Course Goal and its Preliminary Courses

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 build the foundation of further research in database field and to use database system better through the study of this course.

Main Contents

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 implementation methods related to DBMS core.
Main Contents

Because the relational data model is the mainstream data model, and distributed DBMS includes all aspects of classical centralized DBMS, the main thread of this course is **relational distributed database management system**. The implementation of every aspects of DBMS are introduced according to distributed DBMS. Some contents of other kinds of DBMS are also introduced, including federated database systems, parallel database systems and object-oriented database systems, etc. Along with the continuous progress of database technique, new contents will be added at any time.

Course History

- Database System Principles (before 1984) ---- relational model theory, some query optimization algorithms
- Distributed Database Systems (1985~1994) ---- introduce implementation techniques in DDBMS thoroughly and systemically
- Database Management Systems and Their Implementation (after 1995) ---- not limited to DDBMS, hope to introduce the implementation techniques of DBMS more thoroughly. Along with the progress of database technique, new contents can be added without changing the course name.

References

1) Stefano Ceri, “Distributed Databases”
5) S.Bing Yao et al, “Query Optimization in DDBS”
6) Courseware: http://cse.seu.edu.cn/people/lzxu/resource
1. Introduction

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

2. DBMS Architecture

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

3. Access Management of Database

Physical file organization, index, and access primitives

4. Data Distribution

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

5. Query Optimization

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

6. Recovery Mechanism

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

7. Concurrency Control

Basic problems; Concurrency control techniques; Concurrency control in distributed DBMS; Concurrency control in other kinds of DBMS
1.1 The History of Database Technology and its Classification

(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
1.2 Distributed Database Systems

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.

Features of DDBS:
- Distribution
- Correlation
- DDBMS
The advantages of DDBS:
- 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

The disadvantages of DDBS:
- Hard to integrate existing databases
- Too complex (system itself and its using, maintenance, etc. such as DDB design)

The main problems in DDBS:
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)

Another problem specially for DDBS:
- Data distribution

2. The Architecture of DBMS
Main Contains

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

2.1 The Components of DBMS Core

2.2 The Process Structure of DBMS

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

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

Multi processes structure

- One application process corresponding to one DBMS core process

Multi threads structure

- Only one DBMS process, every application process corresponding to a DBMS core thread.
Communication protocols between processes/threads

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

<table>
<thead>
<tr>
<th>Pipe0</th>
<th>DBMS core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ad-hoc interface or</td>
<td></td>
</tr>
<tr>
<td>application program</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>State</th>
<th>TupNum</th>
<th>AttNum</th>
<th>AttName</th>
<th>AttType</th>
<th>AttLen</th>
<th>TmpFileName</th>
</tr>
</thead>
</table>

- 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.

2.3 The Components of DDBMS Core

DB: database management  
DC: communication control  
DD: catalog management  
DDBK: core, responsible for parsing, distributed transaction management, concurrency control, recovery and global query optimization.
An example of global query optimization

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;

2.4 The Process Structure of DDBMS

3. Database Access Management
Main Contains

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

Access Types

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

File Organization

- 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)
Index Technique

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

Bitmap index – index itself is data

<table>
<thead>
<tr>
<th>Date</th>
<th>State</th>
<th>Class</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>3/1/96</td>
<td>NY</td>
<td>A</td>
<td>6</td>
</tr>
<tr>
<td>3/1/96</td>
<td>MA</td>
<td>A</td>
<td>9</td>
</tr>
<tr>
<td>3/1/96</td>
<td>NY</td>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>3/1/96</td>
<td>CT</td>
<td>A</td>
<td>11</td>
</tr>
<tr>
<td>3/1/96</td>
<td>NY</td>
<td>A</td>
<td>9</td>
</tr>
<tr>
<td>3/1/96</td>
<td>RI</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>3/1/96</td>
<td>CT</td>
<td>B</td>
<td>7</td>
</tr>
<tr>
<td>3/1/96</td>
<td>NY</td>
<td>A</td>
<td>12</td>
</tr>
</tbody>
</table>

Bitmap Index for Sales

<table>
<thead>
<tr>
<th>Bit 0</th>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Bit 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class A</th>
<th>Class B</th>
<th>Class C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Total sales = 4\times4\times4\times4\times2+6+1=62
How many class A store in NY? (3)
Sales of class A store in NY = 2\times4+1=9
How many stores in CT? (2)
Join operation (query product list of class A store in NY)

Access Primitives (examples)

- int dbopendb(char * dbname)
  Function: open a database.
- int dbclosedb(unsigned dbid)
  Function: close a database.
- int dbTableInfo(unsigned rid, TableInfo * tinfo)
  Function: get the information of the table referenced by rid.
- int dbopen(char * tname, int mode, int flag)
  Function: open the table tname and assign a rid for it.
- int dbrename(oldname, newname)
  Function: rename the table.
Access Primitives (examples)

- int dbcreateattr(unsigned rid, sttree * attrlist)
  Function: create some attributes in the table referenced by rid.
- int dbupdateattrbyidx(unsigned rid, int nth, sttree attrinfo)
  Function: update the definition of the nth attribute in the table referenced by rid.
- int dbupdateattrbyname(unsigned rid, char * attrname, sttree attrinfo)
  Function: update the definition of attribute attrname in the table referenced by rid.
- int dbinsert(unsigned rid, char * tuple, int length, int flag)
  Function: insert a tuple into the the table referenced by rid.
- int dbdelete(unsigned rid, long offset, int flag)
  Function: delete the tuple specified by offset in the table referenced by rid.
- int dbupdate(unsigned rid, long offset, char * newtuple, int flag)
  Function: update the tuple specified by offset in the table referenced by rid with newtuple.
- int dbgetrecord(unsigned rid, int nth, char* buf)
  Function: fetch out the nth tuple from the table referenced by rid and put it into buffer buf.
- int dbopenidx(unsigned rid, indexattrstruct * attrarray, int flag)
  Function: open the index of the table referenced by rid and assign a iid for it.
- int dbcloseidx(unsigned iid)
  Function: close the index referenced by iid.
- int dbfetch(unsigned rid, char * buf, long offset)
  Function: fetch out the tuple specified by offset from the table referenced by rid and put it into buffer buf.
- int dbfetchtid(unsigned iid, void * pvalue, long*offsetbuf, flag)
  Function: fetch out the TIDs of tuples whose value on indexed attribute has the “flag” relation with pvalue, and put them into offsetbuf. iid is the reference of the index used.
- int dbpack(unsigned rid)
  Function: re-organize the relation, delete the tuples having deleted flag physically.
4. Data Distribution

4.1 Strategies of Data Distribution

(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.

