projection (p): choose a subset of columns.
  - Cross Product (´): Combine two tables.
  - Union (È): unique tuples from either table.
  - Set difference (-): tuples in R1 not in R2.
  - Renaming (r): change names of tables & columns
- Additional Operations (for convenience):
  - Intersection, joins (*very useful*), division, outer joins, aggregate functions, etc.

Now we will see the various operations in relational algebra in detail. So get ready to be in a whole new world of algebra which posses the power to extract data from the database.

Selection Operation s

The **select** command gives a programmer the ability to choose tuples from a relation (rows from a table). Please do not confuse the Relational Algebra **select** command with the more powerful SQL **select** command that we will discuss later.

Idea: choose tuples of a relation (rows of a table)

Format: **sselection-condition**(R). Choose tuples that satisfy the ***selection condition***.

Result has identical schema as the input.

σMajor = 'CS' **(Students)**

This means that, the desired output is to display the name of students who has taken CS as Major. The Selection condition is a Boolean expression including **=, ¹ , <, £, >, ³** , and, or, not.

Students

SID	Name	GPA	Major
456	John	3.4	CS
457	Carl	3.2	CS
678	Ken	3.5	Math

Result

SID	Name	GPA	Major
456	John	3.4	CS
457	Carl	3.2	CS

Once again, all the Relational Algebra select command does choose tuples from a relation. For example, consider the following relation R (A, B, C, D):

R : Table

	A	B	C	D
	a1	b2	c1	d1
	a2	b1	c3	d2
	a3	b3	c3	d2
	a4	b3	c1	d3
	a5	b2	c2	d4

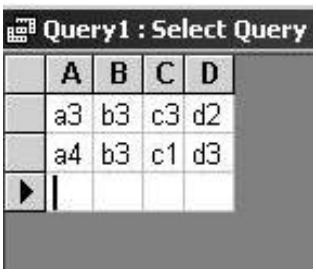
I call this an "abstract" table because there is no way to determine the real world model that the table represents. All we

know is that attribute (column) A is the primary key and that fact is reflected in the fact that no two items currently in the A column of R are the same. Now using a popular variant of Relation Algebra notation…if we were to do the Relational Algebra command

Select R where B > 'b2' Giving R1;

or another variant
```
R1 = R where B > 'b2';
```

We would create a **relation R1** with the ***exact*** same attributes and attribute domains (column headers and column domains) as **R**, but we would select only the tuples where the B attribute value is greater than 'b2'. This table would be



(I wrote the query in Microsoft Access XP). Important things to know about using the Relational Algebra **select** command is that the Relation produced always has the exact same attribute names and attribute domains as the original table - we just delete out certain columns.

Let us Consider the following relations

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S2	P1	300
S2	P2	400
S3	P2	200

P

P#	PNAME	COLOUR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London

Fig.1

Now based on this consider the following examples

e.g. SELECT S WHERE CITY = 'PARIS'

S#	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris

e.g. SELECT SP WHERE (S# = S1 and P# = P1)

S#	P#	QTY
S1	P1	300

The resulting relation has the same attributes as the original relation.

The selection condition is applied to each tuple in turn - it cannot therefore involve more than one tuple.

Project Operation

The Relational Algebra **project** command allows the programmer to choose attributes (columns) of a given relation and delete information in the other attributes

Idea: Choose certain attributes of a relation (columns of a table)

Format: $\pi_{\text{Attribute\_List}}$(Relation)

Returns: a relation with the same tuples as (Relation) but limited to those attributes of interest (in the attribute list).selects some of the *columns* of a table; it constructs a vertical subset of a relation; implicitly removes any duplicate tuples (so that the result will be a relation).

π Major(Students)

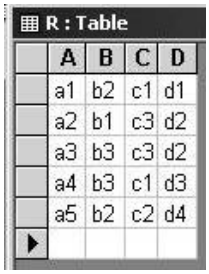
Students

SID	Name	GPA	Major
456	John	3.4	CS
457	Carl	3.2	CS
678	Ken	3.5	Math

Result

Major
CS
Math

For example, given the original abstract relation R(A, B, C, D):

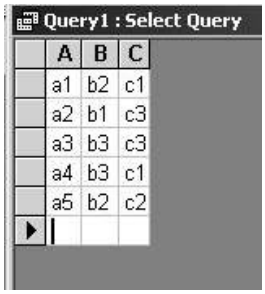


We can pick out columns A, B, and C with the following:

Project R over [A, B, C] giving R2;

```
R2 = R[A, B, C];
```

This would give us a relation R2(A, B, C) with the D attribute gone:

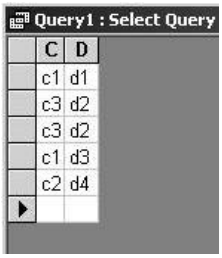


There is one slight problem with this command. Sometimes the result might contain a duplicate tuple. For example, what about

Project R over [C, D] giving R3;

```
R3 = R[C, D];
```

What would R3(C, D) look like?



There are two (c3, d2) tuples. This is not allowed in a "legal" relation. What is to be done? Of course, in Relational Algebra, all duplicates are deleted. Now consider the following examples based on the following examples based on the Fig.1

e.g. PROJECT S OVER CITY

CITY
London
Paris

e.g. PROJECT S OVER SNAME, STATUS

SNAME	STATUS
Smith	20
Jones	10
Blake	30

Sequences of Operations

Now we can see the sequence of operations based on both selection and Projection operations.

E.g. part names where weight is less than 17:

TEMP <- SELECT P WHERE WEIGHT < 17

RESULT <- PROJECT TEMP OVER PNAME

TEMP	P#	PNAME	COLOUR	WEIGHT	CITY
	P1	Nut	Red	12	London
	P4	Screw	Red	14	London

RESULT	PNAME
	Nut
	Screw

or (nested operations)

PROJECT (SELECT P WHERE WEIGHT < 17) OVER PNAME

Renaming Operation

Format: ρS(R) or ρS(A1, A2, …)(R): change the name of relation R, and names of attributes of R

r **CS_Students(sMajor = 'CS' Students))**

Students

SID	Name	GPA	Major
456	John	3.4	CS
457	Carl	3.2	CS
678	Ken	3.5	Math

CS_Students

SID	Name	GPA	Major
456	John	3.4	CS
457	Carl	3.2	CS

Consider the scenario

- $p_{List1}(\mathbf{s}_{cond1}(R1))$
- $p_{List2}(\mathbf{s}_{cond1}(R1))$
- $\mathbf{s}_{cond2}(p_{List1}(\mathbf{s}_{cond1}(R1)))$

It would be useful to have a notation that allows us to "save" the output of an operation for future use.

- Tmp1 ← $\mathbf{s}_{cond1}(R1)$
- Tmp2 ← $p_{List1}(Tmp1)$
- Tmp3 ← $p_{List2}(Tmp2)$
- Tmp4 ← $\mathbf{s}_{cond2}(Tmp2)$

(Of course, we would come up with better names than Tmp1, Tmp2….)

The resulting temporary relations will have the same attribute names as the originals. We might also want to change the attribute names:

- To avoid confusion between relations.
- To make them agree with names in another table. (You'll need this soon.)

For this we will define a Rename operator (rho):

$\mathbf{r}_{S(B_1,B_2,\ldots,B_n)}$ ($R(A_1,A_2,\ldots,A_n)$)

where S is the new relation name, and $B_1 \ldots B_n$ are the new attribute names. Note that the degree(S) = degree(R) = n.

Examples

$\mathbf{r}_{EmpNames(LAST,FIRST)}(p_{LNAME,FNAME}(Employee))$

$\mathbf{r}_{Works\_On2(SSN,PNUMBER,HOURS)}(Works\_On(ESSN,PNO,HOURS))$

The Cartesian Product

The *cartesian product* of two tables combines each row in one table with each row in the other table.

The **Cartesian product** of *n* domains, written $dom(A_1) \times dom(A_2) \times \ldots dom(A_n)$, is defined as follows.

$(A_1 \times A_2 \times .. A_n = \{(a_1, a_2, ..., a_n) \mid a_1 \times A_1$ **AND** $a_2 \times A_2$ **AND** ... **AND** $a_n \times A_n\}$

We will call each element in the Cartesian product a **tuple**. So each $(a_1, a_2, ..., a_n)$ is known as a tuple. Note that in this formulation, the order of values in a tuple matters (we could give an alternative, but more complicated definition, but we shall see later how to swap attributes around in a relation).

Example: The table **E** (for **EMPLOYEE**)

enr	ename	dept
1	Bill	A
2	Sarah	C
3	John	A

Example: The table **D** (for **DEPARTMENT**)

dnr	dname
A	Marketing
B	Sales
C	Legal

SQL	Result					Relational algebra
	enr	**ename**	**dept**	**dnr**	**dname**	
	1	Bill	A	A	Marketing	
	1	Bill	A	B	Sales	
	1	Bill	A	C	Legal	
select *	2	Sarah	C	A	Marketing	E **X** D
from E, D	2	Sarah	C	B	Sales	
	2	Sarah	C	C	Legal	
	3	John	A	A	Marketing	
	3	John	A	B	Sales	
	3	John	A	C	Legal	

Division

$R \div S$

- Defines a relation over the attributes C that consists of set of tuples from R that match combination of *every* tuple in S.
- Expressed using basic operations:

$T1 \leftarrow \Pi C(R)$

$T2 \leftarrow \Pi C((S \times T1) - R)$

$T \leftarrow T1 - T2$

The division operation (÷) is useful for a particular type of query that sometimes occurs in database applications. For example, if I want to organise a study group,I would like to find people who do the same subjects I do. The division operator provides you with the facility to perform this query without the need to "hard code" the subjects involved.

My_Subjects = ðsubj_code (ó Stud_ID = '04111' (Enrolment))

Study_Group = Enrolment ÷ My_Subjects

Joins

Theta-join

The **theta-join** operation is the most general join operation. We can define theta-join in terms of the operations that we are familiar with already.

$R \infty qS = s\,q(R \times S)$

So the join of two relations results in a subset of the Cartesian product of those relations. Which subset is determined by the *join condition*: q . Let's look at an example. The result of

Professions $q_{\text{Job = Job}}$ Careers is shown below.

Name	Job	Job	Pays
Joe	Garbageman	Garbageman	50000
Sue	Doctor	Doctor	40000
Joe	Surfer	Surfer	6500

Equi-join

The join condition, q , can be any well-formed logical expression, but usually it is just the conjunction of equality comparisions between pairs of attributes, one from each of the joined relations. This common case is called an **equi-join**. The example given above is an example of an equi-join.

Outer Joins:

- A join operation is complete, if all the tuples of the operands contribute to the result.
- Tuples not participating in the result are said to be dangling
- Outer join operations are variants of the join operations in which the dangling tuples are appended with NULL fields. They can be categorized into left, right, and full outer joins.

INSTRUCTOR $\overset{\text{left}}{\bowtie}_{\text{right}}$ ADMINISTRATOR

Name	Number	Adm
Don	3412	Dean
Nel	2341	NULL
NULL	4123	Secretary

INSTRUCTOR $\overset{\text{left}}{\bowtie}$ ADMINISTRATOR

Name	Number	Adm
Don	3412	Dean
Nel	2341	NULL

We will discuss further about relational algebra in the next lecture.

Review Questions

1. Explain the basic operations in Relational Algebra
2. Explain the difference between union and intersection
3. Explain Project operation
4. Explain Renaming operation
5. Explain Cartesian Product

Notes:

LESSON 7
RELATIONAL ALGEBRA PART - II

Relational Algebra-Part II

Hi, I would like to extend the discussion on relational algebra and joins.

In general, a **join** is an operation that glues relations together. There are several kinds of joins. The join operation is very important for any relational database with more than a single table as it allows us to process more than one table at a time. A join Operation matches data from 2 or more tables, based on the values of one or more columns in each table.

The **join** is the method whereby two tables are combined so data from two or more tables can be used to extract information. In Relational Algebra, Codd defined the idea of a ***natural join***. The following description describes a natural join

To do a natural join of two relations, you examine the relations for common attributes (columns with the same name and domain). For example, look at the following abstract tables: R (A, B, C, D) and Q (B, E, F)

Q : Table

B	E	F
b1	e2	f3
b2	e7	f2
b3	e2	f3
b4	e1	f2

R : Table

A	B	C	D
a1	b2	c1	d1
a2	b1	c3	d2
a3	b3	c3	d2
a4	b3	c1	d3
a5	b2	c2	d4

Notice that relations R and Q both have attribute B with the same domain. Note also that there are no other attributes common to both. So … to do a ***natural join*** of R and Q, you match rows from R and rows of Q with the same B value.

Next, you must know that the result of a join is another table that contains the same attributes (columns) as both of the tables being joined. The attributes in Q are B, E, and F. The attributes in R are A, B, C, and D. The attributes in the join of Q and R are B, E, F, A, C, D. Notice that B only appears once in this list. For example, let us calculate the join of Q and R and call the resulting table T.

Join Q and R giving T;
Starting with the Q table (B is the primary key in the Q table), choose the first tuple (row). What tuple in R matches the tuple (b1, e2, f3) in Q? Well, it appears that only one tuple in Q fills the qualification that B = 'b1'. This tuple is (a2, b1, c3, d2). This result in the row in the joined table…Notice how the rows from the two tables make up one row in the result and that the B value only occurs once.

QJR : Table

B	E	F	A	C	D
b1	e2	f3	a2	c3	d2

The tuple (a2, b1, c3, d2) is the only tuple in R that matches the tuple (b1, e2, f3) in Q. Let us try the next tuple in Q:

Q : Table

B	E	F
b1	e2	f3
b2	e7	f2
b3	e2	f3
b4	e1	f2

R : Table

A	B	C	D
a1	b2	c1	d1
a2	b1	c3	d2
a3	b3	c3	d2
a4	b3	c1	d3
a5	b2	c2	d4

This is (b2, e7, f2). It matches every tuple in R with B = 'b2'. These are (a1, b2, c1, d1) and (a5, b2, c2, d4). Here you create two rows in the join.

QJR : Table

B	E	F	A	C	D
b1	e2	f3	a2	c3	d2
b2	e7	f2	a1	c1	d1
b2	e7	f2	a5	c2	d4

The next tuple in Q is (b3, e2, f3). There are two matching tuples in R where B = 'b3'. They are (a3, b3, c3, d2) and (a4, b3, c1, d4). So, we create two more rows in the join:

QJR : Table

B	E	F	A	C	D
b1	e2	f3	a2	c3	d2
b2	e7	f2	a1	c1	d1
b2	e7	f2	a5	c2	d4
b3	e2	f3	a3	c3	d2
b3	e2	f3	a4	c1	d4

The next and last tuple in Q is (b4, e1, f2). Note that there no tuples in R that match this tuple in Q with B = 'b4'. This means that no tuples are added to the natural join (to the **natural join** - in some of the other types of join, this is not true). So, the QJR table above is the final table in the join.

These three commands allow us to perform powerful and useful queries of databases. They are generally used together to obtain the results. Consider the following set of tables:

Sponsor : Table

SponsorId	SponsorName
S1	First National Bank
S2	House of Comets
S3	Swimmers Paradise
S4	Dime Store

Team : Table

TeamId	TeamName	SponsorId
T1	Chargers	S1
T2	Streakers	S3
T3	Predators	S2
T4	Strike	S3
T5	Extreme	S2

Player : Table

PlayerId	TeamId	PlayerName	PlayerPhone	PlayerParent
P1	T1	Jill	444-2323	Sally
P2	T1	Cathy	333-5555	Joan
P3	T2	Samantha	222-4565	Susan
P4	T2	Shelly	321-4321	Sarah
P5	T4	Karen	432-5432	Liz
P6	T1	Joyce	543-5432	Martha
P7	T3	Sandy	654-3456	Jan
P8	T3	Lois	989-8998	Carol

Let us consider the following query:

What are the names of the players that play for the Chargers?

What can be done to determine all the players on the Chargers? What would have to be done? First we would have to determine the TeamID of the Chargers. That would be 'T1'. Next, we would look down through the player table printing out all players with TeamId='T1'.

Now, think in terms of **select**, **project**, and **join**. Look at the following sequence of actions

Select Team where TeamName = "Chargers" giving Temp1;

Temp1 : Select Query

TeamId	TeamName	SponsorId
T1	Chargers	S1

Join Temp1 and Player Giving Temp2;

Temp2 : Select Query

TeamId	TeamName	SponsorId	PlayerId	PlayerName	PlayerPhone	PlayerParent
T1	Chargers	S1	P1	Jill	444-2323	Sally
T1	Chargers	S1	P2	Cathy	333-5555	Joan
T1	Chargers	S1	P6	Joyce	543-5432	Martha

Project Temp2 over PlayerName Giving Temp3;

Temp3 : Select Query

PlayerName
Jill
Cathy
Joyce

By looking at the original tables, these are obviously the correct girls.

Note that using the alternative notation that we have addressed above, we can combine the above statements to solve the query

What are the names of the players that play for the Chargers?

in one Relational Algebra statement. Look at how the following 3 statements

Temp1 = Team where TeamName = "Chargers";
Temp2 = Temp1 nJoin Player;
Temp3 = Temp2[PlayerName];

can be combined into

Temp3 = ((Team where TeamName = "Chargers") nJoin Player)

Select Player where PlayerName = "Sandy" Giving Temp4;

Let's try another…

What is the name of the team that Sandy plays for?

Select Player where PlayerName = "Sandy" Giving Temp4;

Temp4 : Select Query

PlayerId	TeamId	PlayerName	PlayerPhone	PlayerParent
P7	T3	Sandy	654-3456	Jan

Join Temp4 and Team Giving Temp5;

Temp5 : Select Query

PlayerId	TeamId	PlayerName	PlayerPhone	PlayerParent	TeamName	SponsorId
P7	T3	Sandy	654-3456	Jan	Predators	S2

Project Temp5 over TeamName Giving Temp6;

Temp6 : Select Query

TeamName
Predators

This is the correct name as can be seen by examining the tables.

Once again, we can combine the three Relational Algebra statements

Temp4 = Player where PlayerName = "Sandy";
Temp 5 = Temp4 nJoin Team;
Temp6 = Temp5[TeamName];into

Temp6 =
((Player where PlayerName = "Sandy") nJoin Team)
[TeamName];

Let us try one more…

What are the names of all the girls on teams sponsored by Swimmers Paradise;

How about…

Select Sponsor where SponsorName = "Swimmers Paradise" giving Temp7;

Temp7 : Select Query

SponsorId	SponsorName
S3	Swimmers Paradise

Join Temp7 and Team Giving Temp8;

Temp8 : Select Query

SponsorId	SponsorName	TeamId	TeamName
S3	Swimmers Paradise	T2	Streakers
S3	Swimmers Paradise	T4	Strike

Join Temp8 and Player Giving Temp9;

Temp9 : Select Query

SponsorId	SponsorName	TeamId	TeamName	PlayerId	PlayerName	PlayerPhone	PlayerParent
S3	Swimmers Paradise	T2	Streakers	P3	Samantha	222-4565	Susan
S3	Swimmers Paradise	T2	Streakers	P4	Shelly	321-4321	Sarah
S3	Swimmers Paradise	T4	Strike	P5	Karen	432-5432	Liz

Project Temp9 over Playername giving Temp10;

Temp10 : Select Query

PlayerName
Samantha
Shelly
Karen

If you will look closely at the original tables, these are the correct girls…

Once more, let us combine the statements into one statement:

Temp7 = Sponsor where SponsorName = "Swimmers Paradise";
Temp 8 = Temp7 nJoin Team;
Temp9 = Temp8 nJoin Player;
Temp10 = Temp9 [PlayerName];

can become

Temp10 = (((Sponsor where SponsorName = "Swimmers Paradise")
nJoin Team) nJoin Player) [PlayerName];

Now we will see the other joins.

SET Operations Union, Intersection and Set Difference
Consider two relations R and S.

The Union Operation
UNION of R and S the union of two relations is a relation that includes all the tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.

The **union** operation is denoted $\cup$ as in set theory. It returns the union (set union) of two compatible relations.

For a union operation $r \cup s$ to be legal, we require that

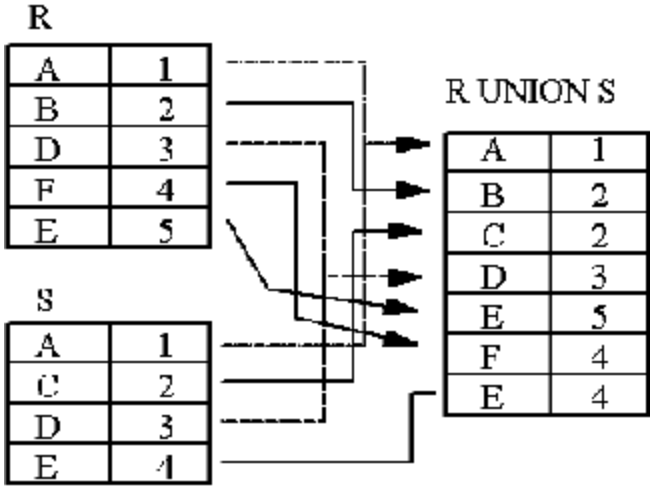
- r and s must have the same number of attributes.
- The domains of the corresponding attributes must be the same.

To find all customers of the SFU branch, we must find everyone who has a loan or an account or both at the branch.

We need both borrow and deposit relations for this:

$$\Pi_{cname}(\sigma_{bname="SFU"}(borrow)) \cup \Pi_{cname}(\sigma_{bname="SFU"}(deposit))$$

UNION Example



Intersection

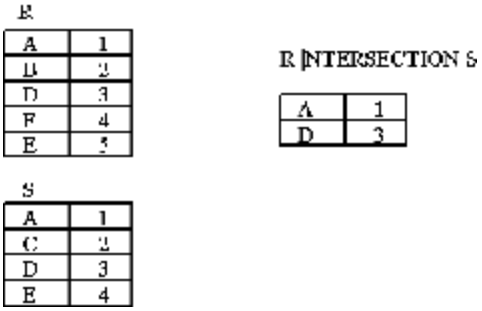
The interference of R and S includes all tuples that are both in R and S. The set intersection is a binary operation that is written as R ∩ S and is defined as the usual set intersection in set theory:

$R \cap S = \{ t : t \in R, t \in S \}$

The result of the set intersection is only defined when the two relations have the same headers. This operation can be simulated in the basic operations as follows:

$R \cap S = R - (R - S)$

Intersection Example



The Set Difference

DIFFERENCE of R and S is the relation that contains all the tuples that are in R but that are not in S. For set operations to function correctly the relations R and S must be union compatible. Two relations are union compatible if

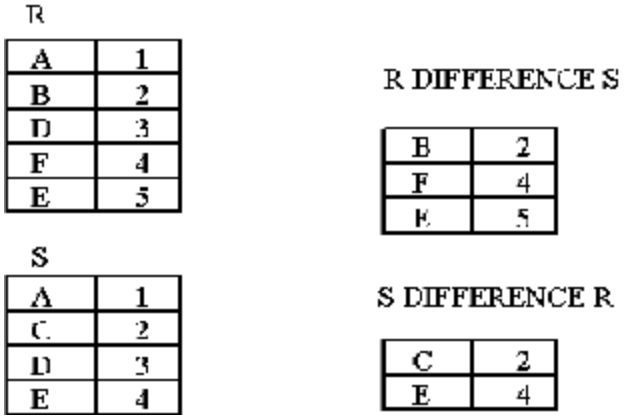
- They have the same number of attributes
- The domain of each attribute in column order is the same in both R and S.

The set difference is a binary operation that is written as R - S and is defined as the usual set difference in set theory:

$R\text{-}S = \{ t : t \in R, \neg t \in S \}$

The result of the set difference is only defined when the two relations have the same headers.

Difference Example



Consider the following relation

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

List all cities where there is a branch office but no properties for rent.

∏ city (Branch) - ∏ city(PropertyForRent)

Relational Calculus

An operational methodology, founded on predicate calculus, dealing with descripitive expressions that are equivalent to the operations of relational algebra. Codd's reduction algorithm can convert from relational calculus to relational algebra. Two forms of the relational calculus exist: the tuple calculus and the domain calculus. Codd proposed the concept of a relational calculus (applied predicate calculus tailored to relational databases).

Why it is called relational calculus?

It is founded on a branch of mathematical logic called the predicate calculus. Relational calculus is a formal query language where we write one **declarative** expression to specify a retrieval request and hence there is no description of how to evaluate a query; a calculus expression specifies *what* is to be retrieved rather than *how* to retrieve it. Therefore, the relational calculus is considered to be a **nonprocedural** language. This differs from relational algebra, where we must write a *sequence of operations* to specify a retrieval request; hence it can be considered as a **procedural** way of stating a query. It is possible to nest algebra operations to form a single expression; however, a certain order among the operations is always explicitly specified in a relational algebra expression. This order also influences the strategy for evaluating the query.

It has been shown that any retrieval that can be specified in the relational algebra can also be specified in the relational calculus, and vice versa; in other words, the **expressive power** of the two languages is *identical*. This has led to the definition of the concept of a relationally complete language. A relational query language L is considered **relationally complete** if we can express in L any query that can be expressed in relational calculus. Relational completeness has become an important basis for comparing the expressive power of high-level query languages. However certain frequently required queries in database applications cannot be expressed in relational algebra or calculus. Most relational query languages are relationally complete but have *more expressive power* than relational algebra or relational calculus because of additional operations such as aggregate functions, grouping, and ordering.

Tuple Calculus

The tuple calculus is a calculus that was introduced by Edgar F. Codd as part of the relational model in order to give a declarative database query language for this data model. It formed the inspiration for the database query languages QUEL and SQL of which the latter, although far less faithful to the original relational model and calculus, is now used in almost all relational database management systems as the ad-hoc query language. Along with the tuple calculus Codd also introduced the domain calculus which is closer to first-order logic and showed that these two calculi (and the relational algebra) are equivalent in expressive power.The SQL language is based on the tuple relational calculus which in turn is a subset of classical predicate logic. Queries in the TRC all have the form:

{QueryTarget | QueryCondition}

The QueryTarget is a tuple variable which ranges over tuples of values. The QueryCondition is a logical expression such that

- It uses the QueryTarget and possibly some other variables.
- If a concrete tuple of values is substituted for each occurence of the QueryTarget in QueryCondition, the condition evaluates to a boolean value of true or false.

The result of a TRC query with respect to a database instance is the set of all choices of values for the query variable that make the query condition a true statement about the database instance. The relation between the TRC and logic is in that the QueryCondition is a logical expression of classical first-order logic.

The tuple relational calculus is based on specifying a number of **tuple variables.** Each tuple variable usually **ranges over** a particular database relation, meaning that the variable may take as its value any individual tuple from that relation. A simple tuple relational calculus query is of the form

{t | COND(t)}

where t is a tuple variable and COND(t) is a conditional expression involving t. The result of such a query is the set of all tuples t that satisfy COND(t). For example, to find all employees whose salary is above $50,000, we can write the following tuple calculus expression:

{t | EMPLOYEE(t) **and** t.SALARY>50000}

The condition EMPLOYEE(t) specifies that the *range relation* of tuple variable t is EMPLOYEE. Each EMPLOYEE tuple t that satisfies the condition t.SALARY>50000 will be retrieved. Notice that t.SALARY references attribute SALARY of tuple variable t; this notation resembles how attribute names are qualified with relation names or aliases in SQL. The above query retrieves all attribute values for each selected EMPLOYEE tuple t. To retrieve only *some* of the attributes-say, the first and last names-we write

{t.FNAME, t.LNAME | EMPLOYEE(t) **and** t.SALARY>50000}

This is equivalent to the following SQL query:

Select T.FNAME, T.LNAME from EMPLOYEE AS T Where T.SALARY>50000;

Informally, we need to specify the following information in a tuple calculus expression:

1. For each tuple variable t, the **range relation** R of t. This value is specified by a condition of the form R(t).
2. A condition to select particular combinations of tuples. As tuple variables range over their respective range relations, the condition is evaluated for every possible combination of tuples to identify the **selected combinations** for which the condition evaluates to TRUE.
3. A set of attributes to be retrieved, the **requested attributes.** The values of these attributes are retrieved for each selected combination of tuples.

Observe the correspondence of the preceding items to a simple SQL query: item 1 corresponds to the FROM-clause relation names; item 2 corresponds to the WHERE-clause condition; and item 3 corresponds to the SELECT-clause attribute list. Before we discuss the formal syntax of tuple relational calculus, consider another query we have seen before.

Retrieve the birthdate and address of the employee (or employees) whose name is ‘John B. Smith’.

Q0 : {t.BDATE, t.ADDRESS | EMPLOYEE(t) **and** t.FNAME=‘John’ **and** t.MINIT=‘B’ **and** t.LNAME=‘Smith’}

In tuple relational calculus, we first specify the requested attributes t.BDATE and t.ADDRESS for each selected tuple t. Then we specify the condition for selecting a tuple following the bar (|)-namely, that t be a tuple of the EMPLOYEE relation whose FNAME, MINIT, and LNAME attribute values are ‘John’, ‘B’, and ‘Smith’, respectively.

Domain Calculus

There is another type of relational calculus called the domain relational calculus, or simply, **domain calculus.** The language QBE that is related to domain calculus was developed almost concurrently with SQL at IBM Research, Yorktown Heights. The formal specification of the domain calculus was proposed after the development of the QBE system.

The domain calculus differs from the tuple calculus in the *type of variables* used in formulas: rather than having variables range over tuples, the variables range over single values from domains of attributes. To form a relation of degree n for a query result,

we must have n of these **domain variables**-one for each attribute. An expression of the Domain calculus is of the form

$\{x_1, x_2, \ldots, x_n \mid \text{COND}(x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m})\}$

where $x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m}$ are domain variables that range over domains (of attributes) and COND is a **condition** or **formula** of the domain relational calculus. A formula is made up of **atoms**

As in tuple calculus, atoms evaluate to either TRUE or FALSE for a specific set of values, called the **truth values** of the atoms. In case 1, if the domain variables are assigned values corresponding to a tuple of the specified relation R, then the atom is TRUE. In cases 2 and 3, if the domain variables are assigned values that satisfy the condition, then the atom is TRUE.

In a similar way to the tuple relational calculus, formulas are made up of atoms, variables, and quantifiers, so we will not repeat the specifications for formulas here. Some examples of queries specified in the domain calculus follow. We will use lowercase letters l, m, n, . . ., x, y, z for domain variables.

Query 0

Retrieve the birthdate and address of the employee whose name is 'John B. Smith'.

Q0 : {uv | (∃q) (∃r) (∃s) (∃t) (∃w) (∃x) (∃y) (∃z) (EMPLOYEE(qrstuvwxyz) **and** q='John' **and** r='B' **and** s='Smith')}

We need ten variables for the EMPLOYEE relation, one to range over the domain of each attribute in order. Of the ten variables q, r, s, . . ., z, only u and v are free. We first specify the *requested attributes*, BDATE and ADDRESS, by the domain variables u for BDATE and v for ADDRESS. Then we specify the condition for selecting a tuple following the bar (|)-namely, that the sequence of values assigned to the variables qrstuvwxyz be a tuple of the EMPLOYEE relation and that the values for q (FNAME), r (MINIT), and s (LNAME) be 'John', 'B', and 'Smith', respectively. For convenience, we will quantify only those variables *actually appearing in a condition* (these would be q, r, and s in Q0) in the rest of our examples.

An alternative notation for writing this query is to assign the constants 'John', 'B', and 'Smith' directly as shown in Q0A, where all variables are free:

Q0A : {uv | EMPLOYEE('John','B','Smith',t,u,v,w,x,y,z) }

Query 1

Retrieve the name and address of all employees who work for the 'Research' department.

Q1 : {qsv | (∃z) (∃l) (∃m) (EMPLOYEE(qrstuvwxyz) **and** DEPARTMENT(lmno) **and** l='Research' **and** m=z)}

A condition relating two domain variables that range over attributes from two relations, such as m = z in Q1, is a **join condition;** whereas a condition that relates a domain variable to a constant, such as l = 'Research', is a **selection condition.**

Query 2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birthdate, and address.

Q2 : {iksuv | (∃j) (∃m)(∃n) (∃t)(PROJECT(hijk) **and** EMPLOYEE(qrstuvwxyz) **and** DEPARTMENT(lmno) **and** k=m **and** n=t **and** j='Stafford')}

Query 6

Find the names of employees who have no dependents.

Q6 : {qs | (∃t) (EMPLOYEE(qrstuvwxyz) **and** (**not**(∃l) (DEPENDENT(lmnop) **and** t=l)))}

Query 6 can be restated using universal quantifiers instead of the existential quantifiers, as shown in Q6A:

Q6A : {qs | (∃t) (EMPLOYEE(qrstuvwxyz) **and** ((∀l) (**not**(DEPENDENT(lmnop)) **or not**(t=l))))}

Query 7

List the names of managers who have at least one dependent.

Q7 : {sq | (∃t) (∃j) (∃l)(EMPLOYEE(qrstuvwxyz) **and** DEPARTMENT(hijk) **and** DEPENDENT(lmnop) **and** t=j **and** l=t)}

As we mentioned earlier, it can be shown that any query that can be expressed in the relational algebra can also be expressed in the domain or tuple relational calculus. Also, any safe expression in the domain or tuple relational calculus can be expressed in the relational algebra.

Review Questions

1. What do you mean by Relational Algebra?
2. Explain the Database Specific Relational Algebraic Expressions?
3. Explain the various set operations?
4. Explain Tuple and domain relational Calculus?

References

Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000

Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld.

Notes:

LESSON 8
DATA INTEGRITY

Hi! Here in this lecture we are going to discuss about the data integrity part of relational data model

Data Integrity

We noted at the beginning of the previous lecture the relational model has three main components; data structure, data manipulation, and data integrity. The aim of data integrity is to specify rules that implicitly or explicitly define a consistent database state or changes of state. These rules may include facilities like those provided by most programming languages for declaring data types which constrain the user from operations like comparing data of different data types and assigning a variable of one type to another of a different type. This is done to stop the user from doing things that generally do not make sense. In a DBMS, integrity constraints play a similar role.

The integrity constraints are necessary to avoid situations like the following:

1. Some data has been inserted in the database but it cannot be identified (that is, it is not clear which object or entity the data is about).
2. A student is enrolled in a course but no data about him is available in the relation that has information about students.
3. During a query processing, a student number is compared with a course number (this should never be required).
4. A student quits the university and is removed from the student relation but is still enrolled in a course.

Integrity constraints on a database may be divided into two types:

1. *Static Integrity Constraints* - these are constraints that define valid states of the data. These constraints include designations of primary keys etc.
2. *Dynamic Integrity Constraints* - these are constraints that define side-effects of various kinds of transactions (e.g. insertions and deletions).

We now discuss certain integrity features of the relational model. We discuss the following features:

1. Primary Keys
2. Domains
3. Foreign Keys and Referential Integrity
4. Nulls

Primary Keys

We have earlier defined the concept of candidate key and primary key. From the definition of candidate key, it should be clear that each relation must have at least one candidate key even if it is the combination of all the attributes in the relation since all tuples in a relation are distinct. Some relations may have more than one candidate keys.

As discussed earlier, the primary key of a relation is an arbitrarily but permanently selected candidate key. The primary key is important since it is the sole identifier for the tuples in a relation. Any tuple in a database may be identified by specifying relation name, primary key and its value. Also for a tuple to exist in a relation, it must be *identifiable* and therefore it must have a primary key. The relational data model therefore imposes the following two integrity constraints:

a. No component of a primary key value can be null;

b. Attempts to change the value of a primary key must be carefully controlled.

The first constraint is necessary because if we want to store information about some entity, then we must be able to identify it, otherwise difficulties are likely to arise. For example, if a relation

CLASS (STUNO, LECTURER, CNO)

has (STUNO, LECTURER) as the primary key then allowing tuples like

3123 NULL CP302
NULL SMITH CP302

is going to lead to ambiguity since the two tuples above may or may not be identical and the integrity of the database may be compromised. Unfortunately most commercial database systems do not support the concept of primary key and it would be possible to have a database state when integrity of the database is violated.

The second constraint above deals with changing of primary key values. Since the primary key is the tuple identifier, changing it needs very careful controls. Codd has suggested three possible approaches:

Method 1

Only a select group of users be authorised to change primary key values.

Method 2

Updates on primary key values be banned. If it was necessary to change a primary key, the tuple would first be deleted and then a new tuple with new primary key value but same other values would be inserted. Of course, this does require that the old values of attributes be remembered and be reinserted in the database.

Method 3

A different command for updating primary keys be made available. Making a distinction in altering the primary key and another attribute of a relation would remind users that care needs to be taken in updating primary keys.

Advantages and disadvantages of each to be discussed.

Domains

We have noted earlier that many commercial database systems do not provide facilities for specifying domains. Domains could be specified as below:

```
create  DOMAIN  NAME1  CHAR(10)
create  DOMAIN  STUNO  INTEGER
create  DOMAIN  NAME2  CHAR(10)
etc.
```

Note that *NAME1* and *NAME2* are both character strings of length 10 but they now belong to different (semantic) domains. It is important to denote different domainsto

a. Constrain unions, intersections, differences, and equijoins of relations.
b. Let the system check if two occurrences of the same database value denote the same real world object.

The constrain on union-compatibility and join-compatibility is important so that only those operations that make sense are permitted. For example, a join on class number and student number would make no sense even if both attributes are integers and the user should not be permitted to carry out such operations (or at least be warned when it is attempted).

Domain Constraints

All the values that appear in a column of a relation must be taken from the same domain. A domain usually consists of the following components.

1. Domain Name
2. Meaning
3. Data Type
4. Size or length
5. Allowable values or Allowable range(if applicable)

Entity Integrity

The Entity Integrity rule is so designed to assure that every relation has a primary key and that the data values for the primary key are all valid. Entity integrity guarantees that every primary key attribute is non null. No attribute participating in the primary key of a base relation is allowed to contain nulls. Primary key performs unique identification function in a relational model. Thus a null primary key performs the unique identification function in a relation would be like saying that there are some entity that had no known identity. An entity that cannot be identified is a contradiction in terms, hence the name entity integrity.

Operational Constraints

These are the constraints enforced in the database by the business rules or real world limitations. For example if the retirement age of the employees in a organization is 60, then the age column of the employee table can have a constraint "Age should be less than or equal to 60". These kinds of constraints enforced by the business and the environment are called operational constraints.

Null constraints

NOT NULL constraint restricts attributes to not allow NULL values.

NULL is a special value:

Many possible interpretations: value unknown, value inapplicable,

Value withheld, etc.

Often used as the default value

Example:

INSERT INTO Student VALUES (135, 'Maggie', NULL, NULL); or

INSERT INTO Student (SID, name) VALUES (135, 'Maggie');

Foreign Keys and Referential Integrity

We have earlier considered some problems related to modifying primary key values. Additional problems arise because primary key value of a tuple may be referred in many relations of the database. When a primary key is modified, each of these references to a primary key must either be modified accordingly or be replaced by NULL values. Only then we can maintain referential integrity.

Before we discuss referential integrity further, we define the concept of a *foreign key*. The concept is important since a relational database consists of relations only (no pointers) and relationships between the relations are implicit, based on references to primary keys of o

We now define foreign key. ***A foreign key in a relation R is a set of attributes whose values are required to match those of the primary key of some relation S.***

In the following relation the supervisor number is a foreign key (it is the primary key of *employee*)

employee (empno, empname, supervisor-no, dept)

In the following relation *Class*, *student-num* and *lecturer-num* are foreign keys since they appear as primary keys in other relations (relations student and lecturer).

Class (student-num, lecturer-num, subject)
student ( )
lecturer ( )

Foreign keys are the implicit references in a relational database. The following constraint is called referential integrity constraint:

If a foreign key F in relation R matches the primary key P of relation S than every value of F must either be equal to a value of P or be wholly null.

The justification for referential integrity constraint is simple. If there is a foreign key in a relation (that is if the relation is referring to another relation) then its value must match with one of the primary key values to which it refers. That is, if an object or entity is being referred to, the constraint ensures the referred object or entity exists in the database.

In the relational model the association between the tables is defined using foreign keys. The association between the SHIPMENT and ELEMENT tables is defined by including the Symbol attribute as a foreign key in the SHIPMENT table. This implies that before we insert a row in the SHIPMENT table,

the element for that order must already exist in the ELEMENT table.

A referential integrity constraint is a rule that maintains consistency among the rows of two tables or relations. The rule states that if there is a foreign key in one relation, either each of the foreign key value must match a primary key value in the other table or else the foreign key value must be null.

When Should Constraints Be Checked?

- Usually they are checked for each modification statement.
- But sometimes deferred constraint checking is necessary.

Review Questions

1. Define Data Integrity
2. What is a primary key
3. Define referential integrity
4. Define a foreign key
5. Define domains constraints

1. **Codd, E. F. (1974), "A Relational Model of Data for Large Shared Data Banks", Comm. ACM, Vol. 13, No. 6, pp. 377-387.**
2. **Codd, E. F. (1974), "Recent Investigations in Relational Data Base Systems", in Information Processing 74, North Holland.**
3. **Codd, E. F. (1982), "Relational Database: A Practical Foundation for Productivity", in Comm. ACM, Vol. 25, No. 2, pp. 109-117.**
4. **Codd, E. F. (1981), "The Capabilities of Relational Database Management Systems", IBM Report RJ3132.**
5. **Codd, E. F. (1971), "Relational Completeness of Data Base Sublanguages" Courant Computer Science Symposium 6, Data Base Systems, Prentice-Hall.**
6. **Codd, E. F. (1971), "Further Normalization of the Data Base Relational Model", Courant Computer Science Symposium 6, Data Base Systems, Prentice-Hall.**
7. **C. J. Date (1987) "A Guide to the SQL Standard", Addison Wesley.**

Summary

The explosion of information made available to enterprise applications by the broad-based adoption of Internet standards and technologies has introduced a clear need for an information integration platform to help harness that information and make it available to enterprise applications. The challenges for a robust information integration platform are steep. However, the foundation to build such a platform is already on the market. DBMSs have demonstrated over the years a remarkable ability to manage and harness structured data, to scale with business growth, and to quickly adapt to new requirements. We believe that a federated DBMS enhanced with native XML capabilities and tightly coupled enterprise application services, content management services and analytics is the right technology to provide a robust end-to-end solution.

Notes:

LESSON 9
FUNCTIONAL DEPENDENCIES

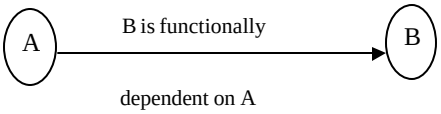
Hi! We are going to discuss functional Dependencies.

For our discussion on functional dependencies assume that a relational schema has attributes (A, B, C... Z) and that the whole database is described by a single universal relation called R = (A, B, C, ..., Z). This assumption means that every attribute in the database has a unique name.

What is functional dependency in a relation?

A ***functional dependency*** is a property of the semantics of the attributes in a relation. The semantics indicate how attributes relate to one another, and specify the functional dependencies between attributes. When a functional dependency is present, the dependency is specified as a constraint between the attributes.

Consider a relation with attributes A and B, where attribute B is functionally dependent on attribute A. If we know the value of A and we examine the relation that holds this dependency, we will find only one value of B in all of the tuples that have a given value of A, at any moment in time. Note however, that for a given value of B there may be several different values of A.



In the figure above, A is the determinant of B and B is the consequent of A.

The determinant of a functional dependency is the attribute or group of attributes on the left-hand side of the arrow in the functional dependency. The consequent of a fd is the attribute or group of attributes on the right-hand side of the arrow.

Identifying Functional Dependencies

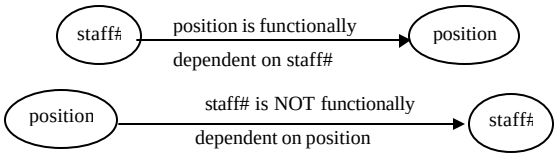
Now let us consider the following Relational schema

Staffbranch

staff#	sname	position	salary	branch#	baddress
SL21	Kristy	manager	30000	B005	22Deer Road
SG37	Debris	assistant	12000	B003	162Main Street
SG14	Alan	supervisor	18000	B003	163Main Street
SA9	Traci	assistant	12000	B007	375Fox Avenue
SG5	David	manager	24000	B003	163Main Street

The functional dependency staff# → position clearly holds on this relation instance. However, the reverse functional dependency position → staff# clearly does not hold.

The relationship between staff# and position is 1:1 – for each staff member there is only one position. On the other hand, the relationship between position and staff# is 1:M – there are several staff numbers associated with a given position.



For the purposes of normalization we are interested in identifying functional dependencies between attributes of a relation that have a 1:1 relationship.

When identifying Fds between attributes in a relation it is important to distinguish clearly between the values held by an attribute at a given point in time and the *set of all possible values* that an attributes may hold at different times.

In other words, ***a functional dependency is a property of a relational schema (its intension) and not a property of a particular instance of the schema (extension).***

The reason that we need to identify Fds that hold for all possible values for attributes of a relation is that these represent the types of integrity constraints that we need to identify. Such constraints indicate the limitations on the values that a relation can legitimately assume. In other words, they identify the legal instances which are possible.

Let's identify the functional dependencies that hold using the relation schema STAFFBRANCH

In order to identify the time invariant Fds, we need to clearly understand the semantics of the various attributes in each of the relation schemas in question.

For example, if we know that a staff member's position and the branch at which they are located determines their salary. There is no way of knowing this constraint unless you are familiar with the enterprise, but this is what the requirements analysis phase and the conceptual design phase are all about!

staff# → sname, position, salary, branch#, baddress

branch# → baddress

baddress → branch#

branch#, position → salary

baddress, position → salary

Trivial Functional Dependencies

As well as identifying Fds which hold for all possible values of the attributes involved in the fd, we also want to ignore trivial functional dependencies. A functional dependency is trivial if, the consequent is a subset of the determinant. In other words, it is impossible for it *not* to be satisfied.

Example: Using the relation instances on page 6, the trivial dependencies include:

{ staff#, sname} → sname

{ staff#, sname} → staff#

Although trivial Fds are valid, they offer no additional information about integrity constraints for the relation. As far as normalization is concerned, trivial Fds are ignored.

Inference Rules for Functional Dependencies

We'll denote as F, the set of functional dependencies that are specified on a relational schema R.

Typically, the schema designer specifies the Fds that are *semantically obvious*; usually however, numerous other Fds hold in all legal relation instances that satisfy the dependencies in F.

These additional Fds that hold are those Fds which can be *inferred* or *deduced* from the Fds in F.

The set of all functional dependencies implied by a set of functional dependencies F is called the closure of F and is denoted F+.

The notation: F $\rightarrow$ X $\rightarrow$ Y denotes that the functional dependency X → Y is implied by the set of Fds F.

Formally, F+ º {X → Y | F →Y}

A set of inference rules is required to infer the set of Fds in F+.

For example, if I tell you that Kristi is older than Debi and that Debi is older than Traci, you are able to infer that Kristi is older than Traci. How did you make this inference? Without thinking about it or maybe knowing about it, you utilized a transitivity rule to allow you to make this inference. The set of all Fds that are implied by a given set S of Fds is called the closure of S, written S+. Clearly we need an algorithm that will allow us to compute S+ from S. You know the first attack on this problem appeared in a paper by Armstrong which gives a set of inference rules. The following are the six well-known inference rules that apply to functional dependencies.

IR1: reflexive rule – if X Ê Y, then X ® Y

IR2: augmentation rule – if X ® Y, then XZ ® YZ

IR3: transitive rule – if X ® Y and Y ® Z, then X ® Z

IR4: projection rule – if X ® YZ, then X ® Y and X ® Z

IR5: additive rule – if X ® Y and X ® Z, then X ® YZ

IR6: pseudo transitive rule – if X ® Y and YZ ® W, then XZ ® W

The first three of these rules (IR1-IR3) are known as Armstrong's Axioms and constitute a necessary and sufficient set of inference rules for generating the closure of a set of functional dependencies. These rules can be stated in a variety of equivalent ways. Each of these rules can be directly proved from the definition of functional dependency. Moreover the rules are complete, in the sense that, given a set S of Fds, all Fds implied by S can be derived from S using the rules. The other rules are derived from these three rules.

Given R = (A,B,C,D,E,F,G,H, I, J) and

F = {AB → E, AG → J, BE → I, E → G, GI → H}

Does F AB →GH?

Proof

1. AB → E, given in F
2. AB → AB, reflexive rule IR1
3. AB → B, projective rule IR4 from step 2
4. AB → BE, additive rule IR5 from steps 1 and 3
5. BE → I, given in F
6. AB → I, transitive rule IR3 from steps 4 and 5
7. E → G, given in F
8. AB → G, transitive rule IR3 from steps 1 and 7
9. AB → GI, additive rule IR5 from steps 6 and 8
10. GI → H, given in F
11. AB → H, transitive rule IR3 from steps 9 and 10
12. AB → GH, additive rule IR5 from steps 8 and 11 - proven

Irreducible sets of Dependencies

Let S1 and S2 be two sets of Fds, if every FD implied by S1 is implied by S2- i.e.; if S1+ is a subset of S2+-we say that S2 is a cover for S1+(Cover here means equivalent set). What this means that if the DBMS enforces the Fds in S2, then it will automatically be enforcing the Fds in S1.

Next

Next if S2 is a cover for S1 and S1 is a cover for S2- i.e.; if S1+=S2+ -we say that S1 and S2 are equivalent, clearly, if s1 and S2 are equivalent, then if the DBMS enforces the Fds in S2 it will automatically be enforcing the Fds in S1, And vice versa.

Now we define a set of Fds to be irreducible(Usually called minimal in the literature) if and only if it satisfies the following three properties

1. The right hand side (the dependent) of every Fds in S involves just one attribute (that is, it is singleton set)
2. The left hand side (determinant) of every in S is irreducible in turn-meaning that no attribute can be discarded from the determinant without changing the closure S+(that is, with out converting S into some set not equivalent to S). We will say that such an Fd is ***left irreducible.***
3. No Fd in S can be discarded from S without changing the closure S+(That is, without converting s into some set not equivalent to S)

Now we will work out the things in detail.

Relation R {A,B,C,D,E,F} satisfies the following Fds

AB → C

C → A

BC →D

ACD → B

BE → C

CE → FA

CF → VD

D → EF

Find an irreducible equivalent for this set of Fds?

Puzzled! The solution is simple. Let us find the solution for the above.

1. AB → C
2. C → A
3. BC → D
4. ACD→ B
5. BE → C
6. CE → A
7. CE→ F
8. CF → B
9. CF→ D
10. D → E
11. D → F

Now:

- 2 implies 6, so we can drop 6
- 8 implies CF →BC (By augmentation), by which 3 implies CF →D (By Transitivity), so we can drop 10.
- 8 implies ACF →AB (By augmentation), and 11 implies ACD →ACF (By augmentation), and so ACD →AB (By Transitivity), and so ACD →B(By Decomposition), so we can drop 4

No further reductions are possible, and so we are left with the following irreducible set:

AB → C

C → A

BC → D

BE → C

CE → F

CF → B

D → E

D → F

Alternatively:

- 2 implies CD→ACD (By Composition), which with 4 implies CD →BE (By Transitivity), so we can replace 4 CD →B
- 2 implies 6, so we can drop 6(as before)
- 2 and 10 implies CF → AD (By composition), which implies CF → ADC (By Augmentation), which with (the original) 4 implies CF → B(By Transitivity), So we can drop 8.

No further reductions are possible, and so we are left with following irreducible set:

AB → C

C → A

BC → D

CD → B

BE → C

CE → F

CF→D

D → E

D → F

Observe, therefore, that there are two distinct irreducible equivalence for the original set of Fds.

Review Questions

1. Define functional dependencies?
2. Explain the inference rules?
3. What does it mean to say that Armstrong's inference rules are sound? Complete?
4. Prove the reflexity, augmentation, transitivity rules, assuming only the basic definition of functional dependence?
5. List all the Fds satisfied by the STAFFBRANCH relation?

Activities

Find out the relation of Darwen's "General Unification Theorem" with the inference rules

Relation R{A,B,C,D,E,F,G,H,I,J) satisfies the following Fds:

ABD →E

AB → G

B → F

C → J

CJ → I

G → H

Is this an irreducible set? What are the candidate keys?

References

1. Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000
2. Elamasri R . and Navathe, S., Fundamentals of Database Systems (3rd Edition), Pearsson Education, 2000.
3. http://www.cs.ucf.edu/courses/cop4710/spr2004

LESSON 10
CONCEPT OF REDUNDANCY (UPDATION ANOMALIES)

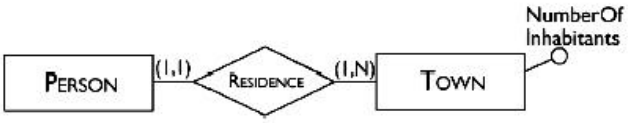
Concept Of redundancy (Updation Anomalies)

Hi! We are going to discuss one of the fascinating and important topics in a DBMS.

Analysis of Redundancies

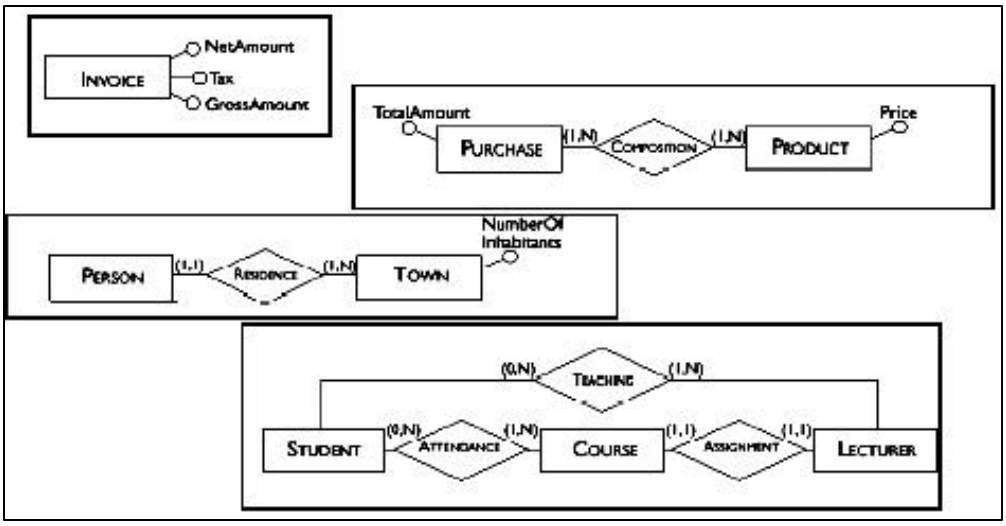
Before we go in to the detail of Normalization I would like to discuss with you the redundancies in the databases.

A redundancy in a conceptual schema corresponds to a piece of information that can be derived (that is, obtained through a series of retrieval operations) from other data in the database.



In this schema the attribute `NumberOfInhabitants` *is redundant.*

Examples of Redundancies



Deciding About Redundancies

The presence of a redundancy in a database may be decided upon the following factores

- **An advantage**: a reduction in the number of accesses necessary to obtain the derived information;
- **Adisadvantage**: because of larger storage requirements, (but, usually. At negligible cost) **and the necessity to carry out additional operations in order to keep the derived data consistent**.

The decision to maintain or delete a redundancy is made by comparing the cost of operations that involve the redundant information and the storage needed, in the case of presence or absence of redundancy.

Cost Comparison: An Example

Now we will see the impact of redundancy with the help of an example.

Load and Operations for the Example

Table of volumes

Concept	Type	Volume
Town	E	200
Person	E	1000000
Residence	R	1000000

Table of operations

Operation	Type	Frequency
Operation 1	I	500 per day
Operation 2	I	2 per day

- ***Operation 1****: add a new person with the person's town of residence.*
- ***Operation 2****: print all the data of a town (including the number of inhabitants).*

Table of Accesses, with Redundancy

Operation 1

Concept	Type	Accesses	Type
Person	Entity	1	W
Residence	Relationship	1	W
Town	Entity	1	W

Operation 2

Concept	Type	Accesses	Type
Town	Entity	1	R

Issues related to Redundancies (Anomalies)

The time has come to reveal the actual facts why normalization is needed. We will look in to the matter in detail now.

The serious problem with using the relations is the problem of update anomalies. These can be classified in to

- Insertion anomalies
- Deletion anomalies
- Modification anomalies

Insertion Anomalies

An ***"insertion anomaly"*** is a failure to place information about a new database entry into all the places in the database where information about that new entry needs to be stored. In a properly normalized database, information about a new entry needs to be inserted into only one place in the database; in an inadequately normalized database, information about a new entry may need to be inserted into more than one place and, human fallibility being what it is, some of the needed additional insertions may be missed.

This can be differentiated in to two types based on the following example

Emp_Dept

EName	SSN	BDate	Address	DNumber	DName	DMGRSSN
Smith	123456789	1965-01-09	Kandivly	5	Research	333445555
Rajeev	333445555	1955-12-08	Vashi	5	Research	333445555
Greta	999887777	1968-07-19	Sion	4	Admin	987654321
Rajesh	987654321	1941-06-20	Dadar	4	Admin	987654321

First Instance: - To insert a new employee tuple in to Emp_Dept table, we must include either the attribute values for the department that the employee works for, or nulls (if the employee does not work for a department as yet). For example to insert a new tuple for an employee who works in department no 5, we must enter the attribute values of department number 5correctly so that they are *consistent*, with values for the department 5 in other tuples in emp_dept.

Second Instance: - It is difficult to insert a new department that has no employees as yet in the emp_dept relation. The only way to do this is to place null values in the attributes for the employee this causes a problem because SSN in the primary key of emp_dept table and each tuple is supposed to represent an employee entity- not a department entity.

Moreover, when the first employee is assigned to that department, we do not need this tuple with null values anymore.

Deletion Anomalies

A ***"deletion anomaly"*** is a failure to remove information about an existing database entry when it is time to remove that entry. In a properly normalized database, information about an old, to-be-gotten-rid-of entry needs to be deleted from only one place in the database; in an inadequately normalized database, information about that old entry may need to be deleted from more than one place, and, human fallibility being what it is, some of the needed additional deletions may be missed.

The problem of deletion anomaly is related to the second insertion anomaly situation which we have discussed earlier, if we delete from emp_dept an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database.

Modification Anomalies

In Emp_Dept, if we change the value of one of the attribute of a particular department- say, the manager of department 5- we must update the tuples of all employees who work in that department; other wise, the database will become inconsistent. If we fail to update some tuples, the same department will be shown to have 2 different values for manager in different employee tuple which would be wrong.

All three kinds of anomalies are highly undesirable, since their occurrence constitutes corruption of the database. Properly normalized databases are much less susceptible to corruption than are unnormalized databases.

Update Anomalies - Redundant information not only wastes storage but makes updates more difficult since, for example, changing the name of the instructor of CP302 would require that all tuples containing CP302 enrolment information be updated. If for some reason, all tuples are not updated, we might have a database that gives two names of instructor for subject CP302. This difficulty is called the *update anomaly*.

Insertional Anomalies *- Inability to represent certain information* - Let the primary key of the above relation be *(sno, cno)*. Any new tuple to be inserted in the relation must have a value for the primary key since existential integrity requires that a key may not be totally or partially NULL. However, if one wanted to insert the number and name of a new course in the database, it would not be possible until a student enrols in the course and we are able to insert values of *sno* and *cno*. Similarly information about a new student cannot be inserted in the database until the student enrols in a subject. These difficulties are called *insertion anomalies.*

Deletion Anomalies - Loss of Useful Information - In some instances, useful information may be lost when a tuple is deleted. For example, if we delete the tuple corresponding to student 85001 doing CP304, we will loose relevant information about course CP304 (viz. course name, instructor, office number) if the student 85001 was the only student enrolled in that course. Similarly deletion of course CP302 from the database may remove all information about the student named Jones. This is called *deletion anomalies*.

Review Questions

1. Explain insertion anomaly
2. Explain deletion and modification anomalies
3. what are the factors which decide the redundancy in a data base

Notes:

LESSON 11
NORMALIZATION-PART I

Hi! Here in this lecture we are going to discuss about the important concepts of DBMS, Normalization.

What is Normalization?

Yes, but what is this normalization all about? If I am simply putting it, normalization is a formal process for determining which fields belong in which tables in a relational database. Normalization follows a set of rules worked out at the time relational databases were born. A normalized relational database provides several benefits:

- Elimination of redundant data storage.
- Close modeling of real world entities, processes, and their relationships.
- Structuring of data so that the model is flexible.
- Normalization ensures that you get the benefits relational databases offer. Time spent learning about normalization will begin paying for itself immediately.

Why do they talk like that?

Some people are intimidated by the language of normalization. Here is a quote from a classic text on relational database design:

A relation is in third normal form (3NF) if and only if it is in 2NF and every nonkey attribute is no transitively dependent on the primary key.

Huh? Relational database theory, and the principles of normalization, was first constructed by people intimately acquainted with set theory and predicate calculus. They wrote about databases for like-minded people. Because of this, people sometimes think that normalization is “hard”. Nothing could be more untrue. The principles of normalization are simple, commonsense ideas that are easy to apply.

Design Versus Implementation

Now we will look in to the aspects regarding the tasks associated with designing and implementing a database.

Designing a database structure and implementing a database structure are different tasks. When you design a structure it should be described without reference to the specific database tool you will use to implement the system, or what concessions you plan to make for performance reasons. These steps come later. After you’ve designed the database structure abstractly, then you implement it in a particular environment-4D in our case. Too often people new to database design combine design and implementation in one step. 4D makes this tempting because the structure editor is so easy to use. Implementing a structure without designing it quickly leads to flawed structures that are difficult and costly to modify. Design first, implement second, and you’ll finish faster and cheaper.

Normalized Design: Pros and Cons

Oh, now we’ve implied that there are various advantages to producing a properly normalized design before you implement your system. Let’s look at a detailed list of the pros and cons:

Pros of Normalizing	Cons of Normalizing
More efficient database structure.	You can't start building the database before you know what the user needs.
Better understanding of your data.	
More flexible database structure.	
Easier to maintain database structure.	
Few (if any) costly surprises down the road.	
Validates your common sense and intuition.	
Avoids redundant fields.	
Ensures that distinct tables exist when necessary.	

We think that the pros outweigh the cons.

Terminology

There are a couple terms that are central to a discussion of normalization: “key” and “dependency”. These are probably familiar concepts to anyone who has built relational database systems, though they may not be using these words. We define and discuss them here as necessary background for the discussion of normal forms that follows.

The Normal Forms

Hi! Now we are going to discuss the definitions of Normal Forms.

1st Normal Form (1NF)

Def: **A table (relation) is in 1NF if**

1. There are no duplicated rows in the table.
2. Each cell is single-valued (i.e., there are no repeating groups or arrays).
3. Entries in a column (attribute, field) are of the same kind.

Note: The order of the rows is immaterial; the order of the columns is immaterial. The requirement that there be no duplicated rows in the table means that the table has a key (although the key might be made up of more than one column - even, possibly, of all the columns).

So we come to the conclusion,

A relation is in 1NF if and only if all underlying domains contain atomic values only.

The first normal form deals only with the basic structure of the relation and does not resolve the problems of redundant information or the anomalies discussed earlier. All relations discussed in these notes are in 1NF.

For example consider the following example relation:

student(sno, sname, dob)
Add some other attributes so it has anomalies and is not in 2NF

The attribute *dob* is the date of birth and the primary key of the relation is *sno* with the functional dependencies *sno -> sname* and $sno \rightarrow dob$. The relation is in 1NF as long as dob is considered an atomic value and not consisting of three components *(day, month, year)*. The above relation of course suffers from all the anomalies that we have discussed earlier and needs to be normalized. (add example with date of birth)

A relation is in first normal form if and only if, in every legal value of that relation every tuple contains one value for each attribute

The above definition merely states that the relations are always in first normal form which is always correct. However the relation that is only in first normal form has a structure those undesirable for a number of reasons.

First normal form (1NF) sets the very basic rules for an organized database:

- Eliminate duplicative columns from the same table.
- Create separate tables for each group of related data and identify each row with a unique column or set of columns (the primary key).

2nd Normal Form (2NF)

Def: A table is in 2NF if it is in 1NF and if all non-key attributes are dependent on the entire key.

The second normal form attempts to deal with the problems that are identified with the relation above that is in 1NF. The aim of second normal form is to ensure that all information in one relation is only about one thing.

A relation is in 2NF if it is in 1NF and every non-key attribute is fully dependent on each candidate key of the relation.

Note: Since a partial dependency occurs when a non-key attribute is dependent on only a part of the (composite) key, the definition of 2NF is sometimes phrased as, "A table is in 2NF if it is in 1NF and if it has no partial dependencies."

Recall the general requirements of 2NF:

- Remove subsets of data that apply to multiple rows of a table and place them in separate rows.
- Create relationships between these new tables and their predecessors through the use of foreign keys.

These rules can be summarized in a simple statement: 2NF attempts to reduce the amount of redundant data in a table by extracting it, placing it in new table(s) and creating relationships between those tables.

Let's look at an example. Imagine an online store that maintains customer information in a database. Their Customers table might look something like this:

CustNum	FirstName	LastName	Address	City	State	ZIP
1	John	Doe	12 Main Street	Sea Cliff	NY	11579
2	Alan	Johnson	82 Evergreen Tr	Sea Cliff	NY	11579
3	Beth	Thompson	1912 NE 1st St	Miami	FL	33157
4	Jacob	Smith	142 Irish Way	South Bend	IN	46637
5	Sue	Ryan	412 NE 1st St	Miami	FL	33157

A brief look at this table reveals a small amount of redundant data. We're storing the "Sea Cliff, NY 11579" and "Miami, FL 33157" entries twice each. Now, that might not seem like too much added storage in our simple example, but imagine the wasted space if we had thousands of rows in our table. Additionally, if the ZIP code for Sea Cliff were to change, we'd need to make that change in many places throughout the database.

In a 2NF-compliant database structure, this redundant information is extracted and stored in a separate table. Our new table (let's call it ZIPs) might look like this:

ZIP	City	State
11579	Sea Cliff	NY
33157	Miami	FL
46637	South Bend	IN

If we want to be super-efficient, we can even fill this table in advance - the post office provides a directory of all valid ZIP codes and their city/state relationships. Surely, you've encountered a situation where this type of database was utilized. Someone taking an order might have asked you for your ZIP code first and then knew the city and state you were calling from. This type of arrangement reduces operator error and increases efficiency.

Now that we've removed the duplicative data from the Customers table, we've satisfied the first rule of second normal form. We still need to use a foreign key to tie the two tables together. We'll use the ZIP code (the primary key from the ZIPs table) to create that relationship. Here's our new Customers table:

CustNum	FirstName	LastName	Address	ZIP
1	John	Doe	12 Main Street	11579
2	Alan	Johnson	82 Evergreen Tr	11579
3	Beth	Thompson	1912 NE 1st St	33157
4	Jacob	Smith	142 Irish Way	46637
5	Sue	Ryan	412 NE 1st St	33157

We've now minimized the amount of redundant information stored within the database and our structure is in second normal form, great isn't it?

Let's take one more example to confirm the thoughts

The concept of 2NF requires that all attributes that are not part of a candidate key be fully dependent on each candidate key. If we consider the relation

student (sno, sname, cno, cname)

and the functional dependencies

cno -> cname

and assume that *(sno, cno)* is the only candidate key (and therefore the primary key), the relation is not in 2NF since *sname* and *cname* are not fully dependent on the key. The above relation suffers from the same anomalies and repetition of information as discussed above since sname and cname will be repeated. To resolve these difficulties we could remove those attributes from the relation that are not fully dependent on the candidate keys of the relations. Therefore we decompose the relation into the following projections of the original relation:

S1 *(sno, sname)*
S2 (*cno, cname)*
SC *(sno, cno)*

Use an example that leaves one relation in 2NF but not in 3NF. We may recover the original relation by taking the natural join of the three relations. If however we assume that sname and cname are unique and therefore we have the following candidate keys

(sno, cno)
(sno, cname)
(sname, cno)
(sname, cname)

The above relation is now in 2NF since the relation has no non-key attributes. The relation still has the same problems as before but it then does satisfy the requirements of 2NF. Higher level normalization is needed to resolve such problems with relations that are in 2NF and further normalization will result in decomposition of such relations

3rd Normal Form (3NF)
Def: A table is in 3NF if it is in 2NF and if it has no transitive dependencies.

The basic requirements of 3NF are as follows

- Meet the requirements of 1NF and 2NF
- Remove columns that are not fully dependent upon the primary key.

Although transforming a relation that is not in 2NF into a number of relations that are in 2NF removes many of the anomalies that appear in the relation that was not in 2NF, not all anomalies are removed and further normalization is sometime needed to ensure further removal of anomalies. These anomalies arise because a 2NF relation may have attributes that are not directly related to the thing that is being described by the candidate keys of the relation. Let us first define the 3NF.

A relation R is in third normal form if it is in 2NF and every non-key attribute of R is non-transitively dependent on each candidate key of R.

To understand the third normal form, we need to define transitive *dependence* which is based on one of Armstrong's axioms. Let A, B and C be three attributes of a relation R such that $A \rightarrow B$ and $B \rightarrow C$. From these FDs, we may derive $A \rightarrow C$. As noted earlier, this dependence $A \rightarrow C$ is transitive. The 3NF differs from the 2NF in that all non-key attributes in 3NF are required to be directly dependent on each candidate key of the relation. The 3NF therefore insists, in the words of Kent (1983) that all facts in the relation are about the key (or the thing that the key identifies), the whole key and nothing but the key. If some attributes are dependent on the keys transitively then that is an indication that those attributes provide information not about the key but about a kno-key attribute. So the information is not directly about the key, although it obviously is related to the key.

Consider the following relation

subject (cno, cname, instructor, office)

Assume that *cname* is not unique and therefore *cno* is the only candidate key. The following functional dependencies exist

$$cno \rightarrow cname$$
$$cno \rightarrow instructor$$
$$instructor \rightarrow office$$

We can derive $cno \rightarrow office$ from the above functional dependencies and therefore the above relation is in 2NF. The relation is however not in 3NF since office is not directly dependent on *cno*. This transitive dependence is an indication that the relation has information about more than one thing (viz. course and instructor) and should therefore be decomposed. The primary difficulty with the above relation is that an instructor might be responsible for several subjects and therefore his office address may need to be repeated many times. This leads to all the problems that we identified at the beginning of this chapter. To overcome these difficulties we need to decompose the above relation in the following two relations:

s (cno, cname, instructor)
ins (instructor, office)

s is now in 3NF and so is ins.

An alternate decomposition of the relation subject is possible:

s(cno, cname)
inst(instructor, office)
si(cno, instructor)

The decomposition into three relations is not necessary since the original relation is based on the assumption of one instructor for each course.

The 3NF is usually quite adequate for most relational database designs. There are however some situations, for example the relation student(sno, sname, cno, cname) discussed in 2NF above, where 3NF may not eliminate all the redundancies and inconsistencies. The problem with the relation student(sno, sname, cno, cname) is because of the redundant information in the candidate keys. These are resolved by further normalization using the BCNF.

Imagine that we have a table of widget orders:

Order Number	Customer Number	Unit Price	Quantity	Total
1	241	$10	2	$20
2	842	$9	20	$180
3	919	$19	1	$19
4	919	$12	10	$120

Remember, our first requirement is that the table must satisfy the requirements of 1NF and 2NF. Are there any duplicative columns? No. Do we have a primary key? Yes, the order number. Therefore, we satisfy the requirements of 1NF. Are there any subsets of data that apply to multiple rows? No, so we also satisfy the requirements of 2NF.

Now, are all of the columns fully dependent upon the primary key? The customer number varies with the order number and it doesn't appear to depend upon any of the other fields. What about the unit price? This field could be dependent upon the customer number in a situation where we charged each customer a set price. However, looking at the data above, it appears we sometimes charge the same customer different prices. Therefore, the unit price is fully dependent upon the order number. The quantity of items also varies from order to order, so we're OK there.

What about the total? It looks like we might be in trouble here. The total can be derived by multiplying the unit price by the quantity; therefore it's not fully dependent upon the primary key. We must remove it from the table to comply with the third normal form:

Order Number	Customer Number	Unit Price	Quantity
1	241	$10	2
2	842	$9	20
3	919	$19	1
4	919	$12	10

Now our table is in 3NF.

Revision Questions

1. What is normalization
2. Explain the pros and cons of normalization
3. Explain First, Second and Third normal forms

Notes:

LESSON 12
NORMALIZATION - PART II

Hi! We are going to continue with Normalization. I hope the basics of normalization are clear to you.

Boyce-Codd Normal Form (BCNF)

The relation student(sno, sname, cno, cname) has all attributes participating in candidate keys since all the attributes are assumed to be unique. We therefore had the following candidate keys:

(sno, cno)

(sno, cname)

(sname, cno)

(sname, cname)

Since the relation has no non-key attributes, the relation is in 2NF and also in 3NF, in spite of the relation suffering the problems that we discussed at the beginning of this chapter.

The difficulty in this relation is being caused by dependence within the candidate keys. The second and third normal forms assume that all attributes not part of the candidate keys depend on the candidate keys but does not deal with dependencies within the keys. BCNF deals with such dependencies.

A relation R is said to be in BCNF if whenever $X \rightarrow A$ ***holds in R, and A is not in X, then X is a candidate key for R.***

It should be noted that most relations that are in 3NF are also in BCNF. Infrequently, a 3NF relation is not in BCNF and this happens only if

a. The candidate keys in the relation are composite keys (that is, they are not single attributes),

b. There is more than one candidate key in the relation, and

c. The keys are not disjoint, that is, some attributes in the keys are common.

The BCNF differs from the 3NF only when there are more than one candidate keys and the keys are composite and overlapping. Consider for example, the relationship

enrol (sno, sname, cno, cname, date-enrolled)

Let us assume that the relation has the following candidate keys:

(sno, cno)
(sno, cname)
(sname, cno)
(sname, cname)

(we have assumed sname and cname are unique identifiers). The relation is in 3NF but not in BCNF because there are dependencies

$sno \rightarrow sname$

$cno \rightarrow cname$

where attributes that are part of a candidate key are dependent on part of another candidate key. Such dependencies indicate that although the relation is about some entity or association that is identified by the candidate keys e.g. *(sno, cno)*, there are attributes that are not about the whole thing that the keys identify. For example, the above relation is about an association (enrolment) between students and subjects and therefore the relation needs to include only one identifier to identify students and one identifier to identify subjects. Providing two identifiers about students *(sno, sname)* and two keys about subjects *(cno, cname)* means that some information about students and subjects that is not needed is being provided. This provision of information will result in repetition of information and the anomalies that we discussed at the beginning of this chapter. If we wish to include further information about students and courses in the database, it should not be done by putting the information in the present relation but by creating new relations that represent information about entities student and subject.

These difficulties may be overcome by decomposing the above relation in the following three relations:

(sno, sname)
(cno, cname)
(sno, cno, date-of-enrolment)

We now have a relation that only has information about students, another only about subjects and the third only about enrolments. All the anomalies and repetition of information have been removed.

So, a relation is said to be in the BCNF if and only if it is in the 3NF and every non-trivial, left-irreducible functional dependency has a candidate key as its determinant. In more informal terms, a relation is in BCNF if it is in 3NF and the only determinants are the candidate keys.

Desirable Properties of Decompositions

So far our approach has consisted of looking at individual relations and checking if they belong to 2NF, 3NF or BCNF. If a relation was not in the normal form that was being checked for and we wished the relation to be normalized to that normal form so that some of the anomalies can be eliminated, it was necessary to decompose the relation in two or more relations. The process of decomposition of a relation *R* into a set of relations $R_1, R_2, \ldots R_n$ was based on identifying different components and using that as a basis of decomposition. The decomposed relations $R_1, R_2, \ldots R_n$ are projections of *R* and are of course not disjoint otherwise the glue holding the information together would be lost. Decomposing relations in this way based on a recognize and split method is not a particularly sound approach since we do not even have a basis to determine that the original relation can be constructed if necessary from the decomposed relations. We now discuss

desirable properties of good decomposition and identify difficulties that may arise if the decomposition is done without adequate care. The next section will discuss how such decomposition may be derived given the FDs.

Desirable properties of decomposition are:

1. Attribute preservation
2. Lossless-join decomposition
3. Dependency preservation
4. Lack of redundancy

We discuss these properties in detail.

Attribute Preservation

This is a simple and an obvious requirement that involves preserving all the attributes that were there in the relation that is being decomposed.

Lossless-Join Decomposition

In these notes so far we have normalized a number of relations by decomposing them. We decomposed a relation intuitively. We need a better basis for deciding decompositions since intuition may not always be correct. We illustrate how a careless decomposition may lead to problems including loss of information.

Consider the following relation

enrol (sno, cno, date-enrolled, room-No., instructor)

Suppose we decompose the above relation into two relations *enrol1* and *enrol2* as follows

enrol1 (sno, cno, date-enrolled)
enrol2 (date-enrolled, room-No., instructor)

There are problems with this decomposition but we wish to focus on one aspect at the moment. Let an instance of the relation enrol be

sno	cno	date-enrolled	room-No.	instructor
830057	CP302	1FEB1984	MP006	Gupta
830057	CP303	1FEB1984	MP006	Jones
820159	CP302	10JAN1984	MP006	Gupta
825678	CP304	1FEB1984	CE122	Wilson
826789	CP305	15JAN1984	EA123	Smith

and let the decomposed relations enrol1 and enrol2 be:

sno	cno	date-enrolled
830057	CP302	1FEB1984
830057	CP303	1FEB1984
820159	CP302	10JAN1984
825678	CP304	1FEB1984
826789	CP305	15JAN1984

date-enrolled	room-No.	instructor
1FEB1984	MP006	Gupta
1FEB1984	MP006	Jones
10JAN1984	MP006	Gupta
1FEB1984	CE122	Wilson
15JAN1984	EA123	Smith

All the information that was in the relation enrol appears to be still available in *enrol1* and *enrol2* but this is not so. Suppose, we wanted to retrieve the student numbers of all students taking a course from Wilson, we would need to join enrol1 and enrol2. The join would have 11 tuples as follows:

sno	cno	date-enrolled	room-No.	instructor
830057	CP302	1FEB1984	MP006	Gupta
830057	CP302	1FEB1984	MP006	Jones
830057	CP303	1FEB1984	MP006	Gupta
830057	CP303	1FEB1984	MP006	Jones
830057	CP302	1FEB1984	CE122	Wilson
830057	CP303	1FEB1984	CE122	Wilson

(add further tuples ...)

The join contains a number of spurious tuples that were not in the original relation Enrol. Because of these additional tuples, we have lost the information about which students take courses from WILSON. (Yes, we have more tuples but less information because we are unable to say with certainty who is taking courses from WILSON). Such decompositions are called lossy decompositions. A nonloss or lossless decomposition is that which guarantees that the join will result in exactly the same relation as was decomposed. One might think that there might be other ways of recovering the original relation from the decomposed relations but, sadly, no other operators can recover the original relation if the join does not (why?).

We need to analyze why some decompositions are lossy. The common attribute in above decompositions was Date-enrolled. The common attribute is the glue that gives us the ability to find the relationships between different relations by joining the relations together. If the common attribute is not unique, the relationship information is not preserved. If each tuple had a unique value of Date-enrolled, the problem of losing information would not have existed. The problem arises because several enrolments may take place on the same date.

A decomposition of a relation R into relations $R_1, R_2, \ldots R_n$ is called a lossless-join decomposition (with respect to FDs F) if the relation R is always the natural join of the relations $R_1, R_2, \ldots R_n$. It should be noted that natural join is the only way to recover the relation from the decomposed relations. There is no other set of operators that can recover the relation if the join cannot. Furthermore, it should be noted when the decomposed relations $R_1, R_2, \ldots R_n$ are obtained by projecting on the relation R, for example R_1 by projection $\pi_1(R)$, the relation R_1 may not always be precisely equal to the projection since the relation R_1 might have additional tuples called the dangling tuples.

It is not difficult to test whether a given decomposition is lossless-join given a set of functional dependencies *F*. We consider the simple case of a relation *R* being decomposed into R_1 and R_2. If the decomposition is lossless-join, then one of the following two conditions must hold

$R_1 \cap R_2 \rightarrow R_1 - R_2$

$R_1 \cap R_2 \rightarrow R_2 - R_1$

That is, the common attributes in R_1 and R_2 must include a candidate key of either R_1 or R_2. How do you know, you have a loss-less join decomposition?

Dependency Preservation

It is clear that decomposition must be lossless so that we do not lose any information from the relation that is decomposed. Dependency preservation is another important requirement since a dependency is a constraint on the database and if $X \rightarrow Y$ holds than we know that the two (sets) attributes are closely related and it would be useful if both attributes appeared in the same relation so that the dependency can be checked easily.

Let us consider a relation *R(A, B, C, D)* that has the dependencies *F* that include the following:

$A \rightarrow B$

$A \rightarrow C$

etc

If we decompose the above relation into *R1(A, B)* and *R2(B, C, D)* the dependency $A \rightarrow C$ cannot be checked (or preserved) by looking at only one relation. It is desirable that decompositions be such that each dependency in F may be checked by looking at only one relation and that no joins need be computed for checking dependencies. In some cases, it may not be possible to preserve each and every dependency in F but as long as the dependencies that are preserved are equivalent to F, it should be sufficient.

Let F be the dependencies on a relation R which is decomposed in relations $R_1, R_2, \ldots R_n$.

We can partition the dependencies given by *F* such that $F_1, F_2, \ldots F_n$. F_n are dependencies that only involve attributes from relations $R_1, R_2, \ldots R_n$ respectively. If the union of dependencies F_i imply all the dependencies in F, then we say that the decomposition has preserved dependencies, otherwise not.

If the decomposition does not preserve the dependencies *F*, then the decomposed relations may contain relations that do not satisfy *F* or the updates to the decomposed relations may require a join to check that the constraints implied by the dependencies still hold.

(Need an example) (Need to discuss testing for dependency preservation with an example... Ullman page 400)

Consider the following relation

sub(sno, instructor, office)

We may wish to decompose the above relation to remove the transitive dependency of office on sno. A possible decomposition is

S1(sno, instructor)
S2(sno, office)

The relations are now in 3NF but the dependency $instructor \rightarrow office$ cannot be verified by looking at one relation; a join of *S1* and *S2* is needed. In the above decomposition, it is quite possible to have more than one office number for one instructor although the functional dependency $instructor \rightarrow office$ does not allow it.

Lack of Redundancy

We have discussed the problems of repetition of information in a database. Such repetition should be avoided as much as possible.

Lossless-join, dependency preservation and lack of redundancy not always possible with BCNF. Lossless-join, dependency preservation and lack of redundancy is always possible with 3NF.

Deriving BCNF

Should we also include deriving 3NF?

Given a set of dependencies F, we may decompose a given relation into a set of relations that are in BCNF using the following algorithm. So far we have considered the "recognize and split" method of normalization. We now discuss Bernstein's algorithm. The algorithm consists of

1. Find out the facts about the real world.
2. Reduce the list of functional relationships.
3. Find the keys.
4. Combine related facts.

Once we have obtained relations by using the above approach we need to check that they are indeed in BCNF. If there is any relation *R* that has a dependency $A \rightarrow B$ and A is not a key, the relation violates the conditions of BCNF and may be decomposed in *AB* and *R - A*. The relation *AB* is now in BCNF and we can now check if *R - A* is also in BCNF. If not, we can apply the above procedure again until all the relations are in fact in BCNF.

- Data Base Management Systems by Alexis Leon, Mathews Leon
- http://databases.about.com/library

Review Questions

1. Explain BCNF
2. Explain attribute preservation and dependence preservation
3. Explain Loss-less join decomposition

LESSON 13
NORMALIZATION - PART III

Hi! Now we are going to discuss the reasons for the higher normal forms like fourth and fifth normal form.

Multivalued Dependencies

Recall that when we discussed database modelling using the E-R Modelling technique, we noted difficulties that can arise when an entity has multivalue attributes. It was because in the relational model, if all of the information about such entity is to be represented in one relation, it will be necessary to repeat all the information other than the multivalue attribute value to represent all the information that we wish to represent. This results in many tuples about the same instance of the entity in the relation and the relation having a composite key (the entity id and the mutlivalued attribute). Of course the other option suggested was to represent this multivalue information in a separate relation.

The situation of course becomes much worse if an entity has more than one multivalued attributes and these values are represented in one relation by a number of tuples for each entity instance such that every value of one the multivalued attributes appears with every value of the second multivalued attribute to maintain consistency. The multivalued dependency relates to this problem when more than one multivalued attributes exist. Consider the following relation that represents an entity employee that has one mutlivalued attribute *proj*:

emp (e#, dept, salary, proj)

We have so far considered normalization based on functional dependencies; dependencies that apply only to single-valued facts. For example, $e\# \rightarrow dept$ implies only one *dept* value for each value of *e#*. Not all information in a database is single-valued, for example, *proj* in an employee relation may be the list of all projects that the employee is currently working on. Although *e#* determines the list of all projects that an employee is working on, $e\# \rightarrow proj$ is not a functional dependency..

So far we have dealt with multivalued facts about an entity by having a separate relation for that multivalue attribute and then inserting a tuple for each value of that fact. This resulted in composite keys since the multivalued fact must form part of the key. In none of our examples so far have we dealt with an entity having more than one multivalued attribute in one relation. We do so now.

The fourth and fifth normal forms deal with multivalued dependencies. Before discussing the 4NF and 5NF we discuss the following example to illustrate the concept of multivalued dependency.

programmer (emp_name, qualifications, languages)

The above relation includes two multivalued attributes of entity *programmer*; *qualifications* and *languages*. There are no functional dependencies.

The attributes qualifications and languages are assumed independent of each other. If we were to consider *qualifications* and *languages* separate entities, we would have two relationships (one between *employees* and *qualifications* and the other between *employees* and programming *languages*). Both the above relationships are many-to-many i.e. one programmer could have several qualifications and may know several programming languages. Also one qualification may be obtained by several programmers and one programming language may be known to many programmers.

The above relation is therefore in 3NF (even in BCNF) but it still has some disadvantages. Suppose a programmer has several qualifications (B.Sc, Dip. Comp. Sc, etc) and is proficient in several programming languages; how should this information be represented? There are several possibilities.

emp name	qualifications	languages
SMITH	B.Sc	FORTRAN
SMITH	B.Sc	COBOL
SMITH	B.Sc	PASCAL
SMITH	Dip.CS	FORTRAN
SMITH	Dip.CS	COBOL
SMITH	Dip.CS	PASCAL

emp name	qualifications	language
SMITH	B.Sc	NULL
SMITH	Dip.CS	NULL
SMITH	NULL	FORTRAN
SMITH	NULL	COBOL
SMITH	NULL	PASCAL

emp name	qualifications	language
SMITH	B.Sc	FORTRAN
SMITH	Dip.CS	COBOL
SMITH	NULL	PASCAL

Other variations are possible (we remind the reader that there is no relationship between qualifications and programming languages). All these variations have some disadvantages. If the information is repeated we face the same problems of repeated information and anomalies as we did when second or third normal form conditions are violated. If there is no repetition, there are still some difficulties with search, insertions and deletions. For example, the role of NULL values in the above relations is confusing. Also the candidate key in the above relations is (emp name, qualifications, language) and existential integrity requires that no NULLs be specified. These problems may be overcome by decomposing a relation like the one above as follows:

emp name	qualifications
SMITH	B.Sc
SMITH	Dip.CS

emp name	languages
SMITH	FORTRAN
SMITH	COBOL
SMITH	PASCAL

The basis of the above decomposition is the concept of multivalued dependency (MVD). Functional dependency $A \rightarrow B$ relates one value of *A* to one value of *B* while multivalued dependency $A \twoheadrightarrow B$ defines a relationship in which a set of values of attribute *B* are determined by a single value of *A*.

The concept of multivalued dependencies was developed to provide a basis for decomposition of relations like the one above. Therefore if a relation like *enrolment(sno, subject#)* has a relationship between *sno* and *subject#* in which *sno* uniquely determines the values of *subject#*, the dependence of *subject#* on *sno* is called a *trivial* MVD since the relation enrolment cannot be decomposed any further. More formally, a MVD $X \twoheadrightarrow Y$ is called trivial MVD if either *Y* is a subset of *X* or *X* and *Y* together form the relation *R*. The MVD is trivial since it results in no constraints being placed on the relation. Therefore a relation having non-trivial MVDs must have at least three attributes; two of them multivalued. Non-trivial MVDs result in the relation having some constraints on it since all possible combinations of the multivalue attributes are then required to be in the relation.

Let us now define the concept of multivalued dependency. *The multivalued dependency* $X \twoheadrightarrow Y$ *is said to hold for a relation R(X, Y, Z) if for a given set of value (set of values if X is more than one attribute) for attributes X, there is a set of (zero or more) associated values for the set of attributes Y and the Y values depend only on X values and have no dependence on the set of attributes Z.*

In the example above, if there was some dependence between the attributes *qualifications* and *language*, for example perhaps, the language was related to the qualifications (perhaps the qualification was a training certificate in a particular language), then the relation would not have MVD and could not be decomposed into two relations as abve. In the above situation whenever $X \twoheadrightarrow Y$ holds, so does $X \twoheadrightarrow Z$ since the role of the attributes *Y* and *Z* is symmetrical.

Consider two different situations.

a. *Z* is a single valued attribute. In this situation, we deal with *R(X, Y, Z)* as before by entering several tuples about each entity.

b. *Z* is multivalued.

Now, more formally, $X \twoheadrightarrow Y$ is said to hold for *R(X, Y, Z)* if *t1* and *t2* are two tuples in *R* that have the same values for attributes *X* and therefore with *t1[x] = t2[x]* then *R* also contains tuples *t3* and *t4* (not necessarily distinct) such that

t1[x] = t2[x] = t3[x] = t4[x]
t3[Y] = t1[Y] and *t3[Z] = t2[Z]*
t4[Y] = t2[Y] and *t4[Z] = t1[Z]*

In other words if *t1* and *t2* are given by

t1 = [X, Y1, Z1], and
t2 = [X, Y2, Z2]

then there must be tuples *t3* and *t4* such that

t3 = [X, Y1, Z2], and
t4 = [X, Y2, Z1]

We are therefore insisting that every value of *Y* appears with every value of *Z* to keep the relation instances consistent. In other words, the above conditions insist that *Y* and *Z* are determined by *X* alone and there is no relationship between *Y* and *Z* since *Y* and *Z* appear in every possible pair and hence these pairings present no information and are of no significance. Only if some of these pairings were not present, there would be some significance in the pairings.

Give example (instructor, quals, subjects) - - explain if subject was single valued; otherwise all combinations must occur. Discuss duplication of info in that case.

(Note: If *Z* is single-valued and functionally dependent on *X* then *Z1 = Z2*. If *Z* is multivalue dependent on *X* then *Z1 <> Z2*).

The theory of multivalued dependencies in very similar to that for functional dependencies. Given *D* a set of MVDs, we may find D^+, the closure of *D* using a set of axioms. We do not discuss the axioms here. (Interested reader is referred to page 203 Korth & Silberschatz or Ullman).

Multivalued Normalization - Fourth Normal Form

Def: A table is in 4NF if it is in BCNF and if it has no multivalued dependencies.

We have considered an example of Programmer(Emp name, qualification, languages) and discussed the problems that may arise if the relation is not normalised further. We also saw how the relation could be decomposed into P1(Emp name, qualifications) and P2(Emp name, languages) to overcome these problems. The decomposed relations are in fourth normal form (4NF) which we shall now define.

We are now ready to define 4NF. A relation R is in 4NF if, whenever a multivalued dependency $X \rightarrow Y$ holds then either

a. the dependency is trivial, or

b. X is a candidate key for R.

As noted earlier, the dependency $X \twoheadrightarrow \phi$? or $X \twoheadrightarrow Y$ in a relation R(X, Y) is trivial since they must hold for all *R(X, Y)*. Similarly $(X, Y) \rightarrow Z$ must hold for all relations *R(X, Y, Z)* with only three attributes.

In fourth normal form, we have a relation that has information about only one entity. If a relation has more than one multivalue attribute, we should decompose it to remove difficulties with multivalued facts.

Intuitively R is in 4NF if all dependencies are a result of keys. When multivalued dependencies exist, a relation should not contain two or more independent multivalued attributes. The decomposition of a relation to achieve 4NF would normally result in not only reduction of redundancies but also avoidance of anomalies.

Fifth Normal Form

Def: A table is in 5NF, also called "Projection-Join Normal Form" (PJNF), if it is in 4NF and if every join dependency in the table is a consequence of the candidate keys of the table.

The normal forms discussed so far required that the given relation R if not in the given normal form be decomposed in two relations to meet the requirements of the normal form. In some rare cases, a relation can have problems like redundant information and update anomalies because of it but cannot be decomposed in two relations to remove the problems. In such cases it may be possible to decompose the relation in three or more relations using the 5NF.

The fifth normal form deals with join-dependencies which is a generalisation of the MVD. The aim of fifth normal form is to have relations that cannot be decomposed further. A relation in 5NF cannot be constructed from several smaller relations.

A relation *R* satisfies join dependency ($R_1, R_2, \ldots R_n$) if and only if R is equal to the join of $R_1, R_2, \ldots R_n$ where R_i are subsets of the set of attributes of *R*.

A relation *R* is in 5NF (or project-join normal form, PJNF) if for all join dependencies at least one of the following holds.

a. ($R_1, R_2, \ldots R_n$) is a trivial join-dependency (that is, one of R_i is *R*)

b. Every R_i is a candidate key for *R*.

An example of 5NF can be provided by the example below that deals with departments, subjects and students.

dept	subject	student
Comp. Sc.	CP1000	John Smith
Mathematics	MA1000	John Smith
Comp. Sc.	CP2000	Arun Kumar
Comp. Sc.	CP3000	Reena Rani
Physics	PH1000	Raymond Chew
Chemistry	CH2000	Albert Garcia

The above relation says that Comp. Sc. offers subjects CP1000, CP2000 and CP3000 which are taken by a variety of students. No student takes all the subjects and no subject has all students enrolled in it and therefore all three fields are needed to represent the information.

The above relation does not show MVDs since the attributes subject and student are not independent; they are related to each other and the pairings have significant information in them. The relation can therefore not be decomposed in two relations

(dept, subject), and
(dept, student)

without loosing some important information. The relation can however be decomposed in the following three relations

(dept, subject), and
(dept, student)
(subject, student)

and now it can be shown that this decomposition is lossless.

Domain-Key Normal Form (DKNF)

Def: A table is in DKNF if every constraint on the table is a logical consequence of the definition of keys and domains.

Review Questions

1. What do you mean by Normalization?
2. Explain BCNF?
3. Explain 4NF and 5NF?
4. Explain Domain Key Normal Form?

References

1. Aho, A. V. and C. Beeri and J. D. Ullman, "The Theory of Joins in Relational Databases", ACM-TODS, Vol 4, No 3, Sept 1979, pp. 297-314.
2. Fagin, R. (1981), "A Normal Form for Relational Databases that is Based on Domains and Keys", ACM-TODS, Vol 6, No 3, Sept 1981, pp 387-415.
3. Beeri, C, and P. A. Bernstein (1979), "Computational Problems Related to the Design of Normal Form Relational Schemas", ACM-TODS, Vol 4, No 1, Sept 1979, pp 30-59.
4. Kent, W. (1983), "A Simple Guide to Five Normal Forms in Relational Database Theory", Comm ACM, Vol 26, No 2, Feb 1983, pp. 120-125.
5. Bernstein, P. A. (1976), "Synthesizing Third Normal Form Relations from Functional Dependencies", ACM-TODS, Vol. 1, No. 4, Oct. 76, pp. 277-298.

Notes:

LESSON 14
NORMALIZATION (A DIFFERENT APPROACH)

Hi! I hope by now normalization is clear to you. We are going to learn and understand with practical examples.

Normalization is the formalization of the design process of making a database compliant with the concept of a **Normal Form**. It addresses various ways in which we may look for repeating data values in a table. There are several levels of the Normal Form, and each level requires that the previous level be satisfied. I have used the wording (indicated in italicized text) for each normalization rule from the *Handbook of Relational Database Design* by Candace C. Fleming and Barbara von Halle.[4]

The normalization process is based on collecting an exhaustive list of all data items to be maintained in the database and starting the design with a few "superset" tables. Theoretically, it may be possible, although not very practical, to start by placing all the attributes in a single table. For best results, start with a reasonable breakdown.

First Normal Form

Reduce entities to first normal form (1NF) by removing repeating or multivalued attributes to another, child entity.

Basically, make sure that the data is represented as a (proper) table. While key to the relational principles, this is somewhat a motherhood statement. However, there are six properties of a relational table (the formal name for "table" is "relation"):

Property 1: Entries in columns are single-valued.

Property 2: Entries in columns are of the same kind.

Property 3: Each row is unique.

Property 4: Sequence of columns is insignificant.

Property 5: Sequence of rows is insignificant.

Property 6: Each column has a unique name.

The most common sins against the first normal form (1NF) are the lack of a Primary Key and the use of "repeating columns." This is where multiple values of the same type are stored in multiple columns. Take, for example, a database used by a company's order system. If the order items were implemented as multiple columns in the Orders table, the database would not be 1NF:

OrderNo	**Line1Item**	**Line1Qty**	**Line1Price**	**Line2Item**	**Line2Qty**	**Line2Price**
245	**PN768**	**1**	**$35**	**PN656**	**3**	**$15**

To make this first normal form, we would have to create a child entity of Orders (Order Items) where we would store the information about the line items on the order. Each order could then have multiple Order Items *related* to it.

OrderNo
245

OrderNo	**Item**	**Qty**	**Price**
245	**PN768**	**1**	**$35**
245	**PN656**	**3**	**$15**

Second Normal Form *Reduce first normal form entities to second normal form (2NF) by removing attributes that are not dependent on the whole primary key.*

The purpose here is to make sure that each column is defined in the correct table. Using the more formal names may make this a little clearer. Make sure each attribute is kept with the entity that it describes.

Consider the Order Items table that we established above. If we place Customer reference in the Order Items table (Order Number, Line Item Number, Item, Qty, Price, Customer) and assume that we use Order Number and Line Item Number as the Primary Key, it quickly becomes obvious that the Customer reference becomes repeated in the table because it is only dependent on a portion of the Primary Key - namely the Order Number. Therefore, it is defined as an attribute of the wrong entity. In such an obvious case, it should be immediately clear that the Customer reference should be in the Orders table, not the Order Items table.

OrderNo	ItemNo	**Customer**	**Item**	**Qty**	**Price**
245	1	**SteelCo**	**PN768**	**1**	**$35**
245	2	**SteelCo**	**PN656**	**3**	**$15**
246	1	**Acme Corp**	**PN371**	**1**	**$2.99**
246	2	**Acme Corp**	**PN015**	**7**	**$5**

We get:

OrderNo	Customer
245	**SteelCo**
246	**Acme Corp**

OrderNo	ItemNo	Item	Qty	Price
245	1	**PN768**	**1**	**$35**
245	2	**PN656**	**3**	**$15**
246	1	**PN371**	**1**	**$2.99**
246	2	**PN015**	**7**	**$5**

Third Normal Form

Reduce second normal form entities to third normal form (3NF) by removing attributes that depend on other, nonkey attributes (other than alternative keys).

This basically means that we shouldn't store any data that can either be derived from other columns or belong in another table. Again, as an example of derived data, if our Order Items table includes both Unit Price, Quantity, and Extended Price, the table would not be 3NF. So we would remove the Extended Price (= Qty * Unit Price), unless, of course, the value saved is a manually modified (rebate) price, but the Unit Price reflects the quoted list price for the items at the time of order.

Also, when we established that the Customer reference did not belong in the Order Items table, we said to move it to the Orders table. Now if we included customer information, such as company name, address, etc., in the Orders table, we would see that this information is dependent not so much on the Order per se, but on the Customer reference, which is a nonkey (not Primary Key) column in the Orders table. Therefore, we need to create another table (Customers) to hold information about the customer. Each Customer could then have multiple Orders related to it.

OrderNo	Customer	Address	City
245	**SteelCo**	**Works Blvd**	**Vinings**
246	**Acme Corp**	**North Drive**	**South Bend**
247	**SteelCo**	**Works Blvd**	**Vinings**

OrderNo	Customer
245	**SteelCo**
246	**Acme Corp**
247	**SteelCo**

Customer	Address	City
SteelCo	**Works Blvd**	**Vinings**
Acme Corp	**North Drive**	**South Bend**

Why Stop Here?

Many database designers stop at 3NF, and those first three levels of normalization do provide the most bang for the buck. Indeed, these were the original normal forms described in E. F. Codd's first papers. However, there are currently four additional levels of normalization, so read on. Be aware of what you don't do, even if you stop with 3NF. In some cases, you may even need to de-normalize some for performance reasons.

Boyce/Codd Normal Form

Reduce third normal form entities to Boyce/Codd normal form (BCNF) by ensuring that they are in third normal form for any feasible choice of candidate key as primary key.

In short, Boyce/Codd normal form (BCNF) addresses dependencies between columns that are part of a Candidate Key.

Some of the normalizations performed above may depend on our choice of the Primary Key. BCNF addresses those cases where applying the normalization rules to a Candidate Key other than the one chosen as the Primary Key would give a different result. In actuality, if we substitute any Candidate Key for Primary Key in 2NF and 3NF, 3NF would be equivalent with BCNF.

In a way, the BCNF is only necessary because the formal definitions center around the Primary Key rather than an entity item abstraction. If we define an entity item as an object or information instance that correlates to a row, and consider the normalization rules to refer to entity items, this normal form would not be required.

In our example for 2NF above, we assumed that we used a composite Primary Key consisting of Order Number and Line Item Number, and we showed that the customer reference was only dependent on a portion of the Primary Key - the Order Number. If we had assigned a unique identifier to every Order Item independent of the Order Number, and used that as a single column Primary Key, the normalization rule itself would not have made it clear that it was necessary to move the Customer reference.

There are some less obvious situations for this normalization rule where a set of data actually contains more than one relation, which the following example should illustrate.

Consider a scenario of a large development organization, where the projects are organized in project groups, each with a team leader acting as a liaison between the overall project and a group of developers in a matrix organization. Assume we have the following situation:

- Each Project can have many Developers.
- Each Developer can have many Projects.
- For a given Project, each Developer only works for one Lead Developer.
- Each Lead Developer only works on one Project.
- A given Project can have many Lead Developers.

In this case, we could theoretically design a table in two different ways:

ProjectNo	Developer	**Lead Developer**
20020123	John Doe	**Elmer Fudd**
20020123	Jane Doe	**Sylvester**
20020123	Jimbo	**Elmer Fudd**
20020124	John Doe	**Ms. Depesto**

Case 1: Project Number and Developer as a Candidate Key can be used to determine the Lead Developer. In this case, the Lead Developer depends on both attributes of the key, and the table is 3NF if we consider that our Primary Key.

Lead Developer	**Developer**	**ProjectNo**
Elmer Fudd	**John Doe**	**20020123**
Sylvester	**Jane Doe**	**20020123**
Elmer Fudd	**Jimbo**	**20020123**
Ms. Depesto	**John Doe**	**20020124**

Case 2: Lead Developer and Developer is another Candidate Key, but in this case, the Project Number is determined by the Lead Developer alone. Thus it would not be 3NF if we consider that our Primary Key.

In reality, these three data items contain more than one relation (Project - Lead Developer and Lead Developer - Developer). To normalize to BCNF, we would remove the second relation and represent it in a second table. (This also illustrates why a *table* is formally named a *relation*.)

ProjectNo	**Lead Developer**
20020123	**Elmer Fudd**
20020123	**Sylvester**
20020123	**Elmer Fudd**
20020124	**Ms. Depesto**

Lead Developer	**Developer**
Elmer Fudd	**John Doe**
Elmer Fudd	**Jimbo**
Sylvester	**Jane Doe**
Ms. Depesto	**John Doe**

Fourth Normal Form

Reduce Boyce/Codd normal form entities to fourth normal form (4NF) by removing any independently multivalued components of the primary key to two new parent entities. Retain the original (now child) entity only if it contains other, nonkey attributes.

Where BCNF deals with dependents of dependents, 4NF deals with multiple, independent dependents of the Primary Key. This is a bit easier to illustrate.

Let us say we wanted to represent the following data: Manager, Manager Awards, and Direct Reports. Here, a Manager could have multiple Awards, as well as multiple Direct Reports. 4NF requires that these be split into two separate tables, one for Manager - Awards, and one for Manager - Direct Reports. We may need to maintain a Managers table for other Manager attributes.

This table:

Manager	**Awards**	**Direct Reports**
Scrooge McDuck	Stingy John	Donald Duck
Minnie Mouse	Mouse of the Month	Mickey Mouse
Minnie Mouse	Mouse of the Year	Pluto
Clara		Goofy

becomes two tables:

Manager Awards Table

Manager	Awards
Scrooge McDuck	Stingy John
Minnie Mouse	Mouse of the Month
Minnie Mouse	Mouse of the Year
Clara	

Direct Reports Table

Manager	Direct Reports
Scrooge McDuck	Donald Duck
Minnie Mouse	Mickey Mouse
Minnie Mouse	Pluto
Clara	Goofy

Fifth Normal Form

Reduce fourth normal form entities to fifth normal form (5NF) by removing pairwise cyclic dependencies (appearing within composite primary keys with three or more component attributes) to three or more parent entities.

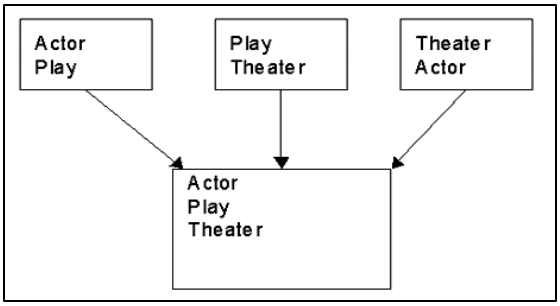
This addresses problems that arise from representing associations between multiple entities with interdependencies. Making

it 5NF consists of adding parent tables, one for each meaningful combination that has children in the original table.

A table with such information is 5NF if the information cannot be represented in multiple smaller entities alone.

An example of such a situation may be the representation of Actors, Plays, and Theaters. In order to know who plays what and where, we need the combination of these three attributes. However, they each relate to each other cyclically. So to resolve this, we would need to establish parent tables with Actor - Play, Play - Theater, and Theater - Actor. These would each contain a portion of the Primary Key in the Actor, Play, and Theater table.

Actor	Play	Theater
Billy Bob	Catcher in the Rye	West 42nd
Ann	Catcher in the Rye	West 42nd
John	Catch-22	Broadway
Lily	Hamlet	Broadway
Lisa	Cats	West 42nd
Andy	Cats	Darlington



Domain Key Normal Form

(Not defined in "Handbook of Relational Database Design."[5]).

The simplest description I have found is at Search Database.com at http://searchdatabase.techtarget.com/sDefinition/0,,sid13_gci212669,00.html:

"A key uniquely identifies each row in a table. A domain is the set of permissible values for an attribute. By enforcing key and domain restrictions, the database is assured of being freed from modification anomalies."

This appears to differ from the other normal forms in that it does not seek to introduce additional tables, but rather ensures that columns are restricted to valid values.

According to http://www.cba.nau.edu/morgan-j/class/subtop2_3/tsld023.htm, "...there is no known process for ensuring that tables are in Domain Key Normal Form."

Conclusion

While we may not always observe all the rules or normalize our databases to the fifth and domain key normal form, it is important to have a basic understanding of the theoretical principles of database design. It will help us not only design normalized databases, but to build more powerful and flexible applications. Also, it will help us ensure that our data remains usable. Now that we have laid the theoretical foundation and defined the formal database design methods for normalization, it may be time to take a break. I need one anyway. :->

When we resume with Part 2 of this article, I will show how we can design a fairly well normalized database using nothing but some common sense, a few simple rules, and a piece of string, so get ready! Part 2 of this article will address a different approach to designing the database, normalization through synthesis, and will describe the SQL language.

The relational model has three major aspects:

Structures	Structures are well-defined objects (such as tables, views, indexes, and so on) that store or access the data of a database. Structures and the data contained within them can be manipulated by operations.
Operations	Operations are clearly defined actions that allow users to manipulate the data and structures of a database. The operations on a database must adhere to a predefined set of integrity rules.
Integrity Rules	Integrity rules are the laws that govern which operations are allowed on the data and structures of a database. Integrity rules protect the data and the structures of a database.

Relational database management systems offer benefits such as:

- Independence of physical data storage and logical database structure
- Variable and easy access to all data
- Complete flexibility in database design
- Reduced data storage and redundancy

Review Questions

1. Explain first, second and third normal forms
2. Explain BCNF
3. Explain fourth and fifth normal forms
4. Define Domain key normal form

Sources

1. C. J. Date, "There's Only One Relational Model!"(see http://www.pgro.uk7.net/cjd6a.htm).
2. Dr. E. F. Codd's 12 rules for defining a fully relational database (see http://www.cis.ohio-state.edu/~sgomori/570/coddsrules.html).
3. C.J.Date *Handbook of Relational Database Design* by Candace C. Fleming and Barbara von Halle (Addison Wesley, 1989).5. *Ibid.*

References

1. Characteristics of a Relational Database by David R. Frick & Co., CPA.
2. Dr. Morgan at Northern Arizona University - College of Business Administration
3. SearchDatabase.com

LESSON 15
A COMMERCIAL QUERY LANGUAGE – SQL

Hi! In this lecture I would like to discuss with you the database language, t**he Structured Query Language.**

In this section we discuss a query language that is now being used in most commercial relational DBMS. Although a large number of query languages and their variants exist just as there are a large number of programming languages available, the most popular query language is SQL. SQL has now become a de facto standard for relational database query languages. We discuss SQL in some detail.

SQL is a non-procedural language that originated at IBM during the building of the now famous experimental relational DBMS called System R. Originally the user interface to System R was called SEQUEL which was later modified and its name changed to SEQUEL2. These languages have undergone significant changes over time and the current language has been named SQL (Structured Query Language) although it is still often pronounced as if it was spelled SEQUEL. Recently, the American Standards Association has adopted a standard definition of SQL (Date, 1987).

A user of a DBMS using SQL may use the query language interactively or through a host language like C or COBOL. We will discuss only the interactive use of SQL although the SQL standard only defines SQL use through a host language.

We will continue to use the database that we have been using to illustrate the various features of the query language SQL in this section. As a reminder, we again present the relational schemes of the three relations *student*, *subject* and *enrolment*.

students(student_id, student_name, address)
enrolment(student_id, subject_id)
subject(subject_id, subject_name, department)

SQL consists of facilities for data definition as well as for data manipulation. In addition, the language includes some data control features. The data definition facilities include commands for creating a new relation or a view, altering or expanding a relation (entering a new column) and creating an index on a relation, as well as commands for dropping a relation, dropping a view and dropping an index.

The data manipulation commands include commands for retrieving information from one or more relations and updating information in relations including inserting and deleting of tuples.

We first study the data manipulation features of the language.

Data Manipulation - Data Retrieval Features

The basic structure of the SQL data retrieval command is as follows

SELECT something_of_interest
FROM relation(s)
WHERE condition_holds

The SELECT clause specifies what attributes we wish to retrieve and is therefore equivalent to specifying a projection. (There can often be confusion between the SELECT clause in SQL and the relational operator called Selection. The selection operator selects a number of tuples from a relation while the SELECT clause in SQL is similar to the projection operator since it specifies the attributes that need to be retrieved). The FROM clause specifies the relations that are needed to answer the query. The WHERE clause specifies the condition to be used in selecting tuples and is therefore like the predicate in the selection operator. The WHERE clause is optional, if it is not specified the whole relation is selected. A number of other optional clauses may follow the WHERE clause. These may be used to group data and to specify group conditions and to order data if necessary. These will be discussed later.

The result of each query is a relation and may therefore be used like any other relation. There is however one major difference between a base relation and a relation that is retrieved from a query. The relation retrieved from a query may have duplicates while the tuples in a base relation are unique.

We present a number of examples to illustrate different forms of the SQL SELECT command.

Simple Queries

We will first discuss some simple queries that involve only one relation in the database. The simplest of such queries includes only a SELECT clause and a FROM clause.

Q1. Find the student id's and names of all students.

SELECT *student_num, student_name*
FROM *student*

When the WHERE clause is omitted, the SELECT attributes FROM relations construct is equivalent to specifying a projection on the relation(s) specified in the FROM clause. The above query therefore results in applying the projection operator to the relation *student* such that only attributes *student_num* and *student_name* are selected.

Q2. Find the names of subjects offered by the Department of Computer Science.

SELECT *subject_name*
FROM *subject*
WHERE *department* = 'Comp. Science'

The above query involves algebraic operators selection and projection and therefore the WHERE clause is required.

One way of looking at the processing of the above query is to think of a pointer going down the table *subject* and at each tuple asking the question posed in the WHERE clause. If the answer is yes, the tuple being pointed at is selected, otherwise rejected.

Q3. Find all students that are enrolled in something.

SELECT DISTINCT (*student_id*)
FROM *enrolment*

The above query is a projection of the relation *enrolment*. Only the *student_id* attribute has been selected and the duplicate values are to be removed.

Q4. Find student id's of all students enrolled in CP302.

SELECT *student_id*
FROM *enrolment*
WHERE *student_id* = 'CP302'

This query involves a selection as well as a projection.

Queries Involving More than One Relations

Q5. Find subject id's of all subjects being taken by Fred Smith.

SELECT *subject_id*
FROM *enrolment*
WHERE *student_id* IN
(SELECT *student_id*
FROM *student*
WHERE *student_name* = 'Fred Smith')

The above query must involve two relations since the subject information is in relation *subject* while the student names are in relation *student*. The above formulation of the query has a sub-query (or a *nested query*) associated with it. The sub-query feature is very useful and we will use this feature again shortly. Queries may be nested to any level.

It may be appropriate to briefly discuss how such a query is processed. The subquery is processed first resulting in a new relation (in the present case a single attribute relation). This resulting relation is then used in the WHERE clause of the outside query which becomes true if the *student_id* value is in the relation returned by the subquery. The IN part of the WHERE clause tests for set membership. The WHERE clause may also use construct NOT IN instead of IN whenever appropriate.

Another format for the IN predicate is illustrated by the following query.

Q6. List subject names offered by the Department of Mathematics or the Department of Computer Science.

SELECT *subject_name*
FROM *subject*
WHERE *department* IN {'Mathematics', 'Comp. Science'}

In the above query, subject name is retrieved when the department name attribute value is either Mathematics or Comp. Science. Any number of values may be included in the list that follows IN.

Constants like 'Mathematics' or 'Comp. Science' are often called a *literal tuple*. A literal tuple of course could have more than one value in it. For example it could be WHERE {CP302, Database, Comp. Science} IN *subject*.

Obviously the set of attributes in the list before IN must match the list of attributes in the list that follows IN for the set membership test carried out by IN to be legal (the list in the example above being a relation).

The above query is equivalent to the following

SELECT *subject_name*
FROM *subject*
WHERE *department* = 'Mathematics'
or *department* = 'Comp. Science'

SQL allows some queries to be formulated in a number of ways. We will see further instances where a query can be formulated in two or more different ways.

Q7. Find the subject names of all the subjects that Fred Smith is enrolled in.

SELECT *subject_name*
FROM *subject*
WHERE *subject_id* IN
(SELECT *subject_id*
FROM *enrolment*
WHERE *student_id* IN
(SELECT *student_id*
FROM *student*
WHERE *student_name* = 'Fred Smith'))

This query of course must involve all the three relations since the subject names are in relation *subject* and the student names are in *student* and students enrollments are available in *enrolment*. The above query results in the last subquery being processed first resulting in a relation with only one tuple that is the student number of Fred Smith. This result is then used to find all enrollments of Fred Smith. The enrollments are then used to find subject names.

The two queries Q5 and Q7 that involve subqueries may be reformulated by using the join operator.

SELECT *subject_name*
FROM *student, enrolment*
WHERE *enrolment.student_id = student.student_id*
AND *student_name* = 'Fred Smith'.

SELECT *subject_name*
FROM *enrolment, student, subject*
WHERE *student.student_id = enrolment.student_id*
AND *enrolment.subject_id = subject.subject_id*
AND *student.name* = 'Fred Smith'.

The above two query formulations imply the use of the join operator. When the FROM clause specifies more than one relation name, the query processor forms a cartesian product of those relations and applies the join predicate specified in the WHERE clause. In practice, the algorithm used is not that simple. Fast algorithms are used for computing the joins.

We should also note that in the queries above, we qualified some of the attributes by specifying the relation name with the attribute name (e.g. *enrolment.student_id*). This qualification is needed only when there is a chance of an ambiguity arising. In queries later, we will also need to give alias to some relation names when more than one instance of the same relation is being used in the query.

Q8. Display a sorted list of subject names that are offered by the department of Computer Science.

```
SELECT subject_name
FROM subject
WHERE department = 'Comp. Science'
ORDER BY subject_name
```

The list returned by the above query would include duplicate names if more than one subject had the same name (this is possible since the subjects are identified by their *subject_id* rather than their name). DISTINCT could be used to remove duplicates if required.

The ORDER BY clause as used above would return results in ascending order (that is the default). Descending order may be specified by using

ORDER BY *subject_name* DESC

The order ascending may be specified explicitly. When the ORDER BY clause is not present, the order of the result depends on the DBMS implementation.

The ORDER BY clause may be used to order the result by more than one attribute. For example, if the ORDER BY clause in a query was ORDER BY *department, subject_name*, then the result will be ordered by *department* and the tuples with the same *department* will be ordered by *subject_name*. The ordering may be ascending or descending and may be different for each of the fields on which the result is being sorted.

Q9. Find student id's of students enrolled in CP302 and CP304.

```
SELECT student_id
FROM enrolment
WHERE subject_id = 'CP302'
AND student_id IN
(SELECT student_id
FROM enrolment
WHERE subject_id = 'CP304')
```

The above query could also be formulated as an intersection of the two queries above. Therefore the clause "AND *student_id* IN" could be replaced by "INTERSECT".

Q10. Find student id's of students enrolled in CP302 or CP304 or both.

```
SELECT student_id
FROM enrolment
WHERE subject_id = 'CP302'
OR subject_id = 'CP304'
```

The above query may also be written using the UNION operator as follows.

```
SELECT student_id
FROM enrolment
WHERE subject_id = 'CP302'

UNION

SELECT student_id
FROM enrolment
WHERE subject_id = 'CP304'
```

It is of course also possible to replace the UNION operator by "OR *student_id* IN" thereby making the second query a sub-query of the first query. It should be noted that the above query would result in presenting duplicates for students that are doing both subjects.

Q11. Find student id's of students enrolled in CP302 but not in CP304.

```
SELECT student_id
FROM enrolment
WHERE subject_id = 'CP302'
AND student_id NOT IN
(SELECT student_id
FROM enrolment
WHERE subject_id = 'CP304')
```

This query also can be formulated in another way. Rather than using a sub-query, we may take the difference of the results of the two queries. SQL provides an operator "MINUS" to take the difference.

So far we have only used the equality comparison in the WHERE clause. Other relationships are available. These include **greater than (>), less than (<), less than equ**(greater than equal (≥) and not equal (<>).

The WHERE clause has several different forms. Some of the common forms are:

1. WHERE C1 AND C2
2. WHERE C1 OR C2
3. WHERE NOT C1 AND C2
4. WHERE A operator ANY
5. WHERE A operator ALL
6. WHERE A BETWEEN x AND y
7. WHERE A LIKE x
8. WHERE A IS NULL
9. WHERE EXISTS
10. WHERE NOT EXISTS

Some of these constructs are used in the next few queries. To illustrate these features we will extend the relation *enrolment* to include information about date of each enrolment and marks obtained by the student in the subject. In addition, we will extend the relation *student* to include the *student_id* of the student's tutor (each student is assigned to a tutor who acts as the student's mentor). Each tutor is also a student. Therefore relational schema for *enrolment* and *student* are now:

enrolment(student_id, subject_id, date, mark)
student(student_id, student_name, address, tutor_id)

Q12. Find student id's of those students who got a mark better than any of the marks obtained by student 881234 in Mathematics subjects.

```
SELECT student_id
FROM enrolment
WHERE student_id <> 881234
AND mark > ANY
(SELECT mark
FROM enrolment
WHERE student_id = 881234
```

```
AND subject_id IN
(SELECT subject_id
FROM subject
WHERE department = 'Mathematics'))
```

Note that if the outermost part of the query did not include the test for *student_id* not being 881234, than we will also retrieve the id 881234.

When ANY is used in the WHERE clause, the condition is true if the value in the outer loop satisfies the condition with at least one value in the set returned by the subquery. When ALL is used, the value in the outer query must satisfy the specified condition with all the tuples that are returned by the subquery.

Q13. Find subject id's of those subjects in which student 881234 is enrolled in which he got a mark better than his marks in all the subjects in Mathematics.

```
SELECT subject_id
FROM enrolment
WHERE student_id = 881234
AND mark > ALL
(SELECT mark
FROM enrolment
WHERE student_id = 881234
AND subject_id IN
(SELECT subject_id
FROM subject
WHERE department = 'Mathematics'))
```

Q14. Find those subjects in which John Smith got a better mark than all marks obtained by student Mark Jones.

```
SELECT subject_id
FROM enrolment, student
WHERE student.student_id = enrolment.student_id
AND student_name = 'John Smith'
AND mark > ALL
(SELECT mark
FROM enrolment
WHERE student_id IN
(SELECT student_id
FROM student
WHERE student_name = 'Mark Jones'))
```

This query uses a join in the outer query and then uses sub-queries. The two sub-queries may be replaced by one by using a join.

Q15. Find student id's of students who have failed CP302 but obtained more than 40%.

```
SELECT student_id
FROM enrolment
WHERE subject_id = 'CP302'
AND mark BETWEEN 41 AND 49
```

Note that we are looking for marks between 41 and 49 and not between 40 and 50. This is because (A BETWEEN x AND y) has been defined to mean $A \geq x$ and $A \leq y$.

Q16. Find the student id's of those students that have no mark for CP302

```
SELECT student_id
FROM enrolment
WHERE subject_id = 'CP302'
AND mark IS NULL
```

Note that mark IS NULL is very different than mark being zero. Mark will be NULL only if it has been defined to be so.

Q17. Find the names of all subjects whose subject id starts with CP.

```
SELECT subject_name
FROM subject
WHERE subject_id LIKE 'CP%'
```

The expression that follows LIKE must be a character string. The characters underscore (_) and percent (%) have special meaning. Underscore represents any single character while percent represents any sequence of n characters including a sequence of no characters. Escape characters may be used if the string itself contains the characters underscore or percent.

Q18. Find the names of students doing CP302.

```
SELECT student_name
FROM student
WHERE EXISTS
(SELECT *
FROM enrolment
WHERE student.student_id = enrolment.student_id
AND subject_id = 'CP302');
```

The WHERE EXISTS clause returns true if the subquery following the EXISTS returns a non-null relation. Similarly a WHERE NOT EXISTS clause returns true only if the subquery following the NOT EXISTS returns a null relation.

Again, the above query may be formulated in other ways. One of these formulations uses the join. Another uses a subquery. Using the join, we obtain the following:

```
SELECT student_name
FROM enrolment, student
WHERE student.student_id = enrolment.student_id
AND subject_id = 'CP302');
```

Q19. Find the student id's of students that have passed all the subjects that they were enrolled in.

```
SELECT student_id
FROM enrolment e1
WHERE NOT EXISTS
(SELECT *
FROM enrolment e2
WHERE e1.student_id = e2.student_id
AND mark < 50))
```

The formulation of the above query is somewhat complex. One way to understand the above query is to rephrase the query as "find student_id's of all students that have no subject in which they are enrolled but have not passed"!

Note that the condition in the WHERE clause will be true only if the sub-query returns a null result. The sub-query will return a null result only if the student has no enrolment in which his/her mark is less than 50.

We also wish to note that the above query formulation has a weakness that an alert reader would have already identified. The

above query would retrieve *student_id* of a student for each of the enrolments that the student has. Therefore if a student has five subjects and he has passed all five, his id will be retrieved five time in the result. A better formulation of the query would require the outer part of the subquery to SELECT *student_id* FROM *student* and then make appropriate changes to the sub-query.

Q20. Find the student names of all students that are enrolled in all the subjects offered by the Department of Computer Science.

```
SELECT student_name
FROM student
WHERE NOT EXISTS
(SELECT *
FROM subject
WHERE department = 'Comp. Science'
AND NOT EXISTS
(SELECT *
FROM enrolment
WHERE student.student_id = enrolment.student_id
AND subject.subject_id = enrolment.subject_id))
```

It is worthwhile to discuss how the above query is processed. The query consists of three components: the outermost part, the middle sub-query and the innermost (or the last) sub-query. Conceptually, the outermost part of the query starts by looking at each tuple in the relation *student* and for the tuple under consideration evaluates whether the NOT EXISTS clause is true. The NOT EXISTS clause will be true only if nothing is returned by the middle subquery. The middle subquery will return a null relation only if for each of the subject offered by the Department of Computer Science, the NOT EXISTS clause is false. Of course, the NOT EXISTS clause will be false only if the innermost subquery returns a non-null result. That is, there is no tuple in *enrolment* for an enrolment in the subject that the middle sub-query is considering for the student of interest.

