DBMS

DBMS – Intoduction

Database -- Collection of interrelated data

DBMS -- Collection of **interrelated data** and **Set of programs** to access those data
-- contains information about a particular organization
-- designed to manage large parts of information.

Management of data involves
both defining **structures** for **storage** of info. and providing mechanisms for the info. **manipulation.**

Goal -- to provide a way to **store** and **retrieve** db information is both **convenient** and **efficient**

Database Applications:

Banking: all transactions

Airlines: reservations, schedules

Universities: registration, grades

Sales: customers, products, purchases

Online retailers: order tracking, customized recommendations

Manufacturing: production, inventory, orders, supply chain

Human resources: employees, salaries, tax deductions

Telecommunication: For keeping records of calls made, generating monthly bills

Purpose of Database Systems

Consider part of a S/B enterprise

-- keeps info about all customers and S/B accounts.

One way to keep the information on a computer is to store it in OS files.

to manipulate -- system has application programs

A program to debit or credit an account

A program to add a new account

A program to find the balance of an account

A program to generate monthly statements

Note: New application programs -- added depends upon the need

Drawbacks of using file systems to store data:

Data Redundancy and Inconsistency

Multiple file formats,
Duplication of information -- in different files

Leads to **Higher Storage** and **Access Cost**.
may lead to **Data Inconsistency**

Difficulty in Accessing Data

Need to write a new program to perform each new task

Data Isolation -- multiple files and formats writing new application programs to retrieve proper data is difficult.

Integrity Problems

Consistency constraints

e.g. account balance >=500

become **hidden** in program code rather than being stated explicitly

Difficult to add new constraints or change existing ones

Atomicity of Updates

Failures may leave db in an inconsistent state with partial updates

e.g., Transfer of funds from one account to another should either **complete** or **not happen at all**

Concurrent-access Anomalies

Concurrent access -- needed for performance

Uncontrolled concurrent accesses can lead to inconsistencies

e.g., Multiple users reading a balance and updating it at the same time must maintain some form of **supervision**

Security problems

Hard to provide user access to some, but not all, data

Note: Db systems offer solutions to all of the above problems

Levels of Abstraction

Purpose -- is to provide users with an abstract view of the data.

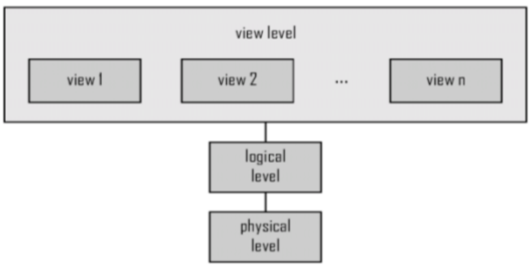
i.e. hides certain information of how the data are stored and maintained

Usually, db users are not computer trained, developers hide the complexity -- several levels of abstraction

Physical Level / Low Level

Logical l Level / Conceptual Level

View Level / High Level



Physical level -- describes **how** a record (e.g. student) is stored.

Logical level -- describes **what** data are stored in database, and the **relationships** among those data.

type *student* = **record**
student _id : string;
student _name : string;
student _course : string;
End;

View level

hide details of data types.

also hide information for security purposes.

e.g. tellers -- information of customer accounts -- an employee's salary

Instances and Schemas

Schema -- the logical structure of the database

Physical schema: db design at the physical level
Logical schema: db design at the logical level

Instance

the actual content of the db at a particular point in time

Analogous to the value of a variable

Physical Data Independence

the ability to modify the physical schema without changing the logical schema
Applications depend on the logical schema

Database Languages

Data Manipulation Language (DML) -- also known as query language

-- Language for accessing and manipulating the data organized by the appropriate data model

Retrieval of information stored in the db

Insertion of new information into the db

Deletion of information from the db

Updation of information stored in the db

Two classes of languages

Procedural

user specifies **what** data is required and **how** to get those data

Declarative (nonprocedural)

user specifies **what** data is required without specifying how to get those data

SQL -- most widely used query language

Data Definition Language (DDL) -- specify the db schema

Specification notation for defining the db schema

Example: **create table** *account* (
account_number **varchar**(10),
branch_name **varchar**(10),
balance **number**)

DDL compiler generates a set of tables stored in a **data dictionary**

Data dictionary contains metadata (i.e., data about data)

Db **schema**

Data **storage and definition** language

- Specifies the storage structure and access methods used

Integrity constraints

- Domain constraints
- Referential integrity (e.g. *branch_name* must correspond to a valid branch in the *branch* table)

Authorization

In practice, the DD and DM languages – not TWO separate languages; instead they simply form parts of a single database language

e.g. widely used SQL language.

Database Users and Administrators

who work with a db can be categorized as db users and db administrators.

Four different types of db-system users

Naive users

unsophisticated users
who interact with the system by invoking one of the application programs

Application programmers

computer professionals who write application programs.

Sophisticated users

interact with the system without writing programs.
they form their requests in a database query language.

Specialized users

sophisticated users who write specialized db applications that do not fit into the traditional data-processing framework.

Database Administrator

A person who has central control over the system

control of both the data and the programs that access those data

Schema definition

The DBA creates the original database schema by executing a set of data definition statements in the DDL.

Storage structure and access-method definition

Schema and physical-organization modification

carries out changes to the schema and physical organization to reflect

a. the changing needs of the organization, or
b. to improve performance by changing physical organization

Granting of authorization for data access

can regulate which parts of the db various users can access.

Routine maintenance

Periodically backing up the db
to prevent loss of data

Ensuring that enough free disk space
available for normal operations, and upgrading disk space as required.

Monitoring jobs running on the db
not to degrade the performance

Database System Structure

A db system -- partitioned into modules -- deal with the system responsibilities

Functional components of a db system -- broadly divided into **Two**

Query processor

Storage manager

Storage manager

-- a **program module --** i/f between the **low-level data stored** in the **db** and the **application programs** and **queries** submitted to the system

-- responsible for the interaction with the file manager.
-- translates DML into low-level file-system commands

Therefore, the storage manager is responsible for **storing, retrieving,** and **updating**

The **components of Storage Manager**

Authorization and Integrity Manager
Transaction Manager
File Manager
Buffer Manager

Authorization and Integrity Manager

tests for the satisfaction of integrity constraints and checks the authorization of the users to access data

Transaction Manager

ensures that the db remains in a consistent state despite system failures

also ensures that concurrent transaction executions without conflicting.

File Manager

manages the allocation of space on disk storage and

the data structures used to represent information stored on disk

Buffer Manager

responsible for fetching data from disk storage into MM, and

deciding what data to cache in main memory.

Storage Manager **implements** several **data structures** as part of the physical system implementation

Data files -- store the **db itself**.

Data dictionary

stores metadata about the **structure of the db**, in particular the schema of the db.

Indices -- provide **fast access** to data items that hold particular values

Statistical data -- information about **data** in the **db**

Query processor

helps the db system **simplify** and **facilitate access** to **data**.

Components of **Query Processor**

DDL interpreter

interprets DDL statements and records the definitions in the data dictionary.

DML compiler

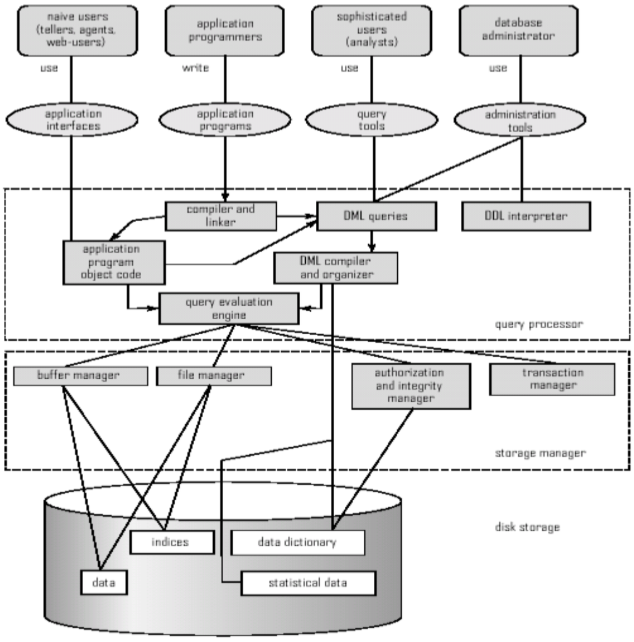
translates DML statements in a query language into an evaluation plan.

also performs query optimization

i.e., it picks the lowest cost evaluation plan from among the alternatives.

Query evaluation engine

executes low-level instructions generated by the DML compiler.



Application Architectures

Today, most users of a db stems are not present at the site of the db, but connects through a NW. The differentiate between

client machines,

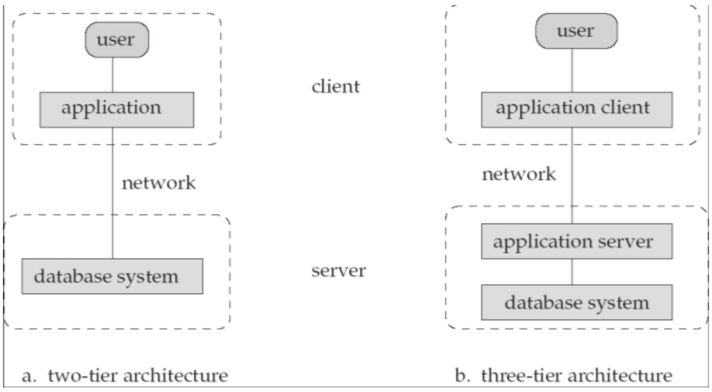
on which remote db users work, and

server machines,

on which the db system runs.

Db applications -- usually partitioned into 2 or 3 parts, (see dig.)

2-tier architecture 3-tier architecture



a. two-tier architecture b. three-tier architecture

2-tier Architecture

the application is partitioned into a component that resides at the client machine

invokes db system functionality at the server machine through query language statements

3-tier Architecture

the client machine acts as merely a front end and does not contain any direct db calls.

client end communicates with an **application server** -- usually through interface.
-- in turn communicates with a db to access data.

are more appropriate for large applications for applications that run on the WWW.

Extended E-R Features: Specialization

-- process of designating subgroupings within an entity set .

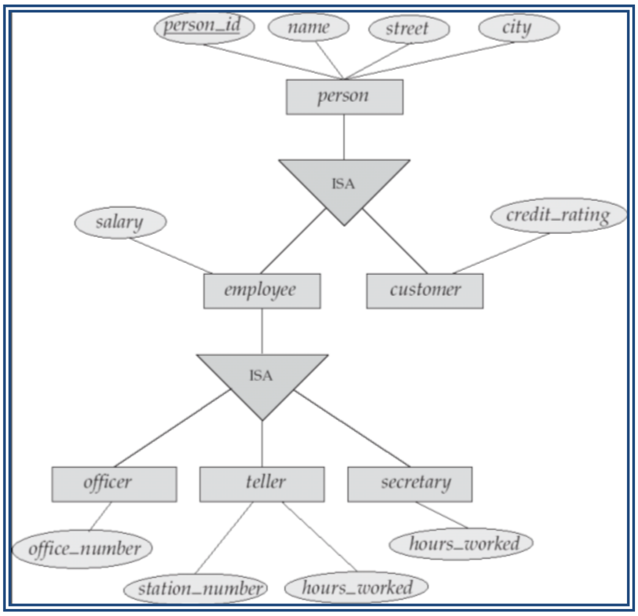
-- are distinctive from other entities in the set.

These subgroupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.

-- Top-down design process

-- by a triangle component labeled ISA

e.g. customer is a person



Extended ER Features: Generalization

-- can be expressed - commonality

e.g. several attributes – common between the customer and the employee entity set

A bottom-up design process -- combine a no. of entity sets that share the same features into a higher-level entity set

-- simple inversions of specialization

Both -- represented in an E-R diagram in the same way.

-- used interchangeably.

Specialization stems from a single entity set

emphasizes differences among entities within the set by creating distinct lower-level entity sets.

Can have multiple specializations of an entity set based on different features.

Generalization proceeds from the recognition that a number of entity sets share some common features

-- used to emphasize the similarities among lower-level entity sets and to hide the differences

-- shared attributes are not repeated.

The ISA relationship also referred to as **super-class – subclass** relationship

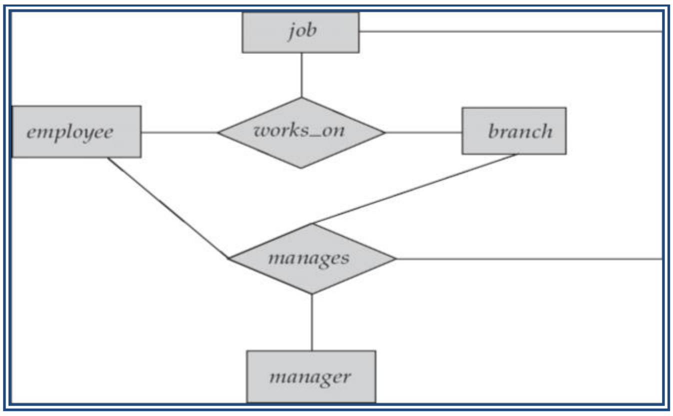
Aggregation

One limitation -- cannot express relationships among relationships

Consider the ternary relationship *works_on*

suppose, we want to record managers for tasks performed by an employee at a branch

Let us assume -- an entity set **manager**



Relationship sets **works_on** and **manages** represent overlapping information

Every **manages** relationship corresponds to a **works_on** relationship

However, some works_on relationships may not correspond to any manages relationships

can't discard the works_on relationship

Eliminate this redundancy via **aggregation**

Treat relationship as an **abstract entity**

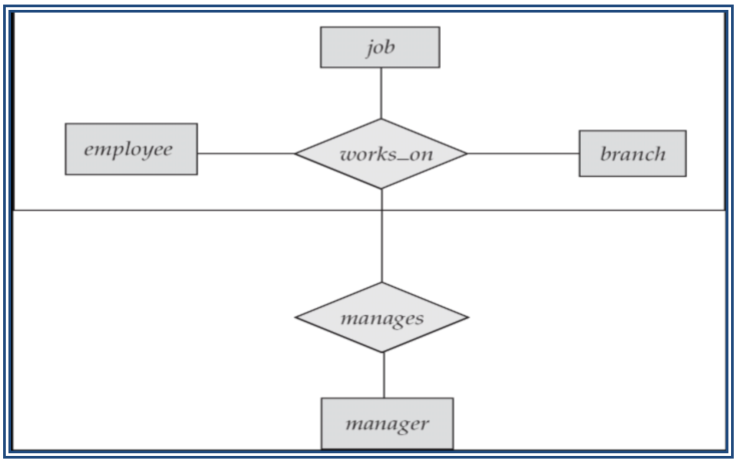
Allows relationships between relationships

Abstraction of relationship into new entity

Without introducing redundancy, the following diagram represents:

An employee works on a particular job at a particular branch

An employee, branch, job combination may have an associated manager



DBMS – RA

The Relational Algebra **--** procedural query language.

The fundamental operations in the relational algebra --

select,
project,
union,
set difference,
Cartesian product, and
rename.

several other operations -- set intersection,
natural join,
division, and
assignment.

Fundamental Operations

SELECT,
PROJECT, and
RENAME operations

-- called **unary operations** -- operate on one relation

Rest of **Three** operations operate on **pairs of relations** --called **binary operations**.

The Select Operation

selects tuples that satisfy a given predicate.

lowercase **Greek** letter **Sigma** (**σ**)

The **Predicate** appears as a **Subscript** to σ.

