By Xu Lizhen
School of Computer Science and Engineering, Southeast University
Nanjing, China

The Preliminary Courses are:

Data Structure
Database Principles
Database Design and Application

The students should already have the basic concepts about database system, such as data model, data schema, SQL, DBMS, transaction, database design, etc.

Now we will introduce the implementation techniques of Database Management Systems.

The goal is to
and to
through the study of this course.

Introduce the inner implementation technique of every kind of DBMS, including the architecture of DBMS, the support to data model and the implementation of DBMS core, user interface, etc. The emphasis is the basic concepts, the basic principles and the

The history, classification, and main research contents of database systems; Distributed database system

The composition of DBMS and its process structure; The architecture of distributed database systems

Physical file organization, index, and access primitives

The fragmentation and distribution of data, distributed database design, federated database design, parallel database design, data catalog and its distribution

Basic problems; Query optimization techniques; Query optimization in distributed database systems; Query optimization in other kinds of DBMS

Basic problems; Updating strategies and recovery techniques; Recovery mechanism in distributed DBMS

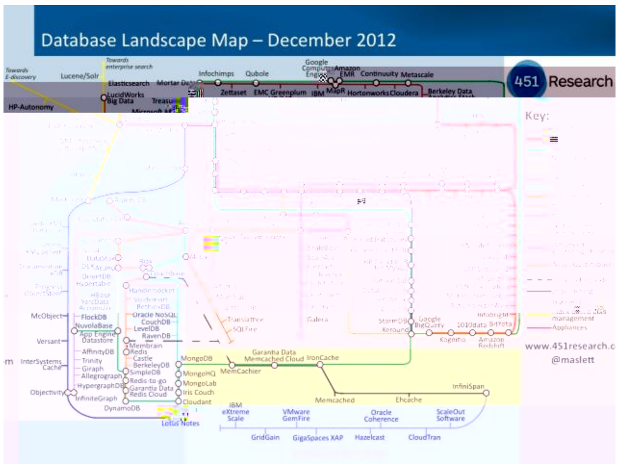
Basic problems; Concurrency control techniques; Concurrency control in distributed DBMS; Concurrency control in other kinds of DBMS

(1) According to the development of data model
- No management(before 1960'): Scientific computing
- File system: Simple data management
- Demand of data management growing continuously, DBMS emerged.
  - 1964, the first DBMS (American): IDS, network
  - 1969, the first commercial DBMS of IBM, hierarchical
  - 1970, E.F.Codd(IBM) bring forward relational data model
  - Other data model: Object Oriented, deductive, ER, ...

(2) According to the development of DBMS architectures
- Centralized database systems
- Parallel database systems
- Distributed database systems (and Federated database systems)
- Mobile database systems

(3) According to the development of architectures of application systems based on databases
- Centralized structure : Host + Terminal
- Distributed structure
- Client / Server structure
- Three tier / multi-tier structure
- Mobile computing
- Grid computing (Data Grid), Cloud Computing

(4) According to the expanding of application fields
- OLTP
- Engineering Database
- Deductive Database
- Multimedia Database
- Temporal Database
- Spatial Database
- Data Warehouse, OLAP, Data Mining
- XML Database
- Big Data, NoSQL, NewSQL



What is DDB?

A DDB is a collection of correlated data which are spread across a network and managed by a software called DDBMS.

Two kinds:

(1) Distributed physically, centralized logically (general DDB)
(2) Distributed physically, distributed logically too (FDBS)

We take the first as main topic in this course.

Distribution

Correlation

DDBMS

Local autonomy

Good availability (because support multi copies)

Good flexibility

Low system cost

High efficiency (most access processed locally, less communication comparing to centralized database system)

Parallel process

Hard to integrate existing databases

Too complex (system itself and its using, maintenance, etc. such as DDB design)

Compared to centralized DBMS, the differences of DDBS are as follows:

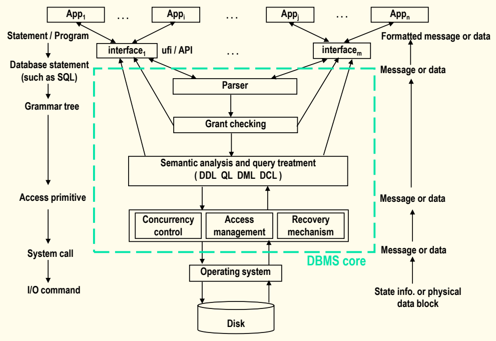
Query Optimization (different optimizing goal)

Concurrency control (should consider whole network)

Recovery mechanism (failure combination)

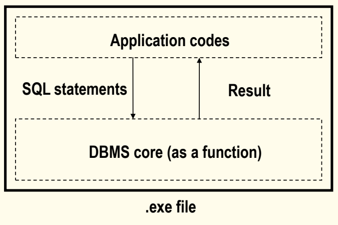
Data distribution

The components of DBMS core
The process structure of DBMS
The components of DDBMS core
The process structure of DDBMS

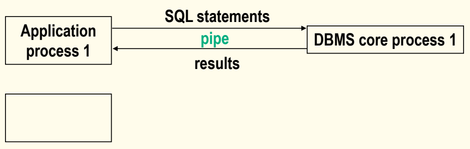


Single process structure
Multi processes structure
Multi threads structure
Communication protocols between processes / threads

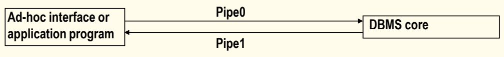
The application program is compiled with DBMS core as a single .exe file, running as a single process.



One application process corresponding to one DBMS core process



Application programs access databases through API or embedded SQL offered by DBMS, according to communication protocol to realize synchronizing control:

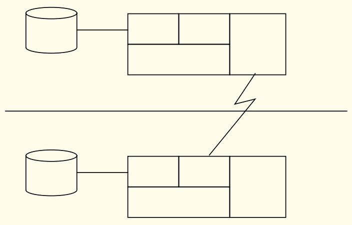


Pipe0: Send SQL statements, inner commands;
Pipe1: return results. The result format:

State	TupNum	AttNum	AttName	AttType	AttLen	……	TmpFileName

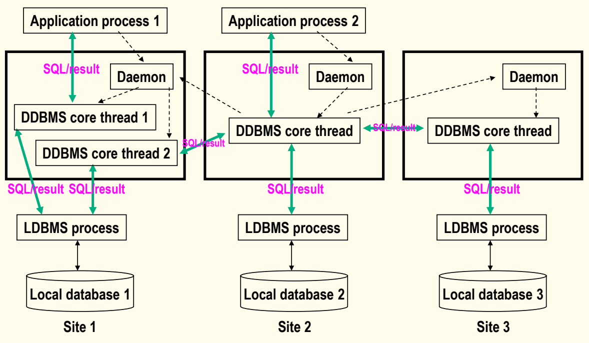
Definition of one attribute Definition of other attributes

State: 0 -- error, 1 -- success for insert,delete,update,
2 -- query success, need to treat result further.
TupNum: tuple number in result.
AttNum: attribute number in result table.
AttName: attribute name.
AttType: attribute type.
AttLen: byte number of this attribute.
TmpFileName: name of the temporary file which store the result data, need the above metadata to explain it.