We look at the query in another way because we suspect the above explanation may not be sufficient. SQL does not have a universal quantifier (*forall* or $\forall$) but does have an existential quantifier (*there exists* or $\exists$). As considered before, the universal quantifier may be expressed in terms of the existential quantifier since $\forall x(P(x))$ is equivalent to $Not(\exists x(NotP(x))$. The query formulation above implements a universal quantifier using the existential quantifier. Effectively, for each student, the query finds out if there exists a subject offered by the Department of Computer Science in which the student is not enrolled. If such a subject exists, that student is not selected for inclusion in the result.

Using Built-in Functions

SQL provides a number of built-in functions. These functions are also called aggregate functions.

AVG
COUNT
MIN
MAX
SUM

As the names imply, the AVG function is for computing the average, COUNT counts the occurrences of rows, MIN finds the smallest value, MAX finds the largest value while SUM computes the sum.

We consider some queries that use these functions.

Q21. Find the number of students enrolled in CP302.

```
SELECT COUNT(*)
FROM enrolment
WHERE subject_id = 'CP302'
```

The above query uses the function COUNT to count the number of tuples that satisfy the condition specified.

Q22. Find the average mark in CP302.

```
SELECT AVG(mark)
FROM enrolment
WHERE subject_id = 'CP302'
```

This query uses the function AVG to compute the average of the values of attribute *mark* for all tuples that satisfy the condition specified.

Further queries using the built-in functions are presented after we have considered a mechanism for grouping a number of tuples having the same value of one or more specified attribute(s).

Using the GROUP BY and HAVING Clauses

In many queries, one would like to consider groups of records that have some common characteristics (e.g. employees of the same company) and compare the groups in some way. The comparisons are usually based on using an aggregate function (e.g. max, min, avg). Such queries may be formulated by using GROUP BY and HAVING clauses.

Q23. Find the number of students enrolled in each of the subjects

```
SELECT subject_id, COUNT(*)
FROM enrolment
GROUP BY subject_id
```

The GROUP clause groups the tuples by the specified attribute (*subject_id*) and then counts the number in each group and displays it.

Q24. Find the average enrolment in the subjects in which students are enrolled.

```
SELECT AVG(COUNT(*))
FROM enrolment
GROUP BY subject_id
```

This query uses two built-in functions. The relation *enrolment* is divided into groups with same *subject_id* values, the number of tuples in each group is counted and the average of these numbers is taken.

Q25. Find the subject(s) with the highest average mark.

```
SELECT subject_id
FROM enrolment
GROUP BY subject_id
HAVING AVG( mark) =
(SELECT MAX(AVG( mark))
FROM enrolment
GROUP BY subject_id);
```

The above query uses the GROUP BY and HAVING clauses. The HAVING clause is used to apply conditions to groups similar to the way WHERE clause is used for tuples. We may look at the GROUP BY clause as dividing the given relation in several virtual relations and then using the HAVING clause selecting those virtual relations that satisfy the condition specified in the clause. The condition must, of course, use a built-in function because any condition on a group must include an aggregation. If a GROUP BY query does not use an aggregation then GROUP BY clause was not needed in the query.

Q26. Find the department with the second largest enrolment.

SELECT *department*
FROM *enrolment* e1, *subject* s1
WHERE e1. *subject_id* = s1. *subject_id*
GROUP BY *department*
HAVING COUNT(*) =
(SELECT MAX(COUNT(*))
FROM *enrolment* e2, *subject* s2
WHERE e2. *subject_id* = s2. *subject_id*
GROUP BY *department*
HAVING COUNT(*) <>
(SELECT MAX(COUNT(*))
FROM *enrolment* e3, *subject* s3
WHERE e3. *subject_id* = s3. *subject_id*
GROUP BY *department*));

Finding the second largest or second smallest is more complex since it requires that we find the largest or the smallest and then remove it from consideration so that the largest or smallest of the remaining may be found. That is what is done in the query above.

Q27. Find the subject with the most number of students.

SELECT *subject_id*
FROM *enrolment*
GROUP BY *subject_id*
HAVING COUNT(*) =
(SELECT MAX(COUNT(*))
FROM *enrolment*
GROUP BY *subject_id*);

Q28. Find the name of the tutor of John Smith

SELECT *s2.student_name*
FROM *student s1, student s2*
WHERE *s1.tutor_id = s2.student_id*
AND *s1.student_name* = 'John Smith'

In this query we have joined the relation *student* to itself such that the join attributes are *tutor_id* from the first relation and *student_id* from the second relation. We now retrieve the *student_name* of the tutor.

Update Commands

The update commands include commands for updating as well as for inserting and deleting tuples. We now illustrate the UPDATE, INSERT and DELETE commands.

Q29. Delete course CP302 from the relation *subject*.

DELETE *subject*
WHERE *subject_id* = 'CP302'

The format of the DELETE command is very similar to that of the SELECT command. DELETE may therefore be used to either delete one tuple or a group of tuples that satisfy a given condition.

Q30. Delete all enrolments of John Smith

DELETE *enrolment*
WHERE *student_id* =
(SELECT *student_id*
FROM *student*
WHERE *student_name* = 'John Smith')

The above deletes those tuples from *enrolment* whose *student_id* is that of John Smith.

Q31. Increase CP302 marks by 5%.

UPDATE *enrolment*
SET *marks = marks* * 1.05 WHERE *subject_id* = 'CP302'

Again, UPDATE may also be used to either update one tuple or to update a group of tuples that satisfy a given condition. Note that the above query may lead to some marks becoming above 100. If the maximum mark is assumed to be 100, one may wish to update as follows

UPDATE *enrolment*
SET *marks* = 100
WHERE *subject_id* = 'CP302'
AND *marks* > 95;

UPDATE *enrolment*
SET *marks = marks* * 1.05 WHERE *subject_id* = 'CP302'
AND *marks* $\leq$ 95;

Q32. Insert a new student John Smith with student number 99 and subject CP302.

INSERT INTO *student(student_id, student_name, address, tutor)*:
<99, 'John Smith', NULL, NULL>

INSERT INTO *subject(subject_id, subject_name, department)*:
<'CP302', 'Database', NULL>

Note that the above insertion procedure is somewhat tedius. It can be made a little less tedius if the list of attribute values that are to be inserted are in the same order as specified in the relation definition and all the attribute values are present. We may then write:

INSERT INTO *student* <99, 'John Smith', NULL, NULL>

Most database systems provide other facilities for loading a database. Also note that often, when one wishes to insert a tuple, one may not have all the values of the attributes. These attributes, unless they are part of the primary key of the relation, may be given NULL values.

Data Definition

We briefly discuss some of the data definition facilities available in the language. To create the relations *student*, *enrolment* and *subject*, we need to use the following commands:

CREATE TABLE *student*
(*student_id* INTEGER NOT NULL,
student_name CHAR(15),
address CHAR(25))

In the above definition of the relation, we have specified that the attribute *student_id* may not be NULL. This is because *student_id* is the primary key of the relation and it would make no sense to have a tuple in the relation with a NULL primary key.

There are a number of other commands available for dropping a table, for altering a table, for creating an index, dropping an index and for creating and dropping a view. We discuss the concept of view and how to create and drop them.

Views

We have noted earlier that the result of any SQL query is itself a relation. In normal query sessions, a query is executed and the relation is materialized immediately. In other situations it may be convenient to store the query definition as a definition of a relation. Such a relation is often called a *view*. A view is therefore a virtual relation, it has no real existence since it is defined as a query on the existing base relations. The user would see a view like a base table and all SELECT-FROM query commands may query views just like they may query the base relations.

The facility to define views is useful in many ways. It is useful in controlling access to a database. Users may be permitted to see and manipulate only that data that is visible through the views. It also provides logical independence in that the user dealing with the database through a view does not need to be aware of the relations that exist since a view may be based on one or more base relations. If the structure of the base relations is changed (e.g. a column added or a relations split in two), the view definition may need changing but the users view will not be affected.

As an example of view definition we define CP302 students as

CREATE VIEW DOING_CP302 (*student_id, student_name, Address*)
AS SELECT *student_id, student_name, Address*
FROM *enrolment, student*
WHERE *student.student_id = enrolment.student_id*
AND *subject_id* = 'CP302';

Another example is the following view definition that provides information about all the tutors.

CREATE VIEW *tutors* (*name, id, address*)
AS SELECT (*student_name, student_id, address*)
FROM *student*
WHERE *student_id* IN
(SELECT *tutor_id*
FROM *student*)

Note that we have given attribute names for the view *tutors* that are different than the attribute names in the base relations.

As noted earlier, the views may be used in retrieving information as if they were base relations. When a query uses a view instead of a base relation, the DBMS retrieves the view definition from meta-data and uses it to compose a new query that would use only the base relations. This query is then processed.

Advantages of SQL

There are several advantages of using a very high-level language like SQL. Firstly, the language allows data access to be expressed without mentioning or knowing the existence of specific access paths or indexes. Application programs are therefore simpler since they only specify what needs to be done not how it needs to be done. The DBMS is responsible for selecting the optimal strategy for executing the program.

Another advantage of using a very high-level language is that data structures may be changed if it becomes clear that such a change would improve efficiency. Such changes need not affect the application programs.

Review Question

1. Explain and List down various command in DML.
2. Give the command for creating table.
3. What are views? Give the command for creating view.
4. List down the advantages of SQL.

References

1. Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000
2. Elamasri R . and Navathe, S., Fundamentals of Database Systems (3rd Edition), Pearsson Education, 2000.
3. http://www.cs.ucf.edu/courses/cop4710/spr2004

Summary

The most popular query language is SQL. SQL has now become a de facto standard for relational database query languages. We discuss SQL in some detail. SQL is a non-procedural language that originated at IBM during the building of the now famous experimental relational DBMS called System R. Originally the user interface to System R was called SEQUEL which was later modified and its name changed to SEQUEL2. These languages have undergone significant changes over time and the current language has been named SQL (Structured Query Language) although it is still often pronounced as if it was spelled SEQUEL. Recently, the American Standards Association has adopted a standard definition of SQL (Date, 1987). A user of a DBMS using SQL may use the query language interactively or through a host language like C or COBOL. We will discuss only the interactive use of SQL although the SQL standard only defines SQL use through a host language.

Notes:

LESSON 16
SQL SUPPORT FOR INTEGRITY CONSTRAINTS

Hi! We are going to discuss SQL support for integrity constraints.

Types of Integrity Constraints

1. Non-null
2. Key
3. Referential integrity
4. Attribute-based
5. Tuple-based
6. General assertions

Non-Null Constraints

Restricts attributes to not allow NULL values

Ex: CREATE TABLE Student (ID integer NOT NULL,
name char(30) NOT NULL,
address char(100),
GPA float NOT NULL,
SAT integer)

Ex: ID is key for Student => no two tuples in Student can have the same values for their ID attribute

There are two kinds of keys in SQL, Key Constraints

1. PRIMARY KEY: at most one per relation, automatically non-null, automatically indexed (in Oracle)
2. UNIQUE: any number per relation, automatically indexed

There are two ways to define keys in SQL:

a. With key attribute
b. Separate within table definition

Ex: CREATE TABLE Student (ID integer PRIMARY KEY,
name char(30),
address char(100),
GPA float,
SAT integer,
UNIQUE (name,address))

Referential Integrity

Referenced attribute must be PRIMARY KEY

(e.g., Student.ID, Campus.location)

Referencing attribute called FOREIGN KEY

(e.g., Apply.ID, Apply.location)

There are two ways to define referential integrity in SQL:

1. With referencing attribute
2. Separate within referencing relation

Ex: CREATE TABLE Apply(ID integer REFERENCES Student(ID),
location char(25),
date char(10),
major char(10),
decision char,
FOREIGN KEY (location) REFERENCES Campus(location))

Can omit referenced attribute name if it's the same as referencing attribute:

ID integer REFERENCES Student, ...

Can have multi-attribute referential integrity. Can have referential integrity within a single relation.

Ex: Dorm(first-name,
last-name,
room-number,
phone-number,
roommate-first-name,
roommate-last-name,
PRIMARY KEY (first-name,last-name),
FOREIGN KEY (roommate-first-name,roommate-last-name)
REFERENCES Dorm(first-name,last-name))

A foreign key is a group of attributes that is a primary key is some other table. Think of the foreign key values as pointers to tuples in another relation. It is important to maintain the consistency of information in the foreign key field and the primary key in the other relation. In general the foreign key should not have values that are absent from the primary key, a tuple with such values is called a dangling tuple (akin to a dangling pointer in programming languages).

Referential integrity is the property that this consistency has been maintained in the database.

A foreign key in the *enrolledIn* table: Assume that the following tables exist. ·

- *student*(*name*, *address*) - *name* is the primary key
- *enrolledIn*(*name*, *code*) - *name*, *code* is the primary key
- *subject*(*code*, *lecturer*) - *code* is the primary key

The *enrolledIn* table records which students are enrolled in which subjects. Assume that the tuple (*joe*, *cp2001*) indicates that the student *joe* is enrolled in the subject *cp2001*. This tuple only models reality if in fact *joe* is a student. If *joe* is not a student then we have a problem, we have a record of a student enrolled in some subject but no record that that student actually exists! We assert this reasonable conclusion by stating that for the *enrolledIn* table, *name* is a foreign key

into the *student* table. It turns out that *code* is also a foreign key, into the *subject* table. Students must be enrolled in subjects that actually exist. If we think about the E/R diagram from which these relations were determined, the *enrolledIn* relation is a relationship type that connects the *student* and *subject* entity types. Relationship types are the most common source of foreign keys constraints.

In SQL, a foreign key can be defined by using the following syntax in the column specification portion of a CREATE TABLE statement.

< column name > < column type > ... REFERENCES ***< table name > ...,***

Let's look at an example.

Creating the *enrolledIn* table:

```
CREATE TABLE enrolledIn (
        name VARCHAR(20) REFERENCES student,
        code CHAR(6) REFERENCES subject
        );
```

Alternatively, if more than one column is involved in a single foreign key, the foreign key constraint can be added after all the column specifications.

CREATE TABLE *<table name>* (

< column specification > ...
REFERENCES(*<list of columns>*) *<table name>*, ...

);

Let's look at an example.

Creating the *enrolledIn* table:

```
CREATE TABLE enrolledIn (
name    VARCHAR(20),
code            CHAR(6),
REFERENCES(name) student,
REFERENCES(code) subject
);
```

Policies for Maintaining Integrity

The foreign key constraint establishes a relationship that must be maintained between two relations, basically, the foreign key information must be a subset of the primary key information. Below we outline the *default* SQL policy for maintaining referential integrity.

- Allow inserts into primary key table.
- Reject updates of primary key values (potentially must update foreign key values as well).
- Reject deletion of primary key values (potentially must delete foreign key values as well).
- Reject inserts into foreign key table if inserted foreign key is not already a primary key.
- Reject updates in foreign key table if the updated value is not already a primary key.
- Allow deletion from foreign key table.

Updating the student mini-world: By default SQL supports the following.

- Inserts into the *student* table are allowed.
- Updates of the *name* attribute in the *student* table are rejected since it could result in a dangling tuple in the *enrolledIn* table.
- Deleting a tuple from the *student* table is rejected since it could result in a dangling tuple in the *enrolledIn* table.
- Inserts into the *enrolledIn* table are permitted only if *name* already exists in the *student* table (and *code* in the *subjects* table).
- Updates of the *name* attribute in the *enrolledIn* table are permitted only if the updated *name* already exists in the *student* table.
- Deleting a tuple from the *enrolledIn* table is permitted.

While this is the default (and most conservative) policy, individual vendors may well support other, more permissive, policies.

Constraints

In SQL we can put constraints on data at many different levels.

Constraints on attribute values

These constraints are given in a CREATE TABLE statement as part of the column specification. We can define a column to be NOT NULL specifying that no null values are permitted in this column. The constraint is checked whenever a insert or update is made to the constrained column in a table.

A general condition can be added to the values in a column using the following syntax after the type of a column.

CHECK *<condition>*

The condition is checked whenever the column is updated. If the condition is satisfied, the update is permitted, but if the check fails, the update is rejected (hopefully with an apropos error message!).

A column for logins: Suppose that the *login* column in a student table should conform to JCU style logins (e.g., sci-jjj or jc222222). In the CREATE TABLE statement, the *login* column can be defined as follows.

Login CHAR(10) CHECK (login LIKE '___-%' OR login LIKE 'jc%') Recall that LIKE is a string matching operator. The '_' character will match a single possible ASCII character while '%' will match zero or more ASCII characters. The condition is checked whenever a new login is inserted or a login is updated.

Another way to check attribute values is to establish a DOMAIN and then use that DOMAIN as the type of a column.

RepresentingGender: Below we create a DOMAIN for representing gender. CREATE DOMAIN genderDomain CHAR(1) CHECK (value IN ('F','M')); This DOMAIN only has two possible values 'F' and 'M'. No other value can be used in a column of genderDomain type. In the CREATE TABLE statement the created DOMAIN can be used like any other type in the column specification.

```
CREATE TABLE student (
        …
        gender genderDomain,
        …
        );
```

A user will only be able to insert or update the gender to some value in the genderDomain.

Constraints on Tuples

Constraints can also be placed on entire tuples. These constraints also appear in the CREATE TABLE statements; after all the column specifications a CHECK can be added.

CREATE TABLE <*table name*> (

...

CHECK <*condition on tuples*>, ...

);

Too many male science students: Let's assume that the powers that be have decided that there are way too many male science students and so only female science students will be allowed. If we assume that all science students have a login that starts with 'sci-' then we can enforce this constraint using the following syntax.

```
CREATE TABLE student (
        …
        gender genderDomain,
        login CHAR(10) CHECK (login LIKE
        '___-%' OR login LIKE 'jc%'),
        …
        CHECK (login LIKE 'sci-%' AND gender
=               'F'),
);
```

Constraints on Tuples in More than One Tables

Constraints can also be added to tuples in several tables. Since these constraints involve more than one table they are not part of the CREATE TABLE statement, rather they are declared separately as follows.

CREATE ASSERTION <*assertion name*> **CHECK** <*constraint*>;

The constraint that is checked usually involves a SELECT.

Example: Science enrollments are down and so the word has gone out that no science student can be enrolled in fewer than five subjects!

```
CREATE ASSERTION RaiseScienceRevenue
CHECK
        5 > ALL
                (SELECT COUNT(code)
                FROM EnrolledIn, Student
                GROUP BY code
                WHERE login LIKE 'sci-%');
```

Later after there are no more science students the administrators decide this is a stupid idea and choose to remove the constraint.

```
DROP ASSERTION RaiseScienceRevenue;
```

General Assertions

Constraints on entire relation or entire database. In SQL, standalone statement:

CREATE ASSERTION CHECK ()

Ex: Average GPA is > 3.0 and average SAT is > 1200

```
CREATE ASSERTION HighVals CHECK(
3.0 < (SELECT avg(GPA) FROM Student) AND
1200 < (SELECT avg(SAT) FROM Student))
```

Ex: A student with GPA < 3.0 can only apply to campuses with rank > 4.

```
CREATE ASSERTION RestrictApps CHECK(
 NOT EXISTS (SELECT * FROM Student, Apply, Campus
        WHERE Student.ID = Apply.ID
        AND Apply.location = Campus.location
        AND Student.GPA < 3.0 AND Campus.rank <= 4))
```

Assertions checked for each modification that could potentially violate them.

Triggers

A trigger has three parts.

1. Event - The trigger waits for a certain event to occur. Events are things like an insertion into a relation.
2. Condition - Once an event is detected the trigger checks a condition (usually somehow connected to the event - e.g., after an insertion check to see if the salary attribute is non-negative). If the condition is false then the trigger goes back to sleep, waiting for the next relevant event. If the condition is true then an *action* (see below) is executed. In effect the trigger is waiting for events that satisfy a certain condition.
3. Action - An action is a piece of code that is executed. Usually this will involve either undoing a transaction or updating some related information.

Shoe salaries cannot be lowered: Suppose that we wanted to ensure that people in the shoe department can *never* have their salary *lowered*. We would like to set up a trigger that waits for update events to the employee relation, and if those updates lower the salary of a shoe employee, then the update should be undone. In SQL3 we could set up the trigger using the following statement.

```
CREATE TRIGGER ShoeSalaryTrigger
AFTER UPDATE OF salary ON employee
REFERENCING
        OLD AS OldTuple
        NEW AS NewTuple
WHEN (OldTuple.salary > NewTuple.salary)
UPDATE employee
SET salary = OldTuple.salary FOR EACH ROW
```

In this statement the *event specification* is in the second line, the condition is the sixth line, and the action is the seventh and eighth lines. The third, fourth, and fifth lines establish tuple variables, OldTuple and NewTuple, that are the original and updated tuple in the employee table. The trigger is activated just after r*ather* than just *before* the update, as specified in line 2. Finally, the ninth line specifies that each row (tuple) in the employee table must be checked.

The example above demonstrates that triggers are useful in situations where an arbitrary action must be performed to maintain the consistency and quality of data in a database. But each trigger imposes an overhead on database processing, not only to wake up triggers and check conditions, but to perform the associated action. So triggers should be used judiciously.

Recursion

The proposed SQL3 standard extends SQL in an interesting direction. A well-known limitations of SQL is that it does not have recursion, or a "looping" mechanism. For example in SQL (or with relations) we can represent a graph by maintaining a table of edges.

from	to
a	**b**
b	**c**
c	**d**

The *Edge* table

Within this graph a simple SQL query can be used to compute nodes that are connected by paths of length two by joining the *Edge* table to itself.

```
SELECT A.from, B.to
FROM Edge as A, Edge as B
WHERE A.to = B.from;
```

The result of this query is given below.

from	to
a	**a**
b	**d**

The result of *Edge* ⋈*Edge*

For this graph there is also a path of length three from *a* to *d* via *b* and *c*, so to get the transitive closure we would have to do a further join with *Edge*. In general since we can't know in advance how many joins need to be done, we need to have some kind of loop where we keep joining *Edge* to the result of the previous round to derive new connections in the graph. SQL3 has such a recursive construct.

WITH RECURSIVE .... TODO

Declaring and Enforcing Constraints

Two times at which constraints may be declared:

1. Declared with original schema.
   Constraints must hold after bulk loading.
2. Declared later.
   Constraints must hold on current database.

After declaration, if a SQL statement causes a constraint to become violated then (in most cases) the statement is aborted and a run-time error is generated.

Notes:

LESSON 17
DATABASE DESIGN INCLUDING INTEGRITY CONSTRAINTS-PART-I

Hi! Here in this session we are actually beginning to design a database. So you are going to learn about how a good database could be designed to meet user requirements and also that will cater the needs of a consistent database.

Planning the Relational Database

When you need to build a database, there is a temptation to immediately sit down at the computer, fire up your RDBMS, and start creating tables. Well, don't. There is a process you need to follow to develop a well-designed relational database and, at the start, you're a long way from actually setting up the tables in the database application. Not necessarily a long way in time, but certainly in thought.

A systematic approach to the design will save you, the designer, a lot of time and work and makes it much more likely that the "client" for the database will get something that fits the need. In this topic, you'll look at the steps of a design process that you will follow.

When you get to the point of drafting your tables and fields, you're going to use a very low tech approach to the design process- pencil and paper. You'll find lots of blank spaces in this manual to work in. When you start building databases on your own, if you're the kind of person who just cannot think unless you're looking at a computer screen, there are software tools available for modeling a database. These CASE (computer-aided software engineering) tools can be used to create diagrams and some will create documentation of the design; they can be particularly useful when a team is working on the design of a database. Additionally, some CASE products can generate commands that will actually create the tables in the RDBMS.

The idea is to draw a picture of your tables and fields and how the data in the tables is related. These are generally called entity-relationship, ER, or E/R diagrams. There are various formal systems for creating these diagrams using a specific set of symbols to represent certain objects and types of relationships. At this point in your design career, you should probably use whatever works for you. Again, a formal system becomes more useful when a group of people are working on the same design. Also, using a recognized method is helpful for documenting your design for those who come after you. For additional information on ER diagrams, you may want to read Entity-Relationship Approach to Information Modeling by P. Chen.

The Design Process

The design process includes

- Identify the purpose of the database.
- Review existing database.
- Make a preliminary list of fields.
- Make a preliminary list of tables and enter the fields.
- Identify the key fields.
- Draft the table relationships.
- Enter sample data and normalize the data.
- Review and finalize the design.

Following a design process merely ensures that you have the information you need to create the database and that it complies with the principles of a relational database. In this topic, you're going to use this process to get the point of having well-designed tables and relationships and understand how you can extract data from the tables. After that, you, as the designer, may also have to create additional objects for the application, such as the queries, forms, reports, and application control objects. Most of those tasks are application-specific and are beyond the scope of this topic.

In this topic, you'll review an outline of the process. You'll go through the first few steps of identifying the purpose of the database and, in subsequent topics, will design the tables and relationships. You're the database designer and the information contained represents the client (the person(s) who has expressed the need for a database).If you wish, you can work on a database of your own where you can be both the client and the designer. Then you have nobody to blame but yourself if it doesn't come out right.

So, where to begin?

1. Identify the purpose of the database.

You will rarely be handed a detailed specification for the database. The desire for a database is usually initially expressed as things the client wants it to do. Things like:

- We need to keep track of our inventory.
- We need an order entry system.
- I need monthly reports on sales.
- We need to provide our product catalog on the Web.

It will usually be up to you to clarify the scope of the intended database. Remember that a database holds related information. If the client wants the product catalog on the Web and sales figures and information on employees and data on competitors, maybe you're talking about more than one database. Everyone involved needs to have the same understanding of the scope of the project and the expected outcomes (preferably in order of importance). It can be helpful to write a statement of purpose for the database that concerned parties can sign off on. Something like: "The Orders database will hold information on customers, orders, and order details. It will be used for order entry and monthly reports on sales." A statement like this can help define the boundaries of the information the database will hold.

The early stages of database design are a people-intensive phase, and clear and explicit communication is essential. The process is not isolated steps but is, to a point, iterative. That is, you'll have to keep going back to people for clarification and additional information as you get further along in the process. As your design progresses, you'll also need to get confirmation that you're on the right track and that all needed data is accounted for.

If you don't have it at the beginning of the design process, along the way you'll also need to develop an understanding of the way the business operates and of the data itself. You need to care about business operations because they involve business rules. These business rules result in constraints that you, as the database designer, need to place on the data. Examples include what the allowable range of values is for a field, whether a certain field of data is required, whether values in a field will be numbers or characters, and will numbers ever have leading zeros. Business rules can also determine the structure of and relationship between tables. Also, it will be difficult for you to determine what values are unique and how the data in different tables relates if you don't understand the meaning of the data. The reasons will be clearer when you actually get to those points in the process.

2. Review Existing Data.

You can take a huge step in defining the body of information for the database by looking at existing data repositories. Is there an existing database (often called a legacy database) even if it isn't fitting the bill anymore? Is someone currently tracking some of the information in spreadsheets? Are there data collection forms in use? Or are there paper files?

Another good way to help define the data is to sketch out the desired outcome. For example, if the clients say they need a monthly report of sales, have them draft what they have in mind. Do they want it grouped by product line? By region? By salesperson? You can't provide groupings if you haven't got a field containing data you can group on. Do they want calculations done? You can't perform calculations if you haven't stored the component values.

Your goal is to collect as much information as you can about the desired products of the database and to reverse engineer that information into tables and fields.

3. Make a Preliminary List of Fields.

Take all you have learned about the needs so far and make a preliminary list of the fields of data to be included in the database. Make sure that you have fields to support the needs. For example, to report on monthly sales, there's going to have to be a date associated with each sale. To group sales by product line, you'll need a product line identifier. Keep in mind that the clients for a database have expressed their need for information; it's your job to think about what data is needed to deliver that information.

Each field should be atomic; this means each should hold the smallest meaningful value and, therefore, should not contain multiple values. The most common disregard of this rule is to store a person's first name and last name in the same field.

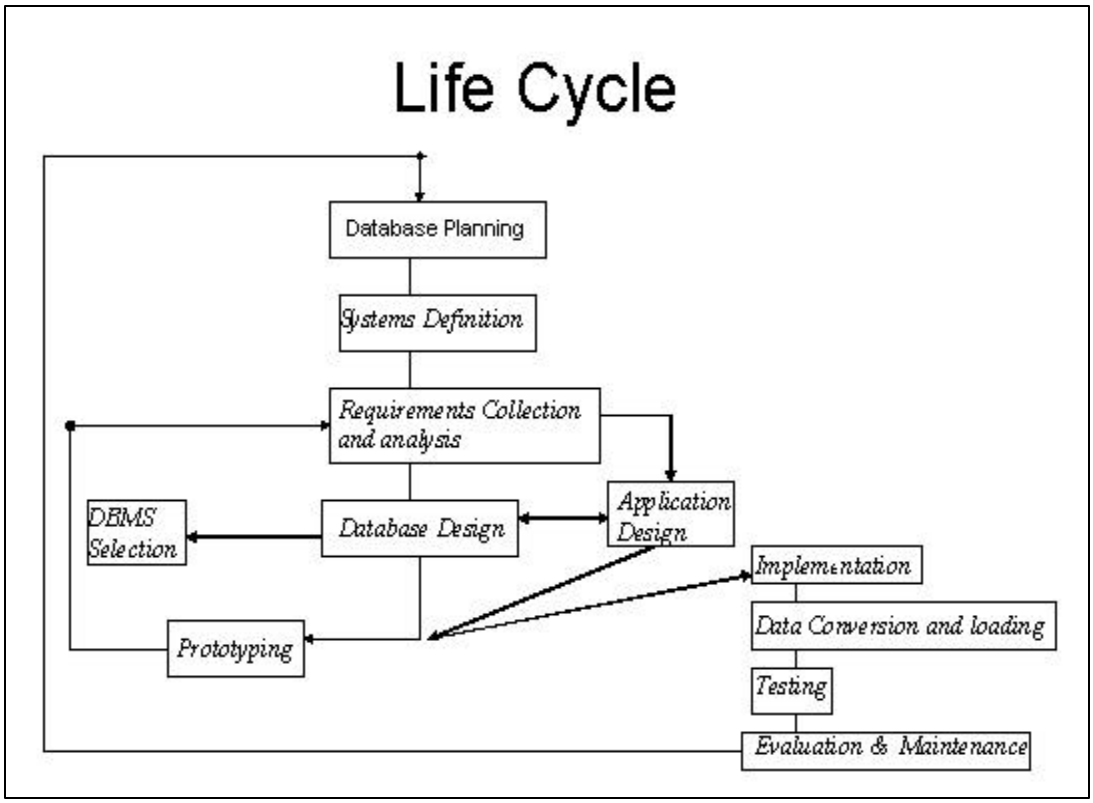
Do not include fields to hold data that can be calculated from other fields. For example, if you had fields holding an employee's hourly pay rate and weekly hours, you would not include a gross pay field.

Database Development Life Cycle

Hi, we have reached up to a point where, like the classical software development life cycle, I would like to discuss with you the various phases in the Database Development Life Cycle. Let us explore this interesting concept.

In this phase the database designers will decide on the database model that is ideally suited or the organization's needs. The database designers will study the documents prepared by the analysts in the requirements analysis phase and then will go about developing a system that satisfies the requirements. In this phase the designers will try to find out answers to the following questions:

- What are the problems in the existing system and how they could be overcome?
- What are the information needs to the different users of the system and how could the conflicting requirements be balanced?
- What data items are required for an efficient decision-making system?
- What are the performance requirements for the system?
- How should the data be structured?
- How will each user access the data?
- How is the data entered in to the database?
- How much data will be added to the database on a daily/weekly/monthly basis?



Designing Good Relational Databases

Hi now you have understood the necessity of the Database design phase. Databases have a reputation for being difficult to construct and hard to maintain. The power of modern database software makes it possible to create a database with a few mouse-clicks. The databases created this way, however, are typically the databases that are hard to maintain and difficult to work with because they are designed poorly. Modern software makes it easy to construct a database, but doesn't help much with the design aspect of database creation.

Database design has nothing to do with using computers. It has everything to do with research and planning. The design process should be completely independent of software choices. The basic elements of the design process are:

1. Defining the problem or objective
2. Researching the current database
3. Designing the data structures
4. Constructing relationships
5. Implementing rules and constraints
6. Creating views and reports
7. Implementing the design

Notice that implementing the database design in software is the final step. All of the preceding steps are completely independent of any software or other implementation concerns.

Defining the problem or objective. The most important step in database design is the first one: defining the problem the database will address or the objective of the database. It is important however, to draw a distinction between:

- How the database will be used and
- What information needs to be stored in it?

The first step of database design is to clearly delineate the nature of the data that needs to be stored, not the questions that will be asked to turn that data into information.

This may sound a little contradictory at first, since the purpose of a database is to provide the appropriate information to answer questions. However, the problem with designing databases to answer specific or targeted questions is that invariably questions are left out, change over time, or even become superseded by other questions. Once this happens, a database designed solely to answer the original questions becomes useless. In contrast, if the database is designed by collecting all of the information that an individual or organization uses to address a particular problem or objective, the information to answer any question involving that problem or objective can theoretically be addressed.

Researching the current database. In most database design situations, there is some sort of database already in existence. That database may be Post-it notes, paper order forms, a spreadsheet of sales data, a word processor file of names and addresses, or a full-fledged digital database (possibly in an outdated software package or older legacy system). Regardless of its format, it provides one essential piece of information: the data that the organization currently finds useful. This is an excellent starting point for determining the essential data structure of the database. The existing database information can also provide the nucleus for the content of the new database.

Designing the data structures. A database is essentially a collection of data tables, so the next step in the design process is to identify and describe those data structures. Each table in a

database should represent some distinct subject or physical object, so it seems reasonable to simply analyze the subjects or physical objects relevant to the purpose of the database, and then arrive at a list of tables.

This can work successfully, but it's a much better to objectively analyze the actual fields that you have identified as essential in your research and see what logical groupings arise. In many cases, structures that seemed distinct are really reflections of the same underlying subject. In other cases, the complete opposite is true. And to complicate matters, organizations can use the same terms to describe data that they use or collect in fundamentally different ways.

Once the tables have been determined and fields have been assigned to each, the next step is to develop the specifications for each field. The perfect field should be atomic: It should be unique in all tables in the database (unless it is used as a key) and contain a single value, and it should not be possible to break it into smaller components. This is also an appropriate time to start thinking about the kind of data that goes in each field. This information should be fairly clear from the research phase of the project, but sometimes questions remain. Some advance planning can be done to make it easier to implement the database in the software at a later time, such as identifying the type of fields and examining (or re-examining) existing data that you've collected to make sure that the data always fits the model you are constructing. It's much easier and cheaper to fix that now than wait until the database is being rolled out!

Constructing relationships. Once the data structures are in place, the next step is to establish the relationships between the databases. First you must ensure that each table has a unique key that can identify the individual records in each table. Any field in the database that contains unique values is an acceptable field to use as a key. However, it is a much better practice to add an arbitrary field to each table that contains a meaningless, but unique value. This value is typically an integer that is assigned to each record as it is entered and never again repeated. This ensures that each entered record will have a unique key.

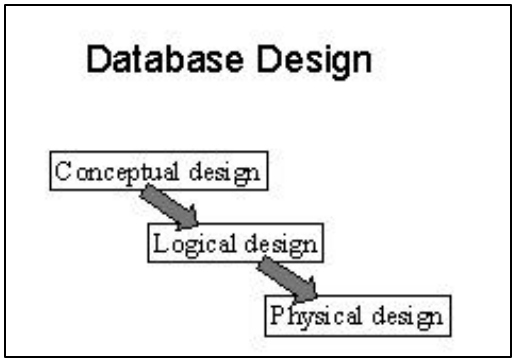
Implementing rules and constraints. In this step, the fields in the database are still fairly amorphous. Defining the fields as text or numeric and getting a rough feel for the types of data that the client needs to store has narrowed them down, but there is room for further refinement. Rules and constraints typically lead to cleaner data entry and thus better information when using the data. Business rules and constraints limit the format that data can take or the ways that data tables can be related to other data tables.

Some of these constraints are imposed by the nature of the data itself; social security numbers are always in the same nine-digit format. This type of constraint is normally implemented to make sure that data is complete and accurate. In other cases, the situation itself explicitly constrains the data. The possible values for the data are usually checked against a list or the choice of values is otherwise constrained. This type of constraint is usually easy to implement and easy to change.

Creating views and reports. Now that the data design is essentially complete, the penultimate step is to create the specifications that help turn the data into useful information in the form of a report or view of the data. Views are simply collections of the data available in the database combined and made accessible in one place. It could be as simple as a subset of an existing data table or as complicated as a collection of multiple tables joined on particular set of criteria. Reports on the other hand, are typically snapshots of the database at a particular point in time.

Implementing the design in software. All of the work to this point has been accomplished without explicitly worrying about the details of the program being used to produce the database. In fact, the design should only exist as diagrams and notes on paper. This is especially important at a later point when you or someone else need to update the database or port it to another package. Now it's time to boot the computer and get started.

Database Design



Conceptual Design

Now we will start with the actual design, the first stage of database designing.

In the conceptual design stage, data model is used to create an abstract database structure that represents the real world scenario.

The Conceptual Design

- Complete understanding of database structure, semantics, constraints, relationships etc
- DBMS independent
- Stable description
- Database users and application users views; aids their understanding
- Communication with users
- True representation of real world

The different steps in the conceptual design are as follows

1. Data Analysis and requirements definition
2. Data Modelling and normalization

Data Analysis and Requirements Definition

In this step the data item and their characteristics are determined. The data items that are required for successful information processing and decision making are identified and their characteristics are recorded. Questions like what kind of information is needed, what outputs (reports and queries) should the system generate who will use the information, how and for what purpose it will be used, what are the sources of the information, etc ; will be answered in this stage.

Data Modelling and Normalization

In this step the database designer creates a data model of the system. The business contains entities and relationships. Each entities will have attributes. In this stage the business entities and relationships are transformed in to a data model usually an E-R Model, using E-R Diagrams. Now many designers have started using data modeling using UML (Unified Modeling Language) instead of E-R diagrams. Once the data model is created then the data will be available in a structured form.

All objects (Entities, relations and so on) are defined in a data dictionary and the data is normalized. During he process the designer will group the data items, define the tables, identify the primary keys, define the relationships (One to One, One to Many and many to Many), Create the data model, normalize the data model and so on. Once the data model is created it is verified against the proposed system in order to ascertain that the proposed model is capable of supporting the real world system. So the data model is tested to find out whether the model can perform various database operations and whether the data model takes care of the issue of the data security, integrity, and concurrency and so on.

Review Questions

1. How will you go about in planning the database?
2. Explain the Database Development Life cycle?
3. Explain the conceptual design?

References

http://www.microsoft-accesssolutions.co.uk

Date, C, J, Introduction to Database Systems, 7th edition

Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld.

Notes:

LESSON 18
DATABASE DESIGN INCLUDING INTEGRITY CONSTRAINTS-PART-II

Hi, we will learn about the importance of logical database design in this lecture.

Creating a database logical design is one of the first important steps in designing a database. There are four logical database models that can be used hierarchical, network, relational, or object-oriented. The design concepts should first start with a data model and those models have to b transformed to the physical considerations which will be specific to the DBMS, that is going to be selected.

What Exactly Is Logical Database Design?

Logical modeling deals with gathering business requirements and converting those requirements into a model. The logical model revolves around the needs of the business, not the database, although the needs of the business are used to establish the needs of the database. Logical modeling involves gathering information about business processes, business entities (categories of data), and organizational units. After this information is gathered, diagrams and reports are produced including entity relationship diagrams, business process diagrams, and eventually process flow diagrams. The diagrams produced should show the processes and data that exist, as well as the relationships between business processes and data. Logical modeling should accurately render a visual representation of the activities and data relevant to a particular business.

Logical modeling affects not only the direction of database design, but also indirectly affects the performance and administration of an implemented database. When time is invested performing logical modeling, more options become available for planning the design of the physical database.

The diagrams and documentation generated during logical modeling is used to determine whether the requirements of the business have been completely gathered. Management, developers, and end users alike review these diagrams and documentation to determine if more work is required before physical modeling commences.

Logical Modeling Deliverables

Typical deliverables of logical modeling include:-

An Entity Relationship Diagram is also referred to as an analysis ERD. The point of the initial ERD is to provide the development team with a picture of the different categories of data for the business, as well as how these categories of data are related to one another.

The **Business process model** illustrates all the parent and child processes that are performed by individuals within a company. The process model gives the development team an idea of how data moves within the organization. Because process models illustrate the activities of individuals in the company, the process model can be used to determine how a database application interface is design.

So the logical database design is the process of constructing a model of information used in tan enterprise based on a specific data model, but independent of a particular DBMS or other physical considerations.

Why a logical data model is required?

A logical data model is required before you can even begin to design a physical database. And the logical data model grows out of a conceptual data model. And any type of data model begins with the discipline of data modeling.

The first objective of conceptual data modeling is to understand the requirements. A data model, in and of itself, is of limited value. Of course, a data model delivers value by enhancing communication and understanding, and it can be argued that these are quite valuable. But the primary value of a data model is its ability to be used as a blueprint to build a physical database.

When databases are built from a well-designed data model the resulting structures provide increased value to the organization. The value derived from the data model exhibits itself in the form of minimized redundancy, maximized data integrity, increased stability, better data sharing, increased consistency, more timely access to data, and better usability. These qualities are achieved because the data model clearly outlines the data resource requirements and relationships in a clear, concise manner. Building databases from a data model will result in a better database implementation because you will have a better understanding of the data to be stored in your databases.

Another benefit of data modeling is the ability to discover new uses for data. A data model can clarify data patterns and potential uses for data that would remain hidden without the data blueprint provided by the data model. Discovery of such patterns can change the way your business operates and can potentially lead to a competitive advantage and increased revenue for your organization.

Data modeling requires a different mindset than requirements gathering for application development and process-oriented tasks. It is important to think “what” is of interest instead of “how” tasks are accomplished. To transition to this alternate way of thinking, follow these three “rules”:

- Don’t think physical; think conceptual - do not concern yourself with physical storage issues and the constraints of any DBMS you may know. Instead, concern yourself with business issues and terms.
- Don’t think process; think structure - how something is done, although important for application development, is not important for data modeling. The things that processes are being done to are what is important to data modeling.
- Don’t think navigation; think relationship - the way that things are related to one another is important because

relationships map the data model blueprint. The way in which relationships are traversed is unimportant to conceptual and logical data modeling.

E-R Diagrams- A Graphical Format for Data Modeling

Data models are typically rendered in a graphical format using an entity-relationship diagram, or E/R diagram for short. An E/R diagram graphically depicts the entities and relationships of a data model. There are many popular data modeling tools on the market from a variety of vendors. But do not confuse the tool as being more important than the process. Of what use is a good tool if you do not know how to deploy it?

A data model is built using many different components acting as abstractions of real world things. The simplest data model will consist of entities and relationships. As work on the data model progresses, additional detail and complexity is added. Let's examine the many different components of a data model and the terminology used for data modeling.

The first building block of the data model is the entity. An entity, at a very basic level, is something that exists and is capable of being described. It is a person, place, thing, concept, or event about which your organization maintains facts. For example: "STUDENT," "INSTRUCTOR," and "COURSE" are specific entities about which a college or university must be knowledgeable to perform its business.

Entities are comprised of attributes. An attribute is a characteristic of an entity. Every attribute does one of three things:

1. Describe - An attribute is descriptive if it does not identify or relate, but is used to depict or express a characteristic of an entity occurrence.
2. Identify -An attribute that identifies is a candidate key. If the values of an identifying attribute changes, it should identify a different entity occurrence. An attribute that identifies should be unchangeable and immutable.
3. Relate - An attribute that relates entities is a foreign key; the attribute refers to the primary key attribute of an occurrence of another (or the same) entity.

Each attribute is assigned a domain that defines the type of data, its size, and the valid values that can be assigned to the attribute. As a general rule of thumb, nouns tend to be entities and adjectives tend to be attributes. But, of course, this is not a hard and fast rule: be sure to apply of the business to determine which nouns and attributes are entities and which are attributes. Every attribute must either identify the entity occurrence, describe the entity occurrence, or relate the entity occurrence to another entity occurrence (in the same or another entity).

Relationships define how the different entities are associated with each other. Each relationship is named such that it describes the role played by an entity in its association with another (or perhaps the same) entity. A relationship is defined by the keys of the participating entities: the primary key in the parent entity and the foreign key in the dependent entity. Relationships are not just the "lines" that connect entities, but provide meaning to the data model and must be assigned useful names.

Keep in mind that as you create your data models, you are developing the lexicon of your organization's business. Much like a dictionary functions as the lexicon of words for a given language, the data model functions as the lexicon of business terms and their usage. Of course, this short introduction just scrapes the tip of the data modeling iceberg.

Bad logical database design results in bad physical database design, and generally results in poor database performance. So, if it is your responsibility to design a database from scratch, be sure you take the necessary time and effort to get the logical database design right. Once the logical design is right, then you also need to take the time to get the physical design right.

Review Questions

1. What do you mean by logical database design?
2. What are the deliverables for logical modeling?

References

http://www.microsoft-accesssolutions.co.uk

Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000

Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld.

Notes:

LESSON 19
MULTI-USER DATABASE APPLICATION

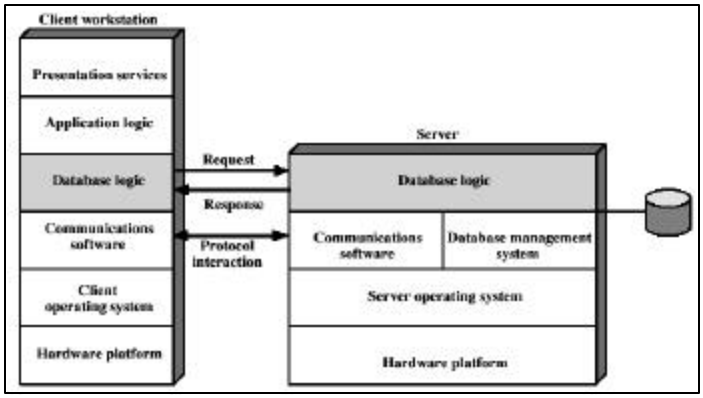
Hi! Today we will discuss one of the finest concepts in the modern world of computers and its relevance in terms of database management systems. Here we are giving emphasis to client-server technologies and distributed databases which facilitates the multi user environment. The security issues in a multi-user environment are dealt in lectures involving database security.

Introduction

Client/server computing is the logical extension of modular programming. Modular programming has as its fundamental assumption that separation of a large piece of software into its constituent parts creates the possibility for easier development and

better maintainability. Client/server computing takes this a step farther by recognizing that those modules need not all be executed within the same memory space. With this architecture, the calling module becomes the "client" (that which requests a service), and the called module becomes the "server" (that which provides the service).The logical extension of this is to have clients and servers running on the appropriate hardware and software platforms for their functions. A "server" sub-system provides services to multiple instances of "client" subsystem Client and server are connected by a network .Control is typically a client requests services from the server provides data access and maintains data integrity To handle load, can have more than one server

Database Client-Server Architecture



What is the function Client?

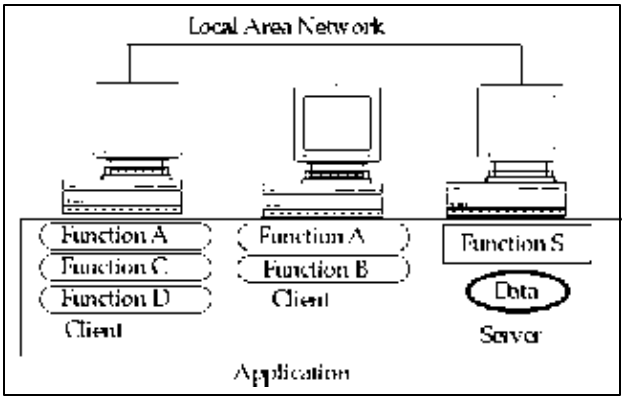
The client is a process that sends a message to a server process, requesting that the server perform a ser vice. Client programs usually manage the user-interface portion of the application, validate data entered by the user, dispatch requests to server programs, and sometimes execute business logic. The client-based process is the front-end of the application that the user sees and interacts with. The client process often manages the local resources that the user interacts with such as the monitor, keyboard, CPU and peripherals. One of the key elements of a client workstation is the graphical user interface (GUI). Normally a par t of operating system i.e. the window manager detects user actions, manages the windows on the display and displays the data in the windows.

What is the function Server?

Server programs generally receive requests from client programs, execute database retrieval and updates, manage data integrity and dispatch responses to client requests. The server-based process may run on another machine on the network. This server could be the host operating system or network file server, providing file system services and application services. The server process often manages shared resources such as databases, printers, communication links, or high powered processors. The server process performs the back-end tasks that are common to similar applications.

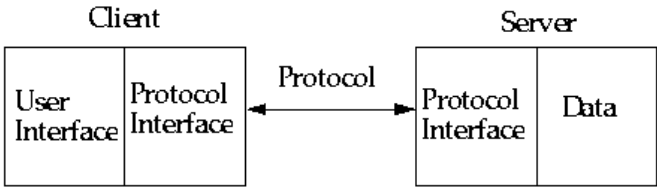
What do mean by Middleware?

- It is a standardized interfaces and protocols between clients and back-end databases.
- It hides complexity of data sources from the end-user
- Compatible with a range of client and server options
- All applications operate over a uniform applications programming interface (API).



Why is Client-Server Different

- Emphasis on user-friendly client applications
- Focus on access to centralized databases
- Commitment to open and modular applications
- Networking is fundamental to the organization



Characteristics of Client/Server Architectures

1. A combination of a client or front-end portion that interacts with the user, and a server or back-end portion that interacts with the shared resource. The client process contains solution-specific logic and provides the interface between the user and the rest of the application system. The server process acts as a software engine that manages shared resources such as databases, printers, modems, or high powered processors.
2. The front-end task and back-end task have fundamentally different requirements for computing resources such as processor speeds, memory, disk speeds and capacities, and input/output devices.
3. The environment is typically heterogeneous and multivendor.The hardware platform and operating system of client and server are not usually the same. Client and server processes communicate through a well-defined set of standard application program interfaces (APIs) and RPCs.
4. An important characteristic of client-server systems is scalability. Horizontal scaling means adding or removing client workstations with only a slight performance impact.

What are the issues in Client-Server Communication?

Addressing

- Hard-wired address
  - Machine address and process address are known a priori
- Broadcast-based
  - Server chooses address from a sparse address space
  - Client broadcasts request
  - Can cache response for future
- Locate address via name server

Blocking Versus Non-blocking

- Blocking communication (synchronous)
  - Send blocks until message is actually sent
  - Receive blocks until message is actually received
- Non-blocking communication (asynchronous)
  - Send returns immediately
  - Return does not block either

Buffering Issues

- Unbuffered communication
  - Server must call receive before client can call send
- Buffered communication
  - Client send to a mailbox
  - Server receives from a mailbox

Reliability

- Unreliable channel
  - Need acknowledgements (ACKs)
  - Applications handle ACKs
  - ACKs for both request and reply
- Reliable channel
  - Reply acts as ACK for request
  - Explicit ACK for response
- Reliable communication on unreliable channels
  - Transport protocol handles lost messages

Server Architecture

- Sequential
  - Serve one request at a time
  - Can service multiple requests by employing events and asynchronous communication
- Concurrent
  - Server spawns a process or thread to service each request
  - Can also use a pre-spawned pool of threads/processes (apache)
- Thus servers could be
  - Pure-sequential, event-based, thread-based, process-based

Scalability

- Buy bigger machine!
- Replicate
- Distribute data and/or algorithms
- Ship code instead of data
- Cache

Client-Server Pros and Cons

- **Advantages**
- Networked web of computers
- Inexpensive but powerful array of processors
- Open systems
- Grows easily
- Individual client operating systems
- Cost-effective way to support thousands of users
- Cheap hardware and software
- Provides control over access to data
- User remains in control over local environment
- Flexible access to information

· **Disadvantages**

- Maintenance nightmares
- Support tools lacking
- Retraining required
- Complexity
- Lack of Maturity
- Lack of trained developers

Distributed Database Concepts

A distributed computing system consists of a number of processing elements, not necessarily homogeneous that are interconnected by a computer network, and that cooperate in performing certain assigned task. As a general goal, distributed computing systems partition a big, unmanageable problem into smaller pieces and solve it efficiently in a coordinated manner. The eonomic viability of this approach stems from two reasons:1) more computer power is harnessed to solve a complex task and 2) each autonomous processing elements can be managed independently and develop its own application.

We can define a **Distributed Database (DDB)** as a collection of multiple logically interrelated databases distributed over a computer network and a **distributed database management system (DDBMS)** as a software system that manages a distributed database while making the distribution transparent to the user

Advantages of Distributed Database

The advantages of distributed database are as follows:

1. Management of distributed data with different levels of transparency:

Ideally a DBMS should be **Distributed Transparent** in the sense of hiding the details of where each file is physically stored within the system.

- Distribution of Network Transparency:

This refers to freedom for the user from the operational details of the network. It may be divided into location transparency and naming transparency. **Location Transparency** refers to the fact that the command used to perform a task is independent of the location of the data and the location of the system where the command was issued. **Naming Transparency** implies that once a name is specified the named objects can be accessed unambiguously without additional specification.

- Replication Transparency:

It makes the user unaware of the existence of the copies of data.

- Fragmentation Transparency:

Two type of fragmentation are possible. **Horizontal Fragmentation** distributes a relation into sets of tuples.**Vertical Fragmentation** distribute a relation into subrelations where each subrelation is defined by a subset of the column of the original relation. Fragmentation transparency makes the user unaware of the existence of fragments

2. Increased Reliability And Availability

Reliability is broadly defined as the probability that a system is running at a certain time point, whereas **availability** is the probability that a system is continuously available during a time interval. So by judiciously Replicating data and data at more than one site in distributed database makes the data accessible in some parts which is unreachable to many users.

3. Improved Performance

A distributed database fragments the database by keeping data closer to where it is needed most. **Data Localization** reduces the contention for CPU and I/O service and simultaneously reduces access delays involved in wide area network. When a large database is distributed over multiple sites smaller database exist at each site .as a result local queries and transactions accessing data at a single site have a better performance because of the smaller local database. In addition each site has a smaller number of transactions executing than if all transactions are submitted to a single centralized database. Moreover interquery and intraquery parallelism can be achieved by executing multiple queries at different sites or by breaking up a query into a number of subqueries at different sites or by breaking up a query into a number of subqueries that execute parallel. This contributes to improved performance.

4. Easier Expansion:

Expansion of the system in terms of adding more data ,increasing database sizes or adding more processors is much easier.

Additional Features Of Distributed Database

Distribution leads to increased complexity in the system design and implementation. To achieve the potential advantages; the DDBMS software must be able to provide the following function in addition to those of a centralized DBMS:

- ***Keeping Track Of data :*** The ability to keep track of the data distribution, fragmentation and replication by expanding the DDBMS catalog.
- ***Distributed query processing:*** the ability to access remote sites and transmit queries and data among the various sites via a communication network.
- ***Distributed transaction management:*** The ability to devise execution strategies for queries and transaction that access data from more than one sites and to synchronize the access to distributed data and maintain integrity of the overall database.
- ***Replicate data management:*** The ability to decide which copy of a replicated data item to access and to maintain the consistency of copies of a replicated data item.
- ***Distributed database recovery:*** the ability to recover from individual site crashes and from new types of failures such as the failure of a communication links.
- ***Security:*** Distributed transaction must be executed with the proper management of the security of the data and the authorization/access privileges of users.
- ***Distributed Directory Management:*** A directory contains information about data in the database. The directory may be global for the entire DDB or local for each site .The placement and distribution of the directory are design and policy issue.

Types of Distributed Database Systems

The term distributed database management system describes various systems that differ from one another in many respects. The main thing that all such systems have in common is the fact that data and software are distributed over multiple sites connected by some form of communication network. The first factor we consider is the **degree of homogeneity** of the DBMS software. If all servers use identical software and all users use identical software ,the DDBMS is called **homogeneous** ;otherwise it is called **heterogeneous** .Another factor related to the degree of homogeneity is the **degree of local autonomy** .If there is no provision for the local site to function as a stand alone DBMS,then the system has no local autonomy. On the other hand if direct access by local transaction to a server is permitted, the system has some degree of local autonomy.

Review Questions

1. What do you understand by Client-Server architecture?
2. What is the function of middleware in C-S architecture?
3. What are the characteristics of C-S architecture?
4. What are the advantages & disadvantages of C-S architecture?

References:

Date, C, J, Introduction to Database Systems, 7th edition

Database Management Systems, By Alexis Leon & Mathews Leon

Notes:

LESSON 20
TWO AND THREE TIER ARCHITECTURE

The Tier

Definition
A *tier* is a distinct part of *hardware* or *software*.

Discussion
The most common *tier* systems are:

- Single Tier
- Two Tier
- Three Tier

Each are defined as follows:

Single Tier

Definition
A single computer that contains a *database* and a *front-end* to access the database.

Discussion
Generally this type of system is found in small businesses. There is one computer which stores all of the company's data on a single database. The *interface* used to interact with the database may be part of the database or another program which ties into the database itself.

Advantages
A *single-tier* system requires only one stand-alone computer. It also requires only one installation of proprietary software. This makes it the most cost-effective system available.

Disadvantages
My be used by only one user at a time. A single tier system is impractical for an organization which requires two or more users to interact with the organizational data store at the same time.

Client/Server

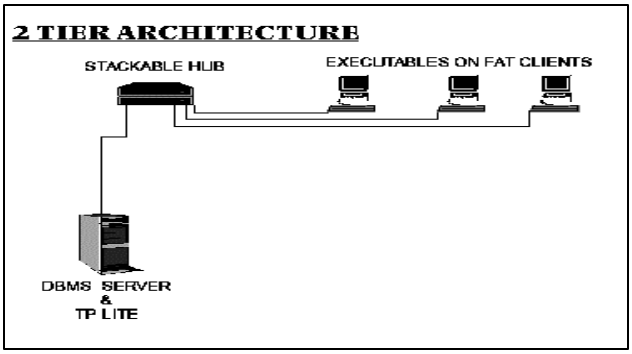
Definition
A *client* is defined as a requester of services and a *server* is defined as the provider of services. A single machine can be both a client and a server depending on the software configuration.

File Sharing Architecture

Definition
Files used by the clients are stored on the server. When files are downloaded to the client all of the processing is done by the client. This processing includes all logic and data.

Discussion
File sharing architectures work if shared usage is low, update contention is low, and the volume of data to be transferred is low. The system gets strained when there are more than 12 users. This system was replaced by the *two tier client/server* architecture.

Two Tier Systems



Definition
A *two tier* system consists of a *client* and a *server.* In a two tier system, the database is stored on the *server,* and the interface used to access the database is installed on the *client.*

Discussion
The user system interface is usually located in the user's desktop environment and the database management services are usually in a server that is a more powerful machine that services many clients. Processing management is split between the user system interface environment and the database management server environment. The database management server provides stored procedures and triggers.

Purpose and Origin

Two tier software architectures were developed in the 1980s from the file server software architecture design. The two tier architecture is intended to improve usability by supporting a forms-based, user-friendly interface. The two tier architecture improves scalability by accommodating up to 100 users (file server architectures only accommodate a dozen users), and improves flexibility by allowing data to be shared, usually within a homogeneous environment The two tier architecture requires minimal operator intervention, and is frequently used in non-complex, non-time critical information processing systems. Detailed readings on two tier architectures can be found in Schussel and Edelstein.

Technical Details
Two tier architectures consist of three components distributed in two layers: client (requester of services) and server (provider of services). The three components are

1. User System Interface (such as session, text input, dialog, and display management services)
2. Processing Management (such as process development, process enactment, process monitoring, and process resource services)
3. Database Management (such as data and file services)

The two tier design allocates the user system interface exclusively to the client. It places database management on the server and splits the processing management between client and server, creating two layers.

Usage Considerations

Two tier software architectures are used extensively in non-time critical information processing where management and operations of the system are not complex. This design is used frequently in decision support systems where the transaction load is light. Two tier software architectures require minimal operator intervention. The two tier architecture works well in relatively homogeneous environments with processing rules (business rules) that do not change very often and when workgroup size is expected to be fewer than 100 users, such as in small businesses.

Advantages:

Since processing was shared between the client and server, more users could interact with such a system.

Disadvantages

When the number of users exceeds 100, performance begins to deteriorate. This limitation is a result of the server maintaining a connection via "keep-alive" messages with each client, even when no work is being done. A second limitation of the two tier architecture is that implementation of processing management services using vendor proprietary database procedures restricts flexibility and choice of DBMS for applications. Finally, current implementations of the two tier architecture provide limited flexibility in moving (repartitioning) program functionality from one server to another without manually regenerating procedural code.

Three Tier Architecture

Purpose and Origin

The three tier software architecture (a.k.a. three layer architectures) emerged in the 1990s to overcome the limitations of the two tier architecture (see Two Tier Software Architectures). The third tier (middle tier server) is between the user interface (client) and the data management (server) components. This middle tier provides process management where business logic and rules are executed and can accommodate hundreds of users (as compared to only 100 users with the two tier architecture) by providing functions such as queuing, application execution, and database staging. The three tier architecture is used when an effective distributed client/server design is needed that provides (when compared to the two tier) increased performance, flexibility, maintainability, reusability, and scalability, while hiding the complexity of distributed processing from the user

Technical Details

A three tier distributed client/server architecture (as shown in Figure 28) includes a user system interface top tier where user services (such as session, text input, dialog, and display management) reside.

The third tier provides database management functionality and is dedicated to data and file services that can be optimized without using any proprietary database management system languages. The data management component ensures that the data is consistent throughout the distributed environment through the use of features such as data locking, consistency, and replication. It should be noted that connectivity between tiers can be dynamically changed depending upon the user's request for data and services.

The middle tier provides process management services (such as process development, process enactment, process monitoring, and process resourcing) that are shared by multiple applications.

The middle tier server (also referred to as the application server) improves performance, flexibility, maintainability, reusability, and scalability by centralizing process logic. Centralized process logic makes administration and change management easier by localizing system functionality so that changes must only be written once and placed on the middle tier server to be available throughout the systems.

Usage Considerations

The middle tier manages distributed database integrity by the two phase commit process. It provides access to resources based on names instead of locations, and thereby improves scalability and flexibility as system components are added or move.

Sometimes, the middle tier is divided in two or more unit with different functions, in these cases the architecture is often referred as multi layer. This is the case, for example, of some Internet applications. These applications typically have light clients written in HTML and application servers written in C++ or Java, the gap between these two layers is too big to link them together. Instead, there is an intermediate layer (web server) implemented in a scripting language. This layer receives requests from the Internet clients and generates html using the services provided by the business layer. This additional layer provides further isolation between the application layout and the application logic.

It should be noted that recently, mainframes have been combined as servers in distributed architectures to provide massive storage and improve security.

Definition

The addition of a middle tier between the user system interface client environment and the database management server environment.

Discussion

There are a variety of ways of implementing this middle tier, such as transaction processing monitors, message servers, or application servers. The middle tier can perform queuing, application execution, and database staging.

Example

If the middle tier provides queuing, the client can deliver its request to the middle layer and disengage because the middle tier will access the data and return the answer to the client. In addition the middle layer adds scheduling and prioritization for work in progress.

Advantages

The three tier client/server architecture has been shown to improve performance for groups with a large number of users (in the thousands) and improves flexibility when compared to

the two tier approach. modules onto different computers in some three tier architectures.

Disadvantages

The three tier architectures development environment is reportedly more difficult to use than the visually-oriented development of two tier systems.

Ecommerce Systems - Application Servers

Definition

Application servers share business logic, computations, and a data retrieval engine on the server. There is now processing required on the client.

Advantages

With less software on the client there is less security to worry about, applications are more scalable, and support and installation costs are less on a single server than maintaining each on a desktop client. The application server design should be used when security, scalability, and cost are major considerations.

Multi-Tier Application Design

An age-old software engineering principle explains that by logically partitioning a piece of software into independent layers of responsibility, one can produce programs that have fewer defects, are better at documenting themselves, can be developed concurrently by may programmers with specific skill sets, and are more maintainable than the alternative of a monolithic hunk of code. Examples of these layers, or tiers, are common: the kernel (privileged CPU mode) and other applications (user mode); the seven ISO/OSI network model layers (or the redivided four used by the Internet); and even the database "onion" containing core, management system, query engine, procedural language engine, and connection interface.

Note that these tiers are entirely logical in nature. Their physical implementation may vary considerably: everything compiled into one EXE, a single tier spread across multiple statically- or dynamically-linked libraries, tiers divided amongst separate networked computers, and so forth.

Each such tier is one further level of abstraction from the raw data of the application (the "lowest" tier). The "highest" tier is therefore the most "abstract" and also the best candidate for communicating directly with the end user.

Individual tiers are designed to be a self-contained as possible, exposing only a well-defined interface (e.g. function names, usually called an Application Programming Interface, or API) that another tier may use. In this respect, tiers are analogous to the classes of Object-Oriented Programming. In theory, a new tier with a compatible interface could easily be substituted for another, but in practice this can't always be done without a bit of fuss.

Tiers only communicate in this downward direction (that is, a lower-level tier does not call a function in a higher-level tier), and a tier may only call into the tier directly beneath it (that is, tiers are not bypassed). One might also say that a higher-level tier is a "consumer" of the services afforded by the lower-level tier, the "provider".

Each tier does introduce a small performance penalty (typically, stack frame overhead for the function calls) but this is usually not significant, and is outweighed by the design advantages of a multi-tier design.

If performance does become an issue, some of the rules above may be broken, with a consequent loss of design consistency.

Three Tier VS Muti tier

These three tiers have proved more successful than other multi-tier schemes for a couple of reasons:

Matching Skill Sets: It turns out that the skills of the various people that might work on building an application tend to correspond neatly with the three tiers. The Presentation tier requires people with some level of either user-interface/ergonomics experience or artistic sensibilities, or the combination of the two (often found in web designers). The Business Logic tier calls upon people very familiar with procedural language techniques and a considerable investment in a particular set of procedural programming languages (e.g. C/C++, Java, VB, PHP). Finally, the Data tier requires intimate knowledge of relational database theory and the SQL DML language. It's next to impossible to find a single person with talent and experience in all three areas, and reasonably difficult to find people with skills in two. Fortunately, the separation of the three layers means that people with just one of these skill sets can work on the project side by side with those possessing the other skills, lessening the "too many cooks" effect.

Multi-Server Scalability: Just as people become "optimized" for a certain task through learning and experience, computer systems can also be optimized for each of the three tiers. While it's possible to run all three logical tiers on a single server (as is done on the course server), as a system grows to accommodate greater and greater numbers of users (a problem typical of web applications), a single server will no longer suffice. It turns out that the processing needs of the three tiers are distinct, and so a physical arrangement often consists of many Presentation tier servers (a.k.a. web servers), a few Business Logic tier servers (a.k.a. application servers), and usually just one, or at most a handful, of Data tier servers (a.k.a. RDBMS servers). RDBMS servers consume every resource a hardware platform can provide: CPU, memory, disk, and gobs of each. RDBMS's also often have innumerable tuning parameters. An application server is typically only CPU and memory-bound, requiring very little disk space. Finally, a web server (just a specialized type of file server) is mostly reliant on memory and disk.

Review Questions

1. Explain what is single tier database architecture?
2. Explain the Two-Tier architecture?
3. Explain the n tier architecture?

References

http://otn.oracle.com/pub/articles/tech_dba.html

http://www.openlinksw.com/licenses/appst.htmDate, C. J. An Introduction to Database Systems. Volume I. Addison/Wesley, 1990, 455–473.

LESSON 21
PERFORMANCE CRITERIA

Hi! Database Performance

Database performance focuses on tuning and optimizing the design, parameters, and physical construction of database objects, specifically tables and indexes, and the files in which their data is stored. The actual composition and structure of database objects must be monitored continually and changed accordingly if the database becomes inefficient. **No amount of SQL tweaking or system tuning can optimize the performance of queries run against a poorly designed or disorganized database.**

Techniques for Optimizing Databases

The DBA must be cognizant of the features of the DBMS in order to apply the proper techniques for optimizing the performance of database structures. Most of the major DBMSs support the following techniques although perhaps by different names. Each of the following techniques can be used to tune database performance and will be discussed in subsequent sections.

- Partitioning — breaking a single database table into sections stored in multiple files.
- Raw partitions versus file systems — choosing whether to store database data in an OS-controlled file or not.
- Indexing — choosing the proper indexes and options to enable efficient queries.
- Denormalization — varying from the logical design to achieve better query performance.
- Clustering — enforcing the physical sequence of data on disk.
- Interleaving data — combining data from multiple tables into a single, sequenced file.
- Free space — leaving room for data growth.
- Compression — algorithmically reducing storage requirements.
- File placement and allocation — putting the right files in the right place.
- Page size — using the proper page size for efficient data storage and I/O.
- Reorganization — removing inefficiencies from the database by realigning and restructuring database objects.

Partitioning

A database table is a logical manifestation of a set of data that physically resides on computerized storage. One of the decisions that the DBA must make for every table is how to store that data. Each DBMS provides different mechanisms that accomplish the same thing -mapping physical files to database tables. The DBA must decide from among the following mapping options for each table:

- *Single table to a single file*. This is, by far, the most common choice. The data in the file is formatted such that the DBMS understands the table structure and every row inserted into that table is stored in the same file. However, this setup is not necessarily the most efficient.
- *Single table to multiple files*. This option is used most often for very large tables or tables requiring data to be physically separated at the storage level. Mapping to multiple files is accomplished by using partitioned tablespaces or by implementing segmented disk devices.
- *Multiple tables to a single file.* This type of mapping is used for small tables such as lookup tables and code tables, and can be more efficient from a disk utilization perspective.

Partitioning helps to accomplish parallelism. Parallelism is the process of using multiple tasks to access the database in parallel. A parallel request can be invoked to use multiple, simultaneous read engines for a single SQL statement. Parallelism is desirable because it can substantially reduce the elapsed time for database queries.

Multiple types of parallelism are based on the resources that can be invoked in parallel. For example, a single query can be broken down into multiple requests each utilizing a different CPU engine in parallel. In addition, parallelism can be improved by spreading the work across multiple database instances. Each DBMS offers different levels of support for parallel database queries. To optimize database performance, the DBA should be cognizant of the support offered in each DBMS being managed and exploit the parallel query capabilities.

Raw Partition vs. File System

For a UNIX-based DBMS environment, the DBA must choose between a raw partition and using the UNIX file system to store the data in the database. A *raw partition* is the preferred type of physical device for database storage because writes are cached by the operating system when a file system is utilized. When writes are buffered by the operating system, the DBMS does not know whether the data has been physically copied to disk or not. When the DBMS cache manager attempts to writes the data to disk, the operating system may delay the write until later because the data may still be in the file system cache. If a failure occurs, data in a database using the file system for storage may not be 100% recoverable. This is to be avoided.

If a raw partition is used instead, the data is written directly from the database cache to disk with no intermediate file system or operating system caching, as shown in Figure 11-1. When the DBMS cache manager writes the data to disk, it will physically be written to disk with no intervention. Additionally, when using a raw partition, the DBMS will ensure that enough space is available and write the allocation pages. When using a file system, the operating system will not preallocate space for database usage.

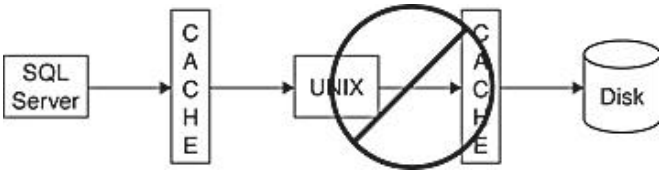


Figure 11-1 *Using raw partitions to avoid file system caching*

From a performance perspective, there is no advantage to having a secondary layer of caching at the file system or operating system level; the DBMS cache is sufficient. Actually, the additional work required to cache the data a second time consumes resources, thereby negatively impacting the overall performance of database operations.

Do not supplement the DBMS cache with any type of additional cache.

Indexing

Creating the correct indexes on tables in the database is perhaps the single greatest performance tuning technique that a DBA can perform. Indexes are used to enhance performance. **Indexes are particularly useful for**

- Locating rows by value(s) in column(s)
- Making joins more efficient (when the index is defined on the join columns)
- Correlating data across tables
- Aggregating data
- Sorting data to satisfy a query

Without indexes, all access to data in the database would have to be performed by scanning all available rows. Scans are very inefficient for very large tables.

Designing and creating indexes for database tables actually crosses the line between database performance tuning and application performance tuning. Indexes are database objects created by the DBA with database DDL. However, an index is built to make SQL statements in application programs run faster. Indexing as a tuning effort is applied to the database to make applications more efficient when the data access patterns of the application vary from what was anticipated when the database was designed.

Before tuning the database by creating new indexes, be sure to understand the impact of adding an index. The DBA should have an understanding of the access patterns of the table on which the index will be built. Useful information includes the percentage of queries that access rather than update the table, the performance thresholds set within any service level agreements for queries on the table, and the impact of adding a new index to running database utilities such as loads, reorganizations, and recovery.

One of the big unanswered questions of database design is: "How many indexes should be created for a single table?" There is no set answer to this question. The DBA will need to use his expertise to determine the proper number of indexes for each table such that database queries are optimized and the performance of database inserts, updates, and deletes does not degrade. Determining the proper number of indexes for each table requires in-depth analysis of the database and the applications that access the database.

The general goal of index analysis is to use less I/O to the database to satisfy the queries made against the table. Of course, an index can help some queries and hinder others. Therefore, the DBA must assess the impact of adding an index to all applications and not just tune single queries in a vacuum. This can be an arduous but rewarding task.

An index affects performance positively when fewer I/Os are used to return results to a query. Conversely, an index negatively impacts performance when data is updated and the indexes have to be changed as well. An effective indexing strategy seeks to provide the greatest reduction in I/O with an acceptable level of effort to keep the indexes updated.

Some applications have troublesome queries that require significant tuning to achieve satisfactory performance. Creating an index to support a single query is acceptable if that query is important enough in terms of ROI to the business (or if it is run by your boss or the CEO). If the query is run infrequently, consider creating the index before the process begins and dropping the index when the process is complete.

Whenever you create new indexes, **be sure to thoroughly test the performance of the queries it supports**. Additionally, be sure to test database modification statements to gauge the additional overhead of updating the new indexes. Review the CPU time, elapsed time, and I/O requirements to assure that the indexes help. Keep in mind that tuning is an iterative process, and it may take time and several index tweaks to determine the impact of a change. There are no hard and fast rules for index creation. Experiment with different index combinations and measure the results.

When to Avoid Indexing

There are a few scenarios where indexing may not be a good idea. When tables are very small, say less than ten pages, consider avoiding indexes. Indexed access to a small table can be less efficient than simply scanning all of the rows because reading the index adds I/O requests.

Index I/O notwithstanding, even a small table can sometimes benefit from being indexed - for example, to enforce uniqueness or if most data access retrieves a single row using the primary key.

You may want to avoid indexing variable-length columns if the DBMS in question expands the variable column to the maximum length within the index. Such expansion can cause indexes to consume an inordinate amount of disk space and might be inefficient. However, if variable-length columns are used in SQL WHERE clauses, the cost of disk storage must be compared to the cost of scanning. Buying some extra disk storage is usually cheaper than wasting CPU resources to scan rows. Furthermore, the SQL query might contain alternate predicates that could be indexed instead of the variable-length columns.

Additionally, **avoid indexing any table that is always accessed using a scan,** that is, the SQL issued against the table never supplies a WHERE clause.

Index Overloading

Query performance can be enhanced in certain situations by overloading an index with additional columns. Indexes are typically based on the WHERE clauses of SQL SELECT statements. For example, consider the following SQL statement.

```
select    emp_no,    last_name,    salary
from                                employee
where     salary     >             15000.00
;
```

Creating an index on the salary column can enhance the performance of this query. However, the DBA can further enhance the performance of the query by overloading the index with the emp_no and last_name columns, as well. With an **overloaded index, the DBMS can satisfy the query by using** the index. The DBMS need not incur the additional I/O of accessing the table data, since every piece of data that is required by the query exists in the overloaded index.

DBAs should consider overloading indexes to encourage index-only access when multiple queries can benefit from the index or when individual queries are very important.

Denormalization

Another way to optimize the performance of database access is to denormalize the tables. In brief, *denormalization*, the opposite of *normalization*, is the process of putting one fact in many places. This speeds data retrieval at the expense of data modification. Denormalizing tables can be a good decision when a completely normalized design does not perform optimally.

The only reason to ever denormalize a relational database design is to enhance performance. As discussed elsewhere in "Database administration," you should consider the following options:

- *Prejoined tables* — when the cost of joining is prohibitive.
- *Report table* — when specialized critical reports are too costly to generate.
- *Mirror table* — when tables are required concurrently by two types of environments.
- *Split tables* — when distinct groups use different parts of a table.
- *Combined tables* — to consolidate one-to-one or one-to-many relationships into a single table.
- *Speed table* — to support hierarchies like bill-of-materials or reporting structures.
- *Physical denormalization* — to take advantage of specific DBMS characteristics.

You might also consider

- Storing *redundant data* in tables to reduce the number of table joins required.
- Storing *repeating groups* in a row to reduce I/O and possibly disk space.
- Storing *derivable data* to eliminate calculations and costly algorithms.

Clustering

A clustered table will store its rows physically on disk in order by a specified column or columns. Clustering usually is enforced by the DBMS with a clustering index. The clustering index forces table rows to be stored in ascending order by the indexed columns. The left-to-right order of the columns as defined in the index, defines the collating sequence for the clustered index. There can be only one clustering sequence per table (because physically the data can be stored in only one sequence).

Figure 11-2 demonstrates the difference between clustered and unclustered data and indexes; the clustered index is on top, the unclustered index is on the bottom. As you can see, the entries on the leaf pages of the top index are in sequential order - in oher words, they are clustered. Clustering enhances the performance of queries that access data sequentially because fewer I/Os need to be issued to retrieve the same data.

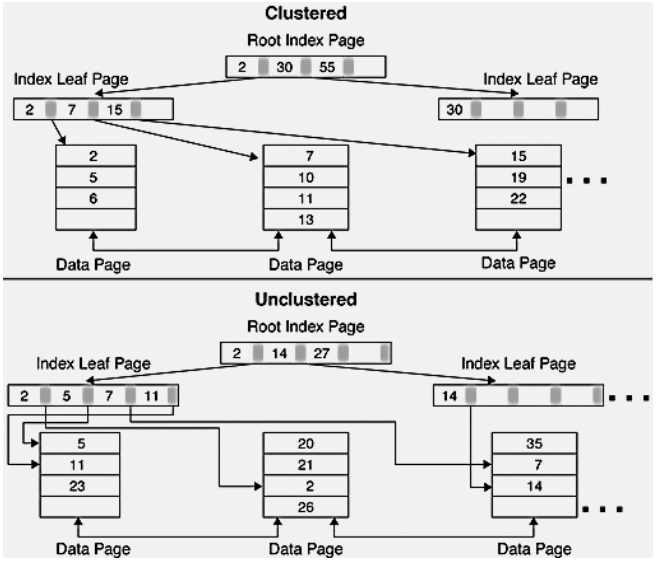


Figure 11-2 ***Clustered and unclustered indexes***

Depending on the DBMS, the data may not always be physically maintained in exact clustering sequence. When a clustering sequence has been defined for a table, the DBMS will act in one of two ways to enforce clustering:

1. When new rows are inserted, the DBMS will physically maneuver data rows and pages to fit the new rows into the defined clustering sequence; or
2. When new rows are inserted, the DBMS will try to place the data into the defined clustering sequence, but if space is not available on the required page the data may be placed elsewhere.

The DBA must learn how the DBMS maintains clustering. If the DBMS operates as in the second scenario, data may become unclustered over time and require reorganization. A detailed discussion of database reorganization appears later in this chapter. For now, though, back to our discussion of clustering.

Clustering tables that are accessed sequentially is good practice. In other words, clustered indexes are good for supporting range access, whereas unclustered indexes are better for supporting random access. Be sure to choose the clustering

columns wisely. Use clustered indexes for the following situations:

- Join columns, to optimize SQL joins where multiple rows match for one or both tables participating in the join
- Foreign key columns because they are frequently involved in joins and the DBMS accesses foreign key values during declarative referential integrity checking
- Predicates in a WHERE clause
- Range columns
- Columns that do not change often (reduces physically reclustering)
- Columns that are frequently grouped or sorted in SQL statements

In general, the clustering sequence that aids the performance of the most commonly accessed predicates should be used to for clustering. When a table has multiple candidates for clustering, weigh the cost of sorting against the performance gained by clustering for each candidate key. As a rule of thumb, though, if the DBMS supports clustering, it is usually a good practice to define a clustering index for each table that is created (unless the table is very small).

Clustering is generally not recommended for primary key columns because the primary key is, by definition, unique. However, if ranges of rows frequently are selected and ordered by primary key value, a clustering index may be beneficial.

Page Splitting

When the DBMS has to accommodate inserts, and no space exists, it must create a new page within the database to store the new data. **The process of creating new pages to store inserted data is called page splitting. A DBMS can perform two types of *page splitting***: normal page splits and monotonic page splits. Some DBMSs support both types of page splitting, while others support only one type. The DBA needs to know how the DBMS implements page splitting in order to optimize the database.

Figure 11-3 depicts a normal page split. To accomplish this, the DBMS performs the following tasks in sequence:

1. Creates a new empty page in between the full page and the next
   page
2. Takes half of the entries from the full page and moves them to the empty page
3. Adjusts any internal pointers to both pages and inserts the row accordingly

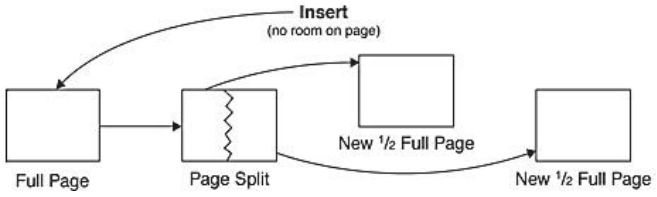


Figure 11-3 *Normal page splitting*

A monotonic page split is a much simpler process, requiring only two steps. The DBMS

- Creates a new page in between the full page and the next page
- Inserts the new values into the fresh page

Monotonic page splits are useful when rows are being inserted in strictly ascending sequence. Typically, a DBMS that supports monotonic page splits will invoke it when a new row is added to the end of a page and the last addition was also to the end of the page.

When ascending rows are inserted and normal page splitting is used, a lot of space can be wasted because the DBMS will be creating half-full pages that never fill up. If the wrong type of page split is performed during database processing, wasted space will ensue, requiring the database object to be reorganized for performance.

Interleaving Data

When data from two tables is frequently joined, it can make sense to physically interleave the data into the same physical storage structure. **Interleaving can be viewed as a specialized form of clustering**

Free Space

Free space, sometimes called *fill factor*, can be used to leave a portion of a tablespace or index empty and available to store newly added data. The specification of free space in a tablespace or index can reduce the frequency of reorganization, reduce contention, and increase the efficiency of insertion. Each DBMS provides a method of specifying free space for a database object in the CREATE and ALTER statements. A typical parameter is PCTFREE, where the DBA specifies the percentage of each data page that should remain available for future inserts. Another possible parameter is FREEPAGE, where the DBA indicates the specified number of pages after which a completely empty page is available.

Ensuring a proper amount of free space for each database object provides the following benefits:

- Inserts are faster when free space is available.
- As new rows are inserted, they can be properly clustered.
- Variable-length rows and altered rows have room to expand, potentially reducing the number of relocated rows.
- Fewer rows on a page results in better concurrency because less data is unavailable to other users when a page is locked.

However, free space also has several disadvantages.

- Disk storage requirements are greater.
- Scans take longer.
- Fewer rows on a page can require more I/O operations to access the requested information.
- Because the number of rows per page decreases, the efficiency of data caching can decrease because fewer rows are retrieved per I/O.

The DBA should monitor free space and ensure that the appropriate amount is defined for each database object. The correct amount of free space must be based on

- Frequency of inserts and modifications
- Amount of sequential versus random access

- Impact of accessing unclustered data
- Type of processing
- Likelihood of row chaining, row migration, and page splits

Don't define a static table with free space - it will not need room in which to expand.

The remaining topics will be discussed in the next lecture.

Compression

Compression can be used to shrink the size of a database. By compressing data, the database requires less disk storage. Some DBMSs provide internal DDL options to compress database files; third-party software is available for those that do not provide such features.

When compression is specified, data is algorithmically compressed upon insertion into the database and decompressed when it is read. Reading and writing compressed data consumes more CPU resources than reading and writing uncompressed data: The DBMS must execute code to compress and decompress the data as users insert, update, and read the data.

So why compress data? Consider an uncompressed table with a row size of 800 bytes. Five of this table's rows would fit in a 4K data page (or block). Now what happens if the data is compressed? Assume that the compression routine achieves 30% compression on average (a very conservative estimate). In that case, the 800-byte row will consume only 560 bytes (800 x 0.30 = 560). After compressing the data, seven rows will fit on a 4K page. Because I/O occurs at the page level, a single I/O will retrieve more data, which will optimize the performance of sequential data scans and increase the likelihood of data residing in the cache because more rows fit on a physical page.

Of course, **compression always requires a trade-off** that the DBA must analyze. On the positive side, we have disk savings and the potential for reducing I/O cost. On the negative side, we have the additional CPU cost required to compress and decompress the data.

However, compression is not an option for every database index or table. For smaller amounts of data, it is possible that a compressed file will be larger than an uncompressed file. This is so because some DBMSs and compression algorithms require an internal dictionary to manage the compression. The dictionary contains statistics about the composition of the data that is being compressed. For a trivial amount of data, the size of the dictionary may be greater than the amount of storage saved by compression.

File Placement and Allocation

The location of the files containing the data for the database can have a significant impact on performance. A database is very I/O intensive, and **the DBA must make every effort to minimize the cost of physical disk reading and writing.**

This discipline entails

- Understanding the access patterns associated with each piece of data in the system
- Placing the data on physical disk devices in such a way as to optimize performance

The first consideration for file placement on disk is to separate the indexes from the data, if possible. Database queries are frequently required to access data from both the table and an index on that table. If both of these files reside on the same disk device, performance degradation is likely. To retrieve data from disk, an arm moves over the surface of the disk to read physical blocks of data on the disk. If a single operation is accessing data from files on the same disk device, latency will occur; reads from one file will have to wait until reads from the other file are processed. Of course, if the DBMS combines the index with the data in the same file, this technique cannot be used.

Another rule for file placement is to analyze the access patterns of your applications and separate the files for tables that are frequently accessed together. The DBA should do this for the same reason he should separate index files from table files.

A final consideration for placing files on separate disk devices occurs when a single table is stored in multiple files (partitioning). It is wise in this case to place each file on a separate disk device to encourage and optimize parallel database operations. If the DBMS can break apart a query to run it in parallel, placing multiple files for partitioned tables on separate disk devices will minimize disk latency.

Database Log Placement

Placing the transaction log on a separate disk device from the actual data allows the DBA to back up the transaction log independently from the database. It also minimizes dual writes to the same disk. Writing data to two files on the same disk drive at the same time will degrade performance even more than reading data from two files on the same disk drive at the same time. Remember, too, every database modification (write) is recorded on the database transaction log.

Distributed Data Placement

The goal of data placement is to optimize access by reducing contention on physical devices. Within a client/server environment, this goal can be expanded to encompass the optimization of application performance by reducing network transmission costs.

Data should reside at the database server where it is most likely, or most often, to be accessed. For example, Chicago data should reside at the Chicago database server, Los Angeles-specific data should reside at the Los Angeles database server, and so on. If the decision is not so clear-cut (e.g., San Francisco data, if there is no database server in San Francisco), place the data on the database server that is geographically closest to where it will be most frequently accessed (in the case of San Francisco, L.A., not Chicago).

Be sure to take fragmentation, replication, and snapshot tables into account when deciding upon the placement of data in your distributed network.

Disk Allocation

The DBMS may require disk devices to be allocated for database usage. If this is the case, the DBMS will provide commands to initialize physical disk devices. The disk initialization command will associate a logical name for a physical disk partition or OS

file. After the disk has been initialized, it is stored in the system catalog and can be used for storing table data.

Before initializing a disk, verify that sufficient space is available on the physical disk device. Likewise, make sure that the device is not already initialized.

Use meaningful device names to facilitate more efficient usage and management of disk devices. For example, it is difficult to misinterpret the usage of a device named DUMP_DEV1 or TEST_DEV7. However, names such as XYZ or A193 are not particularly useful. Additionally, maintain documentation on initialized devices by saving script files containing the actual initialization commands and diagrams indicating the space allocated by device.

Page Size (Block Size)

Most DBMSs provide the ability to specify a page, or block, size. The *page size* is used to store table rows (or more accurately, records that contain the row contents plus any overhead) on disk. For example, consider a table requiring rows that are 125 bytes in length with 6 additional bytes of overhead. This makes each record 131 bytes long. To store 25 records on a page, the page size would have to be at least 3275 bytes. However, each DBMS requires some amount of page overhead as well, so the practical size will be larger. If page overhead is 20 bytes, then the page size would be 3295 - that is, 3275 + 20 bytes of overhead.

This discussion, however, is simplistic. In general practice, most tablespaces will require some amount of free space to accommodate new data. Therefore, some percentage of free space will need to be factored into the above equation.

To complicate matters, many DBMSs limit the page sizes that can be chosen. For example, DB2 for OS/390 limits page size to 4K, 8K, 16K, or 32K. In this case, the DBA will need to calculate the best page size based on row size, the number of rows per page, and free space requirements.

Consider this question: "In DB2 for OS/390, what page size should be chosen if 0% free space is required and the record size is 2500 bytes?"

The simplistic answer is 4K, but it might not be the best answer. A 4K page would hold one 2500-byte record per page, but an 8K page would hold three 2500-byte records. The 8K page would provide for more efficient I/O, because reading 8K of data would return three rows, whereas reading 8K of data using two 4K pages would return only two rows.

Choosing the proper page size is an important DBA task for optimizing database I/O performance.

Database Reorganization

Relational technology and SQL make data modification easy. Just issue an INSERT, UPDATE, or DELETE statement with the appropriate WHERE clause, and the DBMS takes care of the actual data navigation and modification. In order to provide this level of abstraction, the DBMS handles the physical placement and movement of data on disk. Theoretically, this makes everyone happy. The programmer's interface is simplified, and the RDBMS takes care of the hard part -manipulating the actual placement of data. However, things are not quite that simple. The manner in which the DBMS physically manages data can cause subsequent performance problems.

Every DBA has encountered the situation where a query or application that used to perform well slows down after it has been in production for a while. **These slowdowns have many potential causes -** perhaps the number of transactions issued has increased, or the volume of data has expanded. However, the performance problem might be due to database disorganization. Database disorganization occurs when a database's logical and physical storage allocations contain many scattered areas of storage that are too small, not physically contiguous, or too disorganized to be used productively. Let's review the primary culprits.

- The first possibility is *unclustered data.* If the DBMS does not strictly enforce clustering, a clustered table or index can become unclustered as data is added and changed. If the data becomes significantly unclustered, the DBMS cannot rely on the clustering sequence. Because the data is no longer clustered, queries that were optimized to access data cannot take advantage of the clustering sequence. In this case, the performance of queries run against the unclustered table will suffer.
- *Fragmentation* is a condition in which there are many scattered areas of storage in a database that are too small to be used productively. It results in wasted space, which can hinder performance because additional I/Os are required to retrieve the same data.
- *Row chaining* or *row migration* occurs when updated data does not fit in the space it currently occupies, and the DBMS must find space for the row. With row chaining, the DBMS moves a part of the new, larger row to a location within the tablespace where free space exists. With row migrations, the full row is placed elsewhere in the tablespace. In each case, a pointer is used to locate either the rest of the row or the full row. Both row chaining and row migration will result in the issuance of multiple I/Os to read a single row. Performance will suffer because multiple I/Os are more expensive than a single I/O.
- *Page splits* can cause disorganized databases, too. If the DBMS performs monotonic page splits when it should perform normal page splits, or vice versa, space may be wasted. When space is wasted, fewer rows exist on each page, causing the DBMS to issue more I/O requests to retrieve data. Therefore, once again, performance suffers.
- *File extents* can negatively impact performance. An *extent* is an additional file that is tied to the original file and can be used only in conjunction with the original file. When the file used by a tablespace runs out of space, an extent is added for the file to expand. However, file extents are not stored contiguously with the original file. As additional extents are added, data requests will need to track the data from extent to extent, and the additional code this requires is unneeded overhead. Resetting the database space requirements and reorganizing can clean up file extents.

Let's take a look at a disorganized tablespace by comparing Figures 11-4 and 11-5. Assume that a tablespace consists of three tables across multiple blocks, such as the tablespace and tables depicted in Figure 11-4. Each box represents a data page.

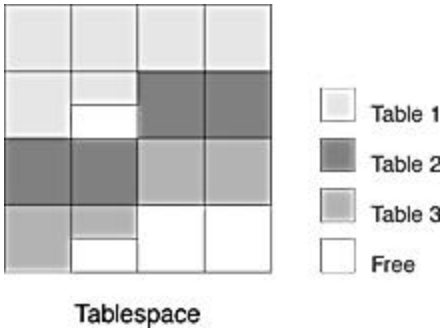


Figure 11-4 Organized tablespace

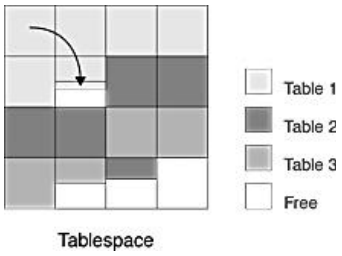


Figure 11-5 disorganized tablespace

Now, let's make a couple of changes to the data in these tables. First, we'll add six rows to the second table. However, no free space exists into which these new rows can be stored. How can the rows be added? The DBMS requires an additional extent to be taken into which the new rows can be placed. This result in fragmentation: The new rows have been placed in a noncontiguous space. For the second change, let's update a row in the first table to change a variable-length column; for example, let's change the value in a LASTNAME column from WATSON to BEAUCHAMP. Issuing this update results in an expanded row size because the value for LASTNAME is longer in the new row: "BEAUCHAMP" contains 9 characters whereas "WATSON" only consists of 6. This action results in row chaining. The resultant tablespace shown in Figure 11-5 depicts both the fragmentation and the row chaining.

Depending on the DBMS, there may be additional causes of disorganization. For example, if multiple tables are defined within a tablespace, and one of the tables is dropped, the tablespace may need to be reorganized to reclaim the space.

To correct disorganized database structures, the DBA can run a database or tablespace reorganization utility, or REORG, to force the DBMS to restructure the database object, thus removing problems such as unclustered data, fragmentation, and row chaining. The primary benefit of reorganization is the resulting speed and efficiency of database functions because the data is organized in a more optimal fashion on disk. **In short, reorganization maximizes availability and reliability for databases.**

Tablespaces and indexes both can be reorganized. How the DBA runs a REORG utility depends on the DBMS. Some DBMS products ship with a built-in reorganization utility. Others require the customer to purchase the utility. Still others claim that the customer will not need the utility at all when using their DBMS. I have found the last claim to be untrue. Every DBMS incurs some degree of disorganization as data is added and modified.

Of course, DBAs can manually reorganize a database by completely rebuilding it. However, accomplishing such a reorganization requires a complex series of steps. Figure 11-6 depicts the steps entailed by a manual reorganization.

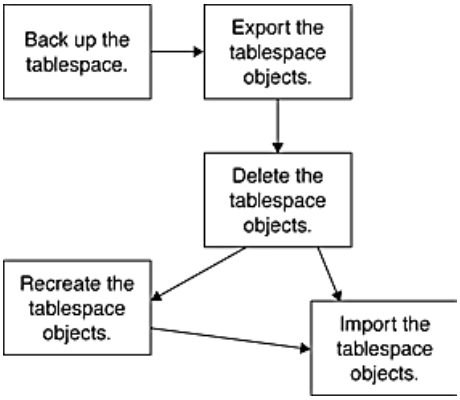


Figure 11-6 Typical steps for a manual reorganization

If a utility is available for reorganizing, either from the DBMS vendor or a third=party vendor, the process is greatly simplified. Sometimes the utility is as simple as issuing a simple command such as

```
Reorg Tablespace Tsname
```

A traditional reorganization requires the database to be down. The high cost of downtime creates pressures both to perform and to delay preventive maintenance - a no-win situation familiar to most DBAs. Some REORG utilities are available that perform the reorganization while the database is online. Such a reorganization is accomplished by making a copy of the data. The online REORG utility reorganizes the copy while the original data remains online. When the copied data has been reorganized, an online REORG uses the database log to "catch up" by applying to the copy any data changes that occurred during the process. When the copy has caught up to the original, the online REORG switches the production tablespace from the original to the copy. Performing an online reorganization requires additional disk storage and a slow transaction window. If a large number of transactions occur during the online reorganization, REORG may have a hard time catching up.

Determining When to Reorganize

System catalog statistics can help to determine when to reorganize a database object. Each DBMS provides a method of reading through the contents of the database and recording statistical information about each database object. Depending on the DBMS, this statistical information is stored either in the system catalog or in special pages within the database object itself.

One statistic that can help a DBA determine when to reorganize is *cluster ratio*. **Cluster ratio is the percentage of rows in a table that are actually stored in a clustering sequence.** The closer the cluster ratio is to 100%, the more closely the actual ordering of the rows on the data pages matches the clustering sequence. A low cluster ratio indicates bad clustering, and a reorganization may be required. A low cluster ratio, however, may not be a performance hindrance if the majority of queries access data randomly instead of sequentially.

Tracking down the other causes of disorganization can sometimes be difficult. Some DBMSs gather statistics on fragmentation, row chaining, row migration, space dedicated to dropped objects, and page splits; others do not. Oracle provides a plethora of statistics in dynamic performance tables that can be queried. Refer to the sidebar "Oracle Dynamic Performance Tables" for more details.

Oracle Dynamic Performance Tables

Oracle stores vital performance statistics about the database system in a series of dynamic performance tables. These tables are sometimes referred to as the "V$ tables" because the table names are prefixed with the characters V$.

The V$ tables are used by the built-in Oracle performance monitoring facilities and can be queried by the DBA for insight into the well-being and performance of an Oracle instance. Examples of some of the statistics that can be found in the V$ tables include

- Free space available
- Chained rows
- Rollback segment contention
- Memory usage
- Disk activity

Of course, there is quite a lot of additional performance information to be found in these tables. Oracle DBAs should investigate the V$ tables and query these tables regularly to analyze the performance of the Oracle system, its databases, and applications.

Tablespaces are not the only database objects that can be reorganized. Indexes, too, can benefit from reorganization. As table data is added and modified, the index too must be changed. Such changes can cause the index to become disorganized.

A vital index statistic to monitor is the number of levels. Recall from Chapter 4 that most relational indexes are b-tree structures. As data is added to the index, the number of levels of the b-tree will grow. When more levels exist in the b-tree, more I/O requests are required to move from the top of the index structure to the actual data that must be accessed. Reorganizing an index can cause the index to be better structured and require fewer levels.

Another index statistic to analyze to determine if reorganization is required is the distance between the index leaf pages, or *leaf distance*. Leaf distance is an estimate of the average number of pages between successive leaf pages in the index. Gaps between leaf pages can develop as data is deleted from an index or as a result of page splitting. Of course, the best value for leaf distance is zero, but achieving a leaf distance of zero in practice is not realistic. In general, the lower this value, the better. Review the value over time to determine a high-water mark for leaf distance that will indicate when indexes should be reorganized.

Review Question

1. Define Techniques for Optimizing Databases

References

http://www.microsoft-accesssolutions.co.uk
http://www.cs.sfu.ca/CC/354

Notes:

LESSON 22
STORAGE AND ACCESS METHOD

Hi! In this chapter I am going to discuss with you about the Storage and access method.

The Physical Store

Storage Medium	Transfer Rate	Capacity	Seek Time
Main Memory	800 MB/s	100 MB	Instant
Hard Drive	10 MB/s	10 GB	10 ms
CD-ROM Drive	5 MB/s	0.6 GB	100 ms
Floppy Drive	2 MB/s	1.44 MB	300 ms
Tape Drive	1 MB/s	20 GB	30 s

Why not all Main Memory?

The performance of main memory is the greatest of all storage methods, but it is also the most expensive per MB.

- All the other types of storage are 'persistent'. A persistent store keeps the data stored on it even when the power is switched off.
- Only main memory can be directly accessed by the programmer. Data held using other methods must be loaded into main memory before being accessed, and must be transferred back to storage from main memory in order to save the changes.
- We tend to refer to storage methods which are not main memory as 'secondary storage'.

Secondary Storage - Blocks

All storage devices have a block size. Block size is the minimum amount which can be read or written to on a storage device. Main memory can have a block size of 1-8 bytes, depending on the processor being used. Secondary storage blocks are usually much bigger.

- Hard Drive disk blocks are usually 4 KBytes in size.
- For efficiency, multiple contiguous blocks can be be requested.
- On average, to access a block you first have to request it, wait the seek time, and then wait the transfer time of the blocks requested.
- Remember, you cannot read or write data smaller than a single block.

Hard Drives

The most common secondary storage medium for DBMS is the hard drive.

- Data on a hard-drive is often arranged into files by the Operating System.
- The DBMS holds the database within one or more files.
- The data is arranged within a file in blocks, and the position of a block within a file is controlled by the DBMS.
- Files are stored on the disk in blocks, but the placement of a file block on the disk is controlled by the O/S (although the DBMS may be allowed to 'hint' to the O/S concerning disk block placement strategies).
- File blocks and disk blocks are not necessarily equal in size.

DBMS Data Items

Data from the DBMS is split into records.

- A record is a logical collection of data items
- A file is a collection of records.
- One or more records may map onto a single or multiple file blocks.
- A single record may map onto multiple file blocks.

Comparing Terminology...

Relational	SQL	Physical Storage
Relation	Table	File
Tuple	Row	Record
Attribute	Column	Data Item/Field
Domain	Type	Data Type

File Organisations

- Serial (or unordered, or heap) - records are written to secondary storage in the order in which they are created.
- Sequential (or sorted, or ordered) - records are written to secondary storage in the sorted order of a key (one or more data items) from each record.
- Hash - A 'hash' function is applied to each record key, which returns a number used to indicate the position of the record in the file. The hash function must be used for both reading and writing.
- Indexed - the location in secondary storage of some (partial index) or all (full index) records is noted in an index.

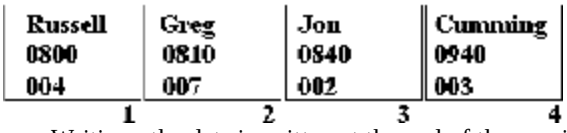
Storage Scenario

To better explain each of these file organisations we will create 4 records and place them in secondary storage. The records are created by a security guard, and records who passes his desk in the morning and at what time they pass.

The records therefore each have three data items; ‘name’, ‘time’, and ‘id number’. Only four people arrive for work:

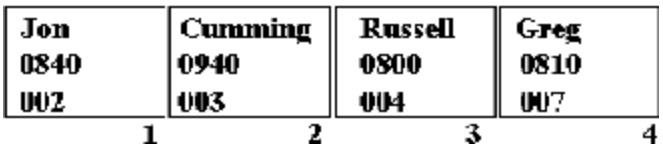
- name=‘Russell’ at time=‘0800’ with id_number=‘004’.
- name=‘Greg’ at time=‘0810’ with id_number=‘007’.
- name=‘Jon’ at time=‘0840’ with id_number=‘002’.
- name=‘Cumming’ at time=‘0940’ with id_number=‘003’.

Serial Organisation



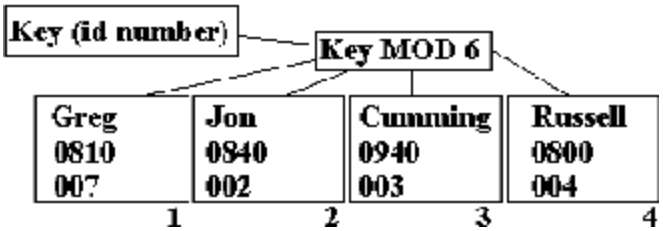
- Writing - the data is written at the end of the previous record.
- Reading - reading records in the order they were written is a cheap operation.
- Trying to find a particular record means you have to read each record in turn until you locate it. This is expensive.
- Deleting - Deleting data in such an structure usually means marking the data as deleted (thus not actually removing it) which is cheap but wasteful or rewriting the whole file to overwrite the deleted record (space-efficient but expensive).

Sequential Organisation



- Writing - records are in ‘id number’ order, thus new records may need to be inserted into the store needing a complete file copy (expensive).
- Deleting - as with serial, either leave holes or perform make file copies.
- Reading - reading records in ‘id number’ order is cheap.
- The ability to chose sort order makes this more useful than serial.
- ‘binary search’ could be used. Goto middle of file - if record key greater than that wanted search the low half, else search the high half, until the record is found. (average accesses to find something is $\log_2$no_of_records.)

Hash Organisation



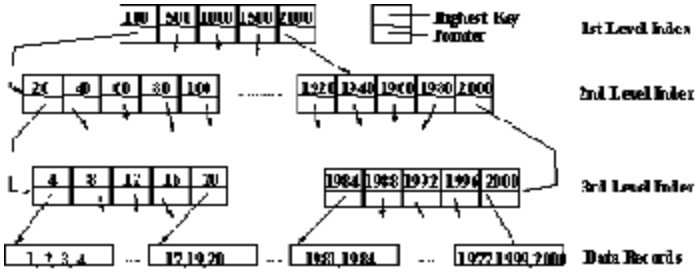
- Writing - Initially the file has 6 spaces (n MOD 6 can be 0-5). To write, calculate the hash and write the record in that location (cheap).
- Deleting - leave holes by marking the record deleted (wasteful of space but cheap to process).
- Reading -
- reading records an order is expensive.
- finding a particular record from a key is cheap and easy.
- If two records can result in the same hash number, then a strategy must be found to solve this problem (which will incur overheads).

Indexed Sequential Access Method

The Indexed Sequential Access Method (ISAM) is frequently used for partial indexes.

- There may be several levels of indexes, commonly 3
- Each index-entry is equal to the highest key of the records or indices it points to.
- The records of the file are effectively sorted and broken down into small groups of data records.
- The indices are built when the data is first loaded as sorted records.
- The index is static, and does not change as records are inserted and deleted
- Insertion and deletion adds to one of the small groups of data records. As the number in each group changes, the performance may deteriorate.

ISAM Example



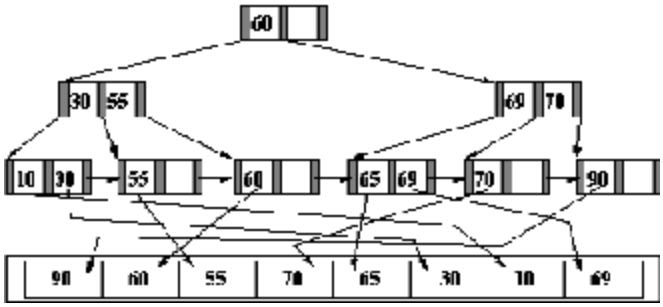
B+ Tree Index

With B+ tree, a full index is maintained, allowing the ordering of the records in the file to be independent of the index. This allows multiple B+ tree indices to be kept for the same set of data records.

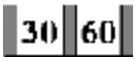
- The lowest level in the index has one entry for each data record.
- The index is created dynamically as data is added to the file.
- As data is added the index is expanded such that each record requires the same number of index levels to reach it (thus the tree stays ‘balanced’).
- The records can be accessed via an index or sequentially.

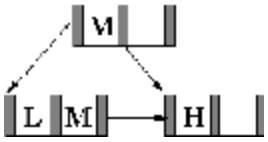
Each index node in a B+ Tree can hold a certain number of keys. The number of keys is often referred to as the ‘order’. Unfortunately, ‘Order 2’ and ‘Order 1’ are frequently confused in the database literature. For the purposes of our coursework and exam, ‘Order 2’ means that there can be a maximum of 2 keys per index node. In this module, we only ever consider order 2 B+ trees.

B+ Tree Example

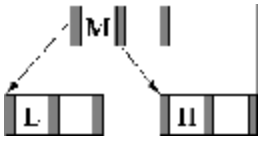


Building a B+ Tree

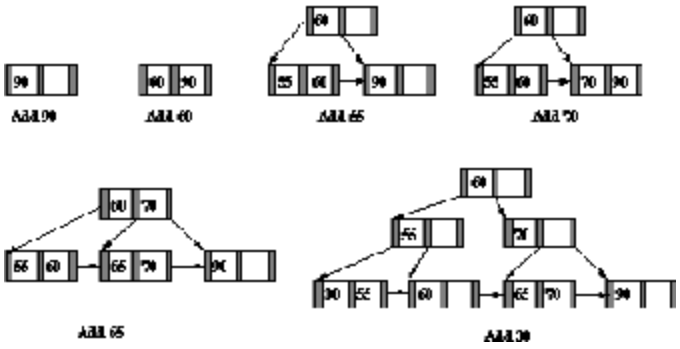
- Only nodes at the bottom of the tree point to records, and all other nodes point to other nodes. Nodes which point to records are called leaf nodes.
- If a node is empty the data is added on the left. 
- If a node has one entry, then the left takes the smallest valued key and the right takes the biggest. 
- If a node is full and is a leaf node, classify the keys L (lowest), M (middle value) and H (highest), and split the node.



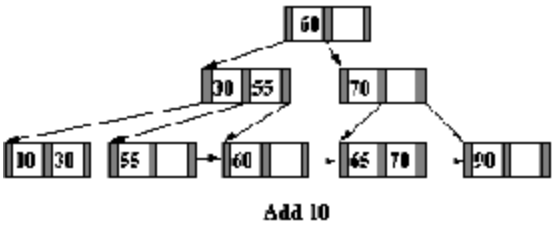
- If a node is full and is not a leaf node, classify the keys L (lowest), M (middle value) and H (highest), and split the node.

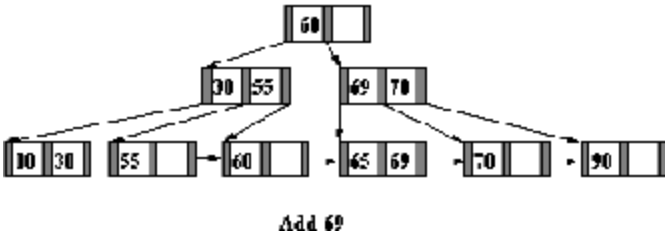


B+ Tree Build Example



Index Structure and Access





Index Structure and Access

- The top level of an index is usually held in memory. It is read once from disk at the start of queries.
- Each index entry points to either another level of the index, a data record, or a block of data records.
- The top level of the index is searched to find the range within which the desired record lies.
- The appropriate part of the next level is read into memory from disc and searched.
- This continues until the required data is found.
- The use of indices reduce the amount of file which has to be searched.

Costing Index and File Access

- The major cost of accessing an index is associated with reading in each of the intermediate levels of the index from a disk (milliseconds).
- Searching the index once it is in memory is comparatively inexpensive (microseconds).
- The major cost of accessing data records involves waiting for the media to recover the required blocks (milliseconds).
- Some indexes mix the index blocks with the data blocks, which means that disk accesses can be saved because the final level of the index is read into memory with the associated data records.

Use of Indexes

- A DBMS may use different file organisations for its own purposes.
- A DBMS user is generally given little choice of file type.
- A B+ Tree is likely to be used wherever an index is needed.
- Indexes are generated:
- (Probably) for fields specified with ‘PRIMARY KEY’ or ‘UNIQUE’ constraints in a CREATE TABLE statement.
- For fields specified in SQL statements such as CREATE [UNIQUE] INDEX indexname ON tablename (col [,col]...);
- Primary Indexes have unique keys.
- Secondary Indexes may have duplicates.

- An index on a column which is used in an SQL 'WHERE' predicate is likely to speed up an enquiry.
- This is particularly so when '=' is involved (equijoin)
- No improvement will occur with 'IS [NOT] NULL' statements
- An index is best used on a column which widely varying data.
- Indexing and column of Y/N values might slow down enquiries.
- An index on telephone numbers might be very good but an index on area code might be a poor performer.
- Multicolumn index can be used, and the column which has the biggest range of values or is the most frequently accessed should be listed first.
- Avoid indexing small relations, frequently updated columns, or those with long strings.
- There may be several indexes on each table. Note that partial indexing normally supports only one index per table.
- Reading or updating a particular record should be fast.
- Inserting records should be reasonably fast. However, each index has to be updated too, so increasing the indexes makes this slower.
- Deletion may be slow.
- Particularly when indexes have to be updated.
- Deletion may be fast if records are simply flagged as 'deleted'.

Review Question

1. What are B trees?

References

http://www.microsoft-accesssolutions.co.uk
http://www.cs.sfu.ca/CC/354

Notes:

LESSON 23
INDEXING AND HASH LOOK UP

Hi! In this chapter I am going to discuss with you about the usage of indexing in DBMS.

Anyway Databases are used to store information. The principle operations we need to perform, therefore, are those relating to

a. Creation of data,

b. Changing some information, or

c. Deleting some information which we are sure is no longer useful or valid.

We have seen that in terms of the logical operations to be performed on the data, relational tables provide a beautiful mechanism for all of the three above tasks. Therefore the storage of a Database in a computer memory (on the Hard Disk, of course), is mainly concerned with the following issues:

1. The need to store a set of tables, where each table can be stored as an independent file.
2. The attributes in a table are closely related, and therefore, often accessed together.

Therefore it makes sense to store the different attribute values in each record contiguously. In fact, the attributes MUST be stored in the same sequence, for each record of a table.

It seems logical to store all the records of a table contiguously; however, since there is no prescribed order in which records must be stored in a table, we may choose the sequence in which we store the different records of a table.

A Brief Introduction to Data Storage on Hard Disks

Each Hard Drive is usually composed of a set disks. Each Disk has a layer of magnetic material deposited on its surface. The entire disk can contain a large amount of data, which is organized into smaller packages called BLOCKS (or pages). On most computers, one block is equivalent to 1 KB of data (= 1024 Bytes).

A block is the smallest unit of data transfer between the hard disk and the processor of the computer. Each block therefore has a fixed, assigned, address. Typically, the computer processor will submit a read/write request, which includes the address of the block, and the address of RAM in the computer memory area called a buffer (or cache) where the data must be stored/ taken from. The processor then reads and modifies the buffer data as required, and, if required, writes the block back to the disk.

How are tables stored on Disk?

We realize that each record of a table can contain different amount of data. This is because in some records, some attribute values may be 'null'. Or, some attributes may be of type varchar (), and therefore each record may have a different length string as the value of this attribute.

Therefore, the record is stored with each subsequent attribute separated by the next by a special ASCII character called a field separator. Of course, in each block, there we may place many records. Each record is separated from the next, again by another special ASCII character called the record separator. The figure below shows a typical arrangement of the data on a disk.

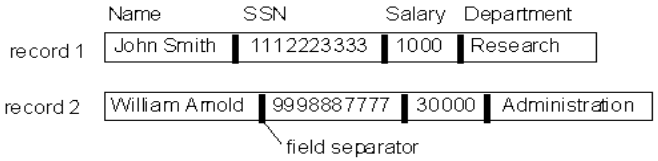
How indexes improve the performance?

Indexes improve the performance of queries that select a small percentage of rows from a table. As a general guideline, create indexes on tables that are queried for less than 2% or 4% of the table's rows. This value may be higher in situations where all data can be retrieved from an index, or where the indexed columns and expressions can be used for joining to other tables.

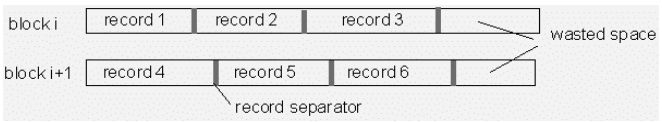
This guideline is based on these assumptions:

- Rows with the same value for the key on which the query is based are uniformly distributed throughout the data blocks allocated to the table
- Rows in the table are randomly ordered with respect to the key on which the query is based
- The table contains a relatively small number of columns
- Most queries on the table have relatively simple WHERE clauses
- The cache hit ratio is low and there is no operating system cache

If these assumptions do not describe the data in your table and the queries that access it, then an index may not be helpful unless your queries typically access at least 25% of the table's rows



The records of such a table may be stored, typically, in a large file, which runs into several blocks of data. The physical storage may look something like the following:



The simplest method of storing a Database table on the computer disk, therefore, would be to merely store all the records of a table in the order in which they are created, on contiguous blocks as shown above, in a large file. Such files are called HEAP files, or a PILE.

Storage Methods in Terms of the Operations

We shall examine the storage methods in terms of the operations we need to perform on the Database:

In a HEAP file:

Operation: Insert a new record

Performance: Very fast

Method: The heap file data records the address of the first block, and the file size (in blocks). It is therefore easy to calculate the last block of the file, which is directly copied into the buffer. The new record is inserted at the end of the last existing record, and the block is written back to the disk.

Operation: Search for a record (or update a record)

Performance: Not too good (on an average, O(b/2) blocks will have to be searched for a file of size b blocks.)

Method: Linear search. Each block is copied to buffer, and each record in the block is checked to match the search criterion. If no match is found, go to the next block, etc.

Operation: Delete a record

Performance: Not too good.

Method: First, we must search the record that is to be deleted (requires linear search). This is inefficient0. Another reason this operation is troublesome is that after the record is deleted, the block has some extra (unused) space. What should we do about the unused space?

To deal with the ***deletion problem***, two approaches are used:

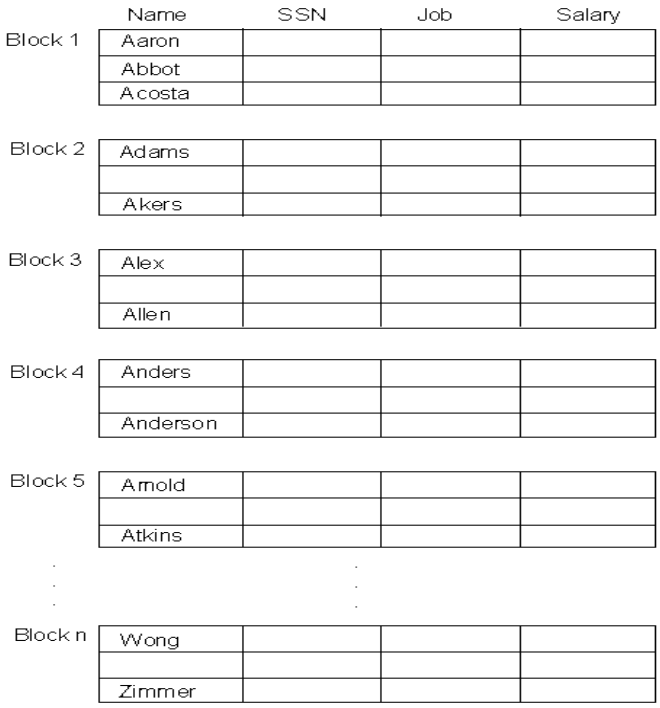
a. Delete the space and rewrite the block. At periodic intervals (say few days), read the entire file into a large RAM buffer and write it back into a new file.

b. For each deleted record, instead of re-writing the entire block, just use an extra bit per record, which is the 'RECORD_DELETED' marker. If this bit has a value 1, the record is ignored in all searches, and therefore is equivalent to deleting the record. In this case, the deletion operation only requires setting one bit of data before re-writing the block (faster). However, after fixed intervals, the file needs to be updated just as in case (a), to recover wasted space.

Heaps are quite inefficient when we need to search for data in large database tables. In such cases, it is better to store the records in a way that allows for very fast searching.

The simplest method to do so is to organize the table in a Sorted File. The entire file is sorted by increasing value of one of the fields (attribute value). If the ordering attribute (field) is also a key attribute, it is called the ordering key.

The figure below shows an example of a sorted file of n blocks.



In addition, most DBMS's provide another mechanism to quickly search for records, at the cost of using some extra disk space. This is the use of INDEXES.

What is an INDEX?

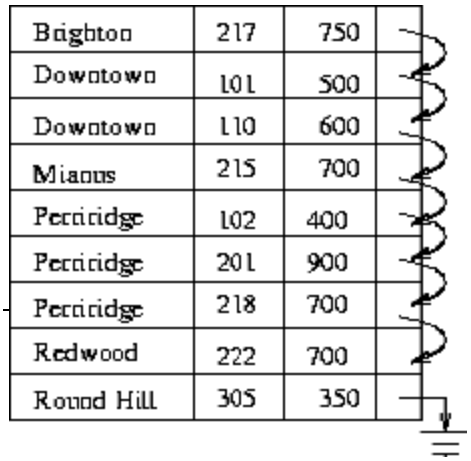
In a book, the index is an alphabetical listing of topics, along with the page number where the topic appears. The idea of an INDEX in a Database is similar. We will consider two popular types of indexes, and see how they work, and why they are useful.

Ordered Indices

1. In order to allow fast **random** access, an index structure may be used.
2. A file may have several indices on different search keys.
3. If the file containing the records is sequentially ordered, the index whose search key specifies the sequential order of the file is the **primary index**, or **clustering index**. Note: The search key of a primary index is usually the primary key, but it is not necessarily so.
4. Indices whose search key specifies an order different from the sequential order of the file are called the **secondary indices**, or **nonclustering indices**.

Primary Indexes

Index-sequential files: Files are ordered sequentially on some search key, and a primary index is associated with it

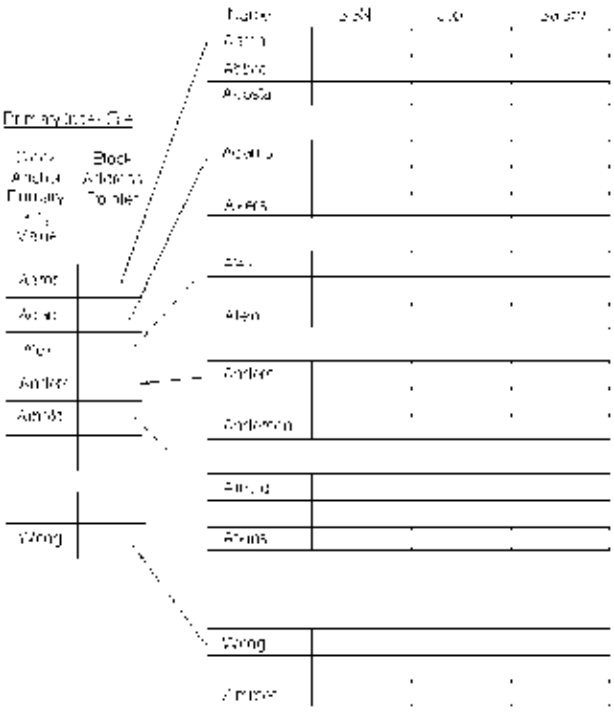


Consider a table, with a Primary Key Attribute being used to store it as an ordered array (that is, the records of the table are stored in order of increasing value of the Primary Key attribute.)

As we know, each BLOCK of memory will store a few records of this table. Since all search operations require transfers of complete blocks, to search for a particular record, we must first need to know which block it is stored in. If we know the address of the block where the record is stored, searching for the record is VERY FAST!

Notice also that we can order the records using the Primary Key attribute values. Thus, if we just know the primary key attribute value of the first record in each block, we can determine quite quickly whether a given record can be found in some block or not.

This is the idea used to generate the Primary Index file for the table. We will see how this works by means of a simple example.



Again a problem arises...

The Primary Index will work only if the file is an ordered file. What if we want to insert a new record?

Answer for that question is

We are not allowed to insert records into the table at their proper location. This would require (a) finding the location where this record must be inserted, (b) Shifting all records at this location and beyond, further down in the computer memory, and (c) inserting this record into its correct place.

Clearly, such an operation will be very time-consuming!

So what is the solution?

The solution to this problem is simple. When an insertion is required, the new record is inserted into an unordered set of records called the overflow area for the table. Once every few days, the ordered and overflow tables are merged together, and the Primary Index file is updated.

Thus any search for a record first looks for the INDEX file, and searches for the record in the indicated Block. If the record is not found, then a further, linear search is conducted in the overflow area for a possible match.