The **relation** -- in **parentheses** after the **σ**.

e.g. select those tuples of the loan relation where the branch is LPool

$\sigma_{\text{branch-name} = \text{"LPool"}}$ (loan)

e.g. find all tuples in which the amount lent is more than 200000

$\sigma_{\text{amount}>200000}$ (loan)

we allow comparisons using =, =, <, ≤, >, ≥ in the selection predicate.
Furthermore, connectives and (∧), or (∨), and not (¬)

e.g. find those tuples pertaining to loans of more than 200000 made at the LPool branch

$\sigma_{\text{branch-name} = \text{"LPool"} \wedge \text{amount}>200000}$ (loan)

The Project Operation

to list all loan numbers and the amount of the loans

-- any duplicate rows are eliminated -- is a unary operation

-- is denoted by the uppercase **Greek** letter **pi** (**Π**).
list those attributes -- wish to appear in the result as a subscript to Π.

e.g. to list all loan numbers and the amount of the loan

$\Pi_{\text{loan-number, amount}}$ (loan)

Composition of Relational Operations

e.g. Find the customer names who live in Mehidipatnam

$\Pi_{\text{customer-name}}$ ($\sigma_{\text{customer-city} = \text{"Mehidipatnam"}}$ (customer))

Union Operation -- the **binary** operation
-- denoted by ∪

e.g. find the names of all customers of a bank who have either an account or a loan or both.

Note: customer relation does not contain the information

-- need the information in the depositor relation

We know how to find the names of all customers with a loan in the bank

$\Pi_{\text{customer-name}}$ (borrower)

also know -- find the names of all customers with bank account in the bank

$\Pi_{\text{customer-name}}$ (depositor)

To answer the query, we need the **union**

$\Pi_{\text{customer-name}}$ (borrower) ∪ $\Pi_{\text{customer-name}}$ (depositor)

Note: we must ensure that **Unions** are taken between **compatible** relations

e.g. it would not make sense -- the union of the loan and borrower

The **Loan** is a relation with **Three** attributes;

The **borrowe**r is a relation with **Two** attributes.

Also, consider a union of a set of customer names & a set of cities--union would not make sense in most situations

for a union operation **r ∪ s** to be **valid** -- 2 conditions hold

The relations r and s must be of the **same arity**.

i.e. must have the **same no.** of **attributes**.

The domains of the **ith attribute of r** and the **ith attribute of s** must be the same, **for all i**.

The Set Difference Operation -- denoted by **__**

allows us to find tuples that are in one relation but are not in another.

The expression **r − s** produces a relation containing those tuples in **r but not in s**.

e.g. find all customers of the bank who have an account but not a loan

$\Pi_{\text{customer-name}}$ (depositor) − $\Pi_{\text{customer-name}}$ (borrower)

Like union -- must ensure that set differences are taken between compatible relations

The Cartesian-Product Operation

-- allows us to combine info. from any two relations.

-- denoted by a cross (×),

-- of relations r1 and r2 as $\mathbf{r_1 \times r_2}$

Same attribute name -- may appear in both r1 and r2

e.g. the relation schema for r = borrower × loan is

(customer-name, borrower.loan-number, loan.loan-number, branch-name, amount)

we know the relation schema for r = borrower × loan

what tuples appear in r ?

a tuple of r out of each possible pair of tuples: one from the borrower and one from the loan

r is a large relation see the below fig. -- includes only a portion of the tuples that make up r.

Assume -- x tuples in borrower and y tuples in loan.

x ∗ y ways of choosing pairs of tuples – 1 tuple from each x ∗ y tuples in r.

Note: for some tuples t in r – there may be a tuple that

t[borrower.loan-number] <> t[loan.loan-number].

custr-name	borrower. loan-no	loan. loan-no	br-name	amount
Abhi	L-16	L-11	S.P.Road	300000
Abhi	L-16	L-16	Kondapur	250000
Abhi	L-16	L-23	Kondapur	150000
Abhi	L-16	L-14	R.P.Road	150000
Abhi	L-16	L-93	L Pool	150000
Laxmi	L-93	L-11	S.P.Road	300000
Laxmi	L-93	L-16	Kondapur	250000
Laxmi	L-93	L-23	Kondapur	150000
Laxmi	L-93	L-14	R.P.Road	150000
Laxmi	L-93	L-93	L Pool	150000
. . .	. . .	. . .	. . .	. . .
. . .	. . .	. . .	. . .	. . .
Ramu	L-23	L-11	S.P.Road	300000
Ramu	L-23	L-16	Kondapur	250000
Ramu	L-23	L-23	Kondapur	150000
Ramu	L-23	L-14	R.P.Road	150000
Ramu	L-23	L-93	L Pool	150000

usually, if we have relations $r_1(R_1)$ and $r_2(R_2)$, then

$r_1 \times r_2$ -- is the concatenation of R_1 and R_2.

Relation R contains all tuples t for which -- is a tuple t_1 in r_1 and a tuple t_2 in r_2 --

$t[R_1] = t1[R_1]$ and $t[R_2] = t2[R_2]$.

Suppose to find the names of all customers who have a loan at the Kondapur branch.

-- need the information in both the loan relation and the borrower relation

$\sigma_{\text{branch-name} = \text{“Kondapur”}}(\text{borrower} \times \text{loan})$

cust-name	borrower. loan-no	loan. loan-no	br-name	amount
Abhi	L-16	L-16	Kondapur	250000
Abhi	L-16	L-23	Kondapur	150000
Laxmi	L-93	L-16	Kondapur	250000
Laxmi	L-93	L-23	Kondapur	150000
Ramu	L-23	L-16	Kondapur	250000
Ramu	L-23	L-23	Kondapur	150000

Observe the above result -- the customer-name -- may contain customers who do not have a loan at the Kondapur branch

Reason: Cartesian product takes all possible pairings of one tuple from borrower with one tuple of loan

if a customer has a loan in the Kondapur branch -- then -- is some tuple in borrower×loan -- contains his name

borrower.loan-number = loan.loan-number.

$\sigma_{\text{borrower .loan-number} = \text{loan.loan-number}}(\sigma_{\text{branch-name} = \text{“Kondapur”}}(\text{borrower} \times \text{loan}))$

-- get only those tuples of customers who have a loan at the Kondapur branch.

Finally, -- want only customer-name

$\Pi_{\text{customer-name}}(\sigma_{\text{borrower .loan-number} = \text{loan.loan-number}}(\sigma_{\text{branch-name} = \text{“Kondapur”}}(\text{borrower} \times \text{loan})))$

The Rename Operation

-- is useful to be able to give names

-- denoted by the lowercase **Greek** letter **rho (ρ)**

Given a relational-algebra expression E, the expression

ρ_x (E) -- returns the result of expression E under the **name x**.

also apply -- to a relation r -- get the same relation with a new name.

$\rho_{x(A1,A2,...,An)}$ (E)

returns the result of expression E with the name x, and with the attributes renamed to $A_1, A_2, \ldots, A_n$

e.g. Find the largest account balance in the bank.

Step 1 compute a temporary relation -- balances that are not the largest

Step 2 take the set difference between the relation

$\Pi_{balance}$ (acc) and the temporary relation

Step 1 Compute temporary relation that consists of the balances that are not the largest

$\Pi_{acc.balance} (\sigma_{acc.balance < d.balance} (acc \times \rho_d (acc)))$

Step 2: to find the largest account balance in the bank

$\Pi_{balance} (acc) - \Pi_{acc.balance} (\sigma_{acc.balance < d.balance} (acc \times \rho_d (acc)))$

Additional Operations

Using fundamental operations certain queries are lengthy to express -- need additional operations

The Set-Intersection Operation -- denoted by ∩

to find all customers who have both a loan and an acct

$\Pi_{customer\text{-}name} (borrower) \cap \Pi_{customer\text{-}name} (depositor)$

Note: can rewrite the above expression with a pair of set- difference operations

$r \cap s = r - (r - s)$

-- is simply more convenient to write $r \cap s$ than $r - (r - s)$

-- is not a fundamental operation & does not add any power to the RA

to simplify certain queries -- require a Cartesian product

Usually, a query -- involves a Cartesian product includes a selection operation on the Cartesian product result .

e.g. Find customers names who have a loan at the bank, along with the loan number and the loan amount.

$\Pi_{customer\text{-}name,\ loan.loan\text{-}number,\ amount} (\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number} (borrower \times loan))$

The Natural-Join Operation

denoted by the “join” symbol

Consider two relations r(R) and s(S).

The natural join of r and s, denoted by r s, is a relation on schema R ∪ S formally defined as follows:

R ∪ S formally defined as follows:

$r \quad s = \Pi_{R \cup S} (\sigma_{r.A1 = s.A1 \wedge r.A2 = s.A2 \wedge \ldots \wedge r.An = s.An}\ r \times s)$

$\Pi_{customer\text{-}name,\ loan\text{-}number,\ amount} (borrower \quad loan)$

find all customers who have both a loan and an account at the bank.

$\Pi_{customer\text{-}name} (borrower \quad depositor)$

Note: we wrote the same query using set
$\Pi_{customer\text{-}name}$ (borrower) ∩ $\Pi_{customer\text{-}name}$ (depositor)

Theta join -- is an extension to the natural-join operation
-- allows us to combine a selection and a Cartesian product into a single operation

Consider relations r(R) and s(S), and let **θ** be a predicate on attributes in the schema R ∪ S.

The theta join operation $r \quad {}_{\theta}\, s = \sigma_{\theta}(r \times s)$

The Division Operation -- ÷
is appropriate to queries -- **for all**.

e.g. to find **all customers** who have an account at **all the branches** located in Secunderabad.

$r_1 = \Pi_{branch\text{-}name}\ (\sigma_{branch\text{-}city = \text{“Secunderabad”}}\ (branch))$

find all (customer-name, branch-name) pairs for which the customer has an account at a branch by writing

$r_2 = \Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \qquad account)$

Now, we need to find customers who appear in r_2 with every branch name in r_1.

formulate the query by writing
$\Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \qquad account)$
÷
$\Pi_{branch\text{-}name}\ (\sigma_{branch\text{-}city = \text{“Secunderabad”}}\ branch))$

formally, let r(R) and s(S) be relations, and let S ⊆ R
i.e. every attribute of schema S is also in schema R.
then r ÷ s is a relation on schema R − S
i.e., on the schema containing all attributes of schema R that are not in schema S.

A tuple t is in r ÷ s, iff both of 2 conditions hold
1. t is in $\Pi_{R-S}(r)$
2. For every tuple t_s in s, there is a tuple t_r in r satisfying both the following
   a. $t_r[S] = t_s[S]$
   b. $t_r[R - S] = t$

-- in terms of the fundamental operations.
Let r(R) and s(S) be given, with S ⊆ R
$r \div s = \Pi_{R-S}\ (r) - \Pi_{R-S}\ ((\Pi_{R-S}\ (r) \times s) - \Pi_{R-S,S}(r))$

The Assignment Operation -- denoted by ←
like assignment in a programming language.
e.g. write r ÷ s as
$temp1 \leftarrow \Pi_{R-S}\ (r)$
$temp2 \leftarrow \Pi_{R-S}\ ((temp1 \times s) - \Pi_{R-S,S}(r))$
$result = temp1 - temp2$

Extended Relational-Algebra Operations

The basic relational-algebra operations have been extended in several ways.
An important extension is to allow aggregate operations

Generalized Projection -- extends the projection operation by allowing arithmetic functions.
-- has the form
$\Pi_{F1,F2,\ \ldots,\ Fn}(E)$

E -- any relational-algebra expression

each of $F_1, F_2, \ldots, F_n$ -- an arithmetic expression involving constants and attributes in the schema of E.

Note: the arithmetic expression may be -- an attribute or a constant.

e.g. suppose we have a relation

credit-data (customer-name, limit, credit-balance)

the credit limit and expenses so far (the credit-balance on the account).

If we want to find how much more each person can spend can write

$\Pi_{\text{customer-name, limit} - \text{credit-balance}}$ (credit-data)

The attribute resulting from the expression

limit − credit-balance does not have a name.

We can apply the rename operation to the result

$\Pi_{\text{customer-name, (limit} - \text{credit-balance) as credit-available}}$ (credit-data)

Aggregate Functions

-- take a collection of values and return a single value as a result.

The symbol G is the letter G in calligraphic font

read it as “calligraphic G.”

Undesirable Properties

Repetition of information

Inability to represent certain information

suppose the information concerning loans is kept in one single relation, *lending,*

Lend-schema = (br-name, br-city, assets, cust-name, loan-no, amount)

br-name	br-city	assets	cust-name	Loan-no	amount
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
NorthTown	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

A tuple t in the lending relation has the following intuitive meaning:

t[assets] is the asset figure for the branch named t[br-name]

t[br-city] is the city in which the branch named t[br-name] is located

t[loan-no] is the number assigned to a loan by the branch named t[br-name] to the customer named t[cust-name]

t[amount] is the amount of the loan whose number is t[loan-no].

e.g. wish to **add** a **new loan** to our db.

Say at Perryridge to Ramu in the amount of 150000

Let the loan-number be L-31.

In our design, we need **add** a **tuple** with **all** the **values** Lend-schema. -- must **repeat** the **asset** and **city** data

(Perryridge, Horseneck, 1700000, Ramu, L-31, 150000)

Repetition of info -- present design -- **undesirable**

Repeating information **wastes space**.

Also, it **complicates updating** the db.

e.g. the assets of the Perryridge branch change from 1700000 to 1900000

Under our **original design** -- **one tuple** of the branch relation needs to be **changed**.

Present design-- ensure that **every tuple** pertaining to the Perryridge branch is **updated**

Thus **updates** are **more costly**

Why the present design is bad?

know -- branch of a **bank** has a **unique value** of **assets**

branch name -- can **uniquely identify** the assets other hand -- a branch may make **many loans**

branch name -- cannot uniquely find a **loan number**
we say -- the FD ***br-name → assets*** holds on Lending
do **not expect** the FD ***br-name → loan-no*** to hold.

Another problem with the *Lending-schema* design
cannot represent **directly** the information concerning a branch (**br-name, br-city, assets**)

unless there exists **at least one loan** at the branch.

because loan-no, amount, and cust-name -- require values
One solution -- is to introduce **null values**
null values are **difficult to handle**

create the branch info only when the 1st loan application at that branch is made
Worse -- delete this info when all the loans have been paid.
Clearly, this situation is **undesirable**

Note: original db design -- the branch information would be available regardless of whether or not loans are currently maintained in the branch

Functional Dependencies

play a key role in differentiating **good** db designs from **bad db designs**.

type of **constraint** that is a **generalization** of the **notion** of *key*

are constraints on the **set** of **legal relations**.

allow us to **express facts** about the enterprise to be modelled a db

Already -- defined the notion of a **superkey** as

Let **R** be a relation schema. A **subset K** of **R** is a **superkey** of R if, in any legal relation r(R), for all pairs t_1 and t_2 of tuples in r such that $t_1 <> t_2$, then $t_1[K] <> t_2[K]$.

i.e., **no two tuples** in any legal relation r(R) may have the **same value** on **attribute set K**.

The notion of **FD generalizes** the notion of **superkey**.
Consider a relation schema R, and let $\alpha \subseteq R$ and $\beta \subseteq R$.
The **Functional Dependency**
$\alpha \rightarrow \beta$
holds on schema R if, in any **legal relation** r(R), for **all pairs** of **tuples** t_1 and t_2 in r **such that** $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.

Using the FD notation -- say that K is a superkey of R if $K \rightarrow R$.
i.e. K is a superkey if, whenever $t_1[K] = t_2[K]$, it is also the case that $t_1[R] = t_2[R]$ (that is, $t_1 = t_2$).

Note: The left and right sides of an FD -- called the **determinanat** and the **dependent** respectively

Allow us to express constraints that we cannot express with superkeys.
e.g., Loan-info (loan-no, br-name, cust-name, amount)
The set of FDs -- expect to hold on this schema is
loan-no →amt
loan-no →br-name
not expect the FD ***loan-no →cust-name*** to hold
loan can be made to **more than one customer**

e.g. husband–wife pai).

use FDs in two ways

1. To **test relations** to see whether they are **legal under** a given set of FDs.

If a relation **r** is **legal** under a **set F** of **FDs**, we say that **r satisfies F**.

