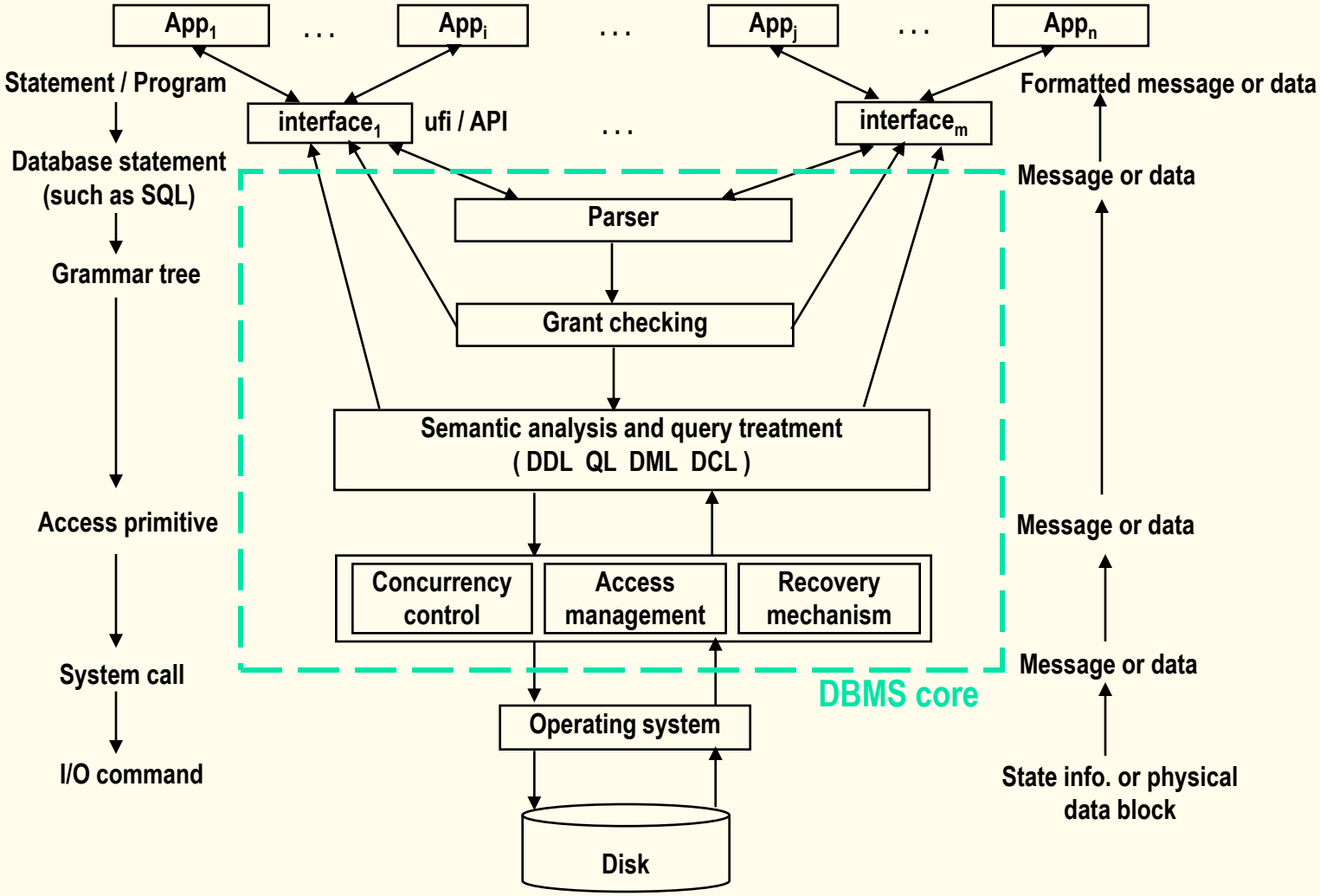
4. Database Management Systems*

Contents

- The Architecture of DBMS
  - The components of DBMS core
  - The process structure of DBMS
- Database Access Management
- Query Optimization
- Transaction Management
  - Recovery
  - Concurrent Control