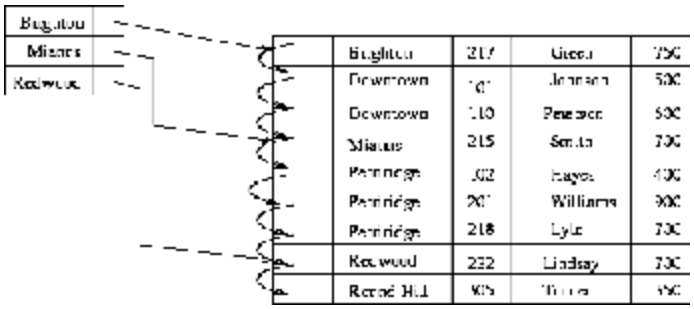
Dense and Sparse Indices

1. There are Two types of ordered indices:
   - Dense Index:
   - An index record appears for **every** search key value in file.
   - This record contains search key value and a pointer to the actual record.

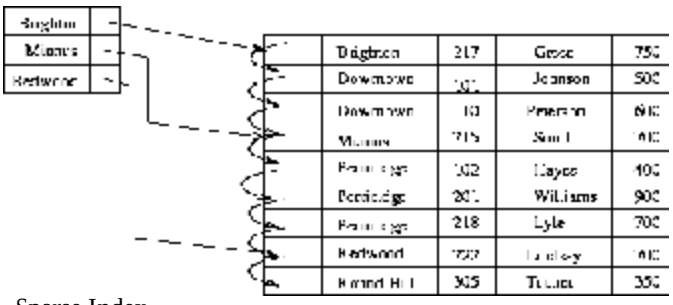
Sparse Index:

- Index records are created only for **some** of the records.
- To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
- We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record.

2. Figures 11.2 and 11.3 show dense and sparse indices for the deposit file.



Dense Index



Sparse Index

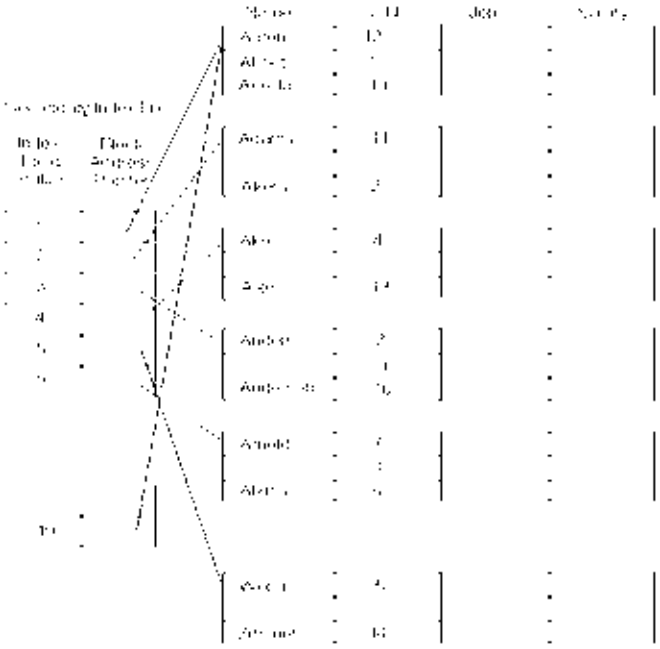
Secondary Indexes

Apart from primary indexes, one can also create an index based on some other attribute of the table. We describe the concept of Secondary Indexes for a Key attribute (that is, for an attribute which is not the Primary Key, but still has unique values for each record of the table). In our previous example, we could, for instance, create an index based on the SSN.

The idea is similar to the primary index. However, we have already ordered the records of our table using the Primary key. We cannot order the records again using the secondary key (since that will destroy the utility of the Primary Index !)

Therefore, the Secondary Index is a two column file, which stores the address of EVERY tuple of the table !

The figure below shows the secondary index for our example:



Unlike the Primary Index, where we need to store only the Block Anchor, and its Block Address, in the case of Secondary Index Files, we need to store one entry for EACH record in the table. Therefore, secondary index files are much larger than Primary Index Files. You can also create Secondary Indexes for non-Key attributes. The idea is similar, though the storage details are slightly different.

You can create as many indexes for each table as you like !

Creating Indexes using SQL

Example

Create an index file for Lname attribute of the EMPLOYEE Table of our Database.

Solution:

CREATE INDEX myLnameIndex ON EMPLOYEE(Lname);

This command will create an index file which contains all entries corresponding to the rows of the EMPLOYEE table sorted by Lname in Ascending Order.

Example

You can also create an Index on a combination of attributes:

CREATE INDEX myNamesIndex ON EMPLOYEE(Lname, Fname);

Finally, you can delete an index by using the following SQL command:

Example:

DROP INDEX myNamesIndex;
which will drop the index created by the previous command?

Note that every index you create will result in the usage of memory on your Database server. This memory space on the hard Disk is used by the DBMS to store the Index File(s). Therefore, the advantage of faster access time that you get by creating indexes also causes the usage of extra memory space.

Now that was all about Indexing, I would like to proceed to one of the similar kind of a topic like indexing, which is hashing.

What is hashing?

A hashed data structure, typically referred to as a hash table provides the following performance pattern:

1. Insertion, deletion, and search for a specific element
2. Search for a successive item in sorted order

A hash table is useful in an application that needs to rapidly store and look up collection elements, without concern for the sorted order of the elements in the collection.

A hash table is not good for storing a collection in sorted order.

Lookup Tables

Before we look into the details of hash tables, we should consider the more general top of a lookup table - a collection in which information is located by some lookup key.

In the collection class examples we've studied so far, the elements of the collection have been simple strings or numbers. In real-world applications, the elements of a collection are frequently more complicated than this, i.e., they're some form of record structure. For example, some applications may need a collection to store a simple list of string names, such as

["Baker", "Doe", "Jones", "Smith"]

In many other cases, there may be additional information associated names, such as age, id, and address; e.g.,

[{"Baker, Mary", 51, 549886295, "123 Main St."},

{"Doe, John", 28, 861483372,"456 1st Ave."},

...

]

In such a structure, collection elements are typically called information records.

Information Records.

Information records can be implemented using a class in the normal way, e.g.,

```
class InformationRecord {
   String name;       // Person name
   int age;           // Age
   int id;            // Unique id
   String address;    // Home address
}
```

When one needs to search for information records in a collection, one of the fields is designated as a **unique** key. This key uniquely identifies each record so it can be located unambiguously. In the above InformationRecord, the id field is a good choice for unique key. A collection structure that holds keyed information records of some form is generally referred to as a **lookup table**.

The unique key is used to lookup an entry in the table. If an entry of a given key is found, the entire entry is retrieved. The implementations we have studied for linked lists and trees can be used as lookup tables, since the type of element has been Object. In the case of a hash table, the structure is specifically suited for use as a lookup table.

The Basic Idea of Hashing.

Suppose we have a collection of personal information records of the form shown above in the InformationRecord class, where the size of the collection is a maximum of 10,000 records. Suppose further that we want rapid access to these records by id. A linked list would be a pretty poor choice for implementing the collection, since search by id would take O(N) time. If we kept the collection sorted by id, a balanced search tree would give us O(log N) access time. If we put the records in an array of 1,000,000,000 elements we could get O(1) access by id, but we'd waste a lot of space since we only have 10,000 active records.

Is there some way that we could get O(1) access as in an array without wasting a lot of space?

The answer is hashing, and it's based on the following idea: Allocate an array of the desired table size and provide a function that maps any key into the range 0 to TableSize-1. The function that performs the key mapping is called the hashing function.

This idea works well when the hashing function evenly distributes the keys over the range of the table size. To ensure good performance of a hash table, we must consider the following issues:

- Choosing a good hashing function that evenly maps keys to table indices;
- Choosing an appropriate table size that's big enough but does not waste too much space;
- Deciding what to do when the hashing function maps two different keys to the same table location, which condition is called a collision.

A Simple Example

Suppose again we need a table of 10,000 InformationRecords with the id field used as the lookup key. We'll choose a hash table size of 10,000 elements. For the hashing function, we'll use the simple modulus computation of id mod 10000; if keys are randomly distributed this will give a good distribution. To resolve collisions, we'll use the simple technique of searching down from the point of collision for the first free table entry.

If we insert the entries listed above for Mary Baker and John Doe.The hashing function computes the indices 6295 and 3372, respectively, for the two keys. The records are placed at these locations in the table array.

Suppose were next to add the record **{"Smith, Jane", 39, 861493372, "789 Front St."}**

In this case, the hashing function will compute the same location for this record as for Mary Baker, since the id keys for Mary Baker and Jane Smith happen to differ by exactly 10,000. To resolve the collision, we'll put the Jane Smith entry at the next available location in the table, which is 6296.

Things that can go wrong.

In the preceding example, things worked out well, given the nature of the keys and the bounded table size. Suppose, however, some or all of the following conditions were to arise: The number or records grew past 10,000. Due to some coincidence of locale, a large number of ids differed by exactly 10,000. e wanted to use the name field as the search key instead of id. In such cases, we need to reconsider one or all of our choices for hashing function, table size, and/or collision resolution strategy.

Choosing a good hash function.

The choice of hash function depends significantly on what kind of key we have.

In the case of numeric key with random distribution, the simple modulus hashing function works fine. However, if numeric keys have some non-random properties, such as divisibility by the table size, the modulus hashing function does not work well at all. If we use a non-numeric key, such as a name string, we must first convert the string into a number of some form before applying mod. In practical applications, lookup keys are frequently strings, hence some consideration of good string-valued hash functions is in order.

Good hashing of string-valued keys

Approach 1: add up the character values of the string and then compute the modulus.

The advantage of this approach is that it's simple and reasonably fast if the number of characters in a string is not too large. The disadvantage is that it may not distribute key values very well at all. For example, suppose keys are eight characters or fewer (e.g., UNIX login ids) and the table size is 10,000. Since ASCII string characters have a maximum value of 127, the summing formula only produces values between 0 and 127*8, which equals 1,016. This only distributes keys to a bit more than 10% of the table.

Approach 2: use a formula that increases the size of the hash key using some multiplier.

This approach is also simple and fast, but it may also not distribute keys well. E.g., one formula could be to sum the first three characters of a key string as follows:

(char[0] + (27 * char[1]) + (729 * char[2])

and then compute the modulus.

The rationale for the number 27 is that it's the number of letters in the alphabet, plus one for a space character; 729 is 27^2.

If string name characters are equally likely to occur, this distributes keys in the range 0 to 26^3 = 17,576.

However, empirical analysis of typical names shows that for the first three characters, there are only 2,851 combinations, which is not good coverage for a 10,000-element table.

Approach 3: sum all key characters with a formula that increases the size and mixes up the letters nicely.

An empirically derived formula to do this is the following:

$(37 * char[0]) + (37^2 * char[1]) + ... + (37^{(l-1)} * char[l])$

where 37 is the empirically-derived constant and l = the string length of the key.

This formula, plus similar ones with variants on the constant multiplier, has been shown to do a good job of mixing up the string characters and providing good coverage even for large table.

Review Questions

1. What is the use of indexing?
2. What are primary index and secondary index?
3. Explain hashing?
4. What are look up tables?

References

http://www.microsoft-accesssolutions.co.uk

http://www.cs.sfu.ca/CC/354

Notes:

LESSON 24
QUERY PROCESSING AND QUERY OPTIMISER PART-I

Hi today you are going to learn how is query processed in the background by DBMS

Introduction

When a program written in a procedural language, for example Pascal or C, is to be executed, the program is first translated usually by a compiler into machine language. The translation involves taking the source code and generating equivalent machine code that preserves the sequence of operations that the source code specifies. As part of the translation process, the compiler carries out code optimization to generate as efficient a code as possible. This might include elimination of unnecessary writing of values to the memory and reading them to the registers, some clever unrolling of the loops, etc.

In a non-procedural or a declarative language like SQL, no sequence of operations is explicitly given and therefore a query may be processed in a number of different ways (often called *plans*) where each plan might have a different sequence of operations. Optimization in non-procedural languages therefore is much more complex task since it not only involves code optimization (how to carry out the operations that need to be carried out) but also selecting the best plan as well as selecting the best access paths. In many situations, especially if the database is large and the query is complex, a very large number of plans are normally possible and it is then not practical to enumerate them all and select the best. Often then, it is necessary to consider a small number of possible plans and select the best option from those. Since the savings from query optimization can be substantial, it is acceptable that a database spend time in finding the best strategy to process the query. Again, of course, we assume that we are dealing with queries that are expected to consume significant amount of computing resources; there is no point in spending time optimizing queries that are so simple that any approach can be used to execute them in a small amount of time.

To illustrate the issues involved in query optimization, we consider the following example database that we have used in previous chapters:

student(student_id, student_name, address)
enrolment(student_id, subject_id)
subject(subject_id, subject_name, department, instructor)

Now consider the query "find the names of subjects that John Smith is enrolled in". As we have shown earlier, this query may be formulated in a number of different ways. Three possible formulations of the query are:

1. SELECT *subject_name*
   FROM *student, enrolment, subject*
   WHERE *student.student_id = enrolment.student_id*
   AND *enrolment.subject_id = subject.subject_id*
   AND *student_name* = "John Smith"

2. SELECT *subject_name*
   FROM *subject*
   WHERE *subject_id* IN
   (SELECT *subject_id*
   FROM *enrolment*
   WHERE *student_id* IN
   (SELECT *student_id*
   FROM *student*
   WHERE *student_name* = "John Smith"))

3. SELECT *subject_name*
   FROM *subject*
   WHERE *exists*
   (SELECT *subject_id*
   FROM *enrolment*
   WHERE *subject.subject_id = enrolment.subject_id*
   AND *exists*
   (SELECT *student_id*
   FROM *student*
   WHERE *enrolment.student_id = student.student_id*
   AND *student_name* = "John Smith"))

The three query formulations above suggest the following three different query processing plans:

a. The first formulation suggests a plan in which the natural join of relations *student* and *enrolment* is carried out followed by the join of the result with the relation *subject*. This is in turn followed by a restriction *student_name = "John Smith"* of the result followed by a projection.

b. The second formulation suggests a plan that involves first applying a restriction and a projection to the relation *student* to obtain a relation that has only the *student_id* of John Smith in it. This is then followed by a restriction and a projection of the relation *enrolment* and finally a restriction and a projection of the relation *subject*.

c. The third query formulation suggests yet another plan. In this formulation, the relation *subject* is scanned one tuple at a time and the tuple is selected only if there is a corresponding tuple in the relation *enrolment*. To find if there is a corresponding tuple in *enrolment*, the relation *enrolment* is scanned one tuple at a time and a tuple is selected only if it has a corresponding tuple in the relation *student* with the name "John Smith".

The difference between the costs of implementing the above queries is quite large as we will show later in this chapter.

Given that the same query may be formulated in several different ways and the one query formulation may be processed in several different ways, the following questions arise:

1. How does a DBMS process a query?
2. Does the formulation of the query determine the query processing plan?
3. Is it possible to recognise different semantically equivalent queries?
4. What is the most efficient plan to process the above query?

This chapter addresses these questions as well as questions dealing with efficient methods for computing relational operators like selection, projection and join.

We should note that we do not believe that the user should know which formulation of a query that can be formulated in many different ways is the most efficient. Users are not expected to be knowledgeable in the workings of a DBMS to be able to predict what formulation would be the most efficient. It is therefore essential that the burden of finding an efficient execution plan be placed on the system and not on the user.

We will in this chapter assume that the database that we are dealing with is too large to fit in the main memory. The database therefore primarily resides on the secondary memory and parts of it are copied to the main memory as necessary. The secondary storage will be assumed to be partitioned in blocks, each block typically storing a number of tuples from a relation, and the data movements between the main and secondary memory take place in blocks. A typical size of a block is assumed to be 1K Bytes although recent machines have tended to use larger block sizes.

Often the cost of secondary memory access is the main bottleneck in a database system and therefore frequently the number of block read or written is taken as the cost of carrying out a particular operation. We assume that processing of each block takes much smaller time than reading or writing it to the secondary storage although this is not always the case.

Query Processing By A DBMS

In most database systems, queries are posed in a non-procedural language like SQL and as we have noted earlier such queries do not involve any reference to access paths or the order of evaluation of operations. The query processing of such queries by a DBMS usually involves the following four phases:

1. Parsing
2. Optimization
3. Code Generation
4. Execution

(See Goetz paper, page 76)

The parser basically checks the query for correct syntax and translates it into a conventional parse-tree (often called a *query-tree*) or some other internal representation. (An example of a query and its corresponding query tree are presented in the paper by Talbot).

If the parser returns with no errors, and the query uses some user-defined views, it is necessary to expand the query by making appropriate substitutions for the views. It is then necessary to check the query for semantic correctness by consulting the system catalogues and check for semantic errors and type compatibility in both expressions and predicate comparisons.

The optimizer is then invoked with the internal representation of the query as input so that a *query plan* or *execution plan* may be devised for retrieving the information that is required. The optimizer carries out a number of operations. It relates the symbolic names in the query to data base objects and checks their existence and checks if the user is authorized to perform the operations that the query specifies.

In formulating the plans, the query optimizer obtains relevant information from the metadata that the system maintains and attempts to model the estimated costs of performing many alternative query plans and then selects the best amongst them. The metadata or system catalog (sometime also called *data dictionary* although a data dictionary may or may not contain other information about the system E&N p479) consists of descriptions of all the databases that a DBMS maintains. Often, the query optimizer would at least retrieve the following information:

1. Cardinality of each relation of interest.
2. The number of pages in each relation of interest.
3. The number of distinct keys in each index of interest.
4. The number of pages in each index of interest.

The above information and perhaps other information will be used by the optimizer in modelling the cost estimation for each alternative query plan.

Considerable other information is normally available in the system catalog:

1. Name of each relation and all its attributes and their domains.
2. Information about the primary key and foreign keys of each relation.
3. Descriptions of views.
4. Descriptions of storage structures.
5. Other information including information about ownership and security issues.

Often this information is updated only periodically and not at every update/insert/delete. (Selinger et al) Also, the system catalog is often stored as a relational database itself making it easy to query the catalog if a user is authorized to do so.

(example of catalog on page 481 E&N)

Information in the catalog is very important of course since query processing makes use of this information extensively. Therefore more comprehensive and more accurate information a database maintains the better optimization it can carry out but maintaining more comprehensive and more accurate information also introduces additional overheads and a good balance therefore must be found.

The catalog information is also used by the optimizer in access path selection. These statistics are often updated only periodically and are therefore not always accurate.

An important part of the optimizer is the component that consults the metadata stored in the database to obtain statistics about the referenced relations and the access paths available on them. These are used to determine the most efficient order of the relational operations and the most efficient access paths. The

order of operations and the access paths are selected from a number of alternate possibilities that normally exist so that the cost of query processing is minimized. More details of query optimization are presented in the next section.

If the optimizer finds no errors and outputs an execution plan, the code generator is then invoked. The execution plan is used by the code generator to generate the machine language code and any associated data structures. This code may now be stored if the code is likely to be executed more than once. To execute the code, the machine transfers control to the code which is then executed.

Query Optimization

The query optimizer is a very important component of a database system because the efficiency of the system depends so much on the performance of the optimizer. Before we discuss optimization in detail, we should note that queries to the database may be posed either interactively or in a batch mode. When queries are posed interactively, the system can only hope to optimize processing of each query separately. In batch environment, where an application program may include a number of queries, it may be desirable to attempt global optimization. In this section we only deal with individual query optimization.

Before query optimization is carried out, one would of course need to decide what needs to be optimized. The goal of achieving efficiency itself may be different in different situations. For example, one may wish to minimize the processing time but in many situations one would wish to minimize the response time. In other situations, one may wish to minimize the I/O, network time, memory used or some sort of combination of these e.g. total resources used. Generally, a query processing algorithm A will be considered more efficient than an algorithm B if the measure of cost being minimized for processing the same query given the same resources using A is generally less than that for B.

Although it is customary to use the term *query optimization* for the heuristic selection of strategies to improve the efficiency of executing a query, query optimization does not attempt to exactly optimize the cost of query processing. Exact optimization of the cost is usually computationally infeasible. Also given that many of the statistics about the database available to the optimizer are likely to be estimates, an exact optimization is not necessarily a desirable goal. The goal therefore often is to design execution plans that are reasonably efficient and close to optimal.

To illustrate the desirability of optimization, we now present an example of a simple query that may be processed in several different ways. The following query retrieves subject names and instructor names of all current subjects in Computer Science that John Smith is enrolled in.

SELECT *subject.name, instructor*
FROM *student, enrolment, subject*
WHERE *student.student_id = enrolment.student_id*
AND *subject.subject_id = enrolment.subject_id*
AND *subject.department* = 'Computer Science'
AND *student.name* = 'John Smith'

To process the above query, two joins and two restrictions need to be performed. There are a number of different ways these may be performed including the following:

1. Join the relations *student* and *enrolment*, join the result with *subject* and then do the restrictions.
2. Join the relations *student* and *enrolment*, do the restrictions, join the result with *subject*
3. Do the restrictions, join the relations *student* and *enrolment*, join the result with *subject*
4. Join the relations *enrolment* and *subject*, join the result with *student* and then do the restrictions.

Before we attempt to compare the costs of the above four alternatives, it is necessary to understand that estimating the cost of a plan is often non- trivial. Since normally a database is disk-resident, often the cost of reading and writing to disk dominates the cost of processing a query. We would therefore estimate the cost of processing a query in terms of disk accesses or block accesses. Estimating the number of block accesses to process even a simple query is not necessarily straight forward since it would depend on how the data is stored and which, if any, indexes are available. In some database systems, relations are stored in packed form, that is, each block only has tuples of the same relation while other systems may store tuples from several relations in each block making it much more expensive to scan all of a relation.

Let us now compare the costs of the above four options. Since exact cost computations are difficult, we will use simple estimates of the cost. We consider a situation where the enrolment database consists of 10,000 tuples in the relation *student*, 50,000 in *enrolment*, and 1,000 in the relation *subject*. For simplicity, let us assume that the relations *student* and *subject* have tuples of similar size of around 100 bytes each and therefore and we can accommodate 10 tuples per block if the block is assumed to be 1 KBytes in size. For the relation *enrolment*, we assume a tuple size of 40 bytes and thus we use a figure of 25 tuples/block. In addition, let John Smith be enrolled in 10 subjects and let there be 20 subjects offered by Computer Science. We can now estimate the costs of the four plans listed above.

The cost of query plan (1) above may now be computed. Let the join be computed by reading a block of the first relation followed by a scan of the second relation to identify matching tuples (this method is called *nested-scan* and is not particularly efficient. We will discuss the issue of efficiency of algebraic operators in a later section). This is then followed by the reading of the second block of the first relation followed by a scan of the second relation and so on. The cost of $R \mid X \mid S$ may therefore be estimated as the number of blocks in R times the number of blocks in S. Since the number of blocks in *student* is 1000 and in *enrolment* 2,000, the total number of blocks read in computing the join of *student* and *enrolment* is $1000 \times 2000 = 2,000,000$ block accesses. The result of the join is 50,000 tuples since each tuple from *enrolment* matches with a tuple from *student*. The joined tuples will be of size approximately 140 bytes since each tuple in the join is a tuple from *student* joined with another from *enrolment*. Given the

tuple size of 140 bytes, we can only fit 7 tuples in a block and therefore we need about 7,000 blocks to store all 50,000 joined tuples. The cost of computing the join of this result with *subject* is $7000 \times 100 = 700,000$ block accesses. Therefore the total cost of plan (1) is approximately 2,700,000 block accesses.

To estimate the cost of plan (2), we know the cost of computing the join of *student* and *enrolment* has been estimated above as 2,000,000 block accesses. The result is 7000 blocks in size. Now the result of applying the restrictions to the result of the join reduces this result to about 5-10 tuples i.e. about 1-2 blocks. The cost of this restriction is about 7000 disk accesses. Also the result of applying the restriction to the relation *subject* reduces that relation to 20 tuples (2 blocks). The cost of this restriction is about 100 block accesses. The join now only requires about 4 block accesses. The total cost therefore is approximately 2,004,604.

To estimate the cost of plan (3), we need to estimate the size of the results of restrictions and their cost. The cost of the restrictions is reading the relations *student* and *subject* and writing the results. The reading costs are 1,100 block accesses. The writing costs are very small since the size of the results is 1 tuple for *student* and 20 tuples for *subject*. The cost of computing the join of *student* and *enrolment* primarily involves the cost of reading *enrolment*. This is 2,000 block accesses. The result is quite small in size and therefore the cost of writing the result back is small. The total cost of plan (3) is therefore 3,100 block accesses.

Similar estimates may be obtained for processing plan (4). We will not estimate this cost, since the above estimates are sufficient to illustrate that brute force method of query processing is unlikely to be efficient. The cost can be significantly reduced if the query plan is optimized. The issue of optimization is of course much more complex than estimating the costs like we have done above since in the above estimation we did not consider the various alternative access paths that might be available to the system to access each relation.

The above cost estimates assumed that the secondary storage access costs dominate the query processing costs. This is often a reasonable assumption although the cost of communication is often quite important if we are dealing with a distributed system. The cost of storage can be important in large databases since some queries may require large intermediate results. The cost of CPU of course is always important and it is not uncommon for database applications to be CPU bound than I/O bound as is normally assumed. In the present chapter we assume a centralised system where the cost of secondary storage access is assumed to dominate other costs although we recognize that this is not always true. For example, system R uses

cost = page fetches + w × cpu utilization

When a query is specified to a DBMS, it must choose the best way to process it given the information it has about the database. The optimization part of query processing generally involves the following operations.

1. A suitable internal representation
2. Logical transformation of the query
3. Access path selection of the alternatives
4. Estimate costs and select best

We will discuss the above steps in detail.

(Discuss prenex NF page 121 Jorge and Koch..show query tree)

- Internal Representation
- Logical Transformations
- Estimating Size of Results
  - Size of a restriction
  - Size of a Projection
  - Cartesian Product
  - Join
- Access Paths
- Estimating Costs

Internal Representation

As noted earlier, a query posed in a query language like SQL must first be translated to a internal representation suitable for machine representation. Any internal query representation must be sufficiently powerful to represent all queries in the query language (e.g. SQL). The internal representation could be relational algebra or relational calculus since these languages are powerful enough (they have been shown to be relationally complete by E.F. Codd) although it will be necessary to modify them from what was discussed in an earlier chapter so that features like Group By and aggregations may be represented. A representation like relational algebra is procedural and therefore once the query is represented in that representation, a sequence of operations is clearly indicated. Other representations are possible. These include object graph, operator graph (or parse tree) and tableau. Further information about other representations is available in Jarke and Koch (1984) although some sort of tree representation appears to be most commonly used (why?). Our discussions will assume that a query tree representation is being used. In such a representation, the leaf nodes of the query tree are the base relations and the nodes correspond to relational operations.

Logical Transformations

At the beginning of this chapter we showed that the same query may be formulated in a number of different ways that are semantically equivalent. It is clearly desirable that all such queries be transformed into the same query representation. To do this, we need to translate each query to some canonical form and then simplify.

This involves transformations of the query and selection of an optimal sequence of operations. The transformations that we discuss in this section do not consider the physical representation of the database and are designed to improve the efficiency of query processing whatever access methods might be available. An example of such transformation has already been discussed in the examples given. If a query involves one or more joins and a restriction, it is always going to be more efficient to carry out the restriction first since that will reduce the size of one of the relations (assuming that the restriction

applies to only one relation) and therefore the cost of the join, often quite significantly.

Heuristic Optimization - In the heuristic approach, the sequence of operations in a query is reorganised so that the query execution time improves. (Talbot?)

Deterministic Optimization - In the deterministic approach, cost of all possible forms of a query are evaluated and the best one is selected.

Common Subexpression - In this technique, common subexpressions in the query, if any, are recognised so as to avoid executing the same sequence of operations more than once. (Common subexpression..Talbot?)

Heuristic optimization

Heuristic optimization often includes making transformations to the query tree by moving operators up and down the tree so that the transformed tree is equivalent to the tree before the transformations. Before we discuss these heuristics, it is necessary to discuss the following rules governing the manipulation of relational algebraic expressions:

1. Joins and Products are commutative. e.g.

   R X S = S X R
   R |X| S = S |X| R

   where |X| may be a θ join or a natural join. The order of attributes in the two products or joins may not be quite the same but the ordering of attributes is not considered significant in the relational model since the attributes are referred to by their name not by their position in the list of attributes.

2. Restriction is commutative. e.g.

   $$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$$

3. Joins and Products are associative. e.g.

   (R X S) X T = R X (S X T)
   (R |X| S) |X| T = R |X| (S |X| T)

   The associativity of the above operations guarantees that we will obtain the same results whatever be the ordering of computations of the operations product and join. Union and intersection are also associative.

4. Cascade of Projections. e.g.

   $$\pi_A(\pi_B(R)) = \pi_A(R)$$

   where the attributes A is a subset of the attributes B. The above expression formalises the obvious that there is no need to take the projection with attributes B if there is going to be another projection which is a subset of B that follows it.

5. Cascade of restrictions. e.g.

   $$\sigma_p(\sigma_q(R)) = \sigma_{p \wedge q}(R)$$

   The above expression also formalises the obvious that if there are two restrictions, one after the other, then there is no need to carry out the restrictions one at a time (since each will require processing a relation) and instead both restrictions could be combined.

6. Commuting restrictions and Projections. e.g.

   $$\sigma_p(\pi_A(R)) = \pi_A \sigma_p(R)$$

   or

   $$\pi_A \sigma_p(R) = \sigma_p(\pi_A(R))$$

   There is no difficulty in computing restriction with a projection since we are then doing the restriction before the projection. However if we wish to commute the projection and the restriction, that is possible only if the restriction used no attributes other than those that are in the projection.

7. Commuting restrictions with Cartesian Product. **In some cases, it is possible to apply commutative law to restrictions and a product. For example,**

   $$\sigma_p(R X S) = \sigma_p(R) X S$$

   or

   $$\sigma_{p \wedge q}(R X S) = \sigma_p(R) X \sigma_q(S)$$

   In the above expressions we have assumed that the predicate p has only attributes from R and the predicate q has attributes from S only.

8. Commuting restriction with a Union.
9. Commuting restriction with a Set Difference.
10. Commuting Projection with a Cartesian Product or a Join - we assume that the projection includes the join predicate attributes.
11. Commuting Projection with a UnioN.

(For an example, refer to the paper by Talbot)

We now use the above rules to transform the query tree to minimize the query cost. Since the cost is assumed to be closely related to the size of the relations on which the operation is being carried out, one of the primary aims of the transformations that we discuss is to reduce the size of intermediate relations.

The basic transformations include the following:

a. Moving restrictions down the tree as far as possible. i The idea is to carry out restrictions as early as possible. If the query involves joins as well as restrictions, moving the restrictions down is likely to lead to substantial savings since the relations that are joined after restrictions are likely to be smaller (in some cases much smaller) than before restrictions. This is clearly shown by the example that we used earlier in this chapter to show that some query plans can be much more expensive than others. The query plans that cost the least were those in which the restriction was carried out first. There are of course situations where a restriction does not reduce the relation significantly, for example, a restriction selecting only women from a large relation of customers or clients.

b. Projections are executed as early as possible. In real-life databases, a relation may have one hundred or more

attributes and therefore the size of each tuple is relatively large. Some relations can even have attributes that are images making each tuple in such relations very large. In such situations, if a projection is executed early and it leads to elimination of many attributes so that the resulting relation has tuples of much smaller size, the amount of data that needs to be read in from the disk for the operations that follow could be reduced substantially leading to cost savings. It should be noted that only attributes that we need to retain from the relations are those that are either needed for the result or those that are to be used in one of the operations that is to be carried out on the relation.

c. Optimal Ordering of the Joins. We have noted earlier that the join operator is associative and therefore when a query involves more than one join, it is necessary to find an efficient ordering for carrying out the joins. An ordering is likely to be efficient if we carry out those joins first that are likely to lead to small results rather than carrying out those joins that are likely to lead to large results.

d. Cascading restrictions and Projections. Sometimes it is convenient to carry out more than one operations together. When restrictions and projections have the same operand, the operations may be carried out together thereby saving the cost of scanning the relations more than once.

e. Projections of projections are merged into one projection. Clearly, if more than one projection is to be carried out on the same operand relation, the projections should be merged and this could lead to substantial savings since no intermediate results need to be written on the disk and read from the disk.

f. Combining certain restrictions and Cartesian Product to form a Join. A query may involve a cartesian product followed by a restriction rather than specifying a join. The optimizer should recognise this and execute a join which is usually much cheaper to perform.

g. Sorting is deferred as much as possible. Sorting is normally a nlogn operation and by deferring sorting, we may need to sort a relation that is much smaller than it would have been if the sorting was carried out earlier.

h. A set of operations is reorganised using commutativity and distribution if a reorganised form is more efficient.

Estimating Size of Results

The cost of a query evaluation plan depends on the sizes of the basic relations referenced as well as the sizes of the intermediate results. To obtain reasonable cost estimates of alternate plans, the query optimizer needs estimates of the various intermediate results.

To estimate sizes of the intermediate results, we define the concept of selectivity factor. The selectivity factor roughly corresponds to the expected fraction of tuples which will satisfy the predicates. For the restriction operation, the selectivity factor is the ratio of the cardinality of the result to the base relation. Selectivity of a restriction is estimated by the DBMS by maintaining profiles of attribute value distribution or by making suitable assumptions of the distribution. The selectivity of projection is the ratio of tuple size reduction. Often however it is not only that some attributes are being removed in a projection, the duplicates are also removed. The join selectivity of one relation with another defines the ratio of the attribute values that are selected in one of the relations.

We now consider methods of estimating the cost of several relational operations. Let us consider relations *R* and *S* with n_R and n_S number of tuples respectively. Let b_R and b_S be the number of tuples per block for the relations *R* and *S* respectively. Also, assume further, that *D(A,S)* be the number of distinct values of attribute A in relation S.

- Size of a restriction
- Size of a Projection
- Cartesian Product
- Join

Size of a Restriction

A restriction may involve a number of types of predicates. The following list is presented by Selinger *et al*:

1. attribute = value
2. attribute1 = attribute2
3. attribute > value
4. attribute between value1 and value2
5. attribute IN (list of values)
6. attribute IN subquery
7. pred expression OR pred expression
8. pred expression AND pred expression

Let us first consider the simplest restriction (1) and consider a relation *R* that has an attribute *A* on which an equality condition has been specified. If we assume that values of attribute *A* are distributed uniformly, we would expect the result of the restriction (1) above to have approximately $n_R / D(A, R)$ tuples where *D(A,R)*, as noted above, is the number of different values that attribute *A* takes in relation *R*. Although the assumption of uniform distribution is almost always unrealistic, the above estimate is easily obtained and is often quite useful.

It is useful to define *selectivity* as the ratio of the number of tuples satisfying the restriction condition to the total number of tuples in the relation.

Size of a Projection

The projection operation removes a number of attributes and the DBMS has enough information to estimate the size of the projected relation if it had no duplicates. When duplicates occur, as often is the case, estimation of the size of the projected relation is difficult (Ahad??).

Cartesian Product

Estimating the size of a cartesian product is quite simple since the number of tuples in the cartesian product is $n_r * n_s$.

Join

It is often necessary to estimate the size of the result of a join R |X| S. If the join attribute(s) of one of the relation R is the key of the relation, the cardinality of the join cannot be bigger than the cardinality of the other relation S. When neither of the join attributes are a key for their relation, finding the size of the resulting relation is more difficult. One approach that is then often used is to make use of the statistics from metadata, for example information like *D(A, R)* which is an estimate of the number of different values of the attribute A in relation R. If A is the join attribute between relations R and S, we may be able to assume that each value of join attribute in R also appears in *S* and that given that there are n_s tuples in S and there are *D(A, S)* distint values of A in S, we can assume uniform distribution and conclude that for each value of A in R there will be $n_s / D(A, S)$ matching tuples in S. Therefore an estimate of the size of the join is $n_r n_s / D(A, S)$. Another estimate would of course be $n_r n_s / D(A, R)$. Often the two estimates are not identical since the assumption of all tuples from one relation participating in the join does not apply equally well to both relations. For example, only 80 tuples in *R* might be participating in the join while only 50 are participating from S. (Explain this??)

(Computing costs p 27 Selinger) Although the cardinality of the join of n relations is the same regardless of the join order, the cost of joining in different order can be substantially different. (page 28 Selinger et al) If we are dealing with relations *R(A, B)* and *S(B, C)*, we need to find how many tuples of *S* on the average match with each tuple of *R* (i.e. have the same value).

Access Paths

[image size = size of index = size of different values] A software component of the DBMS maintains the physical storage of relations and provides access paths to these relations. Often relations in a database are stored as collection of tuples which may be accessed a tuple at a time along a given access path. The access paths may involve an index on one or more attributes of the relation. The indexes are often implemented as B-trees. An index may be scanned tuple by tuple providing a sequential read along the attribute or attributes in the index. The tuples may of course be scanned using a sequential scan.

This involves selection of suitable access paths.

B-tree better for range queries. Hashing is useless.

A relation may have one or more indexes available on it. In fact, a relation with many attributes could well have many indexes such that the storage occupied by the indexes becomes as large as the storage occupied by the relation. Indexes may use hashing or B-tree. They allow very efficient search when the tuple with a given value of an attribute needs to be found.

The two steps are not separate and are often carried out together. The logical transformations may involve using one or more of the following techniques:

A query may involve a number of operations and would often be a number of different ways these operations could be processed. The DBMS must therefore estimate the cost of the different alternatives and choose the alternative with the least estimate. The estimates of costs are often based on statistics about the sizes of the relations and distribution of key values (or ranges).

Consider the task of estimating the cost of one of the options, for example (2) above. The query optimizer needs to estimate the cost of join of student and enrolment, the cost of restriction and the cost of joining the result with subject. To estimate these costs, a DBMS would normally maintain statistics about the sizes of relations and a histogram of the number of tuples within various key ranges. Given the statistics, the size of the join of student and subject can be estimated as

$$n_{join} = s_{student,subject}(n_{student} X n_{subject})$$

where $n_{student}$ and $n_{subject}$ are the number of tuples in *student* and *subject* respectively and $s_{student,subject}$ is called the selectivity of the join. It is clearly very difficult to estimate $s_{student,subject}$ accurately and most query optimizers use quite crude estimates. Fortunately, experiments have shown that selection of optimal plan for processing a plan is not very sensitive to inaccurate estimation of join selectivities.

Query optimization in a DBMS is carried out by a piece of software often called a query planner. The query planner is given

1. The query in canonical form
2. Information about cardinalities and degrees of relations involved in the query.
3. Information about the indexes
4. Information about any fields on which relations have been sorted
5. Estimates of selectivities of joins involved in the query

The planner than produces a plan that presents a suggested sequence of operations as well as details of how the operations should be carried out (for example, when an index is to be used).

Estimating Costs

It should by now be clear that most queries could be translated to to a number of semantically equivalent query plans. The process followed so far should eliminate most alternatives that are unlikely to be efficient but one is still likely to have a number of plans that could well be reasonably efficient. The cost of these alternatives must be estimated and the best plan selected. The cost estimation will of course require the optimizer to consult the metadata.

The metadata or system catalog (sometime also called data dictionary although a data dictionary may or may not contain other information about the system E&N p479) consists of descriptions of the databases that a DBMS maintains. The following information is often available in the system catalog:

1. Name of each relation and all its attributes and their domains.
2. Information about primary key and foreign keys of each relation.

3. Cardinality of each relation.
4. The number of pages in each relation.
5. Descriptions of views.
6. Descriptions of storage structures.
7. The number of distinct keys in each index.
8. The number of pages in an index.
9. Other information including ownership and security issues.

Often there statistics are updated only periodically and not at every update/insert/delete. (Selinger et al) Also, the system catalog is often stored as a relational database itself making it easy to query the catalog if a user is authorized to do so.

(example of catalog on page 481 E&N)

Information in the catalog is very important of course since query processing makes use of this information extensively. Therefore more comprehensive and more accurate information is desirable but maintaining more comprehensive and more accurate information also introduces additional overheads and a balance must be found.

We Will learn more in detail in next lecture

Review Question (Tutorial)

Discuss there:

1. Access path selection - Selinger
2. Clustering and non-clustering page 636 Ullman

References

1. R. Ahad, K. V. Bapa Rao and D. McLeod (1989), "On Estimating the Cardinality of the Projection of a Database Relation", ACM TODS, Vol 14, No 1, pp 29-40, March 1989.
2. Bernstein, P.A. and D.W. Chiu (1981), "Using Semi-Joins to Solve Relational Queries", JACM, Vol. 28, No.1, pp. 25-40.
3. Blasgen, M. W. and Eswaran, K. P. (1977), "On the Evaluation of Queries in a Relational Data Base System", IBM Systems Journal,Vol. 16, No. 4,
4. Bratbergsengen, K. (1984), "Hashing Methods and Relational Algebra Operations", Proc. of the 10th Int. Conf. on VLDB, August 1984, pp.323-??
5. DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M. and Wood, D. (1984), "Implementation Techniques for Main Memory Database Systems", Proc. ACM SIGMOD, June 1984, pp. 1-8.
6. DeWitt, D. and Gerber, R. (1985), "Multiprocessor Hash-based Join Algorithms", Proc. VLDB, August 1985, pp.??
7. Gotlieb, L. R. (1975), "Computing Joins of Relations", ACM-SIGMOD, 1975, pp. 55-63.
8. Graefe, G. (1993), "Query Evaluation Techniques for Large Databases" ACM Computing Surveys, Vol 25, No 2, June 1993, pp 73-170.
9. Haerder, T. (1978), "Implementing a Generalized Access Path Structure for a Relational Database System", ACM-TODS, Vol. 3, No. 3, pp. 285-298.
10. Kim, W. (1980), "A New Way to Compute the Product and Join of Relations", ACM-SIGMOD, 1980, pp. 179-187.
11. Jarke, M. and Koch, J. (1984), "Query Optimization in Database Systems", ACM Computing Surveys, Vol. 16, No. 2, pp. 111-152.
12. Missikoff, M. (1982), "A Domain Based Internal Schema for Relational Database Machines", ACM-SIGMOD, 1982, pp. 215-224.
13. Sacco, G. M. and Yao, S. B. (1982), "Query Optimization in Distributed Data Base Systems", Advances in Computers, Vol. 21, pp. 225-273.
14. Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A. and Price, T. G. (1979), "Access Path Selection in a Relational Database Management System", ACM-SIGMOD, 1977, pp. 23-34.
15. Smith, J. M. and Chang, P. Y. (1975), "Optimizing the Performance of a Relational Algebra Database Interface", Comm. ACM, 21, 9, pp 568-579.
16. Talbot, S. (1984), "An Investigation into Logical Optimization of Relational Query Languages", The Comp. Journal, Vol. 27, No. 4, pp. 301-309.
17. Valduriez, P. and Gardarin, G. (1984), "Join and Semijoin Algorithms for Multiprocessor Database Machine", ACM-TODS, Vol 9, No. 1, March 1984, pp. 133-161.
18. Wong, E. and Youssefi (1976), "Decomposition - A Strategy for Query Processing", ACM TODS, 1, No.3, pp.
19. S. B. Yao (1979), "Optimization of Query Evaluation Algorithms", ACM TODS, Vol 4, No 2, June 1979, pp 133-155.
20. Bernstein , SIAM J Of Computing, 1981, pp. 751-771.
21. Bernstein, Info Systems, 1981, pp. 255-??

Notes:

LESSON 25
QUERY PROCESSING AND QUERY OPTIMISER PART-II

Hi! In this chapter I am going to discuss with you about Query Processing and Query Optimiser in more details.

Algorithms for Algebra Operations

The efficiency of query processing in a relational database system depends on the efficiency of the relational operators. Even the simplest operations can often be executed in several different ways and the costs of the different ways could well be quite different. Although the join is a frequently used and the most costly operator and therefore worthy of detailed study, we also discuss other operators to show that careful thought is needed in efficiently carrying out the simpler operators as well.

- Selection
- Projection
- Join
  - Nested Iteration
  - Using Indexes
  - The Sort Merge Method
  - Simple Hash Join Method
  - Grace Hash-Join Method
  - Hybrid Hash Join Method
  - Semi-Joins
- Aggregation

Selection

Let us consider the following simple query:

SELECT A
FROM R
WHERE p

[see page 641 Ullman II]

The above query may involve any of a number of types of predicates. The following list is presented by Selinger et al: [could have a query with specifying WHERE condition in different ways]

1. **attribute = value**
2. **attribute1 = attribute2**
3. **attribute > value**
4. **attribute between value1 and value2**
5. **attribute IN (list of values)**
6. **attribute IN subquery**
7. **predicate expression OR predicate expression**
8. **predicate expression AND predicate expression**

Even in the simple case of equality, two or three different approaches may be possible depending on how the relation has been stored. Traversing a file to find the information of interest is often called a file scan even if the whole file is not being scanned. For example, if the predicate involves an equality condition on a single attribute and there is an index on that attribute, it is most efficient to search that index and find the tuple where the attribute value is equal to the value given. That should be very efficient since it will only require accessing the index and then one block to access the tuple. Of course, it is possible that there is no index on the attribute of interest or the condition in the WHERE clause is not quite as simple as an equality condition on a single attribute. For example, the condition might be an inequality or specify a range. The index may still be useful if one exists but the usefulness would depend on the condition that is posed in the WHERE clause. In some situations it will be necessary to scan the whole relation R to find the tuples that satisfy the given condition. This may not be so expensive if the relation is not so large and the tuples are stored in packed form but could be very expensive if the relation is large and the tuples are stored such that each block has tuples from several different relations. Another possibility is of course that the relation R is stored as a hash file using the attribute of interest and then again one would be able to hash on the value specified and find the record very efficiently.

As noted above, often the condition may be a conjunction or disjunction of several different conditions i.e. it may be like $p_1 AND p_2$ or $p_1 OR p_2$. Sometime such conjunctive queries can be efficiently processed if there is a composite index based on the attributes that are involved in the two conditions but this is an exception rather than a rule. Often however, it is necessary to assess which one of the two or more conditions can be processed efficiently. Perhaps one of the conditions can be processed using an index. As a first step then, those tuples that satisfy the condition that involves the most efficient search (or perhaps that which retrieves the smallest number of tuples) are retrived and the remaining conditions are then tested on the tuples that are retrieved. Processing disjunctive queries of course requires somewhat different techniques since in this case we are looking at a union of all tuples that satisfy any one of the conditions and therefore each condition will need to be processed separately. It is therefore going to be of little concern which of the conditions is satisfied first since all must be satisfied independently of the other. Of course, if any one of the conditions requires a scan of the whole relation then we can test all the conditions during the scan and retrieve all tuples that satisfy any one or more conditions.

Projection

A projection would of course require a scan of the whole relation but if the projection includes a candidate key of the relation then no duplicate removal is necessary since each tuple in the projection is then guaranteed to be unique. Of course, more often the projection would not include any candidate key and may then have duplicates. Although many database systems do not remove duplicates unless the user specifies so,

duplicates may be removed by sorting the projected relation and then identifying the duplicates and eliminating them. It is also possible to use hashing which may be desirable if the relations are particularly large since hashing would hash identical tuples to the same bucket and would therefore only require sorting the relations in each bucket to find the duplicates if any.

Often of course one needs to compute a restriction and a join together. It is then often appropriate to compute the restriction first by using the best access paths available (e.g. an index).

Join

We assume that we wish to carry out an equi-join of two relations *R* and *S* that are to be joined on attributes a in R and b in S. Let the cardinality of R and S be m and n respectively. We do not count join output costs since these are identical for all methods. We assume $|R| \leq |S|$. We further assume that all restrictions and projections of R and S have already been carried out and neither R nor S is ordered or indexed unless explicitly noted.

Because of the importance of the join operator in relational database systems and the fact that the join operator is considerably more expensive than operators like selection and projection, a number of algorithms have been suggested for processing the join. The more commonly used algorithms are:

1. The Nested Scan Method
2. The Sort-Merge algorithm
3. Hashing algorithm (hashing no good if not equi-join?)
4. Variants of hashing
5. Semi-joins
6. Filters
7. Links
8. Indexes
9. More recently, the concept of join indices has been proposed by Valduriez (1987). Hashing methods are not good when the join is not an equi-join.

- Nested Iteration
- Using Indexes
- The Sort Merge Method
- Simple Hash Join Method
- Grace Hash-Join Method
- Hybrid Hash Join Method
- Semi-Joins

Nested Iteration

Before discussing the methods listed above, we briefly discuss the naive nested iteration method that accesses every pair of tuples and concatenates them if the equi-join condition (or for that matter, any other condition) is satisfied. The cost of the naive algorithm is O(mn) assuming that R and S both are not ordered. The cost obviously can be large when m and n are large.

We will assume that the relation R is the outer relation, that is, R is the relation whose tuples are retrieved first. The relation S is then the inner relation since in the nested iteration loop, tuples of S will only be retrieved when a tuple of R has been read. A predicate which related the join attributes is called the join predicate. [page 643->> Ullman II]

The algorithm may be written as:

for $i = 1$ to m do
access *ith* tuple of *R*;
for $j = 1$ to n do
access *jth* tuple of *S*;

compare ith tuple of *R* and the *jth* tuple of *S*;
if equi-join condition is satisfied
then concatenate and save;
end
end.

This method basically scans the outer relation (*R*) first and retrieves the first tuple. The entire inner relation S is then scanned and all the tuples of *S* that satisfy the join predicate with the first tuple of *R* are combined with that tuple of *R* and output as result. The process then continues with the next tuple of *R* until *R* is exhausted.

This has cost *(m + mn)* which is order*(mn)*. If the memory buffers can store two blocks, one from R and one from *S*, the cost will go down by a factor rs where r and s are the number of tuples per block in *R* and *S* respectively. The technique is sometimes called the nested block method. Some cost saving is achieved by reading the smaller relation in the outer block since this reduces *(m + mn)*. The cost of the method would of course be much higher if the relations are not stored in a packed form since then we might need to retrieve many more tuples.

[what about lot's of MM] Korth p 294

Efficiency of the nested iteration (or nested block iteration) would improve significantly if an index was available on one of the join attributes. If the average number of blocks of relation S accessed for each tuple of *R* was c then the cost of the join would be *(m + mc)* where $c \ll n$.

Using Indexes

The nested iteration method can be made more efficient if indexes are available on both join columns in the relations R and S.

Assume that we have available indexes on both join columns a and b in the relations R and S respectively. We may now scan both the indexes to determine whether a pair of tuples has the same value of the join attribute. If the value is the same, the tuple from R is selected and then all the tuples from S are selected that have the same join attribute value. This is done by scanning the index on the join attribute in S. The index on the join attribute in R is now scanned to check if there are more than the one tuple with the same value of the attribute. All the tuples of R that have the same join attribute value are then selected and combined with the tuples of S that have already been selected. The process then continues with the next value for which tuples are available in R and S.

Clearly this method requires substantial storage so that we may store all the attributes from *R* and *S* that have the same join attribute value.

The cost of the join when the indexes are used may be estimated as follows. Let the cost of reading the indexes be αN_1 and βN_2, then the total cost is $\alpha N_1 + \beta N_2 + N_1 + N_2$.

Cost savings by using indexes can be large enough to justify building an index when a join needs to be computed. page 296 Korth.

The Sort Merge Method

The nested scan technique is simple but involves matching each block of R with every block of S. This can be avoided if both relations were ordered on the join attribute. The sort-merge algorithm was introduced by Blasgen and Eswaran in 1977. It is a classical technique that has been the choice for joining relations that have no index on either of the two attributes.

This method involves sorting the relations R and S on the join attributes (if not already sorted), storing them as temporary lists and then scanning them block by block and merging those tuples that satisfy the join condition. The advantage of this scheme is that all of the inner relation (in the nested iteration) does not need to be read in for each tuple of the outer relation. This saving can be substantial if the outer relation is large.

Let the cost of sorting *R* and *S* be C_s and C_r and let the cost of reading the two relations in main memory be N_s and N_r respectively. The total cost of the join is then $C_r + C_s + N_r + N_s$.

If one or both the relations are already sorted on the join attribute then the cost of the join reduces.

The algorithm can be improved if we use Multiway Merge-Sort..(Ullman p654).

The cost of sorting is $n \log n$. Example?? Ullman page 653.

Simple Hash Join Method

This method involves building a hash table of the smaller relation R by hashing each tuple on its hash attribute. Since we have assumed that the relation R is too large to fit in the main memory, the hash table would in general not fit into the main memory. The hash table therefore must be built in stages. A number of addresses of the hash table are first selected such that the tuples hashed to those addresses can be stored in the main memory. The tuples of R that do not hash to these addresses are written back to the disk. Let these tuples be relation R'. Now the algorithm works as follows:

a. Scan relation R and hash each tuple on its join attribute. If the hashed value is equal to one of the addresses that are in the main memory, store the tuple in the hash table. Otherwise write the tuple back to disk in a new relation R'.

b. Scan the relation *S* and hash each tuple of *S* on its join attribute. One of the following three conditions must hold:

1. The hashed value is equal to one of the selected values, and one or more tuple of *R* with same attribute value exists. We combine the tuples of R that match with the tuple of *S* and output as the next tuples in the join.
2. The hashed value is equal to one of the selected values, but there is no tuple in *R* with same join attribute value. These tuple of *S* are rejected.
3. The hashed value is not equal to one of the selected values. These tuples are written back to disk as a new relation S'.

The above step continues till S is finished.

(c) Repeat steps (a) and (b) until either relation R' or S' or both are exhausted.

Grace Hash-Join Method

This method is a modification of the Simple Hash Join method in that the partitioning of R is completed before S is scanned and partitioning of S is completed before the joining phase. The method consists of the following three phases:

1. Partition *R* - Since *R* is assumed to be too large to fit in the main memory, a hash table for it cannot be built in the main memory. The first phase of the algorithm involves partitioning the relation into n buckets, each bucket corresponding to a hash table entry. The number of buckets n is chosen to be large enough so that each bucket will comfortably fit in the main memory.
2. Partition *S* - The second phase of the algorithm involves partitioning the relation S into the same number (*n*) of buckets, each bucket corresponding to a hash table entry. The same hashing function as for *R* is used.
3. Compute the Join - A bucket of *R* is read in and the corresponding bucket of *S* is read in. Matching tuples from the two buckets are combined and output as part of the join.

Hybrid Hash Join Method

The hybrid hash join algorithm is a modification of the Grace hash join method.

Semi-Joins

Aggregation

Aggregation is often found in queries given the frequency of requirements of finding an average, the maximum or how many times something happens. The functions supported in SQL are average, minimum, maximum, count, and sum. Aggregation can itself be of different types including aggregation that only requires one relation, for example finding the maximum mark in a subject, or it may involve a relation but require something like finding the number of students in each class. The latter aggregation would obviously require some grouping of the tuples in the relation before aggregation can be applied.

Review Question (Tutorial)

Discuss there:

1. Access path selection - Selinger
2. Clustering and non-clustering page 636 Ullman

References

1. R. Ahad, K. V. Bapa Rao and D. McLeod (1989), “On Estimating the Cardinality of the Projection of a Database Relation”, ACM TODS, Vol 14, No 1, pp 29-40, March 1989.

2. Bernstein, P.A. and D.W. Chiu (1981), “Using Semi-Joins to Solve Relational Queries”, JACM, Vol. 28, No.1, pp. 25-40.
3. Blasgen, M. W. and Eswaran, K. P. (1977), “On the Evaluation of Queries in a Relational Data Base System”, IBM Systems Journal,Vol. 16, No. 4,
4. Bratbergsengen, K. (1984), “Hashing Methods and Relational Algebra Operations”, Proc. of the 10th Int. Conf. on VLDB, August 1984, pp.323-??
5. DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M. and Wood, D. (1984), “Implementation Techniques for Main Memory Database Systems”, Proc. ACM SIGMOD, June 1984, pp. 1-8.
6. DeWitt, D. and Gerber, R. (1985), “Multiprocessor Hash-based Join Algorithms”, Proc. VLDB, August 1985, pp.??
7. Gotlieb, L. R. (1975), “Computing Joins of Relations”, ACM-SIGMOD, 1975, pp. 55-63.
8. Graefe, G. (1993), “Query Evaluation Techniques for Large Databases” ACM Computing Surveys, Vol 25, No 2, June 1993, pp 73-170.
9. Haerder, T. (1978), “Implementing a Generalized Access Path Structure for a Relational Database System”, ACM-TODS, Vol. 3, No. 3, pp. 285-298.
10. Kim, W. (1980), “A New Way to Compute the Product and Join of Relations”, ACM-SIGMOD, 1980, pp. 179-187.
11. Jarke, M. and Koch, J. (1984), “Query Optimization in Database Systems”, ACM Computing Surveys, Vol. 16, No. 2, pp. 111-152.
12. Missikoff, M. (1982), “A Domain Based Internal Schema for Relational Database Machines”, ACM-SIGMOD, 1982, pp. 215-224.
13. Sacco, G. M. and Yao, S. B. (1982), “Query Optimization in Distributed Data Base Systems”, Advances in Computers, Vol. 21, pp. 225-273.
14. Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A. and Price, T. G. (1979), “Access Path Selection in a Relational Database Management System”, ACM-SIGMOD, 1977, pp. 23-34.
15. Smith, J. M. and Chang, P. Y. (1975), “Optimizing the Performance of a Relational Algebra Database Interface”, Comm. ACM, 21, 9, pp 568-579.
16. Talbot, S. (1984), “An Investigation into Logical Optimization of Relational Query Languages”, The Comp. Journal, Vol. 27, No. 4, pp. 301-309.
17. Valduriez, P. and Gardarin, G. (1984), “Join and Semijoin Algorithms for Multiprocessor Database Machine”, ACM-TODS, Vol 9, No. 1, March 1984, pp. 133-161.
18. Wong, E. and Youssefi (1976), “Decomposition - A Strategy for Query Processing”, ACM TODS, 1, No.3, pp.
19. S. B. Yao (1979), “Optimization of Query Evaluation Algorithms”, ACM TODS, Vol 4, No 2, June 1979, pp 133-155.
20. Bernstein , SIAM J Of Computing, 1981, pp. 751-771.
21. Bernstein, Info Systems, 1981, pp. 255-??

Notes:

LESSON 26
LANGUAGE SUPPORT FOR OPTIMISER

Hi! In this chapter I am going to discuss with you about Language support for optimiser

I do the very best I know how-the very best I can;
and I mean to keep doing so until the end.

Abraham Lincoln

The Optimizer

This chapter discusses how the Oracle optimizer chooses how to execute SQL statements. It includes:

- What Is Optimization?
  - Execution Plans
  - Execution Order
- Cost-Based and Rule-Based Optimization
  - Optimizer Operations
  - Evaluation of Expressions and Conditions
  - Transforming and Optimizing Statements
  - Choosing an Optimization Approach and Goal
  - Choosing Access Paths
  - Optimizing Join Statements
  - Optimizing "Star" Queries

For more information on the Oracle optimizer, see Oracle8 Server Tuning.

What Is Optimization?

Optimization is the process of choosing the most efficient way to execute a SQL statement. This is an important step in the processing of any data manipulation language (DML) statement: SELECT, INSERT, UPDATE, or DELETE. Many different ways to execute a SQL statement often exist, for example, by varying the order in which tables or indexes are accessed. The procedure Oracle uses to execute a statement can greatly affect how quickly the statement executes.

A part of Oracle called the *optimizer* chooses what it believes to be the most efficient way. The optimizer evaluates a number of factors to select among alternative access paths. Sometimes the application designer, who has more information about a particular application's data than is available to the optimizer, can choose a more effective way to execute a SQL statement. The application designer can use hints in SQL statements to specify how the statement should be executed (see Oracle8 Server Tuning).

Note: The optimizer may not make the same decisions from one version of Oracle to the next. In more recent versions, the optimizer may make different decisions based on better, more sophisticated information available to it.

Execution Plans

To execute a DML statement, Oracle may have to perform many steps. Each of these steps either retrieves rows of data physically from the database or prepares them in some way for the user issuing the statement. The combination of the steps Oracle uses to execute a statement is called an *execution plan*.

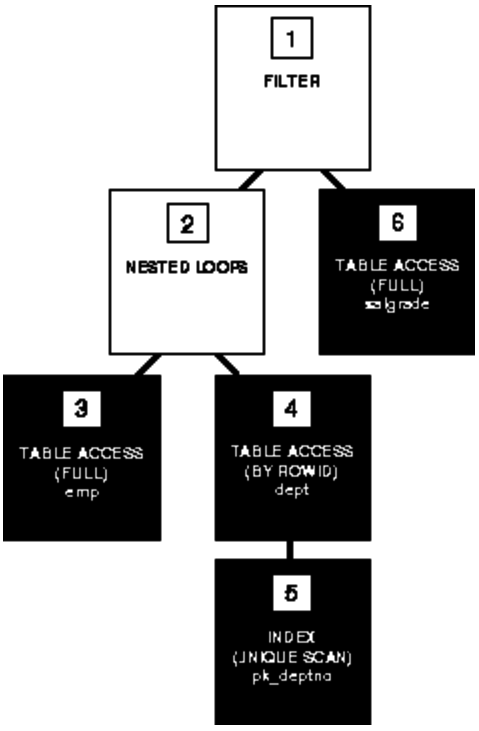
Sample Execution Plan

This example shows an execution plan for the following SQL statement, which selects the name, job, salary, and department name for all employees whose salaries do not fall into a recommended salary range:

```
SELECT ename, job, sal, dname
   FROM emp, dept
  WHERE emp.deptno = dept.deptno
    AND NOT EXISTS
     (SELECT *
        FROM salgrade
       WHERE emp.sal BETWEEN losal AND hisal);
```

Figure 19-1 shows a graphical representation of the execution plan.

Figure 19-1: An Execution Plan



Steps of Execution Plan

Each step of the execution plan returns a set of rows that either are used by the next step or, in the last step, are returned to the user or application issuing the SQL statement. A set of rows returned by a step is called a *row source*.

Figure 19-1 is a hierarchical diagram showing the flow of row sources from one step to another. The numbering of the steps reflects the order in which they are displayed in response to the

EXPLAIN PLAN command (described in the next section). This generally is **not** the order in which the steps are executed (see "Execution Order" on page 19-5).

Each step of the execution plan either retrieves rows from the database or accepts rows from one or more row sources as input:

- Steps indicated by the shaded boxes physically retrieve data from an object in the database. Such steps are called *access paths*:
  - Steps 3 and 6 read all the rows of the EMP and SALGRADE tables, respectively.
  - Step 5 looks up in the PK_DEPTNO index each DEPTNO value returned by Step 3. There it finds the ROWIDs of the associated rows in the DEPT table.
  - Step 4 retrieves from the DEPT table the rows whose ROWIDs were returned by Step 5.
- Steps indicated by the clear boxes operate on row sources:
  - Step 2 performs a nested loops operation, accepting row sources from Steps 3 and 4, joining each row from Step 3 source to its corresponding row in Step 4, and returning the resulting rows to Step 1.
  - Step 1 performs a filter operation. It accepts row sources from Steps 2 and 6, eliminates rows from Step 2 that have a corresponding row in Step 6, and returns the remaining rows from Step 2 to the user or application issuing the statement.

Access paths are discussed further in the section "Choosing Access Paths" on page 19-37. Methods by which Oracle joins row sources are discussed in "Join Operations" on page 19-55.

The EXPLAIN PLAN Command

You can examine the execution plan chosen by the optimizer for a SQL statement by using the EXPLAIN PLAN command. This command causes the optimizer to choose the execution plan and then inserts data describing the plan into a database table. The following is such a description for the statement examined in the previous section:

Id	Operation	Options	Object_Name
0	Select Statement		
1	Filter		
2	Nested Loops		
3	Table Access	Full	Emp
4	Table Access	By Rowid	Dept
5	Index	Unique Scan	Pk_Deptno
6	Table Access	Full	Salgrade

Each box in Figure 19-1 and each row in the output table corresponds to a single step in the execution plan. For each row in the listing, the value in the ID column is the value shown in the corresponding box in Figure 19-1.

You can obtain such a listing by using the EXPLAIN PLAN command and then querying the output table. For information on how to use this command and produce and interpret its output, see Oracle8 Server Tuning.

Execution Order

The steps of the execution plan are not performed in the order in which they are numbered. Rather, Oracle first performs the steps that appear as leaf nodes in the tree-structured graphical representation of the execution plan (Steps 3, 5, and 6 in Figure 19-1 on page 19-3). The rows returned by each step become the row sources of its parent step. Then Oracle performs the parent steps.

To execute the statement for Figure 19-1, for example, Oracle performs the steps in this order:

- First, Oracle performs Step 3, and returns the resulting rows, one by one, to Step 2.
- For each row returned by Step 3, Oracle performs these steps:
  - Oracle performs Step 5 and returns the resulting ROWID to Step 4.
  - Oracle performs Step 4 and returns the resulting row to Step 2.
  - Oracle performs Step 2, joining the single row from Step 3 with a single row from Step 4, and returning a single row to Step 1.
  - Oracle performs Step 6 and returns the resulting row, if any, to Step 1.
  - Oracle performs Step 1. If a row is not returned from Step 6, Oracle returns the row from Step 2 to the user issuing the SQL statement.

Note that Oracle performs Steps 5, 4, 2, 6, and 1 once for each row returned by Step 3. If a parent step requires only a single row from its child step before it can be executed, Oracle performs the parent step (and possibly the rest of the execution plan) as soon as a single row has been returned from the child step. If the parent of that parent step also can be activated by the return of a single row, then it is executed as well.

Thus the execution can cascade up the tree, possibly to encompass the rest of the execution plan. Oracle performs the parent step and all cascaded steps once for each row in turn retrieved by the child step. The parent steps that are triggered for each row returned by a child step include table accesses, index accesses, nested loops joins, and filters.

If a parent step requires all rows from its child step before it can be executed, Oracle cannot perform the parent step until all rows have been returned from the child step. Such parent steps include sorts, sort-merge joins, group functions, and aggregates.

Cost-Based and Rule-Based Optimization

To choose an execution plan for a SQL statement, the optimizer uses one of two approaches: cost-based or rule-based.

The Cost-Based Approach

Using the cost-based approach, the optimizer determines which execution plan is most efficient by considering available access paths and factoring in information based on statistics in the data dictionary for the schema objects (tables, clusters, or indexes) accessed by the statement. The cost-based approach also considers hints, or optimization suggestions placed in a Comment in the statement.

Conceptually, the cost-based approach consists of these steps:

1. The optimizer generates a set of potential execution plans for the statement based on its available access paths and hints.
2. The optimizer estimates the cost of each execution plan based on the data distribution and storage characteristics statistics for the tables, clusters, and indexes in the data dictionary.

The *cost* is an estimated value proportional to the expected resource use needed to execute the statement using the execution plan. The optimizer calculates the cost based on the estimated computer resources, including (but not limited to) I/O, CPU time, and memory, that are required to execute the statement using the plan.

Serial execution plans with greater costs take more time to execute than those with smaller costs. When using a parallel execution plan, however, resource use is not directly related to elapsed time.

3. The optimizer compares the costs of the execution plans and chooses the one with the smallest cost.

Goal of the Cost-Based Approach

By default, the goal of the cost-based approach is the best *throughput*, or minimal resource use necessary to process all rows accessed by the statement.

Oracle can also optimize a statement with the goal of best *response time*, or minimal resource use necessary to process the first row accessed by a SQL statement. For information on how the optimizer chooses an optimization approach and goal, see "Choosing an Optimization Approach and Goal" on page 19-34.

Note: For parallel execution, the optimizer can choose to minimize elapsed time at the expense of resource consumption. Use the initialization parameter OPTIMIZER_PERCENT_PARALLEL to specify how much the optimizer attempts to parallelize. See "Parallel Query Tuning" in Oracle8 Server Tuning for more information.

Statistics for the Cost-Based Approach

The cost-based approach uses statistics to estimate the cost of each execution plan. These statistics quantify the data distribution and storage characteristics of tables, columns, indexes, and partitions. You can generate these statistics using the ANALYZE command. The optimizer uses these statistics to estimate how much I/O, CPU time, and memory are required to execute a SQL statement using a particular execution plan.

You can view the statistics with these data dictionary views:

- USER_TABLES, ALL_TABLES, and DBA_TABLES
- USER_TAB_COLUMNS, ALL_TAB_COLUMNS, and DBA_TAB_COLUMNS
- USER_INDEXES, ALL_INDEXES, and DBA_INDEXES
- USER_CLUSTERS and DBA_CLUSTERS
- USER_TAB_PARTITIONS, ALL_TAB_PARTITIONS, and DBA_TAB_PARTITIONS
- USER_IND_PARTITIONS, ALL_IND_PARTITIONS, and DBA_IND_PARTITIONS
- USER_PART_COL_STATISTICS, ALL_PART_COL_STATISTICS, and DBA_PART_COL_STATISTICS

For information on these statistics, see the Oracle8 Server Reference Manual.

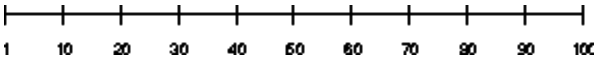
Histograms

Oracle's cost-based optimizer uses data value histograms to get accurate estimates of the distribution of column data. Histograms provide improved selectivity estimates in the presence of data skew, resulting in optimal execution plans with nonuniform data distributions. You generate histograms by using the ANALYZE command.

One of the fundamental capabilities of any cost-based optimizer is determining the selectivity of predicates that appear in queries. Selectivity estimates are used to decide when to use an index and the order in which to join tables. Most attribute domains (a table's columns) are *not* uniformly distributed. The Oracle cost-based optimizer uses height-balanced histograms on specified attributes to describe the distributions of nonuniform domains.

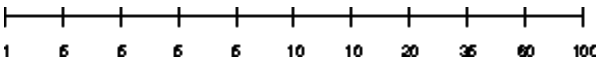
Histogram Examples

Consider a column C with values between 1 and 100 and a histogram with 10 buckets. If the data in C is uniformly distributed, this histogram would look like this, where the numbers are the endpoint values.



The number of rows in each bucket is one tenth the total number of rows in the table. Four-tenths of the rows have values between 60 and 100 in this example of uniform distribution.

If the data is not uniformly distributed, the histogram might look like this:



In this case, most of the rows have the value 5 for the column. In this example, only 1/10 of the rows have values between 60 and 100.

Height-Balanced Histograms

Oracle uses *height-balanced* histograms (as opposed to *width-balanced*).

- Width-balanced histograms divide the data into a fixed number of equal-width ranges and then count the number of values falling into each range.
- Height-balanced histograms place the same number of values into each range so that the endpoints of the range are determined by how many values are in that range.

For example, suppose that the values in a single column of a 1000-row table range between 1 and 100, and suppose that you want a 10-bucket histogram (ranges in a histogram are called *buckets*). In a width-balanced histogram, the buckets would be of equal width (1-10, 11-20, 21-30, and so on) and each bucket would count the number of rows that fall into that bucket's range. In a height-balanced histogram, each bucket has the same height (in this case 100 rows) and the endpoints for each bucket are determined by the density of the distinct values in the column.

Advantages of Height-Balanced Histograms

The advantage of the height-balanced approach is clear when the data is highly skewed. Suppose that 800 rows of a 1000-row table have the value 5, and the remaining 200 rows are evenly distributed between 1 and 100. A width-balanced histogram would have 820 rows in the bucket labeled 1-10 and approximately 20 rows in each of the other buckets. The height-based histogram would have one bucket labeled 1-5, seven buckets labeled 5-5, one bucket labeled 5-50, and one bucket labeled 50-100.

If you want to know how many rows in the table contain the value 5, it is apparent from the height-balanced histogram that approximately 80% of the rows contain this value. However, the width-balanced histogram does not provied a mechanism for differentiating between the value 5 and the value 6. You would compute only 8% of the rows contain the value 5 in a width-balanced histogram. Therefore height-based histograms are more appropriate for determining the selectivity of column values.

When to Use Histograms

For many users, it is appropriate to use the FOR ALL INDEXED COLUMNS option of the ANALYZE command to create histograms because indexed columns are typically the columns most often used in WHERE clauses.

You can view histograms with the following views:

- USER_HISTOGRAMS, ALL_HISTOGRAMS, and DBA_HISTOGRAMS
- USER_PART_HISTOGRAMS, ALL_PART_HISTOGRAMS, and DBA_PART_HISTOGRAMS
- TAB_COLUMNS

Histograms are useful only when they reflect the current data distribution of a given column. If the data distribution is not static, the histogram should be updated frequently. (The data need not be static as long as the *distribution* remains constant.)

Histograms can affect performance and should be used only when they substantially improve query plans. Histograms are *not* useful for columns with the following characteristics:

- All predicates on the column use bind variables.
- The column data is uniformly distributed.
- The column is not used in WHERE clauses of queries.
- The column is unique and is used only with equality predicates.

See Oracle8 Server Tuning for more information about histograms.

When to Use the Cost-Based Approach

In general, you should use the cost-based approach for all new applications; the rule-based approach is provided for applications that were written before cost-based optimization was available. Cost-based optimization can be used for both relational data and object types. The following features can only use cost-based optimization:

- partitioned tables
- partition views
- index-organized tables
- reverse key indexes
- bitmap indexes
- parallel query and parallel DML
- star transformation
- star join

For more information, see Oracle8 Server Tuning.

The Rule-Based Approach

Using the rule-based approach, the optimizer chooses an execution plan based on the access paths available and the ranks of these access paths (shown in Table 19-1 on page 19-39). You can use rule-based optimization to access both relational data and object types.

Oracle's ranking of the access paths is heuristic. If there is more than one way to execute a SQL statement, the rule-based approach always uses the operation with the lower rank. Usually, operations of lower rank execute faster than those associated with constructs of higher rank. For more information, see "Choosing Among Access Paths with the Rule-Based Approach" on page 19-53.

Optimizer Operations

For any SQL statement processed by Oracle, the optimizer does the following:

Evaluation of expressions and conditions	The optimizer first evaluates expressions and conditions containing constants as fully as possible. (See "Evaluation of Expressions and Conditions" on page 19-13.)
Statement transformation	For a complex statement involving, for example, correlated subqueries, the optimizer may transform the original statement into an equivalent join statement. (See "Transforming and Optimizing Statements" on page 19-17.)
View merging	For a SQL statement that accesses a view, the optimizer often merges the query in the statement with that in the view and then optimizes the result. (See "Optimizing Statements That Access Views" on page 19-22.)
choice of optimization approaches	The optimizer chooses either a cost-based or rule-based approach to optimization and determines the goal of optimization. (See "Choosing an Optimization Approach and Goal" on page 19-34.)
choice of access paths	For each table accessed by the statement, the optimizer chooses one or more of the available access paths to obtain the table's data. (See "Choosing Access Paths" on page 19-37.)
choice of join orders	For a join statement that joins more than two tables, the optimizer chooses which pair of tables is joined first, and then which table is joined to the result, on so on. (See "Optimizing Join Statements" on page 19-54.)
choice of join operations	For any join statement, the optimizer chooses an operation to use to perform the join. (See "Optimizing Join Statements" on page 19-54.)

Oracle optimizes these different types of SQL statements:

Simple statement	An INSERT, UPDATE, DELETE, or SELECT statement that involves only a single table.
Simple query	Another name for a SELECT statement.
Join	A query that selects data from more than one table. A join is characterized by multiple tables in the FROM clause. Oracle pairs the rows from these tables using the condition specified in the WHERE clause and returns the resulting rows. This condition is called the join condition and usually compares columns of all the joined tables.
equijoin	A join condition containing an equality operator.
nonequijoin	A join condition containing something other than an equality operator.
Outer join	A join condition using the outer join operator (+) with one or more columns of one of the tables. Oracle returns all rows that meet the join condition. Oracle also returns all rows from the table without the outer join operator for which there are no matching rows in the table with the outer join operator.
Cartesian product	A join with no join condition results in a Cartesian product, or a cross product. A Cartesian product is the set of all possible combinations of rows drawn one from each table. In other words, for a join of two tables, each row in one table is matched in turn with every row in the other. A Cartesian product for more than two tables is the result of pairing each row of one table with every row of the Cartesian product of the remaining tables. All other kinds of joins are subsets of Cartesian products effectively created by deriving the Cartesian product and then excluding rows that fail the join condition.
Complex statement	An INSERT, UPDATE, DELETE, or SELECT statement that contains a subquery, which is a form of the SELECT statement within another statement that produces a set of values for further processing within the statement. The outer portion of the complex statement that contains a subquery is called the *parent statement.*
Compound query	A query that uses set operators (UNION, UNION ALL, INTERSECT, or MINUS) to combine two or more simple or complex statements. Each simple or complex statement in a compound query is called a *component query*.
Statement accessing views	Simple, join, complex, or compound statement that accesses one or more views as well as tables.
Distributed statement	A statement that accesses data on a remote database.

Evaluation of Expressions and Conditions

The optimizer fully evaluates expressions whenever possible and translates certain syntactic constructs into equivalent constructs. The reason for this is either that Oracle can more quickly evaluate the resulting expression than the original expression, or that the original expression is merely a syntactic equivalent of the resulting expression. Different SQL constructs can sometimes operate identically (for example, = ANY (*subquery*) and IN (*subquery*)); Oracle maps these to a single construct.

Constants

Computation of constants is performed only once, when the statement is optimized, rather than each time the statement is executed.

Consider these conditions that test for monthly salaries greater than 2000:

sal > 24000/12

sal > 2000

sal*12 > 24000

If a SQL statement contains the first condition, the optimizer simplifies it into the second condition.

Note that the optimizer does not simplify expressions across comparison operators: in the examples above, the optimizer does not simplify the third expression into the second. For this reason, application developers should write conditions that compare columns with constants whenever possible, rather than conditions with expressions involving columns.

LIKE Operator

The optimizer simplifies conditions that use the LIKE comparison operator to compare an expression with no wildcard characters into an equivalent condition that uses an equality operator instead. For example, the optimizer simplifies the first condition below into the second:

ename LIKE 'SMITH'

ename = 'SMITH'

The optimizer can simplify these expressions only when the comparison involves variable-length datatypes. For example, if ENAME was of type CHAR(10), the optimizer cannot transform the LIKE operation into an equality operation due to the comparison semantics of fixed-length datatypes.

IN Operator

The optimizer expands a condition that uses the IN comparison operator to an equivalent condition that uses equality comparison operators and OR logical operators. For example, the optimizer expands the first condition below into the second:

ename IN ('SMITH', 'KING', 'JONES')

ename = 'SMITH' OR ename = 'KING' OR ename = 'JONES'

ANY or SOME Operator

The optimizer expands a condition that uses the ANY or SOME comparison operator followed by a parenthesized list of values into an equivalent condition that uses equality comparison operators and OR logical operators. For example, the optimizer expands the first condition below into the second:

sal > ANY (:first_sal, :second_sal)

sal > :first_sal OR sal > :second_sal

The optimizer transforms a condition that uses the ANY or SOME operator followed by a subquery into a condition containing the EXISTS operator and a correlated subquery. For example, the optimizer transforms the first condition below into the second:

```
x > ANY (SELECT sal
           FROM emp
           WHERE job = 'ANALYST')
EXISTS (SELECT sal
          FROM emp
          WHERE job = 'ANALYST'
            AND x > sal)
```

ALL Operator

The optimizer expands a condition that uses the ALL comparison operator followed by a parenthesized list of values into an equivalent condition that uses equality comparison operators and AND logical operators. For example, the optimizer expands the first condition below into the second:

sal > ALL (:first_sal, :second_sal)

sal > :first_sal AND sal > :second_sal

The optimizer transforms a condition that uses the ALL comparison operator followed by a subquery into an equivalent condition that uses the ANY comparison operator and a complementary comparison operator. For example, the optimizer transforms the first condition below into the second:

```
x > ALL (SELECT sal
           FROM emp
           WHERE deptno = 10)
NOT (x <= ANY (SELECT sal
                 FROM emp
                 WHERE deptno = 10) )
```

The optimizer then transforms the second query into the following query using the rule for transforming conditions with the ANY comparison operator followed by a correlated subquery:

```
NOT EXISTS (SELECT sal
              FROM emp
              WHERE deptno = 10
                AND x <= sal)
```

BETWEEN Operator

The optimizer always replaces a condition that uses the BETWEEN comparison operator with an equivalent condition that uses the >= and <= comparison operators. For example,

the optimizer replaces the first condition below with the second:

sal BETWEEN 2000 AND 3000

sal >= 2000 AND sal <= 3000

NOT Operator

The optimizer simplifies a condition to eliminate the NOT logical operator. The simplification involves removing the NOT logical operator and replacing a comparison operator with its opposite comparison operator. For example, the optimizer simplifies the first condition below into the second one:

NOT deptno = (SELECT deptno FROM emp WHERE ename = 'TAYLOR')

deptno <> (SELECT deptno FROM emp WHERE ename = 'TAYLOR')

Often a condition containing the NOT logical operator can be written many different ways. The optimizer attempts to transform such a condition so that the subconditions negated by NOTs are as simple as possible, even if the resulting condition contains more NOTs. For example, the optimizer simplifies the first condition below into the second and then into the third.

NOT (sal < 1000 OR comm IS NULL)

NOT sal < 1000 AND comm IS NOT NULL

sal >= 1000 AND comm IS NOT NULL

Transitivity

If two conditions in the WHERE clause involve a common column, the optimizer can sometimes infer a third condition using the transitivity principle. The optimizer can then use the inferred condition to optimize the statement. The inferred condition could potentially make available an index access path that was not made available by the original conditions.

Note: Transitivity is used only by the cost-based approach.

Imagine a WHERE clause containing two conditions of these forms:

WHERE column1 comp_oper constant
AND column1 = column2

In this case, the optimizer infers the condition:

column2 comp_oper constant

where:

comp_oper	is any of the comparison operators =, !=, ^=, <, <>, >, <=, or >=.
constant	is any constant expression involving operators, SQL functions, literals, bind variables, and correlation variables.

Example: Consider this query in which the WHERE clause contains two conditions, each or which uses the EMP.DEPTNO column:

SELECT *
FROM emp, dept
WHERE emp.deptno = 20
AND emp.deptno = dept.deptno;

Using transitivity, the optimizer infers this condition:

dept.deptno = 20

If an index exists on the DEPT.DEPTNO column, this condition makes available access paths using that index.

Note: The optimizer only infers conditions that relate columns to constant expressions, rather than columns to other columns. Imagine a WHERE clause containing two conditions of these forms:

WHERE column1 comp_oper column3
AND column1 = column2

In this case, the optimizer does not infer this condition:

column2 comp_oper column3

Transforming and Optimizing Statements

SQL is a very flexible query language; there are often many statements you could formulate to achieve the same goal. Sometimes the optimizer transforms one such statement into another that achieves the same goal if the second statement can be executed more efficiently.

This section discusses the following topics:

- Transforming ORs into Compound Queries
- Transforming Complex Statements into Join Statements
- Optimizing Statements That Access Views
- Optimizing Compound Queries
- Optimizing Distributed Statements

For additional information about optimizing statements, see "Optimizing Join Statements" on page 19-54 and "Optimizing "Star" Queries" on page 19-63.

Transforming ORs into Compound Queries

If a query contains a WHERE clause with multiple conditions combined with OR operators, the optimizer transforms it into an equivalent compound query that uses the UNION ALL set operator if this makes it execute more efficiently:

- If each condition individually makes an index access path available, the optimizer can make the transformation. The optimizer then chooses an execution plan for the resulting statement that accesses the table multiple times using the different indexes and then puts the results together.
- If any condition requires a full table scan because it does not make an index available, the optimizer does not transform the statement. The optimizer chooses a full table scan to execute the statement, and Oracle tests each row in the table to determine whether it satisfies any of the conditions.
- For statements that use the cost-based approach, the optimizer may use statistics to determine whether to make the transformation by estimating and then comparing the

costs of executing the original statement versus the resulting statement.

- The cost-based optimizer does not use the OR transformation for in lists or ORs on the same column; instead, it uses the inlist iterator operator. For more information, see Oracle8 Server Tuning.

For information on access paths and how indexes make them available, see Table 19-1 on page 19-39 and the sections that follow it.

Example: Consider this query with a WHERE clause that contains two conditions combined with an OR operator:

```
SELECT *
FROM emp
WHERE job = 'CLERK'
OR deptno = 10;
```

If there are indexes on both the JOB and DEPTNO columns, the optimizer may transform this query into the equivalent query below:

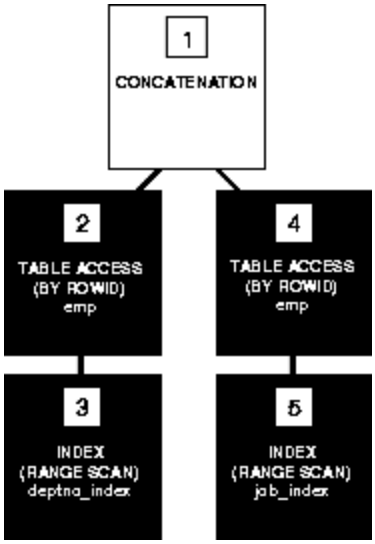
```
SELECT *
FROM emp
WHERE job = 'CLERK'
UNION ALL
SELECT *
FROM emp
WHERE deptno = 10
AND job <> 'CLERK';
```

If you are using the cost-based approach, the optimizer compares the cost of executing the original query using a full table scan with that of executing the resulting query when deciding whether to make the transformation.

If you are using the rule-based approach, the optimizer makes this UNION ALL transformation because each component query of the resulting compound query can be executed using an index. The rule-based approach assumes that executing the compound query using two index scans is faster than executing the original query using a full table scan.

The execution plan for the transformed statement might look like this:

Figure 19-2: Execution Plan for a Transformed Query Containing OR



To execute the transformed query, Oracle performs the following steps:

- Steps 3 and 5 scan the indexes on the JOB and DEPTNO columns using the conditions of the component queries. These steps obtain ROWIDs of the rows that satisfy the component queries.
- Steps 2 and 4 use the ROWIDs from Steps 3 and 5 to locate the rows that satisfy each component query.
- Step 1 puts together the row sources returned by Steps 2 and 4.

If either of the JOB or DEPTNO columns is not indexed, the optimizer does not even consider the transformation, because the resulting compound query would require a full table scan to execute one of its component queries. Executing the compound query with a full table scan in addition to an index scan could not possibly be faster than executing the original query with a full table scan.

Example: Consider this query and assume that there is an index on the ENAME column only:

```
SELECT *

 FROM emp

 WHERE ename = 'SMITH'

  OR sal > comm;
```

Transforming the query above would result in the compound query below:

```
SELECT *

 FROM emp

 WHERE ename = 'SMITH'

UNION ALL

SELECT *

 FROM emp

 WHERE sal > comm;
```

Since the condition in the WHERE clause of the second component query (SAL > COMM) does not make an index available, the compound query requires a full table scan. For this reason, the optimizer does not make the transformation and it chooses a full table scan to execute the original statement.

Transforming Complex Statements into Join Statements

To optimize a complex statement, the optimizer chooses one of these alternatives:

- Transform the complex statement into an equivalent join statement and then optimize the join statement.
- Optimize the complex statement as is.

The optimizer transforms a complex statement into a join statement whenever the resulting join statement is guaranteed to return exactly the same rows as the complex statement. This transformation allows Oracle to execute the statement by taking advantage of join optimization techniques described in "Optimizing Join Statements" on page 19-54.

Consider this complex statement that selects all rows from the ACCOUNTS table whose owners appear in the CUSTOMERS table:

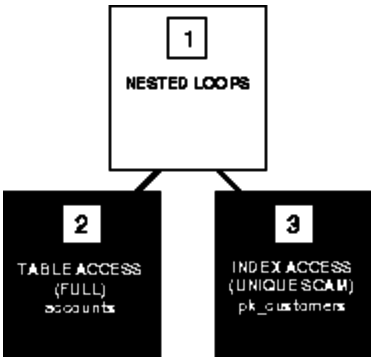
```
SELECT *

  FROM accounts

  WHERE custno IN

    (SELECT custno FROM customers);
```

If the CUSTNO column of the CUSTOMERS table is a primary key or has a UNIQUE constraint, the optimizer can transform the complex query into this join statement that is guaranteed to return the same data:

```
SELECT accounts.*

  FROM accounts, customers

  WHERE accounts.custno = customers.custno;
```

The execution plan for this statement might look like Figure 19-3.

Figure 19-3: Execution Plan for a Nested Loops Join



To execute this statement, Oracle performs a nested-loops join operation. For information on nested loops joins, see "Join Operations" on page 19-55.

If the optimizer cannot transform a complex statement into a join statement, the optimizer chooses execution plans for the parent statement and the subquery as though they were separate statements. Oracle then executes the subquery and uses the rows it returns to execute the parent query.

Consider this complex statement that returns all rows from the ACCOUNTS table that have balances greater than the average account balance:

```
SELECT *

  FROM accounts

  WHERE accounts.balance >

    (SELECT AVG(balance) FROM accounts);
```

No join statement can perform the function of this statement, so the optimizer does not transform the statement. Note that complex queries whose subqueries contain group functions such as AVG cannot be transformed into join statements.

Optimizing Statements That Access Views

To optimize a statement that accesses a view, the optimizer chooses one of these alternatives:

- Transform the statement into an equivalent statement that accesses the view's base tables.
- Issue the view's query, collecting all the returned rows, and then access this set of rows with the original statement as though it were a table.

Accessing the View's Base Table

To transform a statement that accesses a view into an equivalent statement that accesses the view's base tables, the optimizer can use one of these techniques:

- Merge the view's query into the accessing statement.
- Merge the accessing statement into the view's query.

The optimizer then optimizes the resulting statement.

To merge the view's query into the accessing statement, the optimizer replaces the name of the view with the name of its base table in the accessing statement and adds the condition of the view's query's WHERE clause to the accessing statement's WHERE clause.

Example: Consider this view of all employees who work in department 10:

```
CREATE VIEW emp_10
  AS SELECT empno,ename, job, mgr, hiredate, sal, comm,
deptno
FROM emp
WHERE deptno = 10;
```

Consider this query that accesses the view. The query selects the IDs greater than 7800 of employees who work in department 10:

```
SELECT empno
FROM emp_10
WHERE empno > 7800;
```

The optimizer transforms the query into the following query that accesses the view's base table:

```
SELECT empno
FROM emp
WHERE deptno = 10
AND empno > 7800;
```

If there are indexes on the DEPTNO or EMPNO columns, the resulting WHERE clause makes them available.

The optimizer cannot always merge the view's query into the accessing statement. Such a transformation is not possible if the view's query contains

- set operators (UNION, UNION ALL, INTERSECT, MINUS)

- a GROUP BY clause
- a CONNECT BY clause
- a DISTINCT operator in the select list
- group functions (AVG, COUNT, MAX, MIN, SUM) in the select list

To optimize statements that access such views, the optimizer can merge the statement into the view's query.

Example: Consider the TWO_EMP_TABLES view, which is the union of two employee tables. The view is defined with a compound query that uses the UNION set operator:

```
CREATE VIEW two_emp_tables
(empno, ename, job, mgr, hiredate, sal, comm, deptno) AS
SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
FROM emp1
UNION
 SELECT empno, ename, job, mgr, hiredate, sal, comm,
deptno
FROM emp2;
```

Consider this query that accesses the view. The query selects the IDs and names of all employees in either table who work in department 20:

```
SELECT empno, ename
FROM two_emp_tables
WHERE deptno = 20;
```

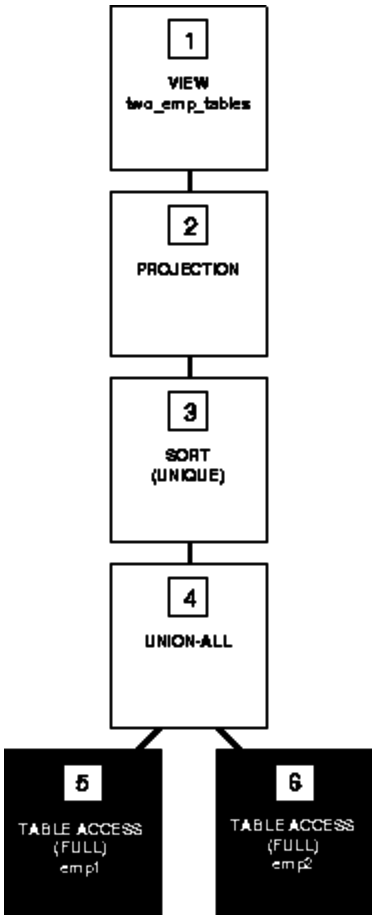
Since the view is defined as a compound query, the optimizer cannot merge the view query into the accessing query. Instead, the optimizer transforms the query by adding its WHERE clause condition into the compound query. The resulting statement looks like this:

```
SELECT empno, ename FROM emp1 WHERE deptno = 20
UNION
SELECT empno, ename FROM emp2 WHERE deptno = 20;
```

If there is an index on the DEPTNO column, the resulting WHERE clauses make it available.

Figure 19-4, "Accessing a View Defined with the UNION Set Operator", shows the execution plan of the resulting statement.

Figure 19-4: Accessing a View Defined with the UNION Set Operator



To execute this statement, Oracle performs these steps:

- Steps 5 and 6 perform full scans of the EMP1 and EMP2 tables.
- Step 4 performs a UNION-ALL operation returning all rows returned by either Step 5 or Step 6, including all copies of duplicates.
- Step 3 sorts the result of Step 4, eliminating duplicate rows.
- Step 2 extracts the desired columns from the result of Step 3.
- Step 1 indicates that the view's query was not merged into the accessing query.

Example: Consider the view EMP_GROUP_BY_DEPTNO, which contains the department number, average salary, minimum salary, and maximum salary of all departments that have employees:

```
CREATE VIEW emp_group_by_deptno
AS SELECT deptno,
AVG(sal) avg_sal,
MIN(sal) min_sal,
MAX(sal) max_sal
FROM emp
GROUP BY deptno;
```

Consider this query, which selects the average, minimum, and maximum salaries of department 10 from the EMP_GROUP_BY_DEPTNO view:

```
SELECT *
FROM emp_group_by_deptno
WHERE deptno = 10;
```

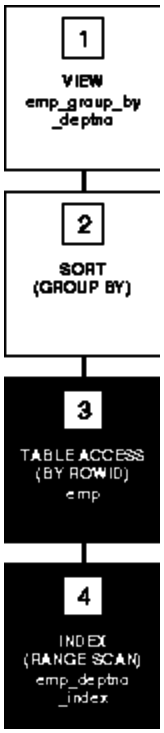
The optimizer transforms the statement by adding its WHERE clause condition into the view's query. The resulting statement looks like this:

```
SELECT deptno,

  AVG(sal) avg_sal,

  MIN(sal) min_sal,

  MAX(sal) max_sal,

 FROM emp

 WHERE deptno = 10

 GROUP BY deptno;
```

If there is an index on the DEPTNO column, the resulting WHERE clause makes it available.

Figure 19-5, "Accessing a View Defined with a GROUP BY Clause", shows the execution plan for the resulting statement. The execution plan uses an index on the DEPTNO column.

Figure 19-5: Accessing a View Defined with a GROUP BY Clause



To execute this statement, Oracle performs these operations:

- Step 4 performs a range scan on the index EMP_DEPTNO_INDEX (an index on the DEPTNO column of the EMP table) to retrieve the ROWIDs of all rows in the EMP table with a DEPTNO value of 10.
- Step 3 accesses the EMP table using the ROWIDs retrieved by Step 4.
- Step 2 sorts the rows returned by Step 3 to calculate the average, minimum, and maximum SAL values.
- Step 1 indicates that the view's query was not merged into the accessing query.

Example: Consider a query that accesses the EMP_GROUP_BY_DEPTNO view defined in the previous example. This query derives the averages for the average department salary, the minimum department salary, and the maximum department salary from the employee table:

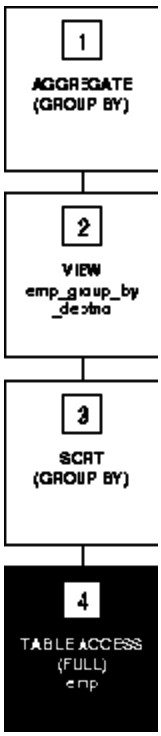
```
SELECT AVG(avg_sal), AVG(min_sal), AVG(max_sal)

  FROM emp_group_by_deptno;
```

The optimizer transforms this statement by applying the AVG group function to the select list of the view's query:

```
SELECT AVG(AVG(sal)), AVG(MIN(sal)), AVG(MAX(sal))

  FROM emp

  GROUP BY deptno;
```

Figure 19-6 shows the execution plan of the resulting statement.

Figure 19-6: Applying Group Functions to a View Defined with GROUP BY Clause



To execute this statement, Oracle performs these operations:

- Step 4 performs a full scan of the EMP table.
- Step 3 sorts the rows returned by Step 4 into groups based on their DEPTNO values and calculates the average, minimum, and maximum SAL value of each group.
- Step 2 indicates that the view's query was not merged into the accessing query.
- Step 1 calculates the averages of the values returned by Step 2.

Optimizing Other Statements That Access Views

The optimizer cannot transform all statements that access views into equivalent statements that access base table(s). To execute a statement that cannot be transformed, Oracle issues the view's query, collects the resulting set of rows, and then accesses this set of rows with the original statement as though it were a table.

Example: Consider the EMP_GROUP_BY_DEPTNO view defined in the previous section:

```
CREATE VIEW emp_group_by_deptno

 AS SELECT deptno,

     AVG(sal) avg_sal,

     MIN(sal) min_sal,

     MAX(sal) max_sal

   FROM emp

   GROUP BY deptno;
```

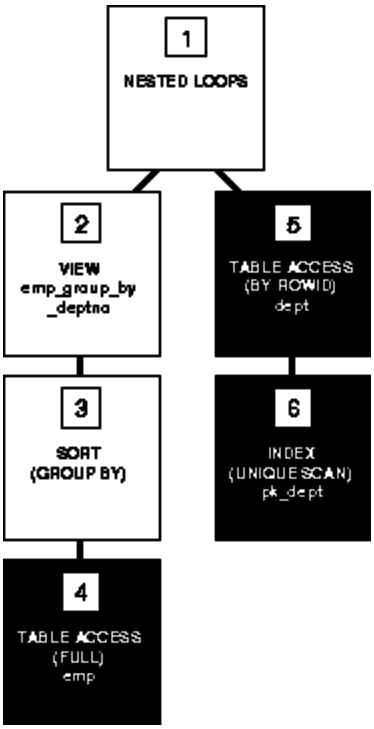
Consider this query, which accesses the view. The query joins the average, minimum, and maximum salaries from each department represented in this view and to the name and location of the department in the DEPT table:

```
SELECT emp_group_by_deptno.deptno, avg_sal, min_sal,

 max_sal, dname, loc

   FROM emp_group_by_deptno, dept

   WHERE emp_group_by_deptno.deptno = dept.deptno;
```

Since there is no equivalent statement that accesses only base tables, the optimizer cannot transform this statement. Instead, the optimizer chooses an execution plan that issues the view's query and then uses the resulting set of rows as it would the rows resulting from a table access.

Figure 19-7, "Joining a View Defined with a Group BY Clause to a Table", shows the execution plan for this statement. For more information on how Oracle performs a nested loops join operation, see "Join Operations" on page 19-55.

Figure 19-7: Joining a View Defined with a Group BY Clause to a Table



To execute this statement, Oracle performs these operations:

- Step 4 performs a full scan of the EMP table.
- Step 3 sorts the results of Step 4 and calculates the average, minimum, and maximum SAL values selected by the query for the EMP_GROUP_BY_DEPTNO view.
- Step 2 used the data from the previous two steps for a view.
- For each row returned by Step 2, Step 6 uses the DEPTNO value to perform a unique scan of the PK_DEPT index.
- Step 5 uses each ROWID returned by Step 6 to locate the row in the DEPTNO table with the matching DEPTNO value.
- Oracle combines each row returned by Step 2 with the matching row returned by Step 5 and returns the result.

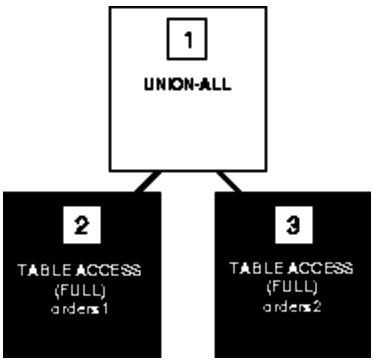
Optimizing Compound Queries

To choose the execution plan for a compound query, the optimizer chooses an execution plan for each of its component queries and then combines the resulting row sources with the union, intersection, or minus operation, depending on the set operator used in the compound query.

Figure 19-8, "Compound Query with UNION ALL Set Operator", shows the execution plan for this statement, which uses the UNION ALL operator to select all occurrences of all parts in either the ORDERS1 table or the ORDERS2 table:

```
SELECT part FROM orders1

UNION ALL

SELECT part FROM orders2;
```

Figure 19-8: Compound Query with UNION ALL Set Operator



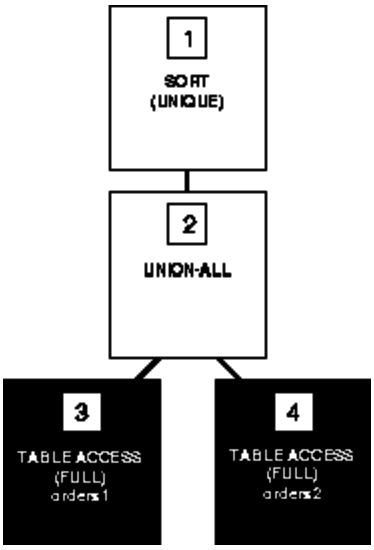
To execute this statement, Oracle performs these steps:

- Steps 2 and 3 perform full table scans on the ORDERS1 and ORDERS2 tables.
- Step 1 performs a UNION-ALL operation returning all rows that are returned by either Step 2 or Step 3 including all copies of duplicates.

Figure 19-9, "Compound Query with UNION Set Operator", shows the execution plan for the following statement, which uses the UNION operator to select all parts that appear in either the ORDERS1 or ORDERS2 table:

```
SELECT part FROM orders1

UNION

SELECT part FROM orders2;
```

Figure 19-9: Compound Query with UNION Set Operator

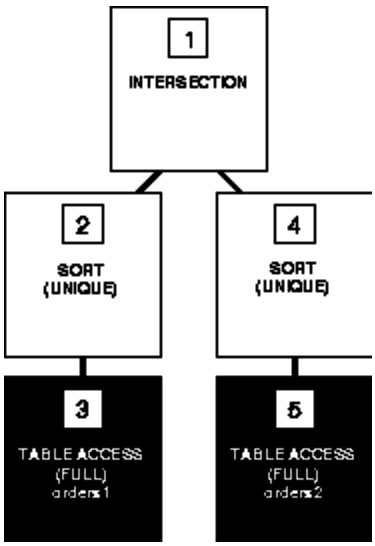


This execution plan is identical to the one for the UNION-ALL operator shown in Figure 19-8 "Compound Query with UNION ALL Set Operator", except that in this case Oracle uses the SORT operation to eliminate the duplicates returned by the UNION-ALL operation.

Figure 19-10, "Compound Query with INTERSECT Set Operator", shows the execution plan for this statement, which uses the INTERSECT operator to select only those parts that appear in both the ORDERS1 and ORDERS2 tables:

```
SELECT part FROM orders1

INTERSECT

SELECT part FROM orders2;
```

Figure 19-10: Compound Query with INTERSECT Set Operator



To execute this statement, Oracle performs these steps:

- Steps 3 and 5 perform full table scans of the ORDERS1 and ORDERS2 tables.
- Steps 2 and 4 sort the results of Steps 3 and 5, eliminating duplicates in each row source.
- Step 1 performs an INTERSECTION operation that returns only rows that are returned by both Steps 2 and 4.

Optimizing Distributed Statements

The optimizer chooses execution plans for SQL statements that access data on remote databases in much the same way it chooses executions for statements that access only local data:

- If all the tables accessed by a SQL statement are collocated on the same remote database, Oracle sends the SQL statement to that remote database. The remote Oracle instance executes the statement and sends only the results back to the local database.
- If a SQL statement accesses tables that are located on different databases, Oracle decomposes the statement into individual fragments, each of which accesses tables on a single database. Oracle then sends each fragment to the database that it accesses. The remote Oracle instance for

each of these databases executes its fragment and returns the results to the local database, where the local Oracle instance may perform any additional processing the statement requires.

When choosing a cost-based execution plan for a distributed statement, the optimizer considers the available indexes on remote databases just as it does indexes on the local database. The optimizer also considers statistics on remote databases for cost-based optimization. Furthermore, the optimizer considers the location of data when estimating the cost of accessing it. For example, a full scan of a remote table has a greater estimated cost than a full scan of an identical local table.

For a rule-based execution plan, the optimizer does not consider indexes on remote tables.

Choosing an Optimization Approach and Goal

The optimizer's behavior when choosing an optimization approach and goal for a SQL statement is affected by these factors:

- the OPTIMIZER_MODE initialization parameter
- statistics in the data dictionary
- the OPTIMIZER_GOAL parameter of the ALTER SESSION command
- hints (comments) in the statement

The OPTIMIZER_MODE Initialization Parameter

The OPTIMIZER_MODE initialization parameter establishes the default behavior for choosing an optimization approach for the instance. This parameter can have these values:

CHOOSE	The optimizer chooses between a cost-based approach and a rule-based approach based on whether the statistics are available for the cost-based Approach. If the data dictionary contains statistics for at least one of the accessed tables, the optimizer uses a cost-based approach and optimizes with a goal of best throughput. If the data dictionary contains no statistics for any of the accessed tables, the optimizer uses a rule-based approach. This is the default value for the parameter.
ALL_ROWS	The optimizer uses a cost-based approach for all SQL statements in the session regardless of the presence of statistics and optimizes with a goal of best throughput (minimum resource use to complete the entire statement).
FIRST_ROWS	The optimizer uses a cost-based approach for all SQL statements in the session regardless of the presence of statistics and optimizes with a goal of best response time (minimum resource use to return the first row of the result set).
RULE	The optimizer chooses a rule-based approach for all SQL statements issued to the instance regardless of the presence of statistics.

If the optimizer uses the cost-based approach for a SQL statement and some tables accessed by the statement have no statistics, the optimizer uses internal information (such as the number of data blocks allocated to these tables) to estimate other statistics for these tables.

Statistics in the Data Dictionary

Oracle stores statistics about columns, tables, clusters, indexes, and partitions in the data dictionary for use by the cost-based optimizer (see "Statistics for the Cost-Based Approach" on page 19-7). Two options of the ANALYZE command generate statistics:

- COMPUTE STATISTICS generates exact statistics.
- ESTIMATE STATISTICS generates estimations by sampling the data.

For more information, see Oracle8 Server Tuning.

The OPTIMIZER_GOAL Parameter of the ALTER SESSION Command

The OPTIMIZER_GOAL parameter of the ALTER SESSION command can override the optimization approach and goal established by the OPTIMIZER_MODE initialization parameter for an individual session.

The value of this parameter affects the optimization of SQL statements issued by stored procedures and functions called during the session, but it does not affect the optimization of recursive SQL statements that Oracle issues during the session. The optimization approach for recursive SQL statements is affected only by the value of the OPTIMIZER_MODE initialization parameter.

The OPTIMIZER_GOAL parameter can have these values:

CHOOSE	The optimizer chooses between a cost-based approach and a rule-based approach based on whether statistics are available for the cost-based approach. If the data dictionary contains statistics for at least one of the accessed tables, the optimizer uses a cost-based approach and optimizes with a goal of best throughput. If the data dictionary contains no statistics for any of the accessed tables, the optimizer uses a rule-based approach.
ALL_ROWS	The optimizer uses a cost-based approach for all SQL statements in the session regardless of the presence of statistics and optimizes with a goal of best throughput (minimum resource use to complete the entire statement).
FIRST_ROWS	The optimizer uses a cost-based approach for all SQL statements in the session regardless of the presence of statistics and optimizes with a goal of best response time (minimum resource use to return the first row of the result set).
RULE	The optimizer chooses a rule-based approach for all SQL statements issued to the instance regardless of the presence of statistics.

The FIRST_ROWS, ALL_ROWS, and RULE Hints

The FIRST_ROWS, ALL_ROWS, CHOOSE, and RULE hints can override the effects of both the OPTIMIZER_MODE initialization parameter and the OPTIMIZER_GOAL parameter of the ALTER SESSION command for an individual SQL statement. For information on hints, see Oracle8 Server Tuning.

Choosing Access Paths

One of the most important choices the optimizer makes when formulating an execution plan is how to retrieve data from the database. For any row in any table accessed by a SQL statement, there may be many access paths by which that row can be located and retrieved. The optimizer chooses one of them.

This section discusses:

- the basic methods by which Oracle can access data
- each access path and when it is available to the optimizer
- how the optimizer chooses among available access paths

Access Methods

This section describes basic methods by which Oracle can access data.

Full Table Scans

A full table scan retrieves rows from a table. To perform a full table scan, Oracle reads all rows in the table, examining each row to determine whether it satisfies the statement's WHERE clause. Oracle reads every data block allocated to the table sequentially, so a full table scan can be performed very efficiently using multiblock reads. Oracle reads each data block only once.

Table Access by ROWID

A table access by ROWID also retrieves rows from a table. The ROWID of a row specifies the datafile and data block containing the row and the location of the row in that block. Locating a row by its ROWID is the fastest way for Oracle to find a single row.

To access a table by ROWID, Oracle first obtains the ROWIDs of the selected rows, either from the statement's WHERE clause or through an index scan of one or more of the table's indexes. Oracle then locates each selected row in the table based on its ROWID.

Cluster Scans

From a table stored in an indexed cluster, a cluster scan retrieves rows that have the same cluster key value. In an indexed cluster, all rows with the same cluster key value are stored in the same data blocks. To perform a cluster scan, Oracle first obtains the ROWID of one of the selected rows by scanning the cluster index. Oracle then locates the rows based on this ROWID.

Hash Scans

Oracle can use a hash scan to locate rows in a hash cluster based on a hash value. In a hash cluster, all rows with the same hash value are stored in the same data blocks. To perform a hash scan, Oracle first obtains the hash value by applying a hash function to a cluster key value specified by the statement. Oracle then scans the data blocks containing rows with that hash value.

Index Scans

An index scan retrieves data from an index based on the value of one or more columns of the index. To perform an index scan, Oracle searches the index for the indexed column values accessed by the statement. If the statement accesses only columns of the index, Oracle reads the indexed column values directly from the index, rather than from the table.

The index contains not only the indexed value, but also the ROWIDs of rows in the table having that value. Therefore, if the statement accesses other columns in addition to the indexed columns, Oracle can find the rows in the table with a table access by ROWID or a cluster scan.

An index scan can be one of these types:

Unique scan	A unique scan of an index returns only a single ROWID. Oracle performs a unique scan only in cases in which a single ROWID is required, rather than many ROWIDs. For example, Oracle performs a unique scan if there is a UNIQUE or a PRIMARY KEY constraint that guarantees that the statement accesses only a single row.
Range scan	A range scan of an index can return zero or more ROWIDs depending on how many rows the statement accesses.
Full scan	Full scan is available if a predicate references one of the columns in the index. The predicate does not have to be an index driver. Full scan is also available when there is no predicate if all of the columns in the table referenced in the query are included in the index and at least one of the index columns is not nullable. Full scan can be used to eliminate a sort operation. It reads the blocks singly.
Fast full scan	Fast full scan is an alternative to a full table scan when the index contains all the columns that are needed for the query. It cannot be used to eliminate a sort operation. It reads the entire index using multiblock reads. Fast full scan is available only with cost-based optimization; you specify it with the INDEX_FFS hint.
Bitmap	Bitmap access is available with cost-based optimization.

Access Paths

Table 19-1 lists the data access paths. The optimizer can only choose to use a particular access path for a table if the statement contains a WHERE clause condition or other construct that makes that access path available.

The rule-based approach uses the rank of each path to choose a path when more than one path is available (see "Choosing Among Access Paths with the Rule-Based Approach" on page 19-53). The cost-based approach chooses a path based on resource use (see "Choosing Among Access Paths with the Cost-Based Approach" on page 19-50).

Table 19-1: Access Paths

Rank	Access Path
1	Single row by ROWID
2	Single row by cluster join
3	Single row by hash cluster key with unique or primary key
4	Single row by unique or primary key
5	Cluster join
6	Hash cluster key
7	Indexed cluster key
8	Composite key
9	Single-column indexes
10	Bounded range search on indexed columns
11	Unbounded range search on indexed columns
12	Sort-merge join
13	MAX or MIN of indexed column
14	ORDER BY on indexed columns
15	Full table scan Fast full index scan (no rank; not available with the rule-based optimizer): see Oracle8 Server Tuning Bitmap index scan (no rank; not available with the rule-based optimizer): see "Bitmap Indexes" on page 7-25

Each of the following sections describes an access path and discusses:

- when it is available
- the method Oracle uses to access data with it
- the output generated for it by the EXPLAIN PLAN command

Path 1: Single Row by ROWID

This access path is available only if the statement's WHERE clause identifies the selected rows by ROWID or with the CURRENT OF CURSOR embedded SQL syntax supported by the Oracle Precompilers. To execute the statement, Oracle accesses the table by ROWID.

Example: This access path is available in the following statement:

```
SELECT * FROM emp WHERE ROWID =
'00000DC5.0000.0001';
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION            OPTIONS      OBJECT_NAME
-------------------------------------------------
SELECT STATEMENT
        TABLE ACCESS       BY ROWID      EMP
```

Path 2: Single Row by Cluster Join

This access path is available for statements that join tables stored in the same cluster if both of these conditions are true:

- The statement's WHERE clause contains conditions that equate each column of the cluster key in one table with the corresponding column in the other table.
- The statement's WHERE clause also contains a condition that guarantees that the join returns only one row. Such a condition is likely to be an equality condition on the column(s) of a unique or primary key.

These conditions must be combined with AND operators. To execute the statement, Oracle performs a nested loops operation. For information on the nested loops operation, see "Join Operations" on page 19-55.

Example: This access path is available for the following statement in which the EMP and DEPT tables are clustered on the DEPTNO column and the EMPNO column is the primary key of the EMP table:

```
SELECT *
 FROM emp, dept
 WHERE emp.deptno = dept.deptno
  AND emp.empno = 7900;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION            OPTIONS      OBJECT_NAME
-------------------------------------------------
SELECT                            STATEMENT
  NESTED LOOPS
    TABLE ACCESS     BY ROWID     EMP
    INDEX            UNIQUE SCAN  PK_EMP
    TABLE ACCESS     CLUSTER      DEPT
```

PK_EMP is the name of an index that enforces the primary key.

Path 3: Single Row by Hash Cluster Key with Unique or Primary Key

This access path is available if both of these conditions are true:

- The statement's WHERE clause uses all columns of a hash cluster key in equality conditions. For composite cluster keys, the equality conditions must be combined with AND operators.
- The statement is guaranteed to return only one row because the columns that make up the hash cluster key also make up a unique or primary key.

To execute the statement, Oracle applies the cluster's hash function to the hash cluster key value specified in the statement to obtain a hash value. Oracle then uses the hash value to perform a hash scan on the table.

Example: This access path is available in the following statement in which the ORDERS and LINE_ITEMS tables are stored in a hash cluster, and the ORDERNO column is both the cluster key and the primary key of the ORDERS table:

```
SELECT *
  FROM orders
  WHERE orderno = 65118968;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION          OPTIONS      OBJECT_NAME
-------------------------------------------
SELECT STATEMENT
  TABLE ACCESS       HASH       ORDERS
```

Path 4: Single Row by Unique or Primary Key

This access path is available if the statement's WHERE clause uses all columns of a unique or primary key in equality conditions. For composite keys, the equality conditions must be combined with AND operators. To execute the statement, Oracle performs a unique scan on the index on the unique or primary key to retrieve a single ROWID and then accesses the table by that ROWID.

Example: This access path is available in the following statement in which the EMPNO column is the primary key of the EMP table:

```
SELECT *
  FROM emp
  WHERE empno = 7900;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION          OPTIONS      OBJECT_NAME
-------------------------------------------
SELECT STATEMENT
 TABLE ACCESS        BY ROWID    EMP
  INDEX            UNIQUE SCAN  PK_EMP
```

PK_EMP is the name of the index that enforces the primary key.

Path 5: Clustered Join

This access path is available for statements that join tables stored in the same cluster if the statement's WHERE clause contains conditions that equate each column of the cluster key in one table with the corresponding column in the other table. For a composite cluster key, the equality conditions must be combined with AND operators. To execute the statement, Oracle performs a nested loops operation. For information on nested loops operations, see "Join Operations" on page 19-55.

Example: This access path is available in the following statement in which the EMP and DEPT tables are clustered on the DEPTNO column:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION          OPTIONS      OBJECT_NAME
-------------------------------------------
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS       FULL       DEPT
  TABLE ACCESS       CLUSTER    EMP
```

Path 6: Hash Cluster Key

This access path is available if the statement's WHERE clause uses all the columns of a hash cluster key in equality conditions. For a composite cluster key, the equality conditions must be combined with AND operators. To execute the statement, Oracle applies the cluster's hash function to the hash cluster key value specified in the statement to obtain a hash value. Oracle then uses this hash value to perform a hash scan on the table.

Example: This access path is available for the following statement in which the ORDERS and LINE_ITEMS tables are stored in a hash cluster and the ORDERNO column is the cluster key:

```
SELECT *
  FROM line_items
  WHERE orderno = 65118968;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION          OPTIONS      OBJECT_NAME
-------------------------------------------
SELECT STATEMENT
  TABLE ACCESS       HASH       LINE_ITEMS
```

Path 7: Indexed Cluster Key

This access path is available if the statement's WHERE clause uses all the columns of an indexed cluster key in equality conditions. For a composite cluster key, the equality conditions must be combined with AND operators. To execute the statement, Oracle performs a unique scan on the cluster index to retrieve the ROWID of one row with the specified cluster key value. Oracle then uses that ROWID to access the table with a cluster scan. Since all rows with the same cluster key value are stored together, the cluster scan requires only a single ROWID to find them all.

Example: This access path is available in the following statement in which the EMP table is stored in an indexed cluster and the DEPTNO column is the cluster key:

```
SELECT *  FROM emp
          WHERE deptno = 10;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION          OPTIONS      OBJECT_NAME
-------------------------------------------
SELECT STATEMENT
 TABLE ACCESS        CLUSTER    EMP
  INDEX            UNIQUE SCAN  PERS_INDEX
```

PERS_INDEX is the name of the cluster index.

Path 8: Composite Index

This access path is available if the statement's WHERE clause uses all columns of a composite index in equality conditions combined with AND operators. To execute the statement,

Oracle performs a range scan on the index to retrieve ROWIDs of the selected rows and then accesses the table by those ROWIDs.

Example: This access path is available in the following statement in which there is a composite index on the JOB and DEPTNO columns:

```
SELECT *
  FROM emp
  WHERE job = 'CLERK'
    AND deptno = 30;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION          OPTIONS      OBJECT_NAME
-------------------------------------------
SELECT STATEMENT
 TABLE ACCESS        BY ROWID     EMP
  INDEX           RANGE SCAN
JOB_DEPTNO_INDEX
```

JOB_DEPTNO_INDEX is the name of the composite index on the JOB and DEPTNO columns.

Path 9: Single-Column Indexes

This access path is available if the statement's WHERE clause uses the columns of one or more single-column indexes in equality conditions. For multiple single-column indexes, the conditions must be combined with AND operators.

If the WHERE clause uses the column of only one index, Oracle executes the statement by performing a range scan on the index to retrieve the ROWIDs of the selected rows and then accessing the table by these ROWIDs.

Example: This access path is available in the following statement in which there is an index on the JOB column of the EMP table:

```
SELECT *
  FROM emp
  WHERE job = 'ANALYST';
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION          OPTIONS      OBJECT_NAME
-------------------------------------------
SELECT STATEMENT
 TABLE ACCESS        BY ROWID     EMP
  INDEX           RANGE SCAN   JOB_INDEX
```

JOB_INDEX is the index on EMP.JOB.

If the WHERE clauses uses columns of many single-column indexes, Oracle executes the statement by performing a range scan on each index to retrieve the ROWIDs of the rows that satisfy each condition. Oracle then merges the sets of ROWIDs to obtain a set of ROWIDs of rows that satisfy all conditions. Oracle then accesses the table using these ROWIDs.

Oracle can merge up to five indexes. If the WHERE clause uses columns of more than five single-column indexes, Oracle merges five of them, accesses the table by ROWID, and then tests the resulting rows to determine whether they satisfy the remaining conditions before returning them.

Example: This access path is available in the following statement in which there are indexes on both the JOB and DEPTNO columns of the EMP table:

```
SELECT *
  FROM emp
  WHERE job = 'ANALYST'
    AND deptno = 20;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION          OPTIONS      OBJECT_NAME
-------------------------------------------
SELECT STATEMENT
 TABLE ACCESS        BY ROWID     EMP
  AND-EQUAL
   INDEX           RANGE SCAN   JOB_INDEX
   INDEX           RANGE SCAN   DEPTNO_INDEX
```

The AND-EQUAL operation merges the ROWIDs obtained by the scans of the JOB_INDEX and the DEPTNO_INDEX, resulting in a set of ROWIDs of rows that satisfy the query.

Path 10: Bounded Range Search on Indexed Columns

This access path is available if the statement's WHERE clause contains a condition that uses either the column of a single-column index or one or more columns that make up a leading portion of a composite index:

column = expr

column >[=] expr AND column <[=] expr

column BETWEEN expr AND expr

column LIKE 'c%'

Each of these conditions specifies a bounded range of indexed values that are accessed by the statement. The range is said to be bounded because the conditions specify both its least value and its greatest value. To execute such a statement, Oracle performs a range scan on the index and then accesses the table by ROWID.

This access path is not available if the expression *expr* references the indexed column.

Example: This access path is available in this statement in which there is an index on the SAL column of the EMP table:

```
SELECT *
  FROM emp
  WHERE sal BETWEEN 2000 AND 3000;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION          OPTIONS      OBJECT_NAME
-------------------------------------------
SELECT STATEMENT

 TABLE ACCESS        BY ROWID     EMP

  INDEX           RANGE SCAN   SAL_INDEX
```

SAL_INDEX is the name of the index on EMP.SAL.

Example: This access path is also available in the following statement in which there is an index on the ENAME column of the EMP table:

```
SELECT *
FROM emp
WHERE ename LIKE 'S%';
```

Path 11: Unbounded Range Search on Indexed Columns
This access path is available if the statement's WHERE clause contains one of these conditions that use either the column of a single-column index or one or more columns of a leading portion of a composite index:

```
WHERE column >[=] expr
WHERE column <[=] expr
```

Each of these conditions specifies an unbounded range of index values accessed by the statement. The range is said to be unbounded because the condition specifies either its least value or its greatest value, but not both. To execute such a statement, Oracle performs a range scan on the index and then accesses the table by ROWID.

Example: This access path is available in the following statement in which there is an index on the SAL column of the EMP table:

```
SELECT *
  FROM emp
  WHERE sal > 2000;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION            OPTIONS      OBJECT_NAME
-------------------------------------------------
SELECT STATEMENT
 TABLE ACCESS        BY ROWID     EMP
  INDEX              RANGE SCAN   SAL_INDEX
```

Example: This access path is available in the following statement in which there is a composite index on the ORDER and LINE columns of the LINE_ITEMS table:

```
SELECT *
  FROM line_items
  WHERE order > 65118968;
```

The access path is available because the WHERE clause uses the ORDER column, a leading portion of the index.

Example: This access path is not available in the following statement in which there is an index on the ORDER and LINE columns:

```
SELECT *
  FROM line_items
  WHERE line < 4;
```

The access path is not available because the WHERE clause only uses the LINE column, which is not a leading portion of the index.

Path 12: Sort-Merge Join
This access path is available for statements that join tables that are not stored together in a cluster if the statement's WHERE clause uses columns from each table in equality conditions. To execute such a statement, Oracle uses a sort-merge operation. Oracle can also use a nested loops operation to execute a join statement. For information on these operations and on when the optimizer chooses one over the other, see "Optimizing Join Statements" on page 19-54.

Example: This access path is available for the following statement in which the EMP and DEPT tables are not stored in the same cluster:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION            OPTIONS      OBJECT_NAME
-------------------------------------------------
SELECT STATEMENT
 MERGE JOIN
  SORT               JOIN
   TABLE ACCESS      FULL         EMP
  SORT               JOIN
 TABLE ACCESS        FULL         DEPT
```

Path 13: MAX or MIN of Indexed Column
This access path is available for a SELECT statement for which all of these conditions are true:

- The query uses the MAX or MIN function to select the maximum or minimum value of either the column of a single-column index or the leading column of a composite index. The index cannot be a cluster index. The argument to the MAX or MIN function can be any expression involving the column, a constant, or the addition operator (+), the concatenation operation (||), or the CONCAT function.
- There are no other expressions in the select list.
- The statement has no WHERE clause or GROUP BY clause.