2. To **specify constraints** on the set of **legal relations**.

thus -- concern **only** those relations that **satisfy** a given set of FDs.

If relations on schema R that **satisfy** a set F of FDs, we say that F **holds** on R.

Let us consider the relation r of Fig, to see which FDs are satisfied.

Observe that FD $A \rightarrow C$ is satisfied.

There are two tuples that have an A value of a1. These tuples have the same C value, c1.

Similarly, the two tuples with an A value of a2 have the same C value, c2.

There are **no** other **pairs** of **distinct tuples** that have the same **A value**.

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

The FD $C \rightarrow A$ is not satisfied.

see **why it is not**

consider the tuples t1 = (a2, b3, c2, d3) and
t2 = (a3, b3, c2, d4).

-- tuples have the same C values, c2, but they have different A values, a2 and a3, respectively.

have found a pair of tuples t1 and t2 such that
t1[C] = t2[C], but t1[A] <> t2[A].

Trivial and Nontrivial

said to be **trivial** -- are satisfied by all relations.

e.g. $A \rightarrow A$ is satisfied by all relations involving attribute A.

Reading the definition of FD, we see that, for all tuples t1 & t2 such that t1[A] = t2[A], it is the case that t1[A] = t2[A]

||ly, $AB \rightarrow A$ is satisfied by all relations involving attribute A.

In general, a FD of the form **$\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$**

i.e., an FD is Trivial iff the **right side** is a **subset** (not necessarily a proper subset) of the **left side**

Closure of a Set of Functional Dependencies

-- is not sufficient to consider the given set of FDs.

need to consider **all FDs** that **hold**.

With the given a set F FDs we can show that certain other FDs hold.

say that such FDs are "**Logically Implied**" by F.

e.g. consider a relation schema R = (A, B, C, G, H, I) and the set of FDs

$A \to B$, $A \to C$, $CG \to H$, $CG \to I$, and $B \to H$

The FD $A \to H$ is logically implied

Suppose that t1 and t2 are tuples such that

t1[A] = t2[A]

Since we have $A \to B$ -- from the definition of FD t1[B] = t2[B]

similarly we have $B \to H$

t1[H] = t2[H]

Therefore whenever t1 and t2 are tuples such that

t1[A] = t2[A]

it must be that

t1[H] = t2[H].

exactly the definition of $A \to H$.

Note: If F were large --would be lengthy and difficult.

Let S be a set of functional dependencies. The **closure of S, denoted by S+,** is the set of all FDs logically implied by S.

Axioms, or Set of Inference Rules

provide a simpler technique for reasoning about FDs

Reflexivity rule.

If B is a subset of A i.e. $B \subseteq A$, then $A \to B$.

Augmentation rule

If $A \to B$ holds, then $AC \to BC$

Transitivity rule

If $A \to B$ and $B \to C$ then $A \to C$

Self-determination

$A \to A$

Decomposition

If $A \to BC$, then $A \to B$ and $A \to C$

Union rule

If $A \to B$ and $A \to C$, then $A \to BC$

Composition rule

If $A \to B$ and $C \to D$ then $AC \to BD$

Pseudotransitivity rule.

If $A \to B$ and $CB \to D$ then $AC \to D$ holds.

Note: D another arbitrary subset of the set of attributes of R e.g. schema R = ((A, B, C, G, H, I) and the set S of FDs

$\{A \rightarrow B,\ A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$.

We Let us apply our rules to list several members of S+

$A \rightarrow B$ and $B \rightarrow H$	Given
$A \rightarrow H$	transitivity rule.
$CG \rightarrow H$	Given
$CG \rightarrow I$	Given
$CG \rightarrow HI$	the union rule
$A \rightarrow C$	Given
$CG \rightarrow I$	Given
$AG \rightarrow I$	pseudotransitivity rule

Another way of finding -- $AG \rightarrow I$ holds is

$A \rightarrow C$	Given
$AG \rightarrow CG$	augmentation rule
$CG \rightarrow I$	Given
$AG \rightarrow I$	transitivity rule

Closure of Attribute Sets

algorithm

Let **α** be a set of attributes.

α+ -- set of all attributes functionally determined by **α** under a set F of FDs

the **closure** of **α** under F

result := α;
while (changes to result) **do**
for each FD β → γ **in** F **do**
begin
if β ⊆ result **then**
result := result ∪ γ;
end

e.g. schema R = ((A, B, C, G, H, I) and the set S of FDs
$\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$.

compute the closure $(AG)+$

start with *result* = *AG*.

Once -- execute the **while** loop to test each FD

$A \rightarrow B$ -- to include *B* in *result*.

since $A \rightarrow B$ is in *F*, $A \subseteq$ *result* (*AG*)
result := *result* $\cup B$.

$A \rightarrow C$ -- *result* to become *ABCG*.

$CG \rightarrow H$ -- *result* to become *ABCGH*.

$CG \rightarrow I$ -- *result* to become *ABCGHI*.

The next time -- execute the **while** loop

no new attributes are added to *result* and terminates

Canonical Cover

Consider the following set F of FDs on schema (A,B,C,D):

$A \to BC$; $B \to C$; $A \to B$; $AB \to C$, $AC \to D$

Let us compute the canonical cover for F.

There are two FDs with the same set of attributes on the left side of the arrow:

1. 1st step – rewrite all FDs such that Each has a singleton

   $A \to B$

   $A \to C$

   $B \to C$

   $A \to B$

   $AB \to C$

   $AC \to D$

   FD $A \to B$ occurs twice – one can be eliminated

2. Next, attribute C can be eliminated from LHS of the FD $AC \to D$

   $A \to AC$ by augmentation

   $AC \to D$ given

   $A \to D$ transitivity

   So the C on the LHS of $AC \to D$ is redundant

3. Next, we observe that the FD $AB \to C$ can be eliminated

   $A \to C$ given

   $AB \to CB$ by augmentation

   $AB \to C$ by decomposition

4. Finally, the $A \to C$, is implied by the FDs

   $A \to B$ and $B \to C$,

   so it can also be eliminated. We are left with: $A \to B$, $B \to C$, $A \to D$

Thus, our canonical cover is

$A \to B$, $B \to C$, $A \to D$

Normalization

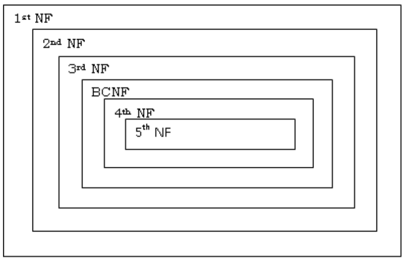
Supplier { s_no, s_name, status, city}

Parts {p_no, P_name, color, wgt, city}

Shipment {s_no, p_no, qty}

-- built around the concept of Normal Forms

-- E. F. Codd – 1^{st}, 2^{nd}, 3^{rd}, and BCNF ; 4^{th} and 5^{th} – Fagin



1^{st} NF -- A relation is in 1^{st} NF, iff every tuple contains exactly one value for each attribute

FIRST

s_no	status	city	p_no	qty
S1	20	Hyd	P1	100
S1	20	Hyd	P2	150
S2	10	Sec'bad	P1	150
S2	10	Sec'bad	P2	200
S3	10	Sec'bad	P1	300
S4	20	B'lore	P2	150
S4	20	B'lore	P3	400

FIRST { s_no, status, city, p_no, qty}

Redundancies in FIRST -- many problems
usually -- updation anomalies

i. s_no, city – Redundancy

ii. Problem occurs each of the following updations
i.e. INSERT, DELETE, and UPDATE

INSERT

cannot insert a particular supplier is located in particular city
until supplier supplies at least ONE part

say, S5 -- located in VSKP -- not included
Reason until supplies some part, **--** **no** appropriate **Primary Key** value

DELETE

Delete the only FIRST tuple for a particular Supplier
not only the **shipment** relating to that **supplier** to a **part info**. but also supplier **located in a city**
e.g. Delete S4 and p_no value P3 loose the info. of S4's city

UPDATE

city info. -- supplier appears in FIRST many times

leads **Redundancy** problems

e.g. supplier S2 moves from Sec'bad to VSKP

problem of searching or producing an inconsistent result

Solution -- **further Normalization**

SECOND { s_no, status, city}   SHIPMENT { s_no, p_no, qty}

SECOND

s_no	status	city
S1	20	Hyd
S2	10	Sec'bad
S3	10	Sec'bad
S4	20	B'lore

s_no	p_no	qty
S1	P1	100
S1	P2	150
S2	P1	150
S2	P2	200
S3	P1	300
S4	P2	150
S4	P3	400

INSERT

can insert S5 is located in particular city though he does supply any part

DELETE

Delete S4 and p_no P3 from **SHIPMENT** by deleting relevant tuple.

without loosing the info. of S4's city

UPDATE

city info. -- supplier appears just once

since s_no -- Primary Key in SECOND

redundancy has been eliminated

2nd NF

A relation is in 2nd NF iff it is in 1st NF and every non-key attribute is irreducibly (transitively) dependent on the Primary Key

s_no -- Primary Key for SECOND

s_no & p_no -- Primary Key for SHIPMENT

SECOND -- **causes problems**

SHIPMENT – Satisfactory

Note: The above def is -- by assuming only one Candidate Key

The collection of projections -- obtained -- equivalent to original Relation

INSERT

cannot insert a **particular city** has a particular **status**

e.g. cannot state -- supplier in CHENNAI should have status value 40

until supplier -- actually located

DELETE

Delete the only tuple in SECOND for a particular city

not only deleting the details of **supplier** but also **status** value of the **city**

e.g. Delete S4

loose the info. of S4's status -- B'lore is 20

UPDATE

Status info. -- City appears in SECOND many times

leads redundancy problems

e.g. change in status value of Sec'bad from 10 to 30

problem of searching or producing an inconsistent result

Solution -- further Normalization

SC { s_no, city}

CS {status, city}

SHIPMENT { s_no, p_no, qty}

s_no	city
S1	Hyd
S2	Sec'bad
S3	Sec'bad
S4	Hyd

city	status
Hyd	20
Sec'bad	10
Vskp	30
B'lore	40

3rd NF

A relation is in 3rd NF iff it is in 2nd NF and every non-key attribute is non-transitively(irreducibly) dependent on the Primary Key

Dependency Preservation

-- another Goal in RDB design

given relation -- non-loss decomposition -- many ways

s_no → city

city → status -- decomposition X

s_no → city

s_no → status -- decomposition Y

Both -- non-loss decompositions

decomposition Y less satisfactory than X

e.g., cannot insert a particular city has a particular status unless some supplier actually located

Observe Both more closely

decomposition X

Two Projections are independent of one another

i.e. Updates -- be made to either one

decomposition Y

updations of either Two projections must be monitored to ensure FD **city → status**

other words TWO projections – not independent each other

BCNF

Codd's original 3rd NF did not treat the general case satisfactorily

To be precise – did not deal with the case of relation that

Had Two or more Candidate Keys, ∍

i. the Candidate Keys were Composite

ii. they overlapped(i.e. had at least One attribute common)

A relation is in BCNF iff every determinant is a Candidate Key OR

A relation is in BCNF iff every nontrivial left-irreducible FD has a Candidate Key as its determinant.

4th NF

A relation R is in 4th NF iff, whenever there exists subsets A & B of the attributes of R ∍ the non-trivial MVD A→→B is satisfied,

then all attributes of R also fully dependent on A

Note: The MVD A→→B trivial if either A is a superset of B or the union AB of A and B is the entire heading

5th NF or P/J NF

A relation R is in 5th NF iff, for every join dependency that holds on R, one of the following statements is true

R_i = R for some i, or

the JD is implied by the set of those FDs over R in which the left side is the key for R

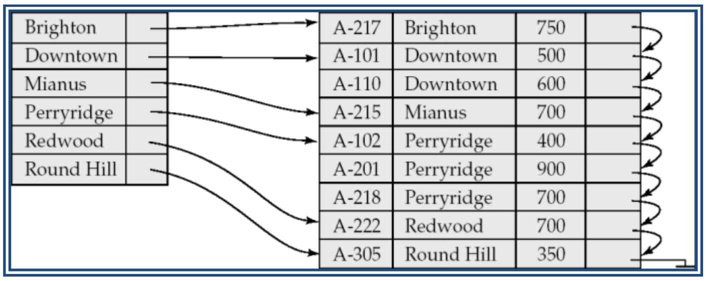
DBMS - Indices

Ordered Indices

assume that all files are **ordered sequentially** on some **search key** -- called **index sequential files**

designed for applications that require both sequential processing of the entire file and random access to individual records.

Dense index -- Index record appears for every search-key value in the file.



Sparse Index:

contains **index** records for only **some search-key value**

Applicable when records are sequentially ordered on search-key

To locate a record with **search-key** value ***K*** we:

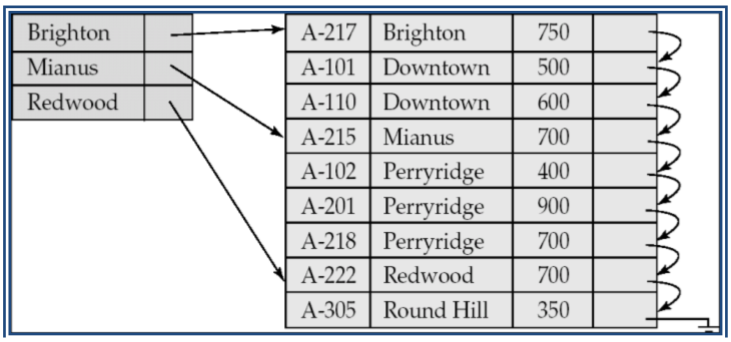
Find index record with **largest search-key** value **<= *K***

Search file sequentially starting at the record to which the index record points

Compared to dense indices:

Less space and **less maintenance** overhead for insertions and deletions.

Generally **slower than dense index** for locating records.



Index Update - Record Insertion

Insertion:

Perform a lookup using the key value from inserted record

Dense indices

if the search-key value does not appear in the index, system inserts with the search-key value at an appropriate position

Otherwise

a. If the index record stores pointers to all records with the same search-key value, the system adds a pointer to the new record to the index record.
b. If not, the system places the record being inserted after the other records with the same search-key values.

Sparse indices

We assume that the index stores an entry for each block.

If a new block is created, the first search-key value appearing in the new block ` is inserted into the index

if the new record has the least search-key value in its block,

the system updates the index entry pointing to the block;

if not, the system makes no change to the index.

Deletion:

To delete a record, the system first looks up the record to be deleted

Dense indices

If deleted record was the only record with its search-key value

the search-key is deleted from the index

Otherwise

a. If the index record stores pointers to all records with the same search-key value,

   the system deletes the pointer to the deleted record from the index record.

b. if the deleted record was the first record with the search-key value,

   updates the index record to point to the next record.

Sparse indices

If the index does not contain an index record with the search-key value of the deleted record

nothing needs to be done to the index

Otherwise

a. If the deleted record was the only record with its search key

   replaces the corresponding index record with an index record for the next search-key value.

   If the next search-key value already has an index entry, the entry is deleted instead of being replaced

b. if the index record for the search-key value points to the record being deleted,

   the system updates the index record to point to the next record with the same search-key value.

Multilevel Index

Even if we use a sparse index, the index itself may become too large for efficient processing. in practice,

e.g. a file with 100,000 records -- unreasonable with 10 records stored in each block.

If we have one index record per block, the index has 10,000 records.

Index records are smaller than data records,

assume that 100 index records fit on a block.

Thus, our index occupies 100 blocks.

Such large indices are stored as sequential files on disk.

If an index is **sufficiently small** to be kept in main memory, the **search time** to find an **entry** is **low**.