4.1 The Components of DBMS Core

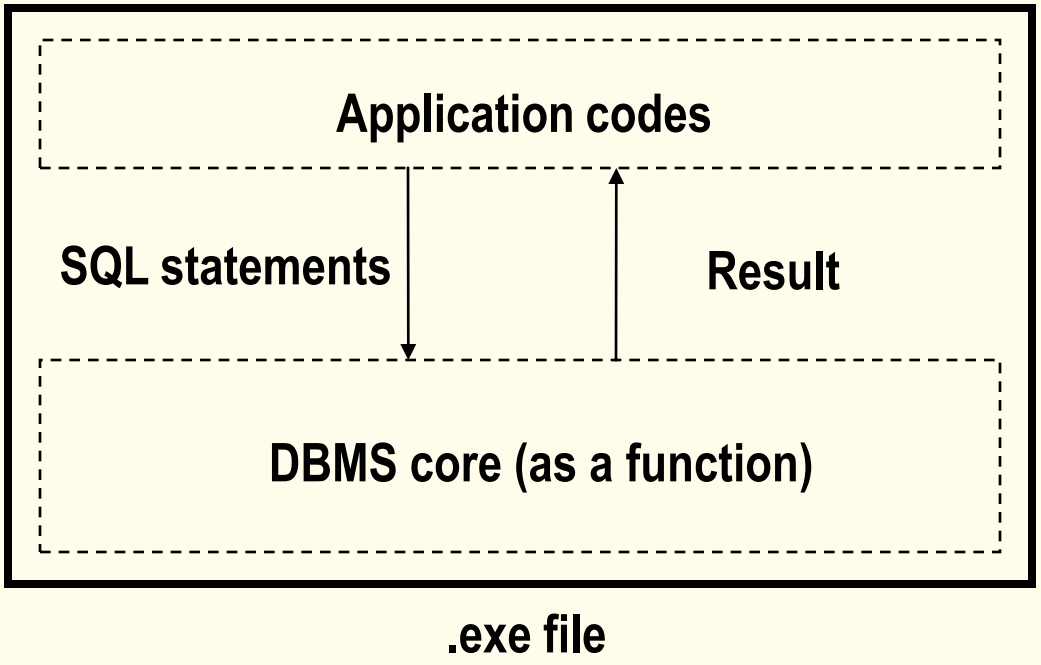


4.2 The Process Structure of DBMS

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

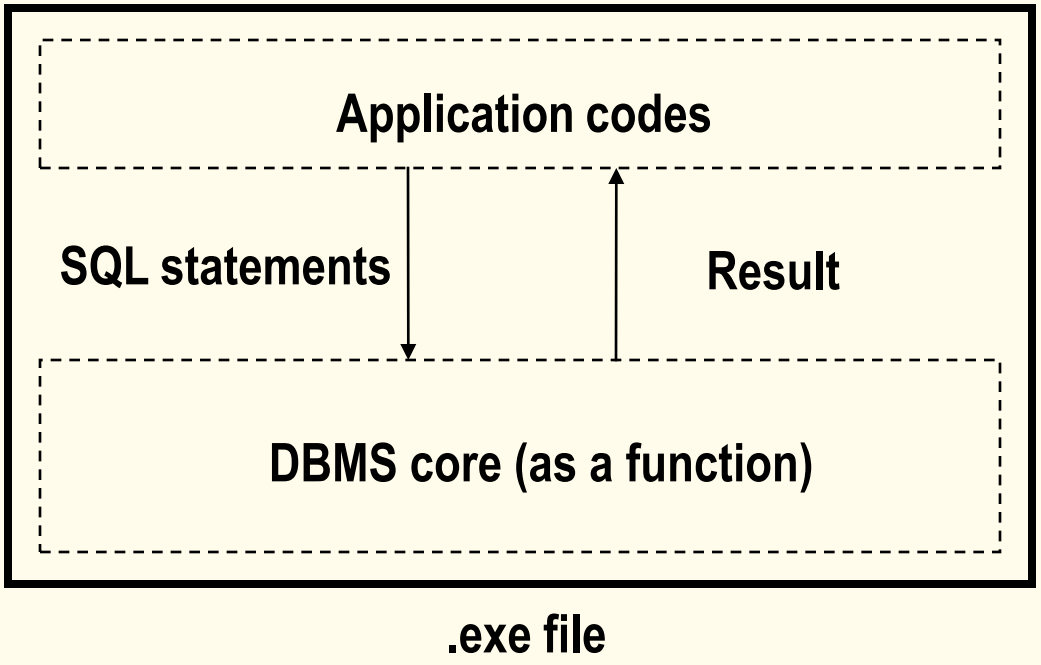
Single process structure