To execute the query, Oracle performs a range scan of the index to find the maximum or minimum indexed value. Since only this value is selected, Oracle need not access the table after scanning the index.

Example: This access path is available for the following statement in which there is an index on the SAL column of the EMP table:

```
SELECT MAX(sal) FROM emp;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION            OPTIONS      OBJECT_NAME
-------------------------------------------------
SELECT STATEMENT
 AGGREGATE           GROUP BY
  INDEX              RANGE SCAN   SAL_INDEX
```

Path 14: ORDER BY on Indexed Column
This access path is available for a SELECT statement for which all of these conditions are true:

- The query contains an ORDER BY clause that uses either the column of a single-column index or a leading portion of a composite index. The index cannot be a cluster index.

- There must be a PRIMARY KEY or NOT NULL integrity constraint that guarantees that at least one of the indexed columns listed in the ORDER BY clause contains no nulls.
- The NLS_SORT parameter is set to BINARY.

To execute the query, Oracle performs a range scan of the index to retrieve the ROWIDs of the selected rows in sorted order. Oracle then accesses the table by these ROWIDs.

Example: This access path is available for the following statement in which there is a primary key on the EMPNO column of the EMP table:

```
SELECT *
  FROM emp
  ORDER BY empno;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION          OPTIONS      OBJECT_NAME
_____________________________________________

SELECT STATEMENT
 TABLE ACCESS        BY ROWID     EMP
  INDEX              RANGE SCAN   PK_EMP
```

PK_EMP is the name of the index that enforces the primary key. The primary key ensures that the column does not contain nulls.

This access path is available for any SQL statement, regardless of its WHERE clause conditions.

This statement uses a full table scan to access the EMP table:

```
SELECT *
  FROM emp;
```

The EXPLAIN PLAN output for this statement might look like this:

```
OPERATION          OPTIONS      OBJECT_NAME
_____________________________________________

SELECT STATEMENT

 TABLE ACCESS        FULL         EMP
```

Note that these conditions do not make index access paths available:

- column1 > column2
- column1 < column2
- column1 >= column2
- column1 <= column2

where *column1* and *column2* are in the same table.

- column IS NULL
- column IS NOT NULL
- column NOT IN
- column != expr
- column LIKE '%pattern'

regardless of whether *column* is indexed.

- expr = expr2

where *expr* is an expression that operates on a column with an operator or function, regardless of whether the column is indexed.

- NOT EXISTS subquery
- any condition involving a column that is not indexed

Any SQL statement that contains only these constructs and no others that make index access paths available must use full table scans.

Choosing Among Access Paths

This section describes how the optimizer chooses among available access paths:

- when using the cost-based approach
- when using the rule-based approach

Choosing Among Access Paths with the Cost-Based Approach

With the cost-based approach, the optimizer chooses an access path based on these factors:

- the available access paths for the statement
- the estimated cost of executing the statement using each access path or combination of paths

To choose an access path, the optimizer first determines which access paths are available by examining the conditions in the statement's WHERE clause. The optimizer then generates a set of possible execution plans using available access paths and estimates the cost of each plan using the statistics for the index, columns, and tables accessible to the statement. The optimizer then chooses the execution plan with the lowest estimated cost.

The optimizer's choice among available access paths can be overridden with hints. For information on hints, see Oracle8 Server Tuning.

To choose among available access paths, the optimizer considers these factors:

- **Selectivity:** The *selectivity* is the percentage of rows in the table that the query selects. Queries that select a small percentage of a table's rows have good selectivity, while a query that selects a large percentage of the rows has poor selectivity.

The optimizer is more likely to choose an index scan over a full table scan for a query with good selectivity than for one with poor selectivity. Index scans are usually more efficient than full table scans for queries that access only a small percentage of a table's rows, while full table scans are usually faster for queries that access a large percentage.

To determine the selectivity of a query, the optimizer considers these sources of information:

- the operators used in the WHERE clause
- unique and primary key columns used in the WHERE clause
- statistics for the table

The following examples in this section illustrate the way the optimizer uses selectivity.

- **DB_FILE_MULTIBLOCK_READ_COUNT:** Full table scans use multiblock reads, so the cost of a full table scan depends on the number of multiblock reads required

to read the entire table, which depends on the number of blocks read by a single multiblock read, which is specified by the initialization parameter DB_FILE_MULTIBLOCK_READ_COUNT. For this reason, the optimizer may be more likely to choose a full table scan when the value of this parameter is high.

Example: Consider this query, which uses an equality condition in its WHERE clause to select all employees named Jackson:

```
SELECT *
  FROM emp
  WHERE ename = 'JACKSON';
```

If the ENAME column is a unique or primary key, the optimizer determines that there is only one employee named Jackson, and the query returns only one row. In this case, the query is very selective, and the optimizer is most likely to access the table using a unique scan on the index that enforces the unique or primary key (access path 4).

Example: Consider again the query in the previous example. If the ENAME column is not a unique or primary key, the optimizer can use these statistics to estimate the query's selectivity:

- USER_TAB_COLUMNS.NUM_DISTINCT is the number of values for each column in the table.
- USER_TABLES.NUM_ROWS is the number of rows in each table.

By dividing the number of rows in the EMP table by the number of distinct values in the ENAME column, the optimizer estimates what percentage of employees have the same name. By assuming that the ENAME values are uniformly distributed, the optimizer uses this percentage as the estimated selectivity of the query.

Example: Consider this query, which selects all employees with employee ID numbers less than 7500:

```
SELECT *
  FROM emp
  WHERE empno < 7500;
```

To estimate the selectivity of the query, the optimizer uses the boundary value of 7500 in the WHERE clause condition and the values of the HIGH_VALUE and LOW_VALUE statistics for the EMPNO column if available. These statistics can be found in the USER_TAB_COLUMNS view. The optimizer assumes that EMPNO values are evenly distributed in the range between the lowest value and highest value. The optimizer then determines what percentage of this range is less than the value 7500 and uses this value as the estimated selectivity of the query.

Example: Consider this query, which uses a bind variable rather than a literal value for the boundary value in the WHERE clause condition:

```
SELECT *
  FROM emp
  WHERE empno < :e1;
```

The optimizer does not know the value of the bind variable E1. Indeed, the value of E1 may be different for each execution of the query. For this reason, the optimizer cannot use the means described in the previous example to determine selectivity of this query. In this case, the optimizer heuristically guesses a small value for the selectivity of the column (because it is indexed). The optimizer makes this assumption whenever a bind variable is used as a boundary value in a condition with one of the operators <, >, <=, or >=.

The optimizer's treatment of bind variables can cause it to choose different execution plans for SQL statements that differ only in the use of bind variables rather than constants. In one case in which this difference may be especially apparent, the optimizer may choose different execution plans for an embedded SQL statement with a bind variable in an Oracle Precompiler program and the same SQL statement with a constant in SQL*Plus.

Example: Consider this query, which uses two bind variables as boundary values in the condition with the BETWEEN operator:

```
SELECT *
  FROM emp
  WHERE empno BETWEEN :low_e AND :high_e;
```

The optimizer decomposes the BETWEEN condition into these two conditions:

```
empno >= :low_e
empno <= :high_e
```

The optimizer heuristically estimates a small selectiviy for indexed columns in order to favor the use of the index.

Example: Consider this query, which uses the BETWEEN operator to select all employees with employee ID numbers between 7500 and 7800:

```
SELECT *
  FROM emp
  WHERE empno BETWEEN 7500 AND 7800;
```

To determine the selectivity of this query, the optimizer decomposes the WHERE clause condition into these two conditions:

```
empno >= 7500
empno <= 7800
```

The optimizer estimates the individual selectivity of each condition using the means described in a previous example. The optimizer then uses these selectivities (*S1* and *S2*) and the absolute value function (ABS) in this formula to estimate the selectivity (*S*) of the BETWEEN condition:

$$S = \text{ABS}(S1 + S2 - 1)$$

Choosing Among Access Paths with the Rule-Based Approach

With the rule-based approach, the optimizer chooses whether to use an access path based on these factors:

- the available access paths for the statement
- the ranks of these access paths in Table 19-1 on page 19-39

To choose an access path, the optimizer first examines the conditions in the statement's WHERE clause to determine which access paths are available. The optimizer then chooses the most highly ranked available access path.

Note that the full table scan is the lowest ranked access path on the list. This means that the rule-based approach always chooses an access path that uses an index if one is available, even if a full table scan might execute faster.

The order of the conditions in the WHERE clause does not normally affect the optimizer's choice among access paths.

Example: Consider this SQL statement, which selects the employee numbers of all employees in the EMP table with an ENAME value of 'CHUNG' and with a SAL value greater than 2000:

```
SELECT empno
  FROM emp
  WHERE ename = 'CHUNG'
    AND sal > 2000;
```

Consider also that the EMP table has these integrity constraints and indexes:

- There is a PRIMARY KEY constraint on the EMPNO column that is enforced by the index PK_EMPNO.
- There is an index named ENAME_IND on the ENAME column.
- There is an index named SAL_IND on the SAL column.

Based on the conditions in the WHERE clause of the SQL statement, the integrity constraints, and the indexes, these access paths are available:

- A single-column index access path using the ENAME_IND index is made available by the condition ENAME = 'CHUNG'. This access path has rank 9.
- An unbounded range scan using the SAL_IND index is made available by the condition SAL > 2000. This access path has rank 11.
- A full table scan is automatically available for all SQL statements. This access path has rank 15.

Note that the PK_EMPNO index does not make the single row by primary key access path available because the indexed column does not appear in a condition in the WHERE clause.

Using the rule-based approach, the optimizer chooses the access path that uses the ENAME_IND index to execute this statement. The optimizer chooses this path because it is the most highly ranked path available.

Optimizing Join Statements

To choose an execution plan for a join statement, the optimizer must choose:

Access paths	As for simple statements, the optimizer must choose an access path to retrieve data from each table in the join statement. (See "Choosing Access Paths" on page 19-37.)
Join operations	To join each pair of row sources, Oracle must perform one of these operations:
	nested loops
	sort-merge
	cluster
	hash join (not available with rule-based optimization)
Join order	To execute a statement that joins more than two tables, Oracle joins two of the tables, and then joins the resulting row source to the next table. This process is continued until all tables are joined into the result.

These choices are interrelated.

Join Operations

This section describes the operations that the optimizer can use to join two row sources:

- Nested Loops Join
- Sort-Merge Join
- Cluster Join
- Hash Join

Nested Loops Join

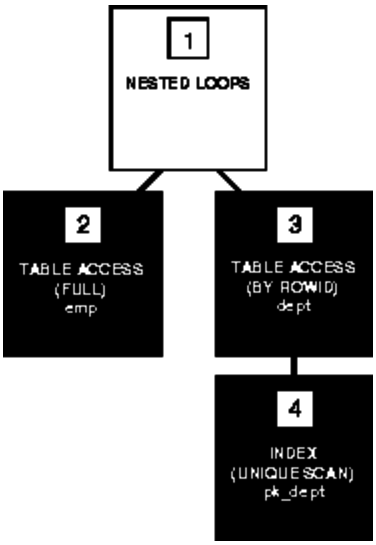
To perform a nested loops join, Oracle follows these steps:

1. The optimizer chooses one of the tables as the *outer table*, or the *driving table*. The other table is called the *inner table*.
2. For each row in the outer table, Oracle finds all rows in the inner table that satisfy the join condition.
3. Oracle combines the data in each pair of rows that satisfy the join condition and returns the resulting rows.

Figure 19-11, "Nested Loops Join", shows the execution plan for this statement using a nested loops join:

```
SELECT *
  FROM emp, dept

  WHERE emp.deptno = dept.deptno;
```

Figure 19-11: Nested Loops Join



To execute this statement, Oracle performs these steps:

- Step 2 accesses the outer table (EMP) with a full table scan.
- For each row returned by Step 2, Step 4 uses the EMP.DEPTNO value to perform a unique scan on the PK_DEPT index.
- Step 3 uses the ROWID from Step 4 to locate the matching row in the inner table (DEPT).
- Oracle combines each row returned by Step 2 with the matching row returned by Step 4 and returns the result.

Sort-Merge Join

To perform a sort-merge join, Oracle follows these steps:

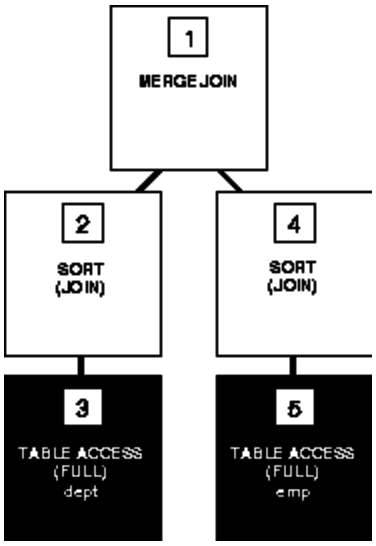
1. Oracle sorts each row source to be joined if they have not been sorted already by a previous operation. The rows are sorted on the values of the columns used in the join condition.
2. Oracle merges the two sources so that each pair of rows, one from each source, that contain matching values for the columns used in the join condition are combined and returned as the resulting row source.

Oracle can only perform a sort-merge join for an equijoin.

Figure 19-12, “Sort-Merge Join”, shows the execution plan for this statement using a sort-merge join:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno;
```

Figure 19-12: Sort-Merge Join



To execute this statement, Oracle performs these steps:

- Steps 3 and 5 perform full table scans of the EMP and DEPT tables.
- Steps 2 and 4 sort each row source separately.
- Step 1 merges the sources from Steps 2 and 4 together, combining each row from Step 2 with each matching row from Step 4 and returns the resulting row source.

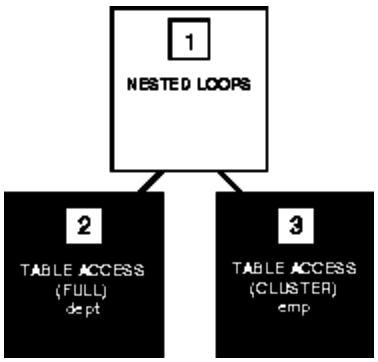
Cluster Join

Oracle can perform a cluster join only for an equijoin that equates the cluster key columns of two tables in the same cluster. In a cluster, rows from both tables with the same cluster key values are stored in the same blocks, so Oracle only accesses those blocks. For information on clusters, including how to decide which tables to cluster for best performance, see the Chapter “Tuning SQL Statements” in Oracle8 Server Tuning.

Figure 19-13, “Cluster Join”, shows the execution plan for this statement in which the EMP and DEPT tables are stored together in the same cluster:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno;
```

Figure 19-13: Cluster Join



To execute this statement, Oracle performs these steps:

- Step 2 accesses the outer table (DEPT) with a full table scan.
- For each row returned by Step 2, Step 3 uses the DEPT.DEPTNO value to find the matching rows in the inner table (EMP) with a cluster scan.

A cluster join is nothing more than a nested loops join involving two tables that are stored together in a cluster. Since each row from the DEPT table is stored in the same data blocks as the matching rows in the EMP table, Oracle can access matching rows most efficiently.

Hash Join

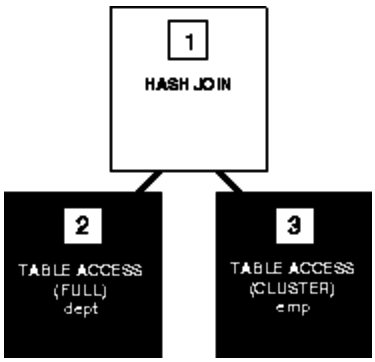
Oracle can only perform a hash join for an equijoin. To perform a hash join, Oracle follows these steps:

1. Oracle performs a full table scan on each of the tables and splits each into as many partitions as possible based on the available memory.
2. Oracle builds a hash table from one of the partitions (if possible, Oracle will select a partition that fits into available memory). Oracle then uses the corresponding partition in the other table to probe the hash table. All partitions pairs that do not fit into memory are placed onto disk.
3. For each pair of partitions (one from each table), Oracle uses the smaller one to build a hash table and the larger one to probe the hash table.

Hash join is not available with rule-based optimization. Figure 19-14, “Hash Join”, shows the execution plan for this statement using a hash join:

```
SELECT *
  FROM emp, dept
  WHERE emp.deptno = dept.deptno;
```

Figure 19-14: Hash Join



To execute this statement, Oracle performs these steps:

- Steps 2 and 3 perform full table scans of the EMP and DEPT tables.
- Step 1 builds a hash table out of the rows coming from 2 and probes it with each row coming from 3.

The initialization parameter HASH_AREA_SIZE controls the memory to be used for hash join operations and the initialization parameter HASH_MULTIBLOCK_IO_COUNT controls the number of blocks a hash join operation should read and write concurrently. See Oracle8 Server Reference Manual for more information about these initialization parameters.

Choosing Execution Plans for Join Statements

This section describes how the optimizer chooses an execution plan for a join statement:

- when using the cost-based approach
- when using the rule-based approach

Note these considerations that apply to the cost-based and rule-based approaches:

- The optimizer first determines whether joining two or more of the tables definitely results in a row source containing at most one row. The optimizer recognizes such situations based on UNIQUE and PRIMARY KEY constraints on the tables. If such a situation exists, the optimizer places these tables first in the join order. The optimizer then optimizes the join of the remaining set of tables.
- For join statements with outer join conditions, the table with the outer join operator must come after the other table in the condition in the join order. The optimizer does not consider join orders that violate this rule.

Choosing Execution Plans for Joins with the Cost-Based Approach

With the cost-based approach, the optimizer generates a set of execution plans based on the possible join orders, join operations, and available access paths. The optimizer then estimates the cost of each plan and chooses the one with the lowest cost. The optimizer estimates costs in these ways:

- The cost of a nested loops operation is based on the cost of reading each selected row of the outer table and each of its matching rows of the inner table into memory. The optimizer estimates these costs using the statistics in the data dictionary.
- The cost of a sort-merge join is based largely on the cost of reading all the sources into memory and sorting them.
- The optimizer also considers other factors when determining the cost of each operation. For example:
  - A smaller sort area size is likely to increase the cost for a sort-merge join because sorting takes more CPU time and I/O in a smaller sort area. Sort area size is specified by the initialization parameter SORT_AREA_SIZE.
  - A larger multi-block read count is likely to decrease the cost for a sort-merge join in relation to a nested loops join. If a large number of sequential blocks can be read from disk in a single I/O, an index on the inner table for the nested loops join is less likely to improve performance over a full table scan. The multi-block read count is specified by the initialization parameter DB_FILE_MULTIBLOCK_READ_COUNT.
  - For join statements with outer join conditions, the table with the outer join operator must come after the other table in the condition in the join order. The optimizer does not consider join orders that violate this rule.

With the cost-based approach, the optimizer's choice of join orders can be overridden with the ORDERED hint. If the ORDERED hint specifies a join order that violates the rule for outer join, the optimizer ignores the hint and chooses the order. You can also override the optimizer's choice of join operations with hints. For information on using hints, see Oracle8 Server Tuning.

Choosing Execution Plans for Joins with the Rule-Based Approach

With the rule-based approach, the optimizer follows these steps to choose an execution plan for a statement that joins R tables:

1. The optimizer generates a set of R join orders, each with a different table as the first table. The optimizer generates each potential join order using this algorithm:
   a. To fill each position in the join order, the optimizer chooses the table with the most highly ranked available access path according to the ranks for access paths in Table 19-1 on page 19-39. The optimizer repeats this step to fill each subsequent position in the join order.
   b. For each table in the join order, the optimizer also chooses the operation with which to join the table to the previous table or row source in the order. The optimizer does this by "ranking" the sort-merge operation as access path 12 and applying these rules:
   c. If the access path for the chosen table is ranked 11 or better, the optimizer chooses a nested loops operation using the previous table or row source in the join order as the outer table.
   d. If the access path for the table is ranked lower than 12, and there is an equijoin condition between the chosen table and the previous table or row source in join order, the optimizer chooses a sort-merge operation.
   e. If the access path for the chosen table is ranked lower than 12, and there is not an equijoin condition, the optimizer chooses a nested loops operation with the previous table or row source in the join order as the outer table.
2. The optimizer then chooses among the resulting set of execution plans. The goal of the optimizer's choice is to maximize the number of nested loops join operations in which the inner table is accessed using an index scan. Since a nested loops join involves accessing the inner table many times, an index on the inner table can greatly improve the performance of a nested loops join.

   Usually, the optimizer does not consider the order in which tables appear in the FROM clause when choosing an execution plan. The optimizer makes this choice by applying the following rules in order:
   a. The optimizer chooses the execution plan with the fewest nested-loops operations in which the inner table is accessed with a full table scan.
   b. If there is a tie, the optimizer chooses the execution plan with the fewest sort-merge operations.
   c. If there is still a tie, the optimizer chooses the execution plan for which the first table in the join order has the most highly ranked access path:
      - If there is a tie among multiple plans whose first tables are accessed by the single-column indexes access path, the optimizer chooses the plan whose first table is accessed with the most merged indexes.
      - If there is a tie among multiple plans whose first tables are accessed by bounded range scans, the optimizer chooses the plan whose first table is accessed with the greatest number of leading columns of the composite index.
   d. If there is still a tie, the optimizer chooses the execution plan for which the first table appears later in the query's FROM clause.

Optimizing "Star" Queries

One type of data warehouse design centers around what is known as a "star" schema, which is characterized by one or more very large *fact* tables that contain the primary information in the data warehouse and a number of much smaller *dimension* tables (or "*lookup*" tables), each of which contains information about the entries for a particular attribute in the fact table.

A *star query* is a join between a fact table and a number of lookup tables. Each lookup table is joined to the fact table using a primary-key to foreign-key join, but the lookup tables are not joined to each other.

The Oracle cost-based optimizer recognizes star queries and generates efficient execution plans for them. (Star queries are not recognized by the rule-based optimizer.)

A typical fact table contains *keys* and *measures*. For example, a simple fact table might contain the measure Sales, and keys Time, Product, and Market. In this case there would be corresponding dimension tables for Time, Product, and Market. The Product dimension table, for example, would typically contain information about each product number that appears in the fact table.

A *star join* is a primary-key to foreign-key join of the dimension tables to a fact table. The fact table normally has a concatenated index on the key columns to facilitate this type of join.

Star Query Example

This section discusses star queries with reference to the following example:

```
SELECT SUM(dollars)
  FROM facts, time, product, market
  WHERE market.stat = 'New York'
    AND product.brand = 'MyBrand'
    AND time.year = 1995
    AND time.month = 'March'
    /* Joins*/
    AND time.key = facts.tkey
    AND product.pkey = facts.pkey
    AND market.mkey = facts.mkey;
```

Tuning Star Queries

To execute star queries efficiently, you must use the cost based optimizer. Begin by using the ANALYZE command to gather statistics for each of the tables accessed by the query.

Indexing

In the example above, you would construct a concatenated index on the columns tkey, pkey, and mkey. The order of the columns in the index is critical to performance. the columns in the index should take advantage of any ordering of the data. If rows are added to the large table in time order, then tkey should be the first key in the index. When the data is a static extract from another database, it is worthwhile to sort the data on the key columns before loading it.

If all queries specify predicates on each of the small tables, a single concatenated index suffices. If queries that omit leading columns of the concatenated index are frequent, additional indexes may be useful. In this example, if there are frequent queries that omit the time table, an index on pkey and mkey can be added.

Hints

Usually, if you analyze the tables the optimizer will choose an efficient star plan. You can also use hints to improve the plan. The most precise method is to order the tables in the FROM clause in the order of the keys in the index, with the large table last. Then use the following hints:

```
/*+ ORDERED USE_NL(facts) INDEX(facts fact_concat) */
```

A more general method is to use the STAR hint /*+ STAR */.

Extended Star Schemas

Each of the small tables can be replaced by a join of several smaller tables. For example, the product table could be normalized into brand and manufacturer tables. Normalization of all of the small tables can cause performance problems. One problem is caused by the increased number of permutations that the optimizer must consider. The other problem is the result of multiple executions of the small table joins. Both problems can be solved by using denormalized views. For example:

```
CREATE VIEW prodview AS SELECT /*+ NO_MERGE
*/ *
  FROM brands, mfgrs WHERE brands.mfkey =
mfgrs.mfkey;
```

This hint will both reduce the optimizer's search space, and cause caching of the result of the view.

Star Transformation

The star transformation is a cost-based query transformation aimed at executing star queries efficiently. Whereas the star optimization works well for schemas with a small number of dimensions and dense fact tables, the star transformation may be considered as an alternative if any of the following holds true:

- The number of dimensions is large.
- The fact table is sparse.
- There are queries where not all dimension tables have constraining predicates.

The star transformation does not rely on computing a Cartesian product of the dimension tables, which makes it better suited for cases where fact table sparsity and/or a large number of dimensions would lead to a large Cartesian product with few rows having actual matches in the fact table. In addition, rather than relying on concatenated indexes, the star transformation is based on combining bitmap indexes on individual fact table columns.

The transformation can thus choose to combine indexes corresponding precisely to the constrained dimensions. There is no need to create many concatenated indexes where the different column orders match different patterns of constrained dimensions in different queries.

The star transformation works by generating new subqueries that can be used to drive a bitmap index access path for the fact table. Consider a simple case with three dimension tables, "d1", "d2", and "d3", and a fact table, "fact". The following query:

```
EXPLAIN PLAN FOR
 SELECT * FROM fact, d1, d2, d3
   WHERE fact.c1 = d1.c1 AND fact.c2 = d2.c1 AND fact.c3 =
d3.c1
   AND d1.c2 IN (1, 2, 3, 4)
   AND d2.c2 < 100
   AND d3.c2 = 35
```

gets transformed by adding three subqueries:

```
SELECT * FROM fact, d1, d2
 WHERE fact.c1 = d1.c1 AND fact.c2 = d2.c1
   AND d1.c2 IN (1, 2, 3, 4)
   AND d2.c2 < 100
   AND fact.c1 IN (SELECT d1.c1 FROM d1 WHERE d1.c2
IN (1, 2, 3, 4))
   AND fact.c2 IN (select d2.c1 FROM d2 WHERE d2.c2 <
100)
   AND fact.c3 IN (SELECT d3.c1 FROM d3 WHERE d3.c2 =
35)
```

Given that there are bitmap indexes on fact.c1, fact.c2, and fact.c3, the newly generated subqueries can be used to drive a bitmap index access path in the following way. For each value of d1.c1 that is retrieved from the first subquery, the bitmap for that value is retrieved from the index on fact.c1 and these bitmaps are merged. The result is a bitmap for precisely those rows in fact that match the condition on d1 in the subquery WHERE-clause.

Similarly, the values from the second subquery are used together with the bitmap index on fact.c2 to produce a merged bitmap corresponding to the rows in fact that match the condition on d2 in the second subquery. The same operations apply to the third subquery. The three merged bitmaps can then be ANDed, resulting in a bitmap corresponding to those rows in fact that meet the conditions in all three subqueries simultaneously.

This bitmap can be used to access fact and retrieve the relevant rows. These are then joined to d1, d2, and d3 to produce the answer to the query. No Cartesian product is needed.

Execution Plan

The following execution plan might result from the query above:

```
SELECT STATEMENT
 HASH JOIN
  HASH JOIN
   HASH JOIN
    TABLE ACCESS          FACT     BY ROWID
     BITMAP CONVERSION              TO ROWIDS
      BITMAP AND
       BITMAP MERGE
        BITMAP KEY ITERATION
         TABLE ACCESS      D3      FULL
         BITMAP INDEX      FACT_C3 RANGE SCAN
       BITMAP MERGE
        BITMAP KEY ITERATION
         TABLE ACCESS      D1      FULL
         BITMAP INDEX      FACT_C1 RANGE SCAN
       BITMAP MERGE
        BITMAP KEY ITERATION
         TABLE ACCESS      D2      FULL
         BITMAP INDEX      FACT_C2 RANGE SCAN
    TABLE ACCESS           D1      FULL
   TABLE ACCESS            D2      FULL
  TABLE ACCESS             D3      FULL
```

In this plan the fact table is accessed through a bitmap access path based on a bitmap AND of three merged bitmaps. The three bitmaps are generated by the BITMAP MERGE row source being fed bitmaps from row source trees underneath it. Each such row source tree consists of a BITMAP KEY ITERATION row source which fetches values from the subquery row source tree, which in this example is just a full table access, and for each such value retrieves the bitmap from the bitmap index. After the relevant fact table rows have been retrieved using this access path, they are joined with the dimension tables to produce the answer to the query.

The star transformation is a cost-based transformation in the following sense. The optimizer generates and saves the best plan it can produce without the transformation. If the transformation is enabled, the optimizer then tries to apply it to the query and if applicable, generates the best plan using the transformed query. Based on a comparison of the cost estimates between the best plans for the two versions of the query, the optimizer will then decide whether to use the best plan for the transformed or untransformed version.

If the query requires accessing a large percentage of the rows in the fact table, it may well be better to use a full table scan and not use the tranformations. However, if the constraining predicates on the dimension tables are sufficiently selective that only a small portion of the fact table needs to be retrieved, the plan based on the transformation will probably be superior.

Note that the optimizer will generate a subquery for a dimension table only if it decides that it is reasonable to do so based on a number of criteria. There is no guarantee that subqueries will be generated for all dimension tables. The optimizer may also decide, based on the properties of the tables and the query, that the transformation does not merit being applied to a particular query. In this case the best regular plan will be used.

Using Star Transformation

You enable star transformation by setting the value of the initialization parameter STAR_TRANSFORMATION_ENABLED to TRUE. Use the STAR_TRANSFORMATION hint to make the optimizer use the best plan in which the transformation has been used.

Restrictions on Star Transformation

Star transformation is not supported for tables with any of the following characteristics:

- tables with a table hint that is incompatible with a bitmap access path
- tables with too few bitmap indexes (there must be a bitmap index on a fact table column for the optimizer to consider generating a subquery for it)
- remote tables (however, remote dimension tables are allowed in the subqueries that are generated)
- anti-joined tables
- tables that are already used as a dimension table in a subquery
- tables that are really unmerged views, which are not view partitions
- tables that have a good single-table access path
- tables that are too small for the transformation to be worthwhile

Selected Bibliography

- [ARIES] C. Mohan, et al.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging., TODS 17(1): 94-162 (1992).
- [CACHE] C. Mohan: Caching Technologies for Web Applications, A Tutorial at the Conference on Very Large Databases (VLDB), Rome, Italy, 2001.
- [CODASYL] ACM: CODASYL Data Base Task Group April 71 Report, New York, 1971.
- [CODD] E. Codd: A Relational Model of Data for Large Shared Data Banks. ACM 13(6):377-387 (1970).
- [EBXML] http://www.ebxml.org.
- [FED] J. Melton, J. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, K. Zeidenstein: SQL and Management of External Data', SIGMOD Record 30(1):70-77, 2001.
- [GRAY] Gray, et al.: Granularity of Locks and Degrees of Consistency in a Shared Database., IFIP Working Conference on Modelling of Database Management Systems, 1-29, AFIPS Press.
- [INFO] P. Lyman, H. Varian, A. Dunn, A. Strygin, K. Swearingen: How Much Information? at http://www.sims.berkeley.edu/research/projects/how-much-info/.
- [LIND] B. Lindsay, et. al: Notes on Distributed Database Systems. IBM Research Report RJ2571, (1979).

LESSON 27
TRANSCATION PROCESSING

Hi! In this chapter I am going to discuss with you about Transcation processing.

What is Concurrency?

Concurrency in terms of databases means allowing multiple users to access the data contained within a database at the same time. If concurrent access is not managed by the Database Management System (DBMS) so that simultaneous operations don't interfere with one another problems can occur when various transactions interleave, resulting in an inconsistent database.

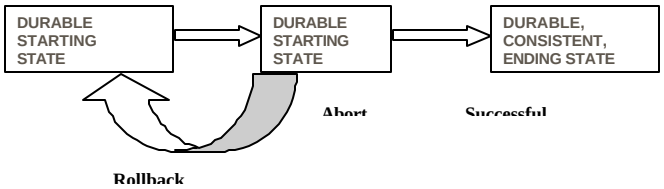
Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions. Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.

Concurrent execution of user programs is essential for good DBMS performance. Because disk accesses are frequent, and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently. Interleaving actions of different user programs can lead to inconsistency: e.g., check is cleared while account balance is being computed. DBMS ensures such problems don't arise: users can pretend they are using a single-user system.

Define Transaction

A transaction is a sequence of read and write operations on data items that logically functions as one unit of work

- It should either be done entirely or not at all
- If it succeeds, the effects of write operations persist (commit); if it fails, no effects of write operations persist (abort)
- These guarantees are made despite concurrent activity in the system, and despite failures that may occur



ACID Properties of Transaction

- **Atomic**

  Process all of a transaction or none of it; transaction cannot be further subdivided (like an atom)

- **Consistent**

  Data on all systems reflects the same state

- **Isolated**

  Transactions do not interact/interfere with one another; transactions act as if they are independent

- **Durable**

  Effects of a completed transaction are persistent

Concurrent Execution

You know there are good reasons for allowing concurrency:-

1. Improved throughput and resource utilization. (THROUGHPUT = Number of Transactions executed per unit of time.)

The CPU and the Disk can operate in parallel. When a Transaction

Read/Write the Disk another Transaction can be running in the CPU.

The CPU and Disk utilization also increases.

2. Reduced Waiting Time.

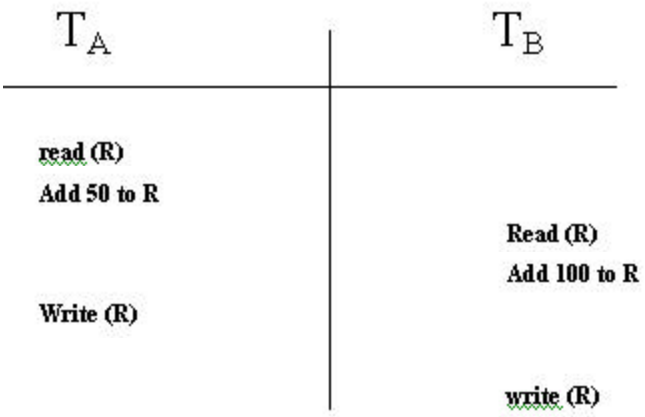
In a serial processing a short Transaction may have to wait for a long transaction to complete. Concurrent execution reduces the average response time; the average time for a Transaction to be completed.

What is Concurrency Control?

Concurrency control is needed to handle problems that can occur when transactions execute concurrently. The following are the concurrency issues:-

Lost Update: an update to an object by some transaction is overwritten by another interleaved transaction without knowledge of the initial update.

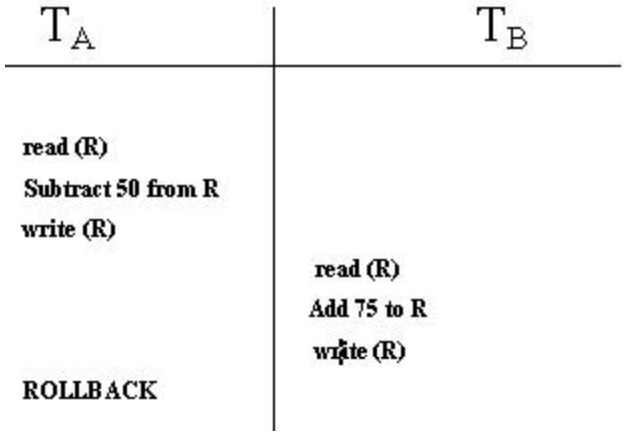
Lost Update Example



Transaction A's update is lost

Uncommitted Dependency: a transaction reads an object updated by another transaction that later falls.

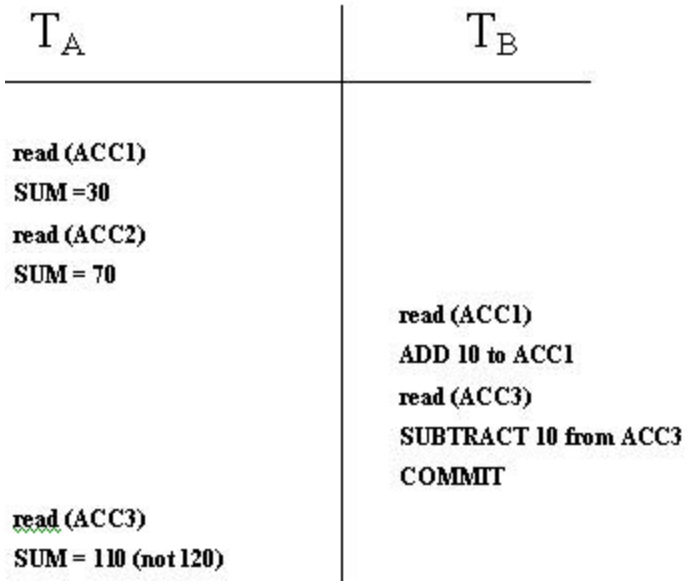
Uncommitted Dependency Example:-



Transaction B reads an uncommitted value for R

Inconsistent Analysis: a transaction calculating an aggregate function uses some but not all updated objects of another transaction.

Inconsistent Analysis Example:-



The value in SUM will be inconsistent

Main goals of Database Concurrency Control

- To point out problem areas in earlier performance analyses
- To introduce queuing network models to evaluate the baseline performance of transaction processing systems
- To provide insights into the relative performance of transaction processing systems
- To illustrate the application of basic analytic methods to the performance analysis of various concurrency control methods
- To review transaction models which are intended to relieve the effect of lock contention
- To provide guidelines for improving the performance of transaction processing systems due to concurrency control; and to point out areas for further investigation.

In the last lecture we discussed about concurrency and transactions. Here I would like to discuss with you in detail regarding the transaction management and concurrency control.

Transaction Properties

To ensure data integrity the DBMS, should maintain the following transaction properties- atomicity, consistency, isolation and durability. These properties often referred to as **acid properties** an acronym derived from the first letter of the properties. In the last lecture we have introduced the above terms, now we will see their implementations.

We will consider the banking example to gain a better understanding of the acid properties and why are they important. We will consider a banking system that contains several accounts and a set of transactions that accesses and updates accounts. Access to a database is accomplished by two operations given below:-

1. Read(x)-This operation transfers the data item x from the database to a local buffer belonging to the transaction that executed the read operation
2. Write(x)-the write operation transfers the data item x from the local buffer of the transaction that executed the write operation to the database.

Now suppose that Ti is a transaction that transfers RS. 2000/- from account CA2090 to SB2359. This transaction is defined as follows:-

```
Ti:
Read(CA2090);
CA2090:=CA2090-2000;
Write(CA2090);
Read(SB2359);
SB2359:=SB2359+2000;
Write(SB2359);
```

We will now consider the acid properties..

Implementing Atomicity

Let's assume that before the transaction take place the balances in the account is Rs. 50000/- and that in the account SB2359 is Rs. 35000/-. Now suppose that during the execution of the transaction a failure(for example, a power failure) occurred that prevented the successful completion of the transaction. The failure occurred after the Write(CA2090); operation was executed, but before the execution of Write(SB2359); in this case the value of the accounts CA2090 and SB2359 are reflected in the database are Rs. 48,000/- and Rs. 35000/- respectively. The Rs. 200/- that we have taken from the account is lost. Thus the failure has created a problem. The state of the database no longer reflects a real state of the world that the database is supposed to capture. Such a state is called an inconsistent state. The database system should ensure that such inconsistencies are not visible in a database system. It should be noted that even during the successful execution of a transaction there exists points at which the system is in an inconsistent state. But the

difference in the case of a successful transaction is that the period for which the database is in an inconsistent state is very short and once the transaction is over the system will be brought back to a consistent state. So if a transaction never started or is completed successfully, the inconsistent states would not be visible except during the execution of the transaction. This is the reason for the atomicity requirement. If the atomicity property provided all actions of the transaction are reflected in the database of none are. The mechanism of maintaining atomicity is as follows The DBMS keeps tracks of the old values of any data on which a transaction performs a Write and if the transaction does not complete its execution, old values are restored o make it appear as though the transaction never took place. The transaction management component of the DBMS ensures the atomicity of each transaction.

Implementing Consistencies

The consistency requirement in the above eg is that the sum of CA2090 and SB2359 be unchanged by the execution of the transaction. Before the execution of the transaction the amounts in the accounts in CA2090 and SB2359 are 50,000 and 35,000 respectively. After the execution the amounts become 48,000 and 37,000. In both cases the sum of the amounts is 85,000 thus maintaining consistency. Ensuring the consistency for an individual transaction is the responsibility of the application programmer who codes the transaction.

Implementing the Isolation

Even if the atomicity and consistency properties are ensured for each transaction there can be problems if several transactions are executed concurrently. The different transactions interfere with one another and cause undesirable results. Suppose we are executing the above transaction Ti. We saw that the database is temporarily inconsistent while the transaction is being executed. Suppose that the transaction has performed the Write(CA2090) operation, during this time another transaction is reading the balances of different accounts. It checks the account CA2090 and finds the account balance at 48,000.

Suppose that it reads the account balance of the other account(account SB2359, before the first transaction has got a chance to update the account.

So the account balance in the account Sb2359 is 35000. After the second transaction has read the account balances, the first transaction reads the account balance of the account SB2359 and updates it to 37000.But here we are left with a problem. The first transaction has executed successfully and the database is back to a consistent state. But while it was in an inconsistent state, another transaction performed some operations(May be updated the total account balances). This has left the database in an inconsistent state even after both the transactions have been executed successfully. On solution to the situation(concurrent execution of transactions) is to execute the transactions serially-one after the other. This can create many problems. Suppose long transactions are being executed first. Then all other transactions will have to wait in the queue. There might be many transactions that are independent(or that do not interfere with one another). There is no need for such transactions to wait in the queue. Also concurrent executions of transactions have significant performance advantages. So the DBMS have found solutions to allow multiple transactions to execute concurrency with out any problem. The isolation property of a transaction ensures that the concurrent execution of transactions result in a system state that is equivalent to a state that could have been obtained if the transactions were executed one after another. Ensuring isolation property is the responsibility of the concurrency-control component of the DBMS.

Implementing Durability

The durability property guarantees that, once a transaction completes successfully, all updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution. We can guarantee durability by ensuring that either the updates carried out by the transaction have been written to the disk before the transaction completes or information about the updates that are carried out by the transaction and written to the disk are sufficient for the data base to reconstruct the updates when the data base is restarted after the failure. Ensuring durability is the responsibility of the recovery- management component of the DBMS

Picture

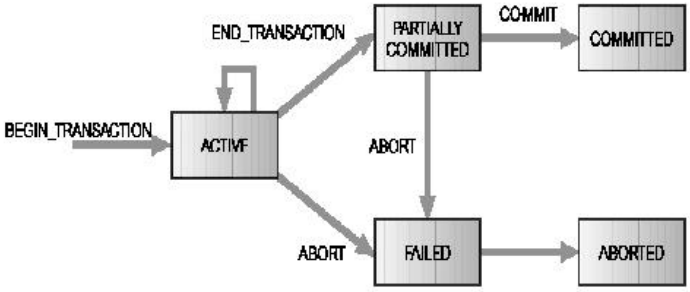
Transaction management and concurrency control components of a DBMS

Transaction States

Once a transaction is committed, we cannot undo the changes made by the transactions by rolling back the transaction. Only way to undo the effects of a committed transaction is to execute a compensating transaction. The creating of a compensating transaction can be quite complex and so the task is left to the user and it is not handled by the DBMS.

The transaction must be in one of the following states:-

1. active:- This is a initial state, the transaction stays in this state while it is executing
2. Partially committed:- The transaction is in this state when it has executed the final statement
3. Failed: - A transaction is in this state once the normal execution of the transaction cannot proceed.
4. Aborted: - A transaction is said to be aborted when the transaction has rolled back and the database is being restored to the consistent state prior to the start of the transaction.
5. Committed: - a transaction is in this committed state once it has been successfully executed and the database is transformed in to a new consistent state. Different transactions states arte given in following figure.



State transition diagram for a Transaction.

Concurrency Control

Concurrency control in database management systems permits many users (assumed to be interactive) to access a database in a multiprogrammed environment while preserving the illusion that each user has sole access to the system. Control is needed to coordinate concurrent accesses to a DBMS so that the overall correctness of the database is maintained. For example, users *A* and *B* both may wish to read and update the same record in the database at about the same time. The relative timing of the two transactions may have an impact on the state of the database at the end of the transactions. The end result may be an inconsistent database.

Why Concurrent Control is needed?

- Several problems can occur when concurrent transactions execute in an uncontrolled manner.
  - The lost update problem : This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of same database item incorrect.
  - The temporary update (or dirty read) problem : This occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.
  - The incorrect summary problem : If one transaction is calculating an aggregate function on a number of records while other transaction is updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.
- Whenever a transaction is submitted to a DBMS for execution, the system must make sure that :
  - All the operations in the transaction are completed successfully and their effect is recorded permanently in the database; or
  - The transaction has no effect whatever on the database or on the other transactions in the case of that a transaction fails after executing some of operations but before executing all of them.

Review Questions

1. Why Concurrent Control is needed?
2. Main goals of Database Concurrency Control.
3. What is concurrency?

Notes:

LESSON 28
ATOMICITY, CONSISTENCY, INDEPENDENCE AND DURABLITY(ACID) PRINCIPLE

Hi! In this chapter I am going to discuss with you about ACID property.

Introduction

Most modern computer systems, except the personal computers and many workstations, are multiuser systems. Mulitple users are able to use a single process simultaneously because of multiprogramming in which the processor (or processors) in the system is shared amongst a number of processes trying to access computer resources (including databases) simultaneously. The concurrent execution of programs is therefore interleaved with each program being allowed access to the CPU at regular intervals. This also enables another program to access the CPU while a program is doing I/O. In this chapter we discuss the problem of synchronization of access to shared objects in a database while supporting a high degree of concurrency.

Concurrency control in database management systems permits many users (assumed to be interactive) to access a database in a multiprogrammed environment while preserving the illusion that each user has sole access to the system. Control is needed to coordinate concurrent accesses to a DBMS so that the overall correctness of the database is maintained. Efficiency is also important since the response time for interactive users ought to be short. The reader may have recognised that the concurrency problems in database management systems are related to the concurrency problems in operating systems although there are significant differences between the two. For example, operating systems only involve concurrent sharing of resources while the DBMS must deal with a number of users attempting to concurrently access and modify data in the database.

Clearly no problem arises if all users were accessing the database only to retrieve information and no one was modifying it, for example, accessing the census data or a library catalogue. If one or more of the users were modifying the database e.g. a bank account or airline reservations, an update performed by one user may interfere with an update or retrieval by another user. For example, users *A* and *B* both may wish to read and update the same record in the database at about the same time. The relative timing of the two transactions may have an impact on the state of the database at the end of the transactions. The end result may be an inconsistent database.

Our discussion of concurrency will be transaction based. A transaction is a sequence of actions $[t_1, t_2, \ldots, t_n]$. As noted in the last chapter, a transaction is a unit of consistency in that it preserves database consistency. We assume that all transactions complete successfully; problems of transactions failures are resolved by the recovery mechanisms. The only detail of transactions that interests us right now is their reads and writes although other computation would often be carried out between the reads and writes. We therefore assume that all actions t_i that form a transaction are either a read or a write. The set of items read by a transaction are called its *read set* and the set of items written by it are called its *write set*. Two transactions T_i and T_j are said to *conflict* if some action t_i of T_i and an action t_j of T_j access the same object and at least one of the actions is a write. Two situations are possible:

1. The write set of one transaction intersects with the read set of another. The result of running the two transactions concurrently will clearly depend on whether the write is done first or the read is. The conflict is called a *RW-conflict*;
2. The write set of one transaction intersects with the write set of another. Again, the result of running the two transactions concurrently will depend on the order of the two writes. The conflict is called a *WW-conflict*.

A concurrency control mechanism must detect such conflicts and control them. Various concurrency control mechanisms are available. The mechanisms differ in the time they detect the conflict and the way they resolve it. We will consider the control algorithms later. First we discuss some examples of concurrency anomalies to highlight the need for concurrency control.

We have noted already that in the discussion that follows we will ignore many details of a transaction. For example, we will not be concerned with any computations other than the READs and the WRITEs and whether results of a READ are being stored in a local variable or not.

ACID Properties

ACID properties are an important concept for databases. The acronym stands for Atomicity, Consistency, Isolation, and Durability.

The ACID properties of a DBMS allow safe sharing of data. Without these ACID properties, everyday occurrences such using computer systems to buy products would be difficult and the potential for inaccuracy would be huge. Imagine more than one person trying to buy the same size and color of a sweater at the same time - a regular occurrence. The ACID properties make it possible for the merchant to keep these sweater purchasing transactions from overlapping each other - saving the merchant from erroneous inventory and account balances.

Atomicity

The phrase “all or nothing” succinctly describes the first ACID property of atomicity. When an update occurs to a database, either all or none of the update becomes available to anyone beyond the user or application performing the update. This update to the database is called a transaction and it either commits or aborts. This means that only a fragment of the update cannot be placed into the database, should a problem occur with either the hardware or the software involved. Features to consider for atomicity: a transaction is a unit of

operation - either all the transaction's actions are completed or none are

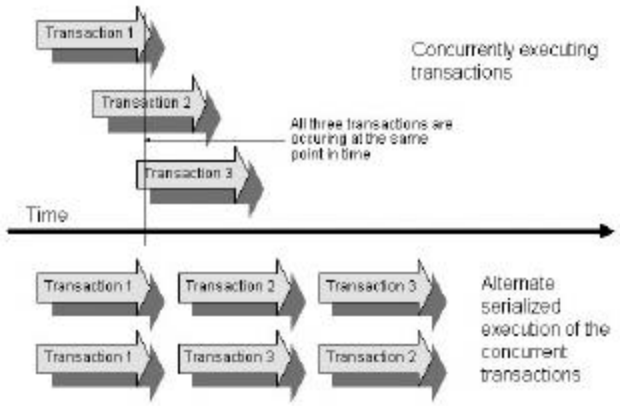
- atomicity is maintained in the presence of deadlocks
- atomicity is maintained in the presence of database software failures
- atomicity is maintained in the presence of application software failures
- atomicity is maintained in the presence of CPU failures
- atomicity is maintained in the presence of disk failures
- atomicity can be turned off at the system level
- atomicity can be turned off at the session level

Consistency

Consistency is the ACID property that ensures that any changes to values in an instance are consistent with changes to other values in the same instance. A consistency constraint is a predicate on data which server as a precondition, post-condition, and transformation condition on any transaction

Isolation

The isolation portion of the ACID properties is needed when there are concurrent transactions. Concurrent transactions are transactions that occur at the same time, such as shared multiple users accessing shared objects. This situation is illustrated at the top of the figure as activities occurring over time. The safeguards used by a DBMS to prevent conflicts between concurrent transactions are a concept referred to as isolation.



As an example, if two people are updating the same catalog item, it's not acceptable for one person's changes to be "clobbered" when the second person saves a different set of changes. Both users should be able to work in isolation, working as though he or she is the only user. Each set of changes must be isolated from those of the other users.

An important concept to understanding isolation through transactions is serializability. Transactions are serializable when the effect on the database is the same whether the transactions are executed in serial order or in an interleaved fashion. As you can see at the top of the figure, Transactions 1 through Transaction 3 are executing concurrently over time. The effect on the DBMS is that the transactions may execute in serial order based on consistency and isolation requirements. If you look at the bottom of the figure, you can see several ways in which these transactions may execute. It is important to note that a serialized execution does not imply the first transactions will automatically be the ones that will terminate before other transactions in the serial order.

Degrees of isolation[1]

- degree 0 - a transaction does not overwrite data updated by another user or process ("dirty data") of other transactions
- degree 1 - degree 0 plus a transaction does not commit any writes until it completes all its writes (until the end of transaction)
- degree 2 - degree 1 plus a transaction does not read dirty data from other transactions
- degree 3 - degree 2 plus other transactions do not dirty data read by a transaction before the transaction commits

[1] These were originally described as *degrees of consistency* by Jim Gray. The following book provides excellent, updated coverage of the concept of isolation along with other transaction concepts

Durability

Maintaining updates of committed transactions is critical. These updates must never be lost. The ACID property of durability addresses this need. Durability refers to the ability of the system to recover committed transaction updates if either the system or the storage media fails. Features to consider for durability:

- recovery to the most recent successful commit after a database software failure
- recovery to the most recent successful commit after an application software failure
- recovery to the most recent successful commit after a CPU failure
- recovery to the most recent successful backup after a disk failure
- recovery to the most recent successful commit after a data disk failure

· **Examples of Concurrency Anomalies**

  - Lost Updates
  - Inconsistent Retrievals
  - Uncommitted Dependency

Examples of Concurrency Anomalies

There are three classical concurrency anomalies. These are lost updates, inconsistent retrievals and uncommitted dependency.

- Lost Updates
- Inconsistent Retrievals
- Uncommitted Dependency

Lost Updates

Suppose two users A and B simultaneously access an airline database wishing to reserve a number of seats on a flight. Let us assume that A wishes to reserve five seats while B wishes to reserve four seats. The reservation of seats involves booking the seats and then updating the number of seats available (N)

on the flight. The two users read-in the present number of seats, modify it and write back resulting in the following situation:

A	Time	B
Read N	1	-
-	2	Read N
N := N-5	3	-
-	4	N := N-4
Write N	5	
-	6	Write N

Figure 1 An Example of Lost Update Anomaly

If the execution of the two transactions were to take place as shown in Fugure 1, the update by transaction A is lost although both users read the number of seats, get the bookings confirmed and write the updated number back to the database. This is called the lost update anomaly since the effects of one of the transactions were lost.

Inconsistent Retrievals

Consider two users A and B accessing a department database simultaneously. The user A is updating the database to give all employees in the department a 5% raise in their salary while the other user wants to know the total salary bill of the department. The two transactions interfere since the total salary bill would be changing as the first user updates the employee records. The total salary retrieved by the second user may be a sum of some salaries before the raise and others after the raise. This could not be considered an acceptable value of the total salary but the value before the raise or the value after the raise is acceptable.

A	Time	B
Read Employee 100	1	-
-	2	Sum = 0.0
Update Salary	3	-
	4	Read Employee 100
Write Employee 100	5	-
-	6	Sum = Sum + Salary
Read Employee 101	7	-
-	8	-
Update Salary	9	
Write Employee 101	10	-
	11	
-	12	Read Employee 101
	13	Sum = Sum + Salary
etc	-	etc

Figure 2 An Example of Inconsistent Retrieval

The problem illustrated in the last example is called the inconsistent retrieval anomaly. During the execution of a transaction therefore, changes made by another transaction that has not yet committed should not be visible since that data may not be consistent.

Uncommitted Dependency

In the last chapter on recovery we discussed a recovery technique that involves immediate updates of the database and the maintenance of a log for recovery by rolling back the transaction in case of a system crash or transaction failure. If this technique was being used, we could have the following situation:

A	Time	B
-	1	Read Q
-	2	-
-	3	Write Q
Read Q	4	-
-	5	Read R
-	6	-
Write Q	7	-
-	8	Failure (rollback)
-	9	-

Figure 3 An Example of Uncommitted Dependency

Transaction A has now read the value of Q that was updated by transaction B but was never committed. The result of Transaction A writing Q therefore will lead to an inconsistent state of the database. Also if the transaction A doesn't write Q but only reads it, it would be using a value of Q which never really existed! Yet another situation would occur if the roll back happens after Q is written by transaction A. The roll back would restore the old value of Q and therefore lead to the loss of updated Q by transaction A. This is called the uncommitted dependency anomaly.

We will not discuss the problem of uncommitted dependency any further since we assume that the recovery algorithm will ensure that transaction A is also rolled back when B is. The most serious problem facing concurrency control is that of the lost update. The reader is probably already thinking of some solutions to the problem. The commonly suggested solutions are:

1. **Once transaction A reads a record Q for an update, no other transaction is allowed to read it until transaction A update is completed. This is usually called locking.**
2. **Both transaction A and B are allowed to read the same record Q but once A has updated the record, B is not allowed to update it as well since B would now be updating an old copy of the record Q. B must therefore read the record again and then perform the update.**
3. **Although both transactions A and B are allowed to read the same record Q, A is not allowed to update the record because another transaction (Transaction B) has the old value of the record.**
4. **Partition the database into several parts and schedule concurrent transactions such that each transaction uses a different partition. There is thus no conflict and database stays consistent. Often this is not feasible**

since most databases have some parts (called hot spots) that most transactions want to access.

These are in fact the major concurrency control techniques. We discuss them in detail later in this chapter. We first need to consider the concept of serializability which deals with correct execution of transactions concurrently. [phantoms???]

Review Question

1. Define ACID Property

Selected Bibliography

- [ARIES] C. Mohan, et al.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging., TODS 17(1): 94-162 (1992).
- [CACHE] C. Mohan: Caching Technologies for Web Applications, A Tutorial at the Conference on Very Large Databases (VLDB), Rome, Italy, 2001.
- [CODASYL] ACM: CODASYL Data Base Task Group April 71 Report, New York, 1971.
- [CODD] E. Codd: A Relational Model of Data for Large Shared Data Banks. ACM 13(6):377-387 (1970).
- [EBXML] http://www.ebxml.org.
- [FED] J. Melton, J. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, K. Zeidenstein: SQL and Management of External Data', SIGMOD Record 30(1):70-77, 2001.
- [GRAY] Gray, et al.: Granularity of Locks and Degrees of Consistency in a Shared Database., IFIP Working Conference on Modelling of Database Management Systems, 1-29, AFIPS Press.
- [INFO] P. Lyman, H. Varian, A. Dunn, A. Strygin, K. Swearingen: How Much Information? at http://www.sims.berkeley.edu/research/projects/how-much-info/.
- [LIND] B. Lindsay, et. al: Notes on Distributed Database Systems. IBM Research Report RJ2571, (1979).

Notes:

LESSON 29
CONCURRENCY ANOMALIES

Following are the problems created due to the concurrent execution of the transactions:-

Multiple Update Problems

In this problem, the data written by one transaction (an update operation) is being overwritten by another update transaction. This can be illustrated using our banking example. Consider our account CA2090 that has Rs. 50000 balance in it. Suppose a transaction T1 is withdrawing RS. 10000 fro the account while another transaction T2 is depositing RS. 20000 to the account. If these transactions were executed serially (one after another), the final balance would be Rs. 60000, irrespective of the order in which the transactions are performed. In other words, if the transactions were performed serially, then the result would be the sameif T1 is performed first or T2 is performed first-order is not important. But idf the transactions are performed concurrently, then depending on how the transactions are executed the results will vary. Consider the execution of the transactions given below

Sequence	T1	T2	Account Balance
01	Begin Transaction		50000
02	Read (CA2090)	Begin Transaction	50000
03	CA2090:=CA 2090-10000	Read (CA2090)	50000
04	Write (CA2090)	CA2090:=CA209 0+20000	40000
05	Commit	Write (CA2090)	70000
06		Commit	70000

Both transactions start nearly at the same time and both read the account balance of 50000. Both transactions perform the operations that they are supposed to perform-T1 will reduce the amount by 10000and will write the result to the data base; T2 will increase the amount by 20000 and will write the amount to the database overwriting the previous update. Thus the account balance will gain additional 10000 producing a wrong result. If T2 were to start execution first, the result would have been 40000 and the result would have been wrong again.

This situation could be avoided by preventing T2 from reading the value of the account balance until the update by T1 has been completed.

Incorrect Analysis Problem

Problems could arise even when a transaction is not updating the database. Transactions that read the database can also produce wrong result, if they are allowed to read the database when the database is in an inconsistent state. This problem is often referred to as **dirty read** or **unrepeatable data**. The problem of dirty read occurs when a transaction reads several values from the data base while another transactions are updating the values.

Consider the case of the transaction that reads the account balances from all accounts to find the total amount in various account. Suppose that there are other transactions, which are updating the account balances-either reducing the amount (withdrawals) or increasing the amount (deposits). So when the first transaction reads the account balances and finds the totals, it will be wrong, as it might have read the account balances before the update in the case of some accounts and after the updates in other accounts. This problem is solved by prevent-ing the first transaction (the one that reads the balances) from reading the account balances until all the transactions that update the accounts are completed.

Inconsistent Retrievals

Consider two users A and B accessing a department database simultaneously. The user A is updating the database to give all employees a 5% salary raise while user B wants to know the total salary bill of a department. The two transactions interfere since the total salary bill would be changing as the first user updates the employee records. The total salary retrieved by the second user may be a sum of some salaries before the raise and others after the raise. Such a sum could not be considered an acceptable value of the total salary (the value before the raise or after the raise would be).

A	Time	B
Read Employee 100	1	-
-	2	Sum = 0.0
Update Salary	3	-
-	4	Read Employee 100
Write Employee 100	5	-
-	6	Sum = Sum + Salary
Read Employee 101	7	-
-	8	Read Employee 101
Update Salary	9	-
-	10	Sum = Sum + Salary
Write Employee 101	11	-
-	12	-
-	13	-
etc	-	etc

Figure 2. An Example of Inconsistent Retrieval

The problem illustrated in the last example is called the inconsistent retrieval anomaly.
During the execution of a transaction therefore, changes made by another transaction that has not yet committed should not be visible since that data may not be consistent.

Uncommitted Dependency

Consider the following situation:

A	Time	B
-	1	Read Q
-	2	-
-	3	Write A
Read Q	4	-
-	5	Read R
-	6	-
Write Q	7	-
-	8	Failure (rollback)
-	9	-

Figure 3. An Example of Uncommitted Dependency

Transaction A reads the value of Q that was updated by transaction B but was never committed. The result of Transaction A writing Q therefore will lead to an inconsistent state of the database. Also if the transaction A doesn't write Q but only reads it, it would be using a value of Q which never really existed! Yet another situation would occur if the roll back happens after Q is written by transaction A. The roll back would restore the old value of Q and therefore lead to the loss of updated Q by transaction A. This is called the uncommitted dependency anomaly.

Serializability

Serializability is a given set of interleaved transactions is said to be serial sable if and only if it produces the same results as the serial execution of the same transactions

Serializability is an important concept associated with locking. It guarantees that the work of concurrently executing transactions will leave the database state as it would have been if these transactions had executed serially. This requirement is the ultimate criterion for database consistency and is the motivation for the two-phase locking protocol, which dictates that no new locks can be acquired on behalf of a transaction after the DBMS releases a lock held by that transaction. In practice, this protocol generally means that locks are held until commit time.

Serializability is the classical concurrency scheme. It ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order. It assumes that all accesses to the database are done using read and write operations. A schedule is called "correct" if we can find a serial schedule that is "equivalent" to it. Given a set of transactions T1...Tn, two schedules S1 and S2 of these transactions are equivalent if the following conditions are satisfied:

Read-Write Synchronization: If a transaction reads a value written by another transaction in one schedule, then it also does so in the other schedule.

Write-Write Synchronization: If a transaction overwrites the value of another transaction in one schedule, it also does so in the other schedule.

These two properties ensure that there can be no difference in the effects of the two schedules

Serializable Schedule

A schedule is serial if, for every transaction T participating the schedule, all the operations of T are executed consecutively in the schedule. Otherwise it is called non-serial schedule.

- Every serial schedule is considered correct; some nonserial schedules give erroneous results.
- A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions; a nonserial schedule which is not equivalent to any serial schedule is not serializable.
- The definition of two schedules considered "equivalent":
  - result equivalent: producing same final state of the database (is not used)
  - conflict equivalent: If the order of any two conflicting operations is the same in both schedules.
  - view equivalent: If each read operation of a transaction reads the result of the same write operation in both schedules and the write operations of each transaction must produce the same results.
- Conflict serializable: if a schedule S is conflict equivalent to some serial schedule. we can reorder the non-conflicting operations S until we form the equivalent serial schedule, and S is a serializable schedule.
- View Serializability: Two schedules are said to be view equivalent if the following three conditions hold. The same set of transactions participate in S and S'; and S and S' include the same operations of those transactions. A schedule S is said to be view serializable if it is view equivalent to a serial schedule.

Testing for Serializability

Since a serializable schedule is a correct schedule, we would like the DBMS scheduler to test each proposed schedule for serializability before accepting it. Unfortunately most concurrency control method do not test for serializability since it is much more time-consuming task than what a scheduler can be expected to do for each schedule. We therefore resort to a simpler technique and develop a set of simple criteria or protocols that all schedules will be required to satisfy. These criteria are not necessary for serializability but they are sufficient. Some techniques based on these criteria are discussed in Section 4.

There is a simple technique for testing a given schedule *S* for conflict serializability. The testing is based on constructing a directed graph in which each of the transactions is represented by one node and an edge between T_i and T_j exists if any of the following conflict operations appear in the schedule:

1. T_i executes WRITE(X) before T_j executes READ(X), or
2. T_i executes READ(X) before T_j executes WRITE(X)
3. T_i executes WRITE(X) before T_j executes WRITE(X).

[Needs fixing] Three possibilities if there are two transactions T_i, T_j that interfere with each other.

This is not serializable since is has a cycle.

Basically this graph implies that T_i ought to happen before T_j and T_j ought to happen before T_i - an impossibility. If there is no cycle in the precedence graph, it is possible to build an equivalent serial schedule by traversing the graph.
The above conditions are derived from the following argument. Let T_i and T_j both access X and T_j consist of either a Read(X) or Write(X). If T_j access is a Read(X) then there is conflict only if T_i had a Write(X) and if T_i did have a Write(X) then T_i must come before T_j. If however T_j had a Write(X) and T_i had a Read(X) (T_i would have a Read(X) even if it had a Write(X)) then T_i must come before T_j. [Needs fixing]

Enforcing Serializability

As noted earlier, a schedule of a set of transactions is serializable if computationally its effect is equal to the effect of some serial execution of the transactions. One way to enforce serializability is to insist that when two transactions are executed, one of the transactions is assumed to be older than the other. Now the only schedules that are accepted are those in which data items that are common to the two transactions are seen by the junior transaction after the older transaction has written them back. The three basic techniques used in concurrency control (locking, timestamping and optimistic concurrency control) enforce this in somewhat different ways. The only schedules that these techniques allow are those that are serializable. We now discuss the techniques.

Recoverability

So far we have studied what schedules are acceptable from the viewpoint of consistency of the database, assuming implicitly that there are no transaction failures. We now address the effect of transaction failures during concurrent execution.

If a transaction T_i fails, for whatever reasons, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction T_j that is dependent on T_i (that is T_j has read data written by T_i) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system

No we are going to look at what schedules are acceptable from the view point of recovery from transaction failure.

Recoverable Schedules

A recoverable schedule is one where, for each pair of transaction T_i and $T_{j \text{ such that}}$ T_j reads a data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j.

Cascade less Schedules

Even if a schedule is recoverable, to recover correctly from the failure of transaction T_i, we may have to rollback several transactions. This phenomenon, in which a single transaction failure which leads to a series of transaction rollbacks, is called **cascading rollbacks.**

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called *cascadeless* schedules. Then we can say that a Cascadeless Schedule is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i., the commit operation of T_i appears before the read operation of T_j. It is easy to verify that every cascadeless schedule is also recoverable

Transaction Definition in SQL

A data manipulation language must include a construct for specifying the set of actions that constitute a transaction. The SQL standard specifies that a transaction begins implicitly. Transactions are ended with one of these SQL statements.

- Commit work - commit the current transaction and begins a new one
- Rollback work - causes the current transaction to abort.

The keyword is optional. If a program terminates without either of these commands, the updates are either committed or rolled back-which of the two happens is not specified by the standard and depends on the implementation.

Review Questions

1. Explain the various transaction properties?
2. Why concurrency is needed?
3. What are the various transaction states?
4. Explain the implementation of acid properties?

References

Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000

Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld

Notes:

LESSON 30
SERIALIZABILITY

Hi! In this chapter I am going to discuss with you about Serialization.

The concept of a transaction has been discussed in the last chapter. A transaction must have the properties of atomicity, consistency, isolation and durability. In addition, when more than one transaction are being executed concurrently, we must have serializability.

When two or more transactions are running concurrently, the steps of the transactions would normally be interleaved. The interleaved execution of transactions is decided by the database system software called the scheduler which receives a stream of user requests that arise from the active transactions. A particular sequencing (usually interleaved) of the actions of a set of transactions is called a schedule. A serial schedule is a schedule in which all the operations of one transaction are completed before another transaction can begin (that is, there is no interleaving). Later in the section we will show several different schedules of the same two transactions.

It is the responsibility of the scheduler to maintain consistency while allowing maximum concurrency. The scheduler therefore must make sure that only correct schedules are allowed. We would like to have a mechanism or a set of conditions that define a correct schedule. That is unfortunately not possible since it is always very complex to define a consistent database. The assertions defining a consistent database (called integrity constraints or consistency constraints) could well be as complex as the database itself and checking these assertions, assuming one could explicitly enumerate them, after each transaction would not be practical. The simplest approach to this problem is based on the notion of each transaction being correct by itself and a schedule of concurrent executions being able to preserve their correctness.

A serial schedule is always correct since we assume transactions do not depend on each other. We further assume that each transaction when run in isolation transforms a consistent database into a new consistent state and therefore a set of transactions executed one after another (i.e. serially) must also be correct. A database may however not be consistent during the execution of a transaction but it is assumed the database is consistent at the end of the transaction. Although using only serial schedules to ensure consistency is a possibility, it is often not a realistic approach since a transaction may at times be waiting for some input/output from secondary storage or from the user while the CPU remains idle, wasting a valuable resource. Another transaction could have been running while the first transaction is waiting and this would obviously improve the system efficiency. We therefore need to investigate schedules that are not serial. However, since all serial schedules are correct, interleaved schedules that are equivalent to them must also be considered correct. There are in fact n! different serial schedules for a set of any n transactions. Note that not all serial schedules of the same set of transactions result in the same consistent state of the database. For example, a seat reservation system may result in different allocations of seats for different serial schedules although each schedule will ensure that no seat is sold twice and no request is denied if there is a free seat available on the flight. However one serial schedule could result in 200 passengers on the plane while another in 202 as shown below.

Let us consider an airline reservation system. Let there be 12 seats available on flight QN32. Three requests arrive; transaction T_1 for 3 seats, transaction T_2 for 5 seats and transaction T_3 for 7 seats. If the transactions are executed in the order $\{T_1, T_2, T_3\}$ then we allocate 8 seats but cannot meet the third request since there are only 4 seats left after allocating seats for the first two transactions. If the transactions are executed in the order $\{T_2, T_3, T_1\}$, we allocate all the 12 remaining seats but the request of transaction T_1 cannot be met. If the transactions are instead executed in the order $\{T_3, T_1, T_2\}$ then we allocate 10 seats but are unable to meet the request of Transaction T_2 for 5 seats. In all there are 3!, that is 6, different serial schedules possible. The remaining three are $\{T_1, T_3, T_2\}$, $\{T_2, T_1, T_3\}$, and $\{T_3, T_2, T_1\}$. They lead to 10, 8 and 12 seats being sold respectively. All the above serial schedules must be considered correct although any one of the three possible results may be obtained as a result of running these transactions.

We have seen that some sets of transactions when executed concurrently may lead to problems, for example, lost update anomaly. Of course, any sets of transactions can be executed concurrently without leading to any difficulties if the read and write sets of these transactions do not intersect. We want to discuss sets of transactions that do interfere with each other. To discuss correct execution of such transactions, we need to define the concept of serializability.

Let *T* be a set of *n* transactions $T_1, T_2, \ldots, T_n$. If the n transactions are executed serially (call this execution *S*), we assume they terminate properly and leave the database in a consistent state. *A concurrent execution of the n transactions in T* (call this execution C) is called *serializable* if the *execution is computationally equivalent to a serial execution*. There may be more than one such serial execution. That is, the concurrent execution *C* always produces exactly the same effect on the database as some serial execution *S* does where *S* is some serial execution of *T*, not necessarily the order $T_1, T_2, \ldots, T_n$. Thus if a transaction T_i writes a data item *A* in the interleaved schedule *C*

before another transaction T_j reads or writes the same data item, the schedule *C* must be equivalent to a serial schedule in which T_i appears before T_j. Therefore in the interleaved transaction T_i appears logically before it T_j does; same as in the equivalent serial schedule. The concept of serializability defined here is sometimes called final state serializability; other forms of serializability exist. The final state serializability has sometimes been criticized for being too strict while at other times it has been criticized for being not strict enough! Some of these criticisms are valid though and it is therefore necessary to provide another definition of serializability called view serializability. Two schedules are called view serializable if the order of any two conflicting operations in the two schedules is the same. As discussed earlier, two transactions may have a RW-conflict or a WW-conflict when they both access the same data item. Discussion of other forms of serializability is beyond the scope of these notes and the reader is referred to the book by Papadimitriou (1986).

If a schedule of transactions is not serializable, we may be able to overcome the concurrency problems by modifying the schedule so that it serializable. The modifications are made by the database scheduler.

We now consider a number of examples of possible schedules of two transactions running concurrently. Consider a situation where a couple has three accounts (A, B, C) with a bank. The husband and the wife maintain separate personal savings accounts while they also maintain a loan account on their house. A is the husband's account, B is the wife's account and C is the housing loan. Each month a payment of $500 is made to the account C. To make the payment on this occasion, the couple walk to two adjoining automatic teller machines and the husband transfers $200 from account A to C (Transaction 1) while the wife on the other machine transfers $300 from account B to C (Transaction 2). Several different schedules for these two transactions are possible. We present the following four schedules for consideration.

Transaction 1	Time	Transaction 2
Read A	1	-
A:=A - 200	2	-
Write A	3	-
Read C	4	-
C := C − 200	5	-
Write C	6	-
-	7	Read B
-	8	B := B - 300
-	9	Write B
-	10	Read C
-	11	C := C + 300
-	12	Write C

Figure 4 A serial schedule without interleaving

Transaction 1	Time	Transaction 2
Read A	1	
A:=A - 200	2	-
Write A	3	-
-	4	Read B
-	5	B := B - 300
-	6	Write B
Read C	7	-
C := C − 200	8	-
Write C	9	-
-	10	Read C
-	11	C := C + 300
-	12	Write C

Figure 5 An interleaved serializable schedule

Transaction 1	Time	Transaction 2
Read A	1	
A:=A - 200	2	-
Write A	3	-
Read C	4	-
-	5	Read B
-	6	B := B - 300
-	7	Write B
-	8	Read C
-	9	C := C + 300
-	10	Write C
C := C + 200	11	-
Write C	12	-

Figure 6 An interleaved non-serializable schedule

Transaction 1	Time	Transaction 2
-	1	Read B
-	2	B := B - 300
-	3	Write B
Read A	4	-
A:=A - 200	5	-
Write A	6	-
Read C	7	-
C := C + 200	8	-
Write C	9	-
-	10	Read C
-	11	C := C + 300
-	12	Write C

Figure 7 Another interleaved serializable schedule

It is therefore clear that many interleaved schedules would result in a consistent state while many others will not. All correct schedules are serializable since they all must be equivalent to either serial schedule (T_1, T_2) or (T_2, T_1).

Testing for Serializability

Since a serializable schedule is a correct schedule, we would like the DBMS scheduler to test each proposed schedule for serializability before accepting it. Unfortunately most concurrency control method do not test for serializability since it is much more time-consuming task than what a scheduler can be expected to do for each schedule. We therefore resort to a simpler technique and develop a set of simple criteria or protocols that all schedules will be required to satisfy. These criteria are not necessary for serializability but they are sufficient. Some techniques based on these criteria are discussed in Section 4.

There is a simple technique for testing a given schedule *S* for conflict serializability. The testing is based on constructing a directed graph in which each of the transactions is represented by one node and an edge between T_i and T_j exists if any of the following conflict operations appear in the schedule:

1. T_i executes WRITE(*X*) before T_j executes READ(*X*), or
2. T_i executes READ(*X*) before T_j executes WRITE(*X*)
3. T_i executes WRITE(*X*) before T_j executes WRITE(*X*).

[Needs fixing] Three possibilities if there are two transactions T_i, T_j that interfere with each other.

This is not serializable since is has a cycle.

Basically this graph implies that T_i ought to happen before T_j and T_j ought to happen before T_i - an impossibility. If there is no cycle in the precedence graph, it is possible to build an equivalent serial schedule by traversing the graph.

The above conditions are derived from the following argument. Let T_i and T_j both access *X* and T_j consist of either a Read(X) or Write(X). If T_j access is a Read(X) then there is conflict only if T_i had a Write(X) and if T_i did have a Write(X) then T_i must come before T_j. If however T_j had a Write(X) and T_i had a Read(X) (T_i would have a Read(X) even if it had a Write(X)) then T_i must come before T_j. [Needs fixing]

Enforcing Serializability

As noted earlier, a schedule of a set of transactions is serializable if computationally its effect is equal to the effect of some serial execution of the transactions. One way to enforce serializability is to insist that when two transactions are executed, one of the transactions is assumed to be older than the other. Now the only schedules that are accepted are those in which data items that are common to the two transactions are seen by the junior transaction after the older transaction has written them back. The three basic techniques used in concurrency control (locking, timestamping and optimistic concurrency control) enforce this in somewhat different ways. The only schedules that these techniques allow are those that are serializable. We now discuss the techniques.

Review Question

1. What is serialization and why it is important?

Selected Bibliography

- [ARIES] C. Mohan, et al.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging., TODS 17(1): 94-162 (1992).
- [CACHE] C. Mohan: Caching Technologies for Web Applications, A Tutorial at the Conference on Very Large Databases (VLDB), Rome, Italy, 2001.
- [CODASYL] ACM: CODASYL Data Base Task Group April 71 Report, New York, 1971.
- [CODD] E. Codd: A Relational Model of Data for Large Shared Data Banks. ACM 13(6):377-387 (1970).
- [EBXML] http://www.ebxml.org.
- [FED] J. Melton, J. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, K. Zeidenstein: SQL and Management of External Data', SIGMOD Record 30(1):70-77, 2001.
- [GRAY] Gray, et al.: Granularity of Locks and Degrees of Consistency in a Shared Database., IFIP Working Conference on Modelling of Database Management Systems, 1-29, AFIPS Press.
- [INFO] P. Lyman, H. Varian, A. Dunn, A. Strygin, K. Swearingen: How Much Information? at http://www.sims.berkeley.edu/research/projects/how-much-info/.
- [LIND] B. Lindsay, et. al: Notes on Distributed Database Systems. IBM Research Report RJ2571, (1979).

Notes:

LESSON 31
LOCK-I

Hi! In this chapter I am going to discuss with you about Locks in DBMS.

Locking

Locking is a common technique by which a database may synchronize execution of concurrent transactions. Using locking a transaction can obtain exclusive or shareable access rights (called *locks*) to an object. If the access provided by the lock is shareable, the lock is often called a *shared lock* (sometime called a *read* lock). On the other hand if the access is exclusive, the lock is called an *exclusive lock* (sometime called a *write* lock). If a number of transactions need to read an object and none of them wishes to write that object, a shared lock would be the most appropriate. Of course, if any of the transactions were to write the object, the transaction must acquire an exclusive lock on the object to avoid the concurrency anomalies that we have discussed earlier. A number of locking protocols are possible. Each protocol consists of

1. A set of locking types
2. A set of rules indicating what locks can be granted concurrently
3. A set of rules that transactions must follow when acquiring or releasing locks

We will use the simplest approach that uses shared and exclusive locks, a transaction would set a read lock on the data item that it reads and exclusive lock on the data item that it needs to update. As the names indicate, a transaction may obtain a shared lock on a data item even if another transaction is holding a shared lock on the same data item at the same time. Of course, a transaction cannot obtain a shared or exclusive lock on a data item if another transaction is holding an exclusive lock on the data item. The shared lock may be upgraded to an exclusive lock (assuming no other transaction is holding a shared lock on the same data item at that time) for items that the transaction wishes to write. This technique is sometime also called *blocking* because if another transaction requests access to an exclusively locked item, the lock request is denied and the requesting transaction is blocked.

A transaction can hold a lock on one or more items of information. The data item must be unlocked before the transaction is completed. The locks are granted by a database system software called the *lock manager* which maintains information on all locks that are active and controls access to the locks. As noted earlier, several modes of locks may be available. For example, in shared mode, a transaction can read the locked data item but cannot update it. In exclusive lock mode, a transaction has exclusive access to the locked data item and no other transaction is allowed access to it until the lock is released. A transaction is allowed to update the data item only as long as an exclusive lock is held on it.

If transactions running concurrently are allowed to acquire and release locks as data item is read and updated, there is a danger that incorrect results may be produced. Consider the following example:

Transaction 1	Time	Transaction 2
XL(A)	1	-
Read (A)	2	-
A:= A - 50	3	-
-	4	SL(B)
Write(A)	5	-
-	6	Read(B)
UL(A)	7	-
	8	UL(B)
-	9	SL(A)
XL(B)	10	-
-	11	Read (A)
Read(B)	12	-
-	13	UL(A)
B:= B + 50	14	-
-	15	Display (A+B)
Write (B)	16	
UL(B)	17	-

Table 8 - Unserializable Schedule using Locks

XL (called exclusively lock) is a request for an exclusive lock and UL is releasing the lock (sometimes called unlock). SL is a request for a shared lock. In the example above, the result displayed by transaction 2 is incorrect because it was able to read *B* before the update but read *A* only after the update. The above schedule is therefore not serializable. As discussed earlier, the problem with above schedule is inconsistent retrieval.

The problem arises because the above schedule involves two RW-conflicts and the first one that involves *A* involves Transaction 1 logically appearing before Transaction 2 while the second conflict involving *B* involves Transaction 2 logically appearing before Transaction 1.

To overcome this problem a *two-phase* locking (*2PL*) scheme is used in which a transaction cannot request a new lock after releasing a lock. Two phase locking involves the following two phases:

1. **Growing Phase (*Locking* Phase)** - During this phase locks may be acquired but not released.
2. **Shrinking Phase (*Unlocking* Phase)** - During this phase locks may be released but not acquired.

In summary, the main feature of locking is that conflicts are checked at the *beginning* of the execution and resolved by *waiting*.

Using the two-phase locking scheme in the above example, we obtain the following schedule by modifying the earlier schedule so that all locks are obtained before any lock is released:

Transaction 1	Time	Transaction 2
XL(A)	1	-
Read (A)	2	-
A:= A - 50	3	-
-	4	SL(B)
Write(A)	5	-
-	6	Read(B)
XL(B)	7	-
UL(A)	8	-
-	9	SL(A)
-	10	UL(B)
-	11	Read (A)
Read(B)	12	-
-	13	UL(A)
B:= B + 50	14	-
-	15	Display (A+B)
Write (B)	16	
UL(B)	17	-

Table 9 - A Schedule using Two-Phase Locking

Unfortunately, although the above schedule would not lead to incorrect results, it has another difficulty: both the transactions are blocked waiting for each other; a classical deadlock situation has occurred! Deadlocks arise because of circular wait conditions involving two or more transactions as above. A system that does not allow deadlocks to occur is called *deadlock-free*. In two-phase locking, we need a deadlock detection mechanism and a scheme for resolving the deadlock once a deadlock has occurred. We will look at such techniques in Section 5.

The attraction of the two-phase algorithm derives from a theorem which proves that the two-phase locking algorithm always leads to serializable schedules that are equivalent to serial schedules in the order in which each transaction acquires its last lock. This is a sufficient condition for serializability although it is not necessary.

To show that conditions of serializability are met if 2PL is used we proceed as follows. We know that a test of serializability requires building a directed graph in which each of the transactions is represented by one node and an edge between T_i and T_j exists if any of the following conflict operations appear in the schedule:

1. T_i executes WRITE(X) before T_j executes READ(X), or
2. T_i executes READ(X) before T_j executes WRITE(X)
3. T_i executes WRITE(X) before T_j executes WRITE(X).

A schedule will not be serializable if a cycle is found in the graph. We assume a simple concurrent execution of two transactions and assume that the graph has a cycle. The cycle is possible only if there is a edge between T_i and T_j because one of the above conditions was met followed by the schedule meeting one of the following conditions:

1. T_j executes WRITE(Y) before T_i executes READ(Y), or
2. T_j executes READ(Y) before T_i executes WRITE(Y)
3. T_j executes WRITE(Y) before T_i executes WRITE(Y).

These two sets of conditions provide a total of nine different combinations of conditions which could lead to a cycle. It can be shown that none of these combinations are possible. For example, assume the following two conditions have been satisfied by a schedule:

1. T_i executes WRITE(X) before T_j executes READ(X),
2. T_j executes WRITE(Y) before T_i executes WRITE(Y).

For these conditions to have met, T_i must have had an XL(X) which it would release allowing T_j to acquire SL(X). Also, for the second condition to have met, T_j must have had an XL(Y) which it would release allowing T_i to acquire XL(Y). This is just not possible using 2PL. Similarly, it can be shown that none of the nine combinations of conditions are possible if 2PL is used.

Several versions of two-phase locking have been suggested. Two commonly used versions are called the *static* two-phase locking and the *dynamic* two-phase locking. The static technique basically involves locking all items of information needed by a transaction before the first step of the transaction and unlocking them all at the end of the transaction. The dynamic scheme locks items of information needed by a transaction only immediately before using them. All locks are released at the end of the transaction. [which is better??]

To use the locking mechanism, the nature and size of the individual data item that may be locked must be decided upon. A lockable data item could be as large as a relation or as small as an attribute value of a tuple. The lockable item could also be of a size in between, for example, a page of a relation or a single tuple. Larger the size of the items that are locked, the smaller the concurrency that is possible since when a transaction wants to access one or more tuples of a relation, it is allowed to lock the whole relation and some other transaction wanting to access some other tuples of the relation is denied access. On the other hand, coarse granularity reduces the overhead costs of locking since the table that maintains information on the locks is smaller. Fine granularity of the locks involving individual tuple locking or locking of even smaller items allows greater concurrency at the cost of higher overheads. Therefore there is a tradeoff between level of concurrency and overhead costs. Perhaps a system can allow variable granularity so that in some situations, for example when computing the projection of a relation, the whole relation may be locked while in other situations only a small part of the relation needs locking.

Timestamping Control

The two-phase locking technique relies on locking to ensure that each interleaved schedule executed is serializable. Now we discuss a technique that does not use locks and works quite well when level of contention between transactions running concurrently is not high. It is called Timestamping.

Timestamp techniques are based on assigning a unique *timestamp* (a number indicating the time of the start of the transaction) for each transaction at the start of the transaction and insisting that the schedule executed is always serializable to the serial schedule in the chronological order of the timestamps. This is in contrast to two-phase locking where any schedule that is equivalent to some serial schedule is acceptable.

Since the scheduler will only accept schedules that are serializable to the serial schedule in the chronological order of the timestamps, the scheduler must insist that in case of conflicts, the junior transaction must process information only after the older transaction has written them. The transaction with the smaller timestamp being the older transaction. For the scheduler to be able to carry this control, the data items also have read and write timestamps. The read timestamp of a data item X is the timestamp of the youngest transaction that has read it and the write timestamp is the timestamp of the youngest transaction that has written it. Let timestamp of a transaction be TS(T).

Consider a transaction T_1 with timestamp = 1100. Suppose the smallest unit of concurrency control is a relation and the transaction wishes to read or write some tuples from a relation named *R*. Let the read timestamp of *R* be *RT(R)* and write timestamp be *WT(R)*. The following situations may then arise:

1. T_1 wishes to read *R*. The read by T_1 will be allowed only if $WT(R) < TS(T_1)$ or $WT(R) < 1100$ that is the last write was by an older transaction. This condition ensures that data item read by a transaction has not been written by a younger transaction. If the write timestamp of *R* is larger than 1100 (that is, a younger transaction has written it), then the older transaction is rolled back and restarted with a new timestamp.
2. T_1 wishes to read *R*. The read by T_1 will be allowed if it satisfies the above condition even if $RT(R) > TS(T_1)$, that is the last read was by a younger transaction. Therefore if data item has been read by a younger transaction that is quite acceptable.
3. T_1 wishes to write some tuples of a relation *R*. The write will be allowed only if read timestamp $RT(R) < TS(T_1)$ or $RT(R) < 1100$, that is the last read of the transaction was by an older transaction and therefore no younger transaction has read the relation.
4. T_1 wishes to write some tuples of a relation *R*. The write need not be carried out if the above condition is met and if the last write was by a younger transaction, that is, $WT(R) > TS(T_1)$. The reason this write is not needed is that in this case the younger transaction has not read *R* since if it had, the older transaction T_1 would have been aborted.

If the data item has been read by a younger transaction, the older transaction is rolled back and restarted with a new timestamp. It is not necessary to check the write (??) since if a younger transaction has written the relation, the younger transaction would have read the data item and the older transaction has old data item and may be ignored. It is possible to ignore the write if the second condition is violated since the transaction is attempting to write obsolete data item. [expand!]

Let us now consider what happens when an attempt is made to implement the following schedule:

???

We assume transaction T_1 to be older and therefore $TS(T_1) < TS(T_2)$.

Read (A) by transaction T_1 is permitted since A has not been written by a younger transaction. Write (A) by transaction T_1 is also permitted since A has not been read or written by a younger transaction. Read (B) by transaction T_2 is permitted since *B* has not been written by a younger transaction. Read (B) by transaction T_1 however is not allowed since *B* has been read by (younger) transaction T_2 and therefore transaction T_1 is rolled back. [needs to be read again and fixed]

Refer to page 383 of Korth and Silberchatz for a schedule that is possible under the timestamping control but not under 2PL

Deadlocks

A deadlock may be defined as a situation in which each transaction in a set of two or more concurrently executing transactions is blocked circularly waiting for another transaction in the set, and therefore none of the transactions will become unblocked unless there is external intervention.

A deadlock is clearly undesirable but often deadlocks are unavoidable. We therefore must deal with the deadlocks when they occur. There are several aspects of dealing with deadlocks: one should prevent them if possible (deadlock prevention), detect them when they occur (deadlock detection) and resolve them when a deadlock has been detected (deadlock resolution). Deadlocks may involve two, three or more transactions. To resolve deadlocks, one must keep track of what transactions are waiting and for which transaction they are waiting for.

Once a deadlock is identified, one of the transactions in the deadlock must be selected and rolled back (such transaction is called a victim) thereby releasing all the locks that that transaction held breaking the deadlock.

Since locking is the most commonly used (and often the most efficient) technique for concurrency control, we must deal with

deadlocks. The deadlocks may either be prevented or identified when they happen and then resolved. We discuss both these techniques.

Deadlock Prevention

Deadlocks occur when some transactions wait for other transactions to release a resource and the wait is circular (in that T_1 waits for T_2 which is waiting for T_4 which in turn is waiting for T_1 or in the simplest case T_1 waits for T_2 and T_2 for T_1). Deadlocks can be prevented if circular waits are eliminated. This can be done either by defining an order on who may wait for whom or by eliminating all waiting. We first discuss two techniques that define an ordering on transactions. [Further info .. Ullman page 373]

Wait-die Algorithm

When a conflict occurs between T_1 and T_2 (T_1 being the older transaction), if T_1 possesses the lock then T_2 (the younger one) is not allowed to wait. It must rollback and restart. If however T_2 possessed the lock at conflict, the senior transaction is allowed to wait. The technique therefore avoids cyclic waits and generally avoids starvations (a transaction's inability to secure resources that it needs). A senior transaction may however find that it has to wait for every resource that it needs, and it may take a while to complete.

Wound-die Algorithm

To overcome the possibility of senior transaction having to wait for every item of data that it needs, the wound-die scheme allows a senior transaction to immediately acquire a data item that it needs and is being controlled by a younger transaction. The younger transaction is then restarted.

Immediate Restart

In this scheme, no waiting is allowed. If a transaction requests a lock on a data item that is being held by another transaction (younger or older), the requesting transaction is restarted immediately. This scheme can lead to starvation if a transaction requires several popular data items since every time the transaction restarts and seeks a new item, it finds some data item to be busy resulting in it being rolled back and restarted.

Deadlock Detection

As noted earlier, when locking is used, one needs to build and analyse a waits-for graph every time a transaction is blocked by a lock. This is called *continuous detection*. One may prefer a *periodic detection* scheme although results of experiments seem to indicate that the continuous scheme is more efficient.

Once a deadlock has been detected, it must be resolved. The most common resolution technique requires that one of the transactions in the waits-for graph be selected as a *victim* and be rolled back and restarted. The victim may be selected as

1. randomly
2. the youngest transaction since this is likely to have done the least amount of work.
3. the one that has written the least data back to the database since all the data written must be undone in the rollback.
4. the one that is likely to affect the least number of other transactions. That is, the transaction whose rollback will lead to least other rollbacks (in cascade rollback).
5. the one with the fewest locks.

One experimental study has suggested that the victim with the fewest locks provides the best performance. Also it has been suggested that if deadlock prevention with immediate restart is to be used, the database performance suffers.

Evaluation of Control Mechanisms

A number of researchers have evaluated the various concurrency control mechanisms that have been discussed here. It is generally accepted that a technique that minimizes the number of restarts (like the two-phase locking technique) is superior when the number of conflicts is high. The timestamping and the optimistic algorithms usually performed much worse under high conflict levels since they wasted the most resources through restarts. Deadlock prevention techniques like those based on immediate restart tend to make the database slow down somewhat. One therefore must have deadlock detection technique in place. At low loads, it doesn't really matter much which technique is used although some studies show that the optimistic techniques then are more efficient. It is of course possible to use more than one technique in a system. For example, two-phase locking may be combined with optimistic technique to produce hybrid techniques that work well in many situations.

Review Question

1. What is two phase locking
2. Define
   - Page locking
   - Cluster locking
   - Class or table locking
   - Object or instance locking

Source

1. Agarwal, R. and DeWitt, D. (1985), "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation", ACM TODS, Vol. 10, pp. 529-564.
2. Agarwal, R., Carey, M. J. and McWoy, L. (1988?), "The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems", IEEE Trans. Software Engg., pp. ??
3. Agarwal, R., Carey, M. J. and Livny, M. (1987), "Concurrency Control Performance Modeling: Alternatives and Implications", ACM TODS, Vol. 12, pp. 609-654.
4. Bernstein, P. A. and Goodman, N. (1980), "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems", Proc VLDB, Oct. 1980, pp. 285-300.

5. Bernstein, P. A. and Goodman N. (1981), "Concurrency Control in Distributed Database Systems", ACM Computing Surveys, June 1981, pp. 185-222.
6. Carey, M. and Muhanna, W. (1986), "The Performance of Multiversion Concurrency Control Algorithms", In ACM Trans. Comp. Syst. Vol 4, pp. 338-378.
7. Carey, M. and Stonebraker, M. (1984), "The Performance of Concurrency Control Algorithms for Database Management Systems", In VLDB 1984, pp. 107-118.
8. Coffman, E., Elphich, M., Shoshani, A. (1971), "System Deadlocks", ACM Computing Surveys, 2, 3, pp. 67-78.
9. Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L. (1976), "The Notions of Consistency and Predicate Locks in a Database System", Comm. ACM, 10, 11, pp. 624-633.
10. Gray, J. (1978), "Notes on Data Base Operating Systems", in *Operating Systems: An Advanced Course*, Eds. R. Bayer, R. M. Graham and G. Seegmuller, Springer-Verlag.
11. Gray, J. (1981), "The Transaction Concept: Virtues and Limitations", Proc VLDB, Sept 1981.
12. Gray, J. N., Lorie, R. A. and Putzolu, G. R.(1975), "Granularity of Locks in a Large Shared Data Base", Proc VLDB, Sept 1975.
13. Gray, J. N., Lorie, R. A., Putzolu, G. R. and Traiger, I. L. (1976), "Granularity of Locks and Degrees of Consistency in a Shared Data Base", in Proc. IFIP TC-2 Working Conference on Modelling in Data Base Management Systems, Ed. G. M. Nijssen, North-Holland.
14. Kung, H. T. and Papadimitriou, C. H. (1979), "An Optimality Theory of Concurrency Control in Databases", Proc ACM SIGMOD, pp. ???
15. Kung, H. T. and Robinson, J. T. (1981), "On Optimistic Methods for Concurrency Control", ACM TODS, Vol. 6, pp.??
16. Ries, D. R. and Stonebraker, M. R.(1977), "Effects of Locking Granularity in a Database Management System", ACM TODS, Vol 2, pp.??
17. Papadimitriou, C. H. (1986), "The Theory of Database Concurrency Control", Computer Science Press.

Notes:

LESSON 32
LOCK-II

Hi! In this chapter I am going to discuss with you about Locks in DBMS in great detail.

As you know now that Concurrency control and locking is the mechanism used by DBMSs for the sharing of data. Atomicity, consistency, and isolation are achieved through concurrency control and locking.

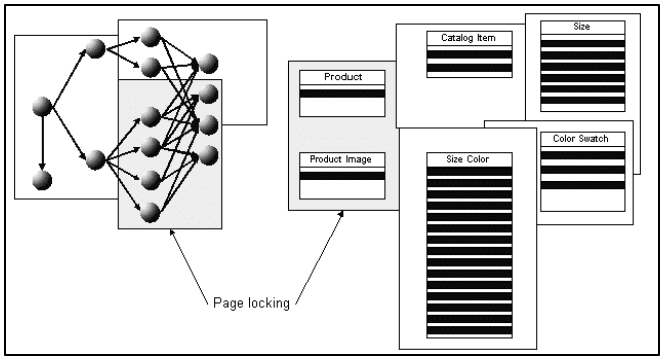
When many people may be reading the same data item at the same time, it is usually necessary to ensure that only one application at a time can change a data item. Locking is a way to do this. Because of locking, all changes to a particular data item will be made in the correct order in a transaction.

The amount of data that can be locked with the single instance or groups of instances defines the granularity of the lock. The types of granularity are illustrated here are:

- Page locking
- Cluster locking
- Class or table locking
- Object or instance locking

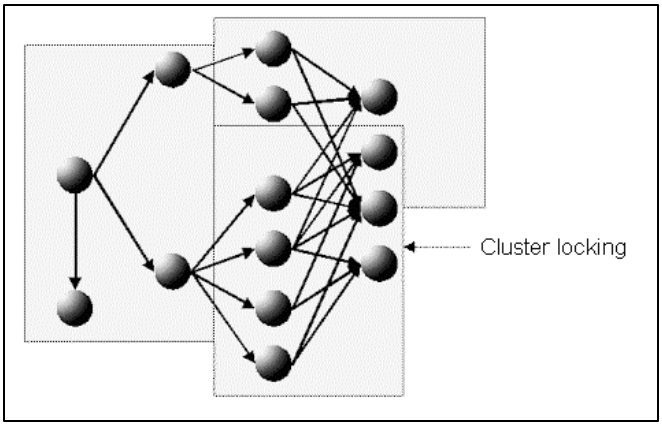
Page Locking

Page locking (or page-level locking) concurrency control is shown in the figure below. In this situation, all the data on a specific page are locked. A page is a common unit of storage in computer systems and is used by all types of DBMSs. In this figure, each rectangle represents a page. Locking for objects is on the left and page locking for relational tuples is on the right. If the concept of pages is new to you, just think of a page as a unit of space on the disk where multiple data instances are stored.



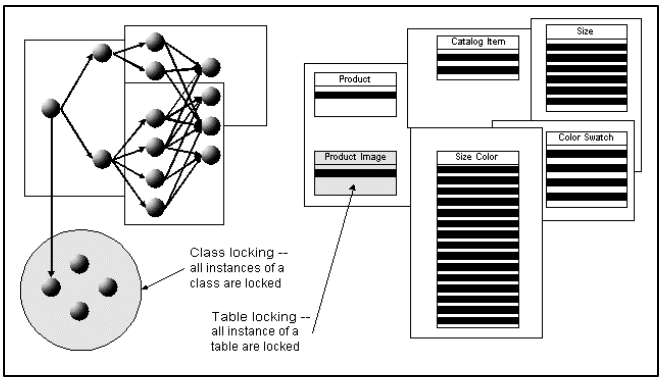
Cluster Locking

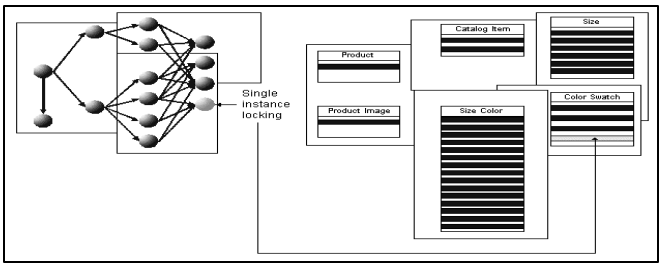
Cluster locking or container locking for concurrency control is illustrated in the figure below. In this form of locking, all data clustered together (on a page or multiple pages) will be locked simultaneously. This applies only to clusters of objects in ODBMSs. Note that in this example, the cluster of objects spans portions of three pages.



Class or Table locking

Class or table locking means that all instances of either a class or table are locked, as is illustrated below. This shows one form of concurrency control. Note the circle at the lower left. It represents all instances of a class, regardless of the page where they are stored.





Two Phase Locking

A two-phase locking (2PL) scheme is a locking scheme in which a transaction cannot request a new lock after releasing a lock. Two phases locking therefore involves two phases:

- Growing Phase (Locking Phase) - When locks are acquired and none released.
- Shrinking Phase (Unlocking Phase) - When locks are released and none acquired.

The attraction of the two-phase algorithm derives from a theorem which proves that the two-phase locking algorithm

always leads to serializable schedules. This is a sufficient condition for Serializability although it is not necessary.

Timestamp Ordering Protocol

In the previous chapter of locking we have studied about the 2 phase locking system. Here I would like to discuss with you certain other concurrency control schemes and protocols.

The timestamp method for concurrency control does not need any locks and therefore there are no deadlocks. Locking methods generally prevent conflicts by making transaction to walk. Timestamp methods do not make the transactions wait. Transactions involved in a conflict are simply rolled back and restarted. A timestamp is a unique identifier created by the DBMS that indicates the relative starting time of a transaction. Timestamps are generated either using the system clock (generating a timestamp when the transaction starts to execute) or by incrementing a logical counter every time a new transaction starts.

Time stamping is the concurrency control protocol in which the fundamentals goal is to order transactions globally in such away that older transactions get priority in the event of a conflict. In the Timestamping method, if a transaction attempts to read or write a data item, then a read or write operation is allowed only if the last update on that data item was carried out by an older transaction. Otherwise the transaction requesting the read or write is restarted and given a new timestamp to prevent it from continually aborting and restarting. If the restarted transaction is not allowed a new timestamp and is allowed a new timestamp and is allowed to retain the old timestamp, it will never be allowed to perform the read or write, because by that some other transaction which has a newer timestamp than the restarted transaction might not be to commit due to younger transactions having already committed.

In addition to the timestamp for the transactions, data items are also assigned timestamps. Each data item contains a read-timestamp and write-timestamp. The read-timestamp contains the timestamp of the last transaction that read the item and the write-timestamp contains the timestamp of the last transaction that updated the item. For a transaction T the timestamp ordering protocol works as follows:

- Transactions T requests to read the data item 'X' that has already been updated by a younger (later or one with a greater timestamp) transaction. This means that an earlier transactions is trying to read a data item that has been updated by a later transaction T is too late to read the previous outdated value and any other values it has acquired are likely to be inconsistent with the updated value of the data item. In this situation, the transaction T is aborted and restarted with a new timestamp.
- In all other cases, the transaction is allowed to proceed with the read operation. The read-timestamp of the data item is updated with the timestamp of transaction T.
- Transaction t requests to write (update) the data item 'X' that has already been read by a younger (later or one with the greater timestamp) transaction. This means that the younger transaction is already using the current value of the data item and it would be an error to update it now. This situation occurs when a transaction is late in performing the write and a younger transaction has already read the old value or written a new one. In this case the transaction T is aborted and is restarted with a new timestamp.
- Transaction T asks to write the data item 'X' that has already been written by a younger transaction. This means that the transaction T is attempting to write an old or obsolete value of the data item. In this case also the transaction T is aborted and is restarted with a new timestamp.
- In all other cases the transaction T is allowed to proceed and the write-timestamp of the data item is updated with the timestamp of transaction T.

The above scheme is called basic timestamp ordering. This scheme guarantees that the transactions are conflict serializable and the results are equivalent to a serial schedule in which the transactions are executed in chronological order by the timestamps. In other words, the results of a basic timestamps ordering scheme will be as same as when all the transactions were executed one after another without any interleaving.

One of the problems with basic timestamp ordering is that it does not guarantee recoverable schedules. A modification to the basic timestamp ordering protocol that relaxes the conflict Serializability can be used to provide greater concurrency by rejecting obsolete write operations. This extension is known as Thomas's write rule. Thomas's write rule modifies the checks for a write operation by transaction T as follows.

- When the transaction T requests to write the data item 'X' whose values has already been read by a younger transaction. This means that the order transaction (transaction T) is writing an obsolete value to the data item. In this case the write operation is ignored and the transaction (transaction T) is allowed to continue as if the write were performed. This principle is called the 'ignore obsolete write rule'. This rule allows for greater concurrency.
- In all other cases the transactions T is allowed to proceed and the write-timestamp of transaction T.

Thus the use of Thomas's write rule allows us to generate schedules that would not have been possible under other concurrency protocols.

Time stamping Control-Contrast to 2PL

The two-phase locking technique relies on locking to ensure that each interleaved schedule executed is serializable. Now we discuss a technique that does not use locks and works quite well when level of contention between transactions running concurrently is not high. It is called Time stamping.

Timestamp techniques are based on assigning a unique *timestamp* (a number indicating the time of the start of the transaction) for each transaction at the start of the transaction and insisting that the schedule executed is always serializable to the serial schedule in the chronological order of the timestamps. This is in contrast to two-phase locking where any schedule that is equivalent to some serial schedule is acceptable.

Since the scheduler will only accept schedules that are serializable to the serial schedule in the chronological order of the

timestamps, the scheduler must insist that in case of conflicts, the junior transaction must process information only after the older transaction has written them. The transaction with the smaller timestamp being the older transaction. For the scheduler to be able to carry this control, the data items also have read and write timestamps. The read timestamp of a data item X is the timestamp of the youngest transaction that has read it and the write timestamp is the timestamp of the youngest transaction that has written it. Let timestamp of a transaction be TS(T).

Consider a transaction T_1 with timestamp = 1100. Suppose the smallest unit of concurrency control is a relation and the transaction wishes to read or write some tuples from a relation named *R*. Let the read timestamp of *R* be *RT(R)* and write timestamp be *WT(R)*. The following situations may then arise:

1. T_1 wishes to read *R*. The read by T_1 will be allowed only if $WT(R) < TS(T_1)$ or $WT(R) < 1100$ that is the last write was by an older transaction. This condition ensures that data item read by a transaction has not been written by a younger transaction. If the write timestamp of *R* is larger than 1100 (that is, a younger transaction has written it), then the older transaction is rolled back and restarted with a new timestamp.
2. T_1 wishes to read *R*. The read by T_1 will be allowed if it satisfies the above condition even if $RT(R) > TS(T_1)$, that is the last read was by a younger transaction. Therefore if data item has been read by a younger transaction that is quite acceptable.
3. T_1 wishes to write some tuples of a relation *R*. The write will be allowed only if read timestamp $RT(R) < TS(T_1)$ or $RT(R) < 1100$, that is the last read of the transaction was by an older transaction and therefore no younger transaction has read the relation.
4. T_1 wishes to write some tuples of a relation *R*. The write need not be carried out if the above condition is met and if the last write was by a younger transaction, that is, $WT(R) > TS(T_1)$. The reason this write is not needed is that in this case the younger transaction has not read *R* since if it had, the older transaction T_1would have been aborted.

If the data item has been read by a younger transaction, the older transaction is rolled back and restarted with a new timestamp. It is not necessary to check the write (??) since if a younger transaction has written the relation, the younger transaction would have read the data item and the older transaction has old data item and may be ignored. It is possible to ignore the write if the second condition is violated since the transaction is attempting to write obsolete data item. [expand!]

Let us now consider what happens when an attempt is made to implement the following schedule:

We assume transaction T_1 to be older and therefore $TS(T_1) < TS(T_2)$.

Read (A) by transaction T_1 is permitted since A has not been written by a younger transaction. Write (A) by transaction T_1 is also permitted since A has not been read or written by a younger transaction. Read (B) by transaction T_2 is permitted since *B* has not been written by a younger transaction. Read (B) by transaction T_1 however is not allowed since *B* has been read by (younger) transaction T_2 and therefore transaction T_1 is rolled back. [needs to be read again and fixed]

Optimistic Control

This approach is suitable for applications where the number of conflicts between the transactions is small. The technique is unsuitable for applications like the airline reservations system where write operations occur frequently at “hot spots”, for example, counters or status of the flight. In this technique, transactions are allowed to execute unhindered and are validated only after they have reached their commit points. If the transaction validates, it commits, otherwise it is restarted. For example, if at the time of validation, it is discovered that the transaction wanting to commit had read data item that has already been written by another transaction, the transaction attempting to commit would be restarted. [Needs more]

These techniques are also called validation or certification techniques.

The optimistic control may be summarized as involving checking at the end of the execution and resolving conflicts by rolling back. It is possible to combine two or more control techniques - for example, very heavily used items could be locked, others could follow optimistic control.

Review Questions

1. Explain the types of granularity?
2. Explain the timestamp ordering protocol?
3. Explain the Timestamping control?
4. Explain optimistic control?

References

Date, C, J, Introduction to Database Systems, 7th edition

Silbershatz, Korth, Sudarshan, Database System Concepts 4th Edition.

LESSON 33
BACKUP AND RECOVERY-I

Hi! In this chapter I am going to discuss with you about Backup and Recovery.

Introduction

An enterprise's database is a very valuable asset. It is therefore essential that a DBMS provide adequate mechanisms for reducing the chances of a database failure and suitable procedures for recovery when the system does fail due to a software or a hardware problem. These procedures are called *recovery techniques*.

In this chapter, as before, we assume that we are dealing with a multiple user database system and not a single user system that are commonly used on personal computers. The problems of recovery are much simpler in a single user system than in a multiple user system.

Before we discuss causes of database failures and the techniques to recover from them, we should note that in our discussion in this chapter we shall assume that the database is resident on a disk and is transferred from the disk to the main memory when accessed. Modifications of data are initially made in the main memory and later propagated to the disk. Disk storage is often called *nonvolatile storage* because the information stored on a disk survives system crashes like processor and power failure. Non-volatile storage may be destroyed by errors like a head crash but that happens only infrequently. A part of the database needs to be resident in the main memory. Main memory is *volatile storage* because the contents of such storage are lost with a power failure. Given that a database on disk may also be lost, it is desirable to have a copy of the database stored in what is called *stable storage.* Stable storage is assumed to be reliable and unlikely to lose information with any type of system failures. No media like disk or tape can be considered stable, only if several independent nonvolatile storage media are used to replicate information can we achieve something close to a stable storage.

A database system failure may occur due to:

1. *Power failures* - perhaps the most common cause of failure. These result in the loss of the information in the main memory.
2. *Operating system or DBMS failure* - this often results in the loss of the information in the main memory.
3. *User input errors* - these may lead to an inconsistent data base.
4. *Hardware failure (including disk head crash)* - hardware failures generally result in the loss of the information in the main memory while some hardware failures may result in the loss of the information on the disk as well.
5. *Other causes like operator errors, fire, flood, etc.* - some of these disasters can lead to loss of all information stored in a computer installation.

A system failure may be classified as *soft-fail* or a *hard-fail*. Soft-fail is a more common failure involving only the loss of information in the volatile storage. A hard-fail usually occurs less frequently but is harder to recover from since it may involve loss or corruption of information stored on the disk.

Definition - Failure

A failure of a DBMS occurs when the database system does not meet its specification i.e. the database is in an inconsistent state.

Definition - Recovery

Recovery is the restoration of the database, after a failure, to a consistent state.

Recovery from a failure is possible only if the DBMS maintains redundant data (called *recovery data*) including data about what users have been doing. To be able to recover from failures like a disk head crash, a backup copy of the database must be kept in a safe place. A proper design of recovery system would be based on clear assumptions of the types of failures expected and the probabilities of those failures occurring.

Of course we assume that the recovery process will restore the database to a consistent state as close to the time of failure as possible but this depends on the type of failure and the type of recovery information that the database maintains. If for example the disk on which the database was stored is damaged, the recovery process can only restore the database to the state it was when the database was last archived.

Most modern database systems are able to recover from a soft-fail by a system restart which takes only a few seconds or minutes. A hard-fail requires rebuilding of the database on the disk using one or more backup copies. This may take minutes or even hours depending on the type of the failure and the size of the database. We should note that no set of recovery procedures can cope with all failures and there will always be situations when complete recovery may be impossible. This may happen if the failure results in corruption of some part of the database as well as loss or corruption of redundant data that has been saved for recovery.

We note that recovery requirements in different database environments are likely to be different. For example, in banking or in an airline reservation system it would be a requirement that the possibility of losing information due to a system failure be made very very small. On the other hand, loss of information in a inventory database may not be quite so critical. Very important databases may require fault-tolerant hardware that may for example involve maintaining one or more duplicate copies of the database coupled with suitable recovery procedures.

The component of DBMS that deals with recovery is often called the *recovery manager*.

The Concept of A Transaction

Before we discuss recovery procedures, we need to define the concept of a transaction. A transaction may be defined as a logical unit of work which may involve a sequence of steps but which normally will be considered by the user as one action. For example, transferring an employee from Department A to Department B may involve updating several relations in a database but would be considered a single transaction. Transactions are straight-line programs devoid of control structures. The sequence of steps in a transaction may lead to inconsistent database temporarily but at the end of the transaction, the database is in a consistent state. It is assumed that a transaction always carries out correct manipulations of the database.

The concept of transaction is important since the user considers them as one unit and assumes that a transaction will be executed in isolation from all other transactions running concurrently that might interfere with the transaction. We must therefore require that actions within a transaction be carried out in a prespecified serial order and in full and if all the actions do not complete successfully for some reason then the partial effects must be undone. A transaction that successfully completes all the actions is said to commit. It otherwise aborts and must be rolled back. For example, when a transaction involves transferring an employee from Department A to Department B, it would not be acceptable if the result of a failed transaction was that the employee was deleted from Department A but not added to Department B. Haerder and Reuter (1983) summarise the properties of a transaction as follows:

a. **Atomicity** - although a transaction is conceptually atomic, a transaction would usually consist of a number of steps. It is necessary to make sure that other transactions do not see partial results of a transaction and therefore either all actions of a transaction are completed or the transaction has no effect on the database. Therefore a transaction is either completed successfully or rolled back. This is sometime called all-or-nothing.

b. **Consistency** - although a database may become inconsistent during the execution of a transaction, it is assumed that a completed transaction preserves the consistency of the database.

c. **Isolation** - as noted earlier, no other transactions should view any partial results of the actions of a transaction since intermediate states may violate consistency. Each transaction must be executed as if it was the only transaction being carried out.

d. **Durability** - once the transaction has been completed successfully, its effects must persist and a transaction must complete before its effects can be made permanent. A committed transaction cannot be aborted. Also a transaction would have seen the effects of other transactions. We assume those transactions have been committed before the present transaction commits.

When an application program specifies a transaction we will assume that it is specified in the following format:

Begin Transaction
(details of the transaction)
Commit

All the actions between the Begin and Commit are now considered part of a single transaction. If any of these actions fail, all the actions carried out before would need to be undone. We assume that transactions are not nested.

We will use the classical example of a bank account transfer transaction to illustrate some of the concepts used in recovery procedures. The transactions is:

Begin Transaction
Transfer 100 from Account A to Account B
Commit

It is clear that there needs to be a transfer program in the system that will execute the transaction. There are at least the following steps involved in carrying out the transaction:

a. Read Account A from Disk

b. If balance is less then 100, return with an appropriate message.

c. Subtract 100 from balance of Account A.

d. Write Account A back to Disk.

e. Read Account B from Disk.

f. Add 100 to balance of Account B.

g. Write Account B back to Disk.

We have ignored several small details; for example, we do not check if accounts A and B exist. Also we have assumed that the application program has the authority to access the accounts and transfer the money.

In the above algorithm, a system failure at step (e) or (f) would leave the database in an inconsistent state. This would result in 100 been subtracted from Account A and not added to Account B. A recovery from such failure would normally involve the incomplete transaction being rolled back.

In the discussion above we ignored how the disk manager transfers blocks from and to the disk. For example, in step (1) above, the block containing information on account A may already be in main memory buffers and then a read from disk is not required. More importantly, in Step (d), the write may not result in writing to the disk if the disk manager decides to modify the buffer but not write to disk. Also, in some situations, a write may first lead to a read if the block being modified is not resident in the main memory.

Now that the basic concept of transaction has been described, we can classify the different failures modes that we have discussed earlier.

Types of Failures

At the beginning of this chapter we discussed a number of failure modes. Gray *et al* (1981) classifies these failures in the following four classes:

1. **Transaction Failure** - A transaction may fail to complete for a number of reasons. For example, a user may cancel the transaction before it completes or the user may enter erroneous data or the DBMS may instruct the transaction to be abandoned and rolled back because of a deadlock or an arithmetic divide by zero or an overflow. Normally a transaction failure does not involve the loss of the contents of the disk or the main memory storage and

recovery procedure only involves undoing changes caused by the failed transaction.

2. **System Failure** - Most of these failures are due to hardware, database management system or operating system failures. A system failure results in the loss of the contents of the main memory but usually does not affect the disk storage. Recovery from system failure therefore involves reconstructing the database using the recovery information saved on the disk.
3. **Media Failure** - Media failures are failures that result in the loss of some or all the information stored on the disk. Such failures can occur due to hardware failures, for example disk head crash or disk dropped on the floor, or by software failures, for example, bugs in the disk writing routines in the operating system. Such failures can be recovered only if the archive version of the database plus a log of activities since the time of the archive are available.
4. **Unrecoverable Failures** - These are failures that result in loss of data that cannot be recovered and happen usually because of operations errors, for example, failure to make regular archive copies of the database or disasters like an earthquake or a flood.

Gray *et al* (1981) notes that in their experience 97 per cent of all transactions were found to execute successfully. Most of the remaining 3 per cent failed because of incorrect user input or user cancellation. All system crashes were found to occur every few days and almost all of these crashes were due to hardware or operating system failures. Several times a year, the integrity of the disk was lost.

The Concept of a Log

We now discuss some techniques of rolling back a partially completed transaction when a failure has occurred during its execution. It is clear that to be able to roll back a transaction we must store information about what that transaction has done so far. This is usually done by keeping a *log* or a *journal* although techniques that do not use a log exist. A log is an abstraction used by the recovery manager to store information needed to implement the atomicity and durable properties of transactions. The *log manager* is the component of a DBMS that implements the log abstraction. Logs are logically an append-only sequence of unstructured records stored on disk to which an insert is generated at each insert, delete and update. A log can therefore become very large quickly and may become a system performance problem. Each record appended to the log is assigned a unique *log sequence number* for identification.

When a transaction failure occurs, the transaction could be in one of the following situations:

a. The database was not modified at all and therefore no roll back is necessary. The transaction could be resubmitted if required.

b. The database was modified but the transaction was not completed. In this case the transaction must be rolled back and may be resubmitted if required.

c. The database was modified and the transaction was completed but it is not clear if all the modifications were written to the disk. In this case there is a need to ensure that all updates carried out by the completed transaction are durable.

To ensure that the uncompleted transactions can be rolled back and the completed transactions are durable, most recovery mechanisms require the maintenance of a log on a non-volatile medium. Without the log or some other similar technique it would not be possible to discover whether a transaction updated any items in the database. The log maintains a record of all the changes that are made to the database, although different recovery methods that use a log may require the maintenance of somewhat different log. Typically, a log includes entries of the following type:

1. Transaction Number 12345 has begun.
2. Transaction Number 12345 has written x that had old value 1000 and new value 2000.
3. Transaction Number 12345 has committed.
4. Transaction Number 12345 has aborted.

Once a transaction writes commit to the log, it is assumed to commit even if all changes have not been propoagated to the disk. It should however be noted that when an entry is made to the log, the log record may not be immediately written to the disk since normally a system will only write a log block when it is full. A system crash therefore may result in some part of the log that was not yet written to disk also being lost but this does not create any problems. Most systems insist that log blocks be forced out at appropriate times.

If the log maintains information about the database before the updates and after the updates, the information in the log may be used to undo changes when a failure occurs and redo changes that a committed trasaction has made to the database that may have not been written to the disk. Another approach is possible in which the log only maintains information about the database after the updates but the changes are made to the database only if the transaction is completed successfully. We consider both techniques.

Recovery is often a significant cost of maintaining a database system. A major component of the recovery cost is the cost of maintaining the log which is needed only when a crash occurs.

We note that log is the complete history of the database and the maintenance of a log is sometimes needed for auditing purposes and the log then is used for both auditing and recovery. The log may also be used for performance analysis. If a database contained sensitive data, it may be necessary to maintain a log of what information was read by whom and what was written back. This could then be used for auditing the use of the system. This is sometimes called an *audit-trail*. We do not discuss audit aspect of log maintenance any further. Since log is the complete history of the database, access to the log should be carefully controlled. Read access to a log could provide a user indirect access to the whole database.

At recovery time we cannot normally look at all the log since the last failure since the log might be very very big. Markers are therefore often put on the log to ensure that not all the log needs to be looked at when a failure occurs. We will discuss this further later.

Before we discuss the recovery techniques it is necessary to briefly discuss buffer management.

Recovery And Buffer Management

The part of main memory that is available for storage of copies of disk blocks is called the *buffer*. A software usually is needed to manage the buffer. The primary purpose of a buffer management strategy is to keep the number of disk accesses to a minimum. An easy way to reduce the disk traffic is to allocate a large fraction of the primary memory to storing blocks from disks but there clearly is a limit to how much primary storage can be assigned to buffers. Buffer manager is a specialised virtual memory manager since needs of a database are somewhat more specialised than an ordinary operating system.

1. Replacement strategy
2. Pinned blocks
3. Forced output of blocks

[See CACM July 1881 p 412]

Buffer manager interacts with the crash recovery system to decide on what can be written back to disk. Crash recovery system may insist that some other blocks be forced output before a particular buffer can be output.

To understand the various recovery mechanisms thoroughly, it is essential that we understand the buffer management strategies used in a DBMS. An operating system uses algorithms like the LRU (least recently used) and MRU (most recently used) in managing virtual memory. Similar strategies can be used in database buffer management but a database management does have some peculiar requirements that make some techniques that are suitable for operating systems unsuitable for database management.

Given that the main memory of the computer is volatile, any failure is likely to result in the loss of its contents which often includes updated blocks of data as well as output buffers to the log file.

[Does it need to be rewritten???] All log based recovery algorithms follow the *write-ahead* log principle. This involves writing the recovery data for a transaction to the log before a transaction commits and recovery data for a recoverable data page must be written to the log before the page is written from main memory to the disk. To enforce the write-ahead principle, the recovery manager needs to be integrated with the buffer manager. Some recovery algorithms permit the buffer manager to *steal* dirty main memory pages (a *dirty page* is a page in which data has been updated by a transaction that has not been committed) by writing them to disk before the transactions that modify them commit. The write-ahead principle requires the log records referring to the dirty pages must be written before the pages are cleaned. *No-steal* buffer management policies result in simpler recovery algorithms but limit the length of the transactions. Some recovery algorithms require the buffer manager to *force* all pages modified by a transaction to disk when the transaction commits. Recovery algorithms with *no-force* buffer management policies are more complicated, but they do less I/O than those which force buffers at commit.

We are now ready to discuss recovery techniques which will be in next lecture

References

1. R. Bayer and P. Schlichtiger (1984), “Data Management Support for Database Management”, Acta Informatica, 21, pp. 1-28.
2. P. Bernstein, N. Goodman and V. Hadzilacos (1987), “Concurrency Control and Recovery in Database Systems”, Addison Wesley Pub Co.
3. R. A. Crus (1984), ???, IBM Sys Journal, 1984, No 2.
4. T. Haerder and A. Reuter “Principles of Transaction-Oriented Database Recovery” ACM Computing Survey, Vol 15, Dec 1983, pp 287-318.
5. J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu and I. Traiger (1981), “The Recovery Manager of the System R Database Manager”, ACM Computing Surveys, Vol 13, June 1981, pp 223-242
6. J. Gray (1978?), “Notes on Data Base Operating Systems”, in Lecture Notes on Computer Science, Volume 60, R. Bayer, R.N. Graham, and G. Seegmueller, Eds., Springer Verlag.
7. J. Gray (1981?), “The Transaction Concept: The Virtues and Limitations” in Proc. of the the 7th International Conf on VLDB, Cannes, France, pp. 144-154.
8. J. Kent, H. Garcia-Molina and J. Chung (1985), “An Experimental Evaluation of Crash Recovery Mechanisms”, ACM-SIGMOD??, pp. 113-122.
9. J.S.M. Verhofstad “Recovery Techniques for Database Systems”, ACM Computing Surveys, Vol 10, Dec 78, pp 167-196.
10. Mohan???