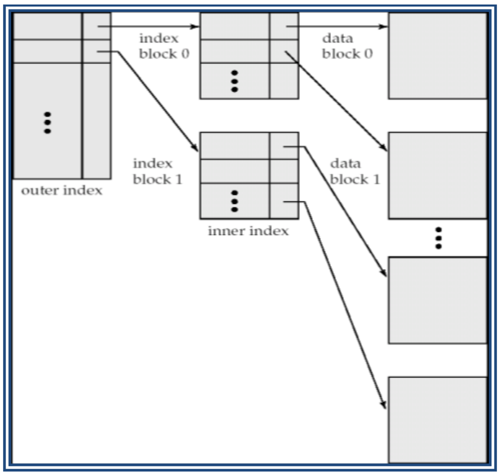
Otherwise -- requires several disk block reads.

If primary index does **not fit** in memory, **access** becomes **expensive**.

Sol: treat primary index kept on disk as a sequential file and construct a **sparse index** on it

Outer index – a sparse index of primary index

Inner index – the primary index file



If even **outer index is too large** to fit in main memory,

yet **another level** of **index** can be **created**, and **so on**.

Indices with **two or more** levels -- **Multilevel Indices**

Dictionary is an example of a multilevel index

-- are closely related to tree structures

requires significantly **fewer I/O operations** than does searching for records by **binary search**.

Note: Indices at all levels must be updated on insertion or deletion from the file.

Indexed Sequential Access Method (ISAM)

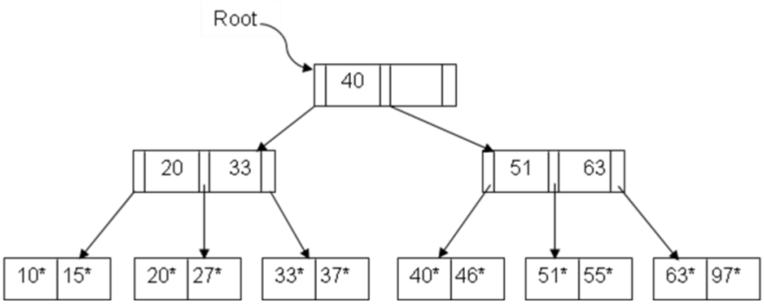
is completely static (except for the overflow pages)

Each tree node is a disk page, and all the data resides in the leaf pages

e.g. consider tree following fig.

All searches begin at the root.

e.g. to locate a record with the key value **27**



start -- root & follow the left ptr, since **27 < 40**.

then follow the middle ptr, since **20 <= 27 < 33**

Assume – each leaf page --contain **two** entries.

If we now insert a record with key value **23**,

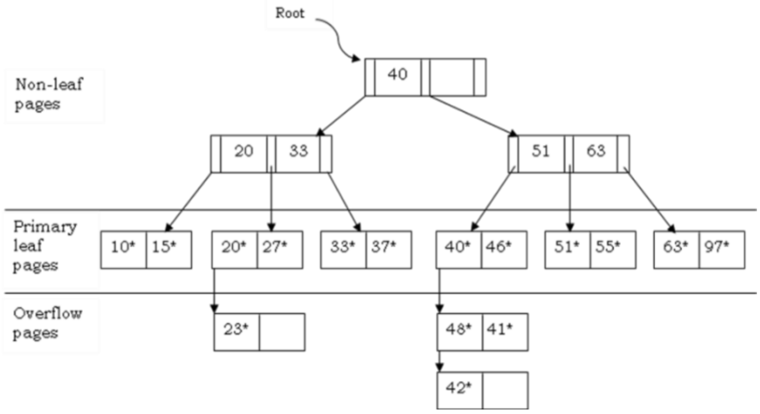
the entry **23*** belongs in the second data page, already contains **20*** and **27***

no more space.

by adding an **overflow page** and putting **23*** in the overflow page.

Chains of overflow pages can easily develop.

For instance, inserting **48*, 41*,** and **42*** leads to an overflow chain of TWO pages
Chains of overflow pages can easily develop.



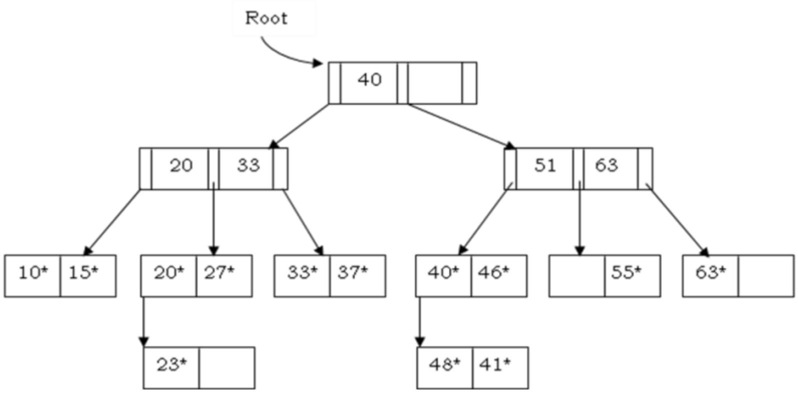
The deletion of an entry k is handled by simply removing the entry.
If this entry is on an overflow page and
the overflow page becomes empty, the page can be removed.
If the entry is on a primary page and
deletion makes the primary page empty,
simply leave the empty primary page for future insertions

Thus, the number of primary leaf pages is fixed at file creation time.

Note: that after deleting 51*, the key value 51 continues to appear in the index level.
A subsequent search for 51* would go to the correct leaf page and determine that the entry is not in the tree.

Note: only leaf pages are **modified**

Fig. shows after deletion of the entries 42*, 51*, and 97*



Advantage of indexed-sequential files
concurrent access, -- index pages -- never modified
Therefore, -- can avoid locking
Disadvantage of indexed-sequential files
performance degrades as file grows, since many overflow pages get created.
Periodic reorganization of entire file is required.

Note: to reduce the overflow pages, tree will be created -- 20% free space of each page.
once free space -- filled -- overflow pages

Note: The number of disk I/Os is = the number of levels of the tree

B+ - Tree Index Files

B^+-tree indices -- an alternative to indexed-sequential files
- -- balanced tree, -- adjusts greatly with insertions and deletions
- -- dynamic, it is not possible to allocate the leaf pages
- -- min of 50 percent occupancy.
- -- All paths from root to leaf are of the same length

Note: because of high fan-out, the **height** of a **B^+-tree** is **rarely** more than **3** or **4**

Advantage of B^+-tree index files

preferable because inserts are handled gracefully **without overflow chains**

Reorganization of entire file is **not required** to maintain performance

Disadvantage of B^+-trees

extra insertion and deletion overhead, space overhead

advantage of B^+-trees outweigh disadvantages B^+- trees are used extensively

B^+-tree -- a rooted tree satisfying the following **properties**

All paths from root to leaf are of the same length

Each node contains **m** entries, where **d <= m <= 2d**.

d is a the **order** of the tree, and is a measure of the **capacity** of a **tree node**. Say, e.g. **d=2**

Leaf nodes must have between 2 and 4 values

Root must have at least 2 children.

Special cases

If the **root** is **not a leaf**, it has **at least 2 children**.

If the **root** is a **leaf** (i.e., there are no other nodes),

is simply required that **1 <= m <= 2d**

Search

use the notation *ptr to denote the value -- pointer variable ptr and & (value) to denote the address of value

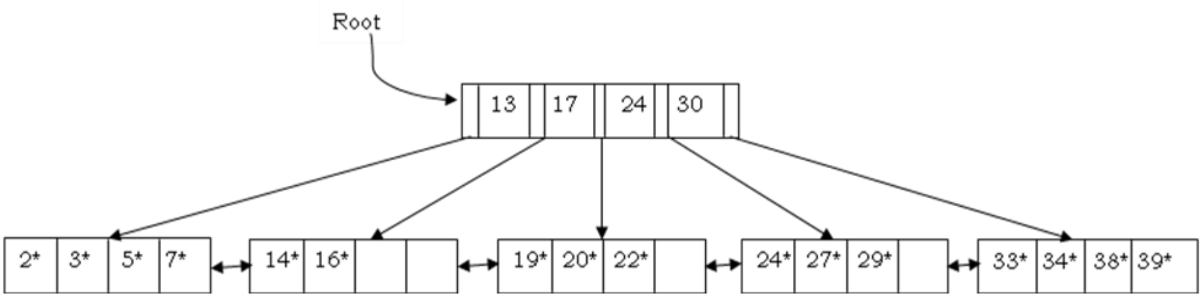
Consider the B+ tree shown below of order d=2.

i.e. each node contains between 2 and 4 entries.

Each non-leaf is a <key-value, node-pointer> pair

To search for entry 5*

follow the left-most child pointer, since 5 < 13



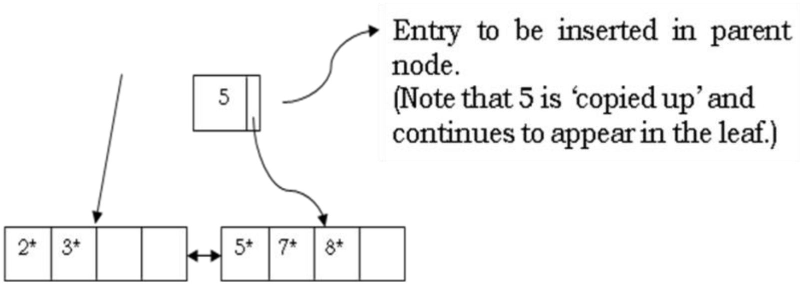
e.g., B+ Tree, Order d=2

INSERT

If a node is full -- must be split.
When the **node** is **split**,
an **entry pointing** to the **node created** by the **split** must be **inserted** into **its parent**;
this entry is pointed to by the pointer variable **new-child-entry**.

If the **root (old)** is **split**,
a new root node is created and the **height** of the tree increases by **one**

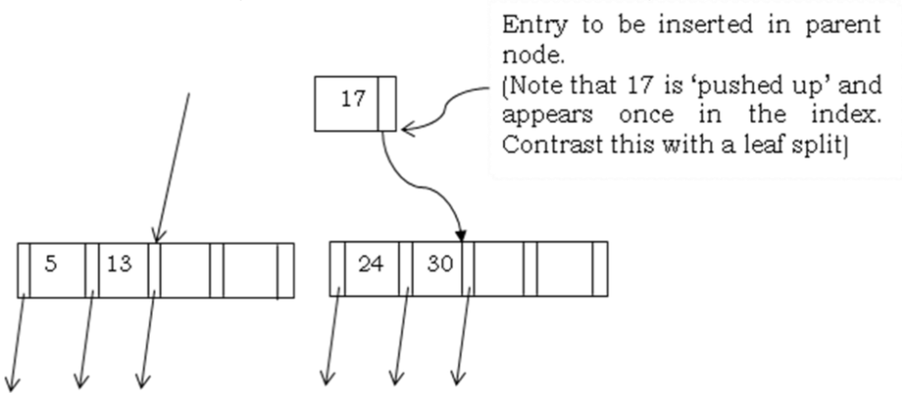
To insert 8*
search for an appropriate node
follow the **left-most** child pointer, since **8 < 13**
it belongs in the **left-most leaf**, which is already **full.**
causes a split of the leaf page; the split pages are shown below



Split Leaf Pages during Insert of Entry 8*

Note: cannot just **push up** 5, because every **data entry** must **appear** in a **leaf page**

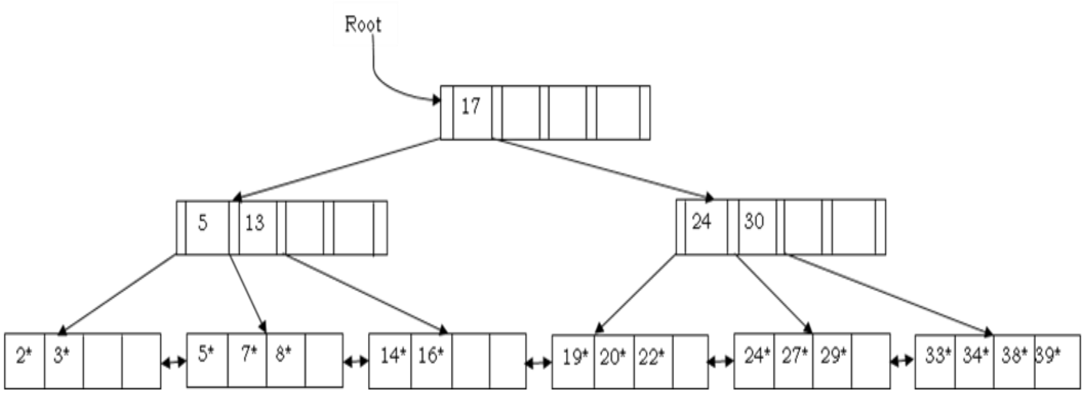
Observe, the parent node is also full, which is non-leaf node
split occurs -- **non-leaf node**
yielding **two minimally** full **non-leaf** nodes, each containing **d keys** and **d+1 pointers**, and an extra key, -- choose to be the **middle** key.



Split Index Pages during Insert of Entry 8*

This key and a pointer to the second non-leaf node constitute an index entry
that must be inserted into the **parent** of the **split non-leaf node.**

The middle key is thus **pushed up** the tree



B+ Tree after Inserting Entry 8*

Redistribution

Reconsider insertion of **entry 8*** into the previous tree

The entry belongs in the **left-most leaf**, which is **full**.

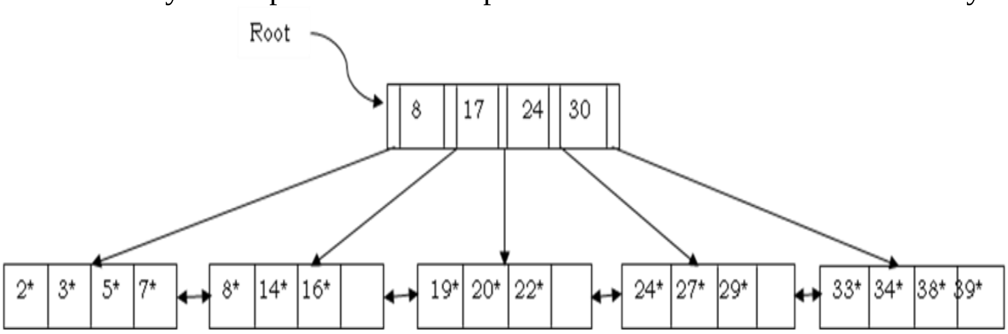
only **sibling** of this leaf node contains only **two** entries and

thus **accommodate** more entries.

-- handle the insertion of 8* with a **redistribution**.

We **copy up** the new low key value on the second leaf. This process is **Redistribution**

Note: how the entry in the parent node that points to the second leaf has a new key value;



B+ Tree after Inserting Entry 8* Using Redistribution

To determine whether redistribution is possible,

retrieve the sibling.

If the sibling happens to be **full**, -- to **split** the node anyway.
Checking whether redistribution -- may **reduce I/O** if the redistribution **succeeds**

If a **leaf** node is **full**,
fetch a **neighbor** node; if it has **space**, and has the **same parent**, redistribute entries.
Otherwise
split the leaf node and adjust the **previous** and **next**-neighbor pointers in the **split node**

Note: checking the redistribution **increases I/O**
especially if we check both siblings

a **limited** form of **redistribution** makes sense

Transactions and Concurrency Control

Transaction -- a **collection of operations** that form a **single** logical **unit** of **work**. OR
a **unit** of **program** execution that accesses and possibly updates various data items.
e.g. transfer of money from one acc to another
consists of two updates, one to each acc.

To **ensure integrity**
--the db maintain the following transactions properties

Atomicity
Either **all operations** of the transaction are **reflected** properly in the db or **none** are.

Consistency
execution of a transaction in **isolation** (i.e. no other transaction executing concurrently) preserves the **consistency** of the db

Isolation
though multiple transactions may execute concurrently, the system guarantees -- for every **pair** of transactions
$\mathbf{T_i}$ and $\mathbf{T_j}$ either
$\mathbf{T_j}$ **finished** execution before $\mathbf{T_i}$ **started**, or
$\mathbf{T_j}$ **started** execution after $\mathbf{T_i}$ **finished**.