Select *
From R1,R2
Where R1.a = R2.b;

Global query optimization may get an execution plan based on cost estimation, such as:
(1)send R2 to site1, R'
(2)execute on site1:
Select *
From R1, R'
Where R1.a = R'.b;



The access to database is transferred to the operations on files (of OS) eventually. The file structure and access route offered on it will affect the speed of data access directly. It is impossible that one kind of file structure will be effective for all kinds of data access

- Access types
- File organization
- Index technique
- Access primitives

- Query all or most records of a file (>15%)
- Query some special record
- Query some records (<15%)
- Scope query
- Update

- Heap file: records stored according to their inserted order, and retrieved sequentially. This is the most basic and general form of file organization.
- Direct file: the record address is mapped through hash function according to some attribute's value.
- Indexed file: index + heap file / cluster
- Dynamic hashing: p115
- Grid structure file: p118 (suitable for multi attributes queries)
- Raw disk (notice the difference between the logical block and physical block of file. You can control physical blocks in OS by using raw disk)

- B+ Tree (√ √)
- Clustering index (√)
- Inverted file
- Dynamic hashing
- Grid structure file and partitioned hash function
- Bitmap index (used in data warehouse)
- Others

–

Date	Store	State	Class	Sales
3/1/96	32	NY	A	6
3/1/96	36	MA	A	9
3/1/96	38	NY	B	5
3/1/96	41	CT	A	11
3/1/96	43	NY	A	9
3/1/96	46	RI	B	3
3/1/96	47	CT	B	7
3/1/96	49	NY	A	12

A B C

Total sales = ? (4*8+4*4+4*2+6*1=62)
How many class A store in NY ? (3)
Sales of class A store in NY = ? (2*8+2*4+1*2+1*1=27)
How many stores in CT ? (2)
Join operation (query product list of class A store in NY)

int dbopendb(char * dbname)
: open a database.
int dbclosedb(unsigned dbid)
: close a database.
int dbTableInfo(unsigned rid, TableInfo * tinfo)
: get the information of the table referenced by *r* .
int dbopen(char * tname,int mode, int flag)
: open the table *w v n* and assign a rid for it.
int dbclose(unsigned rid)
: close the table referenced by *r* and release the *r* .
int dbrename(oldname, newname)
: rename the table.

int dbcreateattr (unsigned rid , sstree * attrlist)

: create some attributes in the table referenced by *r* .

int dbupdateattrbyidx(unsigned rid, int nth, sstree attrinfo)

: update the definition of the nth attribute in the table referenced by *r* .

int dbupdateattrbyname(unsigned rid, char * attrname, sstree attrinfo)

: update the definition of attribute *w v n* in the table referenced by *r* .

int dbinsert(unsigned rid, char * tuple, int length, int flag)

: insert a tuple into the the table referenced by *r* .

int dbdelete(unsigned rid, long offset, int flag)

: delete the tuple specified by *ɷ n* in the table referenced by *r* .

int dbupdate(unsigned rid, long offset, char * newtuple, int flag)

: update the tuple specified by *ɷ n* in the table referenced by *r* with *ɯn yɯ*.

int dbgetrecord(unsigned rid, int nth, char* buf)

: fetch out the nth tuple from the table referenced by *r* and put it into buffer *o*.

int dbopenidx(unsigned rid, indexattrstruct * attrarray, int flag)

: open the index of the table referenced by *r* and assign a *rr* for it.

int dbcloseidx(unsigned iid)

: close the index referenced by *rr* .

int dbfetch(unsigned rid, char * buf, long offset)

: fetch out the tuple specified by *ɷ n* from the table referenced by *r* and put it into buffer *o*.

int dbfetchtid(unsigned iid, void * pvalue, long*offsetbuf, flag)

: fetch out the TIDs of tuples whose value on indexed attribute has the –*ou p*‖ relation with *y u n*, and put them into *ɷ n o*. *rr* is the reference of the index used.

int dbpack(unsigned rid)

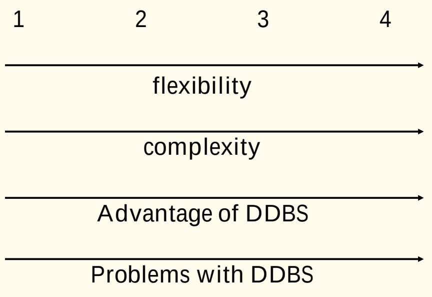
: re-organize the relation, delete the tuples having deleted flag physically.

(1) Centralized: distributed system, but the data are still stored centralized. It is simplest, but there is not any advantage of DDB.

(2) Partitioned: data are distributed without repetition. (no copies)

(3) Replicated: a complete copy of DB at each site. Good for retrieval-intensive system.

(4) Hybrid (mix of the above): an arbitrary fraction of DB at various sites. The most flexible and complex distributing method.



(1) According to relation(or file), that means non partition

(2) According to fragments
Horizontal fragmentation: tuple partition
Vertical fragmentation: attribute partition
Mixed fragmentation: both

(1) Completeness: every tuple or attribute must has its reflection in some fragments.

(2) Reconstruction: should be able to reconstruct the original global relation.

(3) Disjointness: for horizontal fragmentation.

(1) Horizontal Fragmentation
Defined by selection operation with predicate, and reconstructed by union operation.

```
SELECT *
FROM R
WHERE P ;
```

R→n fragments (use P_1 , P_2. . .P_n)
Fulfill: $P_i \wedge P_j$=false (i≠j)
$P_1 \vee P_2 \vee . . . \vee P_n$=true

Derived Fragmentation: relation is fragmented not according to itself‘s attribute, but to another relation‘s fragmentation.

TEACHER(TNAME, DEPT)
COURSE(CNAME,TNAME)
Suppose TEACHER has been fragmented according to DEPT, we want to fragment COURSE even if there is no DEPT attribute in it. This will be the fragmentation derived from TEACHER‘s fragmentation.

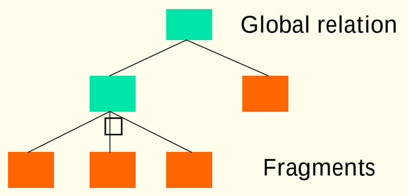
Semi join : $R \ltimes S = \Pi_R (R \bowtie S)$

∴TEACHER9 = SELECT * FROM TEACHER
WHERE DEPT=‗9th‘;

COURSE9=COURSE ⋉ TEACHER9

Defined by project operation, and reconstructed by join operation. Note:
Completeness: each attribute should appear in at least one fragment.
Reconstruction: should fulfill the condition of lossless join decomposition when fragmentizing.
a. Include a key of original relation in every fragment.
b. Include a TID of original relation produced by system in every fragment.

Apply fragmentation operations recursively.
Can be showed with a fragmentation tree:



We can simplify a complex problem through –information hiding‖ method

Level 1: Fragmentation Transparency

User only need to know global relations, he don't have to know if they are fragmentized and how they are distributed. In this situation, user can not feel the distribution of data, as if he is using a centralized database.

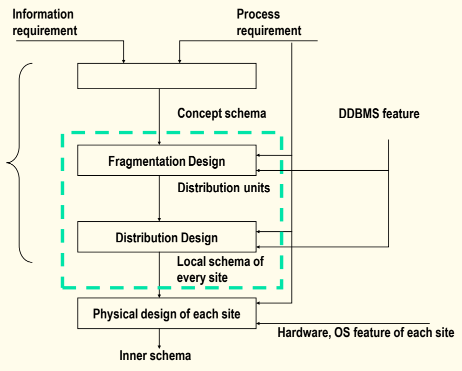
Level 2: Location Transparency

User need to know how the relations are fragmentized, but he don't have to worry the store location of each fragment.

Level 3 Local Mapping Transparency

User need to know how the relations are fragmentized and how they are distributed, but he don't have to worry every local database managed by what kind of DBMS, using what DML, etc.

Level 4 No Transparency



In DDB, it is not true that the fragments should be divided as fine as possible. It should be fragmentized according to the requirement of application. For example, there are following two applications:

App1: SELECT GRADE FROM STUDENT
WHERE DEPT=‗9th‘ AND AGE>20;

App2: SELECT AVG(GRADE)
FROM STUDENT
WHERE SEX=‗Male‘;

if STUDENT should be fragmentized horizontally according to DEPT?

Select some important typical applications which occur often.

Analyze the local feature of the data accessed by these applications.

For horizontal fragmentation:

Select suitable predicate to fragmentize the global relations to fit the local requirement of each site. If there is any contradiction, consider the need of more important application.