Comparison of four strategies

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>flexibility</td>
<td>complexity</td>
<td>Advantage of DDBS</td>
<td>Problems with DDBS</td>
</tr>
</tbody>
</table>
4.2 Unit of Data Distribution

(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

The criteria of fragmentation:

(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.

Fragmentation Methods

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

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

R→m fragments (use P₁, P₂…Pₙ)
Fulfill: P₁∧P₂=false
P₂∨…∨Pₙ=true

Derived Fragmentation: relation is fragmented not according to itself’s attribute, but to another relation’s fragmentation.
An example of Derived 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: \( R \bowtie S = \Pi_{k}(R \bowtie S) \)

\[
\text{TEACHER}9 = \text{SELECT } * \text{ FROM TEACHER WHERE DEPT='9th';}
\]

\[
\text{COURSE9} = \text{COURSE } \bowtie \text{TEACHER9}
\]

(2) Vertical Fragmentation

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.

(3) Mixed Fragmentation

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

Global relation

Fragsments

\[ R \]
\[ R_1 \]
\[ R_2 \]
\[ R_{11} \]
\[ R_{12} \]
\[ R_{13} \]

\[ R \]
\[ R_1 \]
\[ R_2 \]
\[ R_{11} \]
\[ R_{12} \]
\[ R_{13} \]
4.3 Different Transparency Level

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

4.4 Problems Caused by Data Distribution

1) Multi copies’ consistency
   Mainly the change of tuples’ store location resulted by updating operation. Solution:
   (1) Redistribution
       After Update: Select→Move→Insert→Delete
   (2) Piggybacking
       Check tuple immediately while updating, if there is any inconsistency it is sent back along with ACK information and then sent to the right place.
3) Translation of Global Queries to Fragment Queries and Selection of Physical Copies.

4) Design of Database Fragments and Allocation of Fragments.

Above 1)-3) should be solved in DDBMS. While 4) is a problem of distributed database design.

4.5 Distributed Database Design

1) Distributed database
   - The design of fragments
   - The design of fragment distribution solution

   To understand user’s requirements, we should ask the following questions to every application while requirement analysis:
   - The sites where this application may occur
   - The frequency of this application
   - The data object accessed by this application
(1) Fragmentation design

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';

Problem: if STUDENT should be fragmentized horizontally according to DEPT?

General rules:

① Select some important typical applications which occur often.
② Analyze the local feature of the data accessed by these applications.
   • For horizontal fragmentation:
     ③ 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.
     ④ Analyze the join operations in applications to decide if derived fragmentation is needed.
For vertical fragmentation:
③ 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

2) Parallel database

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.

Example:

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

\[ R' = \sigma_{b>20}(R) \]
\[ S' = \sigma_{c<10}(S) \]
\[ R' \bowtie S' \]

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.
Data fragmentation mode in PDB

(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

<table>
<thead>
<tr>
<th></th>
<th>PDB</th>
<th>DDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>The goal of</td>
<td>Promote parallel process degree, use the parallel</td>
<td>Promote the local degree of data access, reduce the data</td>
</tr>
<tr>
<td>fragmentation</td>
<td>computer’s ability as adequately as possible</td>
<td>transferred on network</td>
</tr>
<tr>
<td>and distribution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fragmentation</td>
<td>PDBMS feature and the feature of parallel computer</td>
<td>Application requirements,</td>
</tr>
<tr>
<td>in accordance with</td>
<td>system used, combining application requirements.</td>
<td>combining the feature of DDBMS used.</td>
</tr>
<tr>
<td>Distribution</td>
<td>On multi disks of a parallel computer</td>
<td>On multi sites in the network</td>
</tr>
<tr>
<td>mode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3) Federated 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.
FS = CS + IS
FS is all of the data available for the users on site i.
IS is gained through the negotiation with ES j of other sites (j  i).
User’s query on FS i ⇒ the sub-queries on CS i and IS i ⇒ the sub-queries on corresponding ES j.
The results gained from ES j ⇒ the result forms of corresponding IS i and combined with the results get from the sub-queries on CS i, then synthesized to the eventual result form of FS i.

4.6 The Distribution of Catalog
Catalog — Data about data (Metadata).
Its main function is to transfer the user’s operating demands to the physical targets in system.

4.6.1 Contents of Catalogs
(1) The type of each data object (such as base table, view, . . .) and its schema
(2) Distribution information (such as fragment location, . . .)
(3) Access routing (such as index, . . .)
(4) Grant information
(5) Some statistic information used in query optimization

(1)–(4) are not changed frequently, while (5) will change on every update operation. For it:
- Update it periodically
- Update it after every update operation

4.6.2 The features of catalog
(1) mainly read operation on it
(2) very important to the efficiency and the data distribution transparency of the system
(3) very important to site autonomy
(4) the distribution of catalog is decided by the architecture of DDBMS, not by application requirements
4.6.3 Distribution strategies of catalog

(1) Centralized
   - A complete catalog stored at one site.
   - Extended centralized catalog: centralized at first; saved after being used; notify while there is update

(2) Fully replicated: catalogs are replicated at each site. Simple in retrieval. Complex in update. Poor in autonomy.

(3) Local catalog: catalogs for local data are stored at each site. That means catalogs are stored along with data.

If want the catalog information about data on other site (look for through broadcast):

- Master catalog: store a complete catalog on some site. Make every catalog information has two copies.
- Cache: after getting and using the catalog information about data on other site through broadcast, save it for future use (cache it). Update the catalog cached through the comparison of version number.

R

R's catalog (Ver No.) Cache of R's catalog (Ver No.)

(4) Different Combination of the above
   Use different strategy to different contents of catalog, and then get different combination strategies. Such as:
   a. Use fully replicated strategy to distribution information (2), use local catalog strategy to other parts.
   b. Use local catalog strategy to statistic information, use fully replicated strategy to other parts.
4.6.4 Catalog management in R*

--- Site autonomy

- Characteristics:
  - There is no global catalog
  - Independent naming and data definition
  - The catalog grows reposefully
- The most important concept — System Wide Name (SWN)
  
  \[
  \text{<SWN>::=User@UserSite.ObjectName@BirthSite}
  \]

  - 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.