Exercises

1. Log-based recovery techniques use protocols that include
   a. Do
   b. Redo
   c. Undo
2. A database failure may involve the loss of information in the
   a. main memory
   b. disk
   c. memory cache
   d. all of the above
3. To facilitate recovery, a recovery manager may force buffer manager to
   a. pin some pages
   b. force some pages
   c. read some pages
   d. write some pages
4. All log-based recovery techniques
   a. write-ahead log
   b. use checkpoint or a similar technique
   c. use the order of the log records to UNDO and REDO
   d. write log to disk after each transaction update

5. The primary reason why checkpoints are needed in the log is to
   a. reduce log processing
   b. maintain consistency
   c. maintain transaction durability
   d. reduce database archiving
6. Immediate update may be called logging only the UNDO information.

   Immediate update relinquishes the opportunity to delay writes.

   Immediate update requires that all updated pages be flushed out on transaction commit.

   Immediate update involves updating the database on the disk immediately followed by logging in the update.
7. Deferred update may be called logging only the REDO information.

   Deferred update allows delayed writes.

   Deferred update requires that modified pages in memory be pinned in the main memory until the transaction commits.

   Deferred update does not need a UNDO procedure.

   Deferred update can lead to problems when a transaction requires a large number of pages.
8. Recovery techniques enable recovery from all failures (??)

   A recovery system should be based on clear assumptions about types of failures and probabilities of these failures occurring.

   All recovery techniques require a maintenance of a loop.
9. A transaction is the smallest unit of work.

   A transaction is a logical unit of work which may involve a sequence of steps which preserves the consistency of the database.

   Transactions are not nested.

   Transactions are usually serial programs devoid of loops.
10. UNDO-REDO-DO protocol involves that

    UNDO, REDO (and DO?) be idempotent.

    Only recent log records are looked at.

    UNDO is done backwards and REDO forwards.

    REDO is necessary because all updates of committed transactions may not have been propagated to disk.

    Failures during recovery do not lead to any problems.
11. Which of the following is not true about deferred updates:
    a. the log may not be written until the transaction commits
    b. the log must be force-written to disk before any changes are made to the database on the disk
12. Recovery involves the following steps:

Notes:

LESSON 34
BACKUP AND RECOVERY-II

Hi! In this chapter I am going to discuss with you about Backup and Recovery in great detail.

A database might be left in an inconsistent state when:

- Deadlock has occurred.
- A transaction aborts after updating the database.
- Software or hardware errors.
- Incorrect updates have been applied to the database.

If the database is in an inconsistent state, it is necessary to recover to a consistent state. The basis of recovery is to have backups of the data in the database.

Recovery: the Dump

The simplest backup technique is 'the Dump'.

- Entire contents of the database is backed up to an auxiliary store.
- Must be performed when the state of the database is consistent - therefore no transactions which modify the database can be running
- Dumping can take a long time to perform
- You need to store the data in the database twice.
- As dumping is expensive, it probably cannot be performed as often as one would like.
- A cut-down version can be used to take 'snapshots' of the most volatile areas.

Recovery: the Transaction Log

A technique often used to perform recovery is the transaction log or journal.

- Records information about the progress of transactions in a log since the last consistent state.
- The database therefore knows the state of the database before and after each transaction.
- Every so often database is returned to a consistent state and the log may be truncated to remove committed transactions.
- When the database is returned to a consistent state the process is often referred to as 'checkpointing'.

Recovery Techniques

Consider the following sequence of transactions:

Time	T_1	T_2	T_3
0	Begin		
1	Read N	-	Begin
2	-	-	Read P
3	N := N-5	-	-
4	-	Begin	P := P-4
5	Write N	Read M	-
6	Commit	-	Write P
7	-	Write M	-
8		M := M*1.5	Commit
9	-	Temp := M	-
10	-	B := B+1	-

Now assume that a system failure occurs at time 9. Since transactions T_1 and T_3 have committed before the crash, the recovery procedures should ensure that the effects of these transactions are reflected in the database. On the other hand T_2 did not commit and it is the responsibility of the recovery procedure to ensure that when the system restarts the effects of partially completed T_2 are undone for ever.

Jim Gray presented the following three protocols that are needed by the recovery procedures to deal with various recovery situations:

1. DO(T_i, action) - this procedure carries out the action specified. The log is written before an update is carried out.
2. UNDO(T_i) - this procedure undoes the actions of the transaction T_i using the information in the log.
3. REDO(T_i) - this procedure redoes the actions of the committed transaction T_i using the information in the log.

The above actions are sometime called *DO-UNDO-REDO* protocol. They play a slightly different role in the two recovery techniques that we discuss.

When a transaction aborts, the information contained in the log is used to undo the transaction's effects on the database. Most logging disciplines require that a transaction be undone in reverse of the order that its operations were performed. To facilitate this, many logs are maintained as a singly linked list of log records.

As discussed earlier, a system crash divides transactions into three classes. First, there are transactions that committed before the crash. The effects of these transactions must appear in the database after the recovery. If the recovery manager forces all dirty pages to disk at transaction commit time, there is no work

necessary to redo committed transactions. Often however, a no-force buffer management policy is used and it is therefore necessary to redo some of the transactions during recovery. The second class of transactions includes those that were aborted before the crash and those that are aborted by the crash. Effects of these transactions must be removed from the database. Another class of transactions is those that were completed but did not commit before the crash. It is usually possible to commit such transactions during recovery.

We now discuss two recovery techniques that use a log. Another technique, that does not use a log, is discussed later.

- Immediate Updates
- Deferred Updates
- System Checkpoints
- Summary of Log-based Methods
- Shadow Page Schemes
- Evaluation

Immediate Updates

As noted earlier, one possible recovery technique is to allow each transaction to make changes to the database as the transaction is executed and maintain a log of these changes. This is sometimes called logging only the UNDO information. The information written in the log would be identifiers of items updated and their old values. For a deleted item, the log would maintain the identifier of the item and its last value. As discussed above, the log must be written to the disk before the updated data item is written back.

We consider a simple example of bank account transfer discussed earlier. When the transaction is ready to be executed, the transaction number is written to the log to indicate the beginning of a transaction, possibly as follows:

<T = 1235, BEGIN>

This is then followed by the write commands, for example the log entries might be:

<T = 1235, Write Account A, old value = 1000, new value = 900>
<T = 1235, Write Account B, old value = 2000, new value = 2100>
<T = 1235, COMMIT>

Our brief discussion above has left several questions unanswered, for example:

a. Is the write-ahead principle necessary?

b. If a crash occurs, how does recovery take place?

c. What happens if a failure occurs as the log is being written to the disk?

d. What happens if another crash occurs during the recovery procedure?

We now attempt to answer these questions.

If the updated data item is written back to disk before the log is, a crash between updating the database and writing a log would lead to problems since the log would have no record of the update. Since there would be no record, the update could not be undone if required. If the log is written before the actual update item is written to disk, this problem is overcome. Also if a failure occurs during the writing of the log, no damage would have been done since the database update would not have yet taken place. A transaction would be regarded as having committed only if the logging of its updates has been completed, even if all data updates have not been written to the disk.

We now discuss how the recovery takes place on failure. Assuming that the failure results in the loss of the volatile memory only, we are left with the database that may not be consistent (since some transformations that did not commit may have modified the database) and a log file. It is now the responsibility of the recovery procedure to make sure that those transactions that have log records stating that they were committed are in fact committed (some of the changes that such transactions made may not yet have been propagated to the database on the disk). Also effects of partially completed transactions i.e. transactions for which no commit log record exists, are undone. To achieve this, the recovery manager inspects the log when the system restarts and REDOes the transactions that were committed recently (we will later discuss how to identify transactions that are recent) and UNDOes the transactions that were not committed. This rolling back of partially completed transactions requires that the log of the uncommitted transactions be read backwards and each action be undone. The reason for undoing the uncommited transaction backwards becomes if we consider an example in which a data item is updated more than once as follows.

1. **< T = 1135, BEGIN >**
2. **< T = 1135, Write Account A, 1000, 900 >**
3. **< T = 1135, Write Account A, 900, 700 >**
4. **< T = 1135, Write Account B, 2000, 2100 >**

Only backward UNDOing can ensure that the original value of 1000 is restored.

Let us consider a simple example of the log.

1. **< T = 1235, BEGIN >**
2. **< T = 1235, Write Account A, 1000, 900 >**
3. **< T = 1235, Write Account B, 2000, 2100 >**
4. **< T = 1235 COMMIT >**
5. **< T = 1240, BEGIN >**
6. **< T = 1240, Write Account C, 2500, 2000 >**
7. **< T = 1240, Write Account D, 1500, 2000 >**
8. **< T = 1240, COMMIT >**
9. **< T = 1245, BEGIN >**

Let the log consist of the 9 records above at failure. The recovery procedure first checks what transactions have been committed (transactions 1235 and 1240) and REDOes them by the following actions:

REDO (T = 1235)
REDO (T = 1240)

Transaction 1245 was not committed. UNDO (T = 1245) is therefore issued. In the present example, transaction 1245 does not need to be rolled back since it didn't carry out any updates.

The reason for REDOing committed transactions is that although the log states that these transactions were committed, the log is written before the actual updated data items are written to disk and it is just possible that an update carried out by a committed transaction was not written to disk. If the update was in fact written, the REDO must make sure that REDOing the transaction does not have any further effect on the database. Also several failures may occur in a short span of time and we may REDO transactions 1235 and 1240 and UNDO transaction 1245 several times. It is therefore essential that UNDO and REDO operations be idempotent; that is, the effect of doing the same UNDO or REDO operations several times is the same as doing it once. An operation like add $35 to the account obviously is not idempotent while an operation setting the account balance to $350 obviously is.

Let us consider another example of UNDO. Let the log have only the first seven records at failure. We therefore issue

REDO (T = 1235)
UNDO (T = 1240)

Once REDO is completed, UNDO reads the log backwards. It reads record of the log and restores the old value of Account D, then reads record 6 of the log and restores the old values of Account C.

Note that the recovery procedure works properly even if there is a failure during recovery since a restart after such a failure would result in the same process being repeated again

Deferred Updates

The immediate update scheme requires that before-update values of the data items that are updated be logged since updates carried out by the transactions that have not been committed may have to be undone. In the deferred updates scheme, the database on the disk is not modified until the transaction reaches its commit point (although all modifications are logged as they are carried out in the main memory) and therefore the before-update values do not need to be logged. Only the after-update data values of the database are recorded on the log as the transaction continues its execution but once the transaction completes successfully (called a partial commit) and writes to the log that is ready to commit, the log is force-written to disk and all the writes recorded on the log are then carried out. The transaction then commits. This is sometimes called as logging only the REDO information. The technique allows database writes to be delayed but involves forcing the buffers that hold transaction modifications to be pinned in the main memory until the transaction commits.

There are several advantages of the deferred updates procedure. If a system failure occurs before a partial commit, no action is necessary whether the transaction logged any updates or not. A part of the log that was in the main memory may be lost but this does not create any problems. If a partial commit had taken place and the log had been written to the disk then a system failure would have virtually no impact since the deferred writes would be REDOne by the recovery procedure when the system restarts.

The deferred update method can also lead to some difficulties. If the system buffer capacity is limited, there is the possibility that the buffers that have been modified by an uncommitted transaction are forced out to disk by the buffer management part of the DBMS. To ensure that the buffer management will not force any of the uncommitted updates to the disk requires that the buffer must be large enough to hold all the blocks of the database that are required by the active transactions. We again consider the bank account transfer example considered earlier. When the transaction is ready to be executed, the transaction number is written on the log to indicate the beginning of a transaction, possibly as follows:

<T = 1235, BEGIN>

This is then followed by the write commands in the log, for example:

<Write Account A, new value = 900>
<Write Account B, new value = 2100>
<COMMIT T = 1235>

Note that the deferred update only requires the new values to be saved since recovery involves only REDOs and never any UNDOs. Once this log is written to the disk, the updates may be carried out. Should a failure occur at any stage after writing the log and before updating the database on the disk, the log would be used to REDO the transaction.

Similar to the technique used in the last section, the recovery manager needs a procedure REDO which REDOes all transactions that were active recently and committed. No recovery action needs to be taken about transactions that did not partially commit (those transactions for which the log does not have a commit entry) before the failure because the system would not have updated the database on the disk for those transactions. Consider the following example:

1. **< T = 1111, BEGIN >**
2. **< Write BEGIN > ??**
3. **< T = 1111, COMMIT >**
4. **< T = 1235, BEGIN >**
5. **< Write Account A, 900 >**
6. **< Write Account B, 2100 >**
7. **< T = 1235 COMMIT >**
8. **< T = 1240, BEGIN >**
9. **< T = 1240, Write Account C, 2000 >**
10. **< T = 1240, Write Account D, 2000 >**
11. **< T = 1240, COMMIT >**
12. **< T = 1245, BEGIN >**

Now if a failure occurs and the log has the above entries, the recovery manager identifies all transactions that were active recently and have been committed (Transaction 1235 and 1240). We assume the recovery manager does not need to worry about transactions that were committed well before the crash (for example, transaction 1111). In the next section, we will discuss how we identify recent transactions.

If another failure occurs during the recovery the same transactions may be REDOne again. As noted earlier, the effect of redoing a transaction several times is the same as doing it once.

[chop this para???] If this log is on the disk, it can handle all system failures except the disk crash. When a failure has occurred, the system goes through the log and carries out transactions T = 1235 and T = 1240. Note that in some cases the transactions may be done more than once but this will not create any problems as discussed above.

System Checkpoints

In the discussion above, the Immediate Update method involves REDOing all recently committed transactions and undoing transactions that were not committed while the Deferred Update method only requires REDOing and no UNDOing. We have so far not defined how recent transactions are identified. We do so now.

One technique for identifying recent transactions might be to search the entire log and identify recent transactions on some basis. This is of course certain to be very inefficient since the log may be very long. Also, it is likely that most transactions that are selected for REDOing have already written their updates into the database and do not really need to be REDOne. REDOing all these transactions will cause the recovery procedure to be inefficient.

To limit the number of transactions that need to be reprocessed during recovery after a failure, a technique of marking the log is used. The markers are called system checkpoints and putting a marker on the log (called taking a checkpoint) consists of the following steps:

a. Write a <begin checkpoint> record to the log (on the disk?),

b. All log records currently residing in the main memory are written to the disk followed by the writing to the disk of all modified pages in the buffer. Also identifiers (or names) of all active transactions are written to the log.

c. Write an <end checkpoint> record to the log.

Now when the recovery procedure is invoked on a failure, the last checkpoint is found (this information is often recorded in a file called the restart file) and only the transactions active at the time of checkpoint and those after the checkpoint are processed.

A simple recovery algorithm requires that the recovery manager identify the last checkpoint and builds two lists, one of the transactions that need to be redone and the other of transactions that need to be undone. Initially, all transactions that are listed as active at the checkpoint are included in the UNDO list and the log is scanned forward (show the lists and show the steps???). Any new transaction that becomes active is added to the UNDO list and any transaction that is logged to have committed is removed from the UNDO list and placed on the REDO list. At the end of the log, the log is scanned backwards and all the actions of those transactions that are on the UNDO list are UNDOne in the backward order. Once the checkpoint is reached, the log is scanned forward and all the actions of the transactions on the REDO list are REDOne. (s/a??)

Summary of Log-based Methods

Although log-based recovery methods vary in a number of details, there are a number of common requirements on the recovery log. First, all recovery methods adhere to the write-ahead log procedure. Information is always written to the log before it propagates to non-volatile memory and before transactions commit. Second, all recovery methods rely on the ordering of operations expressed in the log to provide an ordering for the REDO and UNDO operations. Third, all methods use checkpoints or some similar technique to bound the amount of log processed for recovery. Some recovery methods scan the log sequentially forward during crash recovery. Others scan the log sequentially backwards. Still others use both forward and backward scans.

A log manager may manage a log as a file on the secondary storage. When this file has reached a pre-specified length, the log may be switched to another file while the previous log is copied to some archive storage, generally a tape. The approach used by System R is to take a large chunk of disk space and lay it out for the log as a circular buffer of log pages. The log manager appends new blocks sequentially to the circular buffer as the log fills. Although an abstract log is an infinite resource, in practice the online disk space available for storing the log is limited. Some systems will spool truncated log records to tape storage for use in media recovery. Other systems will provide enough online log space for media recovery and will discard truncated log data. (detail??)

Shadow Page Schemes

Not all recovery techniques make use of a log. Shadow page schemes (or careful replacement schemes) are recovery techniques that do not use logging for recovery. In these schemes when a page of storage is modified by a transaction a new page is allocated for the modified data and the old page remains as a shadow copy of the data. This is easily achieved by maintaining two tables of page addresses, one table called the current page table and the other called the shadow page table. At the beginning of a transaction the two page tables are identical but as the transaction modifies pages new pages are allocated and the current page table is modified accordingly while the shadow page table continues to point to the old pages. The current pages may be located in the main memory or on the disk but all current pages are output to the disk before the transaction commits. If a failure occurs before the transaction commits, the shadow page table is used to recover the database state before the transaction started. When a transaction commits, the pages in the current page table (which now must be on the disk) become the pages in the shadow page table. The shadow pages must be carefully replaced with the new pages in an atomic operation. To achieve this, the current page table is output to the disk after all current pages have been output. Now the address of the shadow page table is replaced by the address of the current page table and the current page table becomes the shadow page table committing the transaction. Should a failure occur before this change, the old shadow page table is used to recover the database state before the transaction.

Shadow page recovery technique eliminates the need for the log although the technique is sometime criticized as having poor performance for normal processing. However, the recovery is often fast when shadow paging is used. Also the technique requires a suitable technique for garbage collection to remove all old shadow pages.

Evaluation

(add more to this section? rewrite it??) Gray et al in their paper note that the recovery system was comparatively easy to write and added about 10 percent to the DBMS code. In addition, the cost of writing log records was typically of the order of 5 percent. Cost of checkpoints was found to be minimal and restart after a crash was found to be quite fast. Different recovery algorithms have varying costs, both during normal processing and recovery. For the log algorithms, the costs of algorithms during normal processing include the volume and frequency of the log writes and the algorithms influence on buffer management. Recovery algorithms which force the buffer pool have comparably poor performance. The costs of crash recovery include the number of log records read, and the number of pages of data that must be paged into main memory, restored and paged out. No-force recovery algorithms will read more data pages during recovery than force algorithms. Recovery algorithms with steal buffer pool management policies may read data pages for both redo and undo processing, while no steal buffer managers mainly read pages for redo processing.

Recovering from a Disk Crash (Media failure?)

So far, our discussion about recovery has assumed that a system failure was a soft-failure in that only the contents of the volatile memory were lost and the disk was left intact. Of course, a disk is not immune from failure and a head crash may result in all the contents of the disk being lost. Other, more strange, failures are also possible. For example, a disk pack may be dropped on the floor by a careless operator or there may be some disaster as noted before. We therefore need to be prepared for a disaster involving loss of the contents of the nonvolatile memory.

As noted earlier, the primary technique for recovery from such loss of information is to maintain a suitable back up copy of the database possibly on tape or on a separate disk pack and ensure its safe storage possibly in a fire-proof and water-proof safe preferably at a location some distance away from the database site. Such back up may need to be done every day or may be done less or more frequently depending upon the value the organisation attaches to loss of some information when a disk crash occurs.

Since databases tend to be large, a complete back up is usually quite time consuming. Often then the database may be backed up incrementally i.e. only copies of altered parts of the database are backed up. A complete backup is still needed regularly but it would not need to be done so frequently. After a system failure, the incremental system tapes and the last complete back up tape may be used to restore the database to a past consistent state.

Rollback

The process of undoing changes done to the disk under immediate update is frequently referred to as rollback.

- Where the DBMS does not prevent one transaction from reading uncommitted modifications (a 'dirty read') of another transaction (i.e. the uncommitted dependency problem) then aborting the first transaction also means aborting all the transactions which have performed these dirty reads.
- As a transaction is aborted, it can therefore cause aborts in other dirty reader transactions, which in turn can cause other aborts in other dirty reader transaction. This is referred to as 'cascade rollback'.

Important Note

Deferred Update

Deferred update, or NO-UNDO/REDO, is an algorithm to support ABORT and machine failure scenarios.

- *While a transaction runs, no changes made by that transaction are recorded in the database.*
- *On a commit:*
- *The new data is recorded in a log file and flushed to disk*
- *The new data is then recorded in the database itself.*
- *On an abort, do nothing (the database has not been changed).*
- *On a system restart after a failure, REDO the log.*

If the DMBS fails and is restarted:

- *The disks are physically or logically damaged then recovery from the log is impossible and instead a restore from a dump is needed.*
- *If the disks are OK then the database consistency must be maintained. Writes to the disk which was in progress at the time of the failure may have only been partially done.*
- *Parse the log file, and where a transaction has been ended with 'COMMIT' apply the data part of the log to the database.*
- *If a log entry for a transaction ends with anything other than COMMIT, do nothing for that transaction.*
- *Flush the data to the disk, and then truncate the log to zero.*
- *The process or reapplying transaction from the log is sometimes referred to as 'rollforward'.*

Immediate Update

Immediate update, or UNDO/REDO, is another algorithm to support ABORT and machine failure scenarios.

- *While a transaction runs, changes made by that transaction can be written to the database at any time. However, the original and the new data being written must both be stored in the log BEFORE storing it on the database disk.*
- *On a commit:*
- *All the updates which has not yet been recorded on the disk is first stored in the log file and then flushed to disk.*
- *The new data is then recorded in the database itself.*
- *On an abort, REDO all the changes which that transaction has made to the database disk using the*
- *Log entries.*
- *On a system restart after a failure, REDO committed changes from log.*

If the DMBS fails and is restarted:

- *The disks are physically or logically damaged then recovery from the log is impossible and instead a restore from a dump is needed.*
- *If the disks are OK then the database consistency must be maintained. Writes to the disk which was in progress at the time of the failure may have only been partially done.*

- *Parse the log file, and where a transaction has been ended with 'COMMIT' apply the 'new data' part of the log to the database.*
- *If a log entry for a transaction ends with anything other than COMMIT, apply the 'old data' part of the log to the database.*
- *flush the data to the disk, and then truncate the log to zero.*

Review Question

1. What is Recovery and why it is important?
2. Explain recovery techniques

Exercises

1. Given the log on page 6 of these notes, what transactions need to be REDOne or UNDOne if a failure occurs:
   a. after step 8
   b. after step 10
2. What difference is expected if deferred update procedure is being used?
3. Explain what would happen if a failure occurred during the recovery procedure in Ex. 1.
4. A system crash divides transactions into the following classes:
   a. Committed before the checkpoint
   b. Not completed
   c Not committed at all
   d. Inactive
5. Give an example of each
   a. transaction failure
   b. system failure
   c media failure
   d. unrecoverable failure
6. Given that one million transactions are run on a database system each day, how many transactions are likely to fail:
   a. 10,000 to 100,000
   b. 1,000 to 10,000
   c 100 to 1,000
   d. below 100
7. Which of the following are properties of a transaction
   a. atomicity
   b. consistency
   c isolation
   d. durability
   e. idempotency

References

1. R. Bayer and P. Schlichtiger (1984), "Data Management Support for Database Management", Acta Informatica, 21, pp. 1-28.
2. P. Bernstein, N. Goodman and V. Hadzilacos (1987), "Concurrency Control and Recovery in Database Systems", Addison Wesley Pub Co.
3. R. A. Crus (1984), ???, IBM Sys Journal, 1984, No 2.
4. T. Haerder and A. Reuter "Principles of Transaction-Oriented Database Recovery" ACM Computing Survey, Vol 15, Dec 1983, pp 287-318.
5. J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu and I. Traiger (1981), "The Recovery Manager of the System R Database Manager", ACM Computing Surveys, Vol 13, June 1981, pp 223-242
6. J. Gray (1978?), "Notes on Data Base Operating Systems", in Lecture Notes on Computer Science, Volume 60, R. Bayer, R.N. Graham, and G. Seegmueller, Eds., Springer Verlag.
7. J. Gray (1981?), "The Transaction Concept: The Virtues and Limitations" in Proc. of the the 7th International Conf on VLDB, Cannes, France, pp. 144-154.
8. J. Kent, H. Garcia-Molina and J. Chung (1985), "An Experimental Evaluation of Crash Recovery Mechanisms", ACM-SIGMOD??, pp. 113-122.
9. J.S.M. Verhofstad "Recovery Techniques for Database Systems", ACM Computing Surveys, Vol 10, Dec 78, pp 167-196.
10. Mohan???

Notes:

LESSON 35
CHECKPOINT

Hi! Today you will learn about check point and then only you will understand how important check point is? You will also learn the mechanisms of checkpoint. This lecture is covering check point in great depth. Though this topic is huge, introduction and basic to checkpoint is sufficient to understand. But if you want to understand checkpoint from DBA's point of view then go through the entire lecture.

1. Introduction

Most large institutions have now heavily invested in a data base system. In general they have automated such clerical tasks as inventory control, order entry, or billing. These systems often support a worldwide network of hundreds of terminals. Their purpose is to reliably store and retrieve large quantities of data. The life of many institutions is critically dependent on such systems, when the system is down the corporation has amnesia.

This puts an enormous burden on the implementers and operators of such systems. The systems must on the one hand be very high performance and on the other hand they must be very reliable

System Checkpoint Logic

System checkpoints may be triggered by operator commands, timers, or counters such as the number of bytes of log record since last checkpoint. The general idea is to minimize the distance one must travel in the log in the event of a catastrophe. This must be balanced against the cost of taking frequent checkpoints. Five minutes is a typical checkpoint interval.

Checkpoint algorithms that require a system quiesce should be avoided because they imply that checkpoints will be taken infrequently thereby making restart expensive.

The checkpoint process consists of writing a BEGIN_CHECKPOINT record in the log, then invoking each component of the system so that it can contribute to the checkpoint, and then writing an END_CHECKPOINT record in the log. These records bracket the checkpoint records of the other system components. Such a component may write one or more log records so that it will be able to restart from the checkpoint. For example, buffer manager will record the names of the buffers in the buffer pool, file manager might record the status of files, network manager may record the network status, and transaction manager will record the names of all transactions active at the checkpoint.

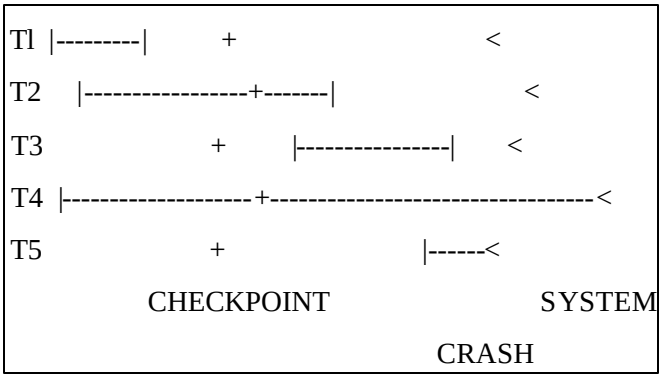
After the checkpoint log records have been written to non-volatile storage, recovery manager records the address of the most recent checkpoint in a warm start file. This allows restart to quickly locate the checkpoint record (rather than sequentially searching the log for it.) Because this is such a critical resource, the restart file is duplexed (two copies are kept) and writes to it are alternated so that one file points to the current and another points to the previous checkpoint log record.

At system restart, the programs are loaded and the transaction manager invokes each component to re-initialize itself. Data communications begins network-restart and the database manager reacquires the database from the operating system (opens the files).

Recovery manager is then given control. Recovery manager examines the most recent warm start file written by checkpoint to discover the location of the most recent system checkpoint in the log. Recovery manager then examines the most recent checkpoint record in the log. If there was no work in progress at the system checkpoint and the system checkpoint is the last record in the log then the system is in restarting from a shutdown in a quiesced state. This is a warm start and no transactions need be undone or redone. In this case, recovery manager writes a restart record in the log and returns to the scheduler, which opens the system for general use.

On the other hand if there was work in progress at the system checkpoint, or if there are further leg records then this is a restart from a crash (emergency restart).

The following figure will help to explain emergency restart logic:



Five transaction types with respect to the most recent system checkpoint and the crash point. Transactions T1, T2, and T3 have committed and must be redone. Transactions T4 and T5 have not committed and so must be undone. Let's call transactions like T1, T2 and T3 winners and lets call transactions like T4 and T5 losers. Then the restart logic is:

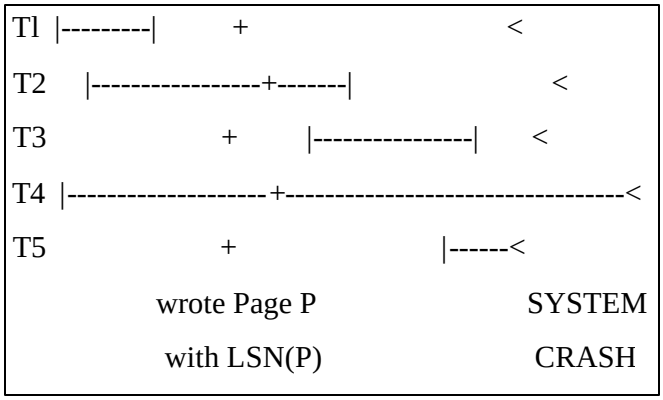
```
RESTART: PROCEDURE;
    DICHOTOMIZE WINNERS AND LOSERS;
    REDO THE WINNERS;
    UNDO THE LOSERS;
    END RESTART;
```

It is important that the REDOs occur before the UNDO (Do you see why (we are assuming page-locking and high-water marks from log-sequence numbers?)

As it stands, this implies reading every log record ever written because redoing the winners requires going back to redo almost all transactions ever run.

Much of the sophistication of the restart process is dedicated to minimizing the amount of work that must be done, so that restart can be as quick as possible, (We are describing here one of the more trivial workable schemes.) In general restart discovers a time T such that redo log records written prior to time T are not relevant to restart.

To see how to compute the time T, we first consider a particular object: a database page P. Because this is a restart from a crash, the most recent version of P may or may not have been recorded on non-volatile storage. Suppose page P was written out with high water mark LSN(P). If the page was updated by a winner "after" LSN(P), then that update to P must be redone. Conversely, if P was written out to nonvolatile storage with a loser's update, then those updates must be undone. (Similarly, message M may or may rot have been sent to its destination.) If it was generated by a loser, then the message should be canceled. If it was generated by a committed transaction but not sent then it should be retransmitted.) The figure below illustrates the five possible types of transactions at this point: T1 began and committed before LSN(P), T2 began before LSN(P) and ended before the crash, T3 began after LSN(P)and ended before the crash, T4 began before LSN(P) but its COMMIT record does not appear in the log, and T5 began after LSN(P) and apparently never ended. To honor the commit of T1, T2 and T3 requires that their updates be added to page P (redone). But T4, T5, and T6 have not committed and so must be undone.



Five transactions types with respect to the most recent write of page P and the crash point,

Notice that none of the updates of T5 are reflected in this state so T5 is already undone. Notice also that all of the updates of T1 are in the state so it need not be redone. So only T2, T3, and T4 remain. T2 and T3 must be redone from LSN(P) forward. The updates of the first half of T2 are already reflected in the page P because it has log sequence number LSN(P). On the other hand, T4 must be undone from LSN(P) backwards. (Here we are skipping over the following anomaly: if after LSN(P), T2 backs up to a point prior to the LSN(P)then some undo work is required for T2. This problem is not difficult, just annoying.)

Therefore the oldest redo log record relevant to P is at or after LSN(P). (The write-ahead-log protocol is relevant here.) At system checkpoint, data manager records MINLSN, the log sequence number of the oldest page not yet written (the minimum LSN(P)of all pages, P, not yet written.) Similarly, transaction manager records the name of each transaction active at the checkpoint. Restart chooses T as the MINLSN of the most recent checkpoint.

Restart proceeds as follows: It reads the system checkpoint log record and puts each transaction active at the checkpoint into the loser set.

It then scans the log forward to the end. If a COMMIT log record is encountered, that transaction is promoted to the winners set. If a BEGIN_TRANSACTION record is found, the transaction is tentatively added to the loser set. When the end of the log is encountered, the winners and losers have been computed. The next thing is to read the log forwards from MINLSN, redoing the winners. Then it starts from the end of the log, read the log backwards undoing the losers.

This discussion of restart is very simplistic. Many systems have added mechanisms to speed restart by:

- Never write uncommitted objects to non-volatile storage (stealing) so that undo is never required.
- Write committed objects to secondary storage at phase 2 of commit (forcing), so that redo is only rarely required (this maximizes "MINLSN).
- Log the successful completion of a write to secondary storage. This minimizes redo.
- Force all objects at system checkpoint, thereby maximizing MINLSN.

5.8.4.5. Media Failure Logic

In the event of a hard system error (one non-volatile storage integrity), there must be minimum of lost work. Redundant copies of the object must be maintained, for example on magnetic tape that is stored in a vault. It is important that the archive mechanism have independent failure modes from the regular storage subsystem. Thus, using doubly redundant disk storage would protect against a disk head crash, but wouldn't protect against a bug in the disk driver routine or a fire in the machine room. The archive mechanism periodically writes a checkpoint of the data base contents to magnetic tape, and writes a redo log of all update actions to magnetic tape, Then recovering from a hard failure is accomplished by locating the most recent surviving version on tape, loading it back into the system, and then redoing all updates from that point forward using the surviving log tapes.

While performing a system checkpoint causes relatively few disk writes, and takes only a few seconds, copying the entire database to tape is potentially a lengthy operation. Fortunately there is a (little used) trick: one can take a fuzzy dump or an object by writing it to archive with an idle task. After the dump is taken, the log generated during the fuzzy dump is merged with the fuzzy dump to produce a sharp dump. The details of this algorithm are left as an exercise for the reader.

5.8.4.6. Cold Start Logic

Cold start is too horrible to contemplate. Since we assumed that the log never fails, cold start is never required. The system should be cold started once: when the implementers create its first version. Thereafter, it should be restarted. In particular moving to new hardware or adding to a new release of the

system should not require a cold start. (i.e. all data should survive.) Note that this requires that the format of the log never change, it can only be extended by adding new types of log records.

5.8.5. Log Management

The log is a large linear byte space. It is very convenient if the log is write-once, and then read-only. Space in the log is never re-written. This allows one to identify log records by the relative byte address of the last byte of the record.

A typical (small) transaction writes 500 bytes of log. One can run about one hundred such transactions per second on current hardware. There are almost 100,000 seconds in a day. So the log can grow at 5 billion bytes per day. (more typically, systems write four log tapes a day at 50 megabytes per tape.) Given those statistics the log addresses should be about 48 bits long (good for 200 years on current hardware.)

Log manager must map this semi-infinite logical file (log) into the rather finite files (32 bit addresses) provided by the basic operating system. As one file is filled, another is allocated and the old one is archived. Log manager provides other resource managers with the operations:

WRITE_LOG: causes the identified log record to be written to the log. Once a log record is written. It can only be read. It cannot be edited. WRITE_LOG is the basic command used by all resource managers to generate log records. It returns the address of the last byte of the written log record.

FORCE-LOG: causes the identified log record and all prior log records to be recorded in nonvolatile storage. When it returns, the writes have completed.

OPEN-LOG: indicates that the issuer wishes to read the log of some transaction, or read the entire log in sequential order. It creates a read cursor on the log.

SEARCH-LOG: moves the cursor a designated number of bytes or until a log record satisfying some criterion is located.

READ-LOG: requests that the log record currently selected by the log cursor be read.

CHECK-LOG: allows the issuer to test whether a record has been placed in the non-volatile log and optionally to wait until the log record has been written out.

GET-CURSOR: causes the current value of the write cursor to be returned to the issuer. The RBA (relative byte address) returned may be used at a later time to position a read cursor.

CLOSE-LOG: indicates the issuer is finished reading the log.

The write log operation moves a new log record to the end of the current log buffer. If the buffer fills, another is allocated and the write continues into the new buffer.

When a log buffer fills or when a synchronous log write is issued, a log daemon writes the buffer to nonvolatile storage. Traditionally, logs have been recorded on magnetic tape because it is so inexpensive to store and because the transfer rate is quite high. In the future disk, CCD (nonvolatile?) or magnetic bubbles may be attractive as a staging device for the log. This is especially true because an on-line version of the log is very desirable for transaction undo and for fast restart.

It is important to doubly record the log. If the log is not doubly recorded, then a media error on the log device will produce a cold start of the system. The dual log devices should be on separate paths so that if one device or path fails the system can continue in degraded mode (this is only appropriate for applications requiring high availability.)

The following problem is left as an exercise for the reader: We have decided to log to dedicated dual disk drives. When a drive fills it will be archived to a mass storage device. This archive process makes the disk unavailable to the log manager (because of arm contention.) Describe a scheme which:

- minimizes the number of drives required, .
- always has a large disk reserve of free disk space, and
- always has a large fraction of the recent section of the log on line.

5.8.5.1. Log Archive and Change Accumulation

When the log is archived, it can be compressed so that it is convenient for media recovery. For disk objects, log records can be sorted by cylinder, then track then sector then time. Probably, all the records in the archived log belong to completed transactions. So one only needs to keep redo records of committed (not aborted) transactions. Further only the most recent redo record (new value) need be recorded. This compressed redo log is called a change accumulation log. Since it is sorted by physical address, media recover becomes a merge of the image dump of the object and its change accumulation tape.

```
FAST_MEDIA_RECOVERY: PROCEDURE (IMAGE,
CHANGE_ACCUMULATION_LOG);
        DO WHILE ( ! END_OF_FILE IMAGE);
                READ IMAGE PAGE;
                UPDATE WITH REDO RECORDS FROM
CHANGE_ACCUMULATION_LOG;
                WRITE IMAGE PAGE TO DISK;
                END
        END;
```

This is a purely sequential process (sequential on input files and sequential on disk being recovered) and so is limited only by the transfer rates of the devices.

The construction of the change accumulation file can be done off-line as an idle task.

If media errors are rare and availability of the data is not a critical problem then one may run the change accumulation utilities when needed. This may save building change accumulation files that are never used.

The same topic check point is dealt in detail as tutorials which are added at the last of these lectures. For further reference, refer those tutorials.

Review Question

1. What is Checkpoint and why it is important?

LESSON 36
SQL SUPPORT

Hi! In this chapter I am going to discuss with you about SQL Support in DBMS.

Introduction to Structured Query Language Version 4.11

This page is a tutorial of the *Structured Query Language* (also known as **SQL**) and is a pioneering effort on the World Wide Web, as this is the first comprehensive SQL tutorial available on the Internet. SQL allows users to access data in relational database management systems, such as Oracle, Sybase, Informix, Microsoft SQL Server, Access, and others, by allowing users to describe the data the user wishes to see. SQL also allows users to define the data in a database, and manipulate that data. This page will describe how to use SQL, and give examples. The SQL used in this document is "ANSI", or standard SQL, and no SQL features of specific database management systems will be discussed until the "Nonstandard SQL" section. It is recommended that you print this page, so that you can easily refer back to previous examples.

Table of Contents

Basics of the SELECT Statement

In a relational database, data is stored in tables. An example table would relate Social Security Number, Name, and Address:

EmployeeAddressTable					
SSN	**FirstName**	**LastName**	**Address**	**City**	**State**
512687458	Joe	Smith	83 First Street	Howard	Ohio
758420012	Mary	Scott	842 Vine Ave.	Losantiville	Ohio
102254896	Sam	Jones	33 Elm St.	Paris	New York
876512563	Sarah	Ackerman	440 U.S. 110	Upton	Michigan

Now, let's say you want to see the address of each employee. Use the SELECT statement, like so:

```
SELECT FirstName, LastName, Address, City, State
FROM EmployeeAddressTable;
```

The following is the results of your *query* of the database:

First Name	Last Name	Address	City	State
Joe	Smith	83 First Street	Howard	Ohio
Mary	Scott	842 Vine Ave.	Losantiville	Ohio
Sam	Jones	33 Elm St.	Paris	New York
Sarah	Ackerman	440 U.S. 110	Upton	Michigan

To explain what you just did, you asked for the all of data in the EmployeeAddressTable, and specifically, you asked for the *columns* called FirstName, LastName, Address, City, and State. Note that column names and table names do not have spaces...they must be typed as one word; and that the statement ends with a semicolon (;). The general form for a SELECT statement, retrieving all of the *rows* in the table is:

```
SELECT ColumnName, ColumnName, ...
FROM TableName;
```

To get all columns of a table without typing all column names, use:

```
SELECT * FROM TableName;
```

Each database management system (DBMS) and database software has different methods for logging in to the database and entering SQL commands; see the local computer "guru" to help you get onto the system, so that you can use SQL.

Conditional Selection

To further discuss the SELECT statement, let's look at a new example table (for hypothetical purposes only):

EmployeeStatisticsTable			
EmployeeIDNo	**Salary**	**Benefits**	**Position**
010	**75000**	**15000**	**Manager**
105	**65000**	**15000**	**Manager**
152	**60000**	**15000**	**Manager**
215	**60000**	**12500**	**Manager**
244	**50000**	**12000**	**Staff**
300	**45000**	**10000**	**Staff**
335	**40000**	**10000**	**Staff**
400	**32000**	**7500**	**Entry-Level**
441	**28000**	**7500**	**Entry-Level**

Relational Operators

There are six Relational Operators in SQL, and after introducing them, we'll see how they're used:

=	Equal
<> or != (see manual)	Not Equal
<	Less Than
>	Greater Than
<=	Less Than or Equal To
>=	Greater Than or Equal To

The *WHERE* clause is used to specify that only certain rows of the table are displayed, based on the criteria described in that *WHERE clause*. It is most easily understood by looking at a couple of examples.

If you wanted to see the EMPLOYEEIDNO's of those making at or over $50,000, use the following:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY >= 50000;
```

Notice that the >= (greater than or equal to) sign is used, as we wanted to see those who made greater than $50,000, or equal to $50,000, listed together. This displays:

```
EMPLOYEEIDNO
------------
010
105
152
215
244
```

The *WHERE* description, SALARY >= 50000, is known as a *condition* (an operation which evaluates to True or False). The same can be done for text columns:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager';
```

This displays the ID Numbers of all Managers. Generally, with text columns, stick to equal to or not equal to, and make sure that any text that appears in the statement is surrounded by single quotes (').

More Complex Conditions: Compound Conditions / Logical Operators

The *AND* operator joins two or more conditions, and displays a row only if that row's data satisfies **ALL** conditions listed (i.e. all conditions hold true). For example, to display all staff making over $40,000, use:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY > 40000 AND POSITION = 'Staff';
```

The *OR* operator joins two or more conditions, but returns a row if **ANY** of the conditions listed hold true. To see all those who make less than $40,000 or have less than $10,000 in benefits, listed together, use the following query:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY < 40000 OR BENEFITS < 10000;
```

AND & OR can be combined, for example:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager' AND SALARY > 60000 OR
BENEFITS > 12000;
```

First, SQL finds the rows where the salary is greater than $60,000 and the position column is equal to Manager, then taking this new list of rows, SQL then sees if any of these rows satisfies the previous AND condition or the condition that the Benefits column is greater then $12,000. Subsequently, SQL only displays this second new list of rows, keeping in mind that anyone with Benefits over $12,000 will be included as the OR operator includes a row if either resulting condition is True. Also note that the AND operation is done first.

To generalize this process, SQL performs the AND operation(s) to determine the rows where the AND operation(s) hold true (remember: all of the conditions are true), then these results are used to compare with the OR conditions, and only display those remaining rows where any of the conditions joined by the OR operator hold true (where a condition or result from an AND is paired with another condition or AND result to use to evaluate the OR, which evaluates to true if either value is true). Mathematically, SQL evaluates all of the conditions, then evaluates the AND "pairs", and then evaluates the OR's (where both operators evaluate left to right).

To look at an example, for a given row for which the DBMS is evaluating the SQL statement Where clause to determine whether to include the row in the query result (the whole Where clause evaluates to True), the DBMS has evaluated all of the

conditions, and is ready to do the logical comparisons on this result:

True AND False OR True AND True OR False AND False

First simplify the AND pairs:

False OR True OR False

Now do the OR's, left to right:

True OR False
True

The result is True, and the row passes the query conditions. Be sure to see the next section on NOT's, and the order of logical operations. I hope that this section has helped you understand AND's or OR's, as it's a difficult subject to explain briefly (especially when you write a version and the editor loses the changes-on multiple occasions no less!).

To perform OR's before AND's, like if you wanted to see a list of employees making a large salary (>$50,000) or have a large benefit package (>$10,000), and that happen to be a manager, use parentheses:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager' AND (SALARY > 50000
OR BENEFIT > 10000);
```

IN & BETWEEN

An easier method of using compound conditions uses *IN* or *BETWEEN*. For example, if you wanted to list all managers and staff:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION IN ('Manager', 'Staff');
```

or to list those making greater than or equal to $30,000, but less than or equal to $50,000, use:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY BETWEEN 30000 AND 50000;
```

To list everyone not in this range, try:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY NOT BETWEEN 30000 AND 50000;
```

Similarly, NOT IN lists all rows excluded from the *IN* list.

Additionally, NOT's can be thrown in with AND's & OR's, except that NOT is a unary operator (evaluates one condition, reversing its value, whereas, AND's & OR's evaluate two conditions), and that all NOT's are performed before any AND's or OR's.

SQL Order of Logical Operations (each operates from left to right)

1. NOT
2. AND
3. OR

Using *LIKE*

Look at the EmployeeStatisticsTable, and say you wanted to see all people whose last names started with "L"; try:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE LASTNAME LIKE 'L%';
```

The percent sign (%) is used to represent any possible character (number, letter, or punctuation) or set of characters that might appear after the "L". To find those people with LastName's ending in "L", use '%L', or if you wanted the "L" in the middle of the word, try '%L%'. The '%' can be used for any characters in the same position relative to the given characters. NOT LIKE displays rows not fitting the given description. Other possiblities of using LIKE, or any of these discussed conditionals, are available, though it depends on what DBMS you are using; as usual, consult a manual or your system manager or administrator for the available features on your system, or just to make sure that what you are trying to do is available and allowed. This disclaimer holds for the features of SQL that will be discussed below. This section is just to give you an idea of the possibilities of queries that can be written in SQL.

Joins

In this section, we will only discuss *inner* joins, and *equijoins*, as in general, they are the most useful. For more information, try the SQL links at the bottom of the page.

Good database design suggests that each table lists data only about a single *entity*, and detailed information can be obtained in a relational database, by using additional tables, and by using a *join*.

First, take a look at these example tables:

Antique Owners		
OwnerID	**OwnerLastName**	**OwnerFirstName**
01	Jones	Bill
02	Smith	Bob
15	Lawson	Patricia
21	Akins	Jane
50	Fowler	Sam

Orders	
OwnerID	**Item Desired**
02	Table
02	Desk
21	Chair
15	Mirror

Antiques		
SellerID	**BuyerID**	**Item**
01	50	Bed
02	15	Table
15	02	Chair
21	50	Mirror
50	01	Desk
01	21	Cabinet
02	21	Coffee Table
15	50	Chair
01	15	Jewelry Box
02	21	Pottery
21	02	Bookcase
50	01	Plant Stand

Keys

First, let's discuss the concept of *keys*. A *primary key* is a column or set of columns that uniquely identifies the rest of the data in any given row. For example, in the AntiqueOwners table, the OwnerID column uniquely identifies that row. This means two things: no two rows can have the same OwnerID, and, even if two owners have the same first and last names, the OwnerID column ensures that the two owners will not be confused with each other, because the unique OwnerID column will be used throughout the database to track the owners, rather than the names.

A *foreign key* is a column in a table where that column is a primary key of another table, which means that any data in a foreign key column must have corresponding data in the other table where that column is the primary key. In DBMS-speak, this correspondence is known as *referential integrity*. For example, in the Antiques table, both the BuyerID and SellerID are foreign keys to the primary key of the AntiqueOwners table (OwnerID; for purposes of argument, one has to be an Antique Owner before one can buy or sell any items), as, in both tables, the ID rows are used to identify the owners or buyers and sellers, and that the OwnerID is the primary key of the AntiqueOwners table. In other words, all of this "ID" data is used to refer to the owners, buyers, or sellers of antiques, themselves, without having to use the actual names.

Performing a Join

The purpose of these *keys* is so that data can be related across tables, without having to repeat data in every table—this is the power of relational databases. For example, you can find the names of those who bought a chair without having to list the full name of the buyer in the Antiques table...you can get the name by relating those who bought a chair with the names in the AntiqueOwners table through the use of the OwnerID, which *relates* the data in the two tables. To find the names of those who bought a chair, use the following query:

```
SELECT OWNERLASTNAME, OWNERFIRSTNAME
FROM ANTIQUEOWNERS, ANTIQUES
WHERE BUYERID = OWNERID AND ITEM = 'Chair';
```

Note the following about this query...notice that both tables involved in the relation are listed in the FROM clause of the statement. In the WHERE clause, first notice that the ITEM = 'Chair' part restricts the listing to those who have bought (and in this example, thereby owns) a chair. Secondly, notice how the ID columns are related from one table to the next by use of the BUYERID = OWNERID clause. Only where ID's match across tables and the item purchased is a chair (because of the AND), will the names from the AntiqueOwners table be listed. Because the joining condition used an equal sign, this join is called an *equijoin*. The result of this query is two names: Smith, Bob & Fowler, Sam.

Dot notation refers to prefixing the table names to column names, to avoid ambiguity, as such:

```
SELECT ANTIQUEOWNERS.OWNERLASTNAME,
ANTIQUEOWNERS.OWNERFIRSTNAME
FROM ANTIQUEOWNERS, ANTIQUES
WHERE ANTIQUES.BUYERID =
ANTIQUEOWNERS.OWNERID AND ANTIQUES.ITEM
= 'Chair';
```

As the column names are different in each table, however, this wasn't necessary.

DISTINCT and Eliminating Duplicates

Let's say that you want to list the ID and names of only those people who have sold an antique. Obviously, you want a list where each seller is only listed once-you don't want to know how many antiques a person sold, just the fact that this person sold one (for counts, see the Aggregate Function section below). This means that you will need to tell SQL to eliminate duplicate sales rows, and just list each person only once. To do this, use the DISTINCT keyword.

First, we will need an equijoin to the AntiqueOwners table to get the detail data of the person's LastName and FirstName. However, keep in mind that since the SellerID column in the Antiques table is a foreign key to the AntiqueOwners table, a seller will only be listed if there is a row in the AntiqueOwners table listing the ID and names. We also want to eliminate multiple occurences of the SellerID in our listing, so we use *DISTINCT* **on the column where the repeats may occur.**

To throw in one more twist, we will also want the list alphabetized by LastName, then by FirstName (on a LastName tie). Thus, we will use the *ORDER BY* clause:

```
SELECT DISTINCT SELLERID, OWNERLASTNAME,
OWNERFIRSTNAME
FROM ANTIQUES, ANTIQUEOWNERS
WHERE SELLERID = OWNERID
ORDER BY OWNERLASTNAME, OWNERFIRSTNAME;
```

In this example, since everyone has sold an item, we will get a listing of all of the owners, in alphabetical order by last name. For future reference (and in case anyone asks), this type of join is considered to be in the category of *inner joins.*

Aliases and In/Subqueries

In this section, we will talk about *Aliases*, *In* and the use of subqueries, and how these can be used in a 3-table example. First, look at this query which prints the last name of those owners who have placed an order and what the order is, only listing those orders which can be filled (that is, there is a buyer who owns that ordered item):

```
SELECT OWN.OWNERLASTNAME Last Name,
ORD.ITEMDESIRED Item Ordered
FROM ORDERS ORD, ANTIQUEOWNERS OWN
WHERE ORD.OWNERID = OWN.OWNERID
AND ORD.ITEMDESIRED IN

(SELECT ITEM
FROM ANTIQUES);
```

This gives:

Last Name	Item Ordered
Smith	Table
Smith	Desk
Akins	Chair
Lawson	Mirror

There are several things to note about this query:

1. First, the "Last Name" and "Item Ordered" in the Select lines gives the headers on the report.
2. The OWN & ORD are aliases; these are new names for the two tables listed in the FROM clause that are used as prefixes for all dot notations of column names in the query (see above). This eliminates ambiguity, especially in the equijoin WHERE clause where both tables have the column named OwnerID, and the dot notation tells SQL that we are talking about two different OwnerID's from the two different tables.
3. Note that the Orders table is listed first in the FROM clause; this makes sure listing is done off of that table, and the AntiqueOwners table is only used for the detail information (Last Name).
4. Most importantly, the AND in the WHERE clause forces the In Subquery to be invoked ("= ANY" or "= SOME" are two equivalent uses of IN). What this does is, the subquery is performed, returning all of the Items owned from the Antiques table, as there is no WHERE clause. Then, for a row from the Orders table to be listed, the ItemDesired must be in that returned list of Items owned from the Antiques table, thus listing an item only if the order can be filled from another owner. You can think of it this way: the subquery returns a *set* of Items from which each ItemDesired in the Orders table is compared; the In condition is true only if the ItemDesired is in that returned set from the Antiques table.
5. Also notice, that in this case, that there happened to be an antique available for each one desired...obviously, that won't always be the case. In addition, notice that when the IN, "= ANY", or "= SOME" is used, that these keywords refer to any possible row matches, not column matches...that is, you cannot put multiple columns in the subquery Select clause, in an attempt to match the column in the outer Where clause to one of multiple possible column values in the subquery; only one column can be listed in the subquery, and the possible match comes from multiple *row* values in that *one* column, not vice-versa.

Whew! That's enough on the topic of complex SELECT queries for now. Now on to other SQL statements.

Miscellaneous SQL Statements

Aggregate Functions

I will discuss five important *aggregate functions*: SUM, AVG, MAX, MIN, and COUNT. They are called aggregate functions because they summarize the results of a query, rather than listing all of the rows.

- SUM () gives the total of all the rows, satisfying any conditions, of the given column, where the given column is numeric.
- AVG () gives the average of the given column.
- MAX () gives the largest figure in the given column.
- MIN () gives the smallest figure in the given column.
- COUNT(*) gives the number of rows satisfying the conditions.

Looking at the tables at the top of the document, let's look at three examples:

```
SELECT SUM(SALARY), AVG(SALARY)
FROM EMPLOYEEST ATISTICS TABLE;
```

This query shows the total of all salaries in the table, and the average salary of all of the entries in the table.

```
SELECT MIN(BENEFITS)
FROM EMPLOYEEST ATISTICS TABLE
WHERE POSITION = 'Manager';
```

This query gives the smallest figure of the Benefits column, of the employees who are Managers, which is 12500.

```
SELECT COUNT(*)
FROM EMPLOYEES TATISTICS TABLE
WHERE POSITION = 'Staff';
```

This query tells you how many employees have Staff status (3).

Views

In SQL, you might (check your DBA) have access to create views for yourself. What a view does is to allow you to assign the results of a query to a new, personal table, that you can use in other queries, where this new table is given the view name in your FROM clause. When you access a view, the query that is defined in your view creation statement is performed (generally), and the results of that query look just like another table in the query that you wrote invoking the view. For example, to create a view:

Create View Antview As Select Itemdesired From Orders;
Now, write a query using this view as a table, where the table is just a listing of all Items Desired from the Orders table:

```
SELECT SELLERID
FROM ANTIQUES, ANTVIEW
WHERE ITEM DESIRED = ITEM;
```

This query shows all SellerID's from the Antiques table where the Item in that table happens to appear in the Antview view, which is just all of the Items Desired in the Orders table. The listing is generated by going through the Antique Items one-by-one until there's a match with the Antview view. Views can be used to restrict database access, as well as, in this case, simplify a complex query.

Creating New Tables

All tables within a database must be created at some point in time...let's see how we would create the Orders table:

```
CREATE TABLE ORDERS
(OWNERID INTEGER NOT NULL,
ITEM DESIRED CHAR(40) NOT NULL);
```

This statement gives the table name and tells the DBMS about each column in the table. *Please note* that this statement uses generic data types, and that the data types might be different, depending on what DBMS you are using. As usual, check local listings. Some common generic data types are:

- Char(x) - A column of characters, where x is a number designating the maximum number of characters allowed (maximum length) in the column.
- Integer - A column of whole numbers, positive or negative.
- Decimal(x, y) - A column of decimal numbers, where x is the maximum length in digits of the decimal numbers in this column, and y is the maximum number of digits allowed after the decimal point. The maximum (4,2) number would be 99.99.
- Date - A date column in a DBMS-specific format.
- Logical - A column that can hold only two values: TRUE or FALSE.

One other note, the NOT NULL means that the column must have a value in each row. If NULL was used, that column may be left empty in a given row.

Altering Tables

Let's add a column to the Antiques table to allow the entry of the price of a given Item:

```
Alter Table Antiques Add (Price Decimal(8,2) Null);
```

The data for this new column can be updated or inserted as shown later.

Adding Data

To insert rows into a table, do the following:

```
INSERT INTO ANTIQUES VALUES (21, 01, 'Ottoman', 200.00);
```

This inserts the data into the table, as a new row, column-by-column, in the pre-defined order. Instead, let's change the order and leave Price blank:

```
INSERT INTO ANTIQUES (BUYERID, SELLERID, ITEM)
VALUES (01, 21, 'Ottoman');
```

Deleting Data

Let's delete this new row back out of the database:

```
DELETE FROM ANTIQUES
WHERE ITEM = 'Ottoman';
```

But if there is another row that contains 'Ottoman', that row will be deleted also. Let's delete all rows (one, in this case) that contain the specific data we added before:

```
DELETE FROM ANTIQUES
WHERE ITEM = 'Ottoman' AND BUYERID = 01 AND
SELLERID = 21;
```

Updating Data

Let's update a Price into a row that doesn't have a price listed yet:

```
UPDATE ANTIQUES SET PRICE = 500.00 WHERE ITEM
= 'Chair';
```

This sets all Chair's Prices to 500.00. As shown above, more WHERE conditionals, using AND, must be used to limit the updating to more specific rows. Also, additional columns may be set by separating equal statements with commas.

Miscellaneous Topics

Indexes

Indexes allow a DBMS to access data quicker (*please note:* this feature is nonstandard/not available on all systems). The system creates this internal data structure (the index) which causes selection of rows, when the selection is based on indexed columns, to occur faster. This index tells the DBMS where a certain row is in the table given an indexed-column value, much like a book index tells you what page a given word appears. Let's create an index for the OwnerID in the AntiqueOwners column:

```
CREATE INDEX OID_IDX ON ANTIQUEOWNERS
(OWNERID);
```

Now on the Names:

```
CREATE INDEX NAME_IDX ON ANTIQUEOWNERS
(OWNERLASTNAME, OWNERFIRSTNAME);
```

To get rid of an index, drop it:

```
DROP INDEX OID_IDX;
```

By the way, you can also "drop" a table, as well (careful!-that means that your table is deleted). In the second example, the index is kept on the two columns, aggregated together-strange behavior might occur in this situation...check the manual before performing such an operation.

Some DBMS's do not enforce primary keys; in other words, the uniqueness of a column is not enforced automatically. What that means is, if, for example, I tried to insert another row into the AntiqueOwners table with an OwnerID of 02, some systems will allow me to do that, even though, we do not, as that column is supposed to be unique to that table (every row value is supposed to be different). One way to get around that is to create a unique index on the column that we want to be a primary key, to force the system to enforce prohibition of duplicates:

```
CREATE UNIQUE INDEX OID_IDX ON
ANTIQUEOWNERS (OWNERID);
```

Group By And Having

One special use of GROUP BY is to associate an aggregate function (especially COUNT; counting the number of rows in each group) with groups of rows. First, assume that the Antiques table has the Price column, and each row has a value for that column. We want to see the price of the most expensive item bought by each owner. We have to tell SQL to *group* each owner's purchases, and tell us the maximum purchase price:

```
SELECT BUYERID, MAX(PRICE)
FROM ANTIQUES
GROUP BY BUYERID;
```

Now, say we only want to see the maximum purchase price if the purchase is over $1000, so we use the HAVING clause:

```
SELECT BUYERID, MAX(PRICE)
FROM ANTIQUES
GROUP BY BUYERID
HAVING PRICE > 1000;
```

More Subqueries

Another common usage of subqueries involves the use of operators to allow a Where condition to include the Select output of a subquery. First, list the buyers who purchased an expensive item (the Price of the item is $100 greater than the average price of all items purchased):

```
SELECT BUYERID
FROM ANTIQUES
WHERE PRICE >

(SELECT AVG(PRICE) + 100
FROM ANTIQUES);
```

The subquery calculates the average Price, plus $100, and using that figure, an OwnerID is printed for every item costing over that figure. One could use DISTINCT BUYERID, to eliminate duplicates.

List the Last Names of those in the AntiqueOwners table, ONLY if they have bought an item:

```
SELECT OWNERLASTNAME
FROM ANTIQUEOWNERS
WHERE OWNERID IN

(SELECT DISTINCT BUYERID
FROM ANTIQUES);
```

The subquery returns a list of buyers, and the Last Name is printed for an Antique Owner if and only if the Owner's ID appears in the subquery list (sometimes called a *candidate list*). *Note:* on some DBMS's, equals can be used instead of IN, but for clarity's sake, since a set is returned from the subquery, IN is the better choice.

For an Update example, we know that the gentleman who bought the bookcase has the wrong First Name in the database...it should be John:

```
UPDATE ANTIQUEOWNERS
SET OWNERFIRSTNAME = 'John'
WHERE OWNERID =

(SELECT BUYERID
FROM ANTIQUES
WHERE ITEM = 'Bookcase');
```

First, the subquery finds the BuyerID for the person(s) who bought the Bookcase, then the outer query updates his First Name.

Remember this rule about subqueries: when you have a subquery as part of a WHERE condition, the Select clause in the subquery must have columns that match in number and type to those in the Where clause of the outer query. In other words, if you have "WHERE ColumnName = (SELECT...);", the Select must have only one column in it, to match the ColumnName in the outer Where clause, *and* they must match in type (both being integers, both being character strings, etc.).

Exists And All

EXISTS uses a subquery as a condition, where the condition is True if the subquery returns any rows, and False if the subquery does not return any rows; this is a nonintuitive feature with few unique uses. However, if a prospective customer wanted to see the list of Owners only if the shop dealt in Chairs, try:

```
SELECT OWNERFIRSTNAME, OWNERLASTNAME
FROM ANTIQUEOWNERS
WHERE EXISTS

(SELECT *
FROM ANTIQUES
WHERE ITEM = 'Chair');
```

If there are any Chairs in the Antiques column, the subquery would return a row or rows, making the EXISTS clause true, causing SQL to list the Antique Owners. If there had been no Chairs, no rows would have been returned by the outside query.

ALL is another unusual feature, as ALL queries can usually be done with different, and possibly simpler methods; let's take a look at an example query:

```
SELECT BUYERID, ITEM
FROM ANTIQUES
WHERE PRICE >= ALL

(SELECT PRICE
FROM ANTIQUES);
```

This will return the largest priced item (or more than one item if there is a tie), and its buyer. The subquery returns a list of all Prices in the Antiques table, and the outer query goes through each row of the Antiques table, and if its Price is greater than or equal to every (or ALL) Prices in the list, it is listed, giving the highest priced Item. The reason ">=" must be used is that the highest priced item will be equal to the highest price on the list, because this Item is in the Price list.

Union And Outer Joins (Briefly Explained)

There are occasions where you might want to see the results of multiple queries together, combining their output; use UNION. To merge the output of the following two queries, displaying the ID's of all Buyers, plus all those who have an Order placed:

```
SELECT BUYERID
FROM ANTIQUES
UNION
```

```
SELECT OWNERID
FROM ORDERS;
```

Notice that SQL requires that the Select list (of columns) must match, column-by-column, in data type. In this case BuyerID and OwnerID are of the same data type (integer). Also notice that SQL does automatic duplicate elimination when using UNION (as if they were two "sets"); in single queries, you have to use DISTINCT.

The *outer join* is used when a join query is "united" with the rows not included in the join, and are especially useful if constant text "flags" are included. First, look at the query:

```
SELECT OWNERID, 'is in both Orders & Antiques'
FROM ORDERS, ANTIQUES
WHERE OWNERID = BUYERID
UNION
SELECT BUYERID, 'is in Antiques only'
FROM ANTIQUES
WHERE BUYERID NOT IN
(SELECT OWNERID
FROM ORDERS);
```

The first query does a join to list any owners who are in both tables, and putting a tag line after the ID repeating the quote. The UNION merges this list with the next list. The second list is generated by first listing those ID's not in the Orders table, thus generating a list of ID's excluded from the join query. Then, each row in the Antiques table is scanned, and if the BuyerID is not in this exclusion list, it is listed with its quoted tag. There might be an easier way to make this list, but it's difficult to generate the informational quoted strings of text.

This concept is useful in situations where a primary key is related to a foreign key, but the foreign key value for some primary keys is NULL. For example, in one table, the primary key is a salesperson, and in another table is customers, with their salesperson listed in the same row. However, if a salesperson has no customers, that person's name won't appear in the customer table. The outer join is used if the listing of all salespersons is to be printed, listed with their customers, whether the salesperson has a customer or not-that is, no customer is printed (a logical NULL value) if the salesperson has no customers, but is in the salespersons table. Otherwise, the salesperson will be listed with each customer.

Another important related point about Nulls having to do with joins: the order of tables listed in the From clause is very important. The rule states that SQL "adds" the second table to the first; the first table listed has any rows where there is a null on the join column displayed; if the second table has a row with a null on the join column, that row from the table listed second does not get joined, and thus included with the first table's row data. This is another occasion (should you wish that data included in the result) where an outer join is commonly used. The concept of nulls is important, and it may be worth your time to investigate them further.

ENOUGH QUERIES!!! you say?...now on to something completely different...

Embedded SQL—an ugly example (do not write a program like this...for purposes of argument ONLY)

- To get right to it, here is an example program that uses Embedded SQL. Embedded SQL allows programmers to connect to a database and include SQL code right in the program, so that their programs can use, manipulate, and process data from a database.
- This example C Program (using Embedded SQL) will print a report. This program will have to be precompiled for the SQL statements, before regular compilation.
- The EXEC SQL parts are the same (standard), but the surrounding C code will need to be changed, including the host variable declarations, if you are using a different language.
- Embedded SQL changes from system to system, so, once again, check local documentation, especially variable declarations and logging in procedures, in which network, DBMS, and operating system considerations are crucial.
- **This program is not compilable or executable**
- **It is for example purposes only**

```
#include <stdio.h>

/* This section declares the host variables; these will be the
   variables your program uses, but also the variable SQL will
put values in or take values out. */
EXEC SQL BEGIN DECLARE SECTION;
  int BuyerID;
  char FirstName[100], LastName[100], Item[100];
EXEC SQL END DECLARE SECTION;

/* This includes the SQLCA variable, so that some error
checking can be done. */
EXEC SQL INCLUDE SQLCA;

main() {

/* This is a possible way to log into the database */
EXEC SQL CONNECT UserID/Password;

/* This code either says that you are connected or checks if an
error code was generated, meaning log in was incorrect or not
possible. */  if(sqlca.sqlcode) { printf(Printer, "Error connect-
ing to database server.\n");
    exit();
  }
  printf("Connected to database server.\n");

/* This declares a "Cursor". This is used when a query returns
more
   than one row, and an operation is to be performed on each
row
   resulting from the query. With each row established by this
query,
   I'm going to use it in the report. Later, "Fetch" will be used to
   pick off each row, one at a time, but for the query to actually
   be executed, the "Open" statement is used. The "Declare" just
   establishes the query. */
EXEC SQL DECLARE ItemCursor CURSOR FOR
  SELECT ITEM, BUYERID
  FROM ANTIQUES
```

```
ORDER BY ITEM;
EXEC SQL OPEN ItemCursor;

/* +— You may wish to put a similar error checking block here
—+ */

/* Fetch puts the values of the "next" row of the query in the
host variables, respectively. However, a "priming fetch"
(programming technique) must first be done. When the cursor
is out of data, a sqlcode will be generated allowing us to leave
the loop. Notice that, for simplicity's sake, the loop will leave on
any sqlcode, even if it is an error code. Otherwise, specific code
checking must be performed. */
EXEC SQL FETCH ItemCursor INTO :Item, :BuyerID;
  while(!sqlca.sqlcode) {

/* With each row, we will also do a couple of things. First,
bump the price up by $5 (dealer's fee) and get the buyer's name
to put in  the report. To do this, I'll use an Update and a Select,
before printing the line on the screen. The update assumes
however, that a given buyer has only bought one of any given
item, or else the price will be increased too many times.
Otherwise, a "RowID" logic would have to be used (see
documentation). Also notice the colon     before host variable
names when used inside of SQL statements. */

EXEC SQL UPDATE ANTIQUES
  SET PRICE = PRICE + 5
  WHERE ITEM = :Item AND BUYERID = :BuyerID;

EXEC SQL SELECT OWNERFIRSTNAME,
OWNERLASTNAME
  INTO :FirstName, :LastName
  FROM ANTIQUEOWNERS
  WHERE BUYERID = :BuyerID;

    printf("%25s %25s %25s", FirstName, LastName, Item);

/* Ugly report—for example purposes only! Get the next row.
*/
EXEC SQL FETCH ItemCursor INTO :Item, :BuyerID;
  }

/* Close the cursor, commit the changes (see below), and exit
the
   program. */
EXEC SQL CLOSE ItemCursor;
EXEC SQL COMMIT RELEASE;
  exit();
}
```

Common SQL Questions-Advanced Topics (see FAQ link for several more)

1. Why can't I just ask for the first three rows in a table? -Because in relational databases, rows are inserted in no particular order, that is, the system inserts them in an arbitrary order; so, you can only request rows using valid SQL features, like ORDER BY, etc.
2. What is this DDL and DML I hear about? -DDL (Data Definition Language) refers to (in SQL) the Create Table statement...DML (Data Manipulation Language) refers to the Select, Update, Insert, and Delete statements.
3. Aren't database tables just files? -Well, DBMS's store data in files declared by system managers before new tables are created (on large systems), but the system stores the data in a special format, and may spread data from one table over several files. In the database world, a set of files created for a database is called a *tablespace*. In general, on small systems, everything about a database (definitions and all table data) is kept in one file.
4. (Related question) Aren't database tables just like spreadsheets? -No, for two reasons. First, spreadsheets can have data in a cell, but a cell is more than just a row-column-intersection. Depending on your spreadsheet software, a cell might also contain formulas and formatting, which database tables cannot have (currently). Secondly, spreadsheet cells are often dependent on the data in other cells. In databases, "cells" are independent, except that columns are logically related (hopefully; together a row of columns describe an entity), and, other than primary key and foreign key constraints, each row in a table is independent from one another.
5. How do I import a text file of data into a database? -Well, you can't do it directly...you must use a utility, such as Oracle's SQL*Loader, or write a program to load the data into the database. A program to do this would simply go through each record of a text file, break it up into columns, and do an Insert into the database.
6. What web sites and computer books would you recommend for more information about SQL and databases? -First, look at the sites at the bottom of this page. I would especially suggest the following: Ask the SQL Pro (self-explanatory), DB Ingredients (more theorical topics), DBMS Lab/Links (comprehensive academic DBMS link listing), Access on the Web (about web access of Access databases), Tutorial Page (listing of other tutorials), and miniSQL (more information about the best known free DBMS). Unfortunately, there is not a great deal of information on the web about SQL; the list I have below is fairly comprehensive (definitely representative).

   As far as books are concerned (go to amazon.com or Barnes & Noble for more information), I would suggest (for beginners to intermediate-level) "Oracle: The Complete Reference" from Oracle and "Understanding SQL" from Sybex for general SQL information. Also, I would recommend O'Reilly Publishing's books, and Joe Celko's writings for advanced users. Additionally, I would suggest mcp.com for samples of computer books, and for specific DBMS info (especially in the Access area), I recommend Que's "Using" series, and the books of Alison Balter (search for these names at the bookstore sites for a list of titles).
7. What is a *schema*? -A schema is a logical set of tables, such as the Antiques database above...usually, it is thought of as simply "the database", but a database can hold more than one schema. For example, a *star schema* is a set of tables where one large, central table holds all of the important information, and is linked, via foreign keys, to *dimension* tables which hold detail information, and can be used in a join to create detailed reports.

8. Show me an example of an *outer join.* -Well, from the questions I receive, this is an extremely common example, and I'll show you both the Oracle and Access queries...

Think of the following Employee table (the employees are given numbers, for simplicity):

Name	Department
1	10
2	10
3	20
4	30
5	30

Now suppose you want to join the tables, seeing all of the employees and all of the departments together...you'll have to use an outer join which includes a null employee to go with Dept. 40.

Department
10
20
30
40

In the book, "Oracle 7: the Complete Reference", about outer joins, "think of the (+), which must immediately follow the join column of the table, as saying add an extra (null) row anytime there's no match". So, in Oracle, try this query (the + goes on Employee, which adds the null row on no match):

```
Select E.Name, D.Department
From Department D, Employee E
Where E.Department(+) = D.Department;
```

This is a left (outer) join, in Access:

```
SELECT DISTINCTROW Employee.Name,
Department.Department
FROM Department LEFT JOIN Employee ON
Department.Department = Employee.Department;
```

And you get this result:

Name	Department
1	10
2	10
3	20
4	30
5	30
	40

9. What are some general tips you would give to make my SQL queries and databases better and faster (*optimized*)?

- You should try, if you can, to avoid expressions in Selects, such as SELECT ColumnA + ColumnB, etc. The *query optimizer* of the database, the portion of the DBMS that determines the best way to get the required data out of the database itself, handles expressions in such a way that would normally require more time to retrieve the data than if columns were normally selected, and the expression itself handled programmatically.
- Minimize the number of columns included in a Group By clause.
- If you are using a join, try to have the columns joined on (from both tables) indexed.
- When in doubt, index.
- Unless doing multiple counts or a complex query, use COUNT(*) (the number of rows generated by the query) rather than COUNT(Column_Name).

10. What is *normalization*?-Normalization is a technique of database design that suggests that certain criteria be used when constructing a table layout (deciding what columns each table will have, and creating the key structure), where the idea is to eliminate redundancy of non-key data across tables. Normalization is usually referred to in terms of *forms*, and I will introduce only the first three, even though it is somewhat common to use other, more advanced forms (fourth, fifth, Boyce-Codd; see documentation).

First Normal Form refers to moving data into separate tables where the data in each table is of a similar type, and by giving each table a primary key.

Putting data in *Second Normal Form* involves removing to other tables data that is only dependent of a part of the key. For example, if I had left the names of the Antique Owners in the items table, that would not be in Second Normal Form because that data would be redundant; the name would be repeated for each item owned; as such, the names were placed in their own table. The names themselves don't have anything to do with the items, only the identities of the buyers and sellers.

Third Normal Form involves getting rid of anything in the tables that doesn't depend solely on the primary key. Only include information that is dependent on the key, and move off data to other tables that are independent of the primary key, and create a primary keys for the new tables.

There is some redundancy to each form, and if data is in *3NF* (shorthand for 3rd normal form), it is already in *1NF* and *2NF*. In terms of data design then, arrange data so that any non-primary key columns are dependent only on the *whole primary key*. If you take a look at the sample database, you will see that the way then to navigate through the database is through joins using common key columns.

Two other important points in database design are using good, consistent, logical, full-word names for the tables and columns, and the use of full words in the database

itself. On the last point, my database is lacking, as I use numeric codes for identification. It is usually best, if possible, to come up with keys that are, by themselves, self-explanatory; for example, a better key would be the first four letters of the last name and first initial of the owner, like JONEB for Bill Jones (or for tiebreaking purposes, add numbers to the end to differentiate two or more people with similar names, so you could try JONEB1, JONEB2, etc.).

11. What is the difference between a *single-row query* and a *multiple-row query* and why is it important to know the difference? —First, to cover the obvious, a single-row query is a query that returns one row as its result, and a multiple-row query is a query that returns more than one row as its result. Whether a query returns one row or more than one row is entirely dependent on the design (or *schema*) of the tables of the database. As query-writer, you must be aware of the schema, be sure to include enough conditions, and structure your SQL statement properly, so that you will get the desired result (either one row or multiple rows). For example, if you wanted to be sure that a query of the AntiqueOwners table returned only one row, consider an equal condition of the primary key-column, OwnerID.

    Three reasons immediately come to mind as to why this is important. First, getting multiple rows when you were expecting only one, or vice-versa, may mean that the query is erroneous, that the database is incomplete, or simply, you learned something new about your data. Second, if you are using an update or delete statement, you had better be sure that the statement that you write performs the operation on the desired row (or rows)...or else, you might be deleting or updating more rows than you intend. Third, any queries written in Embedded SQL must be carefully thought out as to the number of rows returned. If you write a single-row query, only one SQL statement may need to be performed to complete the programming logic required. If your query, on the other hand, returns multiple rows, you will have to use the Fetch statement, and quite probably, some sort of looping structure in your program will be required to iterate processing on each returned row of the query.

12. Tell me about a simple approach to relational database design. —This was sent to me via a news posting; it was submitted by John Frame (jframe@jframe.com) and Richard Freedman (rfreedm@voicenet.com); I offer a shortened version as advice, but I'm not responsible for it, and some of the concepts are readdressed in the next question...

    First, create a list of important things (entities) and include those things you may not initially believe is important. Second, draw a line between any two entities that have any connection whatsoever; except that no two entities can connect without a 'rule'; e.g.: families have children, employees work for a department. Therefore put the 'connection' in a diamond, the 'entities' in squares. Third, your picture should now have many squares (entities) connected to other entities through diamonds (a square enclosing an entity, with a line to a diamond describing the relationship, and then another line to the other entity). Fourth, put descriptors on each square and each diamond, such as customer - airline - trip. Fifth, give each diamond and square any attributes it may have (a person has a name, an invoice has a number), but some relationships have none (a parent just owns a child). Sixth, everything on your page that has attributes is now a table, whenever two entities have a relationship where the relationship has no attributes, there is merely a foreign key between the tables. Seventh, in general you want to make tables not repeat data. So, if a customer has a name and several addresses, you can see that for every address of a customer, there will be repeated the customer's first name, last name, etc. So, record Name in one table, and put all his addresses in another. Eighth, each row (record) should be unique from every other one; Mr. Freedman suggests a 'auto-increment number' primary key, where a new, unique number is generated for each new inserted row. Ninth, a key is any way to uniquely identify a row in a table...first and last name together are good as a 'composite' key. That's the technique.

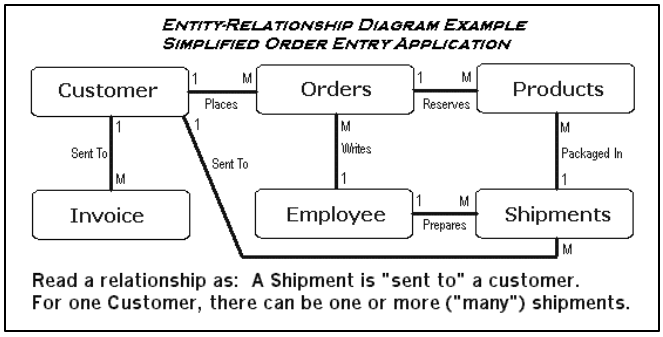
13. What are *relationships?* -Another design question...the term "relationships" (often termed "relation") usually refers to the relationships among primary and foreign keys between tables. This concept is important because when the tables of a relational database are designed, these relationships must be defined because they determine which columns are or are not primary or foreign keys. You may have heard of an **Entity-Relationship Diagram**, which is a graphical view of tables in a database schema, with lines connecting related columns across tables. See the sample diagram at the end of this section or some of the sites below in regard to this topic, as there are many different ways of drawing E-R diagrams. But first, let's look at each kind of relationship...

    A *One-to-one relationship* means that you have a primary key column that is related to a foreign key column, and that for every primary key value, there is one foreign key value. For example, in the first example, the EmployeeAddressTable, we add an EmployeeIDNo column. Then, the EmployeeAddressTable is related to the EmployeeStatisticsTable (second example table) by means of that EmployeeIDNo. Specifically, each employee in the EmployeeAddressTable has statistics (one row of data) in the EmployeeStatisticsTable. Even though this is a contrived example, this is a "1-1" relationship. Also notice the "has" in bold...when expressing a relationship, it is important to describe the relationship with a verb.

    The other two kinds of relationships may or may not use logical primary key and foreign key constraints...it is strictly a call of the designer. The first of these is the *one-to-many relationship* ("1-M"). This means that for every column value in one table, there is one or more related values in another table. Key constraints may be added to the design, or possibly just the use of some sort of identifier column may be used to establish the relationship. An example

would be that for every OwnerID in the AntiqueOwners table, there are one or more (zero is permissible too) Items bought in the Antiques table (verb: buy).

Finally, the *many-to-many relationship* ("M-M") does not involve keys generally, and usually involves idenifying columns. The unusual occurence of a "M-M" means that one column in one table is related to another column in another table, and for every value of one of these two columns, there are one or more related values in the corresponding column in the other table (and vice-versa), or more a common possibility, two tables have a 1-M relationship to each other (two relationships, one 1-M going each way). A [bad] example of the more common situation would be if you had a job assignment database, where one table held one row for each employee and a job assignment, and another table held one row for each job with one of the assigned employees. Here, you would have multiple rows for each employee in the first table, one for each job assignment, and multiple rows for each job in the second table, one for each employee assigned to the project. These tables have a M-M: each employee in the first table has many job assignments from the second table, and each job has many employees assigned to it from the first table. This is the tip of the iceberg on this topic...see the links below for more information and see the diagram below for a *simplified* example of an E-R diagram.



14. What are some important nonstandard SQL features (extremely common question)? —Well, see the next section...

Nonstandard SQL..."check local listings"

- INTERSECT and MINUS are like the UNION statement, except that INTERSECT produces rows that appear in both queries, and MINUS produces rows that result from the first query, but not the second.
- Report Generation Features: the COMPUTE clause is placed at the end of a query to place the result of an aggregate function at the end of a listing, like COMPUTE SUM (PRICE); Another option is to use break logic: define a break to divide the query results into groups based on a column, like BREAK ON BUYERID. Then, to produce a result after the listing of a group, use COMPUTE SUM OF PRICE ON BUYERID. If, for example, you used all three of these clauses (BREAK first, COMPUTE on break second, COMPUTE overall sum third), you would get a report that grouped items by their BuyerID, listing the sum of Prices after each group of a BuyerID's items, then, after all groups are listed, the sum of all Prices is listed, all with SQL-generated headers and lines.
- In addition to the above listed aggregate functions, some DBMS's allow more functions to be used in Select lists, except that these functions (some character functions allow multiple-row results) are to be used with an individual value (not groups), on single-row queries. The functions are to be used only on appropriate data types, also. Here are some **Mathematical Functions:**

ABS(X)	Absolute value-converts negative numbers to positive, or leaves positive numbers alone
CEIL(X)	X is a decimal value that will be rounded up.
FLOOR(X)	X is a decimal value that will be rounded down.
GREATEST(X,Y)	Returns the largest of the two values.
LEAST(X,Y)	Returns the smallest of the two values.
MOD(X,Y)	Returns the remainder of X / Y.
POWER(X,Y)	Returns X to the power of Y.
ROUND(X,Y)	Rounds X to Y decimal places. If Y is omitted, X is rounded to the nearest integer.
SIGN(X)	Returns a minus if X < 0, else a plus.
SQRT(X)	Returns the square root of X.

Character Functions	
LEFT(<string>,X)	Returns the leftmost X characters of the string.
RIGHT(<string>,X)	Returns the rightmost X characters of the string.
UPPER(<string>)	Converts the string to all uppercase letters.
LOWER(<string>)	Converts the string to all lowercase letters.
INITCAP(<string>)	Converts the string to initial caps.
LENGTH(<string>)	Returns the number of characters in the string.
<string>||<string>	Combines the two strings of text into one, *concatenated* string, where the first string is immediately followed by the second.
LPAD(<string>,X,'*')	Pads the string on the left with the * (or whatever character is inside the quotes), to make the string X characters long.
RPAD(<string>,X,'*')	Pads the string on the right with the * (or whatever character is inside the quotes), to make the string X characters long.
SUBSTR(<string>,X,Y)	Extracts Y letters from the string beginning at position X.
NVL(<column>,<value>)	The Null value function will substitute <value> for any NULLs for in the <column>. If the current value of <column> is not NULL, NVL has no effect.

Syntax Summary-For Advanced Users Only

Here are the general forms of the statements discussed in this tutorial, plus some extra important ones (explanations given). REMEMBER that all of these statements may or may not be available on your system, so check documentation regarding availability:

ALTER TABLE <TABLE NAME> ADD|DROP|MODIFY (COLUMN SPECIFICATION[S]...see Create Table); -allows you to add or delete a column or columns from a table, or change the specification (data type, etc.) on an existing column; this statement is also used to change the physical specifications of a table (how a table is stored, etc.), but these definitions are DBMS-specific, so read the documentation. Also, these physical specifications are used with the Create Table statement, when a table is first created. In addition, only one option can be performed per Alter Table statement-either add, drop, OR modify in a single statement**.**

COMMIT; -makes changes made to some database systems permanent (since the last COMMIT; known as a *transaction*)

CREATE [UNIQUE] INDEX <INDEX NAME>
ON <TABLE NAME> (<COLUMN LIST>); —UNIQUE is optional; within brackets.

CREATE TABLE <TABLE NAME>
(<COLUMN NAME> <DATA TYPE> [(<SIZE>)]
<COLUMN CONSTRAINT>,
...other columns); (also valid with ALTER TABLE)
-where SIZE is only used on certain data types (see above), and constraints include the following possibilities (automatically enforced by the DBMS; failure causes an error to be generated):

1. NULL or NOT NULL (see above)
2. UNIQUE enforces that no two rows will have the same value for this column
3. PRIMARY KEY tells the database that this column is the primary key column (only used if the key is a one column key, otherwise a PRIMARY KEY (column, column, ...) statement appears after the last column definition.
4. CHECK allows a condition to be checked for when data in that column is updated or inserted; for example, CHECK (PRICE > 0) causes the system to check that the Price column is greater than zero before accepting the value...sometimes implemented as the CONSTRAINT statement.
5. DEFAULT inserts the default value into the database if a row is inserted without that column's data being inserted; for example, BENEFITS INTEGER DEFAULT = 10000
6. FOREIGN KEY works the same as Primary Key, but is followed by: REFERENCES <TABLE NAME> (<COLUMN NAME>), which refers to the referential primary key.

CREATE VIEW <TABLE NAME> AS <QUERY>;

DELETE FROM <TABLE NAME> WHERE <CONDITION>;

INSERT INTO <TABLE NAME> [(<COLUMN LIST>)] VALUES (<VALUE LIST>);

ROLLBACK; —Takes back any changes to the database that you have made, back to the last time you gave a Commit command...beware! Some software uses automatic committing on systems that use the transaction features, so the Rollback command may not work.

SELECT [DISTINCT|ALL] <LIST OF COLUMNS, FUNCTIONS, CONSTANTS, ETC.>
FROM <LIST OF TABLES OR VIEWS>
[WHERE <CONDITION(S)>]
[GROUP BY <GROUPING COLUMN(S)>]
[HAVING <CONDITION>]
[ORDER BY <ORDERING COLUMN(S)> [ASC|DESC]];
—where ASC|DESC allows the ordering to be done in ASCending or DESCending order

UPDATE <TABLE NAME>
SET <COLUMN NAME> = <VALUE>
[WHERE <CONDITION>]; - if the Where clause is left out, all rows will be updated according to the Set statement

Review Question

1. List down various DML, DDL, DCL command

LESSON 37
TUTORIAL ON BACKUP AND RECOVERY

When data disappears, your boss wants it back, and your job is on the line.

One of the innumerable tasks of the DBA is to ensure that all of the databases of the enterprise are always "available." Availability in this context means that the users must be able to access the data stored in the databases, and that the contents of the databases must be up-to-date, consistent, and correct. It must never appear to a user that the system has lost the data or that the data has become inconsistent. This would totally ruin the user's confidence in the database and the entire system.

Many factors threaten the availability of your databases. These include natural disasters (such as floods and earthquakes), hardware failures (for example, a power failure or disk crash), software failures (such as DBMS malfunctions - read "bugs" - and application program errors), and people failures (for example, operator errors, user misunderstandings, and keyboard trouble). To this list you can also add the threats I listed last month under security, such as malicious attempts to destroy or corrupt the contents of the database.

In a large enterprise, the DBA must ensure the availability of several databases, such as the development databases, the databases used for unit and acceptance testing, the operational online production databases (some of which may be replicated or distributed all over the world), the data warehouse databases, the data marts, and all of the other departmental databases. All of these databases usually have different requirements for availability. The online production databases typically must be available, up-to-date, and consistent for 24 hours a day, seven days a week, with minimal downtime. The warehouse databases must be available and up-to-date during business hours and even for a while after hours.

On the other hand, the test databases need to be available only for testing cycles, but during these periods the testing staff may have extensive requirements for the availability of their test databases. For example, the DBA may have to restore the test databases to a consistent state after each test. The developers often have even more ad hoc requirements for the availability of the development databases, specifically toward the end of a crucial deadline. The business hours of a multinational organization may also have an impact on availability. For example, a working day from 8 a.m. in central Europe to 6 p.m. in California implies that the database must be available for 20 hours a day. The DBA is left with little time to provide for availability, let alone perform other maintenance tasks.

Recovery is the corrective process to restore the database to a usable state from an erroneous state. The basic recovery process consists of the following steps:

1. Identify that the database is in an erroneous, damaged, or crashed state.
2. Suspend normal processing.
3. Determine the source and extent of the damage.
4. Take corrective action, that is:
    - Restore the system resources to a usable state.
    - Rectify the damage done, or remove invalid data.
    - Restart or continue the interrupted processes, including the re-execution of interrupted transactions.
5. Resume normal processing.

To cope with failures, additional components and algorithms are usually added to the system. Most techniques use recovery data (that is, redundant data), which makes recovery possible. When taking corrective action, the effects of some transactions must be removed, while other transactions must be re-executed; some transactions must even be undone and redone. The recovery data must make it possible to perform these steps.

The following techniques can be used for recovery from an erroneous state:

Dump and restart: The entire database must be backed up regularly to archival storage. In the event of a failure, a copy of the database in a previous correct state (such as from a check-point) is loaded back into the database. The system is then restarted so that new transactions can proceed. Old transactions can be re-executed if they are available. The following types of restart can be identified:

- A warm restart is the process of starting the system after a controlled system shutdown, in which all active transactions were terminated normally and successfully.
- An emergency restart is invoked by a restart command issued by the operator. It may include reloading the database contents from archive storage.
- A cold start is when the system is started from scratch, usually when a warm restart is not possible. This may also include reloading the database contents from archive storage. Usually used to recover from physical damage, a cold restart is also used when recovery data was lost.

Undo-redo processing (also called roll-back and re-execute): By using an audit trail of transactions, all of the effects of recent, partially completed transactions can be undone up to a known correct state. Undoing is achieved by reversing the updating process. By working backwards through the log, all of the records of the transaction in question can be traced, until the begin transaction operations of all of the relevant transactions have been reached. The undo operation must be "idempotent," meaning that failures during undo operations must still result in the correct single intended undo operation taking place. From the known correct state, all of the journaled transactions can then be re-executed to obtain the desired correct resultant database contents. The operations of the transactions that were already executed at a previous stage are obtained from the audit trail. The redo operation must also be idempotent, meaning

that failures during redo operations must still result in the correct single intended redo operation taking place. This technique can be used when partially completed processes are aborted.

Roll-forward processing (also called reload and re-execute): All or part of a previous correct state (for example, from a checkpoint) is reloaded; the DBA can then instruct the DBMS to re-execute the recently recorded transactions from the transaction audit trail to obtain a correct state. It is typically used when (part of) the physical media has been damaged.

Restore and repeat: This is a variation of the previous method, where a previous correct state is restored. The difference is that the transactions are merely reposted from before and/or after images kept in the audit trail. The actual transactions are not re-executed: They are merely reapplied from the audit trail to the actual data table. In other words, the images of the updated rows (the effects of the transactions) are replaced in the data table from the audit trail, but the original transactions are not re-executed as in the previous case.

As a result, the DBA has an extensive set of requirements for the tools and facilities offered by the DBMS. These include facilities to back up an entire database offline, facilities to back up parts of the database selectively, features to take a snapshot of the database at a particular moment, and obviously journaling facilities to roll back or roll forward the transactions applied to the database to a particular identified time. Some of these facilities must be used online - that is, while the users are busy accessing the database. For each backup mechanism, there must be a corresponding restore mechanism - these mechanisms should be efficient, because you usually have to restore a lost, corrupt, or damaged database at some critical moment, while the users are waiting anxiously (sometimes highly irritated) and the managers are jumping up and down (often ineffectually)! The backup and restore facilities should be configurable - you may want to stream the backup data to and from multiple devices in parallel, you may want to add compression and decompression (including using third-party compression tools), you may want to delete old backups automatically off the disk, or you may want to label the tapes according to your own standards. You should also be able to take the backup of a database from one platform and restore it on another - this step is necessary to cater for non-database-related problems, such as machine and operating system failures. For each facility, you should be able to monitor its progress and receive an acknowledgment that each task has been completed successfully.

Some organizations use so-called “hot standby” techniques to increase the availability of their databases. In a typical hot standby scenario, the operations performed on the operational database are replicated to a standby database. If any problems are encountered on the operational database, the users are switched over and continue working on the standby database until the operational database is restored. However, database replication is an involved and extensive topic - I will cover it in detail in a subsequent column.

In the remainder of this month’s column I investigate the tools and facilities offered by IBM, Informix, Microsoft, Oracle, and Sybase for backup and recovery.

IBM DB2

IBM’s DB2 release 2.1.1 provides two facilities to back up your databases, namely the BACKUP command and the Database Director. It provides three methods to recover your database: crash recovery, restore, and roll-forward.

Backups can be performed either online or offline. Online backups are only supported if roll-forward recovery is enabled for the specific database. To execute the BACKUP command, you need SYSADM, SYSCTRL, or SYSMAINT authority. A database or a tablespace can be backed up to a fixed disk or tape. A tablespace backup and a tablespace restore cannot be run at the same time, even if they are working on different tablespaces. The backup command provides concurrency control for multiple processes making backup copies of different databases at the same time.

The restore and roll-forward methods provide different types of recovery. The restore-only recovery method makes use of an offline, full backup copy of the database; therefore, the restored database is only as current as the last backup. The roll-forward recovery method makes use of database changes retained in logs - therefore it entails performing a restore database (or tablespaces) using the BACKUP command, then applying the changes in the logs since the last backup. You can only do this when roll-forward recovery is enabled. With full database roll-forward recovery, you can specify a date and time in the processing history to which to recover.

Crash recovery protects the database from being left in an inconsistent state. When transactions against the database are unexpectedly interrupted, you must perform a rollback of the incomplete and in-doubt transactions, as well as the completed transactions that are still in memory. To do this, you use the RESTART DATABASE command. If you have specified the AUTORESTART parameter, a RESTART DATABASE is performed automatically after each failure. If a media error occurs during recovery, the recovery will continue, and the erroneous tablespace is taken offline and placed in a roll-forward pending state. The offline tablespace will need additional fixing up - restore and/or roll-forward recovery, depending on the mode of the database (whether it is recoverable or non-recoverable).

Restore recovery, also known as version control, lets you restore a previous version of a database made using the BACKUP command. Consider the following two scenarios:

- A database restore will rebuild the entire database using a backup made earlier, thus restoring the database to the identical state when the backup was made.
- A tablespace restore is made from a backup image, which was created using the BACKUP command where only one or more tablespaces were specified to be backed up. Therefore this process only restores the selected tablespaces to the state they were in when the backup was taken; it leaves the unselected tablespaces in a different state. A

tablespace restore can be done online (shared mode) or offline (exclusive mode).

Roll-forward recovery may be the next task after a restore, depending on your database's state. There are two scenarios to consider:

- Database roll-forward recovery is performed to restore the database by applying the database logs. The database logs record all of the changes made to the database. On completion of this recovery method, the database will return to its prefailure state. A backup image of the database and archives of the logs are needed to use this method.
- Tablespace roll-forward can be done in two ways: either by using the ROLLFORWARD command to apply the logs against the tablespaces in a roll-forward pending state, or by performing a tablespace restore and roll-forward recovery, followed by a ROLLFORWARD operation to apply the logs.

Informix

Informix for Windows NT release 7.12 has a Storage Manager Setup tool and a Backup and Restore tool. These tools let you perform complete or incremental backups of your data, back up logical log files (continuous and manual), restore data from a backup device, and specify the backup device.

Informix has a Backup and Restore wizard to help you with your backup and restore operations. This wizard is only available on the server machine. The Backup and Restore wizard provides three options: Backup, Logical Log Backup, and Restore.

The Backup and Restore tool provides two types of backups: complete and incremental. A complete backup backs up all of the data for the selected database server. A complete backup - also known as a level-0 backup - is required before you can do an incremental backup. An incremental backup - also known as a level-1 backup - backs up all changes that have occurred since the last complete backup, thereby requiring less time because only part of the data from the selected database server is backed up. You also get a level-2 backup, performed using the command-line utilities, that is used to back up all of the changes that have occurred since the last incremental backup. The Backup and Restore tool provides two types of logical log backups: continuous backup of the logical logs and manual backup of the logical logs. A Logical Log Backup backs up all full and used logical log files for a database server. The logical log files are used to store records of the online activity that occurs between complete backups.

The Informix Storage Manager (ISM) Setup tool lets you specify the storage device for storing the data used for complete, incremental, and logical log backups. The storage device can be a tape drive, a fixed hard drive, a removable hard drive, or none (for example, the null device). It is only available on the server machine. You can select one backup device for your general backups (complete or incremental) and a separate device for your logical log backups. You always have to move the backup file to another location or rename the file before starting your next backup. Before restoring your data, you must move the backup file to the directory specified in the ISM Setup and rename the backup file to the filename specified in ISM Setup.

If you specify None as your logical log storage device, the application marks the logical log files as backed up as soon as they become full, effectively discarding logical log information. Specify None only if you do not need to recover transactions from the logical log. When doing a backup, the server must be online or in administration mode. Once the backup has started, changing the mode will terminate the backup process. When backing up to your hard drive, the backup file will be created automatically.

The Restore option of the Backup and Restore wizard restores the data and logical log files from a backup source. You cannot restore the data if you have not made a complete backup. The server must be in offline mode during the restore operation. You can back up your active logical log files before doing the restore, and you can also specify which log files must be used. A level-1 (incremental) backup can be restored, but you will be prompted to proceed with a level-2 backup at the completion of the level-1 restore. Once the restore is completed, the database server can be brought back online, and processing can continue as usual. If you click on Cancel during a restore procedure, the resulting data may be corrupted.

Microsoft SQL Server

Microsoft SQL Server 6.5 provides more than one backup and recovery mechanism. For backups of the database, the user can either use the Bulk Copy Program (BCP) from the command line to create flat-file backups of individual tables or the built-in Transact-SQL DUMP and LOAD statements to back up or restore the entire database or specific tables within the database.

Although the necessary Transact-SQL statements are available from within the SQL environment, the Microsoft SQL Enterprise Manager provides a much more user-friendly interface for making backups and recovering them later on. The Enterprise Manager will prompt the DBA for information such as database name, backup device to use, whether to initialize the device, and whether the backup must be scheduled for later or done immediately. Alternatively, you can use the Database Maintenance wizard to automate the whole maintenance process, including the backup procedures. These tasks are automatically scheduled by the wizard on a daily or weekly basis. Both the BCP utility and the dump statement can be run online, which means that users do not have to be interrupted while backups are being made. This facility is particularly valuable in 24 X 7 operations.

A database can be restored up to the last committed transaction by also LOADing the transaction logs that were dumped since the previous database DUMP. Some of the LOAD options involve more management. For example, the database dump file and all subsequent transaction-log dump files must be kept until the last minute in case recovery is required. It is up to the particular site to determine a suitable backup and recovery policy, given the available options.

To protect against hardware failures, Microsoft SQL Server 6.5 has the built-in capability to define a standby server for automatic failover. This option requires sophisticated hardware

but is good to consider for 24 X 7 operations. Once configured, it does not require any additional tasks on an ongoing basis. In addition, separate backups of the database are still required in case of data loss or multiple media failure.

Oracle

Oracle7 Release 7.3 uses full and partial database backups and a redo log for its database backup and recovery operations. The database backup is an operating system backup of the physical files that constitute the Oracle database. The redo log consists of two or more preallocated files, which are used to record all changes made to the database. You can also use the export and import utilities to create a backup of a database. Oracle offers a standby database scheme, with which it maintains a copy of a primary database on duplicate hardware, in a constant recoverable state, by applying the redo logs archived off the primary database.

A full backup is an operating system backup of all of the data files, parameter files, and the control file that constitute the database. A full database backup can be taken by using the operating system's commands or by using the host command of the Server Manager. A full database backup can be taken online when the database is open, but only an offline database backup (taken when the database server is shut down) will necessarily be consistent. An inconsistent database backup must be recovered with the online and archived redo log files before the database will become available. The best approach is to take a full database backup after the database has been shut down with normal or immediate priority.

A partial backup is any operating system backup of a part of the full backup, such as selected data files, the control file only, or the data files in a specified tablespace only. A partial backup is useful if the database is operated in ARCHIVELOG mode. A database operating in NOARCHIVE mode rarely has sufficient information to use a partial backup to restore the database to a consistent state. The archiving mode is usually set during database creation, but it can be reset at a later stage.

You can recover a database damaged by a media failure in one of three ways after you have restored backups of the damaged data files. These steps can be performed using the Server Manager's Apply Recovery Archives dialog box, using the Server Manager's RECOVER command, or using the SQL ALTER DATABASE command:

- You can recover an entire database using the RECOVER DATABASE command. This command performs media recovery on all of the data files that require redo processing.
- You can recover specified tablespaces using the RECOVER TABLESPACE command. This command performs media recovery on all of the data files in the listed tablespaces. Oracle requires the database to be open and mounted in order to determine the file names of the tables contained in the tablespace.
- You can list the individual files to be recovered using the RECOVER DATAFILE command. The database can be open or closed, provided that Oracle can take the required media recovery locks.

In certain situations, you can also recover a specific damaged data file, even if a backup file isn't available. This can only be done if all of the required log files are available and the control file contains the name of the damaged file. In addition, Oracle provides a variety of recovery options for different crash scenarios, including incomplete recovery, change-based, cancel-based, and time-based recovery, and recovery from user errors.

Sybase SQL Server

Sybase SQL Server 11 uses database dumps, transaction dumps, checkpoints, and a transaction log per database for database recovery. All backup and restore operations are performed by an Open Server program called Backup Server, which runs on the same physical machine as the Sybase SQL Server 11 process.

A database dump is a complete copy of the database, including the data files and the transaction log. This function is performed using the DUMP DATABASE operation, which can place the backup on tape or on disk. You can make dynamic dumps, which let the users continue using the database while the dump is being made. A transaction dump is a routine backup of the transaction log. The DUMP TRANSACTION operation also truncates the inactive portion of the transaction log file. You can use multiple devices in the DUMP DATABASE and DUMP TRANSACTION operations to stripe the dumps across multiple devices.

The transaction log is a write-ahead log, maintained in the system table called syslogs. You can use the DUMP TRANSACTION command to copy the information from the transaction log to a tape or disk. You can use the automatic checkpointing task or the CHECKPOINT command (issued manually) to synchronize a database with its transaction log. Doing so causes the database pages that are modified in memory to be flushed to the disk. Regular checkpoints can shorten the recovery time after a system crash.

Each time Sybase SQL Server restarts, it automatically checks each database for transactions requiring recovery by comparing the transaction log with the actual data pages on the disk. If the log records are more recent than the data page, it reapplies the changes from the transaction log.

An entire database can be restored from a database dump using the LOAD DATABASE command. Once you have restored the database to a usable state, you can use the LOAD TRANSACTION command to load all transaction log dumps, in the order in which they were created. This process reconstructs the database by re-executing the transactions recorded in the transaction log.

You can use the DUMP DATABASE and LOAD DATABASE operations to port a database from one Sybase installation to another, as long as they run on similar hardware and software platforms.

Prevention is Better than Cure. . .

Although each DBMS I reviewed has a range of backup and recovery facilities, it is always important to ensure that the facilities are used properly and adequately. By "adequately," I mean that backups must be taken regularly. All of the DBMSs I reviewed provided the facilities to repost or re-execute completed transactions from a log or journal file. However,

reposting or re-executing a few weeks' worth of transactions may take an unbearably long time. In many situations, users require quick access to their databases, even in the presence of media failures. Remember that the end users are not concerned with physical technicalities, such as restoring a database after a system crash.

Even better than quick recovery is no recovery, which can be achieved in two ways. First, by performing adequate system monitoring and using proper procedures and good equipment, most system crashes can be avoided. It is better to provide users with a system that is up and available 90 percent of the time than to have to do sporadic fixes when problems occur. Second, by using redundant databases such as hot standby or replicated databases, users can be relieved of the recovery delays: Users can be switched to the hot backup database while the master database is being recovered.

A last but extremely important aspect of backup and recovery is testing. Test your backup and recovery procedures in a test environment before deploying them in the production environment. In addition, the backup and recovery procedures and facilities used in the production environment must also be tested regularly. A recovery scheme that worked perfectly well in a test environment is useless if it cannot be repeated in the production environment -particularly in that crucial moment when the root disk fails during the month-end run!

Database Backup and Recovery from Oracle Point of view

Database Backups

No matter what backup and recovery scheme you devise for an Oracle database, backups of the database's datafiles and control files are absolutely necessary as part of the strategy to safeguard against potential media failures that can damage these files.

The following sections provide a conceptual overview of the different types of backups that can be made and their usefulness in different recovery schemes. The Oracle8 Server Backup and Recovery Guide provides more details, along with guidelines for performing database backups.

Whole Database Backups

A *whole database backup* is an operating system backup of all datafiles and the control file that constitute an Oracle database. A whole backup should also include the parameter file(s) associated with the database. You can take a whole database backup when the database is shut down or while the database is open. You should not normally take a whole backup after an instance failure or other unusual circumstances.

Consistent Whole Backups vs. Inconsistent Whole Backups

Following a clean shutdown, all of the files that constitute a database are closed and consistent with respect to the current point in time. Thus, a whole backup taken after a shutdown can be used to recover to the point in time of the last whole backup. A whole backup taken while the database is open is not consistent to a given point in time and must be recovered (with the online and archived redo log files) before the database can become available.

Backups and Archiving Mode

The datafiles obtained from a whole backup are useful in any type of media recovery scheme:

- If a database is operating in NOARCHIVELOG mode and a disk failure damages some or all of the files that constitute the database, the most recent consistent whole backup can be used to *restore* (not recover) the database.

Because an archived redo log is not available to bring the database up to the current point in time, all database work performed since the backup must be repeated. Under special circumstances, a disk failure in NOARCHIVELOG mode can be fully recovered, but you should not rely on this.

- If a database is operating in ARCHIVELOG mode and a disk failure damages some or all of the files that constitute the database, the datafiles collected by the most recent whole backup can be used as part of database *recovery*.

After restoring the necessary datafiles from the whole backup, database recovery can continue by applying archived and current online redo log files to bring the restored datafiles up to the current point in time.

In summary, if a database is operated in NOARCHIVELOG mode, a consistent whole database backup is the only method to partially protect the database against a disk failure; if a database is operating in ARCHIVELOG mode, either a consistent or an inconsistent whole database backup can be used to restore damaged files as part of database recovery from a disk failure.

Partial Database Backups

A *partial database backup* is any backup short of a whole backup, taken while the database is open or shut down. The following are all examples of partial database backups:

- a backup of all datafiles for an individual tablespace
- a backup of a single datafile
- a backup of a control file

Partial backups are only useful for a database operating in ARCHIVELOG mode. Because an archived redo log is present, the datafiles restored from a partial backup can be made consistent with the rest of the database during recovery procedures.

Datafile Backups

A partial backup includes only some of the datafiles of a database. Individual or collections of specific datafiles can be backed up independently of the other datafiles, online redo log files, and control files of a database. You can back up a datafile while it is offline or online.

Choosing whether to take online or offline datafile backups depends only on the availability requirements of the data - online datafile backups are the only choice if the data being backed up must always be available.

Control File Backups

Another form of a partial backup is a control file backup. Because a control file keeps track of the associated database's physical file structure, a backup of a database's control file

should be made every time a structural change is made to the database.

Note: The Recovery Manager automatically backs up the control file in any backup that includes datafile 1, which contains the data dictionary.

Multiplexed control files safeguard against the loss of a single control file. However, if a disk failure damages the datafiles and incomplete recovery is desired, or a point-in-time recovery is desired, a backup of the control file that corresponds to the intended database structure should be used, not necessarily the current control file. Therefore, the use of multiplexed control files is not a substitute for control file backups taken every time the structure of a database is altered.

If you use Recovery Manager to restore the control file prior to incomplete or point-in-time recovery, Recovery Manager automatically restores the most suitable backup control file.

This section covers the structures and software mechanisms used by Oracle to provide:

- database recovery required by different types of failures
- flexible recovery operations to suit any situation
- availability of data during backup and recovery operations so that users of the system can continue to work

Why Is Recovery Important?

In every database system, the possibility of a system or hardware failure always exists. Should a failure occur and affect the database, the database must be recovered. The goals after a failure are to ensure that the effects of all committed transactions are reflected in the recovered database and to return to normal operation as quickly as possible while insulating users from problems caused by the failure.

Types of Failures

Several circumstances can halt the operation of an Oracle database. The most common types of failure are described below:

User error	User errors can require a database to be recovered to a point in time before the error occurred. For example, a user might accidentally drop a table. To allow recovery from user errors and accommodate other unique recovery requirements, Oracle provides for exact point-in-time recovery. For example, if a user accidentally drops a table, the database can be recovered to the instant in time before the table was dropped.
Statement and process failure	Statement failure occurs when there is a logical failure in the handling of a statement in an Oracle program (for example, the statement is not a valid SQL construction). When statement failure occurs, the effects (if any) of the statement are automatically undone by Oracle and control is returned to the user. A *process failure* is a failure in a user process accessing Oracle, such as an abnormal disconnection or process termination. The failed user process cannot continue work, although Oracle and other user processes can. The Oracle background process PMON automatically detects the failed user process or is informed of it by SQL*Net. PMON resolves the problem by rolling back the uncommitted transaction of the user process and releasing any resources that the process was using. Common problems such as erroneous SQL statement constructions and aborted user processes should never halt the database system as a whole. Furthermore, Oracle automatically performs necessary recovery from uncommitted transaction changes and locked resources with minimal impact on the system or other users.
Instance failure	Instance failure occurs when a problem arises that prevents an instance (system global area and background processes) from continuing work. Instance failure may result from a hardware problem such as a power outage, or a software problem such as an operating system crash. When an instance failure occurs, the data in the buffers of the system global area is not written to the datafiles. Instance failure requires *instance recovery*. Instance recovery is automatically performed by Oracle when the instance is restarted. The redo log is used to recover the committed data in the SGA's database buffers that was lost due to the instance failure.

Media (disk) failure	An error can arise when trying to write or read a file that is required to operate the database. This is called disk failure because there is a physical problem reading or writing physical files on disk. A common example is a disk head crash, which causes the loss of all files on a disk drive. Different files may be affected by this type of disk failure, including the datafiles, the redo log files, and the control files. Also, because the database instance cannot continue to function properly, the data in the database buffers of the system global area cannot be permanently written to the datafiles. A disk failure requires *media recovery*. Media recovery restores a database's datafiles so that the information in them corresponds to the most recent time point before the disk failure, including the committed data in memory that was lost because of the failure. To complete a recovery from a disk failure, the following is required: backups of the database's datafiles, and all online and necessary

Oracle provides for complete and quick recovery from all possible types of hardware failures including disk crashes. Options are provided so that a database can be completely recovered or partially recovered to a specific point in time.

If some datafiles are damaged in a disk failure but most of the database is intact and operational, the database can remain open while the required tablespaces are individually recovered. Therefore, undamaged portions of a database are available for normal use while damaged portions are being recovered.

Structures Used for Recovery

Oracle uses several structures to provide complete recovery from an instance or disk failure: the redo log, rollback segments, a control file, and necessary database backups.

The Redo Log

As described in "Redo Log Files" on page 1-11, the *redo log* is a set of files that protect altered database data in memory that has not been written to the datafiles. The redo log can consist of two parts: the online redo log and the archived redo log.

The Online Redo Log

The *online redo log* is a set of two or more *online redo log files* that record all committed changes made to the database. Whenever a transaction is committed, the corresponding redo entries temporarily stored in redo log buffers of the system global area are written to an online redo log file by the background process LGWR.

The online redo log files are used in a cyclical fashion; for example, if two files constitute the online redo log, the first file is filled, the second file is filled, the first file is reused and filled, the second file is reused and filled, and so on. Each time a file is filled, it is assigned a *log sequence number* to identify the set of redo entries.

To avoid losing the database due to a single point of failure, Oracle can maintain multiple sets of online redo log files. A *multiplexed online redo log* consists of copies of online redo log files physically located on separate disks; changes made to one member of the group are made to all members.

If a disk that contains an online redo log file fails, other copies are still intact and available to Oracle. System operation is not interrupted and the lost online redo log files can be easily recovered using an intact copy.

The Archived Redo Log

Optionally, filled online redo files can be archived before being reused, creating an *archived redo log*. *Archived (offline) redo log files* constitute the archived redo log.

The presence or absence of an archived redo log is determined by the mode that the redo log is using:

ARCHIVELOG	The filled online redo log files are archived before they are reused in the cycle.
NOARCHIVELOG	The filled online redo log files are not archived.

In ARCHIVELOG mode, the database can be completely recovered from both instance and disk failure. The database can also be backed up while it is open and available for use. However, additional administrative operations are required to maintain the archived redo log.

If the database's redo log is operated in NOARCHIVELOG mode, the database can be completely recovered from instance failure, but not from a disk failure. Additionally, the database can be backed up only while it is completely closed. Because no archived redo log is created, no extra work is required by the database administrator.

Control Files

The *control files* of a database keep, among other things, information about the file structure of the database and the current log sequence number being written by LGWR. During normal recovery procedures, the information in a control file is used to guide the automated progression of the recovery operation.

Multiplexed Control Files

This feature is similar to the multiplexed redo log feature: a number of identical control files may be maintained by Oracle, which updates all of them simultaneously.

Rollback Segments

As described in "Data Blocks, Extents, and Segments" on page 1-9, rollback segments record rollback information used by several functions of Oracle. During database recovery, after all

changes recorded in the redo log have been applied, Oracle uses rollback segment information to undo any uncommitted transactions. Because rollback segments are stored in the database buffers, this important recovery information is automatically protected by the redo log.

Database Backups

Because one or more files can be physically damaged as the result of a disk failure, media recovery requires the restoration of the damaged files from the most recent operating system backup of a database. There are several ways to back up the files of a database.

Whole Database Backups

A whole database backup is an operating system backup of all datafiles, online redo log files, and the control file that constitutes an Oracle database. Full backups are performed when the database is closed and unavailable for use.

Partial Backups

A partial backup is an operating system backup of part of a database. The backup of an individual tablespace's datafiles or the backup of a control file are examples of partial backups. Partial backups are useful only when the database's redo log is operated in ARCHIVELOG mode.

A variety of partial backups can be taken to accommodate any backup strategy. For example, you can back up datafiles and control files when the database is open or closed, or when a specific tablespace is online or offline. Because the redo log is operated in ARCHIVELOG mode, additional backups of the redo log are not necessary; the archived redo log is a backup of filled online redo log files.

Basic Recovery Steps

Due to the way in which DBWR writes database buffers to datafiles, at any given point in time, a datafile may contain some data blocks tentatively modified by uncommitted transactions and may not contain some blocks modified by committed transactions. Therefore, two potential situations can result after a failure:

- Blocks containing committed modifications were not written to the datafiles, so the changes may only appear in the redo log. Therefore, the redo log contains committed data that must be applied to the datafiles.
- Since the redo log may have contained data that was not committed, uncommitted transaction changes applied by the redo log during recovery must be erased from the datafiles.

To solve this situation, two separate steps are always used by Oracle during recovery from an instance or media failure: rolling forward and rolling back.

Rolling Forward

The first step of recovery is to *roll forward*, that is, reapply to the datafiles all of the changes recorded in the redo log. Rolling forward proceeds through as many redo log files as necessary to bring the datafiles forward to the required time.

If all needed redo information is online, Oracle performs this recovery step automatically when the database starts. After roll forward, the datafiles contain all committed changes as well as any uncommitted changes that were recorded in the redo log.

Rolling Back

The roll forward is only half of recovery. After the roll forward, any changes that were not committed must be undone. After the redo log files have been applied, then the rollback segments are used to identify and undo transactions that were never committed, yet were recorded in the redo log. This process is called *rolling back*. Oracle completes this step automatically.

The Recovery Manager

The Recovery Manager is an Oracle utility that manages backup and recovery operations, creating backups of database files and restoring or recovering a database from backups.

Recovery Manager maintains a repository called the *recovery catalog*, which contains information about backup files and archived log files. Recovery Manager uses the recovery catalog to automate both restore operations and media recovery.

The recovery catalog contains:

- information about backups of datafiles and archivelogs
- information about datafile copies
- information about archived redo logs and copies of them
- information about the physical schema of the target database
- named sequences of commands called *stored scripts*.

Review Question

1. What are backups and why it is important?

Selected Bibliography

For more information about the Recovery Manager, see the Oracle8 Server Backup and Recovery Guide.

* IBM Corp., Armonk, NY; 800-425-3333 or 914-765-1900; www.ibm.com.
* Informix Software Inc., Menlo Park, CA; 800-331-1763, 415-926-6300, or fax 415-926-6593; www.informix.com.
* Microsoft Corp., Redmond, WA; 800-426-9400, 206-882-8080, or fax 206-936-7329; www.microsoft.com.
* Oracle Corp., Redwood Shores, CA; 800-672-2537, 415-506-7000, or fax 415-506-7200; www.oracle.com.
* Sybase Inc., Emeryville, CA; 800-879-2273, 510-922-3500, or fax 510-922-9441; www.sybase.com.

LESSON 38
DATABASE SECURITY ISSUES

Database security entails allowing or disallowing user actions on the database and the objects within it. Oracle uses schemas and security domains to control access to data and to restrict the use of various database resources.

The centralized and multi-user nature of a DBMS requires that some form of *security control* is in place, both to prevent unauthorized access and to limit access for authorized users. Security control can generally be divided into two areas, *user authorization* and *transaction authorization*.

User Authorization

User authorization helps to protect a database against unauthorized use, usually by requiring that a user enter a user name and a password to gain entry to the system. The password is usually known only to the user and the DBMS, and is protected by the DBMS at least as well as the data in the database. However, it should be noted this user name and password scheme can not *guarantee* the security of the database. It does not prevent you from choosing a password that is easy to guess (like the name of a spouse or pet) or from recording your password in an accessible location (like on the front of your computer!).

Transaction Authorization

Generally, not all users are given the same access rights to different databases or different parts of the same database. In some cases, sensitive data such as employee salaries should only be accessible to those users who need it. In other cases, some users may only require the ability to read some data items, where other users require the ability to both read and update the data.

A Point-of-Sale (POS) system is a good example of the second case: clerks working in a store might need read access for the price of an item, but should not be able to change the price. Employees at the head office may need to read and update the data, in order to enter new prices for the item.

Transaction authorization helps to protect a database against an authorized user trying to access a data item they do not have permission to access (this may occur either intentionally or unintentionally). The DBMS usually keeps a record of what rights have been granted to users on all of the data objects in the database, and checks these rights every time a user transaction tries to access the database. If the user does not have the proper rights to a data item, the transaction will not be allowed. It is the responsibility of the Database Administrator to explicitly grant the rights assigned to each user.

Database Security

Multi-user database systems, such as Oracle, include security features that control how a database is accessed and used. For example, security mechanisms do the following:

- Prevent unauthorized database access
- Prevent unauthorized access to schema objects
- Control disk usage
- Control system resource usage (such as CPU time)
- Audit user actions

Associated with each database user is a *schema* by the same name. A schema is a logical collection of objects (tables, views, sequences, synonyms, indexes, clusters, procedures, functions, packages, and database links). By default, each database user creates and has access to all objects in the corresponding schema.

Database security can be classified into two distinct categories: system security and data security.

System security includes the mechanisms that control the access and use of the database at the system level. For example, system security includes:

- Valid username/password combinations
- The amount of disk space available to the objects of a user
- The resource limits for a user

System security mechanisms check:

- Whether a user is authorized to connect to the database
- Whether database auditing is active
- Which system operations a user can perform

Data security includes the mechanisms that control the access and use of the database at the object level. For example, data security includes

- Which users have access to a specific schema object and the specific types of actions allowed for each user on the object (for example, user SCOTT can issue SELECT and INSERT statements but not DELETE statements using the EMP table)
- The actions, if any, that are audited for each schema object

Security Mechanisms

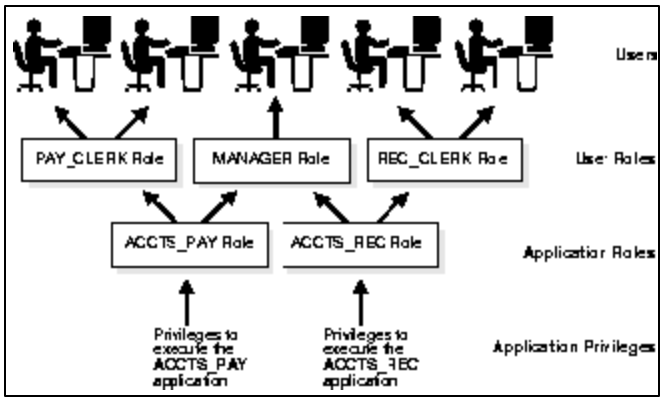
The Oracle Server provides *discretionary access control*, which is a means of restricting access to information based on privileges. The appropriate privilege must be assigned to a user in order for that user to access an object. Appropriately privileged users can grant other users privileges at their discretion; for this reason, this type of security is called “discretionary”.

Oracle manages database security using several different facilities:

- Database users and schemas
- Privileges
- Roles
- Storage settings and quotas
- Resource limits
- Auditing

Figure 1-4 illustrates the relationships of the different Oracle security facilities, and the following sections provide an overview of users, privileges, and roles.

Figure 1-4: Oracle Security Features



Database Users and Schemas

Each Oracle database has a list of usernames. To access a database, a user must use a database application and attempt a connection with a valid username of the database. Each username has an associated password to prevent unauthorized use.

Security Domain

Each user has a *security domain* - a set of properties that determine such things as the:

- Actions (privileges and roles) available to the user
- Tablespace quotas (available disk space) for the user
- System resource limits (for example, CPU processing time) for the user

Each property that contributes to a user's security domain is discussed in the following sections.

Privileges

A *privilege* is a right to execute a particular type of SQL statement. Some examples of privileges include the

- Right to connect to the database (create a session)
- Right to create a table in your schema
- Right to select rows from someone else's table
- Right to execute someone else's stored procedure

The privileges of an Oracle database can be divided into two distinct categories: system privileges and object privileges.

System Privileges

System privileges allow users to perform a particular systemwide action or a particular action on a particular type of object. For example, the privileges to create a tablespace or to delete the rows of any table in the database are system privileges. Many system privileges are available only to administrators and application developers because the privileges are very powerful.

Object Privileges

Object privileges allow users to perform a particular action on a specific schema object. For example, the privilege to delete rows of a specific table is an object privilege. Object privileges are granted (assigned) to end-users so that they can use a database application to accomplish specific tasks.

Granting Privileges

Privileges are granted to users so that users can access and modify data in the database. A user can receive a privilege two different ways:

- Privileges can be granted to users explicitly. For example, the privilege to insert records into the EMP table can be explicitly granted to the user SCOTT.
- Privileges can be granted to *roles* (a named group of privileges), and then the role can be granted to one or more users. For example, the privilege to insert records into the EMP table can be granted to the role named CLERK, which in turn can be granted to the users SCOTT and BRIAN.

Because roles allow for easier and better management of privileges, privileges are normally granted to roles and not to specific users. The following section explains more about roles and their use.

Roles

Oracle provides for easy and controlled privilege management through roles. *Roles* are named groups of related privileges that are granted to users or other roles. The following properties of roles allow for easier privilege management:

- *Reduced Granting of Privileges* - Rather than explicitly granting the same set of privileges to many users, a database administrator can grant the privileges for a group of related users granted to a role. And then the database administrator can grant the role to each member of the group.
- *Dynamic Privilege Management* - When the privileges of a group must change, only the privileges of the role need to be modified. The security domains of all users granted the group's role automatically reflect the changes made to the role.
- *Selective availability of privileges* - The roles granted to a user can be selectively enabled (available for use) or disabled (not available for use). This allows specific control of a user's privileges in any given situation.
- *Application awareness* - A database application can be designed to enable and disable selective roles automatically when a user attempts to use the application.