Analyze the join operations in applications to decide if derived fragmentation is needed.

For vertical fragmentation:

Analyze the affinity between attributes, and consider:

Save storage space and I / O cost

Security. Some attributes should not be seen by some users.

(2) Distribution design

Through cost estimation, decide the suitable store location (site) of each distribution unit. p252

What is parallel database system?

Share Noting (SN) structure

Vertical parallel and horizontal parallel

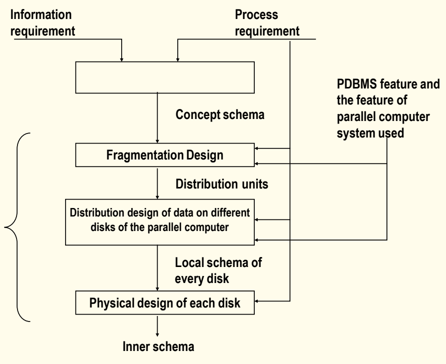
A complex query can be decomposed into several operation steps, the parallel process of these steps is called vertical parallel.

For the scan operation, if the relation to be scanned is fragmentized beforehand into several fragments, and stored on different disks of a SN structured parallel computer, then the scan can be processed on these disk in parallel. This kind of parallel is called horizontal parallel.

SELECT *
FROM R,S
WHERE R.a=S.a AND
R.b>20 AND
S.c<10;

$\bowtie$

The precondition of horizontal parallel is that R, S are fragmentized beforehand and stored on different disks of a SN structured parallel computer. This is the main problem should be solved in PDB design.


Information requirement
Process requirement
Concept schema
PDBMS feature and the feature of parallel computer system used
Fragmentation Design
Distribution units
Distribution design of data on different disks of the parallel computer
Local schema of every disk
Physical design of each disk
Inner schema

(1)Arbitrary

Fragmentize relation R in arbitrary mode, then stored these fragments on the disk of different processor. For example, R may be divided averagely, or hashed into several fragments, etc.

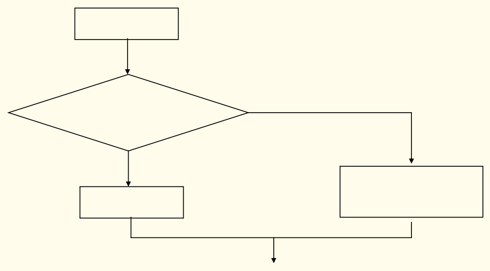
(2)Based on expression

Put the tuples fulfill some condition into a fragment. Suitable for the situation in which the most query are based on fragmentation conditions. --- excluded respectively.

The difference between PDB and DDB about data fragmentation and distribution

	Promote parallel process degree, use the parallel computer's ability as adequately as possible	Promote the local degree of data access, reduce the data transferred on network
	PDBMS feature and the feature of parallel computer system used, combining application requirements.	Application requirements, combining the feature of DDBMS used.
	On multi disks of a parallel computer	On multi sites in the network

Design flow of DDB with PDB as local database



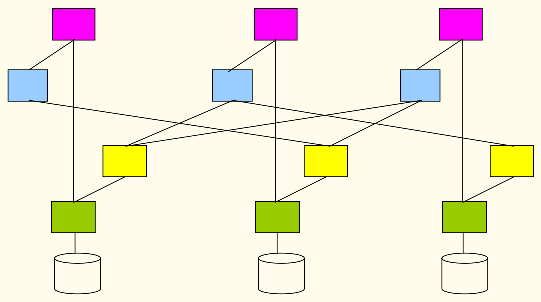
In practical applications, there are strong requirements for solving the integration of multi existing, distributed and heterogeneous databases.

The database system in which every member is autonomic and collaborate each other based on negotiation --- federated database system.

No global schema in federated database system, every federated member keeps its own data schema.

The members negotiate each other to decide respective input/output schema, then, the data sharing relations between each other are established.

The schema structure in federated database System



$FS_i = CS_i + IS_i$

FS_i is all of the data available for the users on $site_i$.

IS_i is gained through the negotiation with ES_j of other

Characteristics:

There is no global catalog

Independent naming and data definition

The catalog grows reposefully

The most important concept --- System Wide Name (SWN)

ObjectName: the name given by user for the data object

User: user's name. With this, different users can access different data object using the same name.

UserSite: the ID of the site where the User is. With this, different users on different sites can use the same user name.

BirthSite: the birth site of the data object. There is no global catalog in R* system. At the BirthSite the information about the data is always kept even the data is migrated to other site.

Print Name (PN): user used normally when they access a data object.

<PN>::=[User[@UserSite].]ObjectName[@BirthSite]

Establish a synonym table for each user using –Define Synonym ... ” statement.

ObjectName	SWN

Mapping PN in different forms according to following rules:

1) PrintName = SWN, need not transform
2) Only have ObjectName: search –ObjectName‖ in the synonym table of current user on current site.
3) User.ObjectName: search the synonym table of user –User‖ on current site.
4) User@UserSite.ObjectName

5) ObjectName@BirthSite

If no match for the ObjectName is found in (2), (3) or print name is in the form of (4) or (5), name completion is used.

A missing User is replaced by current User.

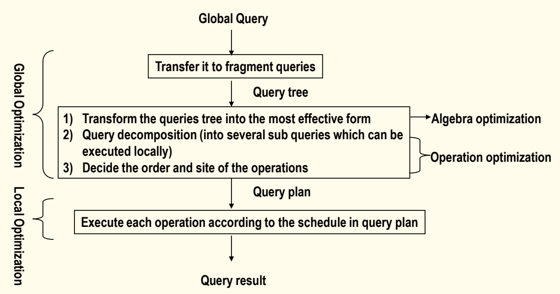
A missing UserSite or BirthSite is replaced by current site ID.

“ Rewrite‖ the query statements submitted by user first, and then decide the most effective operating method and steps to get the result.

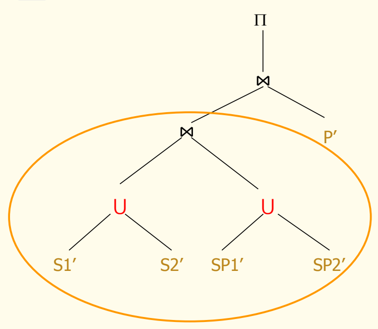
The goal is to gain the result of user's query with the lowest cost and in shortest time.

: a query over global relation.

: a query over fragments.


Global Query
Transfer it to fragment queries
Query tree
1) Transform the queries tree into the most effective form
2) Query decomposition (into several sub queries which can be executed locally)
3) Decide the order and site of the operations
Algebra optimization
Operation optimization
Query plan
Execute each operation according to the schedule in query plan
Query result
Global Optimization
Local Optimization

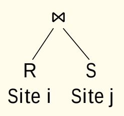
S(SNUM, SNAME, CITY)
SP(SNUM, PNUM, QUAN)
P(PNUM, PNAME, WEIGHT, SIZE)
Suppose the fragmentation is as following:
S1 = σ



First consider distributed JN: (1) (S1‘ U S2‘) ⋈ (SP1‘ U SP2‘)
(2) Distributed Join

Then consider –Site Selection‖, may produce many combination

For every join operation, there are many computing method:



R → Site j, R ⋈ S
S → Site i, R ⋈ S
$\Pi_{JN\ Attr.}$(S) → Site i, R ⋉ S → Site j, (R ⋉ S) ⋈ S

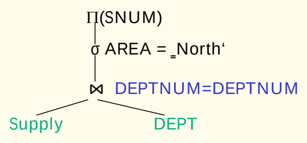
The goal of query optimization is to select a –good‖ solution from so many possible execution strategies. So it is a complex task.

That is so called algebra optimization. It takes a series of transform on original query expression, and transform it into an equivalent, most effective form to be executed.