Name Resolution — Mapping PN to SWN

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

<table>
<thead>
<tr>
<th>ObjectName</th>
<th>SWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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) UserObjectName: search the synonym table of user “User” on current site.
4) User@UserSite.ObjectName
5. Query Optimization

Introduction

- “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.
5.1 Summary of Query Optimization in DDBMS

- **Global Query**: a query over global relation.
- **Fragment Query**: a query over fragments.

**General flow**

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

Execute each operation according to the schedule in query plan

**Example**

S(SNUM, SNAME, CITY)
SP(SNUM, PNUM, QUAN)
P(PNUM, PNAME, WEIGHT, SIZE)

Suppose the fragmentation is as following:

S1 = \( \sigma_{\text{CITY}=\text{Nanjing}}(S) \)
S2 = \( \sigma_{\text{CITY}=\text{Shanghai}}(S) \)
SP1 = SP \bowtie S1
SP2 = SP \bowtie S2
Global Query:

```sql
SELECT SNAME
FROM S, SP, P
WHERE S.SNUM=SP.SNUM AND
SP.PNUM=P.PNUM AND
S.CITY='Nanjing' AND
P.PNAME='Bolt' AND
SP.QUAN>1000;
```

Query tree

1. \( S \times P \) on \( \sigma_{P.NAME='Bolt'} \)
2. \( S \times P \) on \( \sigma_{QUAN>1000} \)
3. \( S \) on \( \sigma_{CITY='Nanjing'} \)

After equivalent transform (Algebra optimization):

1. \( S' \times P' \) on \( \sigma_{P.NAME='Bolt'} \)
2. \( S' \times P' \) on \( \sigma_{QUAN>1000} \)
3. \( S' \) on \( \sigma_{CITY='Nanjing'} \)
5.2 The Equivalent Transform of a Query

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: \( \Pi_{\text{NAME}, \text{DEPT}}(\text{EMP}) \equiv \sigma_{\text{DEPT}=15}(\text{EMP}) \)

(1) Query tree

For example: \( \Pi_{\text{SUPPLY,DEPT}}(\text{DEPT} = '\text{North}'\bowtie\text{SUPPLY,DEPT}) \)

Leaves: global relation

Middle nodes: unary/binary operations

Leaves \( \rightarrow \) root: the executing order of operations
(2) The equivalent transform rules of relational algebra

1) Exchange rule of $\bowtie$: $E_1 \times E_2 \equiv E_2 \times E_1$

2) Combination rule of $\bowtie$: $E_1 \bowtie (E_2 \bowtie E_3) \equiv (E_1 \bowtie E_2) \bowtie E_3$

3) Cluster rule of $\Pi$: $\Pi_{A_1 \ldots A_n}(\Pi_{B_1 \ldots B_m}(E)) \equiv \Pi_{A_1 \ldots A_n}(E)$, legal when $A_1, \ldots, A_n$ is the sub set of $\{B_1, \ldots, B_m\}$

4) Cluster rule of $\sigma$: $\sigma_{F_1}(E) \equiv \sigma_{F_1 \land F_2}(E)$

5) Exchange rule of $\sigma$ and $\Pi$: $\Pi_{A_1 \ldots A_n}(\sigma_{F}(E)) \equiv \Pi_{A_1 \ldots A_n}(\sigma_{F}(E))$

6) If the attributes in $F$ are all the attributes in $E_1$, then $\sigma_{F}(E_1 \bowtie E_2) \equiv \Pi_{A_1 \ldots A_n}(E_1 \bowtie E_2)$

7) $\sigma_{F}(E_1 \cup E_2) \equiv \sigma_{F}(E_1) \cup \sigma_{F}(E_2)$

8) $\sigma_{F}(E_1 - E_2) \equiv \sigma_{F}(E_1) - \sigma_{F}(E_2)$

9) $\Pi_{A_1 \ldots A_n}(E_1 \cup E_2) \equiv \Pi_{A_1 \ldots A_n}(E_1) \cup \Pi_{A_1 \ldots A_n}(E_2)$

10) $\Pi_{A_1 \ldots A_n}(E_1 \cup E_2) \equiv \Pi_{A_1 \ldots A_n}(E_1) \cup \Pi_{A_1 \ldots A_n}(E_2)$

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.
5.3 Transform Global Queries into Fragment Queries

- **Methods:**
  - For horizontal fragmentation: \( R = R_1 \cup R_2 \cup \ldots \cup R_n \)
  - For vertical fragmentation: \( S = S_1 \bowtie S_2 \bowtie \ldots \bowtie S_n \)

  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:**
    \[
    \Pi_{\text{emp.name}}(\text{emp} \bowtie (\sigma_{\text{mgrnum}=373} \text{dept})) - (\sigma_{\text{sal}>35000} \text{emp} \bowtie (\sigma_{\text{mgrnum}=373} \text{dept}))
    \]

- **Properties:**
  - \( R \bowtie R \equiv R \)
  - \( R \cup R \equiv R \)
  - \( R \bowtie F \equiv \sigma_F(R) \)
  - \( R \cup \sigma_F(R) \equiv R \)
  - \( R \bowtie \sigma_{F_1}(R) \equiv \sigma_{F_1 \land F_2}(R) \)
  - \( \sigma_{F_1}(R) \bowtie \sigma_{F_2}(R) \equiv \sigma_{F_1 \land F_2}(R) \)
  - \( \sigma_{F_1}(R) \cup \sigma_{F_2}(R) \equiv \sigma_{F_1 \lor F_2}(R) \)
  - \( R \bowtie \sigma_{F_1}(R) \equiv \sigma_{F_1 \land \neg F_2}(R) \)
Database Management Systems and Their Implementation, Xu Lizhen

3) Find and eliminate the empty sub-expression

Example:

\[ \bigcup R_1 \bowtie R_2 \sigma _{deptnum=1} \]
\[ deptnum \leq 10 \]  \[ 10 < deptnum \leq 20 \]  \[ deptnum > 20 \]

4) Eliminate useless vertical fragments

If \( A \cup \text{Attr}(F) \subseteq \text{Attr}(R_1) \)

5.4 Query Decomposition

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:

\[ M = \bigcup R_1 \bowtie R_2 \sigma _{deptnum=1} \]

Suppose: \( R_i \) — fragment \( R_i \) which stored on site \( j \).
Decomposition method:

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.