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



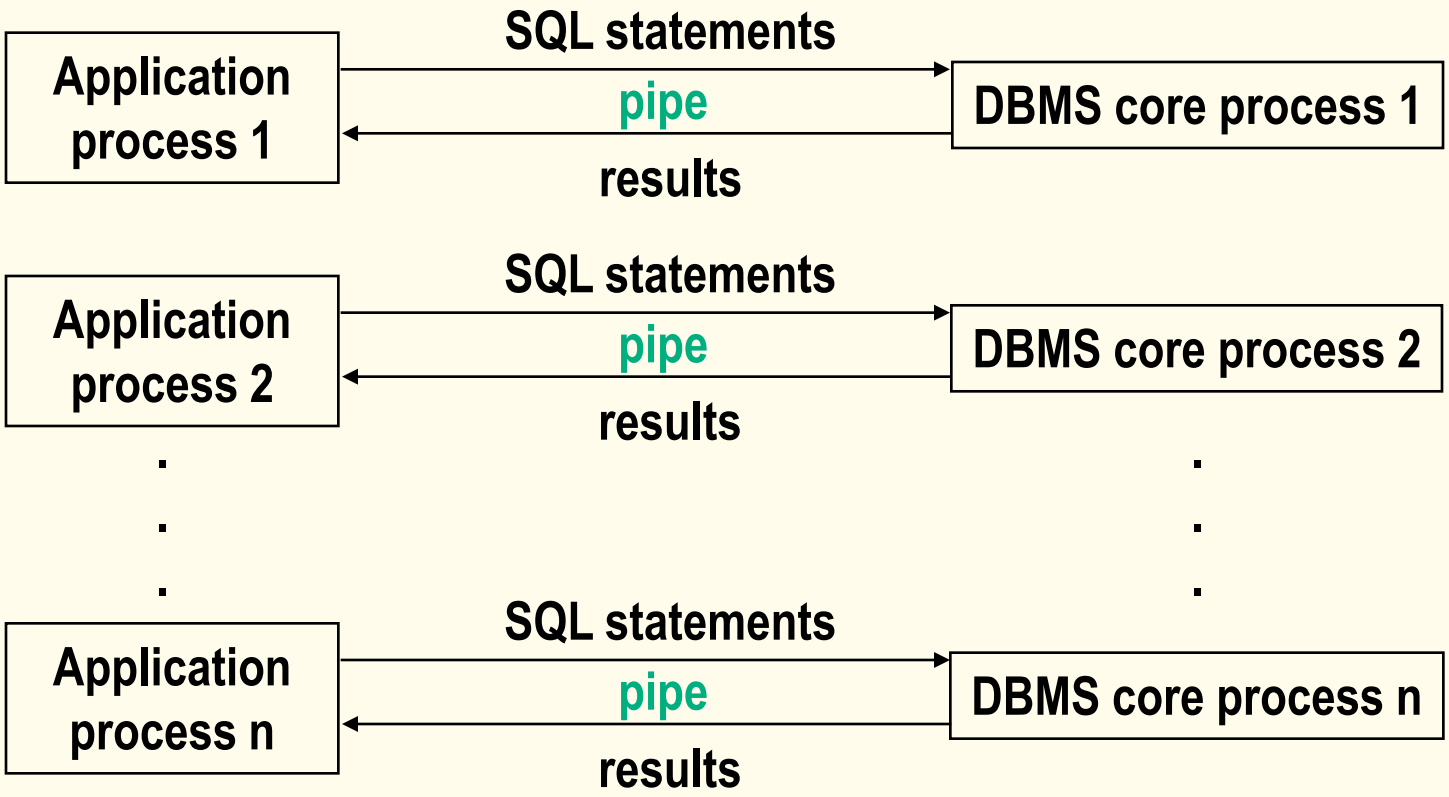
Single process structure

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



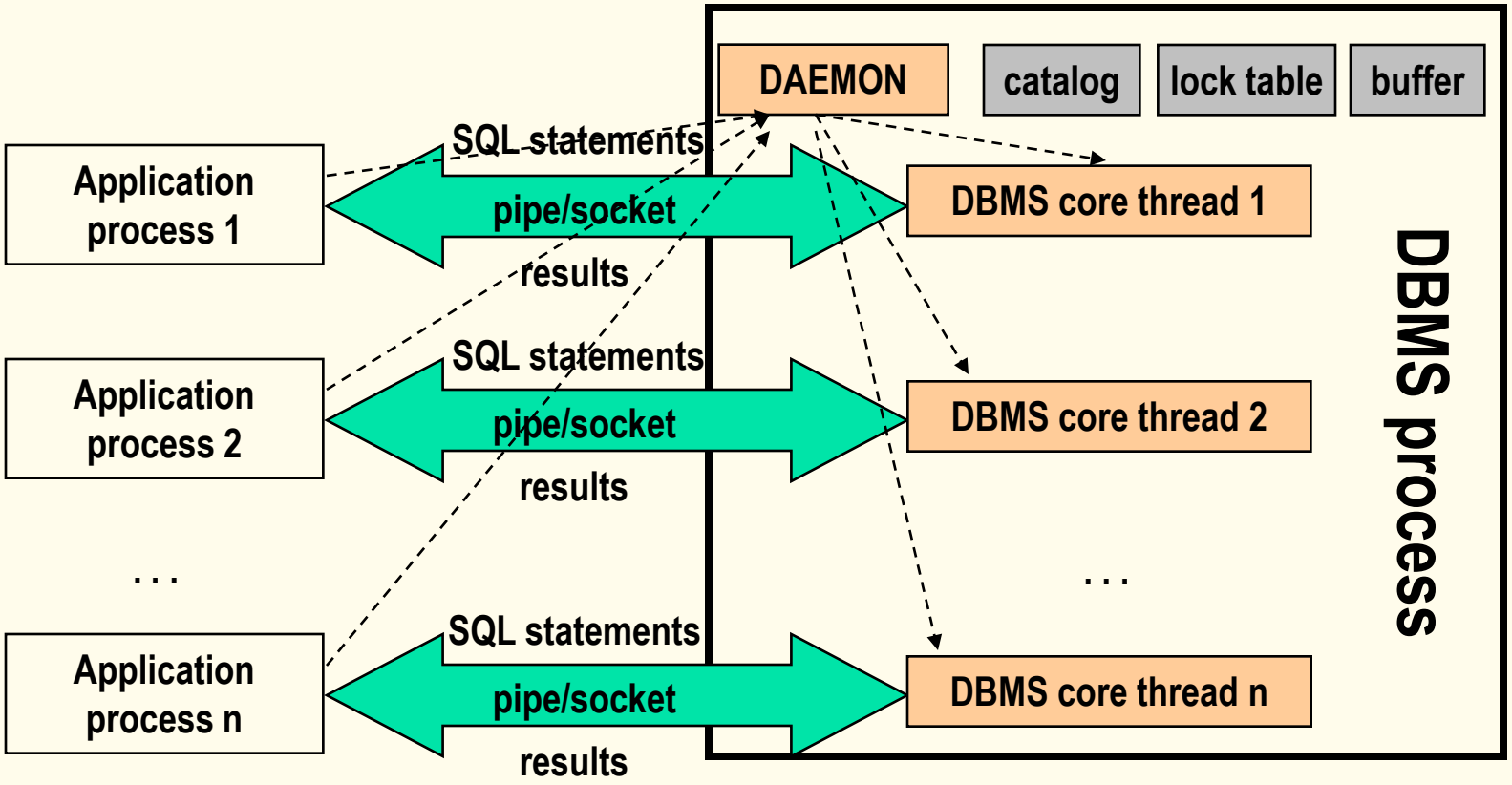
Multi processes structure

- One application process corresponding to one DBMS core process



Multi threads structure

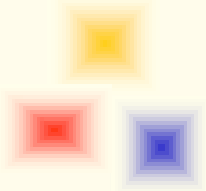
- Only one DBMS process, every application process corresponding to a DBMS core thread.