For example: $\Pi_{NAME,DEPT}\sigma_{DEPT=15}(EMP) \equiv \sigma_{DEPT=15}\Pi_{NAME,DEPT}(EMP)$

(1)Query tree

For example: $\Pi_{SNUM}\sigma_{AREA=‚NORTH‘}(SUPPLY \bowtie_{DEPTNUM} DEPT)$



Leaves: global relation
Middle nodes: unary / binary operations
Leaves → root: the executing order of operations

(2) The equivalent transform rules of relational algebra

1) Exchange rule of ⋈ / ×: $E1 \times E2 \equiv E2 \times E1$
2) Combination rule of ⋈ / ×: $E1\times(E2\times E3) \equiv (E1\times E2)\times E3$
3) Cluster rule of Π: $\Pi_{A1\ldots An}(\Pi_{B1\ldots Bm}(E)) \equiv \Pi_{A1\ldots An}(E)$, legal when $A_1\ldots A_n$ is the sub set of $\{B_1\ldots B_m\}$
4) Cluster rule of σ: $\sigma_{F1}(\sigma_{F2}(E)) \equiv \sigma_{F1\ \ F2}(E)$
5) Exchange rule of σ and Π: $\sigma_F(\Pi_{A1\ldots An}(E)) \equiv \Pi_{A1\ldots An}(\sigma_F(E))$ if F includes attributes $B_1\ldots B_m$ which don‘t belong to $A_1\ldots A_n$, then $\Pi_{A1\ldots An}(\sigma_F(E)) \equiv \Pi_{A1\ldots An}\sigma_F(\Pi_{A1\ldots An,\ B1\ldots Bm}(E))$
6) If the attributes in F are all the attributes in E1, then $\sigma_F(E1\times E2) \equiv \sigma_F(E1)\times E2$

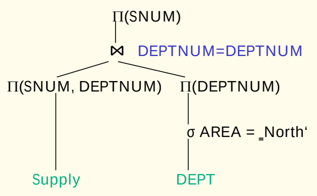
if F in the form of F1   F2, and there are only E1‘s attributes in F1, and there are only E2‘s attributes in F2, then $\sigma_F(E1\times E2) \equiv \sigma_{F1}(E1)\times\sigma_{F2}(E2)$

if F in the form of F1   F2, and there are only E1‘s attributes in F1, while F2 includes the attributes both in E1 and E2, then $\sigma_F(E1\times E2) \equiv \sigma_{F2}(\sigma_{F1}(E1)\times E2)$

7) $\sigma_F(E1 \cup E2) \equiv \sigma_F(E1) \cup \sigma_F(E2)$
8) $\sigma_F(E1 - E2) \equiv \sigma_F(E1) - \sigma_F(E2)$
9) Suppose $A_1\ldots A_n$ is a set of attributes, in which $B_1\ldots B_m$ are E1‘s attributes, and $C_1\ldots C_k$ are E2‘s attributes, then
$\Pi_{A1\ldots An}(E1\times E2) \equiv \Pi_{B1\ldots Bm}(E1)\times\Pi_{C1\ldots Ck}(E2)$

10) $\Pi_{A1\ldots An}(E1 \cup E2) \equiv \Pi_{A1\ldots An}(E1) \cup \Pi_{A1\ldots An}(E2)$
From the above we can see, the goal of algebra optimization is to simplify the execution of the query, and the target is to make the scale of the operands which involved in binary operations be as small as possible.

(3) The general procedure of algebra optimization please refer to p118.



Methods:

For horizontal fragmentation: R = R1 U R2 U … U Rn

For vertical fragmentation: S = S1 $\bowtie$ S2 $\bowtie$ … $\bowtie$ Sn

Replace the global relation in query expression with the above. The expression we get is called canonical expression

Transform the canonical expression with the equivalent transform rules introduced above. Principles:

1) Push down the unary operations as low as possible
2) Look for and combine the common sub-expression

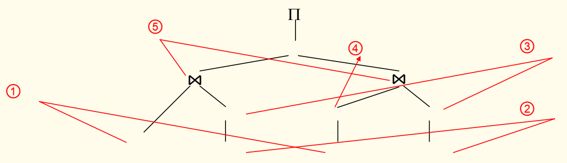
Definition: the sub-expression which occurs more than once in the same query expression. If find this kind of sub-expression and compute it only once, it will promote query efficiency.

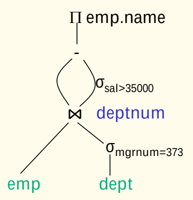
General method:

(1) combine the same leaves in the query tree

(2) combine the middle nodes corresponding to the same operation with the same operands.

example: $\Pi_{emp.name}$(emp $\bowtie$ ($\sigma_{mgrnum=373}$ dept) – ($\sigma_{sal>35000}$ emp) $\bowtie$ ($\sigma_{mgrnum=373}$ dept))





Common sub-expression:

$\bowtie$ σ

Properties:

R $\bowtie$ R ≡ R

R U R ≡ R

R - R ≡ Φ

R $\bowtie$ σ_F(R) ≡ σ_F(R)

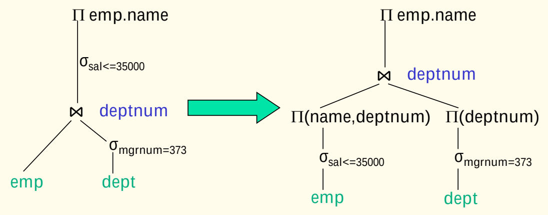
R U σ_F(R) ≡ R

R - σ_F(R) ≡ $\sigma_{not\ F}$ R

σ_{F1}(R) $\bowtie$ σ_{F2}(R) ≡ $\sigma_{F1\ \wedge\ F2}$(R)

σ_{F1}(R) U σ_{F2}(R) ≡ $\sigma_{F1\ \vee\ F2}$(R)

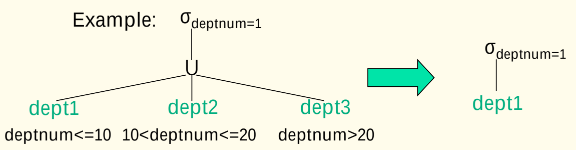
σ_{F1}(R) - σ_{F2}(R) ≡ $\sigma_{F1\ \wedge\ not\ F2}$(R)



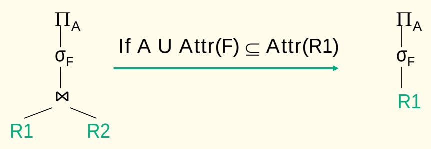
Notice: The last query tree is which can be given by an expert at first. The goal of algebra optimization is to optimize the query expression which is not submitted in best form at first.

3) Find and eliminate the empty sub-expression

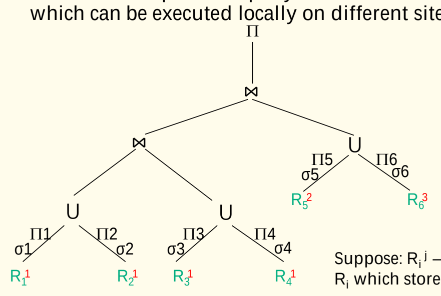
Example:



4) Eliminate useless vertical fragments

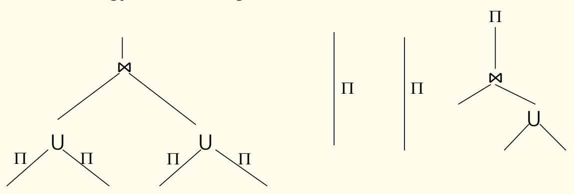


Considering the sites on which the fragments are stored, need to decompose the query into several sub-queries which can be executed locally on different sites:



Suppose: R_i^j — fragment R_i which stored on site j.

Traverse the query tree in post-order, until j become 2, then get the first sub-tree. The rest may be deduced by analogy, so we can get all of the sub-trees.



The executions of local sub-queries are responsible by local DBMS. The query optimization of DDBMS is responsible for the global optimization, that is the execution of assembling tree.