5.5 The Optimization of Binary Operations

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.

I. Main problems in global optimization

1) Materialization: select suitable copies of fragments which involved in the query
2) Strategies for join operation
   - Non_distributed join: \((R^2 \cup R^3) \bowtie R^1\)
   - Distributed join: \((R^1 \bowtie R^2) \cup (R^3 \bowtie R^3)\)
3) Select execution of each operation (mainly to direct join)
Nested loop: one relation acts as outer loop relation (O), the other acts as inner loop relation (I). For every tuple in O, scan I one time to check join condition. Because the relation is accessed from disk in the unit of block, we can use block buffer to improve efficiency. For \( R \bowtie S \), if let \( R \) as O, S as I, \( b_R \) is physical block number of \( R \), \( b_S \) is physical block number of \( S \), there are \( n_B \) block buffers in system, \( n_B \geq 2 \), and \( n_B-1 \) buffers used for O, one buffer used for I, then the total disk access times needed to compute \( R \bowtie S \) is:

\[
b_R + \left\lceil \frac{b_R}{n_B - 1} \right\rceil \times b_S
\]

Merge scan: order the relation \( R \) and \( S \) on disk in ahead, then we can compare their tuples in order, and both relation only need to scan one time. If \( R \) and \( S \) have not ordered in ahead, must consider the ordering cost to see if it is worth to use this method (p122). Using index or hash to look for mapping tuples: in nested loop method, if there is suitable access route on I (say \( B^+ \) tree index), it can be used to substitute sequence scan. It is best when there is cluster index or hash on join attributes.

Hash join: because the join attributes of \( R \) and \( S \) have the same domain, \( R \) and \( S \) can be hashed into the same hash file using the same hash function, then \( R \bowtie S \) can be computed based on the hash file.

The above are all classical algorithms in centralized DBMS. The idea is similar when computing join in DDBMS using direct join strategy.

II. Three general optimization methods:
1) By cost comparison (also called exhaustive search)
2) By heuristic rule: generally used in small systems. (p124)
3) Combination of 1, 2: eliminate the solutions unsuitable obviously by using 2 to reduce solution space, then use 1 to compare cost of the rest solutions carefully.

III. Cost estimation
Total query cost = Processing cost + Transmission cost
According to different environment:

- For wide area network: the transfer rate is about 100bps~50Kbps, far slow than processing speed in computer, so Processing cost can be omitted.
- For local area network: the transfer rate will reach 1000Mbps, both items should be considered.

1) Transmission cost

\[ \text{TC}(x) = C_0 + C_1 x \]

- \(x\): 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.

2) Processing cost

Processing cost = \(\text{cost}_{\text{cpu}} + \text{cost}_{\text{I/O}}\)

- \(\text{cost}_{\text{cpu}}\) can be omitted generally.
- \(\text{cost}_{\text{I/O}}\)
  - 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 (\(\mu\)s, can be omitted)

\[ \text{cost}_{\text{I/O}} = \text{no. of I/O} \times D_0 \]

- 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.

5.6 Implement Join Operation With Semi_join

I. The role of semi_join

- Semi_join is used to reduce transmission cost. So it is suitable for WAN only.
- \(R \bowtie S = \Pi_{A}(R \bowtie S)\)
  - if \(R\) and \(S\) are stored on site 1 and 2 respectively, the steps to realize \(R \bowtie S\) with \(\bowtie\) is as following:
  1) Transfer \(\Pi_{A}(S)\) to site1. \(A\) is join attribute
  2) Execute \(R \bowtie \Pi_{A}(S) = R \bowtie S\) on site1 (compress R)
  3) Transfer \(R \bowtie S\) to site2.
  4) Execute \((R \bowtie S) \bowtie S = R \bowtie S\) on site2
Cost comparison

- Cost of direct join = \( C_0 + C_1 \min(r, s) \)  
  \( r, s \rightarrow |R|, |S| \) (size of the relations)

- Cost of join via semi_join = 
  \[
  \min(2C_0 + C_1 s' + C_1 r'', 2C_0 + C_1 r' + C_1 s'') = \min(s' + r'', r' + s'')
  \]
  \( s', r' \rightarrow |\Pi_A(S)|, |\Pi_A(R)| \) 
  \( s'', r'' \rightarrow |S \bowtie R|, |R \bowtie S| \)

- Only when \( \frac{\text{2}}{\text{1}} < 1 \), 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

II. Comments on semi_join

1. The reduce on transmission cost through \( \bowtie \) is gained through the sacrifice on processing cost.
2. There are many candidate solutions of semi_join. For example: for the query \( R_1 \bowtie R_2 \bowtie R_3 \ldots \bowtie R_n \)
   consider the \( \bowtie \) to \( R_0 \), maybe:
   \( R_1 \bowtie R_2, R_1 \bowtie (R_2 \bowtie R_3), R_1 \bowtie (R_2 \bowtie R_3) \bowtie R_4, \ldots \)
   it is almost impossible to select the best from all possible solutions.
3. Bernstein’s remark
   \( \bowtie \) 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) \( \in \text{RED}(Q, R) \)
(2) reduce \( R \) mostly

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

Example 1: Q is a query with qualification:
\[
q = (R_1.A=R_2.B) \land (R_1.C=R_3.D) \land (R_2.E=R_4.F) \land \\
\]
Query graph:

Link the two relations with a line if there is $\bowtie$ between them, then we can get the query graph. The query whose query graph like the left graph is called tree query (TQ).

Example 2: $q = (R_1.A=R_2.B) \land (R_2.C=R_3.D) \land (R_3.E=R_1.F)$

The query whose query graph like the left graph is called cyclic query (CQ).

Example 3: $q = (R_1.A=R_2.B) \land (R_2.B=R_3.C) \land (R_3.C=R_1.A)$

This is a TQ, not a CQ, because $R_3.C=R_1.A$ can be obtained from transfer relation, it is not a independent condition.

Can be proved:

1) Full reducer exists for TQ.
2) No full reducer exists for CQ in many cases.