Database administrators often create roles for a database application. The DBA grants an application role all privileges necessary to run the application. The DBA then grants the application role to other roles or users. An application can have several different roles, each granted a different set of privileges that allow for more or less data access while using the application.

The DBA can create a role with a password to prevent unauthorized use of the privileges granted to the role. Typically, an application is designed so that when it starts, it enables the proper role. As a result, an application user does not need to know the password for an application's role.

Storage Settings and Quotas

Oracle provides means for directing and limiting the use of disk space allocated to the database on a per user basis, including default and temporary tablespaces and tablespace quotas.

Default Tablespace

Each user is associated with a *default tablespace*. When a user creates a table, index, or cluster and no tablespace is specified to physically contain the object, the user's default tablespace is used if the user has the privilege to create the object and a quota in the specified default tablespace. The default tablespace feature provides Oracle with information to direct space usage in situations where object location is not specified.

Temporary Tablespace

Each user has a *temporary tablespace*. When a user executes a SQL statement that requires the creation of temporary segments (such as the creation of an index), the user's temporary tablespace is used. By directing all users' temporary segments to a separate tablespace, the temporary tablespace feature can reduce I/O contention among temporary segments and other types of segments.

Tablespace Quotas

Oracle can limit the collective amount of disk space available to the objects in a schema. *Quotas* (space limits) can be set for each tablespace available to a user. The tablespace quota security feature permits selective control over the amount of disk space that can be consumed by the objects of specific schemas.

Profiles and Resource Limits

Each user is assigned a *profile* that specifies limitations on several system resources available to the user, including the

- Number of concurrent sessions the user can establish
- CPU processing time
  - Available to the user's session
  - Available to a single call to Oracle made by a SQL statement
- Amount of logical I/O
  - Available to the user's session
  - Available to a single call to Oracle made by a SQL statement
- Amount of idle time for the user's session allowed
- Amount of connect time for the user's session allowed
- Password restrictions
  - Account locking after multiple unsuccessful login attemts
  - Password expiration and grace period
  - Password reuse and complexity restrictions

Different profiles can be created and assigned individually to each user of the database. A default profile is present for all users not explicitly assigned a profile. The resource limit feature prevents excessive consumption of global database system resources.

Review Question

1. Write a short note on Data Security

Selected Bibliography

- [ARIES] C. Mohan, et al.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging., TODS 17(1): 94-162 (1992).
- [CACHE] C. Mohan: Caching Technologies for Web Applications, A Tutorial at the Conference on Very Large Databases (VLDB), Rome, Italy, 2001.
- [CODASYL] ACM: CODASYL Data Base Task Group April 71 Report, New York, 1971.
- [CODD] E. Codd: A Relational Model of Data for Large Shared Data Banks. ACM 13(6):377-387 (1970).
- [EBXML] http://www.ebxml.org.
- [FED] J. Melton, J. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, K. Zeidenstein: SQL and Management of External Data', SIGMOD Record 30(1):70-77, 2001.
- [GRAY] Gray, et al.: Granularity of Locks and Degrees of Consistency in a Shared Database., IFIP Working Conference on Modelling of Database Management Systems, 1-29, AFIPS Press.
- [INFO] P. Lyman, H. Varian, A. Dunn, A. Strygin, K. Swearingen: How Much Information? at http://www.sims.berkeley.edu/research/projects/how-much-info/.
- [LIND] B. Lindsay, et. al: Notes on Distributed Database Systems. IBM Research Report RJ2571, (1979).

Notes:

LESSON 39
DATABASE SECURITY(LEVEL IF SECURITY)

Controlling Database Access

This lecture explains how to control access to database. It includes:

- Database Security
- Schemas, Database Users, and Security Domains
- User Authentication
- User Table space Settings and Quotas
- The User Group PUBLIC
- User Resource Limits and Profiles
- Licensing

Database Security

Database security entails allowing or disallowing user actions on the database and the objects within it. Oracle uses schemas and security domains to control access to data and to restrict the use of various database resources.

Oracle provides comprehensive discretionary access control. Discretionary access control regulates all user access to named objects through privileges. A privilege is permission to access a named object in a prescribed manner; for example, permission to query a table. Privileges are granted to users at the discretion of other users-hence the term "discretionary access control".

Schemas, Database Users, and Security Domains

A user (sometimes called a username) is a name defined in the database that can connect to and access objects. A schema is a named collection of objects, such as tables, views, clusters, procedures, and packages, associated with a particular user. Schemas and users help database administrators manage database security.

To access a database, a user must run a database application (such as an Oracle Forms form, SQL*Plus, or a precompiler program) and connect using a username defined in the database.

When a database user is created, a corresponding schema of the same name is created for the user. By default, once a user connects to a database, the user has access to all objects contained in the corresponding schema. A user is associated only with the schema of the same name; therefore, the terms user and schema are often used interchangeably.

The access rights of a user are controlled by the different settings of the user's security domain. When creating a new database user or altering an existing one, the security administrator must make several decisions concerning a user's security domain. These include

- Whether user authentication information is maintained by the database, the operating system, or a network authentication service
- Settings for the user's default and temporary tablespaces
- A list, if any, of tablespaces accessible to the user and the associated quotas for each listed tablespace
- The user's resource limit profile; that is, limits on the amount of system resources available to the user
- The privileges and roles that provide the user with appropriate access to objects needed to perform database operations

User Authentication

To prevent unauthorized use of a database username, Oracle provides user validation via three different methods for normal database users:

- Authentication by the operating system
- Authentication by a network service
- Authentication by the associated Oracle database

For simplicity, one method is usually used to authenticate all users of a database. However, Oracle allows use of all methods within the same database instance.

Oracle also encrypts passwords during transmission to ensure the security of network authentication.

Oracle requires special authentication procedures for database administrators, because they perform special database operations.

Authentication by the Operating System

Some operating systems permit Oracle to use information maintained by the operating system to authenticate users. The benefits of operating system authentication are:

- Users can connect to Oracle more conveniently (without specifying a username or password). For example, a user can invoke SQL*Plus and skip the username and password prompts by entering
- SQLPLUS /
- Control over user authorization is centralized in the operating system; Oracle need not store or manage user passwords. However, Oracle still maintains usernames in the database.
- Username entries in the database and operating system audit trails correspond.

If the operating system is used to authenticate database users, some special considerations arise with respect to distributed database environments and database links.

Additional Information: For more information about authenticating via your operating system, see your Oracle operating system-specific documentation.

Authentication by the Network

If network authentication services are available to you (such as DCE, Kerberos, or SESAME), Oracle can accept authentication from the network service. To use a network authentication service with Oracle, you must also have the Oracle Secure Network Services product.

Authentication by the Oracle Database

Oracle can authenticate users attempting to connect to a database by using information stored in that database. You must use this method when the operating system cannot be used for database user validation.

When Oracle uses database authentication, you create each user with an associated password. A user provides the correct password when establishing a connection to prevent unauthorized use of the database. Oracle stores a user's password in the data dictionary in an encrypted format. A user can change his or her password at any time.

Password Encryption while Connecting

To protect password confidentiality, Oracle allows you to encrypt passwords during network (client/server and server/server) connections. If you enable this functionality on the client and server machines, Oracle encrypts passwords using a modified DES (Data Encryption Standards) algorithm before sending them across the network.

Account Locking

Oracle can lock a user's account if the user fails to login to the system within a specified number of attempts. Depending on how the account is configured, it can be unlocked automatically after a specified time interval or it must be unlocked by the database administrator.

The CREATE PROFILE statement configures the number of failed logins a user can attempt and the amount of time the account remains locked before automatic unlock.

The database administrator can also lock accounts manually. When this occurs, the account cannot be unlocked automatically but must be unlocked explicitly by the database administrator.

Password Lifetime and Expiration

Password lifetime and expiration options allow the database administrator to specify a lifetime for passwords, after which time they expire and must be changed before a login to the account can be completed. On first attempt to login to the database account after the password expires, the user's account enters the grace period, and a warning message is issued to the user every time the user tries to login until the grace period is over.

The user is expected to change the password within the grace period. If the password is not changed within the grace period, the account is locked and no further logins to that account are allowed without assistance by the database administrator.

The database administrator can also set the password state to expired. When this happens, the users account status is changed to expired, and when the user logs in, the account enters the grace period.

Password History

The password history option checks each newly specified password to ensure that a password is not reused for the specified amount of time or for the specified number of password changes. The database administrator can configure the rules for password reuse with CREATE PROFILE statements.

Password Complexity Verification

Complexity verification checks that each password is complex enough to provide reasonable protection against intruders who try to break into the system by guessing passwords.

The Oracle default password complexity verification routine requires that each password:

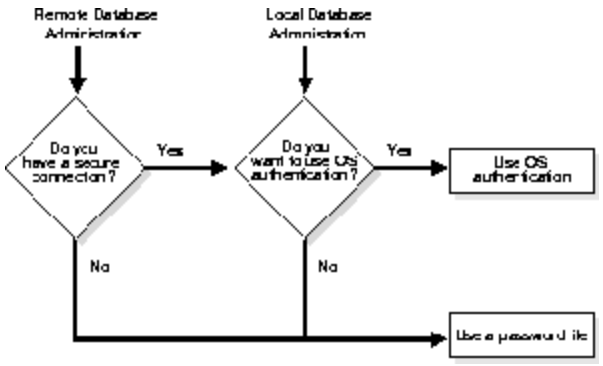
- Be a minimum of four characters in length
- Not equal the userid
- Include at least one alpha, one numeric, and one punctuation mark
- Not match any word on an internal list of simple words like welcome, account, database, user, and so on.
- Differ from the previous password by at least three characters.

Database Administrator Authentication

Database administrators perform special operations (such as shutting down or starting up a database) that should not be performed by normal database users. Oracle provides a more secure authentication scheme for database administrator usernames.

You can choose between operating system authentication or password files to authenticate database administrators; Figure A-1 illustrates the choices you have for database administrator authentication schemes, depending on whether you administer your database locally (on the same machine on which the database resides) or if you administer many different database machines from a single remote client.

Figure A-1: *Database Administrator Authentication Methods*



On most operating systems, OS authentication for database administrators involves placing the OS username of the database administrator in a special group (on UNIX systems, this is the dba group) or giving that OS username a special process right.

Additional Information: For information about OS authentication of database administrators, see your Oracle operating system-specific documentation.

The database uses password files to keep track of database usernames who have been granted the SYSDBA and SYSOPER privileges. These privileges allow database administrators to perform the following actions:

SYSOPER	Permits you to perform STARTUP, SHUT DOWN, ALTER DATABASE OPEN/MOUNT, ALTER DATABASE BACKUP, ARCHIVE LOG, and RECOVER, and includes the RE STRICTED SESSION privilege.
SYSDBA	Contains all system privileges with ADMIN OPTION, and the SYSOPER system privilege; permits CREATE DATABASE and time-based recovery.

For information about password files, see the Oracle8 Server Administrator's Guide.

User Tablespace Settings and Quotas

As part of every user's security domain, the database administrator can set several options regarding tablespace usage:

- The user's default tablespace
- The user's temporary tablespace
- Space usage quotas on tablespaces of the database for the user

Default Tablespace

When a user creates a schema object without specifying a tablespace to contain the object, Oracle places the object in the user's default tablespace. You set a user's default tablespace when the user is created; you can change it after the user has been created.

Temporary Tablespace

When a user executes a SQL statement that requires the creation of a temporary segment, Oracle allocates that segment in the user's temporary tablespace.

Tablespace Access and Quotas

You can assign to each user a tablespace quota for any tablespace of the database. Doing so can accomplish two things:

- You allow the user to use the specified tablespace to create objects, provided that the user has the appropriate privileges.
- You can limit the amount of space allocated for storage of a the's objects in the specified tablespace.

By default, each user has no quota on any tablespace in the database. Therefore, if the user has the privilege to create some type of schema object, he or she must also have been either assigned a tablespace quota in which to create the object or been given the privilege to create that object in the schema of another user who was assigned a sufficient tablespace quota.

You can assign two types of tablespace quotas to a user: a quota for a specific amount of disk space in the tablespace (specified in bytes, kilobytes, or megabytes), or a quota for an unlimited amount of disk space in the tablespace. You should assign specific quotas to prevent a user's objects from consuming too much space in a tablespace.

Tablespace quotas and temporary segments have no effect on each other:

- Temporary segments do not consume any quota that a user might possess.
- Temporary segments can be created in a tablespace for which a user has no quota.

You can assign a tablespace quota to a user when you create that user, and you can change that quota or add a different quota later.

Revoke a user's tablespace access by altering the user's current quota to zero. With a quota of zero, the user's objects in the revoked tablespace remain, but the objects cannot be allocated any new space.

The User Group PUBLIC

Each database contains a user group called PUBLIC. The PUBLIC user group provides public access to specific schema objects (tables, views, and so on) and provides all users with specific system privileges. Every user automatically belongs to the PUBLIC user group.

As members of PUBLIC, users may see (select from) all data dictionary tables prefixed with USER and ALL. Additionally, a user can grant a privilege or a role to PUBLIC. All users can use the privileges granted to PUBLIC.

You can grant (or revoke) any system privilege, object privilege, or role to PUBLIC. See Chapter 25, "Privileges and Roles" for more information on privileges and roles. However, to maintain tight security over access rights, grant only privileges and roles of interest to all users to PUBLIC.

Granting and revoking some system and object privileges to and from PUBLIC can cause every view, procedure, function, package, and trigger in the database to be recompiled.

PUBLIC has the following restrictions:

- You cannot assign tablespace quotas to PUBLIC, although you can assign the UNLIMITED TABLESPACE system privilege to PUBLIC.
- You can create database links and synonyms as PUBLIC (using CREATE PUBLIC DATABASE LINK/ SYNONYM), but no other object can be owned by PUBLIC. For example, the following statement is not legal:
- CREATE TABLE public.emp . . . ;

Note: Rollback segments can be created with the keyword PUBLIC, but these are not owned by the PUBLIC user group. All rollback segments are owned by SYS. See Chapter 2, "Data Blocks, Extents, and Segments", for more information about rollback segments.

User Resource Limits and Profiles

You can set limits on the amount of various system resources available to each user as part of a user's security domain. By doing so, you can prevent the uncontrolled consumption of valuable system resources such as CPU time.

This resource limit feature is very useful in large, multiuser systems, where system resources are very expensive. Excessive consumption of these resources by one or more users can detrimentally affect the other users of the database. In single-user or small-scale multiuser database systems, the system resource feature is not as important, because users' consumption of system resources is less likely to have detrimental impact.

You manage a user's resource limits with his or her profile-a named set of resource limits that you can assign to that user. Each Oracle database can have an unlimited number of profiles. Oracle allows the security administrator to enable or disable the enforcement of profile resource limits universally.

If you set resource limits, a slight degradation in performance occurs when users create sessions. This is because Oracle loads all resource limit data for the user when a user connects to a database.

Types of System Resources and Limits

Oracle can limit the use of several types of system resources, including CPU time and logical reads. In general, you can control each of these resources at the session level, the call level, or both:

Session Level	Each time a user connects to a database, a session is created. Each session consumes CPU time and memory on the computer that executes Oracle. You can set several resource limits at the session level. If a user exceeds a session-level resource limit, Oracle terminates (rolls back) the current statement and returns a message indicating the session limit has been reached. At this point, all previous statements in the current transaction are intact, and the only operations the user can perform are COMMIT, ROLLBACK, or disconnect (in this case, the current transaction is committed); all other operations produce an error. Even after the transaction is committed or rolled back, the user can accomplish no more work during the current session.
Call Level	Each time a SQL statement is executed, several steps are taken to process the statement. During this processing, several calls are made to the database as part of the different execution phases. To prevent any one call from using the system excessively, Oracle allows you to set several resource limits at the call level. If a user exceeds a call-level resource limit, Oracle halts the processing of the statement, rolls back the statement, and returns an error. However, all previous statements of the current transaction remain intact, and the user's session remains connected.

CPU Time

When SQL statements and other types of calls are made to Oracle, an amount of CPU time is necessary to process the call. Average calls require a small amount of CPU time. However, a SQL statement involving a large amount of data or a runaway query can potentially consume a large amount of CPU time, reducing CPU time available for other processing.

To prevent uncontrolled use of CPU time, you can limit the CPU time per call and the total amount of CPU time used for Oracle calls during a session. The limits are set and measured in CPU one-hundredth seconds (0.01 seconds) used by a call or a session.

Logical Reads

Input/output (I/O) is one of the most expensive operations in a database system. I/O intensive statements can monopolize memory and disk use and cause other database operations to compete for these resources.

To prevent single sources of excessive I/O, Oracle let you limit the logical data block reads per call and per session. Logical data block reads include data block reads from both memory and disk. The limits are set and measured in number of block reads performed by a call or during a session.

Other Resources

Oracle also provides for the limitation of several other resources at the session level:

- You can limit the number of concurrent sessions per user. Each user can create only up to a predefined number of concurrent sessions.
- You can limit the idle time for a session. If the time between Oracle calls for a session reaches the idle time limit, the current transaction is rolled back, the session is aborted, and the resources of the session are returned to the system. The next call receives an error that indicates the user is no longer connected to the instance. This limit is set as a number of elapsed minutes.

  Note: Shortly after a session is aborted because it has exceeded an idle time limit, the process monitor (PMON) background process cleans up after the aborted session. Until PMON completes this process, the aborted session is still counted in any session/user resource limit.
- You can limit the elapsed connect time per session. If a session's duration exceeds the elapsed time limit, the current transaction is rolled back, the session is dropped, and the resources of the session are returned to the system. This limit is set as a number of elapsed minutes.

  Note: Oracle does not constantly monitor the elapsed idle time or elapsed connection time. Doing so would reduce system performance. Instead, it checks every few minutes. Therefore, a session can exceed this limit slightly (for example, by five minutes) before Oracle enforces the limit and aborts the session.
- You can limit the amount of private SGA space (used for private SQL areas) for a session. This limit is only important in systems that use multithreaded server configuration; otherwise, private SQL areas are located in

the PGA. This limit is set as a number of bytes of memory in an instance's SGA. Use the characters "K" or "M" to specify kilobytes or megabytes.

Instructions on enabling and disabling resource limits are included in the Oracle8 Server Administrator's Guide.

Profiles

A profile is a named set of specified resource limits that can be assigned to valid username of an Oracle database. Profiles provide for easy management of resource limits.

When to Use Profiles

You need to create and manage user profiles only if resource limits are a requirement of your database security policy. To use profiles, first categorize the related types of users in a database. Just as roles are used to manage the privileges of related users, profiles are used to manage the resource limits of related users. Determine how many profiles are needed to encompass all types of users in a database and then determine appropriate resource limits for each profile.

Determining Values for Resource Limits of a Profile

Before creating profiles and setting the resource limits associated with them, you should determine appropriate values for each resource limit. You can base these values on the type of operations a typical user performs. For example, if one class of user does not normally perform a high number of logical data block reads, then the LOGICAL_READS_PER_SESSION and LOGICAL_READS_PER_CALL limits should be set conservatively.

Usually, the best way to determine the appropriate resource limit values for a given user profile is to gather historical information about each type of resource usage. For example, the database or security administrator can use the AUDIT SESSION option to gather information about the limits CONNECT_TIME, LOGICAL_READS_PER_SESSION, and LOGICAL_READS_PER_CALL. See Chapter 26, "Auditing", for more information.

You can gather statistics for other limits using the Monitor feature of Enterprise Manager, specifically the Statistics monitor.

Licensing

Oracle is usually licensed for use by a maximum number of named users or by a maximum number of concurrently connected users. The database administrator (DBA) is responsible for ensuring that the site complies with its license agreement. Oracle's licensing facility helps the DBA monitor system use by tracking and limiting the number of sessions concurrently connected to an instance or the number of users created in a database.

If the DBA discovers that more than the licensed number of sessions need to connect, or more than the licensed number of users need to be created, he or she can upgrade the Oracle license to raise the appropriate limit. (To upgrade an Oracle license, you must contact your Oracle representative.)

Note: When Oracle is embedded in an Oracle application (such as Oracle Office), run on some older operating systems, or purchased for use in some countries, it is not licensed for either a set number of sessions or a set group of users. In such cases only, the Oracle licensing mechanisms do not apply and should remain disabled.

The following sections explain the two major types of licensing available for Oracle.

Concurrent Usage Licensing

In concurrent usage licensing, the license specifies a number of concurrent users, which are sessions that can be connected concurrently to the database on the specified computer at any time. This number includes all batch processes and online users. If a single user has multiple concurrent sessions, each session counts separately in the total number of sessions. If multiplexing software (such as a TP monitor) is used to reduce the number of sessions directly connected to the database, the number of concurrent users is the number of distinct inputs to the multiplexing front end.

The concurrent usage licensing mechanism allows a DBA to:

- Set a limit on the number of concurrent sessions that can connect to an instance by setting the LICENSE_MAX_SESSIONS parameter. Once this limit is reached, only users who have the RESTRICTED SESSION system privilege can connect to the instance; this allows DBA to kill unneeded sessions, allowing other sessions to connect.
- Set a warning limit on the number of concurrent sessions that can connect to an instance by setting the LICENSE_SESSIONS_WARNING parameter. Once the warning limit is reached, Oracle allows additional sessions to connect (up to the maximum limit described above), but sends a warning message to any user who connects with RESTRICTED SESSION privilege and records a warning message in the database's ALERT file.

The DBA can set these limits in the database's parameter file so that they take effect when the instance starts and can change them while the instance is running (using the ALTER SYSTEM command). The latter is useful for databases that cannot be taken offline.

The session licensing mechanism allows a DBA to check the current number of connected sessions and the maximum number of concurrent sessions since the instance started. The V$LICENSE view shows the current settings for the license limits, the current number of sessions, and the highest number of concurrent sessions since the instance started (the session "high water mark"). The DBA can use this information to evaluate the system's licensing needs and plan for system upgrades.

For instances running with the Parallel Server, each instance can have its own concurrent usage limit and warning limit. The sum of the instances' limits must not exceed the site's concurrent usage license. See the Oracle8 Server Administrator's Guide for more information.

The concurrent usage limits apply to all user sessions, including sessions created for incoming database links. They do not apply to sessions created by Oracle or to recursive sessions. Sessions that connect through external multiplexing software are not counted separately by the Oracle licensing mechanism, although

each contributes individually to the Oracle license total. The DBA is responsible for taking these sessions into account.

Named User Licensing

In named user licensing, the license specifies a number of named users, where a named user is an individual who is authorized to use Oracle on the specified computer. No limit is set on the number of sessions each user can have concurrently, or on the number of concurrent sessions for the database.

Named user licensing allows a DBA to set a limit on the number of users that are defined in a database, including users connected via database links. Once this limit is reached, no one can create a new user. This mechanism assumes that each person accessing the database has a unique user name in the database and that no two (or more) people share a user name.

The DBA can set this limit in the database's parameter file so that it takes effect when the instance starts and can change it while the instance is running (using the ALTER SYSTEM command). The latter is useful for databases that cannot be taken offline.

Review Question

1. What are the different level of Security

Selected Bibliography

- [ARIES] C. Mohan, et al.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging., TODS 17(1): 94-162 (1992).
- [CACHE] C. Mohan: Caching Technologies for Web Applications, A Tutorial at the Conference on Very Large Databases (VLDB), Rome, Italy, 2001.
- [CODASYL] ACM: CODASYL Data Base Task Group April 71 Report, New York, 1971.
- [CODD] E. Codd: A Relational Model of Data for Large Shared Data Banks. ACM 13(6):377-387 (1970).
- [EBXML] http://www.ebxml.org.
- [FED] J. Melton, J. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, K. Zeidenstein: SQL and Management of External Data', SIGMOD Record 30(1):70-77, 2001.
- [GRAY] Gray, et al.: Granularity of Locks and Degrees of Consistency in a Shared Database., IFIP Working Conference on Modelling of Database Management Systems, 1-29, AFIPS Press.
- [INFO] P. Lyman, H. Varian, A. Dunn, A. Strygin, K. Swearingen: How Much Information? at http://www.sims.berkeley.edu/research/projects/how-much-info/.
- [LIND] B. Lindsay, et. al: Notes on Distributed Database Systems. IBM Research Report RJ2571, (1979).

Notes:

ADDITIONAL QUESTION

Files and Databases

Major Questions:

What's the difference between a database and a DBMS?

What's the difference between a DBMS and a traditional file processing system?

- What is data (in)dependence?
- Why are databases important?
- How do file systems work?
  - Data structure is defined in application programs
  - Require "file maintenance" programs (reorder, delete, etc.)
- What are the disadvantages of file systems?
  - Data dependence - occurs when changes in file characteristics, such as changing a field from integer to real, requires changes in all programs that access the file.
  - Proliferation of files
  - Data redundancy, which leads to:
    - data inconsistency - occurs when one occurrence of a redundant field is changed but another is not
    - data anomalies - occurs when one occurrence of a redundant field is changed but another is not, or requires changes to data in multiple places
  - Requires programming expertise to access
  - Non standard names
- How do DBMS address these disadvantages?
  - Software to provide common means of accessing data
  - Data definitions centralized in DBMS (data independence)
- What are the functions of a DBMS?
  - Data definition (dictionary)
  - Manages physical data storage
  - Security management
  - Multi-user access control
  - Backup and recovery management
  - Data integrity
  - Data access language
  - Communication (web, client/server, etc.)
- What is a database model?
  - A way of conceptually organizing data
- What are four major database models?
  - Relational
  - Network
  - hierarchical
  - object-oriented
- What are the advantages of the relational model?
  - Use of SQL
  - Intuitive structure
  - Structural independence through views
- Why do we carefully, methodically, design a database?
  - Even a good DBMS will perform poorly with a badly designed database.
  - A poorly designed database encourages redundant data, i.e., unnecessarily duplicated data, which often makes it difficult to trace errors.
  - Redundant data refers to the condition in which the same data are kept in different locations for the same entity (a person, place, or thing for which data are to be collected and stored.)
  - example: Customer telephone numbers are stored in a customer file, a sales agent file, and an invoice file. If a number is changed then the possibility exists that at least one occurrence of the number will be overlooked, resulting in incorrect information in the database.

The Relational Model

What is an entity?

- Something that we intend to collect data about

What is an attribute?

- A characteristic of an entity

What are the main characteristics of a relational table?

- Two dimensional - composed of rows and columns
- Each row represents information about a single entity
- Each column has a name and represents attributes of that entity set
- Each row/column intersection represents data about an instance of an entity
- Each table must have a primary key
- All values in a column are from the same domain
- Row order is immaterial

How is a table different from a database?

- A database is a collect of related tables
- Database includes metadata (relationships, etc.)

What is a primary key and how is it related to the concept of determination and functional dependence?

- A primary key uniquely identifies a row in a table
- It must be able to determine each attribute value in the row
- Each of the attributes are functionally determined by the key

What is a composite key?

- A key comprised of more than one column from the table

What is a secondary key?

- A column used for data retrieval purposes
- Not truly a key, because it doesn't uniquely identify a row

What is a foreign key?

- An attribute in a table that serves as the primary key of another table in the same database.

What is controlled redundancy and the role of foreign keys in the relational model?

- Allows tables to be "linked" through common values - that is a foreign key introduces some necessary redundancy

What is referential integrity?

- If a foreign key contains a value, the value must refer to an existing row in another relation

What is entity integrity?

- There must be a value in the primary key of a row

What information is stored in the system catalog, and how is it typically accessed?

- Metadata, including:
- Table names
- Who and when table created
- Column names
- Indexes
- Etc.

Introduction to SQL and Data Definition

- Designed for data manipulation in a relational database
- Nonprocedural language-specify what, not how
- ANSI standard does exist, but most DBMS packages provide a superset of SQL commands (while perhaps not fully supporting the standard)
- Most recent standard is SQL-99

Parts of SQL

- Data definition
  - create database, tables, define constraints, etc.
- Data manipulation
  - queries to select data from database, aggregate the data, etc.
- Data control language (security)
  - grant and revoke privileges to view, change data

Data definition operations

- Create table
- Define columns
- Define data types
- Define indexes, keys
- Alter table

Data manipulation operations

- Select
- Insert
- Update
- Delete

Data Definition Examples:

To create a table:

```
create table student
(studentID text (10),
studentName text (25),
studentAddress text (25));
```

To create a table with a primary key:

```
create table student
(studentID text (10) PRIMARY KEY,
studentName text (25),
studentAddress text (25));
```

To add an index:

```
create index studentNameIndex on
student (studentName);
```

Add a non-null GPA column:

```
alter table student
add column GPA integer not null;
```

Because gpa was defined wrong, need to drop GPA:

```
alter table student
drop gpa;
```

And then add it again:

```
alter table student
add column gpa single;
```

update a column:

```
update  student
set gpa = 3.5
where studentName = "Aytes";
```

Additional Info:

- NOT NULL specification insures that the primary key cannot be null, thereby enforcing entity integrity.
- Specification of the primary key as UNIQUE results in automatic enforcement of entity integrity.
- Specification of the foreign key allows the enforcement of referential integrity.
- Attribute descriptions are enclosed in parentheses.
- Attribute descriptions are separated by commas.
- The command sequence is terminated by a semicolon, but some RDBMSs do not require it.
- In MS Access, each SQL statement must be entered as a separate query.

Inserting Data

INSERT lets you insert data into the table, one row at a time. It is used to make the first data entry into a new table structure or to add data to a table that already contains data.

Examples:
One way to use the insert statement is to list each of the columns into which you want to insert data, followed by the actual data values. The column names and the values to be inserted must be in the same order.

insert into student
(studentID, studentName, StudentAddress, GPA)
values ("1234", "Johnson", "Main Street", 3.3);

If you do not have a value for a column (and the column allows nulls) you simply do not include it in your list of columns and of course exclude a value as well. For example, if you did not have an address for this student:

insert into student
(studentID, studentName, GPA)
values ("1234", "Johnson", 3.3);

If you have data for all the columns, you can also insert data by simply giving the values in the same order in which the columns are defined in the table:

insert into student
values ("1234", "Johnson", "Main Street", 3.3);

SQL SELECT Introduction
To view and manipulate data within the database, use the Select statement.

There are three important parts of a select statement:

1. Select (required) - names the columns that you want to view
2. From (required) - lists the tables containing the columns you want
3. Where (NOT required) - places constraints on the data that you want. Usually consists of comparison of column names by using relational and boolean operators.

Relational Operators

- = (equal)
- <= (less than or equal)
- >= (greater than or equal)
- <> (not equal)
- between...and

Boolean Operators

- and
- or
- not

Query Examples
Each of the following examples is using the following tables:

Student table

studentID	studentName	courseID
111	Baker	CIS480
112	Johnson	CIS382

Course table

courseID	courseTitle
CIS480	**Database Management Systems**
CIS382	**Systems Analysis and Design**

To retrieve data from a database, use the SELECT command.

SELECT syntax:

SELECT <list of columns>
FROM <list of tables>
WHERE <search conditions>

Explanation of syntax:
<list of columns> is the list of all columns that you want to be able to see in your RESULT SET (TABLE). For example:

SELECT studentID, studentName

If you want to include all columns from the table(s):

SELECT *

<list of tables> is the list of all tables that you need to obtain the data that you want. You may have to include some tables so that you can properly JOIN the data from each of the tables, even if you don't SELECT data from that table. Example using only one table:

SELECT studentID, studentName
FROM student

If you are selecting data from multiple tables, and those tables have column names in common, then Jet SQL requires you to *qualify* the column names. For example, if you are JOINING two tables that use the same name for a column:

*SELECT studentID, studentName, **student**.courseID, courseName*
FROM student, course

<search conditions> is where you describe how you want to limit the results set. You can use any of the operators listed above.

Example:

SELECT studentID, studentName
FROM student
WHERE studentName = "Baker";

Results in:

studentID	studentName
111	**Baker**

Joining Tables
The real power of SQL and relational databases comes when you use the existing relationships (typically defined through foreign keys) to combine related data across multiple tables. This is referred to as JOINING tables.

However, if tables are not joined properly, your results set is probably useless:

Cartesian Product Example - what happens when you do it wrong!

(use Northwinds database for the following queries)

SELECT categories.CategoryID, Products.ProductID
from categories, products;

Use of the Where clause to create an join the tables:
SELECT categories.CategoryID, Products.ProductID
from categories, products
where categories.categoryID = Products.CategoryID;

Use of INNER JOIN:
SELECT categories.CategoryID, Products.ProductID
from categories INNER JOIN products on
categories.categoryID = Products.CategoryID;

Interesting Note
Relational databases store all metadata in tables (called system tables), so you can perform queries on these tables to learn more about your database. This is even true in MS Access, although the system tables are somewhat cryptic:

SELECT *
FROM MSysObjects
ORDER BY MSysObjects.Name;

Aggregate Functions and Arithmetic Operations

The following sample queries use the Northwinds database.

Arithmetic Operations

Arithmetic operations (+, -, *, /) can be performed on columns.

For example, the following query multiplies Unit Price by Quantity to give Extended Price:

*SELECT [order details].orderid, productid, (unitprice * quantity)* ***as*** *[extended price]*
FROM [Order Details];

Note the "as" after the arithmetic operation. This gives a name (sometimes called an alias) to the new computed column.

SQL Group By and Aggregate Functions

Group By is a means of collapsing multiple rows of data into a single row. Usually, you combine a Group By clause with an aggregate function so that you can summarize the data in some of the columns.

Example: List all categories from the Products table:

SELECT CategoryID, supplierID
FROM Products;

Note that CategoryID is actually displayed as CategoryName - see Lookup in design view of table.

Add the Group By clause (which in this case creates same results as using DISTINCT)

SELECT CategoryID, supplierID
FROM Products
group by categoryID, supplierID;

How GROUP BY works:

1. **Creates a temporary table based on the From clause and the Where clause (if present).**
2. **Groups the rows based on the Group By clause.**
3. **Displays the results based on the Select clause.**

Important: Any field that appears in the Select clause, but is not used as part of an aggregate function, **must** also appear in the Group By clause.

Aggregate Functions: Count, Sum, AVG, Min, Max

Aggregate functions summarize the data so that the numeric data can be collapsed into a single row.

This example returns the average unitprice for each category:

SELECT Products.CategoryID, avg(products.unitprice)
FROM Products
group by Products.CategoryID;

This example counts the number of products in each category.

SELECT Products.CategoryID, count (products.productID)
FROM Products
group by Products.CategoryID;

Having Clause

Because the "where" clause in a select statement works only with individual rows, you can use **Having** to limit the groups that are included.

This example lists all categories that have the sum of quantity on hand for all products in that category greater than 100:

SELECT CategoryID, Sum(UnitsInStock)
FROM Products
GROUP BY CategoryID
HAVING Sum(UnitsInStock) > 100

Advanced SQL

All of the following examples use the Northwind database.

Joins

Inner join is used to join tables on common values. It is functionally equivalent to using "where...and".

These two statements are equivalent:

Q1
SELECT Customers.CompanyName, orders.orderid
from customers, orders
where customers.customerid = orders.customerid;

Q2
SELECT Customers.CompanyName, orders.orderid
from customers inner join orders on
customers.customerid = orders.customerid;

You can also **nest inner join statements,** where the results of one join are then joined to another table.

For example, these two statements are equivalent:

Q3
SELECT Customers.CompanyName
from Customers, Orders, [Order Details], Products
where (customers.CustomerID = Orders.CustomerID) and
(Orders.OrderID = [Order Details].OrderID)
and ([Order Details].ProductID = Products.ProductID)
and (Products.ProductName = 'Aniseed Syrup');

Q4
SELECT Customers.CompanyName
from (((Customers inner join Orders on customers.CustomerID =
Orders.CustomerID) inner join [Order Details] on Orders.OrderID =
[Order Details].OrderID)

inner join Products on [Order Details].ProductID = Products.ProductID)
where (Products.ProductName = 'Aniseed Syrup');

Outer Join

- **Outer (left and right) joins** are used to find all matching values (as in an inner join), along with all the unmatched values in one of the tables.
- A left join includes all the values in the left table, plus the matching values in the right table.
- A right join includes all the values in the right table, plus the matching values in the left table.
- This is one of the few instances that the order in which you list the tables matters.

Example

Say a new category is added to the categories table but no products are yet listed in that category. If you wanted to list all the products and which categories they are in, you would use a simple inner join:

Q5

SELECT categories.categoryname, products.productname
from categories INNER join products on products.categoryid = categories.categoryid

However, this would leave out the new category, since there are no matching records in the products table (i.e., there are no products in this category). To list all of the products and which categories they are in, plus any categories that contain no products, you could use the following query:

Q6

SELECT categories.categoryname, products.productname
from categories LEFT join products on products.categoryid = categories.categoryid

Outer joins can be very useful for finding "unmatched" records. For example, to find customers that do not have an order:

Q7

SELECT Customers.companyname, customers.customerid, orders.orderid
FROM Customers LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
where orders.orderid is null;

Nested Subqueries

Because the result of any properly formed select statement is a temporary table, select statements can be used one within another. This means that the results of the nested select statement are then used as if it were a table.

One way of using nested subqueries is through the **IN** clause. The IN clause is a set operator, and can be used to compare a column or expression to a list of values. For example, Q8 below is equivalent to Q7, but is written using the IN clause and a nested subquery:

Q8

SELECT Customers.companyname, customers.customerid
FROM Customers
where customers.customerid NOT IN
(select customers.customerid
from customers inner JOIN Orders ON Customers.CustomerID = Orders.CustomerID)

Subqueries allow for relatively complex set operations. For example, to find all orders that include BOTH productid 30 (Gorgonzola Telino) and productid 60 (Camembert Pierrot):

Q9

SELECT customers.companyname, [order details].orderid, [order details].productid
FROM customers INNER JOIN (Orders INNER JOIN [Order Details] ON Orders.OrderID = [Order Details].OrderID) ON Customers.CustomerID = Orders.CustomerID
where [order details].orderid in
(select [order details].orderid from [order details]
where [order details].productid = 30)
and [order details].orderid in
(select [order details].orderid from [order details]
where [order details].productid = 60)

Query Q9 could be written as three independent, saved queries:

Query productid30 – just orderids that have productid 30

SELECT [order details].[orderid]
FROM [order details]
WHERE [order details].[productid]=30;

Query productid60 – just orderids that have productid 60

SELECT [order details].[orderid]
FROM [order details]
WHERE [order details].[productid]=60;

Query productid3060 – joins the previous two queries

SELECT productid30.[orderid]
FROM productid30 INNER JOIN productid60 ON productid30.orderid = productid60.orderid;

Query Q10 to join the previous query to the customer data

Q10

SELECT customers.companyname, productid3060.orderid
from customers INNER JOIN (Orders INNER JOIN [productid3060] ON Orders.OrderID = productid3060.OrderID) ON Customers.CustomerID = Orders.CustomerID

Note - if you have duplicate records in your results, use the Distinct clause to get rid of the duplicates.

Database Design Process

Database design is part of information system design.

General information system design approaches:

- Process oriented – focus on business processes
  - Model existing/desired processes first (e.g., Data Flow Diagrams)
  - Data stores (database) may already exist
  - If database must be created/altered, database design begins
- Data oriented – focus on data needs
- Model existing/desired data first (e.g., Entity Relationship Diagrams)
- Assumes new data needs or that data can be significantly reorganized

Database Design Phases

Conceptual Data Modeling - Identify entities, attributes, relationships

Logical Database Design – develop well-designed relations

Physical Database Design – determine data types, default values, indexes, storage on disk, including division of application into tiers

Database / system implementation - includes programs, business processes

Conceptual Data Modeling

Modeling the rules of the organization

- Identify and understand those rules that govern data
- Represent those rules so that they can be unambiguously understood by information systems developers and users
- Implement those rules in database technology
- Business rule
- A statement that defines or constrains some aspect of the business.
- Intended to assert business structure or to control the behavior of the business

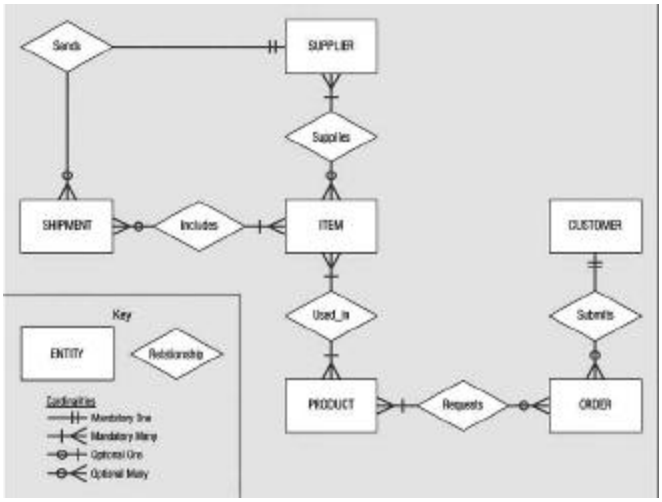
Examples

- Every order must have one and only one shipper
- Each student must have an advisor
- Classes must be assigned to one and only one classroom.
- Where can business rules be found?
- Policies and procedures
- Paper and computer forms
- In employees' heads
- How to represent business rules related to data?
- Entity Relationship Diagrams
- Entity
- Person, place, object, event, or concept in the user environment about which the organization wishes to maintain data
- entity type (collection of entities) often corresponds to a table
- entity instance (single occurrence of an entity) often corresponds to a row in a table
- entities are not the same as system inputs and outputs (e.g., forms, reports)
- entities are not the people that use the system / data
- Attributes
- property or characteristic of an entity type that is of interest to the organization
- identifier attribute uniquely identifies an entity instance (i.e., a key)

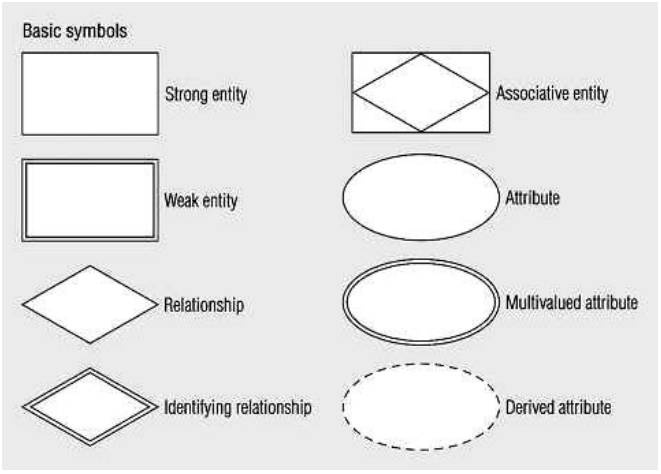
Relationship

- describes the link between two entities
- usually defined by primary key - foreign key equivalencies

Example ERD



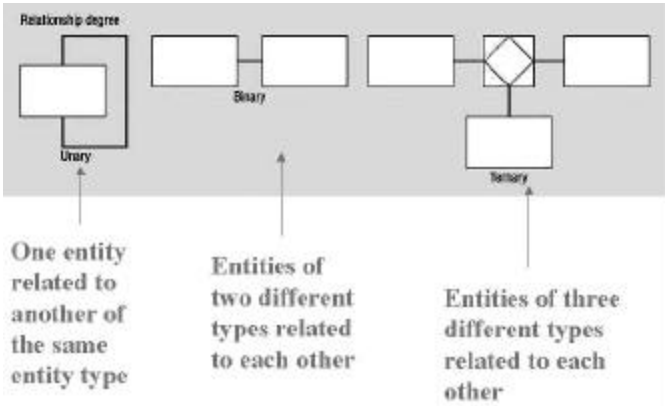
ERD Notation



Degree of a Relationship

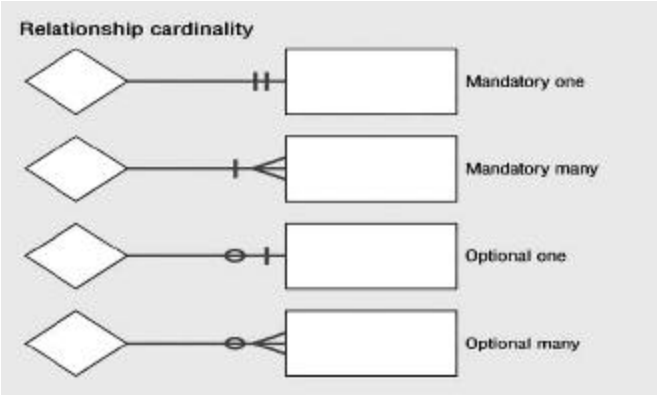
Refers to the number of entity types that participate in it.

- Unary
- Binary
- Ternary



Cardinality of Relationships

- Refers to the number of instances of entity A that can be associated with each instance of entity B
- One-to-one: Each entity in the relationship will have exactly one related entity
- One-to-many: An entity on one side of the relationship can have many related entities, but an entity on the other side will have a maximum of one related entity
- Many-to-many: Entities on both sides of the relationship can have many related entities on the other side
- Cardinality Constraints - the number of instances of one entity that can or must be associated with each instance of another entity.
  - Minimum Cardinality - if zero, then optional If one or more, then mandatory
  - Maximum Cardinality - the maximum number



Normalization

Properties of Relations (Tables)

- entries at intersection of a row and column are atomic (no multivalued attributes in a relation)
- entries in columns are from the same domain
- each row is unique

Well-Structured Relation (Table) - to improve data integrity, ease of application design

- contain minimum redundancy
- avoids errors or inconsistencies (anomalies) when user updates, inserts, or deletes data

Anomalies

- insertion anomaly - cannot insert a fact about one entity until we have an additional fact about another entity
- deletion anomaly - deleting facts about one entity inadvertently deletes facts about another entity
- modification anomaly - must update more than one row in a given table to change one fact in the table

Terminology

- functional dependency - the value of A determines the value of B (Aà B)
- determinant - that attribute on the left side of the arrow (A in the above)
- transitivity rule - if Aà B, and Bà C, then Aà C

Normal Forms

- essence of normalization - keep splitting up relations until there are no more anomalies
- 1st Normal form - atomic values in each column (no repeating groups)
- 2nd Normal form - all nonkey attributes are dependent on all of the key (no partial dependencies)
  - Table not in 2nd Normal form:
  - Professor (empID, courseID name, dept, salary, , date-completed)
    - EmpIDà name, dept, salary
    - EmpID, courseID à date-completed
- 3rd Normal form - in 2nd normal form, and all transitive dependencies have been removed
  - Table not in 3rd Normal Form:
- Sales(custno, name, salesperson, region)
- Custnoà name, salesperson, region
- Salespersonà region
- Insertion anomaly: a new salesperson cannot be assigned to a region until a customer has been assigned to that salesperson
- Deletion anomaly: if a customer is deleted from the table, we may lose the information about who is assigned to that region
- Modification anomaly: if a salesperson is reassigned to a different region, several rows must be changed to reflect that.

Physical Database Design

Objectives of physical database design

- Efficient processing (i.e., speed)
- Efficient use of space

Data and Volume Analysis

1. Estimate the size of each table
2. Estimate the number/type of access to each table

Designing Fields

Choosing data types

Methods for controlling data integrity

- default values
- range control
- null control
- referential integrity

Denormalization

- Objective of denormalization: to create tables that are more efficiently queried than fully normalized tables.
- Denormalization trades off a potential loss of data integrity for query speed
- Denormalization opportunities
  - Combine two tables with a one-to-one relationship (Fig 6-3)

- Combine three tables in an associative relationship into two tables. (Fig 6-4)
- Reference data - combine two tables into one where for a 1:N relationship where 1-side has data not used in any other relationship (Fig. 6.5)

Partitioning

- Horizontal Partitioning: Distributing the rows of a table into several separate files
  - Useful for situations where different users need access to different rows
  - Vertical Partitioning: Distributing the columns of a table into several separate files
  - Useful for situations where different users need access to different columns
- The primary key must be repeated in each file
- Advantages of Partitioning:
  - Records used together are grouped together
  - Each partition can be optimized for performance
  - Security, recovery
  - Partitions stored on different disks:
    - Reduce contention
    - Take advantage of parallel processing capability
- Disadvantages of Partitioning:
  - Slow retrievals across partitions
  - Complexity

Indexing

- Index – a separate table that contains organization of records for quick retrieval
- Primary keys are automatically indexed
- Disadvantage of index - frequent changes to database requires that index be updated, increasing processing time
- When to use indexes
  - Use on larger tables
  - Index the primary key of each table
  - Index search fields (fields frequently in WHERE clause)
  - Fields in SQL ORDER BY and GROUP BY commands
  - When there are >100 values but not when there are <30 values
  - Use indexes heavily for non-volatile databases; limit the use of indexes for volatile databases

Query Optimization

- Wise use of indexes
- Compatible data types
- Simple queries
- Avoid query nesting
- Temporary tables for query groups
- Select only needed columns
- No sort without index

Transaction Processing and Database Backup/Recovery

A DBMS typically provides several means of backing up and recovering a database in the event of a hardware or software failure:

- Backup facilities - periodic backup copies of the database
- Journalizing facilities - records transactions resulting in database changes (audit trail)
- Checkpoint facility - means of suspending processing and synchronizing internal files and journals
- Recovery manager - means of using the above three facilities to recover a database after a failure.

A DBMS must have a way of knowing how various database accesses/changes are related to each other. This is done through the concept of a database "transaction."

Journalizing Facilities

An audit trail is created so that

- in the event of a failure, the database can be put back in a known state (database change log - before and after images)
- it can be determined when a change was made and who made it (transaction log)

Transaction - a series of database accesses (reads and writes) that must be performed as a cohesive unit in order to maintain data integrity.

Example: entering a customer order

Transaction boundaries are defined in SQL by

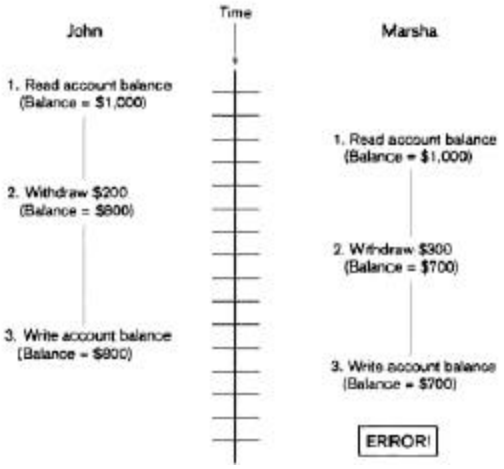
- Placing BEGIN TRANSACTION as the first SQL statement in a series of SQL commands comprising the transaction
- Placing COMMIT at the end of the transaction
- If there is an error, executing a ROLLBACK command

Concurrency Control

Problem – in a multi-user environment, simultaneous access to data can result in interference and data loss

Solution – Concurrency Control: The process of managing simultaneous operations against a database so that data integrity is maintained and the operations do not interfere with each other in a multi-user environment.

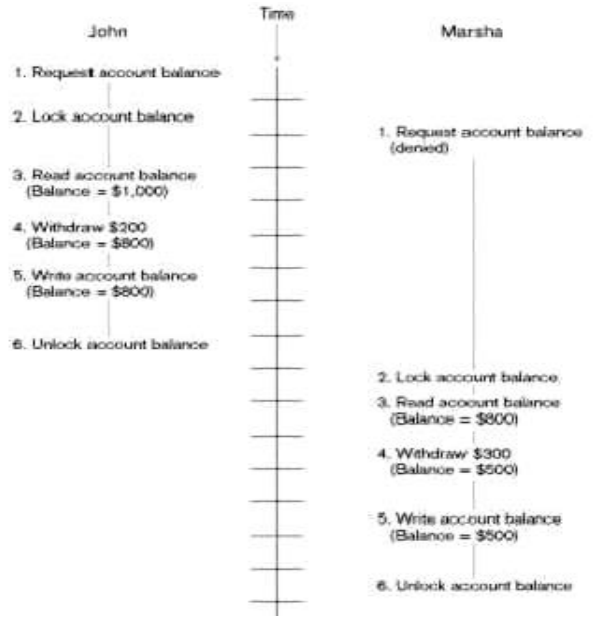
Example of lack of concurrency control causing a lost update:



Concurrency Control Solution

- Serializability – Finish one transaction before starting another
- Locking Mechanisms
  - The most common way of achieving serialization
  - Data that is retrieved for the purpose of updating is locked for the updater
  - No other user can perform update until unlocked

Locking in action:



Databases can be locked at various levels:

- Database
- Table
- Block or page
- Record level
- Field level

Deadlock can occur when two different transactions are waiting for resources the other already has locked.

Client/Server Architecture

- Networked computing model
- Processes distributed between clients and servers
- Client – Workstation (usually a PC) that requests and uses a service
- Server – Computer (PC/mini/mainframe) that provides a service
- Presentation Logic (GUI)
  - Input – keyboard/mouse
  - Output – monitor/printer
- Processing Logic
  - I/O processing
- Business rules
- Storage Logic
  - Data storage/retrieval

File Server (First Generation Client/Server)

- All processing is done at the PC that requested the data
- Entire files are transferred from the server to the client for processing.
- Problems:
  - Huge amount of data transfer on the network
  - Each client must contain full DBMS
  - Heavy resource demand on clients
  - Data integrity - synchronizing data

Database Server Architecture

Two-tiered approach

Client is responsible for

- I/O processing logic
- Some business rules logic

Server performs all data storage and access processing - DBMS is only on server

Advantages

- Clients do not have to be as powerful
- Greatly reduces data traffic on the network
- Improved data integrity since it is all processed centrally
- Stored procedures - some business rules done on server

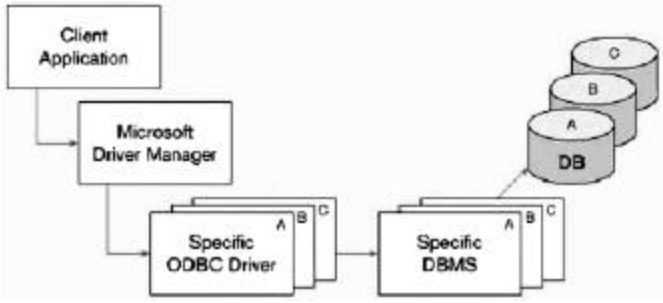
Three-Tier Architecture

- Client - GUI (presentation layer)
- Application server - business rules (application logic layer)
- Database server - data storage (DBMS layer)
- Main advantage - technological flexibility (you can change things at one layer with less affect on other layers)

Software to make client/server architecture work is called middleware

Database middleware:

- ODBC – Open Database Connectivity
  - Most DB vendors support this
  - API that provides a common language for application programs to access and process SQL databases independent of the particular RDBMS that is accessed



ODBC Architecture (above)

- OLE-DB Microsoft enhancement of ODBC
- JDBC – Java Database Connectivity
  - Special Java classes that allow Java applications/applets to connect to databases

ASSIGNMENT

ASSIGNMENT - I

CIS 480

SQL Exercise One

The tables below show some sample data for a suppliers-and-parts database. Suppliers, parts, and projects are uniquely identified by supplier number, partnumber, and project number, respectively. The significance of the shipment relation is that the specified supplier supplies the specified part to the specified project in the specified quantity (therefore, the combination of supplier number, part number, and project number uniquely identifies each row in the shipment table). In MS Access, write a suitable set of CREATE TABLE statements for this database, and then enter at least one record into each table. By using the sample data below, define the columns appropriately as you define the tables. Define keys and indexes as appropriate (keys are in bold). *NOTE - The textbook gives examples for Oracle. Access uses slightly different SQL, so the statements may need to be modified slightly from what your textbook shows to execute in Access.*

To complete this assignment, first open a new database in Access. You will be creating these tables through SQL, *not by using the Design view of the Tables window.* To write the SQL in Access, you need to view the Queries window, then select the Design view. Close the Show Tables dialog box, then select SQL in the upper left corner of the window. You'll now be presented with a window in which you can enter SQL (it assumes you want to write a Select statement). Once you have written your first SQL statement, run the SQL by clicking on the exclamation point on the menu bar, or choose Query and then Run from the menu bar.

If you make a mistake and need to recreate a table, remember to first delete the table you already created. Otherwise you will get an error when you run your SQL again.

Use the I*nsert Into* statement to add data to your tables. Remember, you only need to insert one row into each table. To turn in your work, copy and past each SQL statement into a Word document, then print the word document.

Supplier Table

Supplier #	Supplier Name	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	10	Athens

Project Table

Project #	Project Name	City
J1	Sorter	Paris
J2	Punch	Rome
J3	Reader	Athens
J4	Console	Athens
J5	Collator	London
J6	Terminal	Oslo
J7	Tape	London

Part Table

Part #	Part Name	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome

CIS 480

SQL Assignment Two

Use the Northwinds database to answer these queries. Copy and paste your SQL (use the WHERE syntax in your SELECT statement instead of an INNER JOIN) and the answers into a word-processed document and turn in at the beginning of class. You may want to make a second copy to look at while we go over it in class.

1. Show all products that are have UnitsInStock that are below the ReorderLevel.
2. What are the last names of all employees born before Jan 1, 1960?
3. Show the company names of all customers that have ordered the product “Aniseed Syrup.”
4. Show the company names of all suppliers who have discontinued products.
5. Write a query that shows how many orders are associated with each customer ID. Note that the customerID field is replaced with the Customer’s name. Why is that?
6. Show total inventory (dollar value) on hand by category.

Show the total dollar value of all the orders we have for each customer (one dollar amount for each customer). Order the list in descending order.

Notes:

SQL Assignment Three

Use the Northwinds database for this assignment.

This assignment is worth 20 points (as opposed to 10 points for most others). Do not seek help from, or offer to help, fellow students on this assignment. If you have questions, ask me.

Turn in both your SQL and the results of your queries in a word-processed document. You do not need to include all rows from the results sets - just give me the first few rows.

1. Get supplier names for all orders.
2. Get order ID and product name for all orders supplied by "Tokyo Traders."
3. Get order ID, product name, and dollar value of the amount of that product ordered for each order supplied by "Tokyo Traders."
4. Get the shipper name and supplier name for all orders shipped by Speedy Express (yes, this seems redundant, since all the records returned will have Speedy Express as the shipper name).
5. Count the number of orders shipped by each shipper.
6. Display the quantity ordered of each product.

Notes:

SQL Assignment Four

1. Display each product name and the latest date the product was ordered. (Hint - use the max function on the orderdate column.)
2. Find the company name of all customers who have ordered Aniseed Syrup and Outback Lager on the same order. Include the orderid in your results. First, solve this by using three separate queries. Then, write the query as a single query with nested subqueries.
3. Find all the product name of all products that have never been ordered. You will have to add a new product to the product table to test this. (Hint - you can use the NOT IN clause and a nested subquery).
4. Display the total number of times each product has been ordered, along with the product name.
5. Display the OrderID and Customer.CompanyName for all orders that total over $5,000 (ignore freight and discount).

Note - for each of these queries, assume that the tables are not using the "lookup" feature. In other words, if, for example, you are required to list the product name, get the product name from the Products table and do not rely on the lookup display feature to display it from the Order Details table.

Notes:

CIS 480

ER Exercise One

Create ER diagrams for the following scenarios. You may simply draw the ER diagrams using paper and pencil.

Scenario One

IYSL is a regional soccer association. Each city in the region has one team that represents it. Each team has a maximum of 15 players and a minimum of 11 players. Each team also has up to three coaches. Each team plays two games (home and visitor) against the all the other teams during the season.

Scenario Two - relationships of organizational entities at Acme:

- A division can operate zero to several projects, but each project belongs to a single division.
- A division has many employees, but each employee is assigned to only one division.
- An employee might be assigned to zero or many projects, and each project has at least one employee assigned to it.
- Each division is managed by one of its employees, and an employee can manage only one division.
- An employee may or may not have any dependents.
- The company maintains a list of desired skills, but not all of these skills are currently represented in the employee pool.

ER Exercise Two

Complete *Problems and Exercises* (NOT Review Questions) numbers 3 (a through e), 4, 5, 6, and 8 on page 120 of your text.

You may draw the ER diagrams by hand or use a computer-based drawing tool. Make a photocopy of your work so can turn in the original at the beginning of class and still have a copy to look at as we discuss the assignment.

Note: All you need to do for each of the problems is draw an ER diagram. You do not have to write out definitions as indicated in problems 3 and 6..

ER Exercise Three

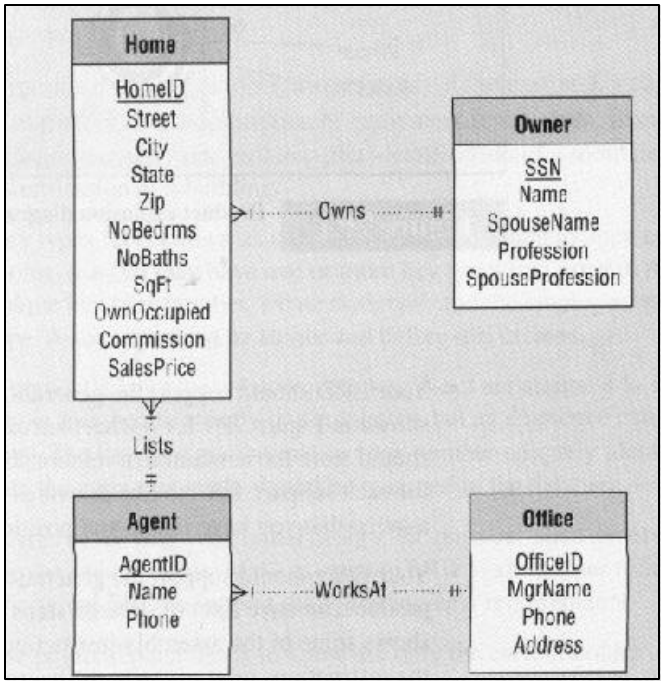
Complete numbers 11 through 15 in Problems and Exercises at the end of Ch. 3 (p. 121-122).

Notes:

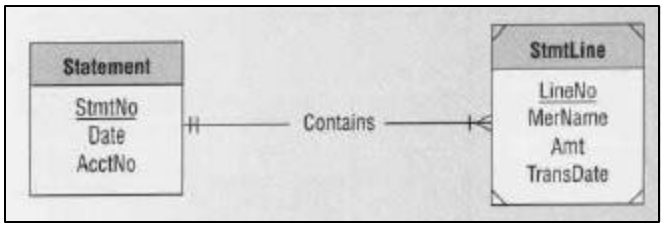
ERD-To-Table Transformation Exercise

Convert each of the following ERDs into a set of tables in third normal form. Write out the contents of each table using a format like this: tablename (field1, field2...). Double underline the primary key and single underline any foreign keys.

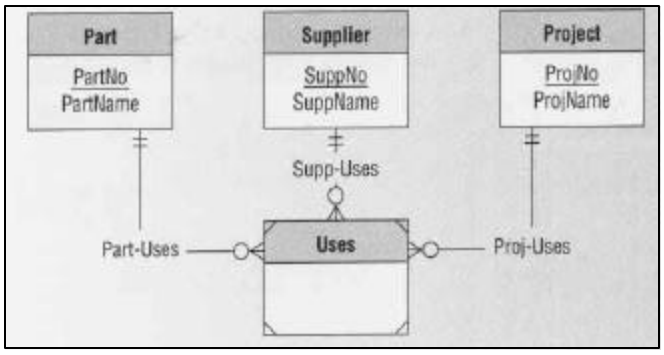
1



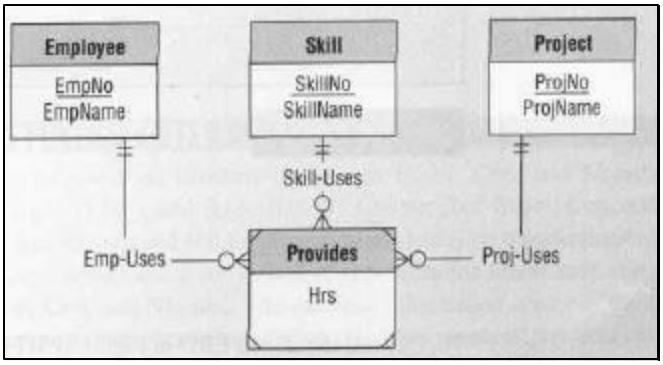
2



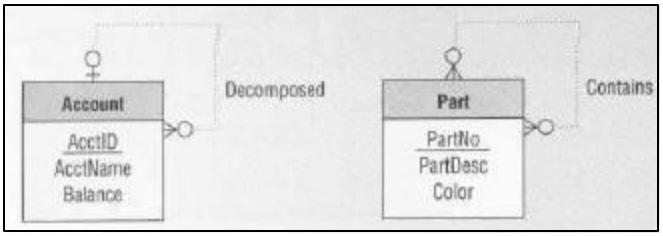
3



4

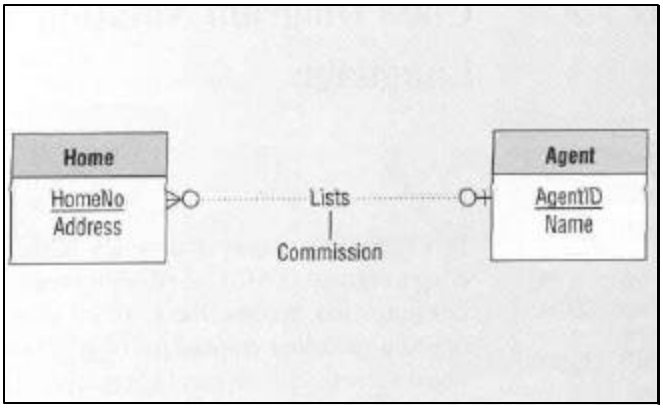


5

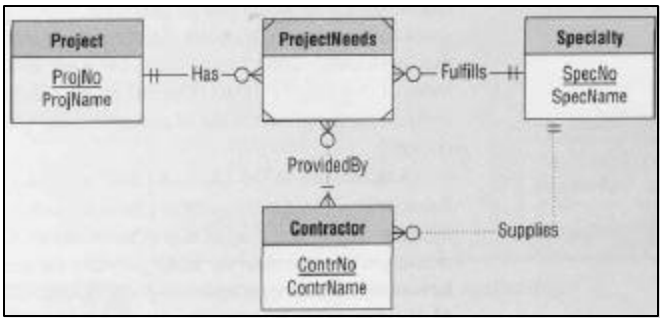


6

(What is shown below is that "commission" is an attribute of the Lists relationship between Home and Agent.)

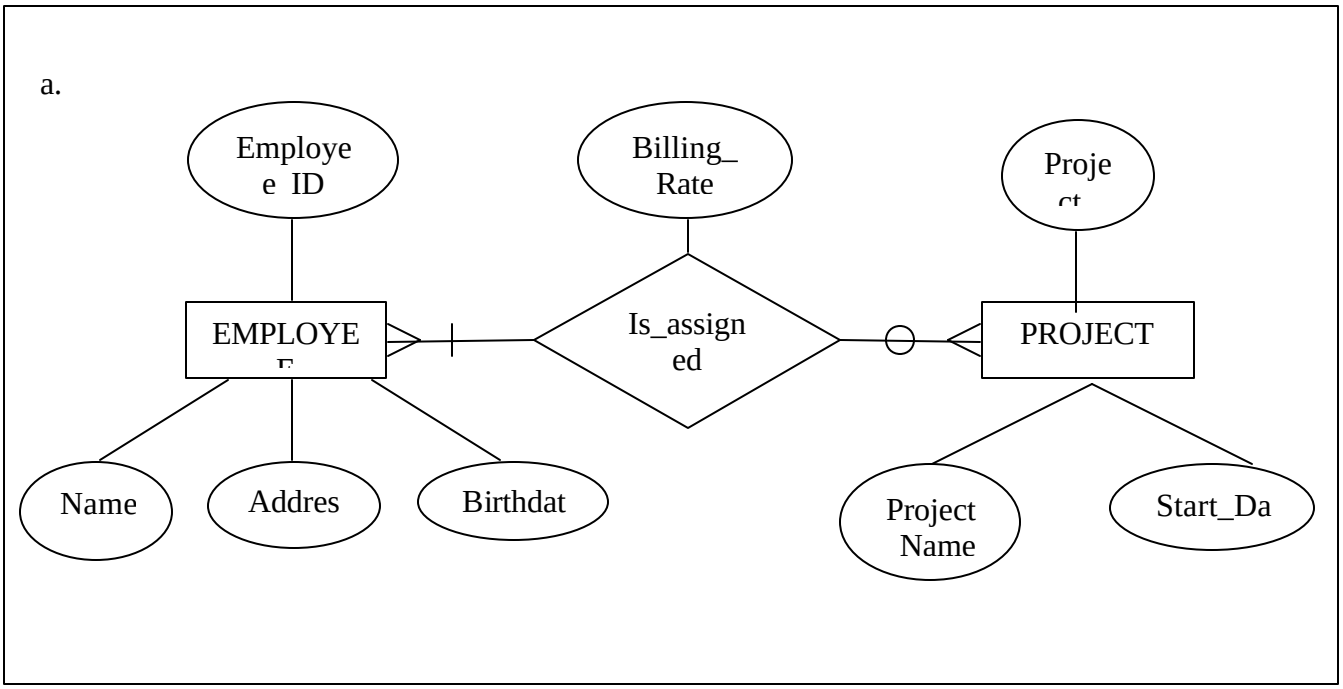


7

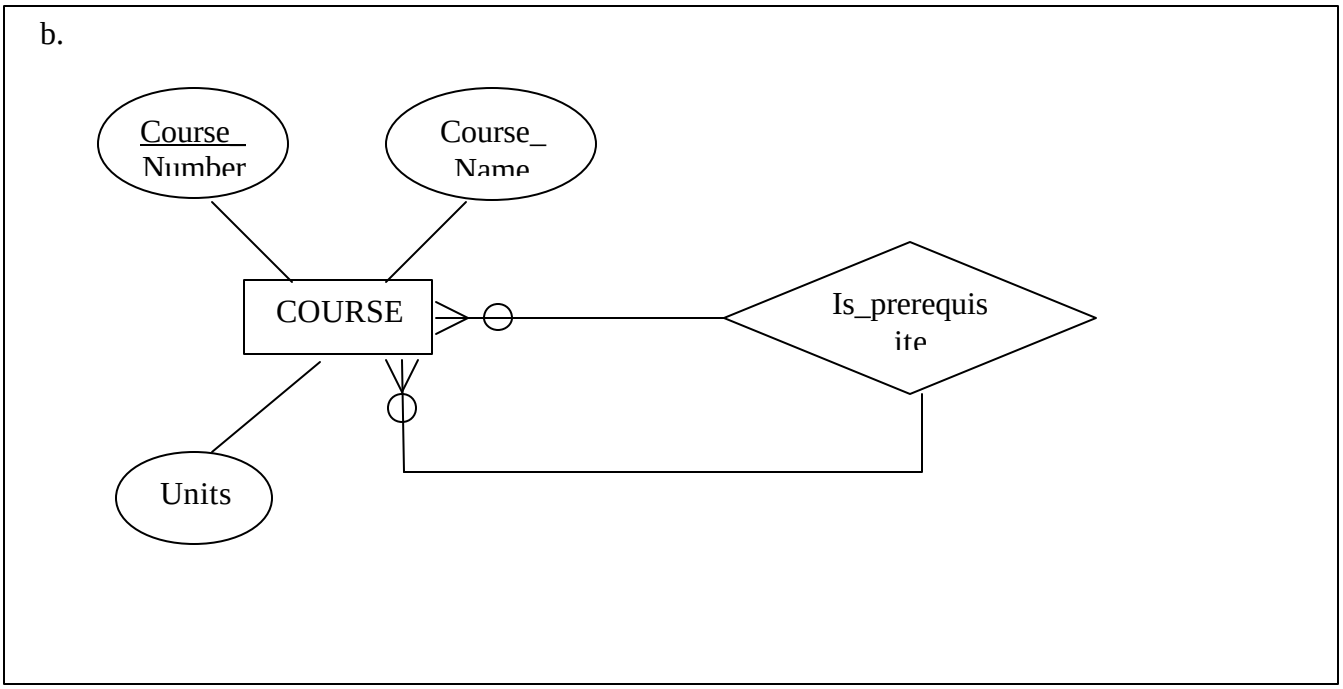


ER Solutions

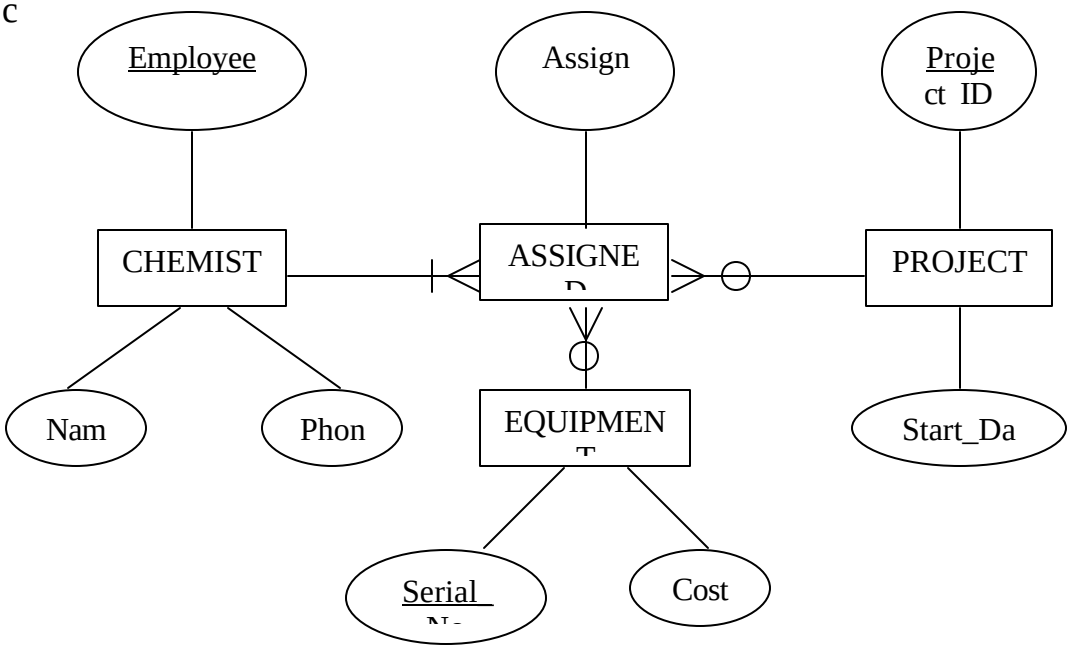
a.



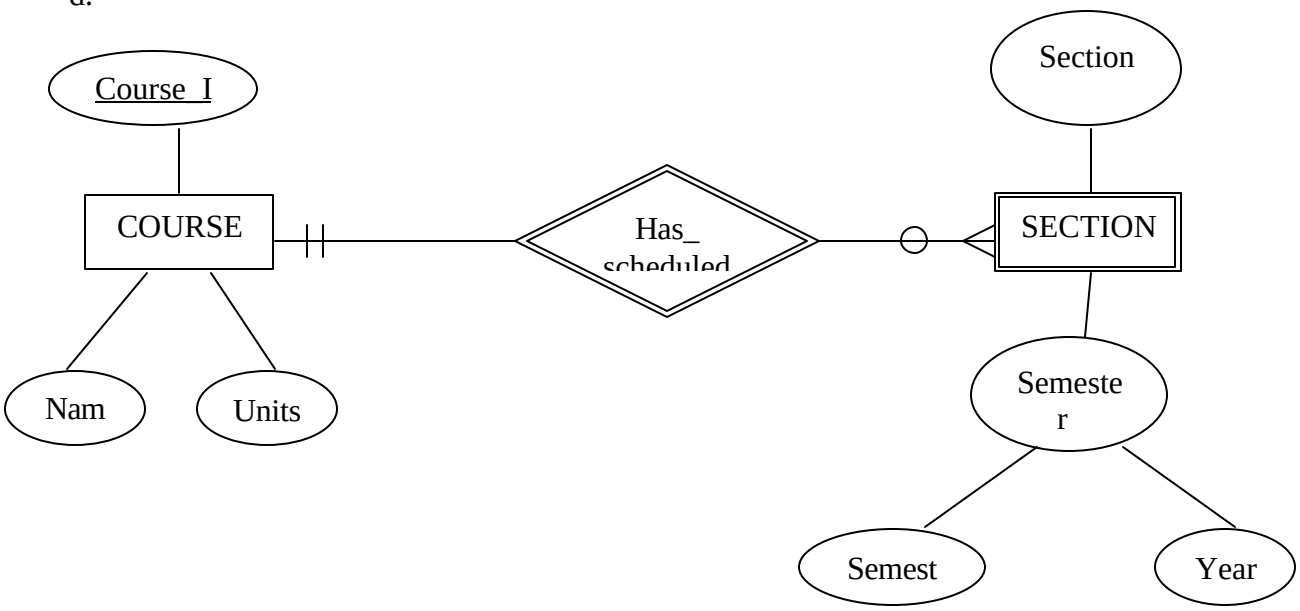
b.

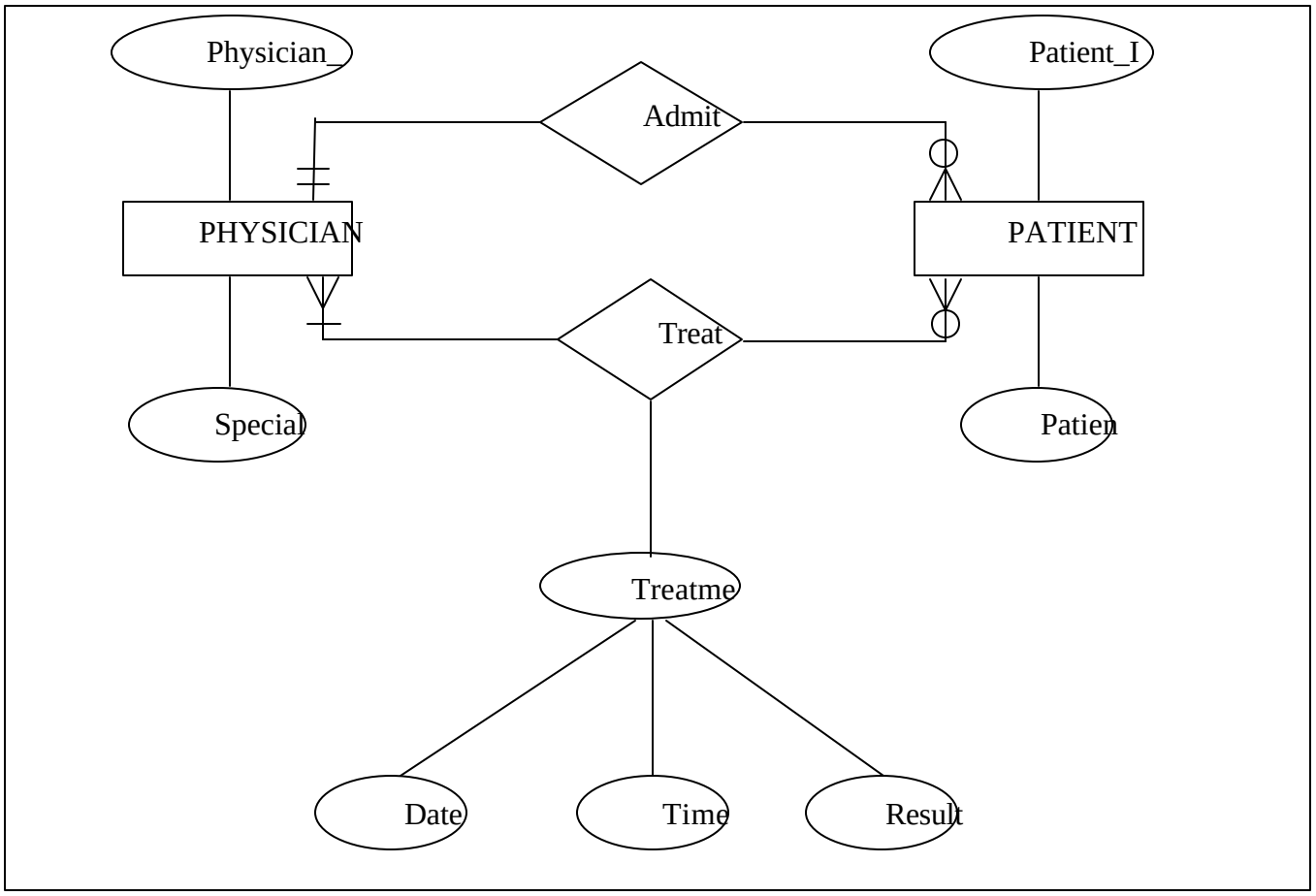


c

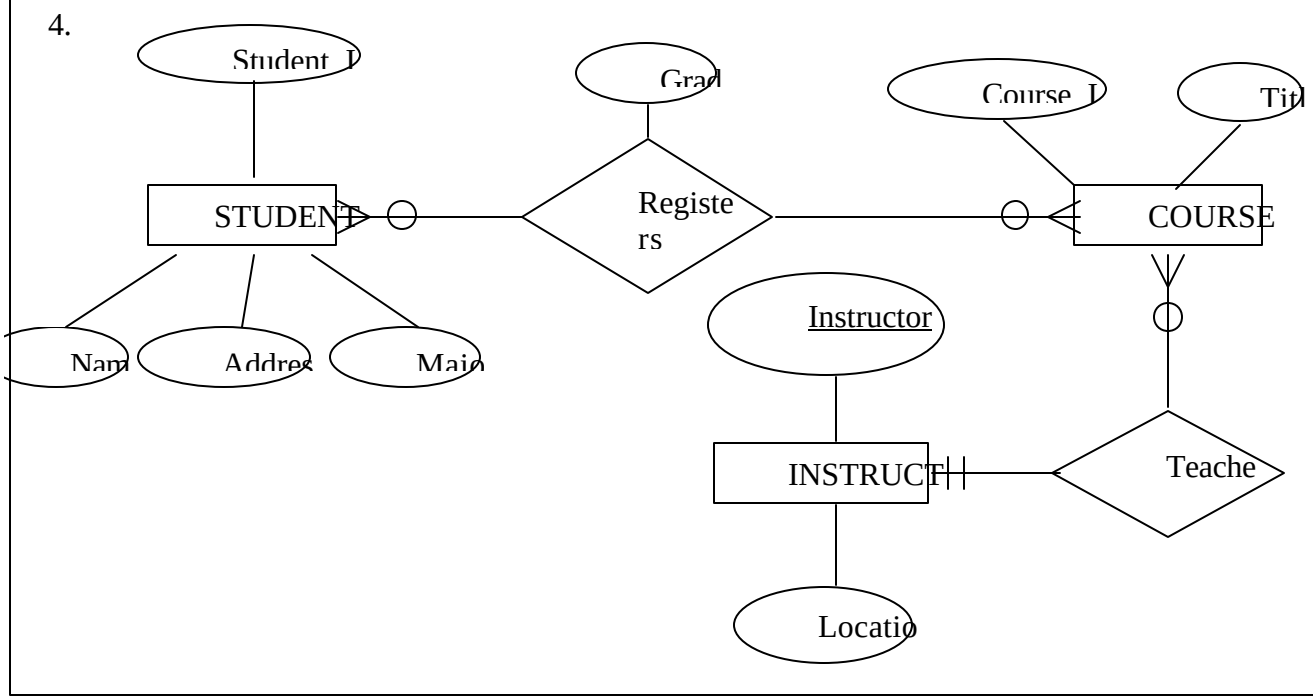

Employee
Assign
Proje
ct ID
CHEMIST
ASSIGNE
D
PROJECT
Nam
Phon
EQUIPMEN
T
Start_Da
Serial_
Cost

d.

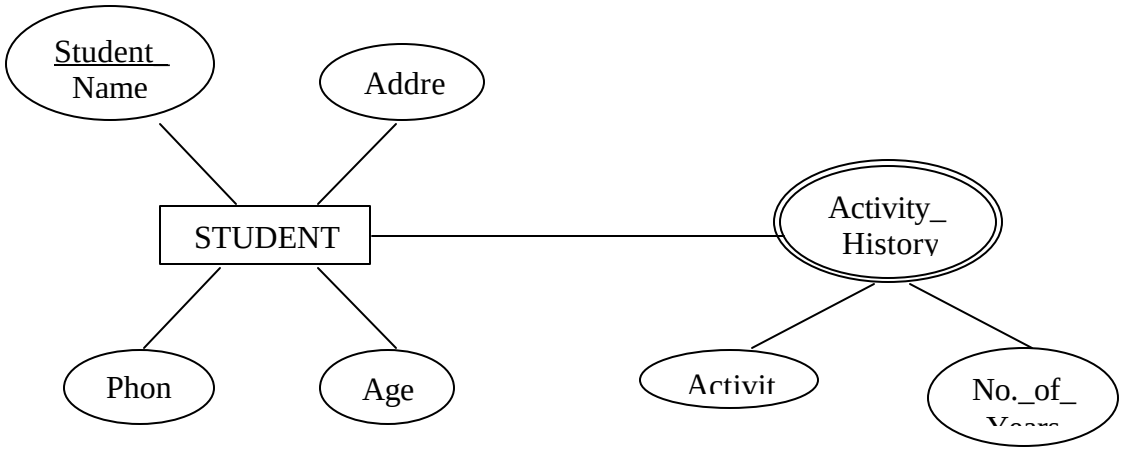

Course_I
Section
COURSE
Has_
scheduled
SECTION
Nam
Units
Semeste
r
Semest
Year


Physician_
Patient_I
Admit
PHYSICIAN
PATIENT
Treat
Special
Patien
Treatme
Date
Time
Result

4.


Student_I
Grad
Course_I
Titl
STUDENT
Registe
rs
COURSE
Instructor
Nam
Addres
Maio
INSTRUCT
Teache
Locatio

5.



Notes:

1. Assume that Student_Name is the identifier.
2. Age cannot be calculated without date-of-birth.

Normalization Exercise One

1. For the following table, draw a dependency diagram. Indicate any partial or transitive dependencies on your diagram. Then, decompose this table into tables in third normal form. The primary key is underlined.

Invoice No.	Product No.	Sale Date	Prod Desc	Vend Code	Vend Name	Qty Sold	Prod Price
211347	AAE33	3/25/98	Drill Bit	211	Black&Decker	3	2.95
211347	AR456	3/25/98	Drill	211	Black&Decker	1	29.99
211348	AR444	4/5/99	Screw	133	Acme	55	.17

2. For the following table, draw a dependency diagram. Indicate any partial or transitive dependencies on your diagram. Then, decompose this table into tables in third normal form.

StuID	StuName	StuMajor	DeptID	DeptName	DeptPh	College	AdvName	AdvOffice	AdvBldg	AdvPh	StuGPA	StuHrs	StuClass
111	Smith	Biology	BIOL	Biology	2363333	A&S	Baker	LS113	Biol	2362222	3.01	65	Junior
123	Jones	Biology	BIOL	Biology	2363333	A&S	Williams	LS114	Biol	2362333	3.33	90	Junior
122	Howard	Acct	ACCT	Accounting	2364444	Bus	Boes	BA333	BA	2363232	3.11	20	Freshman

Notes:

Normalization Exercise Two

1. Given the data below, what would the primary key be? Using that primary key, indicate the functional dependencies and their types. Then create a set of tables in 3NF.

ItemID	ItemDesc	BldgRoom	Bldgcode	Bldgname	BldgManager
2111	HP Deskjet 660C	325	Eng	Engineering	Baker
2111	HP Deskjet 660C	333	BA	Business Admin	Smith
3222	Epson 440	122	BA	Business Admin	Smith

2. Given the data below, what would the primary key be? Indicate the functional dependencies and their types. Then create a set of tables in 3NF.

Attribute Name	Sample Value	Sample Value	Sample Value
emp-code	1003	1110	23453
lastName	Wilson	Baker	Johnson
education	HS, BBA. MBA	HS	HS, BS
deptcode	MKTG	OPS	ENG
department	Marketing	Operations	Engineering
deptMgr	Smith	Calhoun	Roberts
jobClass	23	11	33
title	sales agent	assembler	Engineer
dependents	Gerald, Mary, Joan	Sally	
birthDate	5/5/60	4/5/55	3/3/62
hireDate	1/2/90	5/5/95	1/1/85
training	level 1, level 2	level 1	Level 1, level 2
salary	$32,500	$28,000	$48,000

Notes:

Practice Queries

Use the Entertainment Agency database to answer these. Some of the queries may return empty sets unless you add data to some of the tables for testing purposes.

1. Display the entertainers, start and end date of their contracts, and the contract price.
2. List all the entertainers who played engagements for both customers Bonnicksen and Rosales.
3. List all entertainers that have never been booked.
4. List agents that have never booked an entertainer.
5. List all entertainers and any engagements they have booked.
6. Display each customer and the date of the last booking they made.
7. List customer who have booked entertainers who play country or country rock.

SQL Practice Solutions

1. Display the entertainers, start and end date of their contracts, and the contract price.

```
SELECT Entertainers.EntStageName, Engagements.StartDate,
Engagements.EndDate, Engagements.ContractPrice
FROM Entertainers INNER JOIN Engagements ON
Entertainers.EntertainerID = Engagements.EntertainerID;
```

2. List all the entertainers who played engagements for both customers Bonnicksen and Rosales.

```
SELECT EntBonnicksen.EntStageName
FROM [SELECT DISTINCT Entertainers.EntertainerID,
Entertainers.EntStageName
FROM (Entertainers INNER JOIN Engagements ON
Entertainers.EntertainerID = Engagements.EntertainerID)
INNER JOIN Customers ON Customers.CustomerID =
Engagements.CustomerID
WHERE Customers.CustLastName = 'Bonnicksen']. AS
EntBonnicksen INNER JOIN [SELECT DISTINCT
Entertainers.EntertainerID, Entertainers.EntStageName
FROM (Entertainers INNER JOIN Engagements ON
Entertainers.EntertainerID = Engagements.EntertainerID)
INNER JOIN Customers ON Customers.CustomerID =
Engagements.CustomerID
WHERE Customers.CustLastName = 'Rosales']. AS
EntRosales ON EntBonnicksen.EntertainerID =
EntRosales.EntertainerID;
```

3. List all entertainers that have never been booked.

```
SELECT Entertainers.EntertainerID,
Entertainers.EntStageName
FROM Entertainers LEFT JOIN Engagements ON
Entertainers.EntertainerID = Engagements.EntertainerID
WHERE Engagements.EngagementNumber Is Null;
```

4. List agents that have never booked an entertainer.

```
SELECT Agents.AgentID, Agents.AgtFirstName,
Agents.AgtLastName
FROM Agents LEFT JOIN Engagements ON
Agents.AgentID = Engagements.AgentID
WHERE Engagements.EngagementNumber IS NULL;
```

5. List all entertainers and any engagements they have booked.

```
SELECT Entertainers.EntertainerID,
Entertainers.EntStageName,
Engagements.EngagementNumber, Engagements.StartDate,
Engagements.CustomerID
FROM Entertainers LEFT JOIN Engagements ON
Entertainers.EntertainerID = Engagements.EntertainerID;
```

6. Display each customer and the date of the last booking they made.

```
SELECT Customers.CustFirstName,
Customers.CustLastName, (Select Max(StartDate) FROM
Engagements WHERE Engagements.CustomerID =
Customers.CustomerID) AS LastBooking
FROM Customers;
```

7. List customer who have booked entertainers who play country or country rock.

```
SELECT Customers.CustomerID, Customers.CustFirstName,
Customers.CustLastName
FROM Customers
WHERE Customers.CustomerID IN
(SELECT Engagements.CustomerID
FROM ((Musical_Styles
INNER JOIN Entertainer_Styles
ON Musical_Styles.StyleID = Entertainer_Styles.StyleID)
INNER JOIN Entertainers
ON Entertainers.EntertainerID =
Entertainer_Styles.EntertainerID)
INNER JOIN Engagements ON Entertainers.EntertainerID =
Engagements.EntertainerID
WHERE Musical_Styles.StyleName='Country' Or
Musical_Styles.StyleName='Country Rock');
```

Notes:

Review

1. Describe some of the tasks involved in the development (analysis, design, and implementation) of an information system. With regards to these tasks, what are the differences between a data-oriented and a process-oriented development approach?
2. Describe a sample transaction and explain why it should be viewed as a transaction.
3. What does it mean to say that a transaction should be atomic?
4. What does it mean to say that a transaction should be serializable?
5. Why are transactions defined during application design, rather than when the database tables are defined?
6. What is the role of a transaction log in helping to maintain database integrity? Describe an example of how it is used.
7. Describe through the use of an example how data integrity might be harmed in a multi-user database environment if concurrency mechanisms were not implemented.
8. Explain how a lock helps maintain concurrency control.
9. What is the difference between a read (shared) lock and an exclusive (write) lock?
10. What are the tradeoffs between using locks of differing levels of granularity, such as a database-level lock versus a row-level (record-level) lock?
11. How can deadlocks occur?
12. When compared with using a centralized computer connected to "dumb" terminals, what are the advantages of client/server computing?
13. When compared to using a "file server," what are the advantages of a client/server architecture that implements a database server?
14. What computing task(s) are typically performed by the client computer? What tasks are typically performed by the server?
15. Describe an example thin client implementation. Describe an example fat client implementation.
16. Describe how the client, web server, and database server interact to provide dynamic web content.
17. How does a web server know that a file may contain executable "script" code?
18. What are some examples of database middleware, and what function does it serve?
19. Why is data within a data warehouse often "denormalized?"
20. Compare and contrast an operational database with a data warehouse.
21. Why is a client/server architecture so useful in a DSS/data warehouse environment?
22. Why would an organization use data marts instead of a data warehouse?
23. Why do operational databases typically store transient data instead of periodic data?

Notes:

SQL Statements to Create Tables (First SQL Assignment)

To create the Parts Table:

```
Create Table Parts
(PartNum text(6) constraint PartsPrimary primary key,
PartName text(20),
Color text(6),
Weight integer,
City text(15));
```

To create Projects Table:

```
Create table Projects
(ProjectNum text(4) constraint ProjConstraint primary key,
ProjectName text(10),
City text(15));
```

To create Suppliers Table:

```
Create table Suppliers
(SupplierNum text (5) constraint supplierconstraint primary key,
SupplierName text (20),
Status integer,
city text(15));
```

To create Shipments Table (this will not create relationships between Shipments table and the others):

```
create table Shipments
(SupplierNum text(5),
PartNum text(6),
ProjectNum text(4),
Qty integer);
```

To create Shipments table and create relationships with other tables:

```
create table Shipments
(SupplierNum text(5),
PartNum text(6),
ProjectNum text(4),
Qty integer,
constraint ShipmentKey primary key (SupplierNum, PartNum,
ProjectNum),
constraint ShipmentSupplier foreign key (SupplierNum)
references Suppliers,
constraint ShipmentParts foreign key (PartNum) references
Parts,
constraint ShipmentProjects foreign key (ProjectNum) refer-
ences Projects);
```

SQL Assignment Two Solutions

1. Show all products that are have UnitsInStock that are below the ReorderLevel.

```
SELECT productName
FROM Products
where unitsInStock < Reorderlevel;
```

2. What are the last names of all employees born before Jan 1, 1960?

```
SELECT LastName
from Employees
where BirthDate < #1/1/60#;
```

3. Show the company names of all customers that have ordered the product “Aniseed Syrup.”

```
SELECT Customers.CompanyName
from Customers, Orders, [Order Details], Products
where (customers.CustomerID = Orders.CustomerID)
and (Orders.OrderID = [Order Details].OrderID)
and ([Order Details].ProductID = Products.ProductID)
and (Products.ProductName = ‘Aniseed Syrup’);
```

4. Show the company names of all suppliers who have discontinued products.

```
SELECT Suppliers.CompanyName
from Suppliers, Products
where (Suppliers.SupplierID = Products.SupplierID) and
(Products.Discontinued = true);
```

5. Write a query that shows how many orders are associated with each customer ID. Note that the customerID field is replaced with the Customer’s name. Why is that?

```
SELECT customers.customerID, count(orders.orderID) as
[Number of Orders]
from Customers, Orders
where orders.CustomerID = Customers.CustomerID
GROUP BY Customers.CustomerID;
```

6. Total inventory (dollar value) on hand by category.

```
SELECT products.categoryID,
sum(Products.UnitPrice * Products.UnitsInStock) as
[Inventory Value]
from Products
group by products.CategoryID;
```

7. Show the total dollar value of all the orders we have for each customer (one dollar amount for each customer). Order the list in descending order.

```
SELECT Orders.CustomerID,
format (sum(([order details].unitPrice * [order
details].quantity) * (1 - [Order Details].discount)),
“###,###.00”) as Total
from Orders, [Order Details]
where orders.orderID = [Order Details].OrderID
group by orders.CustomerID
order by sum(([order details].unitPrice * [order
details].quantity) * (1 - [Order Details].discount));
```

SQL Assignment Three Solutions

1. Get supplier names for all orders.

   SELECT suppliers.CompanyName, orders.orderID
   FROM Orders, Suppliers, Products, [Order Details]
   where Suppliers.SupplierID = Products.SupplierID and
   Products.ProductID = [Order Details].ProductID and
   Orders.OrderID = [Order Details].OrderID;

2. Get order ID and product name for all orders supplied by "Tokyo Traders."

   SELECT Suppliers.CompanyName, [order
   details].orderID, products.productName
   FROM (Suppliers INNER JOIN Products ON
   Suppliers.SupplierID = Products.SupplierID)
   INNER JOIN [Order Details] ON Products.ProductID =
   [Order Details].ProductID
   where suppliers.CompanyName = "Tokyo Traders";

3. Get order ID, product name, and dollar value of the amount of that product ordered for each orders supplied by "Tokyo Traders."

   SELECT Suppliers.CompanyName, [order
   details].orderID, products.productName,
   (([order details].UnitPrice*[order details].Quantity) * (1-
   [order details].Discount)) as dollarValue
   FROM (Suppliers INNER JOIN Products ON
   Suppliers.SupplierID = Products.SupplierID)
   INNER JOIN [Order Details] ON Products.ProductID =
   [Order Details].ProductID
   where suppliers.CompanyName = "Tokyo Traders";

4. Get the shipper name and supplier name for all orders shipped by speedy express.

   SELECT Suppliers.CompanyName,
   Shippers.CompanyName
   FROM Shippers INNER JOIN (Orders INNER JOIN
   ((Suppliers INNER JOIN Products ON
   Suppliers.SupplierID = Products.SupplierID)
   INNER JOIN [Order Details] ON Products.ProductID =
   [Order Details].ProductID)
   ON Orders.OrderID = [Order Details].OrderID) ON
   Shippers.ShipperID = Orders.ShipVia
   where shippers.CompanyName = "Speedy Express";

5. Count the number of orders shipped by each shipper.

   SELECT Shippers.CompanyName, count(orders.orderid)
   FROM Shippers INNER JOIN Orders ON
   Shippers.ShipperID = Orders.ShipVia
   group by shippers.companyName;

7. Display the category name of all products that have sales greater than $10,000 for the year 1997. Hint – this can be done using the "having" clause and using a saved, named query as part of another query.

   This query can be interpreted at least two ways. One way is to include the product name and category name for all products that have sold over $10,000 in the given year. The SQL below delivers that result.

   SELECT DISTINCTROW Categories.CategoryName,
   Products.ProductName,
   Sum(CCur([Order Details].[UnitPrice]*[Quantity]*(1-
   [Discount])/100)*100) AS ProductSales
   FROM (Categories INNER JOIN Products ON
   Categories.CategoryID = Products.CategoryID)
   INNER JOIN (Orders INNER JOIN [Order Details] ON
   Orders.OrderID = [Order Details].OrderID) ON
   Products.ProductID = [Order Details].ProductID
   WHERE (((Orders.ShippedDate) Between #1/1/97#
   And #12/31/97#))
   GROUP BY Categories.CategoryName,
   Products.ProductName
   Having Sum(CCur([Order
   Details].[UnitPrice]*[Quantity]*(1-[Discount]))) > 10000;

   The other way to interpret this is to assume that you want only the categories for which there were sales over $10,000 for the given year. In this case, you group by category name only, and do not include the product name.

   The query below works for earlier versions of the database, but the saved query [Product Sales for 1997] in the Access 2000 version creates totals by quarter, rather than by year. [Product Sales for 1997] can be modified to give the same answer as the above, however.

   SELECT DISTINCTROW [Product Sales for
   1997].CategoryName, Sum([Product Sales for
   1997].ProductSales) AS CategorySales
   FROM [Product Sales for 1997]
   GROUP BY [Product Sales for 1997].CategoryName;

SQL Assignment Four Solutions

1. Display each product name and the latest date the product was ordered. (Hint - use the max function on the orderdate column.)

   SELECT [order details].productid, max(orders.orderdate)
   FROM Orders INNER JOIN [Order Details] ON
   Orders.OrderID = [Order Details].OrderID
   group by [order details].productid;

2. Find the company name of all customers who have ordered Aniseed Syrup and Outback Lager on the same order. Include the orderid in your results. First, solve this by using three separate queries.

Query QAS

SELECT [Customers].[CompanyName], [orders].[orderid]
FROM Customers, Orders, [Order Details], Products
WHERE
([customers].[CustomerID]=[Orders].[CustomerID]) And
([Orders].[OrderID]=[Order Details].[OrderID]) And ([Order
Details].[ProductID]=[Products].[ProductID]) And
([Products].[ProductName]='Aniseed Syrup');

Query QOL

SELECT [Customers].[CompanyName], [orders].[orderid]
FROM Customers, Orders, [Order Details], Products
WHERE
([customers].[CustomerID]=[Orders].[CustomerID]) And
([Orders].[OrderID]=[Order Details].[OrderID]) And ([Order
Details].[ProductID]=[Products].[ProductID]) And
([Products].[ProductName]='Outback Lager');

```
SELECT [QAS].[CompanyName], QAS.orderid
FROM QAS inner join QOL on QAS.orderid = QOL.orderid
```

Nested solution:

```
SELECT distinct Customers.CompanyName
from (((Customers inner join Orders on
customers.CustomerID = Orders.CustomerID) inner join
[Order Details] on Orders.OrderID = [Order Details].OrderID)
inner join Products on [Order Details].ProductID =
Products.ProductID)
where [order details].orderid in
(select [order details].orderid
from ((Orders inner join [Order Details] on Orders.OrderID =
[Order Details].OrderID)
inner join Products on [Order Details].ProductID =
Products.ProductID)
where (Products.ProductName = 'Aniseed Syrup'))
and [order details].orderid in
(select [order details].orderid
from ((Orders inner join [Order Details] on Orders.OrderID =
[Order Details].OrderID)
inner join Products on [Order Details].ProductID =
Products.ProductID)
where (Products.ProductName = 'Outback Lager')) ;
```

3. Find all the product name of all products that have never been ordered. You will have to add a new product to the product table to test this. (Hint - you can use the NOT IN clause and a nested subquery).

   ```
   SELECT [products].productname
   from products
   where products.productid not in
   (select [order details].productid
   from [order details]);
   ```

4. Display the total number of times each product has been ordered, along with the product name.

   ```
   SELECT productname, count([Order Details].ProductID)
   FROM Products INNER JOIN [Order Details] ON
   Products.ProductID = [Order Details].ProductID
   group by productname;
   ```

5. Display the OrderID and Customer.CompanyName for all orders that total over $5,000 (ignore freight and discount).

   ```
   SELECT customers.companyname, orders.orderid,
   sum(unitprice * quantity)
   from customers inner join (orders inner join [order details]
   on [order details].orderid = orders.orderid) on
   orders.customerid = customers.customerid
   group by customers.companyname, orders.orderid
   having (sum(unitprice * quantity)) > 5000
   ```

Notes:

1.1. A Sample System

Perhaps it is best to begin by giving an example of such a system. A large bank may have one thousand teller terminals (several have 20,000 tellers but at present no single system supports such a large network). For each teller, there is a record describing the teller's cash drawer and for each branch there is a record describing the cash position of that branch (bank general ledger), It is likely to have several million demand deposit accounts (say 10,000,000 accounts). Associated with each account is a master record giving the account owner, the account balance, and a list of recent deposits and withdrawals applicable to this account. This database occupies over 10,000,000,000 bytes and must all be on-line at all times,

The database is manipulated with application dependent transactions, which were written for this application when it was installed. There are many transactions defined on this database to query it and update it. A particular user is allowed to invoke a subset of these transactions. Invoking a transaction consists of typing a message and pushing a button. The teller terminal appends the transaction identity, teller identity and terminal identity to the message and transmits it to the central data manager. The data communication manager receives the message and translates it to some canonical form.

It then passes the message to the transaction manager, which validates the teller's authorization to invoke the specified transaction and then allocates and dispatches an instance of the transaction. The transaction processes the message, generates a response, and terminates. Data communications delivers the message to the teller.

Perhaps the most common transaction is in this environment is the DEBIT_CREDIT transaction which takes in a message from any teller, debits or credits the appropriate account (after running some validity checks), adjusts the teller cash drawer and branch balance, and then sends a response message to the teller. The transaction flow is:

```
DEBIT_CREDIT:
    BEGIN_TRANSACTION;
    GET MESSAGE;
    EXTRACT ACCOUT_NUMBER, DELTA, TELLER, BRANCH
        FROM MESSAGE;
    FIND ACCOUNT(ACCOUT_NUMBER) IN DATA BASE;
    IF NOT_FOUND | ACCOUNT_BALANCE + DELTA < 0 THEN
        PUT NEGATIVE RESPONSE;
    ELSE DO;
        ACCOUNT_BALANCE = ACCOUNT_BALANCE + DELTA;
        POST HISTORY RECORD ON ACCOUNT (DELTA);
        CASH_DRAWER(TELLER) = CASH_DRAWER(TELLER) + DELTA;
        BRANCH_BALANCE(BRANCH) = BRANCH_BALANCE(BRANCH) + DELTA;
        PUT MESSAGE ('NEW BALANCE =' ACCOUNT_BALANCE);
        END;
    COMMIT;
```

At peak periods the system runs about thirty transactions per second with a response time of two seconds.

The DEBIT_CREDIT transaction is very "small". There is another class of transactions that behave rather differently. For example, once a month a transaction is run which produces a summary statement for each account. This transaction might be described by:

```
MONTHLY_STATEMENT:
  ANSWER :: = SELECT *
        FROM ACCOUNT, HISTORY
        WHERE ACCOUNT. ACCOUNT_NUMBER = HISTORY. ACCOUNT_NUMBER
        AND HISTORY_DATE > LAST_REPORT
        GROUPED BY ACCOUNT. ACCOUNT_NUMBER,
        ASCENDING BY ACCOUNT. ACCOUNT_ADDRESS;
```

That is, collect all recent history records for each account and place them clustered with the account record into an answer file. The answers appear sorted by mailing address.

If each account has about fifteen transactions against it per month then this transaction will read 160,000,000 records and write a similar number of records. A naive implementation of this transaction will take 80 days to execute (50 milliseconds per disk seek implies two million seeks per day.) However, the system must run this transaction once a month and it must complete within a few hours.

There is a broad spread of transactions between these two types. Two particularly interesting types of transactions are conversational transactions that carry on a dialogue with the user and distributed transactions that access data or terminals at several nodes of a computer network,

Systems of 10,000 terminals or 100,000,000,000 bytes of on-line data or 150 transactions per second are generally considered to be the limit of present technology (software and hardware).

1.2. Relationship To Operating System

If one tries to implement such an application on top of a general-purpose operating system it quickly becomes clear that many necessary functions are absent from the operating system. Historically, two approaches have been taken to this problem:

- Write a new, simpler and "vastly superior" operating system.
- Extend the basic operating system to have the desired function.

The first approach was very popular in the mid-sixties and is having a renaissance with the advent of minicomputers. The initial cost of a data management system is so low that almost any large customer can justify "rolling his own". The performance of such tailored systems is often ten times better than one based on a general purpose system. One must trade this off against the problems of maintaining the system as it grows to meet new needs and applications. Group's that followed this path now find themselves maintaining a rather large operating system, which must be modified to support new devices (faster disks, tape archives,...) and new protocols (e. g. networks and displays.) Gradually, these systems have grown to include all the functions of a general-purpose operating system. Perhaps the most successful approach to this has been to implement a hypervisor that runs both the data management operating system and some non-standard operating system. The "standard" operating system runs when the data manager is idle. The hypervisor is simply an interrupt handler which dispatches one or another system.

The second approach of extending the basic operating system is plagued with a different set of difficulties. The principal problem is the performance penalty of a general-purpose operating system. Very few systems are designed to deal with very large files, or with networks of thousands of nodes. To take a specific example, consider the process structure of a general-purpose system: The allocation and deallocation of a process should be very fast (500 instructions for the pair is expensive) because we want to do it 100 times per second. The storage occupied by the process descriptor should also be small (less than 1000 bytes.) Lastly, preemptive scheduling of processes makes no sense since they are not CPU bound (they do a lot of I/O). A typical system uses 16,000 bytes to represent a process and requires 200,000 instructions to allocate and deallocate this structure (systems without protection do it cheaper.) Another problem is that the general-purpose systems have been designed for batch and time-sharing operation. They have not paid sufficient attention to issues such as continuous operation: keeping the system up for weeks at a time and gracefully degrading in case of some hardware or software error.

1.3. General Structure of Data Management Systems

These notes try to discuss issues that are independent of which operating system strategy is adopted. No matter how the system is structured, there are certain problems it must solve. The general structure common to several data management systems is presented. Then two particular problems within the transaction management component are discussed in detail: concurrency control (locking) and system reliability (recovery).

This presentation decomposes the system into four major components:

- Dictionary: the central repository of the description and definition of all persistent system objects.
- Data Communications: manages teleprocessing lines and message traffic.
- Data Base manager: manages the information stored in the system.
- Transaction Management: manages system resources and system services such as locking and recovery.

Each of these components calls one another and in turn depends on the basic operating system for services.

2. Dictionary

2.1. What It Is

The description of the system, the databases, the transactions, the telecommunications network, and of the users are all collected in the dictionary. This repository:

- Defines the attributes of objects such as databases and terminals.
- Cross-references these objects.
- Records natural language (e. g. German) descriptions of the meaning and use of objects.

When the system arrives, the dictionary contains only a very few definitions of transactions (usually utilities), defines a few distinguished users (operator, data base administrator,...), and defines a few special terminals (master console). The system administrator proceeds to define new terminals, transactions, users, and databases. (The system administrator function includes data base administration (DBA) and data communications (network) administration (DCA). Also, the system administrator may modify existing definitions to match the actual system or to reflect changes. This addition and modification process is treated as an editing operation.

For example, one defines a new user by entering the "define" transaction and selecting USER from the menu of definable types. This causes a form to be displayed, which has a field for each attribute of a user. The definer fills in this form and submits it to the dictionary. If the form is incorrectly filled out, it is redisplayed and the definer corrects it. Redefinition follows a similar pattern; the current form is displayed, edited and then submitted. (There is also a non-interactive interface to the dictionary for programs rather than people.)

All changes are validated by the dictionary for syntactic and semantic correctness. The ability to establish the correctness of a definition is similar to ability of a compiler to detect the correctness of a program. That is, many semantic errors go undetected. These errors are a significant problem.

Aside from validating and storing definitions, the dictionary provides a query facility which answers questions such as: "Which transactions use record type A of file B?" or, "What are the attributes of terminal 34261".

The dictionary performs one further service, that of compiling the definitions into a "machine readable" form more directly usable by the other system components. For example, a

terminal definition is converted from a variable length character string to a fixed format "descriptor" giving the terminal attributes in non-symbolic form.

The dictionary is a database along with a set of transactions to manipulate this database. Some systems integrate the dictionary with the data management system so that the data definition and data manipulation interface are homogeneous. This has the virtue of sharing large bodies of code and of providing a uniform interface to the user. Ingress and System R are examples of such systems.

Historically, the argument against using the database for the dictionary has been performance. There is very high read traffic on the dictionary during the normal operation of the system. A user logon requires examining the definitions of the user, his terminal, his category, and of the session that his logon establishes. The invocation of a transaction requires examining his authorization, the transaction, and the transaction descriptor (to build the transaction.) In turn the transaction definition may reference databases and queues which may in turn reference files, records and fields. The performance of these accesses is critical because they appear in the processing of each transaction. These performance constraints combined with the fact that the accesses are predominantly read-only have caused most systems to special-case the dictionary. The dictionary definitions and their compiled descriptors are stored by the data base management component. The dictionary-compiled descriptors are stored on a special device and a cache of them is maintained in high-speed storage on an LRU (Least Recently Used) basis. This mechanism generally uses a coarse granularity of locks and because operations are read only it keeps no log. Updates to the descriptors are made periodically while the system is quiesced.

The descriptors in the dictionary are persistent. During operation, many other short-lived descriptors are created for short-lived objects such as cursors, processes, and messages. Many of these descriptors are also kept in the descriptor cache.

The dictionary is the natural extension of the catalog or file system present in operating systems. The dictionary simply attaches more semantics to the objects it stores and more powerful operators on these objects.

Readers familiar with the literature may find a striking similarity between the dictionary and the notion of conceptual schema, which is "a model of the enterprise". The dictionary is the conceptual schema without its artificial intelligence aspects. In time the dictionary component will evolve in the direction suggested by papers on the conceptual schema.

3. Data Management

The Data management component stores and retrieves sets of records. It implements the objects: network, set of records, cursor, record, field, and view.

3.1. Records And Fields

A record type is a sequence of field types, and a record instance is a corresponding sequence of field instances. Record types and instances are persistent objects. Record instances are the atomic units of insertion and retrieval. Fields are sub-objects of records and are the atomic units of update. Fields have the attributes of atoms (e. g. FIXED(31)or CHAR(*)) and field instances have atomic values (e. g. "3" or "BUTTERFLY"). Each record instance has a unique name called a record identifier (RID).

A field type constrains the type and values of instances of a field and defines the representation of such instances. The record type specifies what fields occur in instances of that record type.

A typical record might have ten fields and occupy 256 bytes although records often have hundreds of fields (e. g. a record giving statistics on a census tract has over 600 fields), and may be very large (several thousand bytes). A very simple record (nine fields and about eighty characters) might be described by:

```
DECLARE 1 PHONE_BOOK_RECORD,
    2 PERSON_NAME CHAR(*),
    2 ADDRESS,
        3 STREET_NUMBER CHAR(*),
        3 STREET_NAME CHAR(*),
        3 CITY CHAR(*),
        3 STATE CHAR(*),
        3 ZIP_CODE CHAR(5).
    2 PHONE_NUMBER,
        3 AREA_CODE CHAR(3),
        3 PREFIX CHAR(3),
        3 STATION CHAR(4);
```

The operators on records include INSERT, DELETE, FETCH, and UPDATE. Records can be CONNECTED to and DISCONNECTED from membership in a set (see below). These operators actually apply to cursors, which in turn point to records.

The notions of record and field correspond very closely to the notions of record and element in COBOL or structure and field in PL/l. Records are variously called entities, segments, tuples, and rows by different subcultures. Most systems have similar notions of records although they may or may not support variable length fields, optional fields (nulls), or repeated fields.

3.2. Sets

A set is a collection of records. This collection is represented by and implemented as an "access path" that runs through the collection of records. Sets perform the functions of :

- Relating the records of the set.
- In some instances directing the physical clustering of records in physical storage.

A record instance may occur in many different sets but it may occur at most once in a particular set.

There are three set types of interest:

- **Sequential set**: the records in the set form a single sequence. The records in the set are ordered either by order of arrival (entry sequenced (ES)), by cursor position at insert (CS), or are ordered (ascending or descending) by some subset of field values (key sequenced (KS)). Sequential sets model indexed-sequential files (ISAM, VSAM).

- **Partitioned set:** The records in the set form a sequence of disjoint groups of sequential sets. Cursor operators allow one to point at a particular group. Thereafter the sequential set operators are used to navigate within the group. The set is thus major ordered by hash and minor ordered (ES, CS or KS) within a group. Hashed files in which each group forms a hash bucket are modeled by partitioned sets,
- **Parent-child set**: The records of the set are organized into a two-level hierarchy. Each record instance is either a parent or a child (but not both). Each child has a unique parent and no children. Each parent has a (possibly null) list of children. Using parent-child sets one can build networks and hierarchies. Positional operators on parent-child sets include the operators to locate parents, as well as operations to navigate on the sequential set of children of a parent. The CONNECT and DISCONNECT operators explicitly relate a child to a parent, One obtains implicit connect and disconnect by asserting that records inserted in one set should also be connected to another. (Similar rules apply for connect, delete and update.) Parent-child sets can be used to support hierarchical and network data models.

A partitioned set is a degenerate form of a parent-child set (the partitions have no parents), and a sequential set is a degenerate form of a partitioned set (there is only one partition.) In this discussion care has been taken to define the operators so that they also subset. This has the consequence that if the program uses the simplest model it will be able to run on any data and also allows for subset implementations on small computers.

Inserting a record in one set map trigger its connection to several other sets. If set “I” is an index for set “F” then an insert, delete and update of a record in “F” may trigger a corresponding insert, delete, or update in set “I”. In order to support this, data manager must know:

- That insertion, update or deletion of a record causes its connection to, movement in, or disconnection from other sets.
- Where to insert the new record in the new set:
- For sequential sets, the ordering must be either key sequenced or entry sequenced.
- For partitioned sets, data manager must know the partitioning rule and know that the partitions are entry sequenced or key sequenced.
- For parent-child sets, the data manager must know that certain record types are parents and that others are children. Further, in the case of children, data manager must be able to deduce the parent of the child.

We will often use the term “file” as a synonym for set.

3.3. Cursors.

A cursor is “opened” on a specific set and thereafter points exclusively to records in that set. After a cursor is opened it may be moved, copied, or closed. While a cursor is opened it may be used to manipulate the record it addresses.

Records are addressed by cursors. Cursors serve the functions of:

- Pointing at a record.
- Enumerating all records in a set.
- Translating between the stored record format and the format visible to the cursor user. A simple instance of this might be a cursor that hides some fields of a record. This aspect will be discussed with the notion of view.

A cursor is an ephemeral object that is created from a descriptor when a transaction is initiated or during transaction execution by an explicit OPEN_CURSOR command. Also one may COPY_CURSOR a cursor to make another instance of the cursor with independent positioning. A cursor is opened on a specific set (which thereby defines the enumeration order (next) of the cursor.) A cursor is destroyed by the CLOSE_CURSOR command.

3.3.2. Operations on Cursors

Operators on cursors include
FETCH (<cursor> [, <position>]) [HOLD]
RETURNS(<record>)

Which retrieves the record pointed at by the named cursor. The record is moved to the specified target. If the position is specified the cursor is first positioned. If HOLD is specified the record is locked for update (exclusive), otherwise the record is locked in share mode.

INSERT (<cursor>[, <position>], <record>)

Inserts the specified record into the set specified by cursor. If the set is key sequenced or entry sequenced then the cursor is moved to the correct position before the record is inserted, otherwise the record is inserted at (after) the current position of the cursor in the set. If the record type automatically appears in other sets, it also inserted in them.

UPDATE (<cursor> [, <position>],<new_record>)

If position is specified the cursor is first positioned. The new record is then inserted in the set at the cursor position replacing the record pointed at by the cursor. If the set is sequenced by the updated fields, this may cause the record and cursor to move in the set.

DELETE (<cursor> [, <position>])

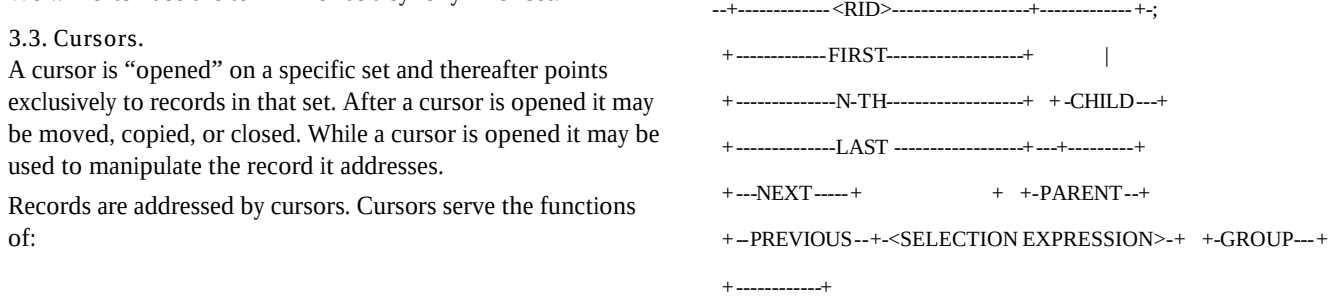
Deletes the record pointed at by the cursor after optionally repositioning the cursor.

MOVE_CURSOR (<cursor>, <position>) HOLD

Repositions the cursor in the set.

3.3.3 Cursor Positioning

A cursor is opened to traverse a particular set. Positioning expressions have the syntax:



where RID, FIRST , N-th, and LAST specify specific record occurrences while the other options specify the address relative to the current cursor position. It is also possible to set a cursor from another cursor.

The selection expression may be any Boolean expression valid for all record types in the set. The selection expression includes the relational operators: =, !=, >, <, <=, >=, and for character strings a "matches-prefix" operator sometimes called generic key. If next or previous is specified, the set must be searched sequentially because the current position is relevant. Otherwise, the search can employ hashing or indices to locate the record. The selection expression search may be performed via an index, which maps field values into RIDs.

Examples of commands are

```
FETCH   (CURSOR1, NEXT NAME='SMITH') HOLD
RETURNS (POP);

DELETE  (CURSOR1, NEXT NAME='JOE' CHILD);

INSERT  (CURSOR1, , NEWCHILD);
```

For partitioned sets one may point the cursor at a specific partition by qualifying these operators by adding the modifier GROUP. A cursor on a parent-child (or partitioned) set points to both a parent record and a child record (or group and child within group). Cursors on such sets have two components: the parent or group cursor and the child cursor. Moving the parent curser, positions the child cursor to the first record in the group or under the parent. For parent-child sets one qualifies the position operator with the modifier NEXT_PARENT in order to locate the first child of the next parent or with the modifier WITHIN PARENT if the search is to be restricted to children of the current parent or group. Otherwise positional operators operate on children of the current parent.

There are rather obscure issues associated with cursor positioning.

The following is a good set of rules:

- A cursor can have the following positions:
- Null.
- Before the first record.
- At a record.
- Between two records.
- After the last record.
- If the cursor points at a null set, then it is null. If the cursor points to a non-null set then it is always non-null.
- Initially the cursor is before the first record unless the OPEN_CURSOR specifies a position.
- An INSERT operation leaves the cursor pointing at the new record.
- A DELETE operation leaves the cursor between the two adjacent records , or at the top if there is no previous record, or at the bottom if there is a previous but no successor record.
- A UPDATE operation leaves the cursor pointing at the updated record.
- If an operation fails the cursor is not altered.

3.4. Various Data Models

Data models differ in their notion of set.

3.4.1. Relational Data Model

The relational model restricts itself to homogeneous (only one record type) sequential sets. The virtue of this approach is its simplicity and the ability to define operators that "distribute" over the set, applying uniformly to each record of the set. Since much of data processing involves repetitive operations on large volumes of data, this distributive property provides a concise language to express such algorithms. There is a strong analogy here with APL that uses the simple data structure of array and therefore is able to define powerful operators that work for all arrays. APL programs are very short and much of the control structure of the program is hidden inside of the operators.

To give an example of this, a "relational" program to find all overdue accounts in an invoice file might be:

```
SELECT ACCOUNT_NUMBER
FROM INVOICE
WHERE DUE_DATE<TODAY;
```

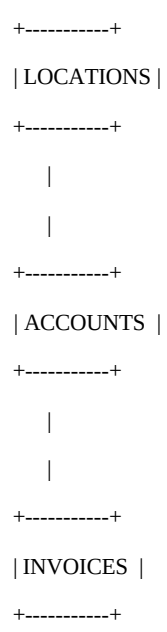
This should be compared to a PL/l program with a loop to get next record, and test for DUE_DATE and END_OF_FILE. The MONTHLY_STATEMENT transaction described in the introduction is another instance of the power and usefulness of relational operators.

On the other hand, if the work to be done does not involve processing many records, then the relational model seems to have little advantage over other models. Consider the DEBIT_CREDIT transaction which (1) reads a message from a terminal, (2) finds an account, (3) updates the account, (4) posts a history record, (5) updates the teller cash drawer, (6) updates the branch balance, and (7)puts a message to the terminal. Such a transaction would benefit little from relational operators (each operation touches only one record.)

One can define aggregate operators that distribute over hierarchies or networks. For example, the MAPLIST function of LISP distributes an arbitrary function over an arbitrary data structure.