Because the executions of unary operations are responsible by local DBMS after algebra optimization and query decomposition, the global optimization of DDBMS only need to consider the binary operations, mainly the join operation.

How to find a –good‖ access strategy to compute the query improved by algebra optimization is introduced in this section.

According to different environment:

For wide area network: the transfer rate is about 100bps~50Kbps, far slow than processing speed in computer, so *n rap* can be omitted.

For local area network: the transfer rate will reach 1000Mbps, both items should be considered.

1) Transmission cost

$TC(x)=C_0 + C_1x$

x: the amount of data transferred; C_0: cost of initialization; C_1: cost of transferring one data unit on network. C_0 , C_1 rely on the features of the network used.

Processing cost = $cost_{cpu} + cost_{I/O}$

$cost_{cpu}$ can be omitted generally.

cost of one I / O = $D_0 + D_1$

D_0: the average time looking for track (ms);

D_1: time of one data unit I / O (μs, can be omitted)

$cost_{I/O}$ = no. of I / O×D_0

Notice: calculate query cost accurately is unnecessary and unpractical. The goal is to find a good solution through the comparison between different solutions, so only need to estimate the execution cost of different solutions under the same execution environment.

I. The role of semi_join

Semi_join is used to reduce transmission cost. So it is suitable for WAN only.

$R \ltimes S = \Pi_R(R \bowtie S)$

if R and S are stored on site 1 and 2 respectively, the steps to realize $R \bowtie S$ with $\ltimes$ is as following:

1) Transfer $\Pi_A(S) \rightarrow$ site1, A is join attribute
2) Execute $R \bowtie \Pi_A(S) = R \ltimes S$ on site1 (compress R)
3) Transfer $R \ltimes S \rightarrow$ site2
4) Execute $(R \ltimes S) \bowtie S = R \bowtie S$ on site2

Cost of direct join = $C_0 + C_1\min(r, s)$ ------

r, s --- |R|, |S| (size of the relations)

Cost of join via semi_join =

$\min(2C_0 + C_1s' + C_1r'', 2C_0 + C_1r' + C_1s'') =$

$2C_0+C_1\min(s' + r'', r' + s'')$ ------

s‘, r‘ --- $|\Pi_A(S)|$, $|\Pi_A(R)|$

s‖, r‖ --- $|S \ltimes R|$, $|R \ltimes S|$

Only when < , use of semi_join is cost-efficient :

(1) C_0 must be small

(2) unsuitable for using multi semi_join

(3) the size of R or S should be reduced greatly through semi_join

1) The reduce on transmission cost through $\ltimes$ is gained through the sacrifice on processing cost.
2) There are many candidate solutions of semi_join.
   For example : for the query $R_1 \bowtie R_2 \bowtie R_3 \dots \bowtie R_n$, consider the $\ltimes$ to R_1, maybe:
   $R_1 \ltimes R_2$, $R_1 \ltimes (R_2 \ltimes R_1)$, $R_1 \ltimes (R_2 \ltimes R_3)$, …
   it is almost impossible to select the best from all possible solutions.
3) Bernstein‘s remark
   $\ltimes$ can be regarded as reducers.
   Definition: A chain of semi_join to reduce R is called reducer program for R.

RED(Q, R): A set of all reducer programs for R in query Q.

Full reducer: the reducer which conforms to the following conditions:

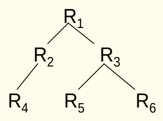
(1) ∊RED(Q, R)

(2) reduce R mostly

But full reducer is not the target which should be pursued in query optimization.

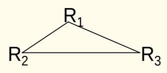
example1: Q is a query with qualification:

q = $(R_1.A=R_2.B)$ $(R_1.C=R_3.D)$ $(R_2.E=R_4.F)$
$(R_3.G=R_5.H)$ $(R_3.J=R_6.K)$



Link the two relations with [illegible] ⋈ between them, the[illegible] query graph. The [illegible] graph like the le[illegible]alled tree query (TQ)

Example 2: q = $(R_1.A=R_2.B)$ $(R_2.$[illegible] $(R_3.E=R_1.F)$



The query wh[illegible]y graph like the left graph is calle[illegible]uery (CQ)

Example 3: q = $(R_1.A=R_2.B)$ $(R_2.B$[illegible]$C=R_1.A)$

This is a TQ, not a CQ, because $R_3.C=$[illegible] from transfer relation, it is not a independ[illegible]

1) Full reducer exists for TQ.
2) No full reducer exists for[illegible]

[illegible]	A	B
	[illegible]	1
	[illegible]	4

R_2	C	[illegible]
	[illegible]	[illegible]

R_3	E	F
	2	3
	5	0

q = $(R_1.B=R_2.C)$ (R[illegible]E) $(R_3.F=R_1.A)$

Even if the result o[illegible]ery is empty, the size of any one of R_1, R_2 and R[illegible]t be de[illegible]So there is not full re[illegible]r thi[illegible]

R	A	B
	1	a
	2	b
	3	c

S	B

q = (R.B=S.B) (S.C=T.C) (T.A=R.A), is there full reducer?

S6

- Nested Loop
  - O: shipped whole
  - I : fetch as needed
- Merge Scan
  - O: shipped whole
  - I :
    - shipped whole
    - fetch as needed
- Shipping whole O and I to a 3rd site (NL or MS)

It is obvious that O should be shipped whole.

In NL, if I is shipped whole, index can‘t be shipped along with it, moreover temporary relation is required. Both processing cost and storage cost are high.

Six strategies don’t include:

Multiple join --- transformed into multi binary joins.

Copy selection --- because R* doesn‘t support multi copies.

5.8 Distributed Grouping & Aggregate Function Evaluation

SELECT PNUM, SUM(QUAN)

FROM SP

GROUP BY PNUM;

That is : $GB_{PNUM,\ SUM(QUAN)}SP$

There are the following conclusions about grouping & aggregate function evaluation in distributed computing environment :

1) Suppose G_i is a group gotten through grouping to $R_1 \cup R_2$ according to some attribute set, iff $G_i \subseteq R_j$ OR $G_i \cap R_j = \Phi$ for all i, j ------ (SNC), then :

$GB_{G,\ AF}(R_1 \cup R_2) = (GB_{G,\ AF}R_1) \cup (GB_{G,\ AF}R_2)$

For example:

SELECT SNUM, AVG(QUAN) FROM SP GROUP BY SNUM;

If SP is derived fragmented according to the supplier‘s city: conform to SNC, so the grouping & aggregate can be evaluated distributed.

If SP is derived fragmented according to the part‘s type: don‘t conform to SNC, because the same supplier may provide more than one kinds of part at same time. In this situation the grouping & aggregate can not be evaluated distributed.

2) If SNC does not hold, it is still possible to compute some aggregate functions of global relation distributed

Suppose	global relation: S
	fragments: $S_1, S_2, \ldots, S_n$

then:

$SUM(S) = SUM(SUM(S_1), SUM(S_2), \ldots, SUM(S_n))$

$COUNT(S) = SUM(COUNT(S_1), \ldots COUNT(S_n))$

$AVG(S) = SUM(S) / COUNT(S)$

$MIN(S) = MIN(MIN(S_1), MIN(S_2), \ldots, MIN(S_n))$

$MAX(S) = MAX(MAX(S_1), MAX(S_2), \ldots, MAX(S_n))$

The consistency between multi copies must be considered while executing update, because any data may have multi copies in DDB.

1) Updating all strategy

The update will fail if any one of copies is unavailable.

p --- probability of availability of a copy.

n --- No. of copies

The probability of success of the update=p^n

$$\lim_{n\to\infty} p^n = 0$$

2) Updating all available sites immediately and keeping the update data at spooling site for unavailable sites, which are applied to that site as soon as it is up again.

3) Primary copy updating strategy

Assign a copy as primary copy. The remaining copies called secondary copies.