4.3 Database Access Management

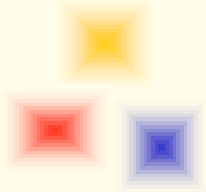
The access to database is transferred to the operations on files (of OS) eventually. The file structure and access route offered on it will affect the speed of data access directly. It is impossible that one kind of file structure will be effective for all kinds of data access.

- 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



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)

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

4.4 Query Optimization

"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.

4.4.1 Summary of Query Optimization

- **Algebra Optimization**
- **Operation Optimization**

Example

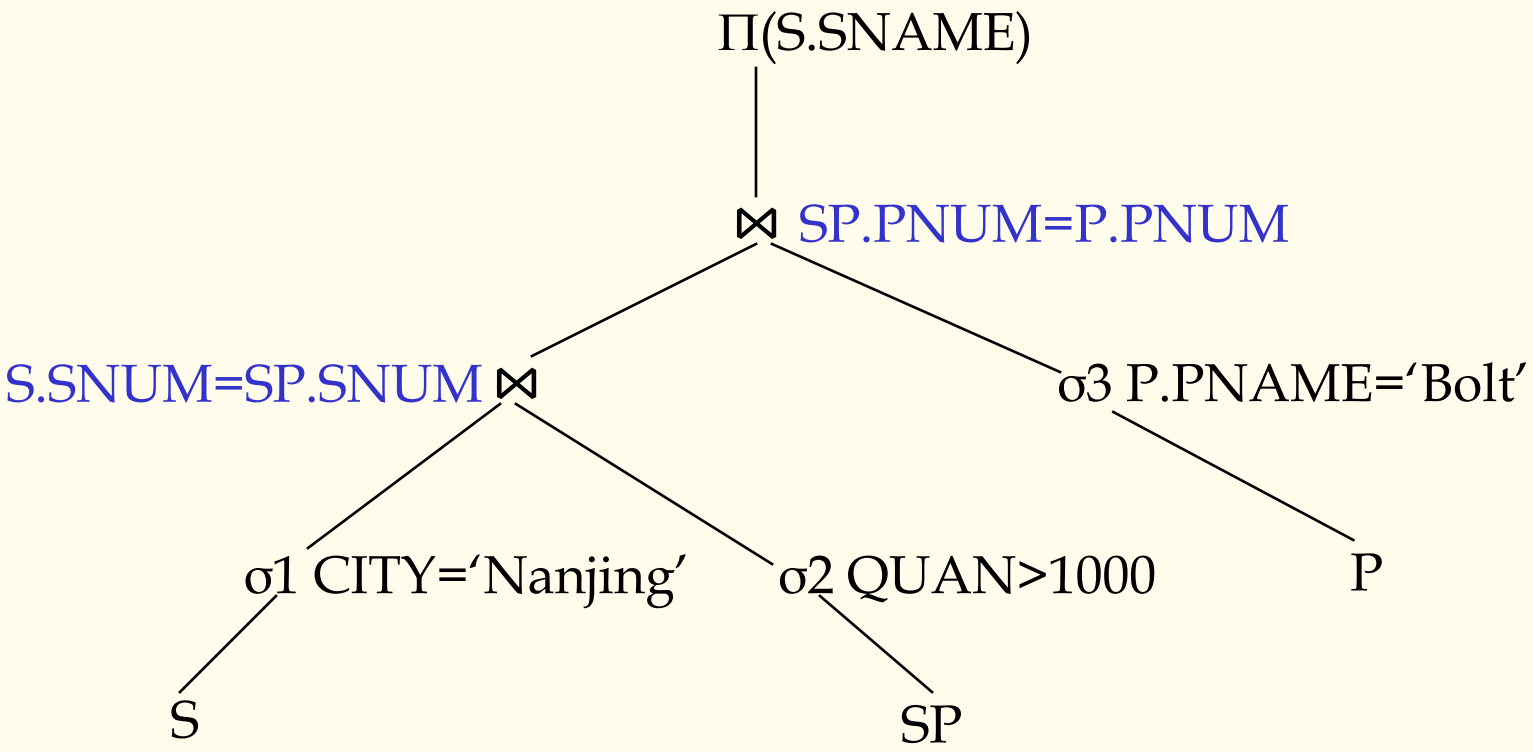
S(SNUM, SNAME, CITY)