3.4.2. Hierarchical Data Model

Hierarchical models use parent-child sets in a stylized way to produce a forest (collection of trees) of records. A typical application might use the three record types: LOCATIONS, ACCOUNTS, and INVOICES and two parent-child sets to construct the following hierarchy: All the accounts at a location are clustered together and all outstanding invoices of an account are clustered with the account, That is, a location has its accounts as children and an account has its invoices as children. This may be depicted schematically by:



This structure has the advantage that records used together may appear clustered together in physical storage and that information common to all the children can be factored into the parent record. Also, one may quickly find the first record under a parent and deduce when the last has been seen without scanning the rest of the database.

Finding all invoices for an account received on a certain day involves positioning a cursor on the location, another cursor on the account number under that location, and a third cursor to scan over the invoices:

```
SET CURSOR1 to LOCATION=NAPA;
SET CURSOR2 TO ACCOUNT=FREEMARK_ABBEY;
SET CURSOR3 BEFORE FIRST_CHILD(CURSOR2);
DO WHILE  (! END_OF_CHILDREN):
    FETCH (CURSOR3) NEXT CHILD;
DO_SOMETHING;
END;
```

Because this is such a common phenomenon, and because in a hierarchy there is only one path to a record, most hierarchical systems abbreviate the cursor setting operation to setting the lowest cursor in the hierarchy by specifying a "fully qualified key" or path from the root to the leaf (the other cursors are set implicitly.) In the above example:

```
SET CURSOR3 TO LOCATION=NAPA,
    ACCOUNT=FREEMARK_ABBEY,
    INVOICE_ANY;
DO WHILE (! END_OF_CHILDREN):
    FETCH (CURSOR3) NEXT CHILD;
DO_SOMETHING;
END;
```

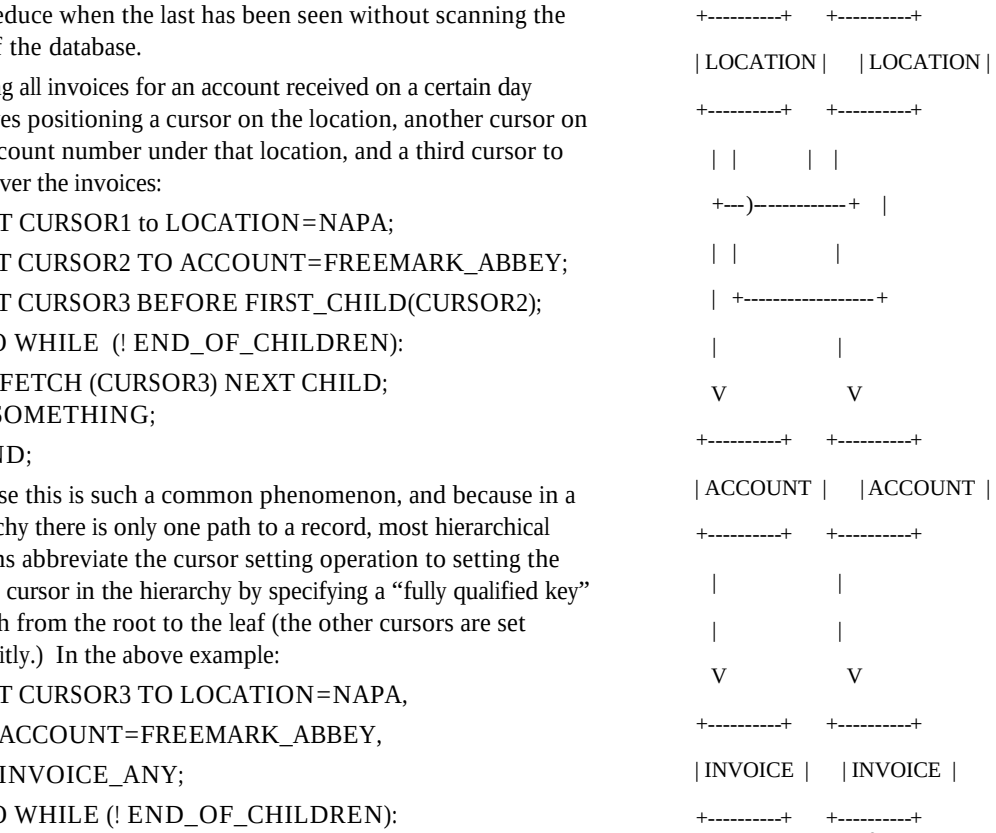
Which implicitly sets up cursors one and two.

The implicit record naming of the hierarchical model makes programming much simpler than for a general network. If the data can be structured as a hierarchy in some application then it is desirable to use this model to address it.

3.4.3. Network Data Model

Not all problems conveniently fit a hierarchical model. If nothing else, different users may want to see the same information in a different hierarchy. For example an application might want to see the hierarchy "upside-down" with invoice at the top and location at the bottom. Support for logical hierarchies (views) requires that the data management system support a general network. The efficient implementation of certain relational operators (sort-merge or join) also require parent-child sets and so require the full capability of the network data model.

The general statement is that if all relationships are nested one-to-many mappings then the data can be expressed as a hierarchy. If there are many-to-many mappings then a network is required. To consider a specific example of the need for networks, imagine that several locations may service the same account and that each location services several accounts, Then the hierarchy introduced in the previous section would require either that locations be subsidiary to accounts and be duplicated or that the accounts record be duplicated in the hierarchy under the two locations. This will give rise to complexities about the account having two balances..... A network model would allow one to construct the structure:



A network built out of two parent-child sets.

3.4.4. Comparison of Data Models

By using "symbolic" pointers (keys), one may map any network data structure into a relational structure. In that sense all three models are equivalent, and the relational model is completely general. However, there are substantial differences in the style and convenience of the different models. Analysis of specific cases usually indicates that associative pointers (keys) cost three

page faults to follow (for a multi-megabyte set) whereas following a direct pointer costs only one page fault. This performance difference explains why the equivalence of the three data models is irrelevant. If there is heavy traffic between sets then pointers must be used. (High-level languages can hide the use of these pointers.)

It is my bias that one should resort to the more elaborate model only when the simpler model leads to excessive complexity or to poor performance.

3.5. Views

Records, sets, and networks that are actually stored are called base objects. Any query evaluates to a virtual set of records which may be displayed on the user's screen, fed to a further query, deleted from an existing set, inserted into an existing set, or copied to form a new base set. More importantly for this discussion, the query definition may be stored as a named m. The principal difference between a copy and a view is that updates to the original sets that produced the virtual set will be reflected in a view but will not affect a copy. A view is a dynamic picture of a query whereas a copy is a static picture.

There is a need for both views and copies. Someone wanting to record the monthly sales volume of each department might run the following transaction at the end of each month (an arbitrary syntax):

```
MONTHLY_VOLUME=
    SELECT DEPARTMENT, SUM(VOLUME)
    FROM SALES GROUPED BY DEPARTMENT;
```

The new base set MONTHLY_VOLUME is defined to hold the answer. On the other hand, the current volume can be gotten by the view:

```
    DEFINE CURRENT_VOLUME (DEPARTMENT, VOLUME) VIEW AS:
        SELECT DEPARTMENT, SUM(VOLUME)
        FROM SALES
        GROUPED BY DEPARTMENT;
```

Thereafter, any updates to the SALES set will be reflected in the CURRENT_VOLUME view. Again, CURRENT_VOLUME may be used in the same ways base sets can be used. For example one can compute the difference between the current and monthly volume.

The semantics of views are quite simple. Views can be supported by a process of substitution in the abstract syntax (parse tree) of the statement. Each time a view is mentioned, it is replaced by its definition.

To summarize, any query evaluates to a virtual set. Naming this virtual set makes it a view. Thereafter, this view can be used as a set. This allows views to be defined as field and record subsets of sets, statistical summaries of sets and more complex combinations of sets.

There are three major reasons for defining views:

- Data independence: giving programs a logical view of data, thereby isolating them from data reorganization.
- Data isolation: giving the program exactly that subset of the data it needs, thereby minimizing error propagation.
- Authorization: hiding sensitive information from a program, its authors, and users.

As the database evolves, records and sets are often "reorganized". Changing the underlying data should not cause all the programs to be recompiled or rewritten so long as the semantics of the data is not changed. Old programs should be able to see the data in the old way. Views are used to achieve this.

Typical reorganization operations include:

- Adding fields to records.
- Splitting records.
- Combining records.
- Adding or dropping access paths.

Simple view of base records may be obtained by:

- Renaming or permuting fields,
- Converting the representation of a field.

Simple variations of base sets may be obtained by:

- Selecting that subset of the records of a set which satisfy some predicate;
- Projecting out some fields or records in the set.
- Combining existing sets together into new virtual sets that can be viewed as a single larger set.

Consider the example of a set of records of the form:

```
+------+---------+------------------+----------------+
| NAME | ADDRESS | TELEPHONE_NUMBER | ACCOUNT_NUMBER |
+------+---------+------------------+----------------+
```

Some applications might be only interested in the name and telephone number, others might want name and address while others might want name and account number, and of course one application would like to see the whole record. A view can appropriately subset the base set if the set owner decides to partition the record into two new record sets:

```
PHONE_BOOK                      ACCOUNTS
+------+---------+-------------+    +-------+----------------+
| NAME | ADDRESS | PHONE_NUMBER |    | NAME  | ACCOUNT_NUMBER |
+------+---------+-------------+    +-------+----------------+
```

Programs that used views will now access base sets (records) and programs that accessed the entire larger set will now access a view (logical set/record). This larger view is defined by:

```
    DEFINE VIEW WHOLE_THING:
        SELECT NAME, ADDRESS, PHONE_NUMBER, ACCOUNT_NUMBER
        FROM PHONE_BOOK, ACCOUNTS
        WHERE PHONE_BOOK.NAME = ACCOUNTS.NAME;
```

3.5.1 Views and Update

Any view can support read operations; however, since only base sets are actually stored, only base sets can actually be updated. To make an update via a view, it must be possible to propagate the updates down to the underlying base set.

If the view is very simple (e. g., record subset) then this propagation is straightforward. If the view is a one-to-one mapping of records in some base set but some fields of the base are missing from the view, then update and delete present no problem but insert requires that the unspecified ("invisible") fields of the new records in the base set be filled in with the "undefined" value. This may or may not be allowed by the integrity constraints on the base set.

Beyond these very simple rules, propagation of updates from views to base sets becomes complicated, dangerous, and sometimes impossible.

To give an example of the problems, consider the WHOLE_THING view mentioned above. Deletion of a record may be implemented by a deletion from one or both of the constituent sets (PHONE_BOOK and ACCOUNTS). The correct deletion rule is dependent on the semantics of the data. Similar comments apply to insert and update.

My colleagues and I have resigned ourselves to the idea that there is no elegant solution to the view update problem. (Materialization (reading) is not a problem!) Existing systems use either very restrictive view mechanisms (subset only), or they provide incredibly ad hoc view update facilities. We propose that simple views (subsets) be done automatically and that a technique akin to that used for abstract data types be used for complex views: the view definer will specify the semantics of the operators NEXT, FETCH, INSERT, DELETE, and UPDATE,

3.6. Structure of Data Manager

Data manager is large enough to be subdivided into several components:

- View component: is responsible for interpreting the request, and calling the other components to do the actual work. The view component implements cursors and uses them to communicate as the internal and external representation of the view.
- Record component: stores logical records on "pages", manages the contents of pages and the problems of variable length and overflow records.
- Index component: implements sequential and associative access to sets. If only associative access is required, hashing should be used. If both sequential and associative accesses are required then indices implemented as B-trees should be used (see Knuth Vol. 3 or IBM's Virtual Sequential Access Method.)
- Buffer manager: maps the data "pages" on secondary storage to a primary storage buffer pool. If the operating system provided a really fancy page manager (virtual memory) then the buffer manager might not be needed. But, issues such as double buffering of sequential I/O, Write Ahead Log protocol (see recovery section), checkpoint, and locking seem to argue against using the page managers of existing systems. If you are looking for a hard problem, here is one: define an interface to page management that is useable by data management in lieu of buffer management.

3.7. A Sample Data Base Design

The introduction described a very simple database and a simple transaction that uses it. We discuss how that database could be structured and how the transaction would access it.

The database consists of the records

ACCOUNT(ACCOUNT_NUMBER, CUSTOMER_NUMBER, ACCOUNT_BALANCE, HISTORY) CUSTOMER(CUSTOMER_NUMBER, CUSTOMER_NAME, ADDRESS,.....)

HISTORY(TIME, TELLER, CODE, ACCOUNT_NUMBER, CHANGE, PREV_HISTORY)

CASH_DRAWER(TELLER_NUMBER, BALANCE)

BRANCH_BALANCE(BRANCH, BALANCE)

TELLER(TELLER_NUMBER, TELLER_NAME,.....)

This is a very cryptic description that says that a customer record has fields giving the customer number, customer name, address and other attributes.

The CASH_DRAWER, BRANCH_BALANCE and TELLER files (sets) are rather small (less than 100,000 bytes). The ACCOUNT and CUSTOMER files are large (about 1,000,000,000 bytes). The history file is extremely large. If there are fifteen transactions against each account per month and if each history record is fifty bytes then the history file grows 7,500,000,000 bytes per month. Traffic on BRANCH_BALANCE and CASH_DRAWER is high and access is by BRANCH_NUMBER and TELLER_NUMBER respectively. Therefore these two sets are kept in high-speed storage and are accessed via a hash on these attributes. Traffic on the ACCOUNT file is high but random. Most accesses are via ACCOUNT_NUMBER but some are via CUSTOMER_NUMBER. Therefore, the file is hashed on ACCOUNT_NUMBER (partitioned set). A key-sequenced index, NAMES, is maintained on these records that gives a sequential and associative access path to the records ascending by customer name. CUSTOMER is treated similarly (having a hash on customer number and an index on customer name.) The TELLER file is organized as a sequential set. The HISTORY file is the most interesting. These records are written once and thereafter are only read. Almost every transaction generates such a record and for legal reasons the file must be maintained forever. This causes it to be kept as an entry sequenced set. New records are inserted at the end of the set. To allow all recent history records for a specific account to be quickly located, a parent child set is defined to link each ACCOUNT record (parent) to its HISTORY records (children). Each ACCOUNT record points to its most recent HISTORY record. Each HISTORY record points to the previous history record for that ACCOUNT.

Given this structure, we can discuss the execution of the DEBIT_CREDIT transaction outlined in the introduction. We will assume that the locking is done at the granularity of a page and that recovery is achieved by keeping a log (see section on transaction management.)

At initiation, the data manager allocates the cursors for the transaction on the ACCOUNTS, HISTORY, BRANCH, and CASH_DRAWER sets. In each instance it gets a lock on the set to insure that the set is available for update (this is an IX mode lock as explained in the locking section 5.7.6.2.) Locking at a finer granularity will be done during transaction execution (see locking section). The first call the data manager sees is a request to find the ACCOUNT record with a given account number. This is done by hashing the account number, thereby computing an anchor for the hash chain. Buffer manager is called to bring that page of the file into fast storage. Buffer manager looks in the buffer pool to see if the page is there. If the page is present, buffer manager returns it immediately. Otherwise, it finds a free buffer page slot, reads the page into that buffer and returns the filled buffer slot. Data manager then locks the page in share mode (so that no one else modifies it). This lock will be held to the end of the transaction. The record component searches the page for a record with that account number. If the record is found, its value is returned to the caller and the cursor is left pointing at the record.

The next request updates account balance of the record addressed by the cursor. This requires converting the share mode lock acquired by the previous call to a lock on the page in exclusive mode, so that no one else sees the new account balance until the transaction successfully completes. Also the record component must write a log record that allows it to undo or redo this update in case of a transaction or system abort (see section on recovery). Further, the transaction must note that the page depends on a certain log record so that buffer manager can observe the write ahead log protocol (see recovery section.)Lastly, the record component does the update to the balance of the record.

Next the transaction fabricates the history record and inserts it in the history file as a child of the fetched account. The record component calls buffer manager to get the last page of the history file (since it is an entry sequence set the record goes on the last page.) Because there is a lot of insert activity on the HISTORY file, the page is likely to be in the buffer pool. So buffer manager returns it, the record component locks it, and updates it. Next, the record component updates the parent-child set so that the new history record is a child of the parent account record. All of these updates are recorded in the system log in case of error.

The next call updates the teller cash drawer. This requires locking the appropriate CASH_DRAWER record in exclusive mode (it is located by hash). An undo-redo log record is written and the update is made.

A similar scenario is performed for the BRANCH_BALANCE file. When the transaction ends, data manager releases all its locks and puts the transaction's pages in the buffer manager's chain of pages eligible for write to disk.

If data manager or any other component detects an error at any point, it issues an ABORT_TRANSACTION command, which initiates transaction undo (see recovery section.)This causes data manager to undo all its updates to records on behalf of this user and then to release all its locks and buffer pages.

The recovery and locking aspects of data manager are elaborated in later sections. I suggest the reader design and evaluate the performance of the MONTHLY_STATATEMENT transaction described in the introduction as well as a transaction which given two dates and an account number will display the history of that account for that time interval.

3.8. Comparison To File Access Methods

From the example above, it should be clear that data manager is a lot more fancy than the typical file access methods (indexed sequential files). File systems usually do not support partitioned or parent-child sets. Some support the notion of record, but none support the notions of field, network or view. They generally lock at the granularity of a file rather than at the granularity of a record. File systems generally do recovery by taking periodic image dumps of the entire file. This does not work well for a transaction environment or for very large files.

In general, data manager builds upon the operating system file system so that

- The operating system is responsible for device support.
- The operating system utilities for allocation, import, export and accounting are useable.
- The data is available to programs outside of the data manager.

4. Data Communication

The area of data communications is the least understood aspect of DB/DC systems. It must deal with evolving network managers, evolving intelligent terminals and in general seems to be in a continuing state of chaos. Do not feel too bad if you find this section bewildering.

Data communications is responsible for the flow of messages. Messages may come via telecommunications lines from terminals and from other systems, or messages may be generated by processes running within the system. Messages may be destined for external endpoints, for buffer areas called queues, or for executing processes. Data communications externally provides the functions of:

- Routing messages.
- Buffering messages,
- Message mapping so sender and receiver can each be unaware of the physical characteristics of the other.

Internally data communications provides:

- Message transformation that maps "external" messages to and from a format palatable to network manager.
- Device control of terminals.
- Message recovery in the face of transmission errors and system errors.

4.1. Messages, Sessions, And Relationship To Network Manager

Messages and endpoints are the fundamental objects of data communications. A message consists of a set of records. Records in turn consist of a set of fields. Messages therefore look very much like database sequential sets. Messages are defined by message descriptors. Typical unformatted definitions might be: A line from typewriter terminal is a one field, one

record message. A screen image for a display is a two field (control and data), one record message. A multi-screen display image is a multi-field multi-record message.

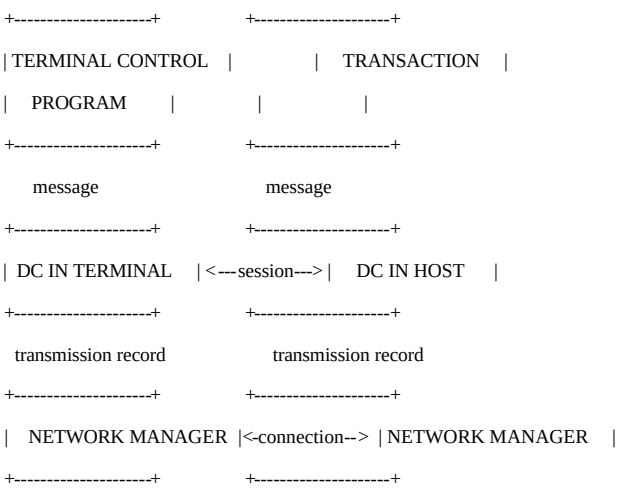
Data communications depends heavily on the network manager provided by the base operating system. ARPANET, DECNET, and SNA (embodied in NCP and VTAM) are examples of such network managers. The network manager provides the notion of endpoint that is the smallest addressable network unit. A workstation, a queue, a process, and a card reader are each examples of endpoints.

Network manager transmits rather stylized transmission records (TRs).

These are simply byte strings. Network manager makes a best effort to deliver these byte strings to their destination. It is the responsibility of data communications to package messages (records and fields) into transmission records and then reconstructs the message from transmission records when they arrive at the other end.

The following figure summarizes this: application and terminal control programs see messages via sessions. DC in the host and terminal map:

these messages into transmission records that are carried by the network manager.



The three main layers of a session.

There are two ways to send messages: A one shot message can be sent to an endpoint in a canonical form with a very rigid protocol. Logon messages (session initiation) are often of this form. The second way to send messages is via an established session between the two endpoints. When the session is established, certain protocols are agreed to (e. g. messages will be recoverable (or not), session is half or full duplex, …) Thereafter, messages sent via the session these protocols. Sessions:

- Establish the message formats desired by the sender and receiver.
- Allow sender and receiver to validate one another's identity once rather than revalidating each message.
- Allow a set of messages to be related together (see conversations).
- Establish recovery, routing, and pacing protocols.

The network operating system provides connections between endpoints. A connection should be thought of as a piece of wire that can carry messages blocked (by data communications) into transmission records. Sessions map many to one onto connections. At any instant, a session uses a particular connection. But if the connection fails or if an endpoint fails, the session may be transparently mapped to a new connection. For example, if a terminal breaks, the operator may move the session to a new terminal. Similarly, if a connection breaks, an alternate connection may be established. Connections hide the problems of transmission management (an SNA term):

- Transmission control, blocking and de-blocking transmission records (TRs), managing TR sequence numbers, and first-level retry logic.
- Path control, or routing of TRs through the network.
- Link control, sending TRs over teleprocessing lines.
- Pacing, dividing the bandwidth and buffer pools of the network among connections.
- The data communications component and the network manager cooperate in implementing the notion of session.

4.2. Session Management

The principal purpose of the session notion is:

- Device independence: the session makes transparent whether the endpoint is ASCII, EBCDIC, one-line, multi-line, program or terminal.
- Abstraction: manages the high level protocols for recovery, related messages and conversations.

Session creation specifies the protocols to be used on the session by each participant. One participant may be an ASCII typewriter and the other participant may be a sophisticated system. In this case the sophisticated system has lots of logic to handle the session protocol and errors on the session. On the other hand if the endpoint is an intelligent terminal and if the other endpoint is willing to accept the terminals protocol the session management is rather simple. (Note: The above is the way it is supposed to work. In practice sessions with intelligent terminals are very complex and the programs are much more subtle because intelligent terminals can make such complex mistakes. Typically, it is much easier to handle a master-slave session than to handle a symmetric session.)

Network manager simply delivers transmission records to endpoints. So, it is the responsibility of data communications to "know" about the device characteristics and to control the device. This means that DC must implement all the code to provide the terminal appearance. There is a version of this code for each device type (display, printer, typewriter,...), This causes the DC component to be very big in terms of thousands (K) Lines Of Code (KLOC).

If the network manager defines a generally useful endpoint model, then the DC manager can use this model for endpoints that fit the model. This is the justification for the TYPE1, TYPE2,... (endpoints) of SNA, and justifies the attempts to define a network logical terminal for ARPANET.

Sessions with dedicated terminals and peer nodes of the network are automatically (re)established when the system is (re)started. Of course, the operator of the terminal must re-establish his identity so that security will not be violated. Sessions for switched lines are created dynamically as the terminals connect to the system.

When DC creates the session, it specifies what protocols are to be used to translate message formats so that the session user is not aware of the characteristics of the device at the other end point.

4.3. Queues

As mentioned before a session may be:

FROM a program or terminal or queue

TO a program or terminal or queue

Queues allow buffered transmission between endpoints. Queues are associated (by DC) with users, endpoints and transactions. If a user is not logged on or if a process is doing something else, a queue can be used to hold one or more messages thereby freeing the session for further work or for termination. At a later time the program or endpoint may poll the queue and obtain the message.

Queues are actually passive so one needs to associate an algorithm with a queue. Typical algorithms are:

- Allocate N servers for this queue,
- Schedule a transaction when a message arrives in this queue.
- Schedule a transaction when N messages appear in the queue.
- Schedule a transaction at specified intervals.

Further, queues may be declared to be recoverable in which case DC is responsible for reconstructing the queue and it's messages if the system crashes or if the message consumer aborts.

4.4. Message Recovery

A session may be designated as recoverable in which case, all messages traveling on the session are sequence numbered and logged. If the transmission fails (positive acknowledge not received by sender), then the session endpoints resynchronize back to that message sequence number and the lost and subsequent messages are re-presented by the sender endpoint. If one of the endpoints fails, when it is restarted the session will be reestablished and the communication resumed. This requires that the endpoints be "recoverable" although one endpoint of a session may assume recovery responsibility for the other.

If a message ends up in a recoverable queue then:

- If the dequeuer of the message (session or process) aborts, the message will be replaced in the queue.
- If the system crashes, the queue will be reconstructed (using the log or some other recovery mechanism).

If the session or queue is not recoverable, then the message may be lost if the transmission, dequeuer or the system fails.

It is the responsibility of the data communications component to assure that a recoverable message is "successfully" processed or presented exactly once. It does this by requiring that receipt of recoverable messages acknowledged. A transaction "acknowledges" receipt of a message after it has processed it, at commit time (see recovery section.)

4.5. Response-mode Processing.

The default protocol for recoverable messages consists of the scenario:

```
TERMINAL                    SYSTEM

        --------request----->

                        log input message (forced)

        <-------acknowledge--

                        process message

                        commit (log reply forced)

        <-------reply-------

        --------acknowledge->

                        log reply acknowledged
```

This implies four entries to the network manager (at each endpoint).

Each of these passes requires several thousand instructions (in typical implementations.) If one is willing to sacrifice the recoverability of the input message then the logging and acknowledgment of the input message can be eliminated and the reply sequence used as acknowledgment. This reduces line traffic and interrupt handling by a factor of two. This is the response mode message processing. The output (commit) message is logged. If something goes wrong before commit, it is as though the message was never received. If something goes wrong after commit then the log is used to re-present the message. However, the sender must be able to match responses to requests (he may send five requests and get three responses). The easiest way to do this is to insist that there is at most one message outstanding at a time (i.e. lock the keyboard).

Another scheme is to acknowledge a batch of messages with a single acknowledge. One does this by tagging the acknowledge with the sequence number of the latest message received. If messages are recoverable, then the sender must retain the message until it is acknowledged and so acknowledges should be sent fairly frequently.

4.5. Conversations

A conversation is a sequence of messages. Messages are usually grouped for recovery purposes so that all are processed or ignored together. A simple conversation might consist of a clerk filling out a form. Each line the operator enters is checked for syntactic correctness and checked to see that the airline flight is available or that the required number of widgets is in stock. If the form passes the test it is redisplayed by the transaction with the unit price and total price for the line item filled in. At any time the operator can abort the conversation. However, if the system backs up the user (because of deadlock) or if the system crashes then when it restarts it would be nice to re-

present the message of the conversation so that the operator's typing is saved. This requires that the group of messages be identified to the data communications component as a conversation so that it can manage this recovery process. (The details of this protocol are an unsolved problem so far as I know.)

4.6. Message Mapping

One of the features provided by DC is to insulate each end-point from the characteristics of the other. There are two levels of mapping to do this: One level maps transmission records into messages. The next level maps the message into a structured message. The first level of mapping is defined by the session; the second level of mapping is defined by the recipient (transaction). The first level of mapping converts all messages into some canonical form (e. g. a byte string of EBCDIC characters.) This transformation may handle such matters as pagination on a screen (if the message will not fit on one screen image). The second level of mapping transforms the message from an uninterpreted string of bytes into a message-record-field structure. When one writes a transaction, one also writes a message mapping description that makes these transformations. For example, an airlines reservation transaction might have a mapping program that first displays a blank ticket. On input, the mapping program extracts the fields entered by the terminal operator and puts them in a set of (multi-field) records. The transaction reads these records (much in the style database records are read) and then puts out a set of records to be displayed. The mapping program fills in the blank ticket with these records and passes the resulting byte string to session management.

4.7. Topics Not Covered

A complete discussion of DC should include:

- More detail on network manager (another lecturer will cover that).
- Authorization to terminals (another lecturer will cover that).
- More detail on message mapping.
- The Logon-Signon process.

5. Transaction Management

The transaction management system is responsible for scheduling system activity, managing physical resources, and managing system shutdown and restart. It includes components that perform scheduling, recovery, logging, and locking.

In general transaction management performs those operating system functions not available from the basic operating system. It does this either by extending the operating system objects (e. g. enhancing processes to have recovery and logging) or by providing entirely new facilities (e-g, independent recovery management.) As these functions become better understood, the duties of transaction management will gradually migrate into the operating system.

Transaction management implements the following objects:

Transaction descriptor: A transaction descriptor is a prototype for a transaction giving instructions on how to build an instance of the transaction. The descriptor describes how to schedule the transaction, what recovery and locking options to use, what data base views the transaction needs, what program the transaction runs, and how much space and time it requires.

Process: A process (domain) that is capable of running or is running a transaction. A process is bound to a program and to other resources. A process is a unit of scheduling and resource allocation. Over time a process may execute several transaction instances although at any instant a process is executing on behalf of at most one transaction instance. Conversely, a transaction instance may involve several processes. Multiple concurrent processes executing on behalf of a single transaction instance are called cohorts. Data management system processes are fancier than operating system processes since they understand locking, recovery and logging protocols but we will continue to use the old (familiar name for them).

Transaction instance: A process or collection of processes (cohorts) executing a transaction. A transaction instance is the unit of locking and recovery.

In what follows, we shall blur these distinctions and generically call each of these objects transactions unless a more precise term is needed.

The life of a transaction instance is fairly simple-A message or request arrives which causes a process to be built from the transaction descriptor. The process issues a BEGIN_TRANSACTION action that establishes a recovery unit. It then issues a series of actions against the system state. Finally it issues the COMMIT_TRANSACTION action that causes the outputs of the transaction to be made public (both updates and output messages.) Alternatively, if the transaction runs into trouble, it may issue the ABORT TRANSACTION action which cancels all actions performed by this transaction.

The system provides a set of objects and actions on these objects along with a set of primitives that allow groups of actions to be collected into atomic transactions. It guarantees no consistency on the objects beyond the atomicity of the actions. That is, an action will either successfully complete or it will not modify the system state at all.

Further, if two actions are performed on an object then the result will be equivalent to the serial execution of the two actions. (As explained below this is achieved by using locking within system actions.)

The notion of transaction is introduced to provide a similar abstraction above the system interface. Transactions are an all or nothing thing, either they happen completely or all trace of then (except in the log) is erased.

Before a transaction completes, it may be aborted and its updates to recoverable data may be undone. The abort can cone either from the transaction itself (suicide: bad input data, operator cancel,….) or from outside (murder: deadlock, timeout, system crash....) However, once a transaction commits (successfully completes), the effects of the transaction cannot be blindly undone. Rather, to undo a committed transaction, one must resort to compensation - running a new transaction that corrects the errors of its predecessor. Compensation is usually highly application dependent and is not provided by the system.

These definitions may be clarified by a few examples. The following is a picture of the three possible destinies of a transaction.

```
BEGIN          BEGIN              BEGIN
action         action             action
action         action             action
  .              .        ABORT =>  action
  .              .
  .            ABORT
action
COMMIT
A successful   A suicidal         A murdered
Transaction    transaction        transaction
```

A simple transaction takes in a single message does something, and then produces a single message. Simple transactions typically make fifteen data base calls. Almost all transactions are simple at present (see Guide/Share Profile of IMS users). About half of all simple transactions are read-only (make no changes to the database.) For simple transactions, the notion of process, recovery unit and message coincide.

If a transaction sends and receives several synchronous messages it is called a conversational transaction. A conversational transaction has several messages per process and transaction instances. Conversational transactions are likely to last for a long time (minutes while the operator thinks and types) and hence pose special resource management problems.

The term batch transaction is used to describe a transaction that is "unusually big". In general such transactions are not on-line, rather they are usually started by a system event (timer driven) and run for a long time as a "background" job. Such a transaction usually performs thousands of data management calls before terminating. Often, the process will commit some of its work before the entire operation is complete. This is an instance of multiple (related) recovery units per process.

If a transaction does work at several nodes of a network then it will require a process structure (cohort) to represent its work at each participating node. Such a transaction is called distributed.

The following table summarizes the possibilities and shows the independence of the notions of process, message and transaction instance (commit). Cohorts communicate with one another via the session-message facilities provided by data communications.

	PROCESSES	MESSAGES	COMMITS
SIMPLE	1	1 in 1 out	1
CONVERSATIONAL	1	many in many out	1
BATCH	1	none(?)	many
DISTRIBUTED	many	1 in 1 out	1
		many among cohorts	

We introduce an additional notion of save point in the notion of transaction. A save point is a firewall that allows a transaction to stop short of total backup. If a transaction gets into trouble (e. g. deadlock, resource limit) it may be sufficient to back up to such an intermediate save point rather than undoing all the work of the transaction. For example a conversational transaction which involves several user interactions might establish a save point at each user message thereby minimizing retyping by the user. Save points do not commit any of the transaction's updates. Each save point is numbered, the beginning of the transaction is save point 1 and successive save points are numbered 2, 3. . . . . The user is allowed to save some data at each save point and to retrieve this data if he returns to that point. Backing up to save point 1 resets the transaction instance to the recovery component provides the actions:

- BEGIN_TRANSACTION: designates the beginning of a transaction.
- SAVE_TRANSACTION: designates a firewall within the transaction.

If an incomplete transaction is backed-up, undo may stop at such a point rather than undoing the entire transaction,

- BACKUP_TRANSACTION: undoes the effects of a transaction to an earlier save point.
- COMMIT_TRANSACTION: signals successful completion of transaction and causes outputs to be committed.
- ABORT_TRANSACTION: causes undo of a transaction to its initial state.

Using these primitives, application programs can construct groups of actions that are atomic. It is interesting that this one level of recovery is adequate to support multiple levels of transactions by using the notion of save point.

The recovery component supports two actions that deal with system recovery rather than transaction recovery:

- CHECKPOINT: Coordinates the recording of the system state in the log.
- RESTART: Coordinates system restart, reading the checkpoint log record and using the log to redo committed transactions and to undo transactions that were uncommitted at the time of the shutdown or crash.

5.1. Transaction Scheduling

The scheduling problem can be broken into many components: listening for new work, allocating resources for new work, scheduling (maintaining the dispatcher list), and dispatching.

The listener is event driven. It receives messages from data communications and from dispatched processes.

A distinguished field of the message specifies a transaction name. Often, this field has been filled in by data communications that resolved the transaction name to a reference to a transaction descriptor. Sometimes this field is symbolic in which case the listener uses the name in a directory call to get a reference to the transaction descriptor, (The directory may be determined by the message source.) If the name is had or if the sender is not authorized to invoke the transaction then the message is discarded and a negative acknowledge is sent to the source of the message.

If the sender is authorized to invoke the named transaction, then the allocator examines the transaction descriptor and the current system state and decides whether to put this message in a work-to-do list or to allocate the transaction right away, Criteria for this are:

- The system may be overloaded (“ full”.)
- There may be a limit on the number of transactions of this type, which can run concurrently.
- There may be a threshold, N, such that N messages of this type must arrive, at which point a server is allocated and the messages are batched to this server.
- The transaction may have an affinity to resources, which are unavailable.
- The transaction may run at a special time (overnight, off-shift,...)

If the transaction can run immediately, then the allocator either allocates a new process to process the message or gives the message to a primed transaction that is waiting for input.

If a new process is to be created, a process (domain) is allocated and all objects mentioned in the transaction descriptor are allocated as part of the domain, Program management sets up the address space to hold the programs, data management will allocate the cursors of the transaction for the process, data communication allocates the necessary queues, the recovery component allocates a log cursor and writes a begin transaction log record, and so on. The process is then set up with a pointer to the input message.

This allocated process is given to the scheduler that eventually places it on the dispatcher queue. The dispatcher eventually runs the process.

once the transaction scheduler dispatches the process, the operating system scheduler is responsible for scheduling the process against the physical resources of the system.

When the transaction completes, it returns to the scheduler. The scheduler may or may not collapse the process structure depending on whether the transaction is batched or primed. If the transaction has released resources needed by waiting unscheduled transactions, the scheduler will now dispatch these transactions.

Primed transactions are an optimization that dramatically reduce allocation and deallocation overhead. Process allocation can be an expensive operation and so transactions that are executed frequently are often primed. A primed transaction has a large part of the domain already built. In particular programs are loaded, cursors are allocated and the program prolog has been executed. The transaction (process) is waiting for input. The scheduler need only pass the message to the transaction (process), Often the system administrator or operator will prime several instances of a transaction. A banking system doing three withdrawals and five deposits per second might have two withdrawal transactions and four deposit transactions primed.

Yet another variant has the process ask for a message after it completes. If a new message has arrived for that transaction type, then the process processes it. If there is no work for the transaction, then the process disappears. This is called batching messages as opposed to priming. It is appropriate if message traffic is “bursty” (not uniformly distributed in time). It avoids keeping a process allocated when there is no work for it to do.

5.2. Distributed Transaction Management

A distributed system is assumed to consist of a collection of autonomous nodes that are tied together with a distributed data communication system in the style of high level ARPANET, DECNET, or SNA protocols. Resources are assumed to be partitioned in the sense that a resource is owned by only one node. The system should be:

- Inhomogeneous (nodes are small, medium, large, ours, theirs,...)
- Unaffected by the loss of messages.
- Unaffected by the loss of nodes (i.e. requests to that node wait for the node to return, ether nodes continue working.)

Each node may implement whatever data management and transaction management system it wants to. We only require that it obey the network protocols, Some node might be a minicomputer running a fairly simple data management system and using an old-master new-master recovery protocol. Another node might be running a very sophisticated data management system with many concurrent transactions and fancy recovery.

If one transaction may access resources in many nodes of a network then a part of the transaction must “run” in each node. We already have an entity that represents transaction instances: processes. Each node will want to

- Authorize local actions of the process (transaction).
- Build an execution environment for the process (transaction).
- Track local resources held by the process (transaction).
- Establish a recovery mechanism to undo the local updates of that process (see recovery section).
- Observe the two-phase commit protocol (in cooperation with its cohorts (see section on recovery)).

Therefore, the structure needed for a process in a distributed system is almost identical to the structure needed by a transaction in a centralized system.

This latter observation is key. That is why I advocate viewing each node as a transaction processor. (This is a minority view.) To install a distributed transaction, one must install prototypes for its cohorts in the various nodes. This allows each node to control access by distributed transactions in the same way it controls access by terminals. If a node wants to give away the keys to its kingdom it can install a universal cohort (transaction) which has access to all data and which performs all requests.

If a transaction wants to initiate a process (cohort) in a new node, some process of the transaction must request that the node construct a cohort and that the cohort go into session with the requesting process (see data communications section for a discussion of sessions). The picture below shows this.

Two cohorts of a distributed transaction in session.

A process carries both the transaction name T1 and the process name (in NODE1 the cohort of T1 is process P2 and in NODE2 the cohort of T1 is process P6.)

The two processes can now converse and carry out the work of the transaction. If one process aborts, they should both abort, and if one process commits they should both commit. Thus they need to:

- Obey the lock protocol of holding locks to end of transaction (see section on locking).
- Observe the two-phase commit protocol (see recovery section).

These comments obviously generalize to transactions of more than two charts.

5.3. The Data Management System As A Subsystem

It has been the recent experience of general-purpose operating systems that the operating system is extended or enhanced by some “application program” like a data management system, or a network management system. Each of these systems often has very clear ideas about resource management and scheduling. It is almost impossible to write such systems unless the basic operating system:

- allows the subsystem to appear to users as an extension of the basic operating system.
- allows the subsystem to participate in major system events such as system shutdown/restart, process termination,....

To cope with these problems, operating systems have either made system calls indistinguishable from other calls (e. g. MULTICS) or they have reserved a set of operating systems calls for subsystems (e. g. user SVCs in OS/360.) These two approaches address only the first of the two problems above.

The notion of subsystem is introduced to capture the second notion. For example, in IBM’s operating system VS release 2.2, notifies each known subsystem at important system events (e. g, startup, memory failure, checkpoint,...) Typically a user might install a Job Entry Subsystem, a Network Subsystem, a Text Processing Subsystem, and perhaps several different Data Management Subsystems on the same operating system, The basic operating system serves as a coordinator among these sub-systems.

- It passes calls from users to these subsystems.
- It broadcasts events to all subsystems.

The data manager acts as a subsystem of the host extending its basic facilities.

The data management component is in turn comprised following is a partial list of the components in the bottom half of the data base component of System R:

- Catalog manager: maintains directories of system.
- Call analyzer: regulates system entry-exit.
  - Record manager: extracts records from pages.
  - Index component: maintains indices on the database.
  - Sort component: maintains sorted versions of sets.
  - Loader: performs bulk insertion of records into a file.
  - Buffer manager: maps database pages to and from secondary storage.
  - Performance monitor: Keeps statistics about system performance and state.
  - Lock component: maintains the locks (synchronization primitives).

- Recovery manager: implements the notion of transaction COMMIT, ABORT, and handles system restart.
- Log manager: maintains the system log.

Notice that primitive forms of these functions are present in most general-purpose operating systems. In the future one may expect to see the operating system subsume most of these data management functions.

5.4. Exception Handling

The protocol for handling synchronous errors (errors which are generated by the process) is another issue defined by transaction management (extending the basic operating systems facilities). In general the data management system wants to abort the transaction if the application program fails. This is generally handled by organizing the exceptions into a hierarchy. If a lower level of the hierarchy fails to handle the error, it is passed to a higher node of the hierarchy. The data manager usually has a few handlers very near the top of the hierarchy (the operating system gets the root of the hierarchy.)

- Either the process or the data management system (or both) may -establish an exception handler to field errors.
- When, an exception is detected then the exception is signaled.
- Exception handlers are invoked in some fixed order (usually order of establishment) until one successfully corrects the error. This operation is called percolation.

PL/l "ON units" or the IBM Operating System set-task-abnormal-exit (STAE) are instances of this mechanism. Examples of exception conditions are: arithmetic exception conditions (i.e., overflow), invalid program reference (i.e., to protected storage) wild branches, infinite loops, deadlock, .. and attempting to read beyond end of file.

There may be several exception handlers active for a process at a particular instant. The program's handler is usually given the first try at recovery if the program has established a handler. The handler will, in general, diagnose the failure as one that was expected (overflow), one that was unexpected but can be handled (invalid program reference), or one that is unexpected and cannot be dealt with by the handler (infinite loop). If the failure can be corrected, the handler makes the correction and continues processing the program (perhaps at a different point of execution.) If the failure cannot be corrected by this handler, then the exception will percolate to the next exception handler for that process.The system generally aborts any process, which percolates to the system recovery routine or does not participate in recovery. This process involves terminating all processing being done on behalf of the process, restoring all non-consumable resources in use by the process to operating system control (i.e., storage), and removing to the greatest extent possible the effects of the transaction.

5.5. Other Components Within Transaction Management

We mention in passing that the transaction management component must also support the following notions:

- Timer services: Performing operations at specified times. This involves running transactions at specified times or intervals and providing a timed wait if it is not available from the base operating system.
- Directory management: Management of the directories used by transaction management and other components of the system. This is a high-performance low-function in-core data management system. Given a name and a type (queue, transaction, endpoint,...)it returns a reference to the object of that name and type. (This is where the cache of dictionary descriptors is kept.)
- Authorization Control: Regulates the building and use of transactions.

These topics will be discussed by other lecturers.

5.7. Lock Management.

This section derives from papers co-authored with Irv Traiger and Franco Putzolu.

The system consists of objects that are related in certain ways. These relationships are best thought of as assertions about the objects. Examples of such assertions are:

"Names is an index for Telephone-numbers."

"Count_of_x is the number of employees in department x."

The system state is said to be consistent if it satisfies all its assertions. In some cases, the database must become temporarily inconsistent in order to transform it to a new consistent state. For example, adding a new employee involves several atomic actions and updating several fields. The database may be inconsistent until all these updates have been completed.

To cope with these temporary inconsistencies, sequences of atomic actions are grouped to form transactions. Transactions are the units of consistency. They are larger atomic actions on the system state that transform it from one consistent state to a new consistent state. Transactions preserve consistency. If some action of a transaction fails then the entire transaction is 'undone,' thereby returning the database to a consistent state. Thus transactions are also the units of recovery. Hardware failure, system error, deadlock, protection violations and program error are each a source of such failure.

5.7.1. Pros And Cons of Concurrency

If transactions are run one at a time then each transaction will see the consistent state left behind by its predecessor. But if several transactions are scheduled concurrently then the inputs of some transaction may be inconsistent even though each transaction in isolation is consistent.

Concurrency is introduced to improve system response and utilization.

- It should not cause programs to malfunction.
- Concurrency control should not consume more resources than it "saves".

If the database is read-only then no concurrency control is needed. However, if transactions update shared data then their concurrent execution needs to be regulated so that they do not update the same item at the same time.

If all transactions are simple and all data are in primary storage then there is no need for concurrency. However, if any transaction runs for a long time or does I/O then concurrency may be needed to improve responsiveness and utilization of the system. If concurrency is allowed, then long-running transactions will (usually) not delay short ones.

Concurrency must be regulated by some facility that regulates access to shared resources. Data management systems typically use locks for this purpose.

The simplest lock protocol associates a lock with each object. Whenever using the object, the transaction acquires the lock and holds it until the transaction is complete. The lock is a serialization mechanism that insures that only one transaction accesses the object at a time. It has the effect of: notifying others that the object is busy; and of protecting the lock requestor from modifications of others.

This protocol varies from the serially reusable resource protocol common to most operating systems (and recently renamed monitors) in that the lock protocol holds locks to transaction commit. It will be argued below that this is a critical difference.

Responsibility for requesting and releasing locks can either be assumed by the user or be delegated to the system. User controlled locking results in potentially fewer locks due to the user's knowledge of the semantics of the data. On the other hand, user controlled locking requires difficult and potentially unreliable application programming. Hence the approach taken by most data base systems is to use automatic lock protocols which insure protection from inconsistency, while still allowing the user to specify alternative lock protocols as an optimization.

5.7.2 Concurrency Problems

Locking is intended to eliminate three forms of inconsistency due to concurrency.

- **Lost Updates:** If transaction T1 updates a record previously updated by transaction T2 then undoing T2 will also undo the update of T1 (i.e. if transaction T1 updates record R from 100 to 101 and then transaction T2 updates A from 101 to 151 then backing out T1 will set A back to the original value of 100 losing the update of T2.) This is called a Write ->Write dependency.
- **Dirty Read:** If transaction T1 updates a record that is read by T2, then if T1 aborts, T2 will have read a record that never existed. (i.e. T1 updates R to 100,000,000, T2 reads this value, T1 then aborts and the record returns to the value 100.) This is called a Write ->Read dependency.
- **Un-repeatable Read:** If transaction T1 reads a record that is then altered and committed by T2, and if T1 re-reads the record, then T1 will see two different committed values for the sane record. Such a dependency is called a Read ->Write dependency.

If there were no concurrency then none of these anomalous cases will arise.

Note that the order in which reads occur does not affect concurrency.

In particular reads commute. That is why we do not care about Read -> Read dependencies,

5.7.3. Model of Consistency and Lock Protocols

A fairly formal model is required in order to make precise statements about the issues of locking and recovery. Because the problems are so complex one must either accept many simplifying assumptions or accept a less formal approach. A compromise is adopted here. First we will introduce a fairly formal model of transactions, locks and recovery that will allow us to discuss the issues of lock management and recovery management. After this presentation, the implementation issues associated with locking and recovery will be discussed.

5.7.3.1. Several Definitions of Consistency

Several equivalent definitions of consistency are presented. The first definition is an operational and intuitive one; it is useful in describing the system behavior to users. The second definition is a procedural one in terms of lock protocols; it is useful in explaining the system implementation. The third definition is in terms of a trace of the system actions; it is useful in formally stating and proving consistency properties.

5.7.3.1.1. Informal Definition of Consistency

An output (write) of a transaction is committed when the transaction abdicates the right to "undo" the write thereby making the new value available to all other transactions (i.e. commits). Outputs are said to be uncommitted or dirty if they are not yet committed by the writer. Concurrent execution raises the problem that reading or writing other transactions' dirty data may yield inconsistent data.

Using this notion of dirty data, consistency may be defined as:

Definition 1: Transaction T sees a consistent state if:

a. T does not overwrite dirty data of other transactions.
b. T does not commit any writes until it completes all its writes (i.e. until the end of transaction (EOT)).
c. T does not read dirty data from other transactions.
d. Other transactions do not dirty any data read by T before T completes.

Clauses (a) and (b) insure that there are no lost updates.

Clause (c) isolates a transaction from the uncommitted data of other transactions. Without this clause, a transaction might read uncommitted values, which are subsequently updated or are undone. If clause (c)is observed, no uncommitted values are read.

Clause (a) insures repeatable reads. For example, without clause (c) a transaction may read two different (committed) values if it reads the same entity twice. This is because a transaction that updates the entity could begin, update, and commit in the interval between the two reads. More elaborate kinds of anomalies due to concurrency are possible if one updates an entity after reading it or if more than one entity is involved (see example below).

The rules specified have the properties that:

1. If all transactions observe the consistency protocols, then any execution of the system is equivalent to some "serial" execution of the transactions (i.e. it is as though there was no concurrency.)
2. If all transactions observe the consistency protocols, then each transaction sees a consistent state.
3. If all transactions observe the consistency protocols, then system backup (undoing all in-progress transactions) loses no updates of completed transactions.
4. If all transactions observe the consistency protocols, then transaction backup (undoing any in-progress transaction) produces a consistent state.

Assertions 1 and 2 are proved in the paper "On the Notions of Consistency and Predicate Locks" CACM Vol. 9, No. 71, Nov. 1976. Proving the second two assertions is a good research problem. It requires extending the model used for the first two assertions and reviewed here to include recovery notions.

5.7.3.1.2. Schedules: Formalize Dirty And Committed Data

The definition of what it means for a transaction to see a consistent state was given in terms of dirty data. In order to make the notion of dirty data explicit, it is necessary to consider the execution of a transaction in the context of a set of concurrently executing transactions. To do this we introduce the notion of a schedule for a set of transactions. A schedule can be thought of as a history or audit trail of the actions performed by the set of transactions. Given a schedule the notion of a particular entity being dirtied by a particular transaction is made explicit and hence the notion of seeing a consistent state is formalized. These notions may then be used to connect the various definitions of consistency and show their equivalence.

The system directly supports objects and actions. Actions are categorized as begin actions, end actions, abort actions, share lock actions, exclusive lock actions, unlock actions, read actions, write actions. Commit actions and abort actions are presumed to unlock any locks held by the transaction but not explicitly unlocked by the transaction. For the purposes of the following definitions, share lock actions and their corresponding unlock actions are additionally considered to be read actions and exclusive lock actions and their corresponding unlock actions are additionally considered to be write actions.

For the purposes of this mad, a transaction is any sequence of actions beginning until a begin action and ending with a commit or abort action and not containing other begin, commit or abort actions. Here are two trivial transactions.

```
T1 BEGIN                T2 BEGIN
SHARE LOCK  A           SHARE LOCK B
EXCLUSIVE LOCK B        READ   B
READ        A           SHARE LOCX A
WRITE B                 READ   A
COMMIT                  ABORT
```

Any (sequence preserving) merging of the actions of a set of transactions into a single sequence is called a schedule for the set of transactions.

A schedule is a history of the order in which actions were successfully executed (it does not record actions which were undone due to backup (This aspect of the model needs to be generalized to prove assertions 3 and 4 above)). The simplest schedules run all actions of one transaction and then all actions of another transaction,... Such one-transaction-at-a-time schedules are called serial because they have no concurrency among transactions. Clearly, a serial schedule has no concurrency-induced inconsistency and no transaction sees dirty data. Locking constrains the set of allowed schedules. In particular, a schedule is legal only if it does not schedule a lock action on an entity for one transaction when that entity is already locked by some other transaction in a conflicting mode. The following table shows the compatibility among the simple lock modes.

```
+--------------------+-----------------------+
|                    |     LOCK MODE         |
+--------------------+-----------------------+
| COMPATIBILITY      |   SHARE   | EXCLUSIVE |
+--------------------+-----------+-----------+
| REQUEST | SHARE    | COMPATIBLE | CONFLICT  |
+---------+----------+------------+-----------+
| MODE    | EXCLUSIVE |  CONFLICT  | CONFLICT  |
+---------+----------+------------+-----------+
```

The following are three example schedules of two transactions. The first schedule is legal, the second is serial and legal and the third schedule is not legal since T1 and T2 have conflicting locks on the object A.

```
Tl BEGIN          Tl BEGIN          T2 BEGIN
T2 BEGIN          Tl SHARE LOCK A   Tl BEGIN
T2 SHARE LOCK B   Tl EXCLUSIVE LOCK B  Tl EXCLUSIVE LOCK A
T2 READ B         Tl READ A         T2 SHARE LOCK B
Tl SHARE LOCK A   Tl WRITE B        T2 READ B
T2 SHARE LOCK A   Tl COMMIT         T2 SHARE LOCK A
T2 READ A         T2 BEGIN          T2 READ A
T2 ABORT          T2 SHARE LOCK B   T2 ABORT
Tl EXCLUSIVE LOCK B  T2 READ B      Tl SHARE LOCK B
Tl READ A         T2 SHARE LOCK A   Tl READ A
Tl WRITE B        T2 READ A         Tl WRITE B
Tl COMMIT         T2 ABORT          Tl COMMIT
```

Legal & not serial legal & serial not legal& not serial

The three varieties of schedules (serial and not legal is impossible).

An initial state and a schedule completely define the system's behavior. At each step of the schedule one can deduce which entity values have been committed and which are dirty: if locking is used, updated data is dirty until it is unlocked.

One transaction instance is said to depend on another if the first takes some of its inputs from the second. The notion of dependency can be useful in comparing two schedules of the same set of transactions.

Each schedule, S, defines a ternary dependency relation on the set: TRANSACTIONS X OBJECTS X TRANSACTIONS as follows. Suppose that transaction T performs action a on entity e at some step in the schedule, and that transaction T' performs action a' on entity e at a later step in the schedule. Further suppose that T and T' are distinct.

Then:

(T, e, T') is in DEP(S)

if a is a write action and a' is a write action

or a is a write action and a' is a read action

or a is a read action and a' is a write action

The dependency set of a schedule completely defines the inputs and outputs each transaction "sees". If two distinct schedules have the same dependency set then they provide each transaction with the same inputs and outputs. Hence we say two schedules are equivalent if they have the same dependency sets. If a schedule is equivalent to a serial schedule, then that schedule must be consistent since in a serial schedule there are no inconsistencies due to concurrency. On the other hand, if a schedule is not equivalent to a serial schedule then it is probable (possible) that some transaction sees an inconsistent state. Hence,

Definition 2: A schedule is *consistent* if it is equivalent to some serial schedule.

The following argument may clarify the inconsistency of schedules not equivalent to serial schedules. Define the relation <<<on the set of transactions by:

T<<<T' 'if for some entity e (T, e, T') is in DEP(S).

Let <<<* be the transitive closure of <<<, then define:

$$BEFORE(T) = \{T' \mid T'<<<*T\}$$

$$AFTER(T) = \{\sim T' \mid T <<<*T'\}.$$

The obvious interpretation of this is that BEFORE(T) is the set of transactions that contribute inputs to T and each AFTER(T) set is the set of transactions that take their inputs from T

If some transaction is both before T and after T in some schedule then no serial schedule could give such results. In this case, concurrency has introduced inconsistency. On the other hand, if all relevant transactions are either before or after T (but not both)then T will see a consistent state. If all transactions dichotomize others in this way then the relation <<<*will be a partial order and the whole schedule will provide consistency.

The above definitions can be related as follows:

Assertion:

A schedule is consistent

if and only if (by definition)

the schedule is equivalent to a serial schedule

if and only if

the relation <<<*is a partial order.

5.7.3.1.3. Lock Protocol Definition of Consistency

Whether an instantiation of a transaction sees a consistent state depends on the actions of other concurrent transactions. All transactions agree to certain lock protocols so that they can all be guaranteed consistency.

Since the lock system allows only legal schedules we want a lock protocol such that: every legal schedule is a consistent schedule.

Consistency can be procedurally defined by the lock protocol, which produces it: A transaction locks its inputs to guarantee their consistency and locks its outputs to mark them as dirty (uncommitted).

For this section, locks are dichotomized as share mode locks which allow multiple readers of the same entity and exclusive mode locks which reserve exclusive access to an entity.

The lock protocols refined to these modes is:

Definition 3: Transaction T observes the consistency lock protocol if:

a. T sets an exclusive lock on any data it dirties

b. T sets a share lock on any data it reads.

c. T holds all locks to end of transaction.

These lock protocol definitions can be stated more precisely and tersely with the introduction of the following notation. A transaction is well-formed if it always locks an entity in exclusive (shared or exclusive) mode before writing (reading) it.

A transaction is two-phase if it does not (share or exclusive) lock an entity after unlocking some entity, A two-phase transaction has a growing phase during which it acquires locks and a shrinking phase during which it releases locks.

The lock consistency protocol can be redefined as:

Definition 3': Transaction T observes the consistency lock protocol if it is well formed and two phase.

Definition 3 was too restrictive in the sense that consistency does not require that a transaction hold all locks to the EOT (i.e. the ROT need not be the start of the shrinking phase). Rather, the constraint that the transaction be two-phase is adequate to insure consistency. On the other hand, once a transaction unlocks an updated entity, it has committed that entity and so cannot be undone without cascading backup to any transactions that may have subsequently read the entity. For that reason, the shrinking phase is usually deferred to the end of the transaction; thus, the transaction is always recoverable and all updates are committed together.

5.7.3.2. Relationship Among Definitions

These definitions may be related as follows: if T sees a consistent state in S.

Assertion:

a. If each transaction observes the consistency lock protocol (Definition 3'), then any legal schedule is consistent (Definition 2) (i.e. each transaction sees a consistent state in the sense of Definition 1).

b. If transaction T violates the consistency lock protocol then it is possible to define another transaction T' that does observe the consistency lock protocol such that T and T' have a legal schedule S but T does not see a consistent state in S.

This says that if a transaction observes the consistency lock protocol definition of consistency (Definition 3') then it is assured of the definition of consistency based on committed and dirty data (Definition 1 or 3). Unless a transaction actually Sets the locks prescribed by consistency one can construct transaction mixes and schedules which will cause the transaction to see an inconsistent state. However, in particular cases such transaction mixes may never occur due to the structure or use of the system. In these cases an apparently inadequate locking may actually provide consistency. For example, a data base reorganization usually need no locking since it is run as an off-line utility which is never run concurrently with other transactions.

5.7.4. Locking, Transaction Backup, And System Recovery

To repeat, there is no nice formal model of recovery (Lampson and Sturgis have a model in their forth-coming CACM paper on two-phase commit processing, but the model in the version I saw was rather vague.) Here, we will limp along with a (even more) vague model.

A transaction T is said to be recoverable if it can be undone before EOT without undoing other transactions' updates. A transaction T is said to be repeatable if it will reproduce the original output if rerun following recovery, assuming that no locks were released in the backup process. Recoverability requires update locks be held to commit point. Repeatability requires that all transactions observe the consistency lock protocol.

The normal (i.e. trouble free) operation of a data base system can be described in terms of an initial consistent state S0 and a schedule of transactions mapping the data base into a final consistent state S3 (see Figure). S1 is a checkpoint state, since transactions are in progress, S1 may be inconsistent. A system crash leaves the database in state S2. Since transactions T3 and T5 were in progress at the time of the crash. S2 is potentially inconsistent. System recovery amounts to bringing the database to a new consistent state in one of the following ways:

a. Starting from state S2, undo all actions of transactions in-progress at the time of the crash.

b. Starting from state S1 first undo all actions of transactions in progress at the time of the crash (i.e. actions of T3 before S1) and then redo all actions of transactions that completed after S1 and before the crash (i.e. actions of T2 and T4 after S1.)

c. Starting at SO redo all transactions that completed before the crash.

Observe that (a) and (c)are degenerate cases of (b).

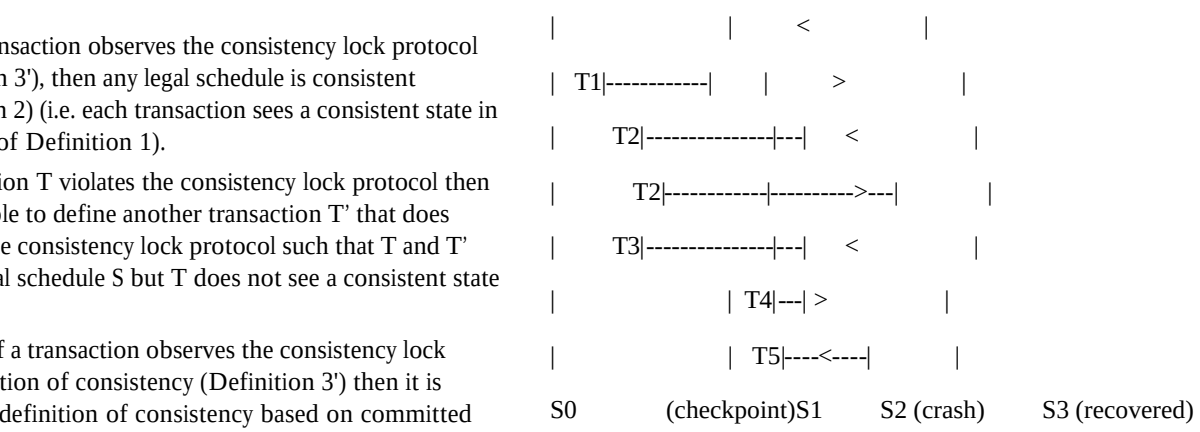


Figure. System states, SO is initial state, S1 is checkpoint state, S2 is a crash and S3 is the state that results in the absence of a crash.

If some transaction does not hold update locks to commit point then:

- Undoing the transaction may deadlock (because undo must reacquire the locks in order to perform undo.)
- Undoing a transaction may loose updates (because an update may have been applied to the output of the undone transaction, but backup will restore the entity to its original value.)

Consequently, backup may cascade: backing up one transaction may require backing up another. (Randall calls this the domino effect.) If for example, T3 writes a record, r, and then T4 further updates r then undoing T3 will cause the update of T4 to r to be lost. This situation can only arise if some transaction does not hold its write locks to commit point. For these reasons all known data management systems (which support concurrent updaters) require that all transactions hold their update locks to commit point.

On the other hand,

- If all the transactions hold all update locks to commit point, then system recovery loses no updates of complete transactions. However there may be no schedule that would give the same result because some transactions may have read outputs of undone transactions (dirty reads).
- If all the transactions observe the consistency lock protocol, then the recovered state is consistent and derives from the schedule obtained from the original system schedule by deleting incomplete transactions, Note that consistency prevents read dependencies on transactions which might be undone by system recovery. The schedule obtained by considering only the actions of completed transactions produces the recovered state.

Transaction crash gives rise to transaction backup that has properties analogous to system recovery.

5.7.5. Lower Degrees of Consistency

Most systems do not provide consistency as outlined here. Typically they do not hold read locks to EOT so that R->W->R dependencies are not precluded. Very primitive systems sometimes set no read locks at all, rather they only set update locks so as to avoid lost update and deadlock during backout. We have characterized these lock protocols as degree 2 and degree 1 consistency respectively and have studied them extensively (see "Granularity of locks and degrees of consistency in a shared data base", Gray, Lorie, Putzolu, and Traiger. in *Modeling & Data Base Systems*, North Holland Publishing (1976).) I believe that the lover degrees of consistency are a bad idea, but several of my colleagues disagree. The motivation of the lower degrees is performance. If less is locked, then less computation and storage is consumed. Furthermore, if less is locked, concurrency is increased since fewer conflicts appear. (Note that minimizing the number of explicit locks set motivates the granularity lock scheme of the next section.)

5.7.5. Lock Granularity

An important issue that arises in the design of a system is the choice of lockable units (i.e. the data aggregates which are atomically locked to insure consistency.) Examples of lockable units are areas, files, individual records, field values, and intervals of field values.

The choice of lockable units presents a tradeoff between concurrency and overhead, which is related to the size or granularity of the units themselves. On the one hand, concurrency is increased if a fine lockable unit (for example a record or field) is chosen. Such unit is appropriate for a "simple" transaction that accesses few records. On the other hand, a fine unit of locking would be costly for a "complex" transaction that accesses many records. Such a transaction would have to set and reset many locks, incurring the computational overhead of many invocations of the lock manager, and the storage overhead of representing many locks. A coarse lockable unit (for example a file) is probably convenient for a transaction that accesses many records. However, such a coarse unit discriminates against transactions that only want to lock one member of the file. From this discussion it follows that it would be desirable to have lockable units of different granularities coexisting in the same system.

The following presents a lock protocol satisfying these requirements and discusses the related implementation issues of scheduling, granting and converting lock requests.

5.7.6.1. Hierarchical Locks

We will first assume that the set of resources to be locked is organized in a hierarchy. Note that this hierarchy is used in the context of a collection of resources and has nothing to do with the data model used in a data base system. The hierarchy of the following figure may be suggestive. We adopt the notation that each level of the hierarchy is given a node type that is a generic name for all the node instances of that type. For example, the database has nodes of type area as its immediate descendants, each area in turn has nodes of type file as its immediate descendants and each file has nodes of type record as its immediate descendants in the hierarchy. Since it is a hierarchy, each node has a unique parent.

DATA BASE

|

AREAS

|

FILES

|

RECORDS

Figure 1: A sample lock hierarchy.

Each node of the hierarchy can be locked. If one requests exclusive access (X) to a particular node, then when the request is granted, the requestor has exclusive access to that node and implicitly to each of its descendants. If one requests shared access (S) to a particular node, then when the request is granted, the requestor has shared access to that node and implicitly to each descendant of that node. These two access modes lock an entire sub-tree rooted at the requested node.

Our goal is to find some technique for implicitly locking an entire sub-tree. In order to lock a sub-tree rooted at node R in share or exclusive mode it is important to prevent locks on the ancestors of R that might implicitly lock R and its descendants in an incompatible mode. Hence a new access mode, intention mode (I), is introduced. Intention mode is used to "tag" (lock) all ancestors of a node to be locked in share or exclusive mode. These tags signal the fact that locking is being done at a "finer" level and thereby prevents implicit or explicit exclusive or share locks on the ancestors.

The protocol to lock a sub-tree rooted at node R in exclusive or share

The protocol to lock a sub-tree rooted at node R in exclusive or share mode is to first lock all ancestors of R in intention mode, and then to lock node R in exclusive or share made. For example, using the figure above, to lock a particular file one should obtain intention access to the database, to the area containing the file, and then request exclusive (or share) access to the file itself. This implicitly locks all records of the file in exclusive (or share) mode.

5.7.6.2. Access Modes And Compatibility

We say that two lock requests for the same node by two different transactions are compatible if they can be granted concurrently. The mode of the request determines its compatibility with requests made by other transactions.. The three modes X, S, and I are incompatible with one another, but distinct S requests may be granted together and distinct I requests may be granted together,.

The compatibilities among modes derive from their semantics. Share mode allows reading but not modification of the corresponding resource by the requestor and by other transactions. The semantics of exclusive mode is that the grantee may read and modify the resource, but no other transaction may read or modify the resource while the exclusive lock is set. The reason for dichotomizing share and exclusive access is that several share requests can be granted concurrently (are compatible) whereas an exclusive request is not compatible with any other request. Intention mode was introduced to be incompatible with share and exclusive mode (to prevent share and exclusive locks).

However, intention mode is compatible with itself since two transactions having intention access to a node will explicitly lock descendants of the node in X, S or I mode and thereby will either be compatible with one another or will be scheduled on the basis of their requests at the finer level. For example, two transactions can simultaneously be granted the database and some area and some file in intention mode. In this case their explicit locks on particular records in the file will resolve any conflicts among them.

The notion of intention mode is refined to intention share mode (IS) and intention exclusive (IX) for two reasons: intention share mode only requests share or intention share locks at the lower nodes of the tree (i.e. never requests an exclusive lock below the intention share node.) Hence IS mode is compatible with S mode. Since read only is a common form of access it will be profitable to distinguish this for greater concurrency. Secondly, if a transaction has an intention share lock on a node it can convert this to a share lock at a later time, but one cannot convert an intention exclusive lock to a share lock on a node, Rather to get the combined rights of share node and intention exclusive mode one must obtain an X or SIX mode lock. (This issue is discussed in the section on rerequests below).

We recognize one further refinement of modes, namely share and intention exclusive mode (SIX). Suppose one transaction wants to read an entire sub-tree and to update particular nodes of that sub-tree. Using the modes provided so far it would have the options of: (a) requesting exclusive access to the root of the sub-tree and doing no further locking or (b) requesting intention exclusive access to the root of the sub-tree and explicitly locking the lover nodes in intention; share or exclusive mode. Alternative (a) has low concurrency. If only a small fraction of the read nodes are updated then alternative (b) has nigh locking overhead, The correct access mode would be share access to the sub-tree thereby allowing the transaction to read all nodes of the sub-tree without further locking and intention exclusive access to the sub-tree thereby allowing the transaction co set exclusive locks on those nodes in the sub-tree that

are to be updated and IX or SIX locks on the intervening nodes. SIX mode is introduced because this is a common case. It is compatible with IS mode since other transactions requesting IS mode will explicitly lock lower nodes in IS or S mode thereby avoiding any updates (IX or X mode) produced by the SIX mode transaction. However SIX mode is not compatible with IX, S, SIX or X mode requests.

The table below gives the compatibility of the request modes, where null mode (NL) represents the absence of a request.

	NL	IS	IX	S	SIX	X
IS	YES	YES	YES	YES	YES	NO
IX	YES	YES	YES	NO	NO	NO
S	YES	YES	NO	NO	NO	NO
SIX	YES	YES	NO	NO	NO	NO
X	YES	NO	NO	NO	NO	NO

Table 1. Compatibilities among access modes.

To summarize, we recognize six modes of access to a resource:

NL: Gives no access to a node, (i.e. represents the absence of a request of a resource.)

IS: Gives intention share access to the requested node and allows the requestor to lock descendant nodes in S or IS mode. (It does no implicit locking.)

IX: Gives intention exclusive access to the requested node and allows the requestor to explicitly lock descendants in X, S, SIX, IX or IS mode. (It does no implicit locking.)

S: Gives shared access to the requested node and to all descendants of the requested node without setting further locks. (It implicitly sets S locks on all descendants of the requested node.)

SIX: Gives share and intention exclusive access to the requested node. In particular, it implicitly locks all descendants of the node in share mode and allows the requestor to explicitly lock descendant nodes in X, SIX or IX mode.)

X: Gives exclusive access to the requested node and to all descendants of the requested node without setting further locks. (It implicitly sets X Locks on all descendants. Locking lover nodes in S or IS mode would give no increased access.)

IS mode is the weakest non-null form of access to a resource. It carries fever privileges than IX or S modes. IX mode allows IS, IX, S, SIX and X mode locks to be set on descendant nodes while S mode allows read-only access to all descendants of the node without further locking. SIX mode carries the privileges of S and of IX mode (hence the name SIX). X mode is the most privileged form of access and allows reading and writing of all descendants of a node without further locking. Hence the modes can be ranked in the partial order of privileges shown the figure below. Note that it is not a total order since IX and S are incomparable.

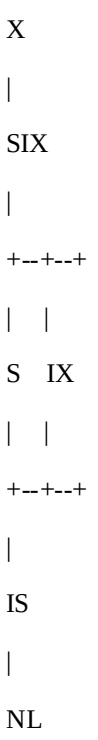


Figure 2. The partial ordering of modes by their privileges.

5.7.6.3. Rules For Requesting Nodes

The implicit locking of nodes will not work if transactions are allowed to leap into the middle of the tree and begin locking nodes at random. The implicit locking implied by the S and X modes depends on all transactions obeying the following protocol:

a. Before requesting an S or IS lock on a node, all ancestor nodes of the requested node must be held in IX or IS mode by the requestor.

b. Before requesting an X, SIX or IX lock on a node, all ancestor nodes of the requested node must be held in SIX or IX mode by the requestor.

c. Locks should be released either at the end of the transaction (in any order) or in leaf to root order. In particular, if locks are not held to end of transaction, one should not hold a lock after releasing its ancestors.

To paraphrase this, locks are requested root to leaf, and released leaf to root. Notice that leaf nodes are never requested in intention mode since they have no descendants, and that once a node is acquired in S or X mode, no further explicit locking is required at lover levels.

5.7.6.4. Several Examples

To lock record R for read:

lock database with mode =IS
lock area containing R with mode =IS
lock file containing R with mode =IS
lock record R with mode = S

Don't panic, the transaction probably already has the database, area and file lock.

To lock record R for write-exclusive access:

lock database with mode =IX
lock area containing R with mode =IX
lock file containing R with mode =IX
lock record R with mode = X

Note that if the records of this and the previous example are distinct, each request can be granted simultaneously to different transactions even though both refer to the same file.

To lock a file F for read and write access:

lock database with mode =IX
lock area containing F with mode =IX
lock file P with mode = X

Since this reserves exclusive access to the file, if this request uses the same file as the previous two examples it or the other transactions will have to wait. Unlike examples 1, 2 and 4, no additional locking need be done (at the record level).

To lock a file F for complete scan and occasional update:

lock database with node =IX
lock area containing F with mode =IX
lock file F with mode =SIX

Thereafter, particular records in F can be locked for update by locking the desired records in X mode. Notice that (unlike the previous example) this transaction is compatible with the first example. This is the reason for introducing SIX mode.

To quiesce the data base:

lock data base with mode =X.

Note that this locks everyone else out.

5.7.6.5. DIRECTED ACYCLIC GRAPHS OF LOCKS

The notions so far introduced can be generalized to work for directed acyclic graphs (DAGs) of resources rather than simply hierarchies of resources. A tree is a simple DAG. The key observation is that to implicitly or explicitly lock a node, one should lock all the parents of the node in the DAG, and so by induction lock all ancestors of the node. In particular, to lock a subgraph one must implicitly or explicitly lock all ancestors of the subgraph in the appropriate mode (for a tree there is only one parent). To give an example of a non-hierarchical structure, imagine the locks are organized as:

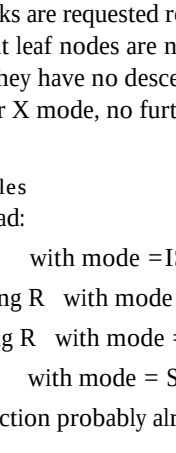


Figure 3. A non-hierarchical lock graph.

We postulate that areas are "physical" notions and that files, indices and records are logical notions. The database is a collection of areas. Each area is a collection of files and indices. Each file has a few corresponding indices in the same area. Each record belongs to some file and to its corresponding indices. A record is comprised of field values and some field is indexed by the index associated with the file containing the record. The file gives a sequential access path to the records and the index gives an associative access path to the records based on field values. Since individual fields are never locked, they do not appear in the lock graph.

To write a record A in file P with index I:

lock data base with mode =IX
lock area containing F with mode =IX
lock file F with mode =IX
lock index I with mode =IX
lock record R with mode = X

Note that all paths to record R are locked. Alternatively, one could lock F and I in exclusive mode thereby implicitly locking R in exclusive mode.

To give a more complete explanation we observe that a node can be locked explicitly (by requesting it) or implicitly (by appropriate explicit locks on the ancestors of the node) in one of five modes: IS, IX. S, SIX, X. However, the definition of implicit locks and the protocols for setting explicit locks have to be extended for DAG's as follows:

A node is implicitly granted in S mode to a transaction if at least one of its parents is (implicitly or explicitly) granted to the transaction in S, SIX or X mode. By induction, that means that at least one of the node's ancestors must be explicitly granted in S, SIX, or X mode to the transaction.

A node is implicitly granted in X mode if all of its parents are (implicitly or explicitly) granted to the transaction in X mode. By induction, this is equivalent to the condition that all nodes in some cut set of the collection of all paths leading from the node to the roots of the graph are explicitly granted to the transaction in X mode and all ancestors of nodes in the cut set are explicitly granted in IX or SIX mode.

By examination of the partial order of modes (see figure above), a node is implicitly granted in IS mode if it is implicitly granted in S mode, and a node is implicitly granted in IS, IX, S and SIX mode if it is implicitly granted in X mode.

5.7.6.6. Protocol for Requesting Locks on a Dag

a. Before requesting an S or IS lock on a node, one should request at least one parent (and by induction a path to a root) in IS (or greater) mode. As a consequence none of the ancestors along this path can be granted to another transaction in a mode incompatible with IS.
b. Before requesting IX, SIX or x mode access to a node, one should request all parents of the node in IX (or greater) mode. As a consequence all ancestors will be held in IX (or greater mode) and cannot be held by other transactions in a mode incompatible with IX (i.e. S, SIX, X).
c. Locks should be released either at the end of the transaction (in any order) or in leaf to root order. In particular, if locks are not held to the end of transaction, one should not hold a lower lock after releasing its ancestors.

To give an example using the non-hierarchical lock graph in the figure above, a sequential scan of all records in file F need not use an index so one can get an implicit share lock on each record in the file by:

lock data base with mode =IS
lock area containing F with mode =IS
lock file F with mode = S

This gives implicit S mode access to all records in P. Conversely, to read a record in a file via the index I for file P, one need not get an implicit or explicit lock on file F:

lock data base with mode =IS
lock area containing R with mode =IS
lock index I with mode = S

This again gives implicit S mode access to all records in index I (in file F). In both these cases, only one path was locked for reading.

But to insert, delete or update a record R in file F with index I one must get an implicit or explicit lock on all ancestors of R.

The first example of this section showed how an explicit X lock on a record is obtained. To get an implicit X lock on all records in a file one can simply lock the index and file in X mode, or lock the area in X mode. The latter examples allow bulk load or update of a file without further locking since all records in the file are implicitly granted in X mode.

5.7.6.7. Proof of Equivalence of the Lock Protocol

We will now prove that the described lock protocol is equivalent to a conventional one which uses only two modes (S and X), and which explicitly locks atomic resources (the leaves of a tree or sinks of a DAG).

Let G =(N, A) be a finite (directed acyclic) graph where N is the set of nodes and A is the set of arcs. G is assumed to be without circuits (i.e. there is no non-null path leading from a node n to itself). A node p is a parent of a node n, and n is a child of p if there is an arc from p to n. A node n is a source (sink), if n has no parents (no children). An ancestor of node n is any node (including n) in a path from a source to n. A node-slice of a sink n is a collection of nodes such that each path from a source to n contains at least one node of the slice. Let Q be the set of all sinks of G.

We also introduce the set pf lock modes M = {NL, IS, IX, S, SIX, X} and the compatibility matrix C : AxM-> (YES, NO) described in Table 1. Let c : AxM -> {YES, NO} be the restriction of C to m ={NL, S, X}.

A lock graph is a mapping L : N->M such that:

a. If L(n) is in {IS, S} then either n is a source or there exists a parent p of n such that L(p) is in { IS, IX, S, SIX, X). By induction there exists a path from a source to n such that L(ni) takes only values in {IS, IX, S, SIX, X} on all nodes

ni of that path. Equivalently, L(ni) is not equal to NL on the path.

b. If L(n) is in {IX, SIX, X} then either n is a root, or for all parents p1,..., pk of n we have L(pi) is in { IX. SIX, X} for i = 1,..., k . By induction L takes only values in {IX, SIX, X} on all the ancestors of n.

The interpretation of a lock-graph is that it gives a map of the explicit locks held by a particular transaction observing the six state lock protocol described above. The notion of projection of a lock-graph is now introduced to model the set of implicit locks on atomic resources acquired by a transaction.

The projection of a lock-graph L is the mapping p: Q->m (from sinks to modes) constructed as follows:

a. p(n)=X if there exists a node-slice {n1,.. , ns} of N such that p(ni)=X for each node in the slice.

b. p(n)=S if (a) is not satisfied and there exists an ancestor na of N such that p(na) is in {S, SIX, X}.

c p(n)=NL if (a) and (b) are not satisfied.

Two lock-graphs L1 and L2 are said to be compatible if C(L1(n), L2(n)) = YES for all n in N. Similarly two projections p1 and p2 are compatible if c(p1(n), p2(n)) = YES for all sink nodes n in Q.

Theorem: If two lock-graphs L1 and L2 are compatible, then their projections P1 and P2 are compatible. In other words if the explicit locks set by two transactions do not conflict then also the three-state locks implicitly acquired do not conflict.

Proof: Assume that L1 and L2 are incompatible. We want to prove that P1 and P2 are incompatible. By definition of compatibility there must exist a sink n such that L1(n)=X and L2(n) is in {S, X} (or vice versa). By definition of projection, there must exist a node-slice {n1,..., ns} of N such that L1(n1)=...=L1(ns)=X. Also there must exist an ancestor na of n such that L2(na) is in {S, SIX, X}. From the definition of lock-graph there is a path Path1 from a source to na on which L2 does not take the value NL. If Path1 intersects the node-slice at ni then L1 and L2 are incompatible since L1(ni)=X which is incompatible with the non-null value of L2(ni). Hence the theorem is proved.

Alternatively there is a path Path2 from na to the sink n that intersects the node-slice at ni. From the definition of lock-graph L1 takes a value in {IX, SIX, X} on all ancestors of ni. In particular L1(na) is in {IX, SIX, X}. Since L2(na) is in {S, SIX, X} we have C(L1(na), L2 (na))=NO.

Q. E. D.

5.7.7. Lock Management Pragmatics

Thus far we have discussed when to lock (lock before access and hold locks to commit point) and why to lock (to guarantee consistency and to make recovery possible without cascading transaction backup,) and what to lock (lock at a granularity that balances concurrency against instruction overhead in setting locks.) The remainder of this section will discuss issues associated with how to implement a lock manager.

5.7.7.1. The Lock Manager Interface

This is a simple version of the System R lock manager.

5.7.7.1.1. Lock Actions

Lock manager has two basic calls:

LOCK <lock>, <mode>, <class>, <control>

where <lock> is the resource name (in System R for example an eight byte name). <mode> is one of the lock modes { S | X | SIX | IX | IS }. <class> is a notion described below. <control> can be either WAIT in which case the call is synchronous and waits until the request is granted or is cancelled by the deadlock detector, or <control> can be TEST in which case the request is canceled if it cannot be granted immediately.

UNLOCK <lock>, <class>

releases the specified lock in the specified class. If the <lock> is not specified, all locks held in the specified class are released.

5.7.7.1.1.2. Lock Names

The associa tion between lock names and objects is purely a convention. Lock manager associates no semantics with names. Generally the first byte is reserved for the subsystem (component) identifier and the remaining seven bytes name the object.

For example, data manager might use bytes (2... 4) for the file name and bytes (5... 7) for the record name in constructing names for record locks.

Since there are so many locks, one only allocates those with non-null queue headers. (i.e. free locks occupy no space.) Setting 1 lock consists of hashing the lock name into a table. If the header already exists, the request enqueues on it, otherwise the request allocates the lock header and places it in the hash table. When the queue of a lock becomes empty, the header is deallocated (by the unlock operation).

5.7.7.1.1.2. Lock Classes

Many operations acquire a set of locks. If the operation is successful, the locks should be retained. If the operation is unsuccessful, or when the operation commits, the locks should be released. In order to avoid double bookkeeping, the lock manager allows users to name sets of locks (in the new DBTG proposal these are called keep lists, in IMS program isolation these are called *Q class locks).

For each lock held by each process, lock manager keeps a list of <class, count> pairs. Each lock request for a class increments the count for that class. Each unlock request decrements the count. When all counts for all the lock's classes are zero then the lock is not held by the process.

5.7.7.1.4. Latches

Lock manager needs a serialization mechanism to perform its function (e.g. inserting elements in a queue or hash chain). It does this by implementing a lower level primitive called a latch. Latches are semaphores. They provide a cheap serialization mechanism without providing the expensive features like deadlock detection, class tracking, and modes of sharing (beyond S or X). They are used by lock manager and by other performance critical managers (notably buffer manager and log manager).

5.7.7.1.5. Performance of Lock Manager

Lock manager is about 3300 lines of (PL/1 like) source code. It depends critically on the Compare and Swap logic provided by the multiprocessor feature of System 370. It comprises three percent of the code and about ten percent of the instruction execution of a program in System R (this may vary a great deal.) A lock-unlock pair currently costs 350 instructions but if these notes are ever finished, this will be reduced to 120 instructions (this should reduce its slice of the execution pie.) A latch-unlatch pair requires 10 instructions (they expand in-line). (Initially they required 120 instructions but a careful redesign improved this dramatically.)

5.7.7.2. Scheduling And Granting Requests

Thus far we have described the semantics of the various request modes and have described the protocol that requestors must follow. To complete the discussion we discuss how requests are scheduled and granted.

The set of all requests for a particular resource are kept in a queue sorted by some fair scheduler. By "fair" we mean that no particular transaction will be delayed indefinitely. First-in first-out is the simplest fair scheduler and we adopt such a scheduler for this discussion modulo deadlock preemption decisions.

The group of mutually compatible requests for a resource appearing at the head of the queue is called the granted group. All these requests can be granted concurrently. Assuming that each transaction has at most one request in the queue then the compatibility of two requests by different transactions depends only on the modes of the requests and may be computed using Table 1. Associated with the granted group is a group mode is the supremum mode of the members of the group which is computed using Figure 2 or Table 3. Table 2 gives a list of the possible types of requests that can coexist in a group and the corresponding mode of the group.

Table 2. Possible request groups and their group mode. Set brackets indicate that several such requests may be present.

Request Modes	Group Mode
X	X
{ SIX, {IS} }	SIX
{ S, {S}, {IS} }	S
{ IX, {IX}, {IS} }	IX
{ IS, {IS} }	IS

The figure below depicts the queue for a particular resource, showing the requests and their modes. The granted group consists of five requests and has group mode IX. The next request in the queue is for S mode that is incompatible with the group mode IX and hence must wait.

```
**********************************
*   GRANTED GROUP: GROUPMODE =IX *
* |IS|-|IX|-|IS|--|IS|-|IS|-|IS|-*-|S|-|IS|-|X|-|IS|-|IX|
**********************************
```

Figure 5. The queue of requests for a resource.

When a new request for a resource arrives, the scheduler appends it to the end of the queue. There are two cases to consider: either someone is already waiting, or all outstanding requests for this resource are granted (i.e. no one is waiting). If waiters exist, then the request cannot be granted and the new request must wait. If no one is waiting and the new request is compatible with the granted group mode then the new request can be granted immediately. Otherwise the new request must wait its turn in the queue, and in the case of deadlock it may preempt some incompatible requests in the queue. (Alternatively the new request could be canceled. In Figure 5 all the requests decided to wait.). When a particular request leaves the granted group, the mode of the group may change. If the mode of the first waiting request is compatible with the new mode of the granted group, then the waiting request is granted. In Figure 5 if the IX request leaves the group, then the group mode becomes IS which is compatible with S and so the S request may be granted. The new group mode will be S and since this is compatible with the IS mode. The IS requests following the S request may also join the granted group. This produces the situation depicted in Figure 6.

```
**********************************
*   GRANTED GROUP: GROUPMODE = S *
* |IS|-|IS|-|IS|–|IS|-|S|-|IS|--*-|X|-|IS|-|IX|
**********************************
```

Figure 6. The queue after the IX request is released.

The X request of Figure 6 will not be granted until all requests leave the granted group since it is not compatible with any of them.

5.7.7.3. Conversions

A transaction might re-request the same resource for several reasons: Perhaps it has forgotten that it already has access to the record; after all, if it is setting many locks it may be simpler to just always request access to the record rather than first asking itself "have I seen this record before". The lock manager has all the information to answer this question and it seems wasteful to duplicate. Alternatively, the transaction may know it has access to the record, but wants to increase its access mode (for example from S to X mode if it is in a read, test, and sometimes update scan of a file). So the lock manager must be prepared for re-requests by a transaction for a lock. We call such re-requests conversions.

When a request is found to be a conversion, the old (granted: node of the requestor to the resource and the newly requested mode are compared using Table 3 to compute the new mode

which is the supremum of the old and the requested mode (see Figure 2.)

Table 3. The new mode given the requested and old mode.

	IS	IX	S	SIX	X
IS	IS	IX	S	SIX	X
IX	IX	IX	SIX	SIX	X
S	S	SIX	S	SIX	X
SIX	SIX	SIX	SIX	SIX	X
X	X	X	X	X	X

So for example, if one has Ix mote and requests S mode then the new mode is SIX.

If the new mode is equal to the old mode (note it is never less than the old mode) then the request can be granted immediately, and the granted mode is unchanged. If the new mode is compatible with the group mode of the other members of the granted group (a requestor is always compatible with himself), then again the request can be granted immediately. The granted mode is the new mode and the group mode is recomputed using Table 2. In all other cases, the requested conversion must wait until the group mode of the other granted requests is compatible with the new mode. Note that this immediate granting of conversions over waiting requests is a minor violation of fair scheduling. If two conversions are waiting, each of which is incompatible with an already granted request of the other transaction, then a deadlock exists and the already granted access of one must be preempted. Otherwise there is a way of scheduling the waiting conversions: namely, grant a conversion when it is compatible with all other granted nodes in the granted group. (Since there is no deadlock cycle, this is always possible.)

The following example may help to clarify these points. Suppose the queue for a particular resource is:

```
*****************************
* GROUPMODE =IS          *
* | IS|---|IS|-------------------------
```

Figure 7. A simple queue.

Now suppose the first transaction wants to convert to X mode. It must wait for the second (already granted)request to leave the queue. If it decides to wait then the situation becomes:

```
*****************************
* GROUPMODE = IS         *
* |IS<-X|---|IS|----------------------
*****************************
```

Figure 8. A conversion to X mode waits.

No new request may enter the granted group since there is now a conversion request waiting. In general, conversions are scheduled before new requests. If the second transaction now converts to IX, SIX, or S mode it may be granted immediately since this does not conflict with the granted (IS) mode of the first transaction. When the second transaction eventually leaves the queue, the first conversion can be made:

```
*****************************
* GROUPMODE = X          *
* |X|---------------------------------
*****************************
```

Figure 9. One transaction leaves and the conversion is granted. However, if the second transaction tries to convert to exclusive mode one obtains the queue:

```
*****************************
* GROUPMODE = IS         *
* |IS<-X|---|IS<-X|----------------------
*****************************
```

Figure 10. Two conflicting conversions are waiting.

Since X is incompatible with IS (see Table 1). this situation implies that each transaction is waiting for the other to leave the queue (i.e. deadlock) and so one transaction must be preempted. In all other cases (i.e. when no cycle exists) there is a way to schedule the conversions so that no already granted access is violated.

5.7.7.4. Deadlock Detection

One issue the lock manager must deal with is deadlock. Deadlock consists of each member of a set of transactions waiting for some other member of the set to give up a lock. Standard lore has it that one can have timeout, or deadlock-prevention, or deadlock detection.

Timeout causes waits to be denied after some specified interval. It has the property that as the system becomes more congested, more and more transactions time out (because time runs slower and because more resources are in use so that one waits more). Also timeout puts an upper limit on the duration of a transaction. 1n general the dynamic properties of timeout make it acceptable for a lightly loaded system but inappropriate for a congested system.

Deadlock prevention is achieved by: requesting all locks at once, or requesting locks in a specified order, or never waiting for a lock, or . . . In general deadlock prevention is a bad deal because one rarely knows what locks are needed in advance (consider looking something up in an index,) and consequently, one locks too much in advance.

Although some situations allow deadlock prevention, general systems tend to require deadlock detection. IMS, for example, started with a deadlock prevention scheme (intent scheduling) but was forced to introduce a deadlock detection scheme to increase concurrency (Program Isolation).

Deadlock detection and resolution is no big deal in a data management system environment. The system already has lots of facilities for transaction backup so that it can deal with other sorts of errors. Deadlock simply becomes another (hopefully infrequent) source of backup. As will be seen, the algorithms for detecting and resolving deadlock are not complicated or time consuming.

The deadlock detection-resolution scenario is:

- Detect a deadlock.
- Pick a victim (a lock to preempt from a process.)
- Back out a victim that will release lock.
- Grant a waiter.
- (optionally) Restart victim.

Lock manager is only responsible for deadlock detection and victim selection, Recovery management implements transaction backup and controls restart logic.

5.7.7.4.1. How To Detect Deadlock

There are many heuristic ways of detecting deadlock (e. g. linearly order resources or processes and declare deadlock if ordering is violated by a wait request.)

Here we restrict ourselves to algorithmic solutions.

The detection of deadlock may be cast in graph-theoretic terms. We introduce the notion of the wait-for graph.

- The nodes of the graph are transactions and locks.
- The edges of the graph are directed and are constructed as follows:
  - If lock L is granted to transaction T then draw an edge from L to T.
  - If transaction T is waiting for transaction L then draw an edge from T to L.

At any instant, there is a deadlock if and only if the wait-for graph has a cycle. Hence deadlock detection becomes an issue of building the wait-for graph and searching it for cycles.

Often this “transaction waits for lock waits for transaction” graph can be reduced to a smaller “transaction waits for transaction” graph. The larger graph need be maintained only if the identities of the locks in the cycle are relevant. I know of no case where this is required.

5.7.7.4.2. When To Look For Deadlock

One could opt to look for deadlock:

- Whenever anyone waits.
- Periodically.
- Never.

One could look for deadlock continuously. Releasing a lock or being granted a lock never creates a deadlock. So one should never look for deadlock more frequently than when a wait occurs.

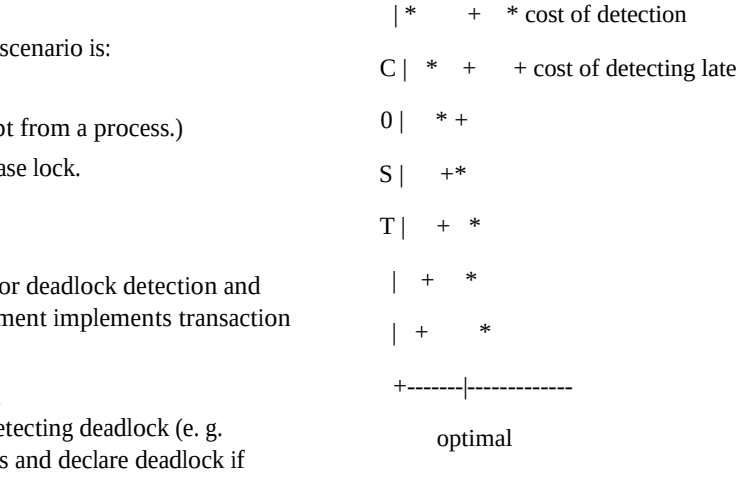
The cost of looking for deadlock every time anyone waits is:

- Continual maintenance of the wait-for graph.
- Almost certain failure since deadlock is (should be) rare (i.e. wasted instructions).

The cost of periodic deadlock detection is:

- Detecting deadlocks late.

By increasing the period one decreases the cost of deadlock and the probability of successfully finding one. For each situation there should be an optimal detection period.



Never testing for deadlocks is much like periodic deadlock detection with a very long period. All systems have a mechanism to detect dead programs (infinite loops, wait for lost interrupt,...) This is usually a part of allocation and resource scheduling. It is probably outside and above deadlock detection.

Similarly, if deadlock is very frequent, the system is thrashing and the transaction scheduler should stop scheduling new work, and perhaps abort some current work to reduce this thrashing. Otherwise the system is likely to spend the majority of its time backing up.

5.7.7.4.3. What To Do When Deadlock Is Detected.

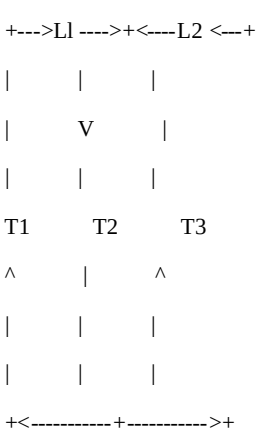
All transactions in a deadlock are waiting. The only way to get things going again is to grant some waiter. But, this can only be achieved after a lock is preempted from some holder. Since the victim is waiting, he will get the ‘deadlock’ response from lock manager rather than the ‘granted’ response.

In breaking the deadlock some set of victims will be preempted. We want to minimize the amount of work lost by these preemptions. Therefore, deadlock resolution wants to pick a minimum cost set of victims to break deadlocks.

Transaction management must associate a cost with each transaction. In the absence of policy decisions: the cost of a victim is the cost of undoing his work and then redoing it. The length of the transaction log is a crude estimate of this cost. At any rate, transaction management must provide lock management with an estimate of the cost of each transaction. Lock manager may implement either of the following two protocols:

- For each cycle, choose the minimum cost victim in that cycle.
- Choose the minimum cost cut-set of the deadlock graph.

The difference between these two options is best visualized by the picture:



If T1 and T3 have a cost of 2 and T2 has a cost of 3 then a cycle-at-a-time algorithm will choose T1 and T3 as victims; whereas, a minimal cut set algorithm will choose T2 as a victim.

The cost of finding a minimal cut set is considerably greater (seems to be NP complete) than the cycle-at-a-time scheme. If there are N common cycles, the cycle-at-a-tine scheme is at most N times worse than the minimal cut set scheme. So it seems that the cycle-at-a-time scheme is better.

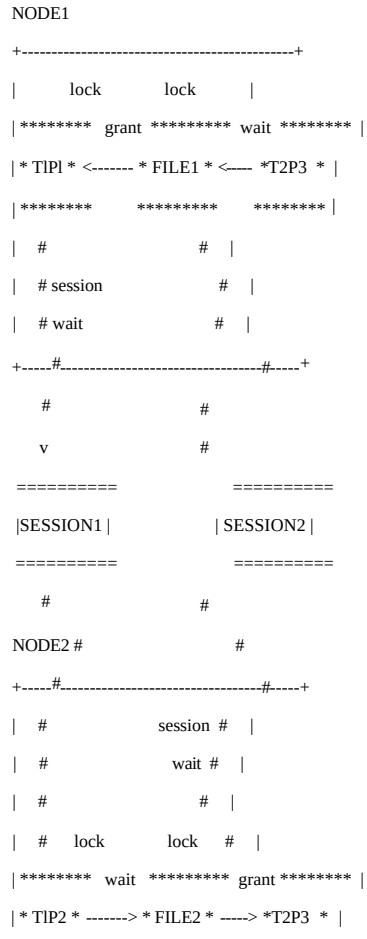
5.7.7.4.4. Lock Management in a Distributed System.
To repeat the discussion in the section OIL distributed transaction management, if a transaction wants to do work at a new node, some process of the transaction must request that the node construct a cohort and that the cohort go into session with the requesting process (see section on data communications for a discussion of sessions.) The picture below shows this.

A cohort carries both the transaction name T1 and the process name (in NODE1 the cohort of T1 is process P2 and in NODE2 the cohort of T1 is process P6.)

The two processes can nor converse and carry out the work of the transaction. If one process aborts, they should both abort, and if one process commits they should both commit.

The lock manager of each node can keep its lock tables in any form it desires. Further, deadlock detectors running in each node may use any technique they like to detect deadlocks among transactions that run exclusively in that node. We call such deadlocks local deadlocks. However, just because there are no cycles in the local wait-for graph, does not mean that there are no cycles. Gluing acyclic local graphs together might produce a graph with cycles (See the example bellow.) So, the deadlock detectors of each node will have to agree on a common protocol in order to handle deadlocks involving distributed transactions. We call such deadlocks global deadlocks.

Inspection of the following figure may help to understand the nature of global deadlocks. Note that transaction T1 has two processes Pl and P2 in nodes 1 and 2 respectively. Pl is session-waiting for its cohort P2 to do some work. P2, in the process of doing this work, needed access to FILE2 in NODE2. But FILE2 is locked exclusive by another process (P4 of NODE2) so P2 is in lock wait state. Thus the transaction T1 is waiting for FILE2. Now Transaction T2 is in a similar state, one of its cohorts is session waiting for the other which in turn is lock waiting for FILE1. In fact transaction T1 is waiting for FILE2, which is granted to transaction T2 that is waiting for file FILE1 that is granted to transaction T1. A global deadlock if you ever saw one.



The notion of wait-for graph must be generalized to handle global deadlock. The nodes of the graph are processes and resources (sessions are resources). The edges of the graph are constructed as follows:

- Draw a directed edge free a process to a resource if
  - The process is in lock wait for the resource,
  - Or the process is in session-wait for the resource (session)-
- Draw a directed edge from a resource to a process if
  - The resource is lock granted to the process
  - Or it is a session of the process and the process is not in session-wait on it.

A local deadlock is a

lock wait ->. . . .-> lockwait cycle.

A global deadlock is a

lockwait->... -> sessionwait -> lockwait ->...-> sessionwait cycle

5.7.7.5.1. How To Find Global Deadlocks

The finding of local deadlocks has already been described. To find global deadlocks, a distinguish task, called the global deadlock detector is started in some distinguished node. This task is in session with all local deadlock detectors and coordinates the activities of the local deadlock detectors. This global deadlock detector can run in any node, but probably should be located to minimize its communication distance to the lock managers.

Each local deadlock Detector needs to find all potential global deadlock paths in his node. In the previous section it was shown that a global deadlock cycle has the form:

lockwait ->... -> sessionwait ->lockwait ->...-> sessionwait ->

So each local deadlock detector periodically enumerates all

session -> lockwait ->...-> sessionwait

paths in his node by working backwards from processes that are in sessionwait (as opposed to console wait, disk wait, processor wait,…) Starting at such a process it sees if some local process is lock waiting for this process. If so the deadlock detector searches backwards looking for some process which has a session in progress.

When such a path is found, the following information is sent to the global deadlock detector:

- Sessions and transactions at endpoints of the path and their local preemption costs.
- The minimum cost transaction in the path and his local pre-emption cost.

(It may make sense to batch this information to the global detector.) Periodically, the global deadlock detector:

- collects these messages,
- glues all these paths together by matching up sessions, and
- enumerates cycles and selects victims just as in the local deadlock detector case.

One tricky point is that the cost of a distributed transaction is the sum of the costs of its cohorts. The global deadlock detector approximates this cost by summing the costs of the cohorts of the transaction known to it (not all cohorts of a deadlocked transaction will be in known to the global deadlock detector.)

When a victim is selected, the lock manager of the node the victim is waiting in is informed of the deadlock. The local lock manager in turn informs the victim with a deadlock return.

The use of periodic deadlock detection (as opposed to detection every time anyone waits)is even more important for a distributed system than for a centralized system, The cost of detection is much higher in a distributed system. This will alter the intersection of the cost of detection and cost of detecting late curves.

If the network is really large the deadlock detector can be staged. That is, we can look for deadlock among four nodes, then among sixteen nodes, and so on.

If one node crashes, then its partition of the system is unavailable.

In this case, its cohorts in other nodes can wait for it to recover or they can abort. If the down node happens to house the global lock manager then no global deadlocks will be detected until the node recovers. If this is uncool, then the lock managers can nominate a new global lock manager whenever the current one crashes. The new manager can run in any node that can be in session with all other nodes. The new global lock manager collects the local graphs and goes about gluing them together, finding cycles, and picking victims.

5.7.7.6. Relationship A Operating System Lock Manager

Most operating systems provide a lock manager to regulate access to files and other system resources. This lock manager usually supports a limited set of lock names, the modes: share, exclusive and beware, and has some form of deadlock detection. These lock managers are usually net prepared for the demands of a data management system (fast calls, lots of locks, many modes, lock classes,.,.)The basic lock manager could be extended and refined and in time that is what will happen.

There is a big problem about having two lock managers in the same host. Each may think it has no deadlock but if their graphs are glued together a “global” deadlock exists. This makes it very difficult to build on top of the basic lock manager.

5.7.7.7. The Convoy Phenomenon: Preemptive Scheduling is Bad

Lock manager has strong interactions with the scheduler. Suppose that there are certain high traffic shared system resources. Operating on these resources consists of locking them, altering them and then unlocking them (the buffer pool and log are examples of this.) These operations are designed to be very fast so that the resource is almost always free. In particular the resource is never held during an I/O operation, For example, the buffer manager latch is acquired every 1000 instructions and is held for about 50 instructions.

If the system has no preemptive scheduling then on a uniprocessor, then when a process begins the resource is free and

when he completes the resource is free (because he does not hold it when he does I/O or yields the processor.) On a multiprocessor, if the resource is busy, the process can sit in a busy wait until the resource is free because the resource is known to be held by others for only a short time.

If the basic system has a preemptive scheduler, and if that scheduler preempts a process holding a critical resource (e. g. the log latch) then terrible things happen: All other processes waiting for the latch will dispatched before this process is dispatched, and because the resource is high traffic each of these processes requests and waits for the resource. Ultimately the holder of the resource is re-dispatched and he almost immediately grants the latch to the next waiter. But because it is high traffic, the process almost immediately rerequests the latch (i.e. about 1000 instructions later.) Fair scheduling requires that he wait, so he goes on the end of the queue waiting for those ahead of him. This queue of waiters is called a convoy. It is a stable phenomenon: once a convoy is established it persists for a very long time.

We (System R) have found several solutions to this problem. The obvious solution is to eliminate such resources. That is a good idea, and can be achieved to some degree by refining the granularity of the lockable unit (e-g, twenty buffer manager latches rather than just one.) However, if a convoy ever forms on any of these latches it will be stable so that is not a solution. I leave it as an exercise for the reader to find a better solution to the problem.

Engles, "Currency and Concurrency in the COBOL Data Base Facility", in *Modeling in Data Base Management Systems,* Nijssen editor, North Holland, 1976 (A nice discussion of how locks are used.)

Eswaran et. al. "On the Notions of Consistency and Predicate Locks in a Relational Database System." CACM, Vol. 19, No. 11, November 1976. (Introduces the notion of consistency, ignore the stuff on predicate locks.)

"Granularity of Locks and Degrees of Consistency in a Shared Data Base", in *Modeling in Data Base Management Systems*, Nijssen editor, North Holland, 1976 (This section is a condensation and then elaboration of this paper. Hence Franco Putzolu and Irv Traiger should be considered co-authors of this section.)

5.8. Recovery Management

5.8.1. Model of Errors

In order to design a recovery system, it is important to have a clear notion of what kinds of errors can be expected and what their probabilities are. The model of errors below is inspired by the presentation by Lampson and Sturgis in "Crash Recovery in a Distributed Data Storage System", which may someday appear in the CACM.

We first postulate that all errors are detectable. That is, if r. o one complains about a situation, then it is OK.

5.8.1.1. Model of Storage Errors

Storage comes in three flavors with independent failure modes and increasing reliability:

- Volatile storage: paging space and main memory,
- On-Line Non-volatile Storage: disks, usually survive crashes. Is more reliable than volatile storage.
- Off-Line Non-volatile Storage: Tape archive. Even more reliable than disks.

To repeat, we assume that these three kinds of storage have independent failure modes.

The storage is blocked into fixed length units called pages that are the unit of allocation and transfer.

Any page transfer can have one of three outcomes:

1. Success (target gets new value)
2. Partial failure (target is a ness)
3. Total failure (target is unchanged)

Any page may spontaneously fail. That is a spec of dust may settle on it or a black hole may pass through it so that it no longer retains its original information,

One can always detect whether a transfer failed or a page spontaneously failed by reading the target page at a later time. (This can be made more and more certain by adding redundancy to the page.)

Lastly, The probability that N "independent" archive pages fail is negligible. Here we choose N=2, (This can be made more and more certain by choosing larger and larger N.)

5.8.1.2. Model of Data Communications Errors

Communication traffic is broken into units called messages via sessions.

The transmission of a message has one of three possible outcomes:

1. Successfully received.
2. Incorrectly received.
3. Not received.

The receiver of the message can detect whether he has received a particular message and whether it gas correctly received.

For each message transmitted, there is a non-zero probability that it will be successfully received.

It is the job of recovery manager to deal with these storage and transmission errors and correct them. This model of errors is implicit in what follows and will appear again in the examples at the end of the section.

5.8.2. Overview of Recovery Management

A transaction is begun explicitly when a process is allocated or when an existing process issues BEGIN_TRANSACTION. When a transaction is initiated, recovery manager is invoked to allocate the recovery structure necessary to recover the transaction. This process places a capability for the COMMIT, SAVE, and BACKUP calls of recovery manager in the transaction's capability list.

Thereafter, all actions by the transaction on recoverable data are recorded in the recovery log, using log manager. In general, each action performing an update operation should write an undo-log record and a redo-log record in the transaction's log. The undo log record gives the old value of the object and the redo log record gives the new value (see below).

At a transaction save point, recovery manager records the save point identifier, and enough information so that each component of the system could be backed up to this point.

In the event of a minor error, the transaction may be undone to a save point in which case the application (on its next or pending call) is given feedback indicating that the data base system has amnesia about all recoverable actions since that save point. If the transaction is completely backed-up (aborted), it may or may not be restarted depending on the attributes of the transaction and of its initiating message.

If the transaction completes successfully (commits), then (logically) it is always redone in case of a crash. On the other hand, if it is in-progress at the time of the local or system failure, then the transaction is logically undone (aborted).

Recovery manager must also respond to the following kinds of failures:

- Action failure: a particular call cannot complete due to a foreseen condition. In general the action undoes itself (cleans up its component)and then returns to the caller. Examples of this are bad parameters, resource limits, and data not found.
- Transaction failure: a particular transaction cannot proceed and so is aborted. The transaction may be reinitiated in some cases. Examples of such errors are deadlock, timeout, protection violation, and transaction-local system errors.
- System failure: a serious error is detected below the action interface. The system is stopped and restarted. Errors in critical tables, wild branches by trusted processes, operating system failures and hardware failures are sources of system failure. Most nonvolatile storage is presumed to survive a system failure.
- Media failure: a non-recoverable error is detected on some usually reliable (nonvolatile) storage device. The recovery of recoverable data from a media failure is the responsibility of the component that implements it. If the device contained recoverable data the manager must reconstruct the data from an archive copy using the log and then place the result on an alternate device. Media failures do not generally force system failure. Parity error, head crash, dust on magnetic media, and lost tapes are typical media failures. Software errors that make the media unreadable are also regarded as media errors, as are catastrophes such as fire, flood, insurrection, and operator error.

The system periodically makes copies of each recoverable object and keeps these copies in a safe place (archive). In case the object suffers a media error, all transactions with locks outstanding against the object are aborted. A special transaction (a utility) acquires the object in exclusive mode. (This takes the object "off-line".) This transaction merges an accumulation of changes to the object since the object copy was made and a recent archive version of the object to produce the most recent committed version. This accumulation of changes may take two forms: it may be the REDO-log portion of the system log, or it may be a change accumulation log that was constructed from the REDO-log portion of the system log when the system log is compressed. After media recovery, the data is unlocked and made public again.

The process of making an archive copy of an object has many varieties. Certain objects, notably IMS queue space, are recovered from scratch using an infinite redo log. Other objects, notably databases, get copied to some external media that can be used to restore the object to a consistent state if a failure occurs. (The resource may or may not be off-line while the cozy is being made.)

Recovery manager also periodically performs system checkpoint by recording critical parts of the system state in a safe spot in nonvolatile storage (sometimes called the warm start file.)

Recovery manager coordinates the process of system restart system shutdown. In performing system restart, it chooses among:

- Warm start: system shut down in controlled manner. Recovery need only locate last checkpoint record and rebuild its control structures.
- Emergency restart: system failed in uncontrolled manner. Non-volatile storage contains recent state consistent with the log. However, some transactions were in progress at time of failure and must be redone or undone to obtain most recent consistent state.
- Cold start: the system is being brought up with amnesia about prior incarnations. The log is not examined to determine previous state.

5.8.3. Recovery Protocols

All participants in a transaction, including all components understand and obey the following protocols when operating on recoverable objects:

- Consistency lock protocol.
- The DO-UNDO-REDO paradigm for log records.
- Write-Ahead-Log protocol (WAL).
- Two-phase commit protocol.

The consistency lock protocol was discussed in the section on lock management. The remaining protocols are discussed below.

Perhaps the simplest and easiest to implement recovery technique is based on the old-master new-master dichotomy common to most batch data processing systems: If the run fails, one goes back to the old-master and tries again. Unhappily, this technique does not seem to generalize to concurrent transactions. If several transactions concurrently access an object, then making a new-master object or returning to the old master may be inappropriate because it commits or backs up all updates to the object by all transactions.

It is desirable to be able to commit or undo updates on a per-transaction basis. Given an action consistent state and a collection of in-progress transactions (i.e., commit not yet executed) one wants to be able to selectively undo a subset of the transactions without affecting the others. Such a facility is called in-progress transaction backup.

A second shortcoming of versions is that in the event of a media error, one must reconstruct the most recent consistent

state. For example, if a page or collection of pages is lost from non-volatile storage, then they must be reconstructed from some redundant information. Doubly-recording the versions on independent devices is quite expensive for large objects. However, this is the technique used for some small objects such as the warm start file.

Lastly, writing a new version of a large database often consumes large amounts of storage and bandwidth.

Having abandoned the notion of versions, we adopt the approach of updating in place, and of keeping an incremental log of changes to the system state. (Logs are sometimes called audit trails or journals.)

Each action that modifies a recoverable object writes a log record giving the old and new value of the updated object. Read operations need generate no log records, but update operations must record enough information in the log so that, given the record at a later time, the operation can be completely undone or redone. These records will be aggregated by transaction, and collected in a common system log that resides in nonvolatile storage. The system log is duplexed and has independent failure modes.

In what follows we assume that the log never fails. By duplexing, triplexing, …., one can make this assumption less false.

Every recoverable operation must have:

- A DO entry that does the action and also records a log record sufficient to undo and to redo the operation,
- An UNDO entry that undoes the action given the log record written by the DO action,
- A REDO entry that redoes the action given the log record written by the DO action, and
- Optionally, a DISPLAY entry that translates the log into a human-readable format.

To give an example of an action and the log record it must write, consider the database record update operator. This action must record in the log the:

1. record name
2. the old record value (used for UNDO)
3. the new record value. (used for REDO)

The log subsystem augments this with the additional fields:

4. transaction identifier
5. action identifier
6. length of log record
7. pointer to previous log record of this transaction

DECLARE

```
1 UPDATE_LOG_RECORD BASED,
        2 LENGTH      FIXED(16),   /*length of log record
        */
        2 TYPE        FIXED(16),   /*code assigned to update
        log recs*/
        2 TRANSACTION FIXED(48),   /*name of
        transaction          */
        2 PREV_LOG_REC POINTER(31),/* relative
        address of prev log   */
        /*record of this transaction      */
        2 SET         FIXED(32),   /* name of updated set
        (table)   */
        2 RECORD      FIXED(32),   /*name of updated
        record        */
        2 NFIELDS     FIXED(l6),   /*number of updated
        fields        */
        2 CHANGES (NFIELDS),       /*for each changed
        field:        */
              3 FIELD FIXED(16);   /* name of field
              */
              3 OLD_VALUE,         /*old value of field
              */
                    4 F_LENGTH FIXED(16),/
                    *length of old field value      */
                    4 F_ATOM CHAR (F_LENGTH),
                    /*value in old field        */
              3 NEW_VALUE LIKE OLD_VALUE, /
              *new value of field       */
              2 LENGTH_AT_END FIXED(16); /
              *allows reading log backwards   */
```

The data manager's undo operation restores the record to its old value appropriately updating indices and sets. The redo operation restores the record to its new value. The display operation returns a text string giving a symbolic display of the log record.

The log itself is recorded on a dedicated media (disk, tape,...). Once a log record is recorded, it cannot be updated. However, the log component provides a facility to open read cursors on the log that will traverse the system log or will traverse the log of a particular transaction in either direction.

The UNDO operation must face a rather difficult problem at restart: The undo operation may be performed more than once if restart itself is redone several times (i.e. if the system fails during restart.) Also one may be called upon to undo operations that were never reflected in nonvolatile storage (i.e. log write occurred but object write did not.)

Similar problems exist for REDO. One may have to REDO an already done action if the updated object was recorded in non-volatile storage before the crash or if restart is restarted.

The write-ahead-log-protocol and high-water-marks solve these problems (see below).

5.8.3. 2. Write Ahead Log Protocol

The recovery system postulates that memory comes in two flavors: volatile and nonvolatile storage. Volatile storage does not survive a system restart. Nonvolatile storage usually survives a system restart.

Suppose an object is recorded in non-volatile storage before the log records for the object are recorded in the non-volatile log. If the system crashes at such a point, then one cannot undo the update. Similarly, if the new object is one of a set that is

committed together, and if a media error occurs on the object, then a mutually consistent version of the set of objects cannot be constructed from their non-volatile versions. Analysis of these two examples indicates that the log should be written to non-volatile storage before the object is written.

Actions are require to write log records whenever modifying recoverable objects. The leg (once recorded in nonvolatile storage) is considered to be very reliable. In general the log is dual recorded on physical media with independent failure modes (e.g., dual tapes or spindles) although single logging is a system option.

The Write Ahead Log Protocol (WAL) is:

- Before over-writing a recoverable object to nonvolatile storage with uncommitted updates, a transaction (process) should first force its undo log for relevant updates to nonvolatile log space.
- Before committing an update to a recoverable object, the transaction coordinator (see below) must force the redo and undo log to nonvolatile storage, so that it can go either way on the transaction commit. (This is guaranteed by recovery management that will synchronize the commit process with the writing of the phase-2 log transition record at the end of phase-1 of commit processing. This point cannot be understood before the section on two phase commit processing is read.)

This protocol needs to be interpreted broadly in the case of messages: One should not send a recoverable message before it is logged (so that the message can be canceled or retransmitted.) In this case, the wires of the network are the “non-volatile storage”.

The write-ahead-log protocol is implemented as follows. Every log record has a unique sequence number. Every recoverable object has a “high water mark” which is the largest log sequence number that applies to it. Whenever an object is updated, its high water mark is set to the log sequence number of the new log record. The object cannot be written to non-volatile storage before the log has been written past the object’s high water mark. Log manager provides a synchronous call to force out all log records up to a certain sequence number.At system restart a transaction may be undone or redone. If an error occurs the restart may be repeated. This means that an operation may be undone or redone more than once. Also, since the log is “ahead of” non-volatile storage the first undo may apply to an already undone (not-yet-done) change. Similarly the first redo may redo an already done change. This requires that the redo and undo operators be repeatable (idempotent) in the sense that doing them once produces the same result as doing them several times. Undo or redo may be invoked repeatedly if restart is retried several times or if the failure occurs during phase 2 of commit processing.

Here again, the high water mark is handy. If the high water mark is recorded with the object, and if the movement of the object to nonvolatile storage is atomic (this is true for pages and for messages) then one can read to high water mark to see if undo or redo is necessary. This is a simple way to make the undo and redo operators idempotent.

Message sequence numbers on a session perform the function of high water marks. That is the recipient can discard messages below the last sequence number received.

As a historical note, the need for WAL only became apparent with the widespread use of LSI memories. Prior to that time the log buffers resided in core storage that survived software errors, hardware errors and power failure. This allowed the system to treat the log buffers in core as non-volatile storage, At power shutdown, an exception handler in the data management dumps the log buffers. If this fails a scavenger is run which reads them out of core to storage. In general the contents of LSI storage does not survive power failures. To guard against power failure, memory failure and wild stores by the software, most systems have opted for the WAL protocol.

5.8.3.3. Two Phase Commit Protocol

5.8.3.3.1. The Generals Paradox

In order to understand that the two-phase commit protocol solves some problem it is useful to analyze this generals paradox.

There are two generals on campaign. They have an objective (a hill) that they want to capture. If they simultaneously march on the objective they are assured of success. If only one marches, he will be annihilated.

The generals are encamped only a short distance apart, but due to technical difficulties, they can communicate only via runners. These messengers have a flaw, every time they venture out of camp they stand some chance of getting lost (they are not very smart.)

The problem is to find some protocol that allows the generals to march together even though some messengers get lost.

There is a simple proof that: no fixed length protocol exists: Let P be the shortest such protocol. Suppose the last messenger in P gets lost. Then either this messenger is useless or one of the generals doesn’t get a needed message. By the minimality of P, the last message is not useless so one of the general doesn’t march if the last message is lost. This contradiction proves that no such protocol P exists.

The generals paradox (which as you now see is not a paradox) has strong analogies to problems faced by a data recovery management when doing commit processing. Imagine that one of the generals is a computer in Tokyo and that the other general is a cash-dispensing terminal in Fuessen Germany. The goal is to

- open a cash drawer with a million marks in it (at Fuessen) and
- debit the appropriate account in the non-volatile storage of the Tokyo computer.

If only one thing happens, either the Germans or the Japanese will destroy the general that did not “march”.

5.8.3.3.2. The Two Phase Commit Protocol

As explained above, there is no solution to the two generals problem.

If however, the restriction that the protocol have some finite fixed maximum length is relaxed then a solution is possible.

The protocol about to be described may require arbitrarily many messages. Usually it requires only a few messages, sometimes it requires more and in some cases (a set of measure zero) it requires an infinite number of messages. The protocol works by introducing a commit coordinator. The commit coordinator has a communication path to all participants. Participants are either cohorts (processes) at several nodes or are autonomous components within a process (like DB and DC)or are both.

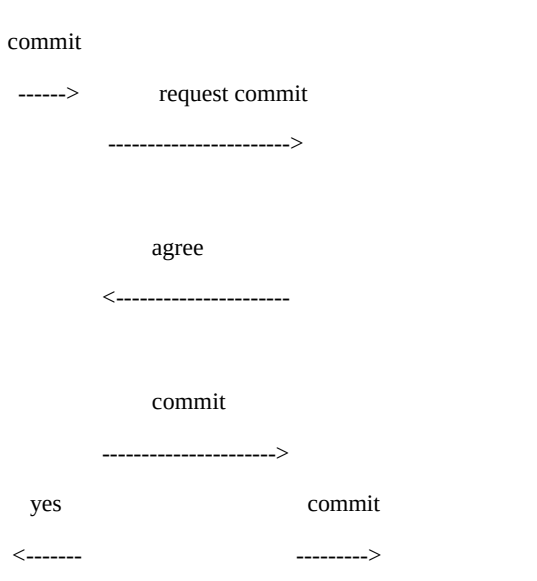
The commit coordinator asks all the participants to go into a state such that, no matter what happens, the participant can either redo or undo the transaction (this means writing the log in a very safe place).

Once the coordinator gets the votes from everyone:

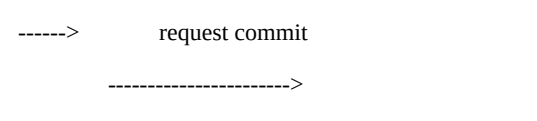
- If anyone aborted, the coordinator broadcasts abort to all participants, records abort in his log and terminates. In this case all participants will abort.
- If all participants voted yes, the coordinator synchronously records a commit record in the log, then broadcasts commit to all participants and when an acknowledge, is received from each participant, the coordinator terminates.

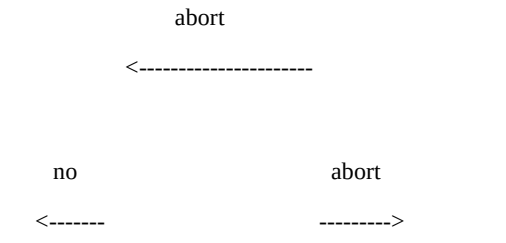
The key to the success of this approach is that the decision to commit has been centralized in a single place and is not time constrained.

The following diagrams show the possible interactions between a coordinator and a participant. Note that a coordinator may abort a participant that agrees to commit. This may happen because another participant has aborted

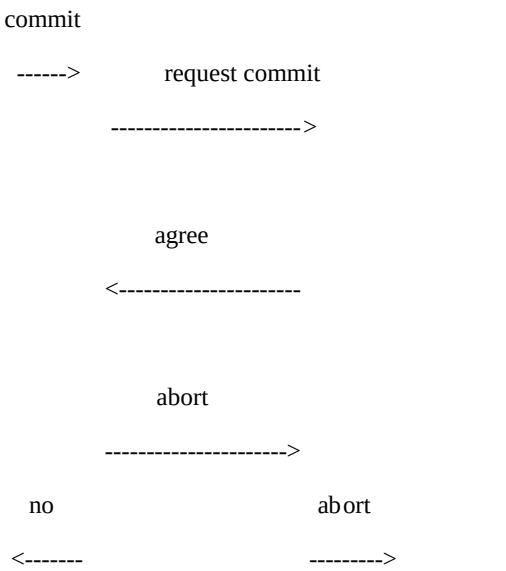


(1) Successful commit exchange.





(2) Participant aborts com



(3)Coordinator aborts commit.

Three possible two-phase commit scenarios.