Update : update P.C, then P.C broadcast the update to S.Cs at sometimes.

P.C maybe inconsistent with S.C temporarily. There is no problem if the next operation is still update. While if the next operation is a read to some S.C, then:

Compare the version No. of S.C with that of P.C, if version No. are equal, read S.C directly; else:

(1) redirect the read to P.C

(2) wait the update of S.C

4) Snapshot

Snapshot is a kind of copy image not followed the changes in DB.

Master copy at one site, many snapshots are distributed at other sites.

Update: master copy only.

$\text{Read:}\left\{\begin{array}{l}\text{master copy}\\\text{snapshots}\end{array}\right\}\text{ is indicated by users}$

The snapshot can be refreshed:

(1) periodically

(2) forced refreshing by REFRESH command

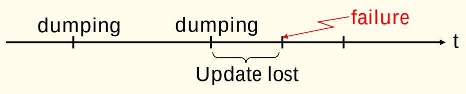
Snapshot is suitable for the application systems in which there is less update, such as census system, etc.

The main roles of recovery mechanism in DBMS are:

(1) Reducing the likelihood of failures (prevention)

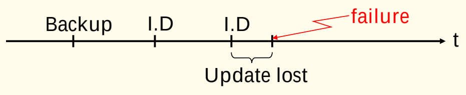
(2) Recover from failures (solving)

Redundancy is necessary.
Should inspect all possible failures.
General method:



Variation : Backup + Incremental dumping

I.D --- updated parts of DB



This method is easy to be implemented and the overhead is low, but the update maybe lost after failure occurring. So it is often used in file system or small DBMS.

Log : record of all changes on DB since the last backup copy was made.

Change: Old value (before image --- B.I)
New value (after image --- A.I)

While recovering:

Some transactions maybe half done, should undo them with B.I recorded in Log.

Some transactions have finished but the results have not been written into DB in time, should redo them with A.I recorded in Log. (finish writing into DB)

It is possible to recover DB to the most recent consistent state with Log.

Advantages:

(1) increase reliability

(2) recovery is very easy

Problems:

(1) difficult to acquire independent failure modes in centralized database systems.

(2) waste in storage space

So this method is not suitable

6.4.1 Commit Rule

Check two lists for every TID while restarting after failure:

Commit list	Active list	
		undo, delete TID from active list
		redo, delete TID from active list
		nothing to do

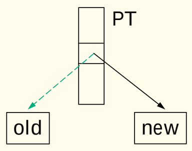
		redo	undo
	a)	✗	
	b)		✗
	c)		
?	d)	✗	✗

Keep two copies for every page of a relation

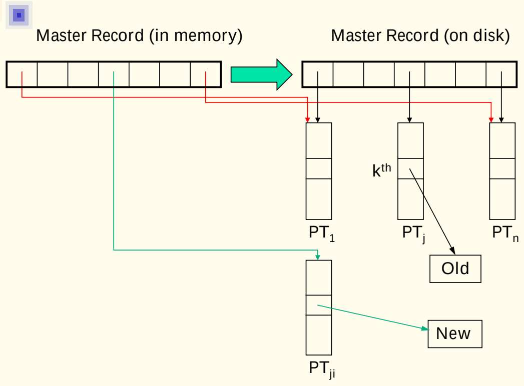
Keep a page table (PT) for every relation

When updating some page, produce a new page out of place, change the corresponding pointer in page table while the transaction committing, let it point to new page.

Suppose relation R has N pages, then the length of its PT is N



P141 : lorie's approach



Failure types :

1) Transaction failure: because of some reason beyond expectation, the transaction has to be aborted.
2) System failure: the operating system collapse, but the DB on disk is not damaged. Such as power cut suddenly.
3) Media failure: disk failure, the DB on the disk is damaged.

Solutions :

1) Transaction failure: because it must occur before committing :
   Undo if necessary
   Delete TID from active list

2) System failure:
   Restore the system
   Undo or redo if necessary
3) Media failure:
   Load the latest dump
   Redo according the log

1) Emergency restart

Start after system or media failure. Recovery is needed before start.

2) Warm start

Start after system shutdown. Recovery is not required.

3) Cold start

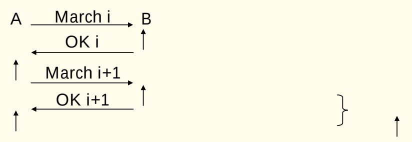
Start the system from scratch. Start after a catastrophic failure or start a new DB.

The transactions in DDBMS are distributed transactions, the key of distributed transaction management is how to assure all sub-transactions either commit together or abort together.

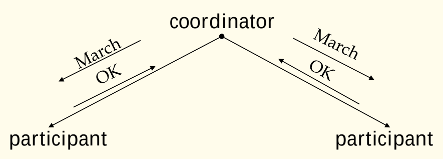
Realize the sub-transactions‘ harmony with each other relies on communication, while the communication is not reliable.

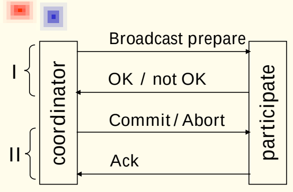
Two general paradox : No fixed length protocol exists.

Solution : number the messages.



When there are multi generals, select one of them as coordinator





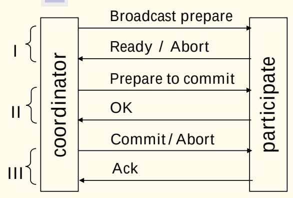
If the sub-transaction successes and can commit, it answers OK, or not OK

If all of the sub-transactions are OK, Commit command is sent out, or Abort command is sent out

OK	--- if success	not OK	--- if fail
Commit	--- all OK	Abort	--- any one not OK

Every participant is self-determining before answering OK, it can abort by itself. Once answers OK, it can only wait for the command come from the coordinator.

If the coordinator has failure after the participates answer OK, the participates have to wait, and is in blocked state. This is the disadvantage of 2PC.



If the sub-transaction successes and can commit, it answers Ready, or Abort

Broadcast to all participants and this MSG is recorded on the disk

If the coordinator has not any failure, phase II is wasted
If the coordinator has failure after the participates answer OK, the participates communicate each other and check the MSG recorded on disk in phase II, and a new coordinator is elected. If the new coordinator finds the Prepare to commit MSG on any participate, it sends out Commit command, or sends out Abort command. So the blocked problem can be solved in 3PC.

In multi users DBMS, permit multi transaction access the database concurrently.

7.1.1 Why concurrency?

1) Improving system utilization & response time.
2) Different transaction may access to different parts of database.

7.1.2 Problems arise from concurrent executions

T_1	T_2	T_1	T_2	T_1	T_2
Read(x)			Read(t[x])	Read(x)	
	Read(x)	Write(t)			Write(x)
x:=x+1					
Write(x)	x:=2*x		Read(t[y])	Read(x)	
	Write(x)	(rollback)			

t ↓

Lost update | Dirty read | Unrepeatable read

So there maybe three kinds of conflict when transactions execute concurrently. They are write – write, write – read, and read – write conflicts. Write – write conflict must be avoided anytime. Write – read and read – write conflicts should be avoided generally, but they are endurable in some applications.

Definition: suppose $\{T_1,T_2,\ldots T_n\}$ is a set of transactions executing concurrently. If a schedule of $\{T_1,T_2,\ldots T_n\}$ produces the same effect on database as some serial execution of this set of transactions, then the schedule is serializable.

Problem: different schedule → different equivalent serial execution → different result? (yes, n!)

T_A	T_B	T_C
	Read R1	
Read R2		Write R1
	Write R2	

The result of this schedule is the same as serial execution $T_A \rightarrow T_B \rightarrow T_C$, so it is serializable. The equivalent serial execution is $T_A \rightarrow T_B \rightarrow T_C$.

qn   ın --- sequences that indicate the chronological order in which instructions of concurrent transactions are executed

a schedule for a set of transactions must consist of all instructions of those transactions

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

Let $_1$ transfer $50 from A to B, and $_2$ transfer 10% of the balance from A to B. The following is a serial schedule, in which $_1$ is followed by $_2$.