$\begin{array}{|c|c|}
R_1 & R_2 & R_3 \\
\hline
A & C & D \\
\end{array}$

$q = (R_1.B=R_2.C) \land (R_2.D=R_3.E) \land (R_3.F=R_1.A)$

Even if the result of this query is empty, the size of any one of $R_1, R_2, R_3$ can not be decreased through $\bowtie$. So there is not full reducer for this query.

Another example about full reducer of CQ

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

$\begin{array}{|c|c|}
R & S & T \\
\hline
A & B & C \\
\end{array}$

$q' = R \bowtie T = A B C \land S \bowtie T = B C D \land T' = T \bowtie S = C A A$
R"\'=R\' \bowtie T'=\Phi. So the full reducer of relation R in
query q is:
① R'=R \bowtie T T S'=S \bowtie R' ③ T'=T \bowtie S' ④ R"=R' \bowtie T'
⑤ S"=S' \bowtie R" \bowtie T' ⑥ R"'=R'' \bowtie T'' = \Phi

Conclusion:
- For CQ: there is no full reducer in general. Even if
  there is, its length increases linearly along with the
  number of tuples of some relation in the query. (about
  3(m-1))
- For TQ: the length of full reducer < n-1 (n is the
  number of nodes in the query graph).
- So the full reducer can not be the optimization target
  when execute join operation under distributed
  environment through \bowtie.

5.7 Direct Join
--- Implementation method of join operation in R*

I. Two basic implementation method of join operation
- Nested Loop: the extension of corresponding
  algorithm in centralized DBMS
  \[\text{cost} = (b_R + b_S/(n_B - 1)) * b_S + D_0\]
- Merge Scan: the extension of corresponding
  algorithm in centralized DBMS
  \[\text{cost} = (b_R + b_S) * D_0 + \text{Cost}_{\text{sort}}(R) + \text{Cost}_{\text{sort}}(S)\]

II. Transmission of relations in these two methods

1) "shipped whole": The whole relation is shipped
   without selections.
   - For I, a temporary relation is established at the
     destination site for further use.
   - For O, the relation doesn’t need storing.
2) "Fetch as need": The whole relation is not shipped.
   The tuples needed by the remote site are sent at its
   request. The request message usually contains the value
   of join attribute.
   Usually there is an index on the join attribute at the
   requested site.
III. Six implementation strategies of join in $R^*$

- Nested Loop
  - $O$: shipped whole
  - $I$: fetch as needed
- Merge Scan
  - $O$: shipped whole
  - $I$: shipped whole
- 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 $R_i \cup R_j$ according to some attribute set, iff $G_i \subseteq R_i \cup R_j \forall i, j$ --- (SNC), then:

$$GB_{G_i}(R_i \cup R_j) = GB_{G_i}(R_i) \cup GB_{G_i}(R_j)$$

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.
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, ..., S_n
then:
SUM(S) = SUM(SUM(S_1), SUM(S_2), ..., SUM(S_n))
COUNT(S) = SUM(COUNT(S_1), COUNT(S_2), ..., COUNT(S_n))
AVG(S) = SUM(S)/COUNT(S)
MIN(S) = MIN(MIN(S_1), MIN(S_2), ..., MIN(S_n))
MAX(S) = MAX(MAX(S_1), MAX(S_2), ..., MAX(S_n))

5.9 Update Strategies

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.

6. Recovery

Read: master copy & snapshots 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.
6.1 Introduction

The main roles of recovery mechanism in DBMS are:
1) Reducing the likelihood of failures (prevention)
2) Recover from failures (solving)

*Restore DB to a consistent state after some failures.*

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

1) Periodical dumping

- Variation: Backup + Incremental dumping
- I.D --- updated parts of DB

2) Backup + Log

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

Change: \[ \text{Old value (before image --- B.I) } \rightarrow \text{Recorded} \rightarrow \text{New value (after image --- A.I) } \]

For
- update op.: B.I A.I
- insert op.: ---- A.I
- delete op.: B.I ----

Backup Log \[
\begin{array}{c}
\downarrow \\
\text{failure}
\end{array}
\]
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.

3) Multiple Copies

There are multi copies for every data object. Recovered with other copies while failure occurs. The system cannot be collapsed because of some copy’s failure.

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 for Centralized DBMS.

6.2 Transaction

A transaction T is a finite sequence of actions on DB exhibiting the following effects:
- Atomic action: Nothing or All.
- Consistency preservation: consistency state of DB → another consistency state of DB.
- Isolation: concurrent transactions should run as if they are independent each other.
- Durability: The effects of a successfully completed transaction are permanently reflected in DB and recoverable even failure occurs later.
Example: transfer money \( s \) from account A to account B

Begin transaction
read A
\( A := A - s \)
if \( A < 0 \) then Display "insufficient fund"
   Rollback /*undo and terminate */
else  \( B := B + s \)
   Display "transfer complete"
   Commit /*commit the update and terminate */

Rollback --- abnormal termination. (Nothing)
Commit --- normal termination. (All)

---

6.3 Some Structures to Support Recovery

Recovery information (such as \( \log \)) should be stored in nonvolatile storage. The following information need to be stored in order to support recovery:

1) Commit list: list of TID which have been committed.
2) Active list: list of TID which is in progress.
3) Log:

<table>
<thead>
<tr>
<th>BID</th>
<th>TID</th>
<th>After image</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BID</th>
<th>TID</th>
<th>Before image</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The reliability demand of \( \log \) is higher than general data, often doubled.
If the interval between two dumps is too long, lack of storage space may occur because of the accumulation of \( \log \). Solutions:

1) Free the space after commit. It will be impossible to recover from disk failure in this situation.
2) Periodically dump to tape.
3) \( \log \) compression:
   - Don’t have to store \( \log \) information for aborted transactions
   - B.I are no longer needed for committed transactions
   - Changes can be consolidated, keep the newest A.I only.
6.4 Commit Rule and Log Ahead Rule

6.4.1 Commit Rule
A.I must be written to nonvolatile storage before commit of the transaction.

6.4.2 Log Ahead Rule
If A.I is written to DB before commit then B.I must first written to log.

6.4.3 Recovery strategies
(1) The features of undo and redo (are idempotent):
undo(undo(undo(...undo(x)...))) = undo(x)
redo(redo(redo(...redo(x)...))) = redo(x)