SP(SNUM, PNUM, QUAN)

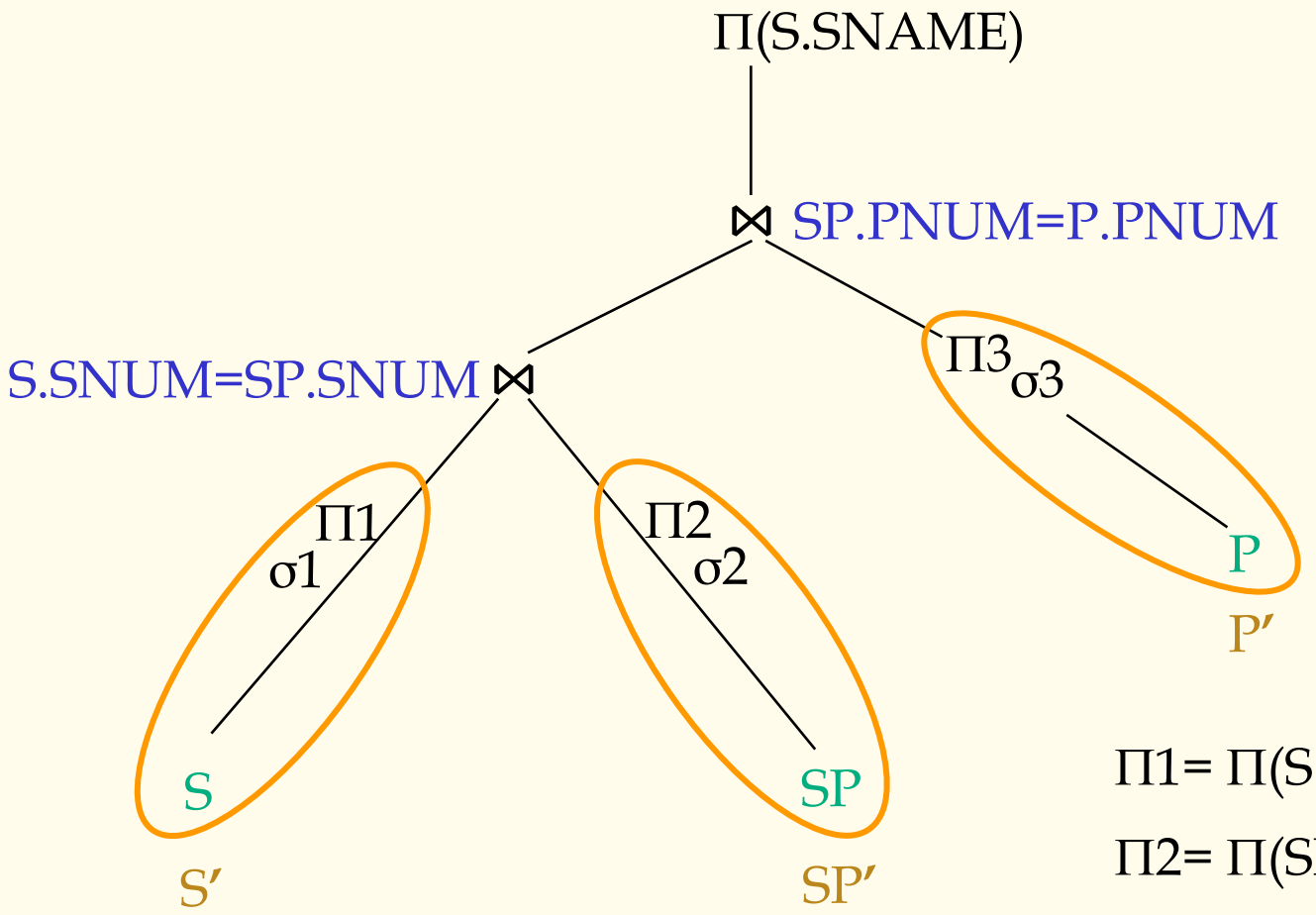
P(PNUM, PNAME, WEIGHT, SIZE)

```
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


Π(S.SNAME)
⋈ SP.PNUM=P.PNUM
S.SNUM=SP.SNUM ⋈
σ3 P.PNAME='Bolt'
σ1 CITY='Nanjing'