1	2
(A)	
$A := A - 50$	
(A)	
(B)	
$B := B + 50$	
(B)	
	(A)
	nw y := A*0.1;
	$A := A -$ *nw y*
	(A)
	(B)
	$B := B +$ *nw y*
	(B)

t ↓

Let $_1$ and $_2$ be the transactions defined previously. The following schedule is not a serial schedule, but it is *n   r   ıuw* to the above.

1	2
(A)	
$A := A - 50$	
(A)	
	(A)
	nw y := A*0.1
	$A = A -$ *nw y*
	(A)
(B)	
$B := B + 50$	
(B)	
	(B)
	$B := B +$ *nw y*
	(B)

t ↓

Let and S' be two schedules with the same set of transactions. and S' are *m n r uw* if they produce the same effect on database based on the same initial execution condition.

Conflict operations : R-W、W-W. The sequence of conflict operation will affect the effect of execution.

Non-conflicting operations: ① R-R ② Even if there are write operation, the data items operated are different. Such as $R_i(x)$ and $W_j(y)$.

If a schedule can be transformed into a schedule S' by a series of swaps of non-conflicting operations, we say that and S' are *wir n r uw*.

Property: if schedule and S' are conflict equivalent, they must be view equivalent. It is not right contrarily.

Serialization can be divided into and
.

Example 1: for the schedule s of transaction set $\{T_1,T_2,T_3\}$

s = $R_2(x)W_3(x)R_1(y)W_2(y) \rightarrow R_1(y)R_2(x)W_2(y)W_3(x)$ = s‘

s is conflict serialization because s‘ is a serial execution.

Example 2: s = $R_1(x)W_2(x)W_1(x)W_3(x)$

There is no conflict equivalent schedule of s, but we can find a schedule s‘

s‘ = $R_1(x)W_1(x)W_2(x)W_3(x)$

It is view equivalent with s, and s‘ is a serial execution, so s is view serialization.

The test algorithm of view equivalent is a NP problem, while conflict serialization covers the most instances of serializable schedule, so the serialization we say in later parts will point to conflict serialization if without special indication.

Directed graph G = <V,E>

V --- set of vertexes, including all transactions participating in schedule.

Locking method is the most basic concurrency control method. There maybe many kinds of locking protocols.

7.2.1 X locks

Only one type of lock, for both read and write.

Compatibility matrix : NL --- no lock X --- X lock

Y --- compatible N --- incompatible

	NL	X
NL	Y	Y
X	Y	N

T_A	T_B
X_lock R	X_lock R
Update R	wait ↓
X_unlock R	
EOT	X_lock R
	Read R

Dwrur w1: In a transaction, if all locks precede all unlocks, then the transaction is called two phase transaction. This restriction is called two phase locking protocol.

Dwrur r w2: In a transaction, if it first acquires a lock on the object before operating on it, it is called well-formed.

qn nw : If S is any schedule of well-formed and two phase transactions, then S is serializable.

(proving is on p151)

	T_1	T_2
Growing phase	Lock A	Lock A
	Lock B	Lock B
	Lock C	Unlock A
		Unlock B
Shrinking phase	Unlock A	Lock C
	Unlock B	
	Unlock C	Unlock C
	2PL	not 2PL

1) Well-formed + 2PL : serializable
2) Well-formed + 2PL + unlock update at EOT: serializable and recoverable. (without domino phenomena)
3) Well-formed + 2PL + holding all locks to EOT: strict two phase locking transaction.

7.2.2 (S,X) locks

S lock --- if read access is intended.

X lock --- if update access is intended.

	NL	S	X
NL	Y	Y	Y
S	Y	Y	N
X	Y	N	N

U lock --- update lock. For an update access the transaction first acquires a U-lock and then promote it to X-lock.

Purpose: shorten the time of exclusion, so as to boost concurrency degree, and reduce deadlock.

	NL	S	U	X
NL	Y	Y	Y	Y
S	Y	Y	Y	N
U	Y	Y	N	N
X	Y	N	N	N

X (S,X) (S,U,X)

$\longrightarrow$

Concurrency degree

$\longrightarrow$

Overhead

Dead lock: wait in cycle, no transaction can obtain all of resources needed to complete.

Live lock: although other transactions release their resource in limited time, some transaction can not get the resources needed for a very long time.

T_A	T_B
X_lock R1	X_lock R2
X_lock R2	X_lock R1
wait ↓	wait ↓

R ← T: x-lock

T1: S-lock

T2: S-lock

Live lock is simpler, only need to adjust schedule strategy, such as FIFO

Deadlock: (1) Prevention(don't let it occur); (2) Solving(permit it occurs, but can solve it)

1) Timeout: If a transaction waits for some specified time then deadlock is assumed and the transaction should be aborted.
2) Detect deadlock by wait-for graph G=<V,E>

V : set of transactions $\{T_i \mid T_i$ is a transaction in DBS $(i=1,2,\ldots n)\}$

E : $\{<T_i, T_j> \mid T_i$ waits for T_j $(i \neq j)\}$

If there is cycle in the graph, the deadlock occurs.

When to detect?

1) whenever one transaction waits.
2) periodically

What to do when detected?

1) Pick a victim (youngest, minimum abort cost, …)
2) Abort the victim and release its locks and resources
3) Grant a waiter
4) Restart the victim (automatically or manually)

7.3.2 Deadlock avoidance

1) Requesting all locks at initial time of transaction.
2) Requesting locks in a specified order of resource.
3) Abort once conflicted.
4) Transaction Retry

Every transaction is uniquely time stamped. If T_A requires a lock on a data object that is already locked by T_B, one of the following methods is used:

a) Wait-die: T_A waits if it is older than T_B, otherwise it –diesǁ, i.e. it is aborted and automatically retried with original timestamp.
b) Wound-wait: T_A waits if it is younger than T_B, otherwise it –woundǁ T_B, i.e. T_B is aborted and automatically retried with original timestamp.

In above, both have only one direction wait, either older → younger or younger → older. It is impossible to occur wait in cycle, so the dead lock is avoided.

7.4.1 Locking in multi granularities

To reduce the overhead of locking, the lock unit should be the bigger, the better; To boost the concurrency degree of transactions, the lock unit should be the smaller, the better.

In large scale DBMS, the lock unit is divided into several levels:

DB - File - Record - Field

In this situation, if a transaction acquires a lock on a data object of some level then it acquires implicitly the same lock on each descendant of that data object.

So, there are two kinds of locks in multi granularity lock method:

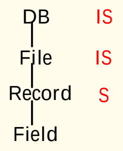
Explicit lock

Implicit lock

How to check conflicts on implicit locks

Intention lock: provide three kinds of intension locks which are IS, IX and SIX. For example, if a transaction adds a S lock on some lower level data object, all the higher level data object which contains it should be added an IS lock as a warning information. If another transaction want to apply an X lock on a higher level data object later, it can find the implicit conflict through IS lock.

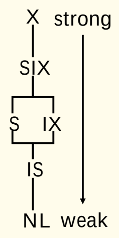
IS --- Intention share lock
IX --- Intention exclusive lock
SIX --- S + IX



IS
IS
S

Compatibility matrix while lock in multi granularities :

	NL	IS	IX	S	SIX	X
NL	Y	Y	Y	Y	Y	Y
IS	Y	Y	Y	Y	Y	N
IX	Y	Y	Y	N	N	N
S	Y	Y	N	Y	N	N
SIX	Y	Y	N	N	N	N
X	Y	N	N	N	N	N



The lock with strong exclusion degree can substitute the lock with weak exclusion degree while locking, but it is not right contrarily.

Locks are requested from root to leaves and released from leaves to root.