(2) Three kinds of update strategy

a) A.I → DB before commit
   TID → active list
   ✓ B.I → Log  (Log Ahead Rule)
   ✓.A.I → DB
   ✓ TID → commit list
   ✓ delete TID from active list

The recovery after failure in this situation
Check two lists for every TID while restarting after failure:

<table>
<thead>
<tr>
<th>Commit list</th>
<th>Active list</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓ undo, delete TID from active list</td>
</tr>
<tr>
<td>✓</td>
<td>✓ delete TID from active list</td>
</tr>
<tr>
<td>✓</td>
<td>✓ nothing to do</td>
</tr>
</tbody>
</table>
b) A.I→DB after commit
   TID→active list
   TID→commit list
   A.I→DB
   delete TID from active list

The recovery after failure in this situation

Check two lists for every TID while restarting after failure:

<table>
<thead>
<tr>
<th>Commit list</th>
<th>Active list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓ delete TID from active list</td>
</tr>
<tr>
<td>✓</td>
<td>✓ redo, delete TID from active list</td>
</tr>
<tr>
<td>✓</td>
<td>✓ nothing to do</td>
</tr>
</tbody>
</table>

c) A.I→DB concurrently with commit
   TID→active list
   \[ A.I, B.I \rightarrow \text{Log} \] (Two Rules)
   \[ A.I \rightarrow \text{DB} \] (partially done)
   TID→commit list
   A.I→DB (completed)
   delete TID from active list
The recovery after failure in this situation

Check two lists for every TID while restarting after failure:

<table>
<thead>
<tr>
<th>Commit list</th>
<th>Active list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓ undo, delete TID from active list</td>
</tr>
<tr>
<td>✓</td>
<td>✓ redo, delete TID from active list</td>
</tr>
<tr>
<td>✓</td>
<td>✓ nothing to do</td>
</tr>
</tbody>
</table>

Conclusion:

<table>
<thead>
<tr>
<th></th>
<th>redo</th>
<th>undo</th>
</tr>
</thead>
<tbody>
<tr>
<td>a)</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>b)</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>c)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>d)</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

6.5 Update Out of Place

- 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.
6.6 Recovery Procedures

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
### 6.7 System Start Up

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.

### 6.8 Two Phase Commit

- 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.

\[
\begin{align*}
A & \text{March } i \\
\downarrow & \text{OK } i \\
B & \uparrow \\
\downarrow & \text{March } i+1 \\
\uparrow & \text{OK } i+1 \\
\end{align*}
\]

A sends March i+1 again, for B:
1) B has received OK i+1:
2) OK i+1 has been lost

\[
\begin{align*}
A & \text{March } i+1 \\
\downarrow & \text{OK } i+1 \\
\downarrow & \text{March } i+1 \\
\end{align*}
\]

If A has not received OK i+1:
1) B had not received March i+1
2) OK i+1 has been lost

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

Diagram:

- Coordinator
- Participant
- OK
**Two Phase Commit**

- **Coordinator**
  - Broadcast prepare
  - OK / not OK
  - Commit/Abort
  - Ack

- **Participate**
  - If the sub-transaction succeeds 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 participate answer OK, the participate have to wait, and is in blocked state. This is the disadvantage of 2PC.

**Three Phase Commit**

- **Coordinator**
  - Broadcast prepare
  - Ready / Abort

- **Participate**
  - If the sub-transaction succeeds and can commit, it answers Ready, or Abort
  - Broadcast to all participants and this MSG is recorded on the disk

  - OK
  - Commit/Abort
  - Ack

- 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.

7. Concurrency Control
7.1 Introduction

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(x)</td>
<td></td>
<td>Read(x)</td>
<td></td>
<td></td>
<td>Write(x)</td>
</tr>
<tr>
<td>x:=x+1</td>
<td></td>
<td>Write(t)</td>
<td></td>
<td></td>
<td>(rollback)</td>
</tr>
<tr>
<td>Write(x)</td>
<td></td>
<td>Read(t[y])</td>
<td></td>
<td></td>
<td>Read(s)</td>
</tr>
<tr>
<td>t</td>
<td></td>
<td>Write(s)</td>
<td></td>
<td></td>
<td>Unrepeatable read</td>
</tr>
</tbody>
</table>

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.

7.1.3 Serialization --- the criterion for concurrency consistency

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 \( \rightarrow \) different equivalent serial execution \( \rightarrow \) different result? (yes, n!)

\[
T_A \quad T_B \quad T_C \\
<table>
<thead>
<tr>
<th>Read R1</th>
<th>Write R1</th>
<th>Write R2</th>
</tr>
</thead>
</table>
| 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 \( T_A \rightarrow T_B \rightarrow T_C \).

\]
### 7.1.4 View equivalent and conflict equivalent

**Schedules** — 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.

---

#### Example Schedules

Let $T_1$ transfer $50$ from $A$ to $B$, and $T_2$ transfer 10% of the balance from $A$ to $B$. The following is a serial schedule, in which $T_1$ is followed by $T_2$.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>$A := A - 50$</td>
</tr>
<tr>
<td>write($A$)</td>
<td>$B := B + 50$</td>
</tr>
<tr>
<td>read($B$)</td>
<td>write($B$)</td>
</tr>
<tr>
<td>$t$</td>
<td>$t$</td>
</tr>
</tbody>
</table>

Example Schedules

Let $T_1$ and $T_2$ be the transactions defined previously. The following schedule is not a serial schedule, but it is equivalent to the above.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>$A := A - 50$</td>
</tr>
<tr>
<td>write($A$)</td>
<td>$B := B + 50$</td>
</tr>
<tr>
<td>read($B$)</td>
<td>write($B$)</td>
</tr>
<tr>
<td>$t$</td>
<td>$t$</td>
</tr>
</tbody>
</table>
Let $S$ and $S'$ be two schedules with the same set of transactions. $S$ and $S'$ are **view equivalent** 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 $S$ can be transformed into a schedule $S'$ by a series of swaps of non-conflicting operations, we say that $S$ and $S'$ are **conflict equivalent**.

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

Serialization can be divided into **view serialization** and **conflict serialization**.

**Example 1**: for the schedule $s$ of transaction set $\{T_1,T_2,T_3\}$ $s = R_2(x)W_3(x)R_1(y)W_2(y) → R_1(y)R_2(x)W_2(y)W_3(x) = s'$ $s$ 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.
7.1.5 Preceding graph

Directed graph \( G = <V,E> \)

\( V \) — set of vertexes, including all transactions participating in schedule.