The logic for the coordinator is best described by a simple program:

```
COORDINATOR: PROCEDURE;
   VOTE='COMMIT'; /*collect votes */
   DO FOR EACH PARTICIPANT
WHILE(VOTE='COMMIT');
      DO;
      SEND HIM REQUEST_COMMIT;
      IF REPLY != 'AGREE' THEN VOTE='ABORT';
      END;
   IF VOTE='COMMIT' THEN
      DO; /*if all agree then commit+/
      WRITE_LOG(PHASE12_COMMIT)FORCE;
      FOR EACH PARTICIPANT;
         DO UNTIL (+ACK);
         SEND HIM COMMIT;
         WAIT +ACKNOWLEDGE;
         IF TIME LIMIT THEN RETRANSMIT;
         END;
      END;
      ELSE
```

```
        DO; /*if any abort, then abort*/
        FOR EACH PARTICIPANT
          DO UNTIL (+ACK);
          SEND MESSAGE ABORT;
          WAIT +ACKNOWLEDGE;
          IF TIME_LIMIT THEN RETRANSMIT;
          END
        END;
    WRITE_LOG(COORDINATOR_COMPLETE);/
*common exit*/
    RETURN;
    END COORDINATOR;
```

The protocol for the participant is simpler:

```
PARTICIPANT: PROCEDURE;
    WAIT FOR REQUEST_COMMIT; /*phase 1 */
    FORCE UNDO REDO LOG TO NONVOLATILE
STORE;
    IF SUCCESS THEN /*writes AGREE in log */
        REPLY 'AGREE';
    ELSE
        REPLY 'ABORT';
    WAIT FOR VERDICT; /*phase2 */
    IF VERDICT ='COMMIT' THEN
        DO;
        RELEASE RESOURCES & LOCKS;
        REPLY +ACKNOWLEDGE;
        END;
    ELSE
        DO;
        UNDO PARTICIPANT;
        REPLY +ACKNOWLEDGE;
        END;
    END PARTICIPANT;
```

There is a last Piece of logic that needs to be included: In the event of restart, recovery manager has only the log and the nonvolatile store. If the coordinator crashed before the PHASE12_COMMIT record appeared in the log, then restart will broadcast abort to all participants. If the transaction's PHASE12_COMMIT record appeared and the COORDINATOR_COMPLETE record did not appear, then restart will re-broadcast the COMMIT message. If the transaction's COORDINATOR_COMPLETE record appears in the log, then restart will ignore the transaction. Similarly transactions will be aborted if the log has not been forced with AGREE. If the AGREE record appears, then restart asks the coordinator whether the transaction committed or aborted and acts accordingly (redo or undo.)

Examination of this protocol shows that transaction commit has two phases:

1. Before its PHASE12_COMMIT or AGREE_COMMIT log record has been written and,
2. After its PHASE12_COMMIT or AGREE_COMMIT log record has been written.

This is the reason it is called a two-phase commit protocol. A fairly lengthy analysis is required to convince oneself that a crash or lost message will not cause one participant to "march" the wrong way.

Let us consider a few cases. If any participant aborts or crashes in his phase 1 then the entire transaction will be aborted (because the coordinator will sense that he is not replying using timeout).

If an participant crashes in his phase 2 then recovery manager, as a part of restart of that participant, will ask the coordinator whether or not to redo or undo the transaction instance. Since the participant wrote enough information for this in the log during phase 1, recovery manager can go either way on completing this participant. This requires that the undo and redo be idempotent operations. Conversely, if the coordinator crashes before it writes the log record, then restart will broadcast abort to all participants. No participant has committed because the coordinator's PHASE12_COMMIT record is synchronously written before any commit messages are sent to participants. On the other hand if the coordinator's PHASE12_COMMIT record is found in the log at restart, then the recovery manager broadcasts commit to all participants and waits for acknowledge. This redoes the transaction (coordinator).

This rather sloppy argument can be (has been) made more precise. The net effect of the algorithm is that either all the participants commit or that none of them commit (all abort.)

5.8.3.3.3. Nested Two Phase Commit Protocol

Many optimizations of the two-phase commit protocol are possible. As described above, commit requires 4N messages if there are N participants. The coordinator invokes each participant once to take the vote and once to broadcast the result. If invocation and return are expensive (e.g., go over thin wires) then a more economical protocol may be desired.

If the participants can be linearly ordered then a simpler and faster commit protocol that has 2N calls and returns is possible. This protocol is called the nested two-phase commit. The protocol works as follows:

- Each participant is given a sequence number in the commit call order.
- In particular, each participant knows the name of the next participant and the last participant knows that he is the last.

Commit consists of participants successively calling one another (N-l calls) after performing phase 1 commit. At the end of the calling sequence each participant will have successfully completed phase 1 or some participant will have broken the call chain. So the last participant can perform phase 2 and returns success. Each participant keeps this up so that in the end there are N-l returns to give a grand total of 2(N-1) calls and returns on a successful commit. There is one last call required to signal the coordinator (last participant)that the commit completed so that restart can ignore redoing this transaction. If some participant

does not succeed in phase 1 then he issues abort and transaction undo is started.

The following is the algorithm of each participant:

```
COMMIT: PROCEDURE;
  PERFORM PHASE 1 COMMIT;
  IF FAIL THEN RETURN FAILURE;
  IF I_AM_LAST THEN
    WRITE_LOG(PHASE12)FORCE;
    ELSE
      DO;
      CALL COMMIT(I+l);
      IF FAIL THEN
        DO;
        ABORT;
        RETURN FAILURE;
        END;
      END;
    PERFORM PHASE 2 COMMIT;
    IF I_AM_FIRST THEN
        INFORM LAST THAT COMMIT COMPLETED;
    RETURN SUCCESS;
    END;
```

The following gives a picture of a three deep nest:

```
                Rl
        commit.
        --------->
                --PHASE1 --> R2
                        --PHASEl -> R3
                        <--PHASE2 –
                <--PHASE2 –
         yes            Fin
        <---------  ------------------------>
                (a) a successful commit.
                Rl
        commit
        --------->
                --Phase1 --> R2
                        --PHASEl--> R3
                        <--ABORT --
                <--ABORT --
            no
        <--------
```

b. an unsuccessful commit.

5.8.4.3.3.. Comparison Between General And Nested Protocols

The nested protocol is appropriate for a system in which

- The message send-receive cost is high and broadcast not available.
- The need for concurrency within phase 1 and concurrency within phase 2 is low,
- The participant and cohort structure of the transaction is static or universally known.

Most data management systems have opted for the nested commit protocol for these reasons. On the other hand the general two phase commit protocol is appropriate if:

- Broadcast is the normal mode of interprocess communication (in that case the coordinator sends two messages and each process sends two messages for a total of 2N messages.)Aloha net, Ethernet, ring-nets, and space-satellite nets have this property.
- Parallelism among the cohorts of a transaction is desirable (the nested protocol has only one process active at a time during commit processing.)

5.8.3.4. Summary of Recovery Protocols

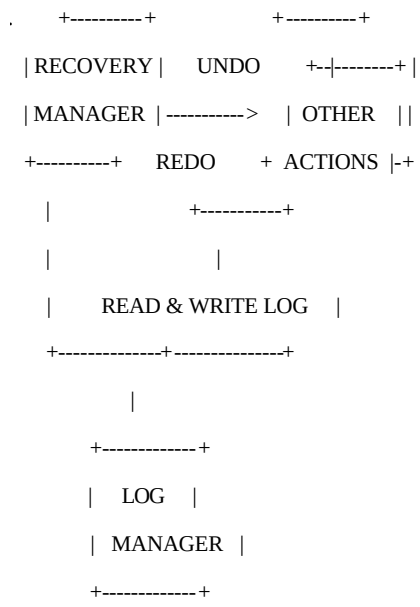
- The consistency lock protocol isolates transactions from inconsistencies due to concurrency.
- The DO-REDO-UNDO log record protocol allows for and uncommitted actions.
- The write-ahead-log protocol insures that the log is ahead of nonvolatile storage, so that undo and redo can always be performed.
- The two-phase commit protocol coordinates the commitment of autonomous participants (or cohorts) within a transaction. The following table explains the virtues of the write-ahead-log and two-phase-commit protocols. It examines the possible situations after a crash. The relevant issues are whether an update to the object survived (was written to nonvolatile storage), and whether the log record corresponding to the update survived. One mill never have to redo an update whose log record is not written because: Only committed transactions are redone, and COMMIT writes out the transaction's log records before the commit completes. So the (no, no, redo) case is precluded by two-phase commit. Similarly, write-ahead-log (WAL) precludes the (no, yes,*) cases, because an update is never written before its log record. The other cases should be obvious.

LOG RECORD WRITTEN	OBJECT WRITTEN	REDO OPERATION	UNDO OPERATION
NO	NO	IMPOSSIBLE	NONE
		BECAUSE OF TWO PHASE COMMIT	
NO	YES	IMPOSSIBLE BECAUSE OF WRITE AHEAD LOG	
YES	NO	REDO	NONE
YES	YES	NONE	UNDO

5.8.4. Structure of Recovery Manager

Recovery management consists of two components.

- Recovery manager that is responsible for tracking transactions and the coordination of transaction COMMIT and ABORT, and system CHECKPOINT and RESTART (see below).
- Log manager that is used by recovery manager and other components to record information in the system log for the transaction or system.



Relationship between Log manager and component actions.

The purpose of the recovery system is two-fold: First, the recovery system allows an in-progress transaction to be “un-done” in the event of a “minor” error, without affecting other transactions. Examples of such errors are operator cancellation of the transaction, deadlock, timeout, protection or integrity violation, resource limit,..,.

Second, in the event of a “serious” error, the recovery sub-system minimizes the amount of work that is lost and by restoring all data to its most recent committed state. It does this by periodically recording copies of key portions of the system state in nonvolatile storage, and by continuously maintaining a log of changes to the state, as they occur. In the event of a catastrophe, the most recent transaction consistent version of the state is reconstructed from the current state on nonvolatile storage by using the log to

- undo any transactions that were incomplete at the time of the crash.
- redo any transactions that completed in the interval between the checkpoints and the crash.

In the case that on-line nonvolatile storage does not survive, one must start with an archival version of the state and reconstruct the most recent consistent state from it. This process requires:

- Periodically making complete archive copies of objects within the system.
- Running a change accumulation utility against the logs written since the dump. This utility produces a much smaller list of updates that will bring the image dump up to date. Also this list is sorted by physical address so that adding it to the image dump is a sequential operation.
- The change accumulation is merged with the image to reconstruct the most recent consistent state.

  Other reasons for keeping a lag of the actions of transactions include auditing and performance monitoring since the log is a trace of system activity.

There are three separate recovery mechanisms:

1. Incremental log of updates to the state.
2. Current on-line version of the state.
3. Archive versions of the state.

5.8.4.1. Transaction Save Logic

When the transaction invokes SAVE, a log record is recorded which describes the current state of the transaction. Each component involved in the transaction is then invoked, and it must record whatever it needs to restore its recoverable objects to their state at this point. For example, the terminal handler might record the current state of the session so that if the transaction backs up to this point, the terminal can be reset to this point. Similarly, database manager might record the positions of cursors. The application program may also record log records at a save point.

A save point does not commit any resources or release any locks.

5.8.4.2. Transaction Commit Logic

When the transaction issues COMMIT, recovery manager invokes each component (participant) to perform commit processing. The details of commit processing were discussed under the topics of recovery protocols above. Briefly, commit is a two-phase process. During phase 1, each manager writes a log

record that allows it to go either way on the transaction (undo or redo). If all resource managers agree to commit, then recovery manager forces the log to secondary storage and enters phase 2 of commit. Phase 2 consists of committing updates: sending messages, writing updates to nonvolatile storage and releasing locks.

In phase 1 any resource manager can unilaterally abort the transaction thereby causing the commit to fail. Once a resource manager agrees to phase 1 commit, that resource manager must be willing to accept either abort or commit from recovery manager.

5.8.4.3. Transaction Backup Logic

The effect of any incomplete transaction can be undone by reading the log of that transaction backwards undoing each action in turn. Given the log of a transaction T:

```
UNDO(T):
    DO WHILE (LOG(T)!= NULL);
        LOG_RECORD = LAST_ RECORD (LOG(T));
        UNDOER = WHO_WROTE(LOG_ RECORD);
        CALL UNDOER(LOG_RECORD);
        INVALIDATE(LOG_RECORD);
        END UNDO;
```

Clearly, this process can be stopped half-day, thereby returning the transaction to an intermediate save point. Transaction save points allow the transaction to backtrack in case of some error and yet salvage all successful work.

From this discussion it follows that a transaction's log is a push down stack, and that writing a new record pushes it onto the stack, while undoing a record pops it off the stack (invalidates it). For efficiency reasons, all transaction logs are merged into one system log that is then mapped into a log file. But, the log records of a particular transaction are threaded together and anchored off of the process executing the transaction.

Notice that UNDO requires that while the transaction is active, the log must be directly addressable. This is the reason that at least one version of the log should be on some direct access device. A tape-based log would not be convenient for in-progress transaction undo for this reason (tapes are not randomly accessed).

The undo logic of recovery manager is very simple. It reads a record, looks at the name of the operation that wrote the record and calls the undo entry point of that operation using the record type. Thus recovery manager is table driven and therefore it is fairly easy to add new operations to the system.

Another alternative is to defer updates until phase 2 of commit processing. Once a transaction gets to phase 2, it must complete successfully, thus if all updates are done in phase 2 no undo is ever required (redo logic is required.) IMS data communications and IMS Fast Path use this protocol.

Bibliography

3.1. Bibliography

These notes are rather nitty-gritty; they are aimed at system implementers rather than at users. If this is the wrong level of detail for you (is too detailed) then you may prefer the very readable books:

Martin, *Computer Data Base Organization*, Prentice Hall, 1977 (What every DP Vice President should know.)

Martin, *Computer Data Base Organization, (2nd edition)*, Prentice Hall, 1976 (What every application programmer should know.)

The following is a brief list of sane of the more popular general-purpose data management systems that are commercially available:

Airlines Control Program, International Business Machines Corporation

Customer Information Computer System, International Business Machines Corporation

Data Management System 1100, Sperry Univac Corporation

Extended Data Management system, Xerox Corporation

Information Management System / Virtual Systems, International Business Machines Corporation

Integrated Database Management System, Cullinane Corporation

Integrated Data Store / 1, Honeywell Information Systems Inc

Model 204 Data Base Management System, Computer Corporation of America

System 2000, MRI Systems Corporation.

Total, Cincom Systems Corporation

Each of these manufacturers will be pleased to provide you with extensive descriptions of their systems.

Several experimental systems are under construction at present. Some of the more interesting are:

Astrahan et. al., "System R: a Relational Approach to Database Management", Astrahan et. al., ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976.

Marill and Stern, "The Datacomputer-A Network Data Utility." Proc. 1975 National Computer Conference, AFIPS Press, 1975,

Stonebraker et. al., "The Design and Implementation of INGRESS." ACM Transactions on Database Systems, Vol. 1, No. 3, Sept 1976,

There are very few publicly available case studies of data base usage. The following are interesting but may not be representative:

IBM Systems Journal, Vol. 16, No. 2, June 1977. (Describes the facilities and use of IMS and ACP).

"IMS / VS Primer," IBM World Trade Systems Center, Palo Alto California, Form number S320-5767-1, January 1977.

"Share Guide IMS User Profile, A Summary of Message Processing Program Activity in Online IMS Systems" IBM Palo Alto-Raleigh Systems Center Bulletin, form number 6320-6005, January 1977

Also there is one "standard" (actually "proposed standard" system):

CODASYL Data Base Task Group Report, April 1971. Available from ACM

3.9. Bibliography

Chamberlin et. al., "Views, Authorization, and Locking in a Relational Data Base System", 1975 NCC, Spartan Press. 1975. (Explains what views are and the problems associated with them.)

Computing Surveys, Vol. 8 No. 1, Barth 1976. (A good collection of papers giving current trends and issues related to the data management component of data base systems.)

Date, *Introduction to Database Systems*, Addison Wesley, 1975. (The seminal book on the data management part of data Management systems.)

Date, "An Architecture for High Level Language Database Extension;," Proceedings of 1976 SIGMOD Conference, ACM, 1976. (Unifies the relational, hierarchical and network models.)

Knuth, The Art of Computer Programming: Sorting and Searching, Vol. 3, Addison Wesley, 1975. (The seminal data structures book. Explains all about B-trees among other things.)

McGee, IBM Systems Journal, Vol. 16, No. 2, 1977, pp-84-160. (A very readable tutorial on IMS, what it does, how it works, and how it is used.)

Senko, "Data Structures and Data Accessing in Data Base Systems, Past, Present, Future," IBM Systems Journal, Vol. 16, No. 3, 1977, pp. 208-257. (A short tutorial on data models.)

4.8. Bibliography

Kimbelton, Schneider, "Computer Communication Networks: Approaches, Objectives, and Performance Considerations," *Computing Surveys*, Vol. 7, No. 3, Sept. 1975, (A survey paper on network managers.)

"Customer Information Control System/Virtual Storage (CICS/VS), System/Application Design Guide." IBM, form number SC33-3068, 1977 (An eminently readable manual on all aspects of data management systems. Explains various session management protocols and explains a rather nice message mapping facility.)

Eade, Homan, Jones, "CICS/VS and its Role in Systems Network Architecture," IBM Systems Journal, Vol. 16, No. 3, 1977 (Tells how CICS joined SNA and what SNA did for it.)

IBM Systems Journal, Vol. 15, No. 1, Jan. 1976. (All about SNA, IBM's network manager architecture.)

5.6. Bibliography.

Stonebraker, Neuhold, "A Distributed Data Base Version of INGRESS", Proceedings of Second Berkeley Workshop on Networks and Distributed Data, Lawrence Livermore Laboratory, (1977)-(Gives another approach to distributed transaction management.)

*'Information Management System/Virtual Storage (IBS/VS) System Manual Vol. 1: Logic. ll, IBM, form number LY20-8004-2. (Tells all about IMS. The discussion of scheduling presented here is in the tradition of JMS/VS pp 3.36-3.41.)

"OS/W2 System Logic Library", IBM, form number SY28-0763, (Documents the subsystem interface of OS/VS pp-3.159-168)

"OS/VS MVS Supervisor Services and Macro Instructions.", IBM, form number GC28-0756, (Explains percolation on pages 53-62.)

5.8.8 Bibliography

Alsberg, "A Principle for Resilient Sharing of Distributed Resources," Second National Conference on Software Engineering, IEEE Cat. No. 76CH1125-4C, 1976, pp. 562-570. (A novel proposal (not covered in these notes) which describes a protocol whereby multiple hosts can cooperate to provide a reliable transaction processing. It is the first believable proposal for system duplexing or triplexing I have yet seen. Merits further study and development.)

Bjork, "Recovery Scenario for a DB/DC System, I1 Proceedings ACM National Conference, 1973, pp. 142-146.

Davies, "Recovery Semantics for a DB/DC System," Proceedings ACM National Conference, 1973, pp. 136-141. (The above two companion papers are the seminal work in the field. Anyone interested in the topic of software recovery should read them both at least three times and then once a year thereafter.)

Lampson, Sturgis, "Crash Recovery in a Distributed System," Xerox Palo Alto Research Center, 1976 To appear in CACM. (A very nice paper which suggests the model of errors presented in section 5.8.1 and goes on to propose a three-phase commit protocol. This three-phase commit protocol is an elaboration of the two-phase commit protocol. This is the first (only) public mention of the two-phase commit protocol.)

Rosenkrantz, Sterns, Lewis, "System Level Concurrency Control for Data Base Systems, General Electric Research, Proceedings of Second Berkeley Workshop on Distributed Data Management and Data Management, Lawrence Berkeley Laboratory, LBL-6146, 1977, pp. 132-145. also, to appear in Transactions on Data Systems, AC& (Presents a form of nested commit protocol, allows only one cohort at a time to execute.)

Information Management System/Virtual Storage (IMS/VS), System Programming Reference Manual, IBM Form No SH20-9027-2, p. 5-2. (Briefly describes WAL.)

2.2. Bibliography

DB/DC Data Dictionary General Information Manual, IBM, Form number GH20-9104-1, May 1977

UCC TEN, Technical Information Manual, University Computing, Corporation, 1976

Lefkovits, *Data Dictionary Systems*, Q. E. D. Information Sciences Inc., 1977, (A buyer's guide for data dictionaries.)

Nijssen (editor), *Modeling in Data Base Management Systems*, North Holland, 1976. (All you ever wanted about conceptual schema.)

"SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control." Chamberlain et. al., IBM Journal of Research and Development, Vol. 20, No. 6, November 1976. (Presents a unified data definition, data manipulation facility.)

Notes:

POINTS TO REMEMBER - I

* last time: talked about transactions, and desired properties for transactions, and concurrency control when transactions are done concurrently; and we talked about using locks to implement such concurrency control.

* today: proceed from there...but FIRST (because now you should be ready for it):

FIRST: some terminology that you'll need :

* Some TERMINOLOGY you should know: (yes, it'll be on the final!)

* **entity integrity**

* if the DBMS requires "legal", unique primary keys,then you have this; every row has a unique attribute that uniquely identifies/uniquely determines the row;

* **domain integrity**

* you have this if you can specify that an attribute must be one of a particular set of values, and that restriction is enforced by the DBMS;

* that is, you can specify the domain of a column, and that restriction is enforced.

* **referential integrity**

* you have this if you can specify a foreign key relationship, and subsequently that referential constraint is enforced by the DBMS-

* so, for example, if you try to insert a child containing a value for the foreign key that is not in the parent table, that would not be allowed; and, likewise, you could not delete a parent if a child existed referencing that parent in its foreign key field (unless "on delete cascade", "on delete set null", or some such is specified)

* **transaction integrity**

* you have this if you have atomicity of transactions (completely done, or not even begun/nothing done at all), supported by the DBMS

* The Oracle DBMS does support all four of these (to VARYING degrees, however!)

* (Note from A. Burroughs: Domain integrity is sort of a problem since I don't think you can, at least in Oracle, specify some data type such as "telephone numbers" and have the database enforce that.

You'd have to specify the field as character and then somehow evaluate it yourself, possibly in a trigger, to ascertain if the value was a legitimate telephone number.

You can probably think of other examples where specifying the "CHECK" condition is too complicated for the available syntax. So Oracle (and other databases) support domain integrity only to a limited degree.

However, I understand the new SQL standard supports the idea of a user-defined data type, which can then be used in the definition of a field's type, so the vendors are on the way to solving this problem.

/Ann)

* Now, back to where we left off:

* we talked about the basics of BINARY locks and of SHARED/EXCUSIVE lockslast time; BUT!!!

* ON THEIR OWN, locks do NOT ensure serializability!!!

* However, when using locks, serializability can be achieved in several ways ... one way is to process transactions using two-phased locking

* (that is, using locks does not necessarily result in a serializable transaction schedule; but, serializability can be guaranteed if a two-phased locking protocol is used...)

* with this strategy, transactions may obtain locks as necessary, but once the first lock (of any kind?) is released, no other lock can be obtained.

* and, it is called two-phased because, when you enforce the above, it results in a transaction having a growing phase, in which locks are obtained, and a shrinking phase, in which locks are released;

* (Note that a more-restrictive (and easier-to-implement) variation of two-phased locking, used by DB2 (and some other DBMS's, also?), simply does not release any locks until a COMMIT or ROLLBACK command is issued

* so, no lock is released until the transaction has essentially been completely done, or completely not donenote that locks can thus be obtained throughout the transaction...shrinking phase simply does not begin until the transaction is complete or aborted.

* locks can create deadlocks - while locking helps solve inconsistency problems, it introduces another class of problems...deadlock conditions!

* (review, but just in case) consider the following scenario:

* A wants to get some pencils, and if she can get them, then she wants to get paper

* B wants to get some paper, and if he can get them, then he wants to get pencils

* A locks paper record

* B locks pencil record

* A gets paper, modifies pencil record accordingly

* B gets pencils, modifies paper record accordingly

* A waits to get lock for paper

* B waits to get lock for pencils

* see the problem? A and B are in a state of deadlock, (locked in the deadly embrace)

* each is waiting for a resource that the other has locked

* four strategies for deadlock management (three mentioned in book,plus one more)

* detection - alowing it to occur, detecting it, and breaking it.

DBMS periodically tests the db for deadlocks

* once you're past how to detect it, then note that (usually?) the only way to break the deadlock is to kill (and rollback) one of the transactions... (and the other then continues)

* prevention - preventing deadlock from occurring at all

* a transaction requesting a new lock is aborted if there is the possibility that a deadlock can occur (and that aborting releasing all locks obtained)

* transaction is then rescheduled for execution

* works because it avoids the conditions that lead to deadlocking

* avoidance - one of numerous ways is to simply require the transaction to lock all of the resources desired at once before it can be executed (e.g., A would be required to lock paper and pencils at the beginning, and then proceed.)

* (sometimes this is grouped under deadlock prevention)

* avoids rollback of conflicting transactions, but increases action response times

* ignore the problem (!) — hope it doesn't happen, and require human intervention to kill one of the transactions involved if it occurs (!!)

* not an uncommon approach for operating systems in relation to locking files, I'm told! Not sure how common it is for DBMS's;

* TIMESTAMPS

* time stamping algorithms are another class of algorithms for scheduling concurrent transactions;

* a time stamp is assigned to each transaction;

* note — the time stamp is global, it must be unique for each transaction, and must have the property of monotonicity (the time stamp values must always increase);

* all database operations (READ and WRITE) within the same transaction must have the same time stamp (that is, all database operations within the same transaction will be given the same time stamp...)

* the DBMS then executes conflicting operations in time stamp order, thereby ensuring serializability of the transactions; if 2 transactions conflict, one often is stopped, rescheduled, and assigned a new time stamp.

* I'd like to now go into a bit more detail than the book does about this approach...

* With each data item Q, you associate two timestamp values:

* W-timestamp(Q), the largest time stamp of any transaction that executed write(Q) successfully.

* R-timestamp(Q), the largest time stamp of any transaction that executed read(Q) successfully.

* then, the timestamp-ordering protocol ensures that any conflicting reads and writes are executed in timestamp order (thus ensuring serializability):

* Assume that TS(Ti) returns transaction Ti's timestamp;

* if Ti issues read(Q):

* if TS(Ti) < W-timestamp(Q), then Ti needs to read a value of Q that was "already" overwritten by a "later" transaction (according to timestamp ordering);

* read will be rejected, and Ti will be rolled back.

* if TS(Ti) >= W-timestamp(Q), then it is "safe" to execute the read;

* read will be executed and R-timestamp(Q) will be set to max(R-timestamp(Q), TS(Ti))

* (need to note that this read has been done, so another write does not get done "before" this read...)

* if Ti issues write(Q),

* if TS(Ti) < R-timestamp(Q), then a "later" transaction has already read and used this value - overwriting it now would be bad, because there's no way for that "later" transaction to see the new value.

* write will be rejected, and Ti will be rolled back.

* if TS(Ti) < W-timestamp(Q), then Ti is trying to write an obsolete value of Q (already overwritten, according to timestamp ordering).

* write will be rejected, and Ti will be rolled back.

* otherwise, write is executed, and W-timestamp(Q) updated to the timestamp of Ti

* one drawback: note the overhead for these timestamp, within the database! each value stored in the database needs 2 additional time stamp fields, for R-timestamp and W-timestamp.

* (although, as someone in class astutely pointed out, one could likely use a hash table or some such in a clever fashion to only store timestamps as they are set, to save space!)

* (if a data "chunk" Q had no R-timestamp or W-timestamp, then it is "safe" to read or write it, after all, and *then* it would need the appropriate timestamp stored for it;)

* optimistic methods - based on the assumption that the majority of database operations do not conflict;

* does not require locking or time-stamping —

* instead, a transaction is executed without restrictions until it is committed;

* the transaction moves through three phases: read phase, validation phase, write phase;

* read phase: transaction reads the database, executes the needed computations, and makes the updates to a private copy of the database values;

* all updates are recorded in a temporary update file, accessible only by that one transaction (and not any others running simultaneously)

* validation phase: the trnasaction is validated to assure that the changes made will not affect the database's integrity or consistency;

* if validation test succeeds? The transaction goes to the write phase.

* if it fails? Trnasaction is restarted, and its changes discarded

* write phase: the changes are permanently applied to the db.

* this approach is acceptable for mostly read or query db systems that require very few update transactions...(not covered in 11-28-2000 lecture, leaving here in case it might be

of interest; SOME of this WAS covered in 11-14-2000 lecture.)

* database recovery mgmt:

* what is recovery?

* recovery restores a db from a given (usually inconsistent) state to a previously consistent state.

* in recovery, want to make sure, of course, that transactions are treated as atomic — like a single, logical, indivisible piece of work — each transaction either has all of its actions applied and completed, or none of them;

* to recover from a partially-completed transaction, then, all thechanges made by that transaction must be reversed so that the database state is as it was before the transaction was aborted;

* so, we sometimes need to recover from transactions — but, also, one needs to apply recovery techniques to the whole database or system after critical error has occurred;

* backup and recovery capabilities are a very important component of modern DBMS's!!

* some allow for automatic db backups, that the DBA can schedule, to automatically backup the db to permanent secondary storage devices like disks or tapes;

* different levels of backup are possible:

* a full backup of the db or dump of the db

* a differential backup of the db — only the last modifications to the db (when compared with a previous backup copy) are copied.

* faster, since you only copy the modifications, not the entire database;

* but complicates/slows down recovery, since you must go to the previous backup copy of the db, and then perform all of the recorded modifications since;

* a backup of the transaction log only; backs up all the transaction log operations not reflected in a previous backup copy of the db

* isn't this just a variation on a differential backup...?

* failures that can affect db's can have many sources...for example:

* software: software-indiced failures may be traceable to the OS, the DBMS, application programs, viruses, etc.;

* hardware: hardware-induced failures may be due to, for example, memory chip errors, disk crashes, bad disk sectors, disk full errors, etc.!

* programming exemption: application programs or end users may roll back transactions when certain conditions are defined (end user types a break sequence (Ctl-C, for ex) either intentionally or unintentionally, someone attempts to withdraw money from an account with balance 0, etc.)

* transaction: system, for example, may detect deadlock and abort one of the deadlocked transactions

* external: system suffers complete destruction due to external factors, such as due to fire, earthquake, flood, etc.

* recovery may be almost unnoticeable to the users (rolling back and rescheduling a deadlocked transaction where the deadlock was quickly detected) to a major long-term rebuild (computer system destroyed by flood at a time where one will have to start from an old backup and perform a large log of action-done-since...)

* note that, usually, recovery is not possible without backup of the database!

* there are a number of options for how the database recovery may proceed...

* first, determine the type and extent of the required recovery (just a single transaction? The entire database?)

* if the entire database needs to be recovered to a consistent state, even then there may be different appraches; here are a couple.

* use the most recent backup copy of the database that is known to be in a consistent state.

* deferred-write or deferred-update approach:(this is roughly what you should assume for the homework problem — I'm asking which transactions will actually have their actions redone...)

* go through the transaction log since the backup, and write all changes (previous and new values) not immediately to the database, but to the end of the transaction log;

* only actually update the database when each transaction reaches its commit point (upon reaching a commit, only then would you actually perform the changes logged for that transaction).

* if the transaction fails before reaching that commit point, note that no changes need to be made to the database, because none of that transaction's changes have actually been made yet; so, won't need to do a ROLLBACK at this point.

* (you'd need to log a rollback, though, wouldn't you? Not sure)

* (assume, for the homework, that transactions NOT committed at the time of the failure are to be rescheduled and restarted...if the "end" of the log (as of the time of the failure) is reached and no commit is found for that transaction, it is considered to need rescheduling...)

* in write-through or immediate-update approach,

* the database is immediately updated by transaction operations logged during its pass through that log, even before you know if the transaction committed or not;

* (the transaction log is also updated);

* if the transaction fails (or you reach the end before the failure without it having been committed?), the database uses the log information to roll back the database in terms of that transaction.

* it is an interesting question: what's easier, to backtrack to all of a transaction's operations when a commit is done (deferred-write) or to backtrack to all of its operations when it is noted as having failed, or when you've reached the time of failure and found it had not been committed (write-through);

Notes:

POINTS TO REMEMBER - II

Intro to Client-Server Database Systems

* also see: Connolly and Begg, Chapter 2, pp. 57-60

* First came the Mainframe/Terminal paradigm.

* Then came the equal Peer-to-Peer paradigm.

* Now comes something in between - something that offers the hierarchical structure of mainframes and the distributed computing nature of peer-to-peer. CLIENT/SERVER.

* what do we mean by client-server?

* "A client-server system is a network of computers in which some of the computers (called clients) process applications while other computers (the servers) provide services to the clients";

* hmm; I DON'T LIKE THAT; seems overly simplified;

* yes, there is a network of computers involved;

* some computers are servers of various services - other computers, clients, can access these services;

* ever heard of the idea of a process? can think of a process as a program-in-operation;

* I think you can think of a server program as being a process running on one computer, waiting for requests;

* when a client computer program - process -on another computer wants to make such a request, it sends the request to that server process on that first computer;

* print-server? When you want to print, your program sends a message to the computer running/handling printer services;

* web-server? Your browser process sends messages and receives replies from a web-server process;

* and, of course, for this class, we are particularly interested in a database server - clients can contact a database server process;

* By distributing the computing away from one place, the bigmuscle of the mainframe can be replaced by less expensive servers. (UNIX vendors are advertising heavily to fill this market, threatening to push mainframes out.)

* On the software side, one can now buy off-the-shelf (no local development costs!) client packages for the PCs and server packages for the mainframes/servers to implement all sorts of things nowadays. Such oftware has to be

* flexible,

* customizable, and

* capable of being run on all sorts of client and server platforms.

* in Kroenke, they consider each database to have a single database server - if a database were spread among two or more servers, that would be a distributed database system; (more on that 2nd half of today's lecture);

* Remember, from the beginning we've pointed out how: (draw usual pic?)

    * users often use application program

    * application program communicates with DBMS

    * DBMS handles nasty little file details, maintains abstraction that database is a collection of tables — and, of course, it is the interface to the database.

    * now, consider —

        * what does the application program do, in this "relationship"?

            * provides/processes/handles the application's user interface

            * invokes the application logic

            * may enforce some business rules;

            * calls the DBMS for database services;

                * and, if the DBMS is relational, often uses SQL statements to express such service requests;

        * what does the DBMS do, in this "relationship"?

            * processes the SQL statements/requests it receives;

            * may enforce business rules, too (those involving triggers, for example, or those "set up" by foreign key table constraints, for example)

            * it returns data (in the form of tables) as well as error or informational messages (the result of the SQL statements sent?);

            * and, of course, it (may!) provide reliability (concurrency control, backups, etc.) and security (passwords, access control, etc.)

* and, the DBMS will issue various commands that it needs to to the operating system — file input/output, etc.

* so, think about it — all of these functions would be required in any database system; they could easily all be done on a single computer.

* and, on a personal database running on a personal computer, they are;

* one can also imagine a teleprocessing system, where terminals access a mainframe, and everything runs on the mainframe; (I prefer to think of this as the mainframe/terminal paradigm)

* BUT - one can also imagine this being split up onto several computers in various ways!

* (most obviously?) the application program could be on 1 computer, and the DBMS and underlying database on another; the DBMS and underlying database could be on different computers; etc.

* according to Kroenke, p. 348, "With client-server database systems, the application program is placed on the client computer and the DBMS and operating systems file management programs are placed on the server computer..."

* this makes sense to me, so I'll trust him on this!

* so, notice that redwood is our Oracle database server —

* but, so far, in our work, we've been using it, really, in more of a mainframe/terminal mode (I think) than a client/server mode; we connect to redwood, and do ALL our work there! (None is really being done by the client)

* could we use Oracle on redwood as a database server? Yes, indeed;

* but, you would need (as Kroenke says, p. 348) "a piece of the DBMS, called the DBMS driver" on the client;

* drivers are, I believe, commonly needed for client/server interaction -'ve certainly heard of printer drivers? (They're also common in terms of software that interacts between the OS and hardware, in general - CD-ROM drivers, printer drivers, math card drivers, etc.?)

* "The function of the DBMS driver is to receive processing requests from the application and format them for delivery to the DBMS";

* "it also receives responses from the DBMS and formats them for the application";

* (DBMS lite? 8-) The local rep/liaison for the DBMS? 8-))

* There's also "a layer of communications software placed on both the client and the server."

* "The role of this software is to interact with the communications hardware to send and receive messages between the DBMS driver and the DBMS"

* sometimes this communications software is called an APPLICATIONS PROGRAMMING INTERFACE (API), an implementation of a protocol or set of protocols designed specifically for this purpose. (I THINK)

* These protocols are implemented in the (in network layer terms!) Application layer usually and are sometimes referred to as MIDDLEWARE.

* [add: client may also be known as front-end application, server may also be known as back-end application]

* Let's consider Oracle, for a moment.

* Oracle database server uses SQL*Net as its communications protocol with the client;

* (SQL*Net is a proprietary solution limited to ORACLE databases, and is used by Oracle to send SQL requests over a network)

* (seek to recreate pic on p. 348, fig 14-2 -

User <—> | applic program | dbms driver | comm | <— nw—> | comm | dbms | os | <—> db

client computer(s) server computer

* so, note that the client computer

* manages the user interface,

* accepts data from the user,

* processes application logic,

* enforces (some) business rules,

* generates requests (SQL, usually?) for database services

* transmits those requests to the server and receives results

* formats the results for the user

* and, note that the server computer:

* accepts the clients' requests,

* processes them,

* returns a response

* while enforcing business rules,

* performing database integrity checking,

* maintaining the db overhead data,

* providing concurrent access control,

* providing recovery, and

* providing security!

* advantages and disadvantages

* (you know there are always tradeoffs in this sort of thing! 8-))

* this kind of client/server system places application processing closer to the user -

* MAY result in better performance - several CPU's, in a sense, processing stuff in parallel!

* perhaps? less communication costs (than if all was being done REMOTELY) (than for file-sharing systems?)

* only requests for DBMS processing and the responses need to be sent over the communication network;

* CPU power is available to make user interface more elaborate; (GUI's more practical!)

* potential for possibly a choice of clients

* control might be more difficult? (more truly parallel?)

* after all, the clients operate in parallel and process applications in parallel;

* more chance of lost update, other multi-user control problems; (hmm! Because I don't see how you'd "lock" stuff based on SQL commands hmm)

* seque to distributed db's: in the above, we assumed that a database was on a single computer, and that a single server provided access to that database - what if the database itself could be on more than one computer?

That's getting into distributed database... more on that later!

Notes:

GLOSSARY

A

Abend

The result of erroneous software logic or hardware failure and can be compared to Oracle's ORA-600 errors. Derived from "ABnormal END," an error message on the IBM360 Mainframe.

ACID

The basic properties of a database transaction: Atomicity, Consistency, Isolation, and Durability. All Oracle transactions comply with these properties.

- Atomicity - The entire sequence of actions must be either completed or aborted. The transaction cannot be partially successful.
- Consistency - The transaction takes the resources from one consistent state to another.
- Isolation - A transaction's effect is not visible to other transactions until the transaction is committed.
- Durability - Changes made by the committed transaction are permanent and must survive system failure.

Ada

A third generation computer programming language. Earlier versions of Oracle's PL/SQL language was based on the *ADA* language syntax and structure.

ADF

Oracle's Application Development Framework (*ADF*) - part of Oracle JDeveloper 10g (also used to develop Oracle e-Business Suite). *ADF* provides a productivity layer for J2EE developers to build applications that are well-architected, portable, and high-performance.

Advanced Queuing

Oracle AQ (Advanced Queueing) provides a message queuing infrastructure as integral part of the Oracle server engine. It provides an API for enqueing messages to database queues. These messages can later be dequeued for asynchronous processing. Oracle AQ also provides functionality to preserve, track, document, correlate, and query messages in queues.

Advanced Replication

Oracle database replication technique in which parts, or the entire contents of a database, are replicated between several sites. All sites can update data simultaneously. Replication conflicts are detected and resolved through default, or site specific conflict handling procedures. Also see Basic Replication.

Aggregation

A process of grouping distinct data. The aggregated data set has a smaller number of data elements than the input data set.

AIX

Advanced Interactive Executive - IBM's version of Unix and one of the platforms Oracle runs on.

Alert Log

The Alert log file is a chronological log of messages and errors. Oracle will create a new file whenever the old one is deleted. Typical messages found in this file is: database startup, shutdown, log switches, space errors, etc.

Alias

Giving a temporary name to a column or table in a SQL statement. In this example, BOSS and SLAVE are column aliases, and X and Y are table aliases:

```
select X.ENAME BOSS, Y.ENAME SLAVE
from   EMP X, EMP Y
where  X.EMPNO = Y.MGRNO;
```

Alternate key

Any candidate key that is not the primary key. Also called a secondary key.

Analyze

Oracle SQL command used to compute or estimate statistics for a table or index. If tables and indexes are not analyzed regularly, the Cost Based Optimizer (CBO) will choose non-optimal SQL plans, leading to slow query performance.

Anonymous Block

An unnamed sequence of actions. Since they are unnamed, anonymous blocks cannot be referenced by other program units. This is an example anonymous block written in PL/SQL:

```
begin dbms_output.put_line('Hello world!'); end;
```

ANSI

The American National Standards Institute (*ANSI*) is a privately funded, non-profit organization which coordinates the development and approval of voluntary standards in the United States.

AOL

Application Object Library - the components and GUI standards that Oracle employs when it develops applications. *AOL* is shipped with Oracle's E-Business Suite (Oracle Applications).

API

Application Program Interface. A set of routines provided in libraries that extends a language's functionality.

Applet

A Java program written in such a way that it will run from within a Web browser like Netscape Navigator or Internet Explorer.

Application

A set of Oracle programs that solve a company's business needs.

ARC Relationship

An arc symbol used on a data model to indicate that two or more relationships are mutually exclusive (XOR). For example, a PERSON entity is either an EMPLOYEE or a CUSTOMER, but not both. ARC relationships are implemented as foreign keys on database level.

ARCH

Oracle ARCHiver Process. *ARCH* is an Oracle background process created when you start an instance in ARCHIVE LOG MODE. The *ARCH* process will archive on-line redo log files to some backup media.

ARCHIVELOG Mode

A database mode in which filled on-line Redo Log files are archived as Archive Logs before reusing them in the cycle. *ARCHIVELOG mode* allows the database to guarantee complete data recoverability. In NOARCHIVELOG mode on-line log files will be overwritten when required and no copies will be archived.

Archiving

Copying of on-line redo log files to the archive destination. This is done by the ARCH process if the database is in ARCHIVELOG mode. *Archiving* should be turned on if you want to do on-line database backups.

Arithmatic Operators

Symbols used to define mathmatical operations with data. Common operators are +, -, *, and /.

Array

A series of variables (or objects) that are of the same type and size. Each of these variables (objects) are indexed; individual elements are called *array* elements. Arrays can be used in SQL and PL/SQL programs to reduce programming time and improve performance.

Artificial key

See surrogate key.

ASCII

The American Standard Code for Information Interchange (*ASCII*) is used extensively in data transmission. The *ASCII* character set includes 128 upper and lower case letters, numerals and special purpose symbols, each encoded by a unique 7-bit binary number.

ASP

Active Server Pages. The default scripting language used for writing *ASP* is VBScript, although you can use other scripting languages like JScript. Also see PSP, JSP and JSP.

Attribute

Each *attribute* of a relation stores a piece of information about an object. Attributes are represented as columns in a table. Each *attribute* in a relation is unique and contains atomic values. The number of attributes in a relation is called the degree of the relation.

Audit

An Oracle SQL statement for auditing statements, privileges and objects.

Auditing

The process of recording database activity and access to database objects as it occurs in the database. See the audit statment.

Authentication

The process of determining whether someone or something is, in fact, who or what it claim to be.

Authorization

The granting of authority, which includes granting access based on access rights.

B

B*Tree

Data structure used by Oracle for storing indexes. A B*Tree index consists of levels of branch blocks, each level containing pointers to the next lower level, with a set of leaf blocks at the lowest level.

B2B

Business to Business.

B2C

Business to Consumer.

Backbone

A central high speed network that connects smaller, independent networks.

Backup

Copying or saving data to a different location. One can restore the *backup* copy if data is lost or damaged.

BASIC

The Beginners' All-purpose Symbolic Instruction Code (*BASIC*) is a computer language developed by Kemeny & Kurtz in 1964. Products like Oracle Power Objects and Microsoft Visual *Basic* uses *Basic* as its programming language.

Basic Replication

Type of data replication in which read-only materialized viewes (snapshots) are used to replicate data between sites. Also see Advanced Replication.

BEQ

See Bequeath Protocol

Bequeath Protocol

A SQL*Net protocol that is similar to the IPC protocol in that it is only used for local connections (when client and server programs reside on the same system). BEQ conections does not require a listener and can only establish dedicated server connections.

Beta

The last software testing phase before a production rollout. Also see UAT.

BI

Business Intelligence. Technologies that help companies make better business decisions.

Binary

A numbering system with only two values: 0 (zero) and 1 (one).

Bind Variable

A bind variable is a place-holder variable in a SQL statement that must be replaced with a valid value (or address of a value) before the statement can successfully execute. For example, "DELETE FROM EMPLOYEES WHERE EMPNO = :EMPNO_BIND_VAR".

BIT

Binary digit. A byte consists of 8 bits.

Bitmap index

A type of index that uses a string of bits to quickly locate rows in a table. Bitmap indexes are normally used to index low cardinality columns in a warehouse environment.

BLOB

A *BLOB* (Binary Large Object) is an Oracle data type that can hold up to 4 GB of data. Also see CLOB. *BLOB*'s are handy for storing digitized information (e.g., images, audio, video)

Block

A container for items (eg. fields) in Oracle Forms. Blocks can be related to tables in the database.

Boolean

PL/SQL Data Type used to declare variables that can take one of the following values: TRUE, FALSE or NULL. Please note that "boolean" cannot be used as the data type of a column. Use something like this instead:

```
... col1 NUMBER(1,0) CHECK (col1 IN (1,0)),
... col2 VARCHAR2(1) CHECK (col2 IN ('T','F')),
```

Bootstrap segment

An Oracle data block in the SYSTEM tablespace containing code that is used to start a database.

Bounce

A database is bounsed or recycled when it is shutdown and re-started.

Buffer Cache

The portion of the SGA that holds copies of Oracle data blocks. All user processes that connect to an instance share access to the buffer cache. Performance of the buffer cache is indicated by the BCHR (Buffer Cache Hit Ratio).

The buffers in the cache are organized in two lists: the dirty list and the least recently used (LRU) list. The dirty list holds dirty buffers, which contain data that has been modified but has not yet been written to disk. The least recently used (LRU) list holds free buffers (unmodified and available), pinned buffers (currently being accessed), and dirty buffers that have not yet been moved to the dirty list.

Bug

A mistake, or unexpected occurrence, in a piece of software or hardware.

BYTE

A *byte* is a series of 8 bits. Also called a character. Computer storage space is measured in bytes. A kilobyte (or 1 KB) represents 1024 bytes. A megabyte (1 MB) represents 1024 KB. A gigabyte represents 1024 MB.

C

Cache

A memory area where frequently accessed data can be stored for rapid access.

Candidate key

A key that uniquely identifies rows in a table. Any of the identified candidate keys can be used as the table's primary key. Any of the candidate keys that is not part of the primary key is called an alternate key.

Canvas

A *canvas* is the surface, in Oracle Forms, on which interface items and prompts are drawn. Canvasses are displayed in a window.

Cardinality

The number of rows in a table or the number of indexed entries in a defined index.

Cartesian Join

This is a join of every row of one table to every row of another table. Something to be avoided!

CASE

Computer-Aided Software Engineering (*CASE*) is a collection of tools and techniques that promise revolutionary gains in analyst and programmer productivity. The two prominent delivered technologies are application generators and PC-based workstations that provide graphics-oriented automation of the development process.

Catalog

See Data Dictionary.

CBO

See Cost Based Optimizer.

CBT

Computer-Based Training (*CBT*) is training which is delivered via a computer. Computer-based training includes tutorials, drill and practice, simulations and testing. Also see ILT.

CDE

Oracle's Cooperative Development Environment (*CDE*) is a set of CASE and development tools for modeling and application generation; application development tools for creating complex forms and report; end user tools for ad hoc query, reporting, and graphical display of information; and other tools for access to third party software and data architectures. *CDE* provides support for graphical user interfaces such as Microsoft Windows, Macintosh, OS/2 Presentation Manager, and Motif, as well as character mode devices. Additionally, *CDE* tools provide full support for multimedia data and display including text, sound, images, and full motion video.

CFS

Cluster File System.

CGA

Call Global Area - part of the PGA. The *CGA* is created at the beginning of a call, it exists for the duration of that call, and is freed when the call completes. The *CGA* only exists while a process is running. If a process is not running, there is no *CGA* within its PGA.

CGI Script

The Common Gateway Interface (CGI) is an application programming interface (API) for writing programs that perform functions on the Web. The Oracle WebServer supports this standard, but also offers the Web Request Broker as an alternative.

char

Data type used to store fixed-length character data. A *char* value can contain up to 2000 bytes of data. If you insert a value shorter than the specified length, Oracle will blank-pad the value. If you insert a value that is too long, Oracle will return an error. Also see VARCHAR2.

Character Set

An encoding scheme in which each character is represented by a different binary value. For example, ISO8859-1 is an extended Latin *character set* that supports more than 40 Western European languages.

Checkpoint

A *checkpoint* occurs when the DBWR (database writer) process writes all modified buffers in the SGA buffer cache to the database data files. Checkpoints occur AFTER (not during) every redo log switch and also at intervals specified by initialization parameters. Set parameter LOG_CHECKPOINTS_TO_ALERT=TRUE to observe *checkpoint* start and end times in the database alert log. Checkpoints can be forced with the "ALTER SYSTEM *CHECKPOINT*;" command. Also see CKPT.

CKPT

CKPT (Oracle Checkpoint Process) is the Oracle background process that timestams all datafiles and control files to indicate that a checkpoint has occurred.

CLI

Command Line Inteface

Client/Server

Client/Server is a term that refers to an architecture where client code/programs are separate from the server code/programs. Client and server portions can run on the same or different computer systems. For example in a database environment a client program can connect across the network to an Oracle database running on a database server.

CLOB

A *CLOB* (Character Large Object) is an Oracle data type that can hold up to 4 GB of data. Also see BLOB.

Cluster

- A *cluster* is an oracle object that allows one to store related rows from different tables in the same data block. Table clustering is very seldomly used by Oracle DBA's and Developers.
- Two or more computers that share resources and work together to form a larger logical computing unit. RAC and Oracle Parallel Server can be used to access Oracle from multiple nodes of a clustered configuration.

Cluster Join

A nested loops join involving two or more tables that are stored together in a cluster. Also see Sort Merge Join, Nested Loops Join and Hash Join.

Clustered Index

An SQL Server term for an index that shares the storage of its table. A clustered index is not like an Oracle cluster, but more like and Oracle IOT (Index-Organized Table).

CMAN

Connection manager. The Oracle Connection Manager (*CMan*) is a Net8 process that relays network traffic to a different address, and optionally changes its characteristics.

The Connenction manager is commonly used for the following:

- Connenction Concentration
- Access Control
- Multiprotocol Support
- Provide Java applets to connect to a database that is not on the machine the Java applet was downloaded from.

Coalesce

To unite or merge into one. For example, when you *coalesce* a tablespace, small chunks of free space will e merged into larger chunk.

Column

Component of a relational table (the other is called a row). Every table in a relational database has one or more

columns. A *column* is named and contains related information. In non-relational terminology, a *column* can be thought of as a field.

COM

Component Object Model (*COM*) is Microsoft's object-oriented programming model that defines how objects interact within a single application or between applications. In *COM*, client software accesses an object through a pointer to an interface (a related set of funcations called methods) on objects. Both OLE and ActiveX are based on *COM*. IBM's verion of *COM* is called SOM.

Command

Statement presented by a human and accepted by a computer in such a manner as to make the human feel as if he is in control.

Commit

An Oracle reserved word instructing the database to save all changes made to the database. Also see Rollback and Savepoint.

Composite Key

A primary key which consists of more than one column. Also known as the Concatenated Key. For example, a table LINE_ITEMS may have a *composite key* {orderID, itemNumber}.

Concurrent Manager

Oracle e-Business suite's workload management utility. Work are queued and executed by the *Concurrent Manager* in the background, allowing users to continue to use their terminals.

Constraint

Data rule or restriction that is enforced within the database rather than at application or object level. The following *constraint* types are available in Oracle: primary key, unique, foreign key (references), check, NOT NULL, etc.

Control File

- An important database file containing information needed to maintain and verify database integrity. Every time an Oracle instance mounts a database, the control file is read to locate data and redo log files. All control files is updated continuously during database use.
- File used by SQL*Loader to guide data loading activities.

CORBA

CORBA or Common Object Request Broker Architecture is a language-independent object model and specification for a distributed applications development environment. See http://www.omg.org for details.

Cost Based Optimizer

Cost Based Optimizer (CBO) - SQL Query optimizer that uses data statistics to identify the query plan with lowest cost for execution. The cost is based on the number of rows in a table, index efficiency, etc. Also see RBO. All applications should be converted to use CBO as RBO will not be available in Oracle 10 and above.

CRUD

Create, Retrieve, Update and Delete

CSI

A CPU Support Identification (*CSI*) number is issued to Oracle Support Customers and must be quoted every time you log a fault with Oracle Support.

CSV

A file with Comma Separated Values. One can sava a Microsoft Excel spreadsheet to a *CSV* file and use Oracle's SQL*Loader program to upload the *CSV* file into a table.

Cube

See Data Cube.

Cursor

Cursors are pointers used to fetch rows from a result set. One can think of a *cursor* as a data structure that describes the results returned from a SQL SELECT statement. One of the variables in this structure is a pointer to the next record to be fetched from the query results.

CVS

Concurrent Versioning System. *CVS* is an open source version control and collaboration system.

D

DA

See Data Administrator.

DAD

A Database Access Descriptor (*DAD*) contains the information needed by products like Oracle Portal, WebDB and Oracle Internet Application Server to connect to an Oracle database. Information in a *DAD* includes: username, password and Net8 connect-string. One can create DADs from the Portal / OAS (Oracle Application Server) Manager web-page.

Data

Unprocessed raw information. *Data* is normally stored in a database or a file.

Data Administration

Organizational function that plans and looks after institutional data. Typical functions include: data modeling, naming standards for databases and data elements, manage metadata (data dictionaries), maintain data access policies, etc.

Data Administrator

Person doing Data Administration functions.

Data Cube

A multi-dimensional representation of data in which the cells contain measures (i.e. facts) and the edges represent data dimensions by which the data may be reported (sliced and diced). For example: A SALES cube can have measures "PROFIT" and "COMMISSION". The dimensions can be

TIME, PRODUCT, REGION, SALESPERSON, etc. Cubes can be defined with Oracle's Warehouse Builder tool.

Data Dictionary

A repository of metadata (information about data), such as its meaning, relationships to other data, origin, usage and format. The dictionary assists company management, database administrators, systems analysts and application programmers in effectively planning, controlling and evaluating the collection, storage and use of data.

Data Dimensions

Properties by which data cubes are described. For example, mesures in a SALES cube can be reported by TIME, PRODUCT, REGION, SALESPERSON, etc.

Data Independence

Hide implementation and storage details from programs that use the data. DBMS systems, like Oracle, provide physical and logical independence as data can be managed separately from the applications that use the data.

Physical Data Independence - Capability to change the physical schema without effecting access to the data. For example when one re-partitions a table, add indexes, or change a table's storage organization.

Logical Data Dependence - Capability to change the logical schema without changing application programs. For example, one can add views, triggers and constraints without effecting the application.

Data Mart

A subset of information relevant to a group of users which is transferred to a separate departmental server. The database involved can be relational, although a multi-dimensional OLAP server is often more appropriate. A range of end-user tools can be used to access the data in a data mart. Examples: Oracle Discoverer, Business Objects, etc.

Data Mining

The process of detecting hidden patterns, trends and associations witin data stored in a large databses like a data warehouse. Typical methods used: neural networks, swarm intelligence, pattern recognition, and statistical analysis.

Data Model

The logical data structure developed during the logical database design process is a data model or entity model. It is also a description of the structural properties that define all entries represented in a database and all the relationships that exist among them.

Data Type

Unit of data storage in a software system. Some of Oracle's built-in data types are: NUMBER, CHAR, VARCHAR2, DATE, TIMESTAMP, BLOB, CLOB, etc. Abstract data types can also be defined.

Data Warehouse

A separate database dedicated to decision support. Data is transferred from transaction processing systems and integrated. It is accessed to provide management information through report writers, query tools, data access and retrieval tools, OLAP servers and enterprise information systems. It is a software architecture, not a product.

Database

A *database* is a collection of data stored together as a unit. Databases are useful for storing data and making it available for retrieval. Within the *database*, data is organized into different tables. Each table has columns and rows. Indexes on tables provide speedy access to data. Information in the *database* can be retrieved, modified, or deleted using a query language like SQL. Some common *database* systems are Oracle, SQL Server, DB/2, Sybase, etc.

Database Administration

Job of looking after the database environment and infrastructure. Typical functions include: software installations, backup and recovery, create and manage databases and database objects, database design, database tuning, database security, etc.

Database Administrator

The Database Baby Sitter. Person that performs Database Administration duties.

Database Engine

The various processes and tools that interpret commands, retrieve and modify data etc. Also called the DBMS.

Database Link

One-way pointer to a remote Oracle Database. A connection is established when information is requested from the remote database.

DATE

Data type used to store *date* and time values in a 7-byte structure. Also see TIMESTAMP.

DB

See Database.

DB/2

DB2 is a DBMS provided by IBM.

DBA

See Database Administrator.

DBCA

Database Configuration Assistant (*DBCA*) - utility for creating, configuring and removing Oracle databases. One can setup templates to automate and standardise the creation of new databases or generate the scripts required to create a database later.

DBF

Typical file name extention (*.dbf) used for Oracle Database Files.

DBMS

DataBase Management System (sometimes also referred to as a Database Manager or Database Engine). Software designed to manipulate the information in a database. It can add, delete, modify, sort, display and search for specific

information, and perform many other tasks on a database. Examples of legacy flat-file *DBMS* systems are: dBase, Clipper, FoxPro and DataEase. Some of the specialized *DBMS* types in existence are: RDBMS (most popular), ODBMS, ORDBMS, HDBMS, NDBMS, etc.

DBU.NET

Utility to connect to Oracle databases from a Microsoft .NET environment.

DBUA

Database Upgrade Assistant - utility used for upgrading Oracle databases from one version to the next.

DBV (DB Verify)

DB Verify is an Oracle utility shipping with Oracle releases 7.3.2 and above. It is used to check the integrity of database data files.

Example usage on Unix: *dbv file=/u01/oradata/ORCL/sysORCL.dbf*
Windows NT example: *DBVERFxx FILE=d:\orant\database\systORCL.dbf*

Notes: If your file extension is not ".dbf", you might need to setup a symbolic link (alias) to a dbf name before using DBV.
For RAW devices you should use START and END parameters.

DBWR

DBWR (Oracle DataBase WRiter) is an Oracle background process created when you start a database instance. The *DBWR* writes data from the SGA to the Oracle database files. When the SGA buffer cache fills the *DBWR* process selects buffers using an LRU algorithm and writes them to disk.

DCL

Data Control Language. The category of SQL statements that control access to the data and to the database. Examples are the GRANT and REVOKE statements.

DDL

DDL Data Definition Language. A language used by a database management system which allows users to define the database, specifying data types, structures and constraints on the data. Examples are the CREATE TABLE, CREATE INDEX, ALTER, and DROP statements.

Note: *DDL* statements will implicitly commit any outstanding transaction.

Decode

Oracle SQL function to handle conditional logic. Example:

```
select decode(SEX, 'M', 'Male', 'F', 'Female', 'Unknown')
from EMP;
```

Dedicated Server

A *dedicated server* is an Oracle background process that remains associated to the user process for the remainder of the connection. Also see MTS and Shared Server.

Defacto Standard (Proprietary Standard)

A standard which has been endorsed by industry or government, but not officially approved by an accredited standards body such as ISO.

Delete

DML command used to remove data from a table. Also see Truncate.

Denormalization

The opposite of data normalization (almost). In a denormalized database, some duplicated data storage is allowed. The benefits are quicker data retrieval and/or a database structure that is easier for end-users.

Designer/2000

See Oracle Designer.

Developer/2000

See Oracle Developer.

Dictionary Cache

Cache used to speed-up data dictionary lookups. Also known as the row cache, because it holds data as rows instead of buffers (which hold entire blocks of data).

Dimension Table

A table, typically in a data warehouse, that contains further information about an attribute in a fact table. For example, a SALES table can have the following dimension tables TIME, PRODUCT, REGION, SALESPERSON, etc. Also see Data Dimensions.

Discoverer/2000

See Oracle Discoverer.

Distributed Database

A database whose objects (tables, views, columns and/or files) reside on more than one system in a network, and can be accessed or updated from any system in the network.

Distributed System

Refers to computer systems in multiple locations throughout an organization working in a cooperative fashion, with the system at each location serving the needs of that location but also able to receive information from other systems, and supply information to other systems within the network.

DML

Data Manipulation Language. The data manipulation language for Oracle is SQL. These commands manipulate data within existing database objects. Examples are the SELECT, UPDATE, INSERT, DELETE statements.

Note: You must explicitly state COMMIT or ROLLBACK to complete a *DML* transaction.

DMT

Dictionary Managed Tablespace. Also see LMT.

Domain

The set of different values that a table's column can have.

Domain Name Server (DNS)

A system that translates strings of word segments (denoting user names and locations) into numeric Internet addresses. Eg. www.oracle.com —> 192.86.154.104

Driving Table

The table in a SQL query's FROM clause that is read first when joining data from two or more tables. The Rule Based optimizer will always choose the last table in the FROM clause as the driving table. The Cost Based Optimizer should choose a more appropriate table (likely the one with the least amount of rows) as the driving table. If the wrong table is chosen, the query will perform more I/O requests to return the same data.

DRP

Disaster Recovery Plan (*DRP*). Plan to resume or recover, a specific essential operation, function or process of an enterprise. Database *DRP* plans normally include Backup and Recovery Procedures, Standby Databases, Data Replication, Fail-save options, etc.

DSL

Data Sub Language - a language concerned with database objects and operations. In SQL terms, *DSL* is a combination of both DDL and DML.

DSS

Decision Support System: Interactive computer-based systems intended to help decision makers utilize data and models to identify and solve semistructured (or unstructured) problems.

DUAL

A view owned by the SYS user containing one row with value 'X'. This is handy when you want to select an expression and only get a single row back.
Sample Usage: *select sysdate from DUAL;*

According to legend this table originally contained two rows, from there the name *DUAL*.

Dump File

Normally refers to a binary file containing exported data that can be re-imported into Oracle. Can also refer to a trace file.

Dynamic SQL

SQL statement that is constructed and executed at program execution time. In contrast to this, Static SQL statemets are hard-coded in the program and executed "as-is" at run-time. Dynamic SQL provides more flexability, nevertheless, static SQL is faster and more secure than dynamic SQL. Also see Embedded SQL.

E

E-Mail

Electronic Mail.

EBCDIC

Extended Binary-Coded Decimal Interchange Code. A standard character-to-number encoding (like ASCII) used by some IBM computer systems. For example, Oracle on OS390 (IBM MVS) stores data as *EBCDIC* characters.

EBU

Enterprise Backup Utility (Oracle 7). *EBU* was superceded by RMAN.

Electronic Data Interchange (EDI)

The inter-organizational, computer-to-computer exchange of structured information in a standard, machine-processable format.

Embedded SQL

Embedded SQL statemets are hard-coded in the program and executed "as-is" at run-time. Emedded SQL is also known as static SQL. Except for host bind variables, these statement cannot be altered at run rime. Also see Dynamic SQL.

Encapsulation

Encapsulation describes the ability of an object to hide its data and methods from the rest of the world - one of the fundamental principles of OOP (Object Oriented Programming).

Enterprise

Is a collection of organizations and people formed to create and deliver products to customers.

Enterprise JavaBeans

A Java standard for creating reusable server components for building applications. They facilitate writing code that accesses data in a database.

Entity

An *entity* is a thing of significance, either real or conceptual, about which the business or system being modeled needs to hold information. Sample entities: EMPLOYEE, DEPARTMENT, CAR, etc. Each *entity* in a ERD generally correspond to a physical table on database level.

Entity-Relationship diagram

A diagram showing entities and their relationships. Relates to business data analysis and data base design. Entity-Relationship Diagrams can be constructed with Oracle Designer. Also see UML.

Equi Join

An Equi Join (aka. Inner Join or Simple Join) is a join statement that uses an equivalency operation (i.e: colA = colB) to match rows from different tables. The converse of an equi join is a nonequijoin operation.

ER Diagram

See Entity-Relationship diagram.

ERD

See Entity-Relationship diagram.

ERP

Enterprise Resource Planning. An information system that integrates all manufacturing and related applications for an entire enterprise.

ETL

Data Warehouse acquisition processes of Extracting, Transforming (or Transporting) and Loading (*ETL*) data from source systems into the data warehouse.

Exception

Error control/recovery mechanism in PL/SQL.

Execution Plan

The operations that the Oracle Server performs to execute a SQL statement. Also see Explain Plan.

EXP

Oracle utility used to export data (and schema definitions) from an Oracle database to a proprietary binary file format. This file can be re-imported into Oracle databases across various platforms. Also see IMP.

Expert System

A software system with two basic components: a knowledge base and an inference engine. The system mimics an expert's reasoning process.

Explain Plan

A report that shows how Oracle plans to execute a given SQL query to retrieve the requested data. Commonly used by developers and DBA's to diagnose poorly performing SQL queries. For example, check if the query is doing a quick index lookup or a lengthy full table scan. Also see Execution Plan.

Express

Oracle *Express* is a multi-dimensional database and engine used for OLAP analysis. See the Express FAQ for more details.

Extent

An *extent* is a contiguous set of Oracle blocks allocated to a segment in a tablespace.

The size of an *extent* is controlled by storage parameters used when you CREATE or ALTER the segment, including INITIAL, NEXT and PCT_INCREASE.

External table

A table that is not stored in an Oracle database. Data gets loaded via an access driver (normally ORACLE_LOADER) when the table is accessed. One can think of an external table as a view that allows running SQL queries against external data without requiring that the data first be loaded into the database.

F

Fact Table

A table, typically in a data warehouse, that contains the measures and facts (the primary data). Also see Dimension Table.

FAQ

Frequently Asked Questions. A *FAQ* is a list of answers to Frequently Asked Questions. On the Internet a *FAQ* may exist as a feature of an interest groups or a mailing list. Each *FAQ* addresses a specific topic with a list of questions and their answers.

Field

In an application context a *field* is a position on a form that is used to enter, view, update, or delete data.

In a database context a *field* is the same as a column. Aslo see column.

File

A collection of related data stored together for later use. A *file* is stored in a directory on a file system on disk.

File System

Method of storing and organizing files on disk. Some of the common file systems are: FAT and NTFS on Windows Systems and UFS and JFS on Unix Systems.

File Transfer Protocol

File Transfer Protocol (FTP) - A way of transferring files between computers. A protocol that describes file transfers between a host and a remote computer.

FIPS (Federal Information Processing Standard)

Standards published by the U.S. National Institute of Standards and Technology, after approval by the Dept. of Commerce; used as a guideline for federal procurements.

Firewall

A computer system that sits between the Internet and a company's network. It acts as an active gateway to keep non-company entities from accessing company confidential data.

Foreign key

A column in a table that does not uniquely identify rows in that table, but is used as a link to matching columns in other tables.

FORMS

See Oracle Forms.

Fragmentation

The scattering of data over a disk caused by successive insert, update and delete operations. This eventually results in slow data access times as the disk needs to do more work to construct a contiguous copy of the data on disk. A database reorganization is sometimes required to fix *fragmentation* problems.

Freelist

When records are inserted into a table, Oracle examines the *freelist* to determine which data blocks have available storage space that can be used to store the new rows.

FTP

See File Transfer Protocol.

FUD

Short for Fear, Uncertainty, and Doubt.

Function

Block of PL/SQL code stored within the database. A *function* always returns a single value to its caller. Also see Package and Procedure.

G
GB

1 *GB* (Gigabyte) is 1024 MB. See BYTE.

GIF

A standard graphics file format used on the Web, recognized by all Web browsers.

GIGO

Garbage In, Garbage Out. Computer output is only as good as the data entered.

GIS

Geographic Information System. A computer software system with which spatial information (eg. maps) can be captured, stored, analyzed, displayed and retrieved.

Glue

Oracle Objects for OLE's (OO4O) predecessor was called Oracle *Glue*.

Grid Computing

Applying resources from many computers in a network to a single problem or application.

Group Function

Oracle functions that groups data. Eg: AVG, COUNT, MIN, MAX, STDDEV, SUM, VARIANCE, etc. Example usage: select MIN(sal), MAX(sal), AVG(sal) from emp;

GUI

Graphical User Interface. Some popular *GUI* environments: Linux KDesktop, Microsoft Windows, Macintosh, Sun Openlook and HP Motif.

H
HA

High Availability. Measures that can be implemented to prevent the entire system from failing if components of the system fail.

Hash function

A formula that is applied to each value of a table column or a combination of several columns, called the index key, to get the address of the area in which the row should be stored. When locating data, the database uses the hash function again to get the data's location.

Hash Join

Join optimization method where two tables are joined based on a hashing algorithm. Also see Sort Merge Join, Nested Loops Join and Cluster Join.

Hashing

The conversion of a column's primary key value to a database page number on which the row will be stored. Retrieval operations that specify the key column value use the same *hashing* algorithm and can locate the row directly. *Hashing* provides fast retrieval for data that contains a unique key value.

HDBMS

Hierarchical Database Management System. Type of DBMS that supports a hierarchical data model. Example *HDBMS* Systems: IMS and System 2000.

Heap-organized Table

A table with rows stored in no particular order. This is a standard Oracle table; the term "heap" is used to differentiate it from an index-organized table or external table.

Hierarchical data model

Data model that organizes data in a tree structure. Records are connected by parent-child relationships (PCR). The hierarchical data model is commonly used for LDAP directories and HDBMS databases.

Hint

Code embedded into a SQL statement suggesting to Oracle how it should be processed. Some of the available hints: ALL_ROWS, FIRST_ROWS, CHOOSE, RULE, INDEX, FULL, ORDERED, STAR. Example suggesting a FULL TABLE SCAN method:

```
SELECT /*+ FULL(x) */ FROM tab1 x WHERE col1 = 10;
```

Histogram

Frequency distribution. Metadata describing the distribution of data values within a table. Histograms are used by the Oracle Query Optimizer to predict better query plans. The ANALYZE command is used to compute histograms.

Host

- Command in SQL*Plus and Oracle Forms that runs an operating system command.
- Machine on which an Oracle server resides

HP

Hewlett Packard - One of the computer systems that Oracle runs on. Operating system is HP-UX.

HP-UX

Operating system used on HP machines.

HTML

Hyper Text Mark-Up Language (*HTML*), a subset of Standard Generalized Mark-Up Language (SGML) for electronic publishing, the specific standard used for the World Wide Web.

HTTP

Hyper Text Transfer Protocol (*HTTP*), the actual communications protocol that enables Web browsing.

Hypertext

Textual data which is "linked" across mutliple documents or location

I

I/O

Input/ Output operations. For example, reading from a disk, writing to a printer, etc.

iAS

Oracle Internet Application Server (*iAS*). Software package that provides Web/HTTP Services, Data Caching, Portal Services, Forms Services, etc.

IBM

International Business Machines Corporation. Company that develops hardware, operating systems, database systems, and applications that work with (and sometimes compete with) with Oracle products.

IEEE (Institute of Electrical and Electronics Engineers)

Organization of engineers, scientists and students involved in electrical, electronics, and related fields. It also functions as a publishing house and standards-making body.

iFS

Oracle's Internet File System allows one to store files in an Oracle database. Access allowed from Standard Windows (SMB protocol), FTP, POP3, HTTP, etc.

ILT

Instructor Led Training (ITL) is Oracle Training classes. Also see TBT and CBT.

IMP

Oracle utility used to import/load data from export files created with the Oracle export utility. Also see EXP.

Impedance Mismatch

Intrinsic difference in the way in which actual data is represented by databases vs. modelling and programming languages are called the *impedance mismatch*. For example, data from relational tables need to be joined to construct single objects. Another example, databases return record sets, while programs process records one-by-one.

IMS

IBM's Information Management System (*IMS*). IMS was developed in 1969 to manage data for NASA's Apollo Moon Landing project. It was later released as the world's first commercially available DBMS. *IMS* supports the hierarchical data model.

Index

A special database object that lets you quickly locate particular records based on key column values. Indexes are essential for good database performance.

Index-Organized Table

Type of table where the data is stored in a B*-tree index structure. Also see Heap-organized Table.

Information

Information is the result of processing, manipulating and organizing data in a way that adds to the knowledge of the person receiving it.

Informix

A Relational Database Management System recently bought out by IBM. It is expected that IBM will integrate Infomix into DB/2.

INIT.ORA

Oracle's initialization parameter file (similar to DB/2's DSNZPARM). On Unix this file is located under *$ORACLE_HOME/dbs/init${ORACLE_SID}.ora*

Initial Extent

The size of the first extent allocated when the object (typically a table or index) is created. Also see Next Extent.

Inner Join

See Equi Join.

Insert

DML command used to add data to a table. Also see Update and Delete.

Instance

An Oracle *Instance* is a running Oracle Database made up of memory structures (SGA) and background processes (SMON, PMON, LGWR, DBW0, etc.). An *instance* only exists while it is up and running. Simply put, a database resides on disk, while an *instance* resides in memory. A database is normally managed by one, and only one, *instance*. However, when using RAC, multiple instances can be started for a single database (on different machines). Each *instance* is identified with a unique identifier known as the ORACLE_SID.

Internet

An electronic network of computers that includes nearly every university, government, and research facility in the world. Also included are many commercial sites. It started with four interconnected computers in 1969 and was known as ARPAnet.

Internet Developer Suite

Oracle Internet Developer Suite (iDS) is a bundling of Oracle development tools like Oracle Forms, Oracle Reports, Oracle Discoverer Oracle Designer and JDeveloper.

InterNIC

The official source of information about the Internet. Its goal is to provide Internet information services, supervise the registration of Internet addresses, and develop and provide databases that serve as white and yellow pages to the Internet.

Intersect

SQL set operation. Select common elements from two different select statements. E.g:

```
select * from table_A INTERSECT select * from table_B;
```

IOR

Oracle v5 DBA utility for starting and stopping databases. *IOR* was later replaced by SQL*Dba in V6 and SQL*Plus in Oracle8i.

IOR INIT - Initialises a new Oracle database for the first time

IOR WARM - Warm start an Oracle database

IOR SHUT - closes down an Oracle database

IOR CLEAR - Like the modern "SHUTDOWN IMMEDIATE"

IOT

See Index-Organized Table.

IPC

Inter Process Communications. A SQL*Net protocol similar to the BEQ protocol in that it is only used for local connections (when client and server programs reside on the same system). *IPC* can be used to establish dedicated server and shared server connections. A listener is required to make *IPC* connections.

ISO

ISO (International Standards Organization) is the International Standards Organizations. They do not create standards but (as with ANSI) provide a means of verifying that a proposed standard has met certain requirements for due process, consensus, and other criteria by those developing the standard.

ISO 9000

ISO 9000 is a series of international standards that provides quality management guidance and identifies quality system elements.

ITIL

IT Infrastructure Library. *ITIL* is an integrated set of best-practice recommendations with common definitions and terminology. *ITIL* covers areas such as Incident Management, Problem Management, Change Management, Release Management and the Service Desk.

ITL

The Interested Transaction List (*ITL*) is an array of 23-byte entries in the variable portion of all Oracle data blocks. Any process that changes data, must store it's transaction id and rollback segment address in an *ITL*.

J

J2EE

Java 2 Platform, Enterprise Edition (*J2EE*) - a version of Java for developing and deploying enterprise applications.

JAR

Java Archive file. An archive (like a ZIP file) contraining Java class files and images. *JAR* files are used to package Java applications for deployment.

Java

An multi-platform, object-oriented programming language from Sun Microsystems. *Java* can be used to program applications and applets.

Java Pool

Memory area in the SGA similar to the Shared Pool and Large Pool. The size of the java pool is defined by the JAVA_POOL_SIZE parameter.

JavaBean

A reusable component that can be used in any Java application development environment. JavaBeans are dropped into an application container, such as a form, and can perform functions ranging from a simple animation to complex calculations.

JavaScript

A scripting language produced by Netscape for use within HTML Web pages.

JBOD

Just A Bunch Of Disks (*JBOD*) - hard disks that aren't configured in a RAID configuration.

JDBC

JDBC (Java Database Connectivity) is a Sun Microsystems standard defining how Java applications access database data.

Join

The process of combining data from two or more tables using matching columns. Also see Equi Join, Outer Join, Self Join, Natural Join, etc.

JPEG

Joint Photograhic Experts Group - a common image format. Art and photographic pictures are usually encoded as *JPEG* files.

JSP

Java Server Pages (*JSP*) are normal HTML with Java code pieces embedded in them. A *JSP* compiler is used to generate a Servlet from the *JSP* page. Also see PSP, PHP and ASP. Example:

```
<H1>Today is:</H1>
<%= new java.util.Date() %>
```

JVM

Java Virtual Machine. A software "execution engine" that runs compiled java byte code (in class files).

K

KB

1 *KB* (Kilobyte) is 1024 bytes. See BYTE.

Kerberos

Kerberos is an Internet Engineering Task Force (IETF) standard for providing authentication. *Kerberos* works by having a central server grant a "ticket" honoured by all networked nodes running *Kerberos*.

Kernel

The heart of an operating system. The *kernel* is the part of the operating system that controls the hardware.

Key Fields

A subset of the fields within a table for which data must be entered and validated before a new record may be added to the table. Failure to correctly enter data in all the key fields will prevent a new record from being added to the table.

KLOC

KLOC - Short for thousands (Kilo) of Lines Of Code. *KLOC* is a measure of a program's (or project's) complexity and size.

L

LAN

Local Area Network. A user-owned and operated data transmission facility connecting a number of communicating devices (e.g. computers, terminals, word processors, printers, and storage units) within a single building or floor.

Large Pool

Memory area in ithe SGA similar to the Shared Pool and Java Pool. The *Large Pool* is mainly used for storing UGA areas (when running in Shared Server mode), and for buffering sequential file IO (i.e. when using RMAN). The size of the *large pool* is defined by the LARGE_POOL_SIZE parameter.

Latch

A *latch* is an internal Oracle mechanism used to protect data structures in the SGA from simultaneous access. Atomic hardware instructions like TEST-AND-SET are used to implement latches. Latches are more restrictive than locks in that they are always exclusive. Latches are never queued, but will spin or sleep until it obtains a resource or times out. Latches are important for performance tuning.

LCKn (Oracle Parallel Server Lock Process)

LCKn is the Oracle background processes created when you start an instance with the Oracle Parallel Server Option (OPS). The number of LCKn lock processes created are determined by the GC_LCK_PROCS=n INIT.ORA parameter.

LDAP

Lightweight Directory Access Protocol. A protocol used to access a directory listing. It is being implemented in Web browsers and e-mail programs to enable lookup queries.

Legacy Data

Existing data that has been acquired by an organization.

Legacy System

An existing system that is deployed in an organization. In the fast moving IT industry, a system is considered stable and "old" as soon as it is properly implemented. Legacy systems will eventually be upgraded, replaced or archived.

LGWR

Oracle Log Writer. *LGWR* is an Oracle background process created when you start a database instance. The *LGWR* writes the redo log buffers to the on-line redo log files. If the on-line redo log files are mirrored, all the members of the group will be written out simultaneously.

Library cache

The library cache is a memory area in the database SGA where Oracle stores table information, object definitions, SQL statements, etc. Each namespace (or library) contains different types of object. See the v$librarycache view for library cache statistics.

Linux

Linux is a free open-source operating system based on Unix. *Linux* was originally created by Linus Torvalds with the assistance of developers from around the globe.

Listener

The Oracle *Listener* is a process listening for incoming database connections. This process is only needed on the database server side. The *listener* is controlled via the "lsnrctl" utility. Configuration is done via the *LISTENER*.ORA file.

LMT

Locally Managed Tablespace. Also see DMT.

Lock

Database locks are used to provide concurrency control. Locks are typically acquired at row or table level. Common *Lock* types are: Shared, eXclusive, Row Share, Row eXclusive, etc. Common uses of locks are:

- ensure that only one user can modify a record at a time;
- ensure that a table cannot be dropped if another user is querying it;
- ensure that one user cannot delete a record while another is updating it.

Log Buffer

See Redo Log Buffer.

Log File

A file that lists actions that have occurred. Also see Alert Log and Redo Log.

Logical Read

A *logical read* occurs whenever a user requests data from the database buffer cache. If the required data is not present, Oracle will perform physical I/Os to read the data into the cache. Oracle keeps track of logical and physical reads in the SYS.V_$SYSSTAT dynamic table.

LogMiner

A component of the Oracle server that lets you parse and view the contents of the archived redo log files. With *LogMiner*, one can generate undo and redo SQL for transactions.

LRU (Least Recently Used)

An algorithm Oracle uses when it needs to make room for new information in the memory space allocated. It replaces the oldest (LRU) data to make room for new data.

LUW

Logical Unit of Work. Also called a database transaction.

M

Marshalling

Marshalling is the process of packaging and sending interface method parameters across thread, process or machine boundaries.

Materialized View

A materialized view (MV) is similar to a view but the data is actually stored on disk (view that materializes). Materialized views are often used for summary and pre-joined tables, or just to make a snapshot of a table available on a remote system. A MV must be refreshed when the data in the underlying tables is changed.

MB

1 *MB* (Megabyte) is 1024 KB. See BYTE.

MDAC

Microsoft Data Access Components - includes ADO, ODBC and OLE DB. If installed, see "C:\Program Files\Common Files\System\ADO\MDACReadMe.txt" for additional info.

Merge

SQL command that performs a series of conditional update and insert operations. Also see upsert.

Metadata

Data that is used to describe other data. Data definitions are sometimes referred to as *metadata.* Examples of *metadata* include schema, table, index, view and column definitions.

Microsoft

A software company, based in the USA, that develops the Windows operating system and SQL Server database management system.

Microsoft Windows

See Windows.

Motif

Graphical user interface specified by the Open Software Foundation and built on the Massachusetts Institute of Technology's X Windows.

MPP (Massively Parallel Processor)

A computer which contains two or more processors which co-operate to carry out an operation. Each processor has its own memory, operating system and hard disk. It is also known as a "shared nothing" architecture. The processors pass messages to each other.

MTS

- MTS (Multithreaded Server) is an Oracle server configuration that uses less memory. With *MTS* a dispatcher process enables many user processes to share a few server processes. While running in *MTS* mode, a user can still request a dedicated server process.
- In the Windows world, *MTS* stands for Microsoft Transaction Server.

Multimedia

Used essentially to define applications and technologies that manipulate text, data, images, voice and full motion video objects.

Mutating Table

"Mutating" means "changing". A mutating table is a table that is currently being modified by an update, delete, or insert statement. When a trigger tries to reference a table that is in state of flux (being changed), it is considered "mutating" and raises an error since Oracle should not return data that has not yet reached its final state.

Another way this error can occur is if the trigger has statements to change the primary, foreign or unique key columns of the table off which it fires. If you must have triggers on tables that have referential constraints, the workaround is to enforce the referential integrity through triggers as well.

MV

See Materialized View.

MVS

See OS390.

MySQL

MySQL is a simple, yet powerful, Open Source Software relational database management system that uses SQL. For more details, see www.mysql.com.

N

Natural Join

A join statement that compares the common columns of both tables with each other. One should check whether common columns exist in both tables before doing a natural join. Example:

```
SELECT DEPT_NAME, EMPLOYEE_NAME FROM
    DEPT NATURAL JOIN EMPLOYEE;
```

Natural key

A key made from existing attributes. Opposite of a Surrogate key.

Navigate

Move between windows, fields, buttons and menus with a mouse, keyboard or other input device.

NDBMS

Network Database Management System. Type of DBMS system that supports the network data model. Example: IBM's IDMS, mainly used on Mainframe Systems.

NDIS

Network Driver Interface Specification. A Microsoft specification for a type of device driver that allows multiple transport protocols to run on one network card simultaneously.

Nested Loops Join

Join optimization method where every row in the driving table (or outer table) is compared to the inner table. Also see Sort Merge Join, Nested Loops Join and Cluster Join.

Net8

NET8 (called SQL*NET prior to Oracle8) is Oracle's client/server middleware product that offers transparent connection from client tools to the database, or from one database to another. SQL*Net/ *Net8* works across multiple network protocols and operating systems.

Neural Network

Artificial Neural Networks (ANN) are non-linear predictive models that learn through training. They atempt to emulate the processing of a biological brain.

Next Extent

The size of each subsequent extent to be allocated to a segment. The size specified may remain constant for each new extent or may change according to to the value of PCTINCREASE. Also see Initial Extent.

NLS

National Language Support is used to define national date, number, currency and language settings. I.e. used to change the currency symbol from $ to € (Euro).

Nonequijoin

A join statement that does not use an equality operation (i.e: colA <> colB). The converse of a *nonequijoin* is a equi join.

Normalization

A series of steps followed to obtain a database design that allows for efficient access and storage of data. These steps reduce data redundancy and the chances of data becoming inconsistent.

First Normal Form eliminates repeating groups by putting each into a separate table and connecting them with a one-to-many relationship.

Second Normal Form eliminates functional dependencies on a partial key by putting the fields in a separate table from those that are dependent on the whole key.

Third Normal Form eliminates functional dependencies on non-key fields by putting them in a separate table. At this stage, all non-key fields are dependent on the key, the whole key and nothing but the key.

Fourth Normal Form separates independent multi-valued facts stored in one table into separate tables.

Fifth Normal Form breaks out data redundancy that is not covered by any of the previous normal forms.

NOS

Network Operating System. The programs that manage the resources and services of a network and provide network security. Examples: Novell Netware, Windows NT.

Null

A *null* value represents missing, unknown, or inapplicable data. Do not use *null* to represent a value of zero, because they are not equivalent. Any arithmetic expression containing a *null* always evaluates to *null*. For example, 10 + *NULL = NULL*. In fact, all operators (except concatenation) return *null* when given a *null* operand.

nvl

Oracle function that will return a non-NULL value if a NULL value is passed to it. Example: SELECT nvl(salary, 'Sorry, no pay!') FROM employees;

nvl2

Oracle function that will return different values based on whether the input value is NULL or not. Syntax: nvl2(input_value, return_if_null, return_if_not_null)

O

OCCI

The Oracle C++ Call Interface (*OCCI*) is a development component based on the OCI (Oracle Call Interfaces) API. *OCCI* makes it easier for developers to develop OCI applications, while maintaining the OCI performance benefit. The Oracle *OCCI* API is modelled after JDBC.

OCI

The Oracle Call Interface (*OCI*) is a set of low-level APIs to perform Oracle database operations (eg. logon, execute, parse, fetch records).

OCI programs are normally written in C or C++, although they can be written in almost any programing language. Unlike with the Oracle Precompilers (like Pro*C), *OCI* programs are not precompiled.

OCP

Oracle Certified Professional. A person who passed all the *OCP* exam tracks.

OCS

Oracle Collaboration Suite. Integrated communications system for E-mail, calendar, fax, voice and files.

ODBC

ODBC stands for Open Data Base Connectivity and was developed by Microsoft. *ODBC* offers connectivity to a wide range of backend databases from a wide range of front-ends. *ODBC* is vendor neutral.

Oracle (and other organizations) provides *ODBC* drivers that allow one to connect to Oracle Databases. Also see MDAC and OLE DB.

ODBMS

Object-oriented Database Management System. A special type of DBMS where data is stored in objects.

ODS

Operational Data Store (*ODS*): An *ODS* is an integrated database of operational data. Its sources include legacy systems and it contains current or near term data. An *ODS* may contain 30 to 60 days of information, while a data warehouse typically contains years of data.

OEM

See Oracle Enterprise Manager.

OERI

OERI is used as a short notation for ORA-600. Commonly used on support sites like Metalink.

oerr

Oerr is an Oracle / Unix utility that extracts error messages with suggested actions from the standard Oracle message files. This utility is very useful as it can extract OS-specific errors that are not in the generic *Error Messages and Codes* Manual.

Usage: $ *oerr* ORA 600

OFA

- Optimal Flexible Architecture (*OFA*) is an Oracle standard for file placement. See your *Installations Guide* for details.
- The Oracle Financial Analyser product, normally used with Oracle Express.

OLAP

Online Analytical Processing. *OLAP* systems allow workers to, quickly, and flexibly manipulate operational data using familiar business terms, in order to provide analytical insight.

OLE DB

OLEDB (Object Linking and Embedding for databases) is a data-access provider used to communicate with both relational and non-relational databases. OLEDB is provided with MDAC. Also see ODBC.

OLE2

Object Linking and Embedding v2, an improved version of DDE. A Microsoft standard that allows data from one application to be "drag and drop" into another application in such a way that you can edit the object using the first application's capabilities without leaving the second application.

OLTP

On-Line Transaction Processing (*OLTP*) systems capture, validate and store large amounts of transactions. *OLTP* systems are optimized for data entry operations and consist of large amounts of relatively short database transactions. Example: an Order-Entry system.

OMS

OMS (Oracle Management Server) is part of the OEM (Oracle Enterprise Manager) architecture. *OMS* is the middle tier between OEM consoles and database servers.

OO4O

Oracle Objects for OLE. A custom control (OCX or ActiveX) combined with an OLE in-process server that lets you use native Oracle database functionality within Windows applications.

Open System

A system capable of communicating with other open systems by virtue of implementing common international standard protocols.

Operating System

The software that manages a computer system (schedule tasks and control the use of system resources). An operating system is made up of a kernel and various system utility programs. Example operating systems: Linux, Microsoft Windows, Unix, OS390, etc.

OPMN

OPMN (Oracle Process Manager and Notification Server) allows one to manage Oracle Application Server processes. It consists of the following subcomponents: Oracle Notification Server (ONS), Oracle Process Manager (PM) and Oracle Process Manager Modules (PM Modules). *OPMN* allows one to start, stop, monitor and manage processes with the "opmnctl" command-line utility. For example, to start *opmn* and all managed processes, use the "opnmctl startall" command. To list all services, use the "opmnctl status -l" command.

OPO

See Oracle Power Objects.

OPS

See Oracle Parallel Server and RAC.

ORA

- An *ORA* (Operational Readiness Assessment) is an assessment of a customer system provided by Oracle Consulting.
- Sometimes used as

P

Package

A *package* is a stored collection of procedures and functions. A *package* usually has a specification and a body stored separately in the database. The specification is the interface to the application and declares types, variables, exceptions, cursors and subprograms. The body implements the specification.

When a procedure or function within the *package* is referenced, the whole *package* gets loaded into memory. So when you reference another procedure or function within the *package*, it is already in memory.

Parity

An error dectection scheme that uses an extra checking bit, called the *parity* bit, to allow the receiver to verify that the data is error free.

Parse

Analysis of the grammar and structure of a computer language (like SQL).

Parse Tree

A parsed representation of the grammar of a computer language. This parsed representation is stored in a tree structure. For example, the grammar of a SQL statement must be parsed into a parse tree before it can be understood and executed by a computer.

Partitioning

Feature of the Oracle Database to store data in partitions (or sub-tables).

Patch

Software update designed to repair known problems or "bugs" in previous software releases.

PCTFREE

Block storage parameter used to specify how much space should be left in a database block for future updates. For example, for PctFree=30, Oracle will keep on adding new rows to a block until it is 70% full. This leaves 30% for future updates (row expansion).

PCTINCREASE

The percentage by which each next extent (beginning with the third extend) will grow. The size of each subsequent extent is equal to the size of the previous extent plus this percentage increase.

PCTUSED

Block storage parameter used to specify when Oracle should consider a database block to be empty enough to be added to the freelist. Oracle will only insert new rows in blocks that is enqueued on the freelist. For example, if PctUsed=40, Oracle will not add new rows to the block unless sufficient rows are deleted from the block so that it falls below 40% empty. This parameter is ignored for objects created locally managed tablespaces with Segment Space Management specified as AUTO.

PGA

The Program Global Area is a nonshared per process memory area in Oracle. Also called Process Global Area. The *PGA* contains a variable sized chunk of memory called the Call Global Area (CGA). If the server is running in dedicated server mode, the *PGA* also contains a variable chunk of memory called the User Global Area (UGA).

PHP

PHP is a recursive acronym for "PHP Hypertext Preprocessor". It is an open source, interpretive, HTML centric, server side scripting language. *PHP* is especially suited for Web development and can be embedded into HTML pages. Also see PSP, JSP and ASP.

Pivot Table

A data mining feature that enables one to summarize and analyse large amounts of data in lists and tables. Pivot tables can quickly be rearranged by dragging and dropping columns to different row, column or summary positions. Pivot tables are frequently used in products like Oracle Discoverer, Business Objects and Microsoft Excel.

PL/SQL

PL/SQL is Oracle's Procedural Language extension to SQL. *PL/SQL*'s language syntax, structure and data types are similar to that of ADA. The language includes object oriented programming techniques such as encapsulation, function overloading, information hiding (all but inheritance), and so, brings state-of-the-art programming to the Oracle database server and a variety of Oracle tools.

PL/SQL Table

An associative array (or INDEX-BY table) that can be indexed by NUMBER or VARCHAR2. Elements are retrieved using number or string subscript values. Unlike with data tables, PL/SQL tables are stored in memory. PL/SQL Tables are sparse and elements are unordered.

PMON

Oracle Process MONitor. *PMON* is an Oracle background process created when you start a database instance. The *PMON* process will free up resources if a user process fails (eg. release database locks).

Port

A number that TCP uses to route transmitted data to and from a particular program. For example, the Oracle Listener listens for incoming connections on a predefined *port* number.

POSIX (Portable Operating System Interface)

This standard defines a C programming language interface to an operating system environment. This standard is used by computing professionals involved in system and application software development and implementation.

PostgreSQL

PostgreSQL is an Open Source Software object relational database management system. For more details, see www.PostgreSQL.com.

Precompilers

Precompilers are used to embed SQL statements into a host language program. Oracle provides *precompilers* for the various host languages such as Pro*C, Pro*COBOL, Pro*ADA, etc.

Predicate

Syntax that specifies a subset of rows to be returned. Predicates are specified in the WHERE clause of a SQL statement.

Primary key

A column in a table whose values uniquely identify the rows in the table. A *primary key* value cannot be NULL. Also see candidate key.

Privilege

A special right or permission granted to a user or a role to perform specific actions. Granted privileges can be revoked when necessary. For example, one must grant the CREATE SESSION *privilege* to a database user before that user is allowed to login. Likewise, the CREATE TABLE *privilege* is required before a user can create new database tables.

Procedure

Block of PL/SQL code stored within the database. Also see Function and Package.

Program

A magic spell cast over a computer allowing it to turn one's input into error messages. More seriously: A *program* is a combination of computer instructions and data definitions that enable computer hardware to perform computational and control functions. A *program* is designed to systematically solve a certain kind of problem.

Propriety Standard (Defacto Standard)

A standard which has been endorsed by industry or government as the accepted international standard, but not officially approved by an accredited standards body such as ISO.

Protocol

A set of procedures for establishing and controlling data transmission. Examples include TCP/IP, NetWare IPX/SPX, and IBM's SDLC (Synchronous Data Link Control) protocols.

Proxy Copy

Feature of RMAN (Oracle 8i and above). RMAN sends the Media Management Vendor (MMV) a list of Oracle datafiles to backup, rather than sending the data itself. This allows the MMV to implement optimized backup and restore strategies. One example of this is EMC's split-mirror BCV backups.

Pseudo-column

Oracle assigned value (pseudo-field) used in the same context as an Oracle Database Column, but not stored on disk. Examples of pseudo-columns are: SYSDATE, SYSTIMESTAMP, ROWID, ROWNUM, LEVEL, CURRVAL, NEXTVAL, etc.

PSP

PL/SQL Server Pages. PSP enables developers to embed PL/SQL code into server side HTML pages. PSP is similar to JSP, PHP and ASP.

Q

QBE

Query By Example (QBE) is a method of extracting information from the database by giving it an example of what you want.

Query

A Query is a SQL SELECT statement returning data from an Oracle table of view in a database.

Formal Definition: The collection of specifications used to extract a set of data needed from a database. In traditional terms this could be called a "computer program." Queries are probably the most frequently used aspect of SQL. Queries do not change the information in the tables, but merely show it to the user. Queries are constructed from a single command. The structure of a query may appear deceptively simple, but queries can perform complex and sophisticated data evaluation and processing.

Queue

A first-in first-out data structure used to process multiple demands for a resource such as a printer, processor or communications channel. Objects are added to the tail of the queue and taken off the head.

Queue Tables can be created on an Oracle database with the Oracle Advanced Queueing Option.

Quiesce

To render quiescent, i.e. temporarily inactive or disabled. For example a database can be quiested with the ALTER SYSTEM QUIESCE RESTRICTED command. Oracle will wait for all active sessions to become inactive. Activity will resume after executing the ALTER SYSTEM UNQUIESCE command.

Quota

The amount of space allocated to a user in a tablespace

R

RAC

Real Application Clusters or *RAC* is a replacement for Oracle Parallel Server (OPS) shipped with previous database releases. *RAC* allows multiple instances on different nodes to access a shared database on a cluster system. The idea is to allow multiple servers to share the load.

RAID

RAID (Redundant Array of Independent Disks). A collection of disk drives that offers increased performance and fault tolerance. There are a number of different *RAID* levels. The three most commonly used are 0, 1, and 5:

- **Level 0:** striping without parity (spreading out blocks of each file across multiple disks).
- **Level 1:** disk mirroring or duplexing.
- **Level 2:** bit-level striping with parity
- **Level 3:** byte-level striping with dedicated parity. Same as Level 0, but also reserves one dedicated disk for error correction data. It provides good performance and some level of fault tolerance.
- **Level 4:** block-level striping with dedicated parity
- **Level 5:** block-level striping with distributed parity
- **Level 6:** block-level striping with two sets of distributed parity for extra fault tolerance
- **Level 7:** Asynchronous, cached striping with dedicated parity

RAW Device

A *RAW device* is a portion of a physical disk. The content of a *RAW device* is not managed by the operating system. Information on it cannot be identified or accessed by users (unlike with file systems). Sometimes Oracle database files are created on RAW devices to improve disk I/O performance.

RBO

See Rule Based Optimizer.

RDA

RDA (Remote Data Access) is an OSI standard that defines a service that application programs can use to access remote data. *RDA* is intended to allow different database systems from different vendors, running on different machines in different operating environments, to interoperate.

RDBMS

Relational Database Management System. A type of DBMS in which the database is organized and accessed according to the relationships between data values. The *RDBMS* was invented by a team lead by Dr. Edmund F. Codd and funded by IBM in the early 1970's. The Relational Model is based on the principles of relational algebra. Example *RDBMS* Systems: Oracle, SQL Server, DB/2, Sybase, etc.

Real-Time

The description for a system that responds to an external event, unlike a batch or time-sharing system, within a short and predictable time frame.

RECO

RECO (Oracle RECOverer Process) is an Oracle background process created when you start an instance with DISTRIBUTED_TRANSACTIONS= in the initialization parameter file. The *RECO* process will try to resolve in-doubt transactions across Oracle distributed databases.

Redo Log

A set of two or more files that record all changes made to an Oracle database. A database MUST have at least two *redo log* files. Log files can be multiplexed on multiple disks to ensure that they will not get lost.

Redo Log Buffer

A circular buffer in the SGA that contains information about changes made to the database. The LGWR process writes information from this buffer to the Redo Log Files.

Relation

Mathematical term for a table.

Relational Algebra

Mathematical system used to formally describe data access and manipulation. *Relational Algebra* consist of a collection of operators (like [SELECT], [PROJECT] and [JOIN]) that operate on a relation or relations.

Relational Database

A database system in which the database is organized and accessed according to the relationships between data items without the need for any consideration of physical orientation and relationship. Relationships between data items are expressed by means of tables.

Relationship

Association between two entities in an ERD. Each end of the *relationship* shows the degree of how the entities are related and the optionality.

Replication

Replication may be defined as a duplicate copy of similar data on the same or a different platform. The process of duplicating the data records in one database to one or more other databases in near-real time, is known as *replication*. The data can be presented in different formats.

Reports

See Oracle Reports.

Repository

A facility for storing descriptions and behaviors of objects in an enterprise, including requirements, policies, processes, data, software libraries, projects, platforms and personnel, with the potential of supporting both software development and operations management. A single point of definition for all system resources.

Restore

Get data back to a prior consistent state. See Backup.

Result Set

The set of rows the Oracle database returns when a SELECT statement is executed. The format of the rows in the *result set* is defined by the column-list of the SELECT statement. Since the application of a relational operation on a table always results in another table, a *result set* is a derived results table. This table exits only until all rows were fetched from it and the associated CURSOR is closed.

RMAN

RMAN (Recover Manager) is a utility provided by Oracle to perform database backup and recoveries. *RMAN* can do off-line and on-line database backups. It integrated with 3rd-party vendors (like Veritas, Omiback, etc) to handle tape library management.

ROLAP

Ralational Online Analytical Processing. *ROLAP* is a flexible architecture that scales to meet the widest variety of DSS and OLAP needs. *ROLAP* architectures access data directly from data warehouses using SQL.

Role

A named list or group of priviledges that are collected together and granted to users or other roles.

Rollback

Activity Oracle performs to restore data to its prior state before a user started to change it. Also see Commit and Savepoint.

Rollback Segment

Database objects containing before-images of data written to the database. Rollback segments are used to:

- Undo changes when a transaction is rolled back
- Ensure other transactions do not see uncommitted changes made to the database
- Recover the database to a consistent state in case of failures

Row

A component of a relational table. In nonrelational terms, a *row* is called a record. A *row* is not named (in contrast to a

column). Each *row* in a table has one value in each column of the table.

ROWID

Every record has a unique *ROWID* within a database representing the physical location on disk where the record lives. Note that *ROWID*'s will change when you reorganize or export/import a table.

From Oracle 8.0 the *ROWID* format and size changed from 6 to 10 bytes. The Oracle7 format is Block.Row.File. The Oracle 8 format is dataObjectNumber, block, row.

ROWNUM

ROWNUM is a pseudo-column that returns a row's position in a result set. *ROWNUM* is evaluated AFTER records are selected from the database and BEFORE any sorting takes place. The following type of queries will ALWAYS return NO DATA:

```
... WHERE ROWNUM > x
... WHERE ROWNUM BETWEEN x AND y
... WHERE ROWNUM IN (x, y, z, ...)
```

However, this will work:

```
... WHERE ROWNUM < x
```

This query will return *ROWNUM* values that are out of numerical order:

```
select ROWNUM, deptno from emp where ROWNUM < 10
order by deptno, ROWNUM;
```

RPT/ RPF

Report writing and formatting tools provided with older desupported releases of Oracle. The RPT process queried the database, producing rows of data with embedded formatting commands recognized by RPF.

RTFM

RTFM stands for "Read The F#?%$! Manual", and leave me alone!!!.

Rule Based Optimizer

Rule Based Optimizer (RBO) - SQL Query optimizer that uses heuristic rules to derive optimal query execution plans. RBO is enabled by setting OPTIMIZER_MODE=RULE in the server initialization parameter file, or by altering OPTIMIZER_GOAL for your session. Also see CBO. All applications should be converted to use CBO as RBO will not be available in Oracle 10 and above.

S

SA

System Administrator. Person that looks after the operating system.

SAP/R3

SAP/R3 is an ERP (enterprise resource planning) system from SAP AG. *SAP/R3* stands for Systems, Applications and Products, Real time, 3 tier architecture. *SAP/R3* is a direct competitor to Oracle's *Oracle Applications* product suite.

Savepoint

Set a point to which you can later roll back. Also see Commit and Rollback.

Schema

A *schema* is the set of objects (tables, views, indexes, etc) belonging to an account. It is often used as another way to refer to an Oracle account. The *CREATE SCHEMA* statement lets one specify (in a single SQL statement) all data and privilege definitions for a new *schema.* One can also add definitions to the *schema* later using DDL statements.

SCN

SCN - System Change Number - A number, internal to Oracle that is incremented over time as change vectors are generated, applied, and written to the Redo log.

SCN - System Commit Number - A number, internal to Oracle that is incremented with each database COMMIT.

System Commit Numbers and System Change Numbers share the same internal sequence generator.

Scott

Scott is a database user used for demonstration purposes containing the famous EMP and DEPT tables. The scott/tiger user is created by running the ?/rdbms/admin/utlsampl.sql script.

According to legend, *Scott* once worked for Oracle, and his cat was named TIGER.

SDLC

System Development Life Cycle (*SDLC*) - a methodology used to develop, maintain, and replace information systems. Typical phases in the *SDLC* are: Analysis, Design, Development, Integration and Testing, Implementation, etc.

Segment

Any database object that has space allocated to it is called a *SEGMENT*. A *segment* consists of one or more EXTENTS allocated within a tablespace. See catalog views: USER_SEGMENTS and DBA_SEGMENTS. Note there is no ALL_SEGMENTS view.

Select

SQL command used to query data from one or more database tables.

Self Join

A join in which a table is joined with itself.

Sequence

A database object that generates unique numbers, mostly used for primary key values. Sequences were introduced with the Transaction Processing Option in Oracle 6. One can select the NEXTVAL and CURRVAL from a *sequence*. Selecting the NEXTVAL will automatically increment the *sequence*.

Serialization

Execution order of transactions in the database.

In the Java world, *serialization* is the storing of an object's current state on any permanent storage media for later reuse. This is done using when you use the seriazlizable interface or when using the ObjectOutputStream and ObjectInputStream classes.

Session

The set of events that occurs from when a user connects to the Oracle database to when that user disconnects from the database. *Session* information is recorded in the SYS.V_$SESSION view.

SGA

The System Global Area (*SGA*) is an area of memory allocated when an Oracle Instance starts up. The *SGA*'s size and function are controlled by INIT.ORA (initialization) parameters. The *SGA* is composed of areas like the Shared Pool, Buffer Cache, Log Buffer, etc.

Shared Pool

A memory cache that is part the SGA. The *Shared Pool* is composed of the Library Cache, Dictionary Cache, and other Control Structures. The size of this area is determined by the SHARED_POOL_SIZE parameter.

Shared Server

A *Shared Server* is an Oracle background process that executes user requests. Users put requests for work on a common request queue. The Oracle Dispatcher then assigns these requests to free *shared server* processes. Also, see MTS and Dedicated Server.

SID

The Oracle System ID or *SID* is used to identify a particular database. Set the ORACLE_SID environment variable on UNIX and Windows, or ORA_SID on VMS systems.

SLA

Service Level Agreement. Formal agreement between a Service Provider and customers to provide a certain level of service. Penalty clauses might apply if the *SLA* is not met.

SMON

Oracle System MONitor. *SMON* is an Oracle background process created when you start a database instance. The *SMON* process performs instance recovery, cleans up after dirty shutdowns and coalesces adjacent free extents into larger free extents.

Snapshot

Copy of a table on a remote system. See Materialized View.

SOAP

Simple Object Access Protocol. *SOAP* is a lightweight XML based protocol used for invoking web services and exchanging structured data and type information on the Web.

Socket

The combination of an IP address and a port number.

Solaris

Operating system used on SUN Systems.

Sort Merge Join

Join optimization method where two tables are sorted and then joined. Also see Hash Join, Nested Loops Join and Cluster Join.

SPOF

Single Point of Failure. Component that, if it fails, will cause the entire system to go down.

SQL

Structured Query Language (*SQL*), pronounced "sequel", is a language that provides an interface to relational database systems. It was developed by IBM in the 1970s for use in System R. *SQL* is a *de facto* standard, as well as an ISO and ANSI standard.

SQL Server

SQL Server is a DBMS system provided by Microsoft. SQL Server is sometimes mistakenly referred to as SQL.

SQL*DBA

SQL*DBA was an administration utility used to start, stop and manage databases. SQL*DBA was replaced with Server Manager (svrmgrl) in Oracle7. From Oracle8i and above all administrative functions can be performed from SQL*Plus.

SQL*Loader

Utility used for loading data from external files into Oracle database tables.

SQL*Net

Oracle's Networking Software that allows remote data-access between user programs and databases, or among multiple databases. Applications and databases can be distributed physically to different machines and continue to communicate as if they were local. Based on the Transparent Network Substrate, a foundation network technology that provides a generic interface to all popular network protocols for connectivity throughout a network of applications. Uses a companion product (the MultiProtocol Interchange) to connect disparate networks.

SQL*Plus

SQL*Plus is an Oracle command line utility used for executing SQL and PL/SQL commands. The GUI version is called SQL Worksheet. The corresponding utility for Microsoft's SQL Server and Sybase is "isql" (interactive SQL).

SQL1

The original, 1989-vintage ANSI/ISO SQL standard.

SQL2

An extended version of the ANSI/ISO SQL standard released in 1992 that adds advanced join operations and other interesting features.

SQL3

Another extension of the SQL standard that supports object extensions.

SQLCA

SQL Communication Area. A reserved space in a client application used to receive status information from a server application with which it is communicating for the purpose of accessing data.

SQLDA

SQL Descriptor Area. Used to describe data that is passed between an RDBMS and an application that references data in a database, and vice versa.

SQLNET.ORA

SQLNET.ORA is an ASCII text file that provides SQL*Net with configuration details like tracing options, default domain, encryption, etc. This file normally resides in the ORACLE_HOME\NETWORK\ADMIN directory.

SSL

Secure Socket Layer.

Standby database

Physical copy of a database standing by to take over in the event of a failure. The standby database in a permanent state of recovery and can be opened with minimal recovery required.

Stored Procedure

A program running in the database that can take complex actions based on the inputs you send it. Using a stored procedure is faster than doing the same work on a client, because the program runs right inside the database server. Stored procedures are nomally written in PL/SQL or Java.

Striping

Storing data on multiple disk drives by splitting up the data and accessing all of the disk drives in parallel. Also see RAID.

SUN

Computer system from *SUN* Microsystems that can be used to run Oracle on. *SUN*'s operating system is Solaris.

Surrogate Key

A system generated key with no business value. Usually implemented with database generated sequences.

Sybase

A Relational Database Management System provided by *Sybase* Inc.

Synonym

An alternative name (alias) for an object in the database created with the CREATE *SYNONYM* command.

SYS

SYS is the username for the Oracle Data Dictionary or Catalog. The default password for *SYS* is CHANGE_ON_INSTALL. If you are a DBA, CHANGE IT NOW!!! Never use this user for your own scripts. You can really wreck a database from *SYS*.

SYSTEM

- SYSTEM is an Oracle username with default password of MANAGER. This user is normally used by DB administrators.
- SYSTEM is the name of the first compulsory tablespace containing the Oracle data dictionary.
- An Application *System*

System Analyst

A person responsible for studying the requirements, feasibility, cost, design, specification, and implementation of a computer based system for an organization/ business.

System R

System R is a DBMS built as a research project at IBM San Jose Research (now IBM Almaden Research Centre) in the 1970's. System R introduced the SQL language and showed that a relational system could provide good transaction processing performance. Eventually System R evolved into SQL/DS that later became DB/2. Oracle released the first commercial SQL database in early the 1980's based on the System R specs.

T

TAB

- A catalog view listing all available tables in the current schema.
- ACSII character 9 (control-I in the vi-editor). Normally used for spacing and indentation.

Table

A collection of computer data that is organized, defined and stored as rows and columns. In non-relational systems, a *table* is called a FILE.

Tablespace

A *tablespace* is a container for segments. A databases consists of one or more tablespaces, each made up of one or more data files. Tables and indexes are created within a particular *tablespace*. Make sure you do not create objects in the SYSTEM tablespace!

TAF

Transparent Application Failover (*TAF*) is a feature of Real Application Clusters (RAC). *TAF* allows users to fail-over to another node without them realizing it.

TB

1 *TB* (Terabyte) is 1024 GB. See BYTE.

TBT

Technology Based Training (*TBT*) incorporates the entire spectrum of electronic delivery through a variety of media including, Internet, LAN or WAN (intranet or extranet), satellite broadcast, audio or video tape, interactive TV, or CD-ROM. *TBT* includes both CBT (Computer Based Training) and WBT (Web Based Training).

TCL

Tool Command Language. A popular scripting language. Other popular scripting languages include: Perl, PHP, Python, etc.

TCO

Total Cost of Ownership. Cost to purchase and maintain software over time.

TCP/IP

Transmission Control Protocol/Internet Protocol. A compilation of network and transport level protocols that allow a computer to speak the same language as other computers on the Internet and/or other network.

Telnet

Telnet is a utility program and protocol that allows one to connect to another computer on a network. After providing a username and password to login to the remote computer, one can enter commands that will be executed as if entered directly from the remote computer's console.

Timestamp

An extension of the DATE datatype that can store date and time data (including fractional seconds). The *timestamp* type takes 11 bytes of storage.

Timestamp with Timezone

A variant of the TIMESTAMP datatype that includes the time zone displacement in its value.

TKPROF

Utility for analysing SQL statements executed during an Oracle database session. Trace files are produced with the ALTER SESSION SET SQL_TRACE = TRUE; command. These trace files are written to the USER_DUMP_DEST directory and are used as input to *TKPROF*.

TNS

TNS or Transparent Network Substrate is Oracle's networking architecture. *TNS* provides a uniform application interface to enable network applications to access the underlying network protocols transparently.

TNSNAMES.ORA

TNSNAMES.ORA is an ASCII text file that provides SQL*Net with server location and necessary connection strings needed to connect to Oracle databases. This file normally resides in the ORACLE_HOME\NETWORK\ADMIN directory.

TPO

Transaction Processing Option (*TPO*) was an Oracle6 database option that was later replaced with the Oracle7 Procedural Option.

TPS

Transactions Per Second (*TPS*) - a metric used to measure database performance.

Transaction

An inseparable list of database operations which must be executed either in its entirety or not at all. Transactions maintain data integrity and guarantee that the database will always be in a consistent state. Transactions should either end with a COMMIT or ROLLBACK statement. If it ends with a COMMIT statement, all the changes made to the database are made permanent. If the *transaction* fails, or ends with a ROLLBACK, none of the statements takes effect. Also see LUW.

Transportable Tablespaces

A feature of Oracle8i releases and above. This option allows one to detach a tablespace from a database and attch it to another database.

Trigger

A program in a database that gets called each time a row in a table is INSERTED, UPDATED, or DELETED. Triggers allow you to check that any changes are correct, or to fill in missing information before it is commited. Triggers are normally written in PL/SQL or Java.

TRUNCATE

DDL command that removes all data from a table. One cannot ROLLBACK after executing a *TRUNCATE* statment. Also see Delete.

Tuple

A row in a table is called a *tuple* of the relation. The number of tuples in a relation is known as the cardinality of the relation. Tuples in a table are unique and can be arranged in any order.

Two-Phase Commit

A strategy in which changes to a database are temporarily applied. Once it has been determined that all parts of a change can be made successfully, the changes are permanently posted to the database.

TWO_TASK

Environment variable used to specify that connections should be made to a remote database without specifying a service name. This is equivalent to LOCAL registry entry on Windows platforms

U

UAT

User Acceptance Testing. Also known as Beta Testing, QA Testing, Application Testing or End User Testing.

UGA

User Global Area - area that contains data required to support a (user) session. The *UGA* is located in the PGA when running in dedicated server mode. With MTS, the *UGA* is located in the LARGE_POOL (if specified), otherwise in the SHARED_POOL.

UID

A pseudo-column returning a numeric value identifying the current user.

```
SQL> SELECT USER, UID FROM DUAL;
USER                            UID
------------------------------------------------------------------
SCOTT                           207
```

UML

Unified Modelling Language (*UML*) is an Object Management Group (OMG) standard for modelling

software artifacts. Using *UML*, developers and architects can make a blueprint of a project, much like ERD diagrams are used for relational design. See http://www.rational.com/uml/ for more details. *UML* diagrams can be constructed with Oracle JDeveloper.

Undo Information

The information the database needs to undo or rollback a user transaction due to a number of reasons.

Unique Constraint

A constraint that enforces all non-NULL values in a column to be different from each other.

Unique Key

Is used to uniquely identify each record in an Oracle table. There can be one and only one row with each *unique key* value.

Unix

An operating system co-created by AT&T researchers Dennis Ritchie and Ken Thompson. *Unix* is well known for its relative hardware independence and portable application interfaces. Lots of big companies are using *Unix* servers for its reliability and scalability. Some of the popular *Unix* flavours are: Linux, Solaris, HP-UX, AIX, etc.

Update

DML command used to change data in a table. Also see Insert and Delete.

Upgrade

Finding existing bugs and replace them with new bugs.

Upsert

A series of conditional update and insert operatons. Records that exist within a table will be updated. New records will be inserted into the table. *Upsert* functionality is implemented in Oracle with the MERGE command.

URL

Universal Resource Locator. An Internet World Wide Web Address.

User

- A pseudo-column returning a srting value with the name of the current *user*.
- Those people that hassle us poor techies.

USER_% views

A group of catalog (or data dictionary) views that detail database objects owned by the current user/schema.

V

V$ Views

V$ views are dynamic performance views based on the X$ tables. V$ views are owned by SYS. They can be accessed by anyone with the "SELECT ANY TABLE" system privilege.

VAN

Value-Added Network. A system where a network leases communication lines from a communications common carrier, enhances them by adding improvements such as error detection and/or faster response time, and then allows others to use this service on those lines for a fee.

varchar2

Data type used to store variable-length character data. A *varchar2* value can contain up to 4000 bytes of data. Also see CHAR.

Variable

Programmer-defined name to hold information in an program or PL/SQL block.

View

A *view* is the result of a SQL query stored in the Oracle Data Dictionary. One can think of it as a virtual table or presentation of data from one or more tables. Views are useful for security and information hiding, but can cause problems if nested too deep. *View* details can be queried from the dictionary by querying either USER_VIEWS, ALL_VIEWS or DBA_VIEWS.

Virtual Memory

The memory that the operating system allocates to programs. *Virtual memory* is mapped to RAM (physical memory). When there is not enough RAM to run all programs, some memory pages cane be temporarily paged or swapped from RAM to disk.

VRML

Virtual Reality Modeling Language. A programming language used for modeling and retrieval of virtual reality environments.

W

WAN (Wide-Area Network)

A data transmission facility that connects geographically dispersed sites using long-haul networking facilities.

Warehouse

See Data Warehouse.

WBT

Web Based Training (*WBT*) is delivered via a Web browser, such as Netscape Navigator or Internet Explorer, to access the Internet or a company's intranet for courses that reside on servers. Web-based training can be conducted either synchronously or asynchronously. Also see CBT and TBT.

Web

See WWW.

Web Browser

A program that end users utilize to read HTML documents and programs stored on a computer (serviced by a Web server). Popular web browsers are: Netscape Navigator and Internet Explorer.

Web Cartridge

A program executed on a Web server via the Oracle WRB (Web Request Broker).

Web Server

A server process (HTTP daemon) running at a Web site which sends out Web pages in response to HTTP requests from remote Web browsers.

Windows

Family of GUI operating systems produced by Microsoft Corporation. Some examples: *Windows* 2003, *Windows* 2000, *Windows* NT, *Windows* XP, *Windows* 95, etc.

Some jokingly describes it as:

32 bit extensions and a graphical shell for a
16 bit patch to an
8 bit operating system originally coded for a
4 bit microprocessor, written by a
2 bit company that can't stand
1 bit of competition.

Wizard

Graphical representations of program actions (commands or shortcut keys) used to make the program easier to use.

WORM

Write Once, Read Many times (i.e., Read Only).

Wrapper

An object, function or procedure that encapsulates and delegates to (call) another object to alter its interface or behavior in some way.

WRB

The Oracle *WRB* (Web Request Broker) is part of the Oracle Internet/Web Application Server. It provides a distributed environment for developing and deploying applications for the Web. The *WRB* enables developers to write applications that are independent of, and work with a number of, Web servers.

WWW

World Wide Web. A network of servers that uses hypertext links to find and access documents.

WYSIWYG

WYSIWYG (What You See Is What You Get - pronounced "whizzy-wig") means that information will print out exactly as it appears on screen.

X
X Windows

X Windows is a public domain windowing system that is mainly used on UNIX systems. The system includes a standard library of routines that can be used to develop GUI applications. The system also includes standard utilities like xclock, xcalc, xeyes, etc.

X$ Tables

X$ tables are internal in-memory tables which hold information about the database instance. X$ tables can only be viewed by SYS.

X.25

A data communication protocol that ensures data integrity while data is being transmitted to, from and within the network. This standard defines the interconnection of packet-switching networks and their associated computers or terminals. These types of networks make efficient use of the telecommunications networks by taking the data generated by a computer or a remote terminal and chopping it up into small identified packets and then looking for the most efficient way of sending this information to its destination.

X.400

A CCITT recommendation specifying an OSI standard for electronic mail transfer.

XA

XA is a two-phase commit protocol defined by the X/Open DTP group. *XA* is natively supported by many databases (like Oracle) and transaction monitors (like Tuxedo).

XML

XML (Extensible Markup Language) is a W3C initiative that allows information and services to be encoded with meaningful structure and semantics that computers and humans can understand. *XML* is great for information exchange, and can easily be extended to include user-specified and industry-specified tags.

XPG

X/Open Portability Guide. A comprehensive set of APIs, protocols and other specifications designed to promote open, interoperable computing. Operating systems can be branded with a certain level of compliance, for example XPG3 or XPG4.

Oracle is ported to various operating systems, and will most likely work better on systems that are *XPG* compliant.

XSL

Extensible Stylesheet Language (*XSL*) is a language for transforming XML documents into other document formats like HTML.

Y
Y2K

Year 2000 or *Y2K* referred to the millennium problem. A total non-event.

This problem resulted from the common practice of using **two digits to represent dates**. When computers were first developed, memory and physical storage space for the systems were very expensive. Using only two numbers for dates saved money and saved data-entry time. Over the years, programmers felt that their programs would probably not be around in 1999, so the problem wouldn't arise.

Z

Zero Suppression

Replaces leading zeroes with spaces in numeric data for display purposes.

Zip

Compressed version of a program or document that can be uncompressed.

Zone

Zones in Oracle Applications are groups of related areas on a screen. A screen may have many zones.

Zoom

A feature that lets you magnify text or graphics images onscreen.

#

%ROWTYPE

%ROWTYPE can be used in PL/SQL to declare a record with the same types as found in the specified table, view or cursor. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet new business needs. Example:

```
DECLARE
  v_EmpRecord  emp%ROWTYPE;
```

%TYPE

%TYPE can be used in PL/SQL to declare a field with the same type as that of a specified table's column. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet new business needs. Example:

```
DECLARE
  v_EmpNo  emp.empno%TYPE;
```

100BaseT

Same as 10BaseT, but running at 100Mbps.

10BaseT

IEEE 802.3 Ethernet LAN specification, using unshielded twisted pair wiring running at 10Mbps.

1:1

One-to-One relationship in an ERD.

1:N

One-to-many relationship in an ERD.

1GL

First-generation computer language. Binary machine code instructions.

1NF

First Normal Form in the data normalization process. During this step repeating groups are eliminated by putting each into a separate table and connecting them with a one-to-many relationship.

24x7

24x7 normally indicates database or system availability of 24 hours a day, 7 days a week without ant downtime.

2GL

Second-generation computer language. Assembler language which use cryptic mnemonic commands.

2NF

Second Normal Form in the data normalization process. During this step functional dependencies on a partial key are eliminated by putting the fields in a separate table from those that are dependent on the whole key.

2PC

Oracle's Two-Phase Commit protocol. *2PC* is used to ensure that all databases involved in a transaction either commits or roll-back together, leaving a distributed database in a consistent state.

3270

Family of IBM information display stations, printers, and control units. *3270* is the de facto standard for mainframe terminals.

3GL

Third-generation computer language. PL/SQL, COBOL, Fortran, Java, Ada, Smalltalk, C and C++ are example 3GLs.

3NF

Third Normal Form in the data normalization process. During this step functional dependencies on non-key fields are eliminated by putting them in a separate table. At this stage, all non-key fields are dependent on the key, the whole key and nothing but the key.

4GL

Fourth-generation computer language. SQL, Oracle Forms, Oracle Power Objects and Visual Basic are example 4GLs.

4NF

Fourth Normal Form in the data normalization process. During this step independent multi-valued facts stored in one table are separated into different tables.

5GL

Fifth-generation computer language. A language that incorporates the concepts of EXPERT SYSTEMs, inference engines, and natural language processing.

5NF

Fifth Normal Form in the data normalization process. During this step data redundancy that is not covered by any of the previous normal forms are handled.

Rai Technology University

ENGINEERING MINDS

Rai Technology University Campus

Dhodballapur Nelmangala Road, SH -74, Off Highway 207, Dhodballapur Taluk, Bangalore - 561204

E-mail: info@raitechuniversity.in | Web: www.raitechuniversity.in