Durability
after **successful completion** of a transaction, the changes it has made to the **db persist**, inspite of **system failures.**

These properties -- the **ACID** properties

Transactions access data using two operations:
Read(A) -- transfers the data item **A** from the db to a local buffer
Write(A) -- transfers the data item **A** from the local buffer and write back to the db.

ACID Properties

Assume, **Write** operation updates the db **immediately**
Let $\mathbf{T_i}$ be a transaction that transfers 5000 from account **A** to account **B**.

```
Ti : read(A);
     A := A − 5000;
     write(A);
     read(B);
     B := B + 5000;
     write(B).
```

NOTE: In real db, the write operation does **not necessarily** result in the **immediate data update** on the disk

Atomicity -- before the execution of transaction T_i the values of **A** and **B** --10000 & 20000 respectively
if the transaction fails **after** step 3 and **before** step 6, money will be **lost** -- to an **inconsistent** db state
failure -- be due to **SW or HW**
-- is present, all actions of the transaction -- reflected in the **db or none** are.

Basic idea -- db **keeps** track of the **old values** of any data on which a transaction performs a **write**
if the transaction does **not complete** its **execution**, the db **restores the old values**

-- is handled by a component called the **transaction- mgt component**

Consistency

-- is that the sum of **A** and **B** be unchanged after the transaction execution

can be verified easily --

if the db is consistent **before** transaction **execution**, the db remains consistent **after** the **execution**.

is the responsibility of the **application programmer** who codes.

Isolation

-- even if the consistency & atomicity are ensured, if several transactions -- executed concurrently -- **not** resulting in an **inconsistent** state.

Ensuring this -- the responsibility of the **concurrency-control Component**

e.g.	T_1	T_2
	read(A)	
	A := A – 5000	
	write(A)	
		read(A)
		read(B)
		print(A+B)
	read(B)	
	B := B + 5000	
	write(B)	

A+B -- an **inconsistent** value.

Also, if this 2nd transaction then performs updates on **A** and **B** based on the **inconsistent values** that it read,

the db may be left in an **inconsistent state** even after **both transactions** have completed

to avoid concurrent execution --

execute transactions **serially** i.e. one after the other

Durability

property guarantees -- all the updates -- carried out on the db persist,

irrespective of **system failure** after the transaction completes execution.

can guarantee -- by ensuring that either

the **updates** carried out by the transaction have been **written** to **disk** before it **completes**.

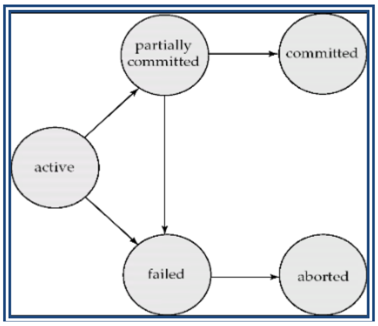
to enable the db to **reconstruct** the **updates** when the **db** is **restarted** after the **failure**.

Ensuring durability -- responsibility of **recovery-mgt component**

Transaction State

Active -- the initial state; the transaction stays in this state while it is executing
Partially Committed -- after the final statement has been executed.
Failed -- after the discovery that normal execution can no longer proceed.

Aborted -- after the transaction has been rolled back and the db restored to its prior state.
Two choices after it has been aborted
restart the transaction
can be done only if no internal logical error kill the transaction
Committed -- after successful completion



Implementation of Atomicity and Durability

Recovery-management component

can support for atomicity and durability by a variety of schemes

1st -- a simple, but extremely inefficient, scheme called the **shadow copy scheme** -- making **db copies**

all updates are made on a **shadow copy** of the db

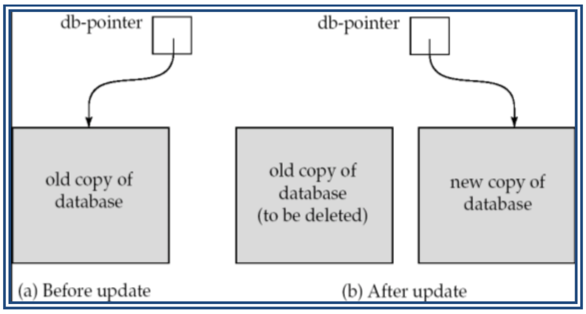
assumes -- only one transaction is active at a time

also assumes -- the db is simply a file on disk

db_pointer -- pointer maintained on disk always points to the current consistent db copy

If transaction fails,

old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted



Look -- how the technique handles transaction and system failures.

1st -- consider **transaction failure**.

If the transaction fails at any time **before** db-pointer is **updated**

the **old contents** of the db are **not affected**.

can abort the transaction

Once the transaction has been committed -- all the updates are in the **db pointed** to by **db-pointer**

either all updates -- reflected, or none are reflected

2nd -- consider the issue of **system failure**.

1. Suppose that the system **fails before** the **updated** db-pointer is written to disk.

   when the **system restarts** -- will **read** db-pointer

   see the original contents of the db, and

   none of the effects of the transaction will be visible on the db

2. suppose that the system **fails after** db-pointer has been **updated** on disk.

   all updated pages of the **new copy** of the db were **written** to disk.

   its contents will **not** be **damaged**

   when the **system restarts**, it will read **current** db-pointer thus **see** the contents of the **db** after **all updates**

Unfortunately, -- is extremely **inefficient** -- **db** is **large**

executing a **single** transaction -- **copy** the **entire db**.

Also, it does **not allow concurrent** execution of transactions

There are practical ways of implementing atomicity and durability -- are much **less expensive** and **more powerful**

study these recovery techniques

Concurrent Executions

Advantages of allowing concurrent transactions execution

Increased Processor and **Disk Utilization** -- leading to better transaction **throughput**

e.g., **one transaction** -- be using the **CPU** while another is **reading** from **or writing** to the disk

Reduced Waiting Time -- short transactions need not wait behind long ones.

If transactions **run Serially** -- a **short transaction** may have to **wait** till long transaction to complete

-- lead to **unpredictable delays**

Schedule

-- a **sequences** of instructions -- specify the **chronological** order of e**xecution** of concurrent transactions

must preserve the order in which the instructions appear in each individual transaction.

A transaction that **successfully** completes its **execution** will have a **commit** instructions as the **last statement**

Note: By default transaction **assumed** to execute **commit** instruction as its last step

fails to successfully complete its execution -- have an **abort** instruction as the last statement

Schedule 1

Let T_1 transfer 5000 from A to B, and T_2 transfer 10% of the balance from A to B.
A **serial** schedule in which T_1 is followed by T_2

T_1	**T_2**
read(A)	
A := A – 5000	
write(A)	
read(B)	
B := B + 5000	
write(B)	
	read(A);
	temp := A * 0.1;
	A := A − temp;
	write(A);
	read(B);
	B := B + temp;
	write(B).

Schedule 2

A serial schedule where T_2 is followed by T_1 the sum A and B is preserved

T_1	T_2
	read(A);
	temp := A * 0.1;
	A := A − temp;
	write(A);
	read(B);
	B := B + temp;
	write(B).
read(A)	
A := A – 5000	
write(A)	
read(B)	
B := B + 5000	
write(B)	

Schedule 3 - Concurrent Executions

Schedule is not a serial schedule, but it is **equivalent** to Schedule 1

T_1	T_2
read(A)	
A := A – 5000	
write(A)	read(A);
	temp := A * 0.1;
	A := A − temp;
	write(A);
read(B)	
B := B + 5000	
write(B)	read(B);
	B := B + temp;
	write(B).

The schedules 1, 2 and 3, the sum A + B is preserved

Note: The possible schedules for a set of **n** transactions is much **larger than n!**

Schedule 4 - Concurrent Executions

Not all concurrent executions result in a **correct** state. Consider the following

T_1	T_2
read(A)	
A := A – 5000	read(A);
	temp := A * 0.1;
	A := A − temp;
	write(A);
read(B)	
write(A);	
read(B);	
B := B + 5000	
write(B)	B := B + temp;
	write(B).

final values of accounts **A** and **B** -- an **inconsistent state**

If control of **concurrent execution** is left entirely to the **OS**, many schedules -- possible including ones that leave the db in an inconsistent state

We can ensure consistency of the db --
the schedule must, in some sense, be **equivalent** to a **serial** schedule.

Note: The db system to ensure -- any **schedule** that gets executed will leave the d b in a **consistent** state.

The **concurrency-control component** of the **db** system **carries out this** task

Serializability

-- to ensure that the db state remains consistent

Before that -- first understand which schedules will ensure consistency, and which will not.

we consider only two operations -- read and write.
thus assume that a **read(X)** instruction and a **write(X)** instruction on a **Data item X**

Schedule 3 -- considered as

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Basic Assumption -- Each transaction preserves db consistency

Thus serial execution of a set of transactions preserves db consistency

discuss different forms of schedule equivalence; they lead to

Conflict Serializability

View Serializability

Conflict Serializability

consider a schedule S -- two consecutive instructions I_i and I_j of transactions T_i and T_j respectively (i<>j)

If $\mathbf{I_i}$ and $\mathbf{I_j}$ refer to **different data items** -- can **swap** I_i and I_j without affecting the results

if I_i and I_j refer to the same data item X, then the order of the two steps may matter

1. **I_i = read(X), I_j = read(X).** The order of $\mathbf{I_i}$ and $\mathbf{I_j}$ does not matter
   same value of X is **read by T_i** and **T_j**, regardless of the order.
2. **I_i = read(X), I_j = write(X).**
   If $\mathbf{I_i}$ comes before $\mathbf{I_j}$, then $\mathbf{T_i}$ does **not read** the **value** of X that is written by T_j

   If $\mathbf{I_j}$ comes before $\mathbf{I_i}$, then $\mathbf{T_i}$ **reads** the value of X that is written by T_j.
   Thus, the order of $\mathbf{I_i}$ and $\mathbf{I_j}$ matters.
3. **I_i = write(X), I_j = read(X).** The order of $\mathbf{I_i}$ and $\mathbf{I_j}$ **matters** for reasons similar to the previous case.

4. **I_i = write(X), I_j = write(X).** Since both are write operations

the **order** of these instructions does **not affect** either **T_i or T_j** .

But, the value obtained by the **next read(X)** of S is **affected**

since the result of only the **latter** of the **two write** instructions is **preserved** in the db.

If there is **no** other **write(X)** after **I_i** and **I_j** in S, then
the **order** of **I_i** and I_j **directly affects** the **final** value of X in the db state that results

say that I_i and I_j **conflict** if different transactions operate on the same data item
and at least one of these instructions is a write .

to understand the concept of conflicting instructions, consider schedule 3

The **write(A)** instruction of **T_1 conflicts** with the **read(A)** instruction of **T_2**.

However, the **write(A)** of **T_2** does **not conflict** with the **read(B)** of **T_1**

two access different data items.

Let I_i and I_j be consecutive instructions of a schedule S.

If I_i and I_j are of different transactions and I_i and I_j do not conflict,
then we can swap the order of I_i and I_j to produce a new schedule S.

We expect **S** to be **equivalent** to **S^I**

in schedule 3 does not conflict with the read(B) instruction of T_1
we can swap these instructions to generate an equivalent schedule see in schedule 5

Note: Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

Schedule 5 -- schedule 3 after swapping of a pair of instructions

T_1	**T_2**
read(A)	
write(A)	
	read(A);
read(B)	
	write(A);
write(B)	
	read(B);
	write(B).

We continue to swap non-conflicting instruction

Swap the read(B) of T1 with the read(A) of T2.

Swap the write(B) of T1 with the write(A) of T2.

Swap the write(B) of T1 with the read(A) of T2.

The final result of these swaps, see the below schedule 6

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A);
	write(A);
	read(B);
	write(B).

Thus, we have shown that schedule 3 is equivalent to a serial schedule

If a schedule S can be transformed into a schedule S^I by a series of swaps of non-conflicting instructions,

we say that S and S^I are **conflict equivalent**

In our previous e.g schedule 1 is not conflict equivalent to schedule 2.

However, schedule 1 is conflict equivalent to schedule 3,

since the read(B) and write(B) of T_1 can be swapped with the read(A) and write(A) of T_2

We say that a schedule S is **conflict serializable** if it is **conflict equivalent** to a **serial** schedule

Observe the below schedule 7 -- is not conflict serializable

unable to swap instructions in the above schedule to obtain either

the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

T_3	T_4
read(X)	
	write(X)
write(X)	

View Serializability

Let *S* and S^I be two schedules -- the same set of transactions

S and S^I are **view equivalent** if the following three conditions are met

1. For each data item X,
   if transaction T_i reads the **initial value** of **X** in **S**, then T_i must also read the **initial value** of **X** in **S^I**
2. For each data item X,
   if transaction T_i executes **read(X) in S** and if that value was produced by a **write(X)** executed by transaction **T_j**, then the read(X) of T_i must also **similar** in **S^I**
3. For each data item X,
   the transaction (if any) that performs the **final write(X)** in **S** must perform the **final write(X)** in **S^I**.

A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.

Every **conflict serializable** schedule is **also view serializable**, but **all the view serializable** schedules are **not conflict serializable**

Below is an example of a schedule which is view serializable

T_3	T_4	T_6
read(X)		
	write(X)	
write(X)		
		write(X)

it is view equivalent to the serial schedule <T_3, T_4, T_6>,

since the **one read(X)** reads the initial value of X in **both schedules**, and

T_6 performs the **final write of X** in both schedules.

Observe the above schedule, **T_4** and **T_6** perform write(X) without a read(X). These Writes -- called **blind writes**.

Every view serializable schedule that is not conflict serializable has **blind writes**

Recoverability

we have seen the schedules -- **assuming implicitly** that there are **no transaction failures**

If a transaction T_i fails

-- to ensure the atomicity

any transaction **T_j dependent on T_i** (i.e. T_j has read data written by T_i) is **also aborted**

See the issues of what schedules are acceptable from the recovery point of view when a transaction fails

Consider below schedule

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

T_9 -- performs only one instruction - read(A)

-- assume if it **commit immediately** after read(A)

Thus, **T_9 commits before T_8** does.

Now suppose that **T_8 fails before** it commits.

Since **T_9** has **read** the value of **A written** by **T_8**, we must **abort T_9** to ensure transaction atomicity.

However, **T_9** -- already committed & **cannot** be **aborted**.

Thus -- a situation where it is impossible to recover correctly from the **failure** of **T_8**.

Previous **Schedule** -- an e.g. of a **non-recoverable schedule**

should not be allowed.

A **Recoverable Schedule**

if a transaction **T_j reads** a data item **previously written** by a transaction **T_i**,

then the **commit** operation of **T_i** appears **before** the **commit** operation of **T_j**.

If a schedule is recoverable -- may have to roll back several transactions.

consider the partial schedule

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

T_{10} writes a value of **A** that is read by **T_{11}**. T_{11} writes a value of A that is read by T_{12}

Suppose that, at this point, **T_{10} fails**.

T_{10} must be **rolled back**.

Since T_{11} is dependent on T_{10},

T_{11} must be **rolled back**.

Since T_{12} is dependent on T_{11},

T_{12} must be **rolled back**.

a single transaction failure leads to a series of transaction rollbacks -- called **cascading rollback.**

it leads to the **undoing** of a **significant** amount of **work**.

Note: Cascading rollback is undesirable

It is desirable to restrict the schedules -- where cascading rollbacks cannot occur.
schedules are -- cascadeless schedules.

Cascadeless Schedule

each pair of transactions **T_i** and **T_j** such that T_j reads a data item previously written by T_i,

the **commit** of **T_i** appears **before** the **read** of **T_j**.

Note: Every **cascadeless schedule** is also **recoverable schedule.**

Concurrency Control

One way to ensure serializability -- require that data items be accessed in a **mutually exclusive** manner

i.e. while **one** transaction is **accessing** a **data**, **no other** transaction can **modify** that data item.

method used to implement this -- **locking**.