σ2 QUAN>1000
P
S
SP

After equivalent transform (Algebra optimization) :

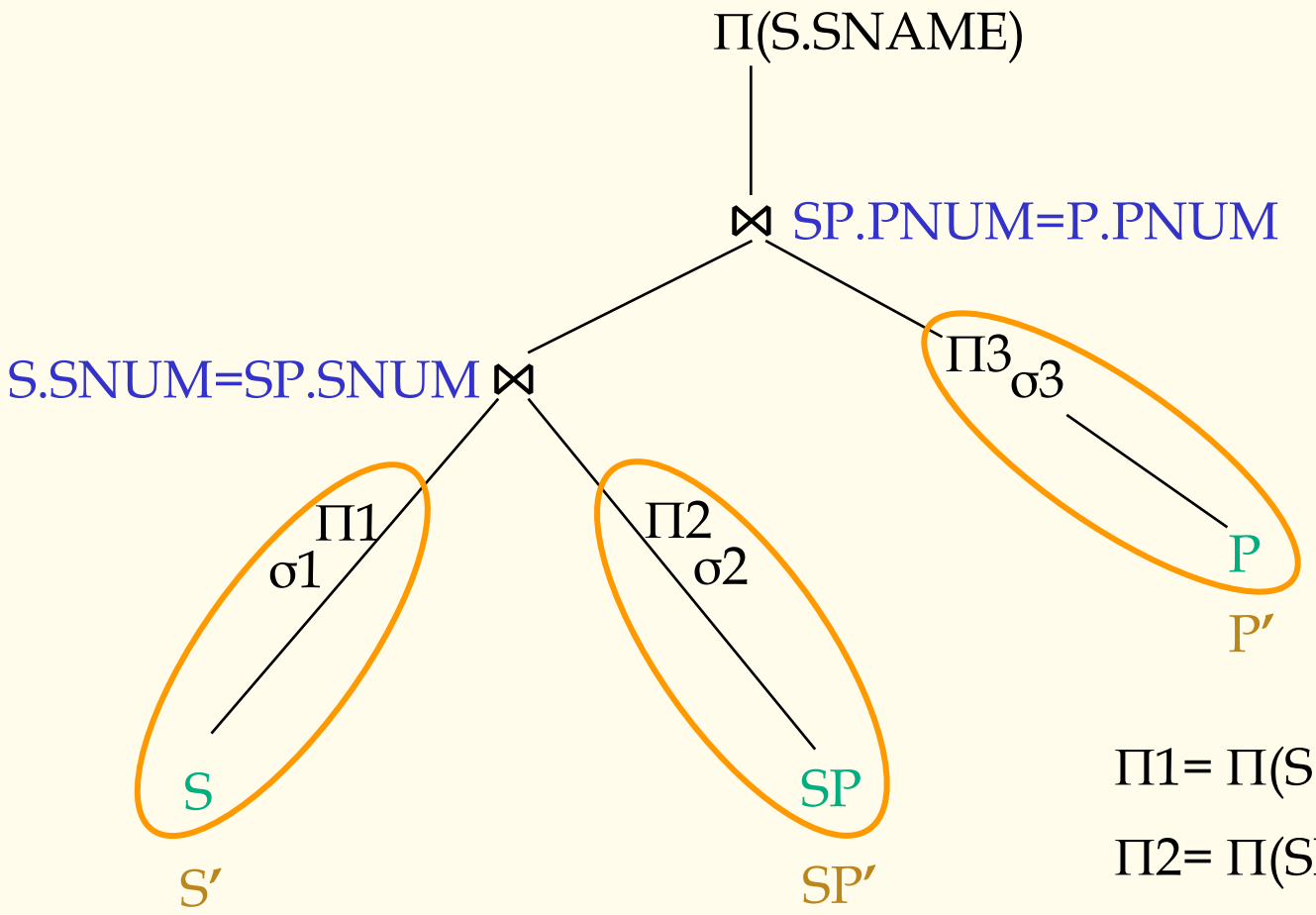


Π1= Π(S.SNUM,S.SNAME)

Π2= Π(SP.SNUM,SP.PNUE)

Π3= Π(P.PNUM)

After equivalent transform (Algebra optimization) :