\( E \) — set of edges, decided through the analysis of conflict operations. If any of the following conditions is fulfilled, add an edge \( T_i \rightarrow T_j \) into \( E \):
- \( R_i(x) \) precedes \( W_j(x) \)
- \( W_i(x) \) precedes \( R_j(x) \)
- \( W_i(x) \) precedes \( W_j(x) \)

Finally, check if there is cycle in the preceding graph. If there is cycle in it, the schedule is not serializable, or it is serializable.

Find equivalent serial execution while serialization

1) Because there is no cycle, there must be some vertexes whose in-degree is 0. Remove these vertexes and relative edges from the preceding graph, and store these vertexes into a queue.

2) Process the left graph in the same way as above, but the vertexes removed should be stored behind the existing vertexes in the queue.

3) Repeat 1 and 2 until all vertexes moved into the queue.

**Example:** for schedule \( s \) on \( \{T_1,T_2,T_3,T_4\} \), suppose:

\[ s = W_3(y)R_1(x)R_2(y)W_3(x)W_2(x)W_1(z)R_4(z)W_4(x) \]

Is it serializable? Find out the equivalent serial execution if it is.

The commission of concurrency control is to enforce the concurrent transactions executed in a serializable schedule.
7.2 Locking Protocol

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:

<table>
<thead>
<tr>
<th></th>
<th>NL</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

7.2.2 (S,X) locks

S lock --- if read access is intended.

X lock --- if update access is intended.

---

*Two Phase Locking*

- **Definition1:** 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.
- **Definition2:** In a transaction, if it first acquires a lock on the object before operating on it, it is called well-formed.
- **Theorem:** If S is any schedule of well-formed and two phase transactions, then S is serializable. (proving is on p151)

Conclusions:

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.
### 7.2.3 (S,U,X) locks

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.

<table>
<thead>
<tr>
<th></th>
<th>NL</th>
<th>S</th>
<th>U</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>S</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>U</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

### 7.3 Deadlock & Live Lock

**Deadlock:** 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.

- 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)

### 7.3.1 Deadlock Detection

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
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 "dies", 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 "wound" $T_B$, i.e. $T_B$ is aborted and automatically retried with original timestamp.

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

7.4 Lock Granularities

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
7.4.2 Intention 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

Compatibility matrix while lock in multi granularities:

<table>
<thead>
<tr>
<th></th>
<th>NL</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>IS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>IX</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>S</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SIX</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

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

Locking Rules:

- Locks are requested from root to leaves and released from leaves to root.
- Example:
  - Locking order: DB, file, record
  - (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
7.5 Locking on Index (B+ Tree)

- Transactions access concurrently not only the data in DB, but also the indexes on these data, and apply operations on indexes, such as search, insert, delete, etc. So indexes also need concurrency control while multi granularity locking is supported.
- How can we efficiently lock a particular leaf node?
  - Btw, don’t confuse this with multiple granularity locking!
  - One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL.
- This has terrible performance!
  - Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root.

Some Useful Observations

- Every access to B+ tree need a traverse from root to some leaf. Only leaves have detail information about data, such as TID, while higher levels of the tree only direct searches for leaf pages.
- One node occupies one page generally, so the lock unit of B+ tree is page. Don’t need multi granularity locks, only S, X lock on page level are enough.
- B+ tree is key resource accessed frequently, liable to be the bottleneck of system. Performance is very important in index concurrency control.
- If occur conflict while traverse the tree, discard all locks applied, and search from root again after some delay. Avoid deadlock resulted by wait.

Some Useful Observations

- Originally, locks on index are only used to keep the consistency of index itself. The correctness of concurrent transactions is responsible by 2PL. From this sense, the locks on index don’t need keeping to EOT, they can release immediately after finish the mapping from attribute value to tuples’ addresses. But in 7.6 we will know, even the strict 2PL has leak while multi granularity locks are permitted. The lock on leaf of B+ tree should be kept to EOT in order to make up the leak of strict 2PL in this situation, while locks on other nodes of the tree can be released after finishing of search.
Tree Locking Algorithm

1. While traversing, apply S lock on root first, then apply S lock on the child node selected. Once get S lock on the child, the S lock on parent can be released, because traversing can’t go back. Search like this until arrive leaf node. After traversing, only S lock on wanted leaf is left. Keep this S lock till EOT.

2. While inserting new index item, traverse first, find the leaf node where the new item should be inserted in. Apply X lock on this leaf node.
   - If it is not full, insert directly.
   - If it is full, split node according to the rule of B+ tree. While splitting, besides the original leaf, the new leaf and their parent should add X lock. If parent is also full, the splitting will continue.
   - In every splitting, must apply X lock on each node to be changed. These X locks can be released when the changes are finished.
   - After the all inserting process is completed, the X lock on leaf node is changed to S lock and kept to EOT.

3. While deleting an index item from the tree, the procedure is similar as inserting. Deleting may cause the combination of nodes in B+ tree. The node changed must be X locked first and X lock released after finishing change. The X lock on leaf node is also changed to S lock and kept to EOT.
### 7.6 Phantom and Its Prevention

- The **assumption** 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 \( \text{rating} = 1 \), and finds *oldest* sailor (say, \( \text{age} = 71 \)).
  - Next, T2 inserts a new sailor; \( \text{rating} = 1, \text{age} = 96 \).
  - T2 also deletes oldest sailor with \( \text{rating} = 2 \) (and, say, \( \text{age} = 80 \)), and commits.
  - T1 now locks all pages containing sailor records with \( \text{rating} = 2 \), and finds *oldest* (say, \( \text{age} = 63 \)).
  - No consistent DB state where T1 is “correct”!

### The Problem

- T1 implicitly assumes that it has locked the set of all sailor records with \( \text{rating} = 1 \).
  - 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: 
    \[
    \text{select s\#, average(grade) from SC group by s\#;}
    \]

### Index Locking

- If there is a dense index on the \( \text{rating} \) field, T1 should lock the index node containing the data entries with \( \text{rating} = 1 \) and keep it until EOT.
  - If there are no records with \( \text{rating} = 1 \), T1 must lock the index node where such a data entry would be, if it existed!
  - When T2 wants to insert a new sailor ( \( \text{rating} = 1, \text{age} = 96 \) ), he can’t get the X lock on the index node containing the data entries with \( \text{rating} = 1 \), 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 whole table, no new records can be added before T1 commit of course.
**Predicate Locking**

- Grant lock on all records that satisfy some logical predicate, e.g. \( \text{age} > 2 \times \text{salary} \).
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.
- What is the predicate in the sailor example?
- In general, predicate locking has a lot of locking overhead. It is almost impossible to realize it.