Note: Lock requests are made to concurrency-control manager.

Transaction -- proceed only after **request** – **granted**

Concurrency Control -- Lock-Based Protocols

Lock-- a mechanism -- to **control concurrent access** to a data

Data items can be locked in **TWO** modes

1. **Exclusive** -- if a T_i has obtained an **exclusive-mode lock** on data item -- can be both **read** and **write**

   is requested using **lock-X**.

2. **Shared** -- if a T_i has obtained a **shared-mode** lock on data item -- T_i can **read** but **cannot write**

   is requested using **lock-S**.

Lock-compatibility Matrix

	S	X
S	true	false
X	false	false

A transaction -- be **granted** a **lock** on an item, if the requested lock is **compatible** with locks **already held** on the **item**

Let A and B be Two Accounts -- are accessed by T_1 and T_2

T_1 transfers 5000 from account B to account A

```
T1:     lock-X(B);
        read(B);
        B := B − 5000;
        write(B);
        unlock(B);
        lock-X(A);
        read(A);
        A := A + 5000;
        write(A);
        unlock(A).
```

T_2 displays the total amount in accounts A and B

```
T2:     lock-S(A);
        read(A);
        unlock(A);
        lock-S(B);
        read(B);
        unlock(B);
        display(A + B).
```

Assume the values of accounts A and B --10000 & 20000 respectively

If T_1 and T_2 are executed serially, i.e., either -- order T_1, T_2 or the order T_2, T_1

the value of **A+B** -- 30000

If these transactions i.e. T_1 and T_2 -- executed **concurrently,** the possibility of **schedule1** shown below

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
B := B – 5000		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display(A + B)	
lock-X(A)		
		grant-X(A, T_2)
read(A)		
A := A + 5000		
write(A)		
unlock(A)		

Schedule 1.

Here $\mathbf{T_2}$ displays 25000, -- is **incorrect**.

The reason -- T_1 unlocked data item B too early
T_2 -- saw an inconsistent state.

Suppose -- unlocking is delayed to the end of the transaction

T_3 corresponds to T_1 with unlocking delayed .

T_4 corresponds to T_2 with unlocking delayed .

T_3: lock-X(B);
read(B);
B := B − 5000;
write(B);
lock-X(A);
read(A);
A := A + 5000;
write(A);
unlock(B);
unlock(A).
	T_4: lock-S(A);
	read(A);
	lock-S(B);
	read(B);
	display(A + B)
	unlock(A);
	unlock(B);

Unfortunately, locking can lead to an **undesirable** situation

e.g. Consider the partial schedule

T_3	**T_4**
lock-X(B);	
read(B);	
B := B − 5000;	
write(B);	
	lock-S(A);
	read(A);
	lock-S(B);
lock-X(A);	

Neither T_3 nor T_4 can make progress

-- executing **lock-S**(B) causes **T_4** to **wait** for T_3 to release its lock on B,

while executing **lock-X**(A) causes **T_3** to **wait** for T_4 to release its lock on A

situation is called a **deadlock**

To handle a deadlock one of **T_3 or T_4** must be **rolled back**

as a result, its locks released

Granting of Locks -- When a transaction **requests a lock** on a data item in a **particular mode**,

no other transaction has a **lock** on the **same data** item in a conflicting mode, then

the **lock** -- be **granted**.

However, consider the following scenario.

Suppose T_2 -- a **S-mode** lock on a data item, and T_1 requests an **X-mode** lock on the same.

Clearly, **T_1** -- to **wait** for **T_2** to **release** the **S**-mode

Meantime **T_3** may request a **S-mode** lock on the same the lock request is **compatible** with T_2

T_3 may be **granted** the S-mode lock.

At this point **T_2** may **releas**e the lock,

but **still T_1** has to **wait** for T_3 to finish

But again there may be a new transaction **T_4** requests a S-mode lock on the same data item

and is granted the lock before T_3 releases it … so on

but **T_1 never gets** the **X-mode** lock and

may never make progress -- **starved.**

To avoid **starvation** by granting locks based on the following

the concurrency-control manager grants the lock provided that

1. There is no other transaction holding a lock on data item that conflicts with particular mode.
2. There is no other transaction that is **waiting** for a lock on data item, and that made its lock request **before T_i**.

Thus, a lock request will never get blocked by a lock request that is made later.

The Two-Phase Locking Protocol

Two-phase Locking Protocol -- ensures serializability

requires that each transaction issue **lock** and **unlock** requests in two phases:

1. Growing Phase -- A transaction may **obtain locks**,

but may **not release any lock**.

2. Shrinking Phase -- A transaction **may release locks**,

but may **not obtain any new locks**.

e.g. T_3 and T_4 are **two** phase. Where as T_1 and T_2 are **not two** phase

Two-phase locking does **not ensure freedom** from **deadlocks**

Observe -- T_3 and T_4 are two phase, but they are deadlocked (see the previous partial schedule)

Cascading roll-back is **possible** under two-phase locking.

e.g. consider the partial schedule

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

the failure of T_5 **after** the **read(A)** step of T_7 **leads** to cascading **rollback** of T_6 and T_7

To avoid this, follow a modified protocol called **strict two-phase locking**.

Here a transaction must hold all its **X-locks** till it **commits/aborts**.

Rigorous two-phase locking -- even stricter

here **all locks** are held till **commit/abort**.

In this protocol transactions can be serialized in the order in which they commit.

Note: Most db systems implement either strict or rigorous two-phase locking

Implementation of Locking

A **lock manager** --

implemented as a separate process to which transactions send lock and unlock requests

-- **replies** to a lock request by sending a **lock grant** messages or

a message asking the transaction to **roll back**, in case of a **deadlock**

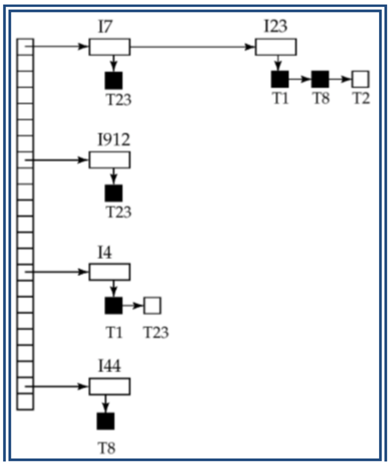
The requesting **transaction wait**s until its request is answered

– maintains a **data-structure**-- a **lock table**

Lock table -- record **granted locks** and **pending** requests

usually implemented as an **in-memory hash table** indexed on the name of the data item being locked

Lock table also records the type of lock granted or requested



Lock mgr processes reqts this way

New request is added to the end of the linked list of requests, otherwise it creates a new linked list

Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted

If transaction aborts, all waiting or granted requests of the transactions are deleted

lock mgr. may keep a list of locks held by each transactions, to implement this efficiently

Note: Omitted the lock mode to keep the figure simple

Graph-Based Protocols

2-PL -- both necessary and sufficient for ensuring serializability in the absence of information -- the manner in which data items are accessed

Graph-Based Protocols

-- are an **alternative** to 2PL that are **not two phase**

need **additional information** on how each transaction will access the db

various models -- give us the additional info

each differing in the amount of information provided

-- simplest model requires -- we have **prior knowledge** about the **order** in which the **db items** will be **accessed**

Given such information, it is possible to construct locking protocols that are not two phase

To acquire such prior information

-- impose a **partial ordering** $\rightarrow$ on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.

If $\mathbf{d_i} \rightarrow \mathbf{d_j}$ then any transaction accessing **both** $\mathbf{d_i}$ and $\mathbf{d_j}$ must access $\mathbf{d_i}$ **before** accessing $\mathbf{d_j}$

The partial ordering implies that the set **D** may now be viewed as a **directed acyclic** graph, called a **db graph**

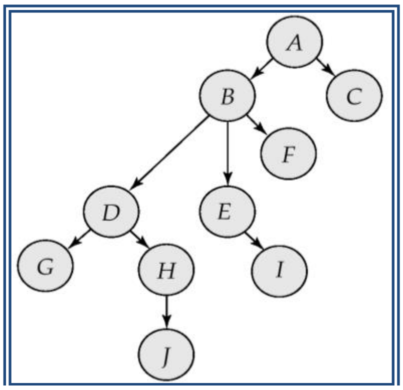
will present a simple protocol -- the **tree-protocol**

Tree Protocol -- Only **exclusive locks** are allowed

1. The **first lock** by $\mathbf{T_i}$ may be on **any** data item.
2. Subsequently, a data **D** can be **locked** by $\mathbf{T_i}$ only if the **parent** of **D** is currently locked by T_i.
3. Data items may be **unlocked** at **any time**.
4. A data item that has been **locked** and **unlocked by** $\mathbf{T_i}$ cannot subsequently be **relocked** by $\mathbf{T_i}$

All schedules that are legal under the tree protocol are conflict serializable

To illustrate this protocol, consider the db graph of Figure shown below



The following 4 transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

T10: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).

T11: lock-X(D); lock-X(H); unlock(D); unlock(H).

T12: lock-X(B); lock-X(E); unlock(E); unlock(B).

T13: lock-X(D); lock-X(H); unlock(D); unlock(H).

T10	T11	T12	T13
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
	l	ock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock(G)			

Fig. Serializable schedule under the tree protocol

The tree protocol in Fig(previous slide) **does not ensure** recoverability and cascadelessness

To ensure **recoverability** and **cascadelessness**,

protocol can be modified to **not** permit **release of locks** till the **end** of the transaction.

Holding X-locks until the end of the transaction **reduces concurrency**

an alternative that improves concurrency, but ensures only recoverability

Need to introduce **commit dependency**

For each data item with an **uncommitted write**

record which transaction performed the **last write** to the data item.

Whenever a transaction $\mathbf{T_i}$ **performs** a **read** of an **uncommitted data** item,

record a **commit dependency** of $\mathbf{T_i}$ on the **transaction** that performed the **last write** to the data item.

$\mathbf{T_i}$ is then **not permitted** to **commit** until the commit of **all transactions** on which it has a **commit dependency**.

If any of these transactions **aborts,** $\mathbf{T_i}$ must also be **aborted**.

Advantage over the two-phase locking

The tree protocol ensures **conflict serializability** as well as **freedom** from **deadlock**.

Unlocking may **occur earlier** in the tree-locking protocol than in the **2PL** protocol.

Shorter waiting times, and **increase** in **concurrency**

Protocol is **deadlock-free**,

No rollbacks are required

Disadvantage

a transaction may have to **lock data items** that it does **not access**.

e.g. a transaction that needs to access data items **A** and **J** in the **db graph** (previous. fig) must lock not only A and J, but also **B, D,** and **H**

increased locking overhead,

additional waiting time, and

a potential decrease in concurrency.

Timestamp-Based Protocols

Another method for determining the serializability order is to select an ordering among transactions in advance

Timestamps

Each transaction is **issued** a **timestamp** when it enters the system.

If an **old** transaction **T_i** has time-stamp **$TS(T_i)$**,
a **new** transaction **T_j** is assigned time-stamp **$TS(T_j)$**

such that $TS(T_i) < TS(T_j)$.

TWO simple methods for implementing this scheme

1. Use the value of the **system clock** as the timestamp
   i.e. a transaction's timestamp = to the value of the clock when the transaction enters the system.
2. Use a **logical counter --** is incremented after a new timestamp has been assigned
   i.e. a transaction's timestamp = the counter value when the transaction enters the system.

Transactions determine the serializability order.

Thus, if $TS(T_i) < TS(T_j)$.

then the system must ensure that the produced schedule is equivalent to a serial schedule in which T_i appears before T_j.

In order to assure such behavior, the protocol maintains for each data D two timestamp values

W-timestamp(D) -- the largest time-stamp of any transaction that executed **write**(D) successfully

R-timestamp(D) -- the largest time-stamp of any transaction that executed **read**(D) successfully

The Timestamp Ordering Protocol

-- ensures that any conflicting **read** and **write** operations are executed in timestamp order

1. Suppose a transaction T_i issues a **read**(D)

    a. If $TS(T_i) <$ **W**-timestamp(D),

    then T_i needs to read a value of D that was already overwritten.

    Hence, the read operation is rejected, and T_i is rolled back.

    b. If $TS(T_i) \geq$ **W**-timestamp(D),

    then the **read** operation is executed, and R-timestamp(D) is set to **max**(R-timestamp(D) and $TS(T_i)$)

2. Suppose a transaction T_i issues a **write**(D)

    a. If $TS(T_i) <$ **R**-timestamp(D),

    then the value of D that T_i is producing was needed previously, and the system assumed that value would never be produced.

    Hence, the system rejects the write & rolls T_i back.

    b. If $TS(T_i) <$ **W**-timestamp(D),

    then T_i is attempting to write an obsolete value of D.

    the system rejects this write & rolls T_i back.

    c. Otherwise, the system executes the write operation and

    sets W-timestamp(D) to $TS(T_i)$

-- ensures conflict serializability.

conflicting operations -- processed in timestamp order

-- ensures freedom from deadlock
no transaction ever waits.

However, there is a possibility of starvation

if a long trans sequence conflicting short transaction causes repeated restarting of the long transaction.

Note: schedules that are possible under the 2-PL locking protocol are not possible under the timestamp protocol, and vice versa.

The protocol -- generate schedules that are not recoverable

-- be extended to make the schedules recoverable, in one of several ways:

Recoverability and cascadelessness – be ensured by performing **all writes together** at the end.

The **writes** must be **atomic** in the following sense:

While the writes are in progress, no transaction is permitted to access any of the data items that have been written.

also be guaranteed by using a **limited form of locking**,

whereby reads of uncommitted items are postponed until the transaction that updated the item commits

Recoverability alone –
be ensured by tracking uncommitted writes,

and allowing a T_i to commit only after the commit of any transaction that wrote a value that T_i read.

Commit dependencies, can be used for this purpose.

Thomas' Write Rule

e.g. consider schedule 4 and apply the timestamp-ordering protocol

T_{16}	T_{17}
read(D)	
	write(D)
write(D)	

Schedule 4

Since T_{16} starts before T_{17}, assume that $TS(T_{16})<TS(T_{17})$

The read(D) operation of T_{16} succeeds,
as does the write(D) operation of T_{17}.

When T_{16} attempts its write(D) operation, observe that $TS(T_{16})$ < W-timestamp(D),

since W-timestamp(D) = $TS(T_{17})$.

Thus, the write(D) by T_{16} is rejected and transaction T_{16} must be rolled back

The modification to the timestamp-ordering protocol, called **Thomas' Write Rule**

2. Suppose a transaction T_i issues a **write**(D)

b. If $TS(T_i)$< **W**-timestamp(D),
then then T_i is attempting to write an obsolete value of D.

the write operation – be ignored.

i.e. Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored.

Thomas' Write Rule allows greater potential concurrency

Validation-Based Protocol

Execution of transaction T_i is done in **Three phases** in its lifetime, depending on whether it is a read-only or an update transaction.

1. **Read and execution phase**: Transaction T_i writes only to temporary local variables
2. **Validation phase**: Transaction T_i performs a "validation test" to determine if local variables can be written without violating serializability.
3. **Write phase**: If T_i is validated, the updates are applied to the db;

   if not, T_i is rolled back.

Each transaction T_i has 3 timestamps

Start(T_i) : the time when T_i started its execution

Validation(T_i) **:** the time when T_i entered its validation phase

Finish(T_i) : the time when T_i finished its write phase

Serializability order is determined by timestamp given at validation time, to increase concurrency.

therefore, $TS(T_i)$ is given the value of Validation(T_i)

This protocol is useful and gives greater degree of concurrency if probability of conflicts are low.

because the serializability order is not pre-decided, and relatively few transactions will have to be rolled back.

The validation test for transaction T_j requires that, for all T_i with $TS(T_i) < TS(T_j)$ either one of the following condition holds

finish(T_i) < **start**(T_j)

Since T_i completes its execution before T_j started, the serializability order is indeed maintained.

start(T_j) < **finish**(T_i) < **validation**(T_j) and the set of data items written by T_i does not intersect with the set of data items read by T_j.

then validation succeeds and T_j can be committed.