Example:

Locking order ←

	record	file	DB
(1)	S	IS	IS
(2)	X	IX	IX
(3)	All read and some write	SIX	IX, IS

Request X lock to records need updating

Substitute with stronger exclusive lock

DB
File
Record
Field

The that the DB is a fixed collection of objects is not true when multi granularity locking is permitted. Then even Strict 2PL will not assure serializability:

- T1 locks all pages containing sailor records with *rap* = 1, and finds oldest sailor (say, *pn* = 71).
- Next, T2 inserts a new sailor; *rap* = 1, *pn* = 96.
- T2 also deletes oldest sailor with rating = 2 (and, say, *pn* = 80), and commits.
- T1 now locks all pages containing sailor records with *rap* = 2, and finds oldest (say, *pn* = 63).

No consistent DB state where T1 is –correct‖!

T1 implicitly assumes that it has locked the set of all sailor records with *rap* = 1.

- Assumption only holds if no sailor records are added while T1 is executing!
- Need some mechanism to enforce this assumption. (Index locking and predicate locking)

Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!

If the system don't support multi granularity locking, or even if support multi granularity locking, the query need to scan the whole table and add S lock on the table, then there is not this problem. For example :

select s#, average(grade) from SC group by s#;

If there is a dense index on the *rap* field, T1 should lock the index node containing the data entries with *rap* = 1 and keep it until EOT.

- If there are no records with *rap* = 1, T1 must lock the index node where such a data entry *u* be, if it existed!

When T2 wants to insert a new sailor (*rap* = 1, *pn* = 96), he can't get the X lock on the index node containing the data entries with *rap* = 1, so he can't insert the new index item to realize the insert of a new sailor.

If there is no suitable index, T1 must lock the

1) Lock granularity: object is the smallest lock granularity in OODB generally. DB - Class - Object
2) Single level locking: lock the object operated with S or X lock directly. Suitable for the OODBMS faced to CAD application, etc. not suitable for the application occasion in which association queries are often.
3) multi granularity lock: use S, X, IS, IX, SIX locks introduced in last section. It is a typical application of multi granularity lock.
   But in this situation, the class level lock can only lock the objects directly belong to this class, can not include the objects in its child classes. So it is not suitable for cascade queries on inheriting tree or schema update.
4) Complex multi granularity lock: two class hierarchy locks are added.

RL --- add a S lock at a class and all of its child classes
WL --- add a X lock at a class and all of its child classes

	NL	IS	IX	S	SIX	X	RL	WL
NL	Y	Y	Y	Y	Y	Y	Y	N
IS	Y	Y	Y	Y	Y	N	Y	N
IX	Y	Y	Y	N	N	N	N	N
S	Y	Y	N	Y	N	N	Y	N
SIX	Y	Y	N	N	N	N	N	N
X	Y	N	N	N	N	N	N	N
RL	Y	Y	N	Y	N	N	Y	N
WL	Y	N	N	N	N	N	N	N

a) Add IS(IX) lock on any super class chain of this class and DB
b)

1. Compared with lock method, the most obvious advantage is that there is no dead lock, because of no wait.
2. Disadvantage: every transaction and every data object has T.S, and every operation need to update tr or tw, so the overhead of the system is high.
3. Solution:

   Enlarge the granularity of data object added T.S. (Low concurrency degree)

   T.S of data object are not actually stored in nonvolatile storage but in main memory and preserved for a specified time and the T.S of data objects whose T.S is not in main memory are assumed to be zero.

The key idea of optimistic method is that it supposes there is rare conflict when concurrent transactions execute. It doesn't take any check while transactions are executing. The updates are not written into DB directly but stored in main memory, and check if the schedule of the transaction is serializable when a transaction finishes. If it is serializable, write the updating copies in main memory into DB; Otherwise, abort the transaction and try again.

The lock method and time stamp method introduced above are called –pessimistic method‖.

1. *n yq n*: read data from database and execute every kind of processing, but update operations only form update copies in memory.
2. *ur nyq n*:

Write: n lock MSG
n lock grants
n update MSG
n ACK
[n unlock MSG]
4n

Read: 1 lock MSG
1 lock grants
1 read MSG
Can be merged
2

7.11.2 Majority locking

Read R ---S_lock on a majority of copies of R

Write R---X_lock on a majority of copies of R

Hold the locks to EOT

Communication overhead : Majority ---(n+1)/2

Write: (n+1)/2 lock MSG
(n+1)/2 lock grants
n update MSG
n ACK
3n+1

Read: (n+1)/2 lock MSG
(n+1)/2 lock grants
1 read MSG
n+1

In 7.11.1, if there are two transaction compete the X lock for update, maybe each get a part , but no one can X-lock all. The deadlock will occur very easily. In majority locking method, this kind of dead lock is impossible to occur as long as w is an odd.

Write R---X_lock on k copies of R, k>n/2

Read R ---S_lock on n-k+1 copies of R

Hold the locks to EOT

For read-write conflict: k+(n-k+1)=n+1>n, so the conflict can be found on at least one copy.

For write-write conflict: 2k>n, so it is also sure that the conflict can be detected.

The above two methods are the special situations of it:

7.11.1 is k=n; 7.11.2 is k=(n+1)/2

k can be changed between (n+1)/2 ~ n, the bigger of k, the better for read operations.

R---data object

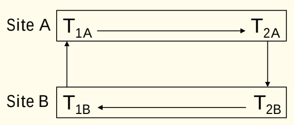
Assign the lock responsibility for locking R to a given site. This site is called primary site of R.

Communication overhead :

Write: 1 lock MSG
1 lock grants
n update MSG
n ACK
2n+1

Read: 1 lock MSG
1 lock grants
1 read MSG
2

It is efficient but liable to fail, so there are many variations. It is often used together with primary copy updating strategy (see 5.9).



- Suppose both T_1 and T_2 are distributed transactions, and have two sub transactions on site A and B respectively.
- T_{1A} and T_{1B} must commit simultaneously
- T_{2A} and T_{2B} must commit simultaneously

The above shows a global dead lock. How to find out this kind of dead lock?

add EXT nodes based on general wait-for graph. If transaction T is a distributed transaction, and has sub transactions on other sites, and T is the head of wait-for chain of current site, add EXT$\rightarrow$T into the graph; if T is the tail of wait-for chain of current site, add T$\rightarrow$EXT into the graph.

If on some site has: EXT$\rightarrow T_i \rightarrow T_j \rightarrow \quad \rightarrow T_k \rightarrow$EXT

1) Check other sites if has: EXT$\rightarrow T_k \rightarrow T_l \rightarrow \quad \rightarrow T_x \rightarrow$EXT

2) if $T_x = T_i$: global deadlock is detected.
if $T_x \neq T_i$: merge two wait-for graphs:
EXT$\rightarrow T_i \rightarrow T_j \rightarrow \quad \rightarrow T_k \rightarrow T_l \rightarrow \quad \rightarrow T_x \rightarrow$EXT

3) Repeat step 1 and 2, check if T_x will result in global dead lock like T_k when the condition in 2 is true. If wait-for graph on all sites have been check like above and no global cycle is found, no global dead lock occur.

7.12.1 global time stamp

To keep the uniqueness of transaction time stamp in the whole distributed system, define a global time stamp:

Global T.S = Local T.S + Site ID

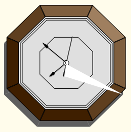
The clock on different site maybe different. It is not important. The key is to assure :

time of receipt >= time of delivery

Solution: $t_{\text{at receipt site}} := \max(t_1, t_2)$

t_1---current T.S at receipt site

t_2---T.S of MSG



1) Write---update t_w of all copies.
2) Read ---update t_r of the copy read.
3) When writing we check T.S of all copies. If $t<t_r$ or $t<t_w$ for any copy the transaction should be aborted. When reading we check T.S of the copy read. If $t<t_w$ the transaction should be aborted.