---

**7.7 Isolation Level of Transaction**

- Support for isolation level of transaction is added from SQL-92. Each transaction has an access mode, a diagnostics size, and an isolation level.
- **SET TRANSACTION statement**

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Possible result</th>
<th>Lock demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirty Read</td>
<td>Unrepeatable</td>
<td>Dirty</td>
</tr>
<tr>
<td>Uncommitted</td>
<td>Read</td>
<td>Maybe</td>
</tr>
<tr>
<td>Committed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Repeatable</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Serializable</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

- **Example:**

```
SET TRANSACTION READ ONLY
ISOLATION LEVEL REPEATABLE READ;
```

```
SET TRANSACTION ISOLATION LEVEL
{ READ COMMITTED | READ UNCOMMITTED | REPEATABLE READ | SERIALIZABLE }
```
7.8 Lock Mechanism in OODBMS

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

<table>
<thead>
<tr>
<th>NL</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
<th>RL</th>
<th>WL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>IS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>IX</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>S</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>SIX</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>RL</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>WL</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Locking steps of RL(WL) locks:
1) Add IS(IX) lock on any super class chain of this class and DB
2) Add RL(WL) on this class
3) Check top-down if there is lock conflicting with RL(WL) on the child classes of this class. If there is no conflict, add a RL(WL) lock on the child class who has multi parents met first.
4) If find any conflict in above, the lock application fail.
5) Locking complex object: lock referred object only when accessed.(according to general multi granularity lock protocol)
7.9 The Time Stamp Method

1. T.S --- A number generated by computer’s internal clock in chronological order.
2. T.S for a transaction --- the current T.S when the transaction initials.
3. T.S for an data object:
   1) Read time (tr) --- highest T.S possessed by any transaction to have read the object.
   2) Write time (tw) --- highest T.S possessed by any transaction to have written the object.
4. The key idea of T.S method is that the system will enforce the concurrent transactions to execute in the schedule equivalent with the serial execution according to T.S order.

Read/Write operations under T.S method

Let R --- a data object with T.S tr and tw.
T --- a transaction with T.S t.

Transaction T reads R:
read R
if (t >= tw)
   then /* OK */
      tr = Max (tr, t)
   else /* a transaction younger than T has already write R ahead of T, conflict */
      restart T with a new T.S

Transaction T writes R:
if (t >= tr)
   then if (t >= tw)
      then /* OK */
         write R
tw = t
   else /* tr <= t < tw */
      do nothing
   else /* a transaction younger than T has already read R ahead of T, conflict */
      restart T with a new T.S

No other reads
Remarks:

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.

7.10 Optimistic Concurrency Control Method

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”.

Three phases of transaction execution:

1. **Read phase**: read data from database and execute every kind of processing, but update operations only form update copies in memory.
2. **Validate phase**: check if the schedule of the transaction is serializable.
3. **Write phase**: if pass the check successfully, write the update copies in memory into DB and commit the transaction; Or throw away the update copies in memory and abort the transaction.
Information must be reserved:
1. Read set of each transaction
2. Write set of each transaction
3. The start and end time of each phase of each transaction

Checking method while transaction ends:
When transaction $T_i$ ends, only need to check if there is conflict among $T_i$ and the transactions which have committed and other transactions which are also in checking phase. The transactions which are in read phase don’t need to be considered.
Suppose $T_j$ is any transaction which has committed or is being checked, $T_i$ passes the check if it fulfills one of the following conditions for all $T_j$:
1. $T_j$ had finished write phase when $T_i$ began read phase, $T_j \rightarrow T_i$
2. The intersection of $T_i$’s read set and $T_j$’s write set is empty, and $T_i$ began write phase after $T_j$ finished write phase.
3. Both $T_i$’s read set and write set don’t intersect with $T_j$’s write set.
4. There is not any access conflict between $T_i$ and $T_j$.

7.11 Locking in DDBMS
The concurrency control in DDBMS is the same as that in centralized DBMS, demand concurrent transactions to be scheduled serializably. The problems in DDBMS are:
- Multi_copy
- Communication overhead
7.11.1 write lock all, read lock one
- Read $R$ —— S_lock on any copy of $R$
- Write $R$ —— X_lock all copies of $R$
- Hold the locks to EOT
Communication overhead: suppose $n$ —— No. of copies
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 lock MSG
- n lock grants
- n update MSG
- n ACK
- [n unlock MSG]

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

4n

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 deadlock is impossible to occur as long as n is an odd.

7.11.3 k-out-of-n locking

- 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.
### 7.11.4 Primary Copy Method

R---data object

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

Communication overhead:

<table>
<thead>
<tr>
<th>Write</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 lock MSG</td>
<td>1 lock MSG</td>
</tr>
<tr>
<td>1 lock grants</td>
<td>1 lock grants</td>
</tr>
<tr>
<td>n update MSG</td>
<td>1 read MSG</td>
</tr>
<tr>
<td>n ACK</td>
<td>2</td>
</tr>
</tbody>
</table>

2n+1

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).

### 7.11.5 Global Deadlock

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

**Global wait-for graph:** 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→T into the graph; if T is the tail of wait-for chain of current site, add T→EXT into the graph.

**Processing method of global wait-for graph:**

1. Check other sites if has: EXT→T₁→...→Tₖ→EXT
2. if Tₖ=T₁: global deadlock is detected.
3. if Tₖ≠T₁: merge two wait-for graphs:
   - EXT→T₁→...→Tₖ→T₁→...→Tₖ→EXT
4. Repeat step 1 and 2, check if Tₖ will result in global dead lock like T₁ 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 Time Stamp Technique in DDBMS

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:

\[ t_{at\ receipt\ site} := \max(t_1, t_2) \]

- \( t_1 \) --- current T.S at receipt site
- \( t_2 \) --- T.S of MSG

7.12.2 Read and write operations

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.