Otherwise, validation fails and T_j is aborted.

Justification:

Either the first condition is satisfied, and there is no overlapped execution, or

the second condition is satisfied and the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.

the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i

Example of schedule produced using validation

T_{14}	T_{15}
read(B)	
	read(B) B:= B-5000 **read**(A) A:= A+5000
read(A) (validate) **display** (A+B)	
	(validate) **write** (B) **write** (A)

$TS(T_{14}) < TS(T_{15})$

Then, the validation phase succeeds in the schedule 5

Note: the writes to the actual variables are performed only after the validation phase of T_{15}.

Hence, T_{14} reads the **old values of B and A**, and this schedule is serializable.

The validation scheme automatically guards against cascading rollbacks, since the actual writes take place only after the transaction issuing the write has committed.

yet, there is a possibility of **starvation**

long transactions, Vs short transactions -- cause repeated restarts of the long transaction.

To avoid starvation

conflicting transactions -- be **temporarily blocked**, to enable the long transaction to finish.

This validation scheme is called the **optimistic concurrency control scheme**

Since transactions execute optimistically, assuming they will be able to finish execution and validate at the end.

Multiple Granularity

allow data items of **various sizes** (grouping several data items) and define a hierarchy of **data granularities**

where the small granularities are nested within larger ones

advantage -- to treat them as one individual **synchronization unit**

e.g. if a T_i needs to access the entire db – locking is used then T_i must lock each item in the db.

executing these locks is time consuming.

would be better if T_i could issue a single lock request to lock the db

If T_i needs to access only a few data items -- not to lock the entire db

otherwise **concurrency** is **lost**.

needed a mechanism to allow the system to define multiple levels of **granularity**

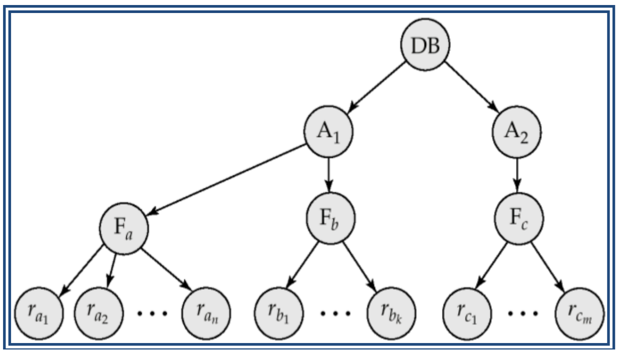
-- can be represented graphically as a **tree** (but don't confuse with tree-locking protocol)

See the figure below -- consists of **4 levels** of nodes.

highest level represents the entire db

The levels, starting from the top are

Db
Area
File
Record



Each node in the tree can be locked individually (similar to 2-phase locking protocol)

When a transaction **locks a node** in the **tree** explicitly, it **implicitly locks all** the **node's descendents** in the same mode

e.g. if trans T_i gets a lock on F_c (Fig) in X-mode

then it has an implicit lock in X-mode all the records belonging to that file.

No need to lock the individual records of F_c explicitly

e.g. T_j wishes to lock record r_{b6} of file F_b.

Since T_i has locked F_b explicitly, it follows that r_{b6} is also locked (implicitly)

But, when T_j issues a lock request for r_{b6},
r_{b6} is not explicitly locked

In addition to S and X lock modes, -- **3** additional lock modes with multiple granularity

intention-shared (IS):

indicates explicit locking at a lower level of the tree but only with **shared** locks.

intention-exclusive (IX):

indicates locking at a lower level with **exclusive** or **shared** locks

shared and intention-exclusive (SIX):

the sub-tree rooted by that node is locked in **shared** mode and explicit locking is being done at a **lower level with exclusive-**mode locks.

intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Multiple Granularity Locking Scheme

Transaction T_i can lock a node D, using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node D can be locked by T_i in S or IS mode only if the parent of D is currently locked by T_i in either IX or IS mode.
4. A node D can be locked by T_i in X, SIX, or IX mode only if the parent of D is currently locked by T_i in either IX or SIX mode.
5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
6. T_i can unlock a node D only if none of the children of D are currently locked by T_i.

Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

Deadlock Handling

if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set.

set of waiting transactions { $T_0, T_1, \ldots, T_n$} Э

T_0 is waiting for a data item that T_1 holds, and

T_1 is waiting for a data item that T_2 holds, and

… ,

T_{n-1} is waiting for a data item that T_n holds, and

T_n is waiting for a data item that T_0 holds.

None of the transactions can make progress in such a situation.

sol. rolling back some of the transactions involved in the deadlock.

e.g., Consider the following two transactions:

T_1: write (X) T_2: write(Y)

write(Y) write(X)

Schedule with deadlock

2 Main Methods for dealing -- the deadlock problem

1. Use a **deadlock prevention protocol** to ensure that the system will **never enter** a **deadlock** state.
2. Allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme

Note: **detection** and **recovery** scheme requires **overhead** that includes not only the **run-time cost** but also the **potential losses inherent** in **recovery** from a deadlock

Deadlock Handling - Deadlock Prevention

Deadlock Prevention -- 2 approaches

1. Ensures that **no cyclic waits** can occur or requiring locks to be acquired together
2. is closer to deadlock recovery, and performs transaction **rollback instead** of **waiting** for a lock

1st approach -- requires that each transaction locks all its data items before it begins execution.

Moreover, either **all** are **locked** in **one step** or none are locked.

Disadvantages

1. it is often **hard to predict**, before the transaction begins, what data items need to be locked
2. data-item **utilization** may be **very low** since many of the data items may be locked but **unused** for a **long time**

Another approach for preventing deadlocks

Impose **partial ordering** of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

2nd approach -- use **preemption** and transaction rollbacks

In preemption,

When **T_2** requests a lock that **T_1 holds**

the lock granted to T_1 may be **preempted** by rolling back of T_1, and granting of the lock to T_2.

To control the preemption,

We **assign** a **unique timestamp** to each transaction.

these timestamps -- used only to decide whether a transaction should **wait or roll back**.

Note: If a transaction is rolled back, it **retains** its **old timestamp** when restarted

Two different **deadlock prevention** schemes using **timestamps** have been proposed

1. **Wait-die** scheme -- **non-preemptive**

When **T_i requests** a data item currently **held** by **T_j**

T_i is allowed to **wait** only if it has a timestamp **smaller than T_j**

i.e., T_i is **older** than T_j

Otherwise, **T_i** is **rolled back** (**dies**)

a transaction may **die several** times before acquiring needed data item

Note: Whenever the system rolls back transactions, it is important to ensure that there is **no starvation**

e.g. Suppose T_a, T_b, and T_c have timestamps 10, 14, and 17, respectively.

If T_a requests a data item held by T_b

then T_a -- **wait**.

If T_c requests a data item held by T_b

then T_c will be **rolled back**

Older transaction may **wait** for **younger one** to release data item.

Younger transactions **never wait** for **older** ones are **rolled back** instead

2. **Wound–wait** scheme -- a **pre-emptive** technique.

When T_i requests a data item currently held by T_j,

T_i is allowed to **wait** only if it has a **timestamp larger than** that of T_j

i. e., T_i is **younger** than T_j

Otherwise, T_j is **rolled back** (T_j is **wounded** by T_i)

e.g. If **T_a** requests a data item **held** by **T_b**

data item will be pre-empted from $\mathbf{T_b}$ and

$\mathbf{T_b}$ will be **rolled back**

If $\mathbf{T_c}$ requests a data item **held** by $\mathbf{T_b}$

then T_c – **wait**

older transaction **wounds** (forces rollback) of younger transaction instead of waiting for it.

Younger transactions may **wait** for **older** ones.

may be **fewer rollbacks** than **wait-die** scheme.

Disadvantage

with both of these schemes is that unnecessary rollbacks may occur

Both the schemes **avoid starvation**

At any time, there is a transaction with the smallest timestamp and cannot be required to roll back in either scheme.

Since **timestamps always increase**

Deadlock Handling - Timeout-Based Schemes

simple approach to deadlock handling is based on **lock timeouts**

a transaction -- has requested a lock **waits** for at most a **specified amount** of **time**.

If the lock has **not** been **granted** within that time & is said to time out.

it rolls itself back and restarts

easy to implement, & works well if transactions are short

Too **long** a **wait** results in **unnecessary delays** once a deadlock has occurred

Too **short** a **wait** results in transaction **rollback** even when there is no deadlock,

leads to **wastage** of **resources**.

Starvation is also a possibility with this scheme

Therefore, the timeout-based scheme has limited applicability

Deadlock Handling - Deadlock Detection & Recovery

To examine the state of the system

An algorithm is invoked periodically to determine whether a deadlock has occurred

If so, the system must attempt to recover from the deadlock. Then, the system must:

1. Maintain information of the **current allocation** of **data items** to transactions

   also any **outstanding** data item **requests**

2. Provide an algorithm -- uses this information to determine whether the **system** has **entered a deadlock** state
3. **Recover** from the deadlock when the detection algorithm determines that a **deadlock exists**.

Deadlocks can be described as a **wait-for** graph, which consists of a pair G = (V,E),

V -- a set of **vertices** (all the transactions in the system)
E -- a set of **edges**;

each element is an **ordered pair $T_i \rightarrow T_j$**

If $T_i \rightarrow T_j$ is in E, then -- is a **directed edge** from T_i to T_j,

i.e., **T_i** is **waiting** for **T_j** to release a data item.

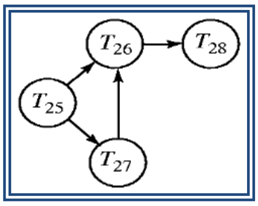
When **T_i** requests a data item currently being **held** by **T_j,**

then the **edge $T_i \rightarrow T_j$** is **inserted** in the **wait-for** graph.

This edge is removed only when T_j is no longer holding a data item needed by T_i.

To understand, consider the **wait-for graph** in Fig -- shows the following situation:

Transaction T_{25} is waiting for transactions T_{26} and T_{27}.
Transaction T_{27} is waiting for transaction T_{26}.
Transaction T_{26} is waiting for transaction T_{28}.



Since the graph has **no cycle**, the system is **not** in a **deadlock** state

Suppose T_{28} is requesting an item held by T_{27}.

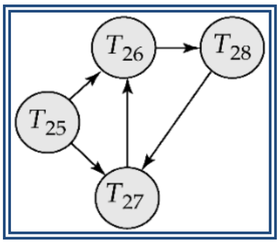
The edge $T_{28} \rightarrow T_{27}$ is added to the wait-for graph,

resulting in the new system state (see the below figure).

it leads -- the graph contains the cycle

$T_{26} \rightarrow T_{28} \rightarrow T_{27} \rightarrow T_{26}$

implying that transactions **T_{26}, T_{27},** and **T_{28}** are all **deadlocked.**



When deadlock is detected :

Some transaction -- have to rolled back (made a **victim**) to **break deadlock**.

Select that transaction as **victim** that will incur minimum cost.

Rollback -- determine **how far to roll back** transaction

Total rollback: **Abort the transaction** and then restart it.

More effective to roll back transaction only as far as necessary to break deadlock.

Starvation happens if same transaction is always chosen as victim.

Include the number of rollbacks in the cost factor to avoid starvation

Recovery System

Failure Classification

Transaction failure -- 2 types of errors -- a transaction to fail.

Logical errors:

transaction cannot complete due to some internal error condition

e.g., bad i/p, data not found, overflow, or resource limit exceeded

System errors:

the db system must terminate an active transaction due to an error condition

e.g., deadlock

System crash:

is a **HW malfunction**, or a **bug** in the db SW or the **OS**

causes the **loss** of the **content** of **volatile** storage

therefore, **halts** transaction process.

The content of **nonvolatile** storage remains **intact**, and is not corrupted

Fail-stop assumption:

assumption that HW errors and bugs in the SW bring the system to a halt, but do not corrupt the nonvolatile storage contents

Disk failure:

either a **head crash** or **failure** during a data transfer operation.

disk block loses its content

Copies of the data on other disks, or archival backups on tertiary media are used to recover

e.g., tapes,

To recover the from failures

1st i**dentify** the **failure** modes of those devices used for **storing** data.

2nd **how** these **failure** modes **affect** the contents of the db

then propose algorithms

Recovery algorithms

to ensure **db consistency** and transaction **atomicity** and **durability** despite failures

-- comprises **two parts**

Actions taken during **normal transaction** processing to ensure **enough** information **exists** to **recover**

Actions taken **after** a **failure** to recover the db contents to a state that ensures **atomicity,** and **durability**

Storage Structure

Volatile storage:

does **not** usually **survive** system **crashes**

e.g., main memory, cache memory

Nonvolatile storage:

survives system crashes

e.g., disk, tape

Stable storage:

a mythical form of storage that **survives all failures**

approximated by maintaining multiple copies on distinct nonvolatile media

Data Access

Physical blocks
those blocks residing on the disk.
Buffer blocks
the blocks residing temporarily in main memory.
Disk buffer
The area of memory where blocks reside temporarily

Block movements between **disk** and **main memory** are initiated through the following **two** operations:

input(X) transfers the **physical block X** to main memory.

output(X) transfers the **buffer block X** to the **disk**, and replaces the appropriate physical block there

Recovery and Atomicity

Modifying the **db** without ensuring that the transaction's **commit** may leave the db in an **inconsistent** state.

Consider $\mathbf{T_i}$ that transfers 5000 from account A to account B;
goal -- either to perform **all db modifications** made by $\mathbf{T_i}$ or **none** at all.

To ensure **atomicity** despite failures,
1st **o/p** information describing the modifications to **stable storage** without modifying the **db itself**.

Two approaches to perform such o/ps:
log-based recovery

shadow-paging

Assume (initially) that transactions run serially

i.e., one after the other.
most widely used structure for recording db modifications is the **log**

Log-Based Recovery

Other special log records exist to record significant events during transaction processing,
e.g., the **start** and **commit** or **abort**

When transaction T_i starts, it registers itself by writing a

<T_i **start**>log record

Before T_i executes **write**(X), a log record <$\mathbf{T_i}$**, X, $\mathbf{V_1}$, $\mathbf{V_2}$**> is written,

$\mathbf{V_1}$ **--** the value of **X** **before** the **write**, and

$\mathbf{V_2}$ -- the value **to be written** to X.

When $\mathbf{T_i}$ **finishes** it last statement, the log record

<T_i **commi**t> is written.

When $\mathbf{T_i}$ **aborts**, the log record

< T_i **abort**>. Transaction T_i has aborted.

We assume for now that log records are written directly to stable storage (i.e., they are not buffered)

Two approaches using logs

Deferred db modification

Immediate db modification

Deferred Db Modification

scheme records **all modifications** to the **log**, but defers all the **write**s to after **partial commit**.

Assume that transactions execute serially

The **execution** of **T_i proceeds** as follows

Transaction starts by writing <T_i **start**> record to **log**.

A **write**(X) operation results in a log record

<T_i, X, V> being written,

where V -- the **new value** for X

The **write** is **not performed** on X at this time, but is **deferred**.

When **T_i** partially commits, <T_i **commit**> is written to the **log**

Finally, the **log records** are read and used to actually execute the previously deferred writes.

Note: **old value** is **not needed** for this scheme

During **recovery after** a **crash**,

a transaction needs to be **redone** iff both <T_i **start**> and <T_i **commit**> are there in the **log**.

Redoing a T_i (**redo**T_i) sets the value of all data items updated by the transaction to the **new values**

Crashes can occur while

the transaction is executing the original updates, or while recovery action is being taken

e.g., transactions T_0 and T_1 (T_0 executes before T_1):

T_0: **read** (A)
A: = A - 5000
Write (A)
read (B)
B:- B + 5000
write (B)

T_1 : **read** (C)
C:= C - 1000
write (C)

Below we show the log as it appears at three instances of time

< T_0 start>	< T_0 start>	< T_0 start>
< T_0 , A, 5000 >	< T_0 , A, 5000>	< T_0 , A, 5000 >
< T_0 , B, 25000>	< T_0 , B, 25000>	< T_0 , B, 25000>
	< T_0 commit>	< T_0 commit>
	< T_1 start>	< T_1 start>
	< T_1 , C, 6000>	< T_1 , C, 6000>
		< T_1 commit>
(a)	(b)	(c)

If log on stable storage at the time of crash is as in case:

(a) **No redo** actions need to be taken

(b) **redo**(T_0) must be performed since <T_0 **commit**> is present

(c) **redo**(T_0) must be performed followed by **redo(T_1)** since

<T_0 **commit**> and <T_1 **commit**> are present

Note: The redo operation must be idempotent;

i.e., executing it **several times** must be **equivalent** to **executing it once**

Immediate Db Modification

scheme allows db updates of an **uncommitted** transaction to be made as the **writes** are **issued**

since **undoing** may be needed, update **logs** must have both **old value** and **new value**

Update log record -- be written **before db item** is written

We assume that the **log** record is **o/p** (output) **directly** to stable storage

Output of **updated** blocks can take place at **any time before** or **after** transaction **commit**

Order in which blocks are output can be different from the order in which they are written.

Log	**Write**	**Output**
<T_0 **start**>		
<T_0, A, 10000, 5000>		
<T_0, B, 20000, 25000>		
	A = 5000	
	B = 25000	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 7000, 6000>		
	C = 6000	
		B_B, B_C
<T_1 **commit**>		
		B_A

Note: B_X denotes block containing X.

Recovery procedure has two operations instead of one:

undo(T_i) **restores** the value of **all data items** updated by **T_i** to their **old** values, going backwards from the **last log** record for T_i

Redo(T_i) sets the **value** of **all data** items **updated** by **T_i** to the **new** values, going forward from the first log record for T_i

Both operations must be **idempotent** -- ensure correct behavior even if a failure occurs during the recovery process

i.e., even if the operation is **executed multiple times** the **effect** is the **same** as if it is executed once

When recovering after failure:

T_i needs to be **undone** if the log contains the record

<T_i **start**>, but does not contain the record <T_i **commit**>

T_i needs to be **redone** if the log contains both the record

<T_i **start**> and the record <T_i **commit**>

Undo operations are performed **first**, then **redo** operations.

e.g., with T_0 and T_1 **executed one after** the **other** say, T_0 followed by T_1.

Suppose that the **system crashes before** the completion of the transactions.

We shall consider **three** cases.

```
<T0 start>
<T0 , A, 10000, 5000>
<T0 , B, 20000, 25000>
```

(a)

```
<T0 start>
<T0 , A, 10000, 5000>
<T0 , B, 20000, 25000>
<T0 commit>
<T1 start>
<T1 , C, 7000, 6000>
```

(b)

```
<T0 start>
<T0 , A, 10000, 5000>
<T0 , B, 20000, 25000>
<T0 commit>
<T1 start>
<T1 , C, 7000, 6000>
<T1 commit>
```

(c)

Recovery actions in each case above are:

(a) **undo** (T_0): B is restored to 20000 and A to 10000.

(b) **redo** (T_0) and **undo** (T_1) : C is restored to 7000, and then A and B are set to 5000 and 25000 respectively.

(c) **redo** (T_0) and redo (T_1): A and B are set to 5000 and 25000 respectively. Then C is set to 6000

Checkpoints

Problems in recovery procedure as discussed earlier :

1. searching the **entire log** is **time-consuming**
2. we might **unnecessarily redo** transactions which have already
3. output their updates to the db.

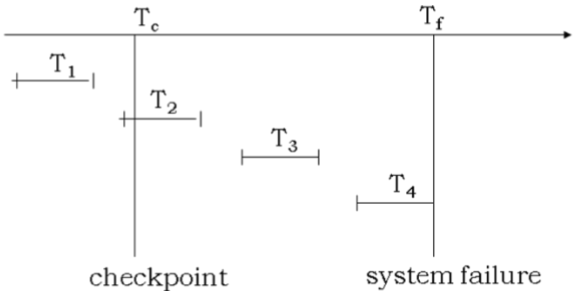
Streamline recovery procedure by periodically performing **checkpointing**

1. Output **all log** records currently residing in **main memory** onto stable storage.
2. Output **all modified buffe**r blocks to the **disk**.
3. Write a log record < **checkpoint**> onto stable storage

During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.

1. Scan backwards from end of log to find the most recent <**checkpoint**> record
2. Continue scanning backwards till a record <T_i **start**> is found.
3. Need only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
4. For all transactions (starting from T_i or later) with no <T_i **commit**>, execute **undo(**T_i). (Done only in case of immediate modification.)
5. Scanning forward in the log, for all transactions starting from T_i or later with a <T_i **commit**>, execute **redo(**T_i).

Example of Checkpoints



T_1 can be ignored (updates already output to disk due to checkpoint)

T_2 and T_3 redone.

T_4 undone

Recovery With Concurrent Transactions

We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.

All transactions share a **single disk buffer** and a **single log**

A **buffer block** can have **data items** updated by **one** or **more** transactions

We assume concurrency control using **strict 2PL**;

i.e., the updates of uncommitted transactions should not be visible to other transactions

Otherwise how to perform undo if T_1 updates A, then T_2 updates A and commits, and finally T_1 has to abort?

Transaction Rollback

Logging is done as described earlier.

roll back a failed transaction, T_i, by using the **log**. The system scans the log backward;

for every log record of the form <$\mathbf{T_i, X_j, V_1, V_2}$> found in the log, the system restores the data item $\mathbf{X_j}$ to its old value $\mathbf{V_1}$.

Log records of different transactions may be interspersed in the log.

Checkpoints

The **checkpointing** technique and **actions taken** on recovery have to be **changed** since **several** transactions may be **active** when a checkpoint is performed.

-- are performed as before, except that the checkpoint log record is now of the form

< **checkpoint L**>

L -- list of transactions active at the time of the checkpoint

We assume **no updates** are in progress while the checkpoint is carried out (will relax this later)

Restart Recovery

When the system recovers from a crash, it first does the following:

Initialize **undo-list** and **redo-list** to empty

Scan the **log backwards** from the end,

stop when the first <**checkpoint** L> record is found.

For **each record** found during the **backward** scan

if the record is <T_i **commit**>, **add** T_i to **redo-list**

if the record is <T_i **start**>, if T_i is **not** in **redo-list**, then **add** T_i to **undo-list**

For every T_i in **L**, if T_i is **not** in **redo-list,** then add T_i to **undo-list**

At this point

undo-list consists of **incomplete** transactions which must be **undone**, and

redo-list consists of **finished** transactions that must be **redone**.

Once the **redo-lis**t and **undo-list** have been made,

Recovery now continues as follows:

Scan log backwards from **most recent** record,

stop when

<T_i **start**> records have been encountered for every T_i in **undo-list.**

During the scan, perform **undo** for **each log** record that belongs to a transaction in **undo-list**.

Locate the most recent <**checkpoint** ***L***> record.

Scan log forwards from the <**checkpoint L**> record till the **end** of the log.

During the scan, perform **redo** for **each log** record that belongs to a transaction on **redo-list**

Buffer Management - Log Record Buffering

Log record buffering:

log records are **buffered** in **main memory**, instead of being **o/p** directly to **stable storage**.

Log records are **o/p** to **stable storage** only when the **buffer** is **full**, or

a **log force** operation is **executed**.

Log force is performed to **commit** a transaction by forcing all its log records (including the commit record) to **stable storage.**

Thus, several log records -- be **o/p** using a **single o/p** operation**, reducing** the **I/O** cost.

The rules below must be followed if log records are buffered:

Log records are **o/p** to **stable storage** in the order in which they are created.

Transaction T_i enters the **commit** state only when the log record

<T_i **commit**> has been o/p to stable storage.

Before a **data block** in **main memory** is **o/p** to the **db**,

all log records pertaining to data in that block must have been **o/p** to **stable storage**.

This rule -- the **write-ahead logging** or **WAL** rule

WAL only requires **undo** information to be **o/p**

Database Buffering

The system **stores** the **db** in nonvolatile storage (**disk**), and brings data blocks into main memory(**MM**) as needed

Since MM is typically much **smaller** than the entire db

Db maintains an in-memory buffer of data blocks

When a **new block** is **needed**,

if **buffer** is **full** an **existing** block needs to be **removed from buffer**

If the block **chosen** for **removal** has been updated, it must be **o/p to disk**

If an **uncommitted updates** block is **o/p** to disk, log records with **undo** information for the updates are **o/p** to the log on **stable storage first** (WAL)

e.g., If the **i/p** of block B_2 causes block B_1 to be chosen for **o/p**, all log records pertaining to data in B_1 must be **o/p** to **stable storage** before B_1 is o/p.

Thus, actions by the system would be:

O/p log records to **stable storage** until all log records pertaining to block B_1 have been **o/p**.

O/p block B_1 to disk.

I/p block B_2 from disk to main memory.

No updates should be in progress on a block when it is output to disk. Can be ensured as follows.

Before writing a data item, transaction acquires exclusive lock on block containing the data item

Lock can be released once the write is completed.

Such locks held for short duration are called **latches**.

Before a block is o/p to disk, the system acquires an exclusive latch on the block

Ensures no update can be in progress on the block

To illustrate the need for the write-ahead logging requirement,
e.g., with transactions T_0 and T_1.

Suppose that the state of the log is

<T_0 start>
<T_0, A, 10000, 5000> and T_0 issues a read(B)

Assume that the block on which B resides is not in MM, and that MM is full.

Suppose that the block on which A resides is chosen to be o/p to disk.

If the system o/ps this block to disk and then a crash occurs,

the values in the db for accounts A, B, and C are Rs. 5000, 20000, and 7000, respectively.

This db state is inconsistent.

However, with the WAL requirements,

the **log record** <T_0, A, 10000, 5000> must be **o/p** to **stable storage** prior to o/p of the block on which A resides.

uses this log record during **recovery** to bring the db back to a consistent state.

Failure with Loss of Nonvolatile Storage

So far we assumed no loss of non-volatile storage

Technique similar to checkpointing used to deal with loss of non-volatile storage

Periodically **dump** the entire content of the db to **stable storage**

No transaction may be **active** during the **dump** procedure;

a procedure similar to checkpointing must take place

O/p all log records **currently** residing in **MM** onto **stable storage**.

O/p all buffer blocks onto the **disk**.

Copy the contents of the db to **stable storage**.

O/p a record <**dump**> to log on stable storage.

To recover from disk failure

restore db from most **recent dump**.

Consult the **log** and **redo all** transactions that committed **after** the **dump**

Can be extended to allow transactions to be active during dump;

known as fuzzy dump or online dump (see fuzzy checkpointing later)

ARIES

ARIES is a state of the art recovery method

Incorporates numerous optimizations

to **reduce overhead**s during normal processing and
to **speed up recovery**

uses a number of techniques

to reduce the **time taken for recovery**, and
to reduce the **overheads** of **checkpointing**.

Unlike other advanced recovery algorithm, ARIES uses several data structures

1. Uses **log sequence number (LSN)** to identify log records

   Use of LSNs -- to identify what updates have already been applied to a db page

2. Supports **Physiological redo**

   the affected page is physically identified, but can be logical within the page.

3. Uses **Dirty page table**

   to avoid unnecessary redos during recovery

4. Uses **Fuzzy checkpointing**

   records information about dirty pages, and does not require dirty pages to be written out to disk

Data Structures

Log sequence number (LSN) -- uniquely **identifie**s each **log** record

is generated in such a way that it can also be used to **locate** the **log record** on disk

Usually, ARIES splits a **log** into **multiple** log files, each -- a **file number**.

As a log file **grows** to some **limit**,

ARIES **adds** more **log** records to a **new log file**;

the **new log** file has a **file number** that is **higher** by **ONE** than the previous log file.

PageLSN -- an identifier maintained by each page.

which is the LSN of the last log record whose effects are reflected on the page

To update a page:

X-latch the page, and write the log record

Update the page

Record the **LSN** of the **log** record in PageLSN

Unlock page

To flush page to disk, must first **S-latch** page

PageLSN -- used during **recovery** to prevent **repeated redo**

Thus ensuring idempotence

Physiological redo

Affected page is physically identified, action within page can be logical

Used to **reduce logging** overheads

e.g. when a record is **deleted** and all other records have to be moved to **fill hole**

Physiological **redo** can **log** just the record deletion

Physical **redo** would require **logging** of **old** and **new** values for much of the page

Requires page to be o/p to disk atomically

Easy to achieve with HW RAID, also supported by some disk systems

Log Record

Each log record contains LSN of previous log record of the same transaction

Special redo-only log record called **compensation log record (CLR)**

used to **log actions** taken during **recovery** that never need to be **undone**

Serves the role of **operation-abort** log records used in advanced recovery algorithm

Has a field UndoNextLSN to note next (earlier) record to be undone

Records in between would have already been **undone**

Required to **avoid repeated undo** of already undone actions

Dirty Page Table

List of pages in the buffer that have been updated

Contains, for each such page

PageLSN of the page

RecLSN is an LSN such that log records before this LSN have already been applied to the page version on disk

Set to current end of log when a page is inserted into dirty page table (just before being updated)

Recorded in checkpoints, helps to minimize redo work

Checkpoint log record Contains:

Dirty Page Table and **active** transactions **list**

For each active transaction, LastLSN, the LSN of the last log record written by the transaction

Fixed position on disk notes LSN of last completed checkpoint log record

Dirty pages are **not written** out **at checkpoint time**

Instead, they are **flushed out continuously**, in the background

Checkpoint is thus **very low overhead**

can be done frequently

ARIES Recovery Algorithm -- involves **Three** passes

Analysis pass: Determines

Which transactions to **undo**

Which pages were **dirty** (disk version not up to date) at time of crash

RedoLSN: LSN from which redo should start

Redo pass:

Repeats history, redoing all actions from RedoLSN

RecLSN and PageLSNs are used to avoid redoing actions already reflected on page

Undo pass:

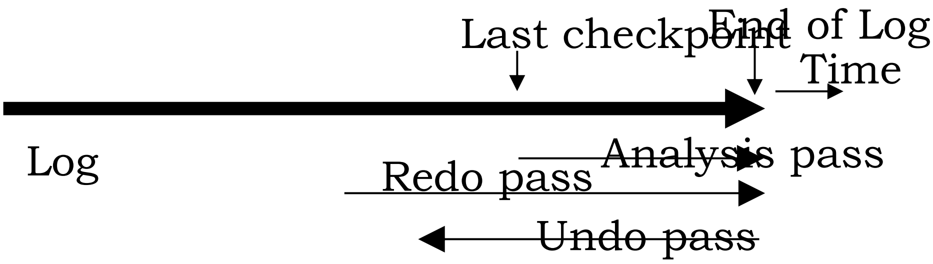
Rolls back **all incomplete** transactions

Transactions whose abort was complete earlier are not undone

Key idea: no need to **undo** these transactions: earlier undo actions were logged, and are **redone** as required

Analysis determines where redo should start

Undo has to go back till start of earliest incomplete transaction



ARIES Recovery Algorithm - Features

Recovery Independence

Pages can be recovered independently of others

e.g., if some disk pages fail they can be recovered from a backup while other pages are being used

Savepoints:

Transactions can record savepoints and roll back to a savepoint

Useful for complex transactions

Also used to rollback just enough to release locks on deadl

Fine-grained locking:

Index concurrency algorithms that permit tuple level locking on indices can be used

These require logical undo, rather than physical undo, as in advanced recovery algorithm

Recovery optimizations:

For example:

Dirty page table can be used to prefetch pages during redo

Out of order redo is possible:

redo can be postponed on a page being fetched from disk, and

performed when page is fetched.

Meanwhile other log records can continue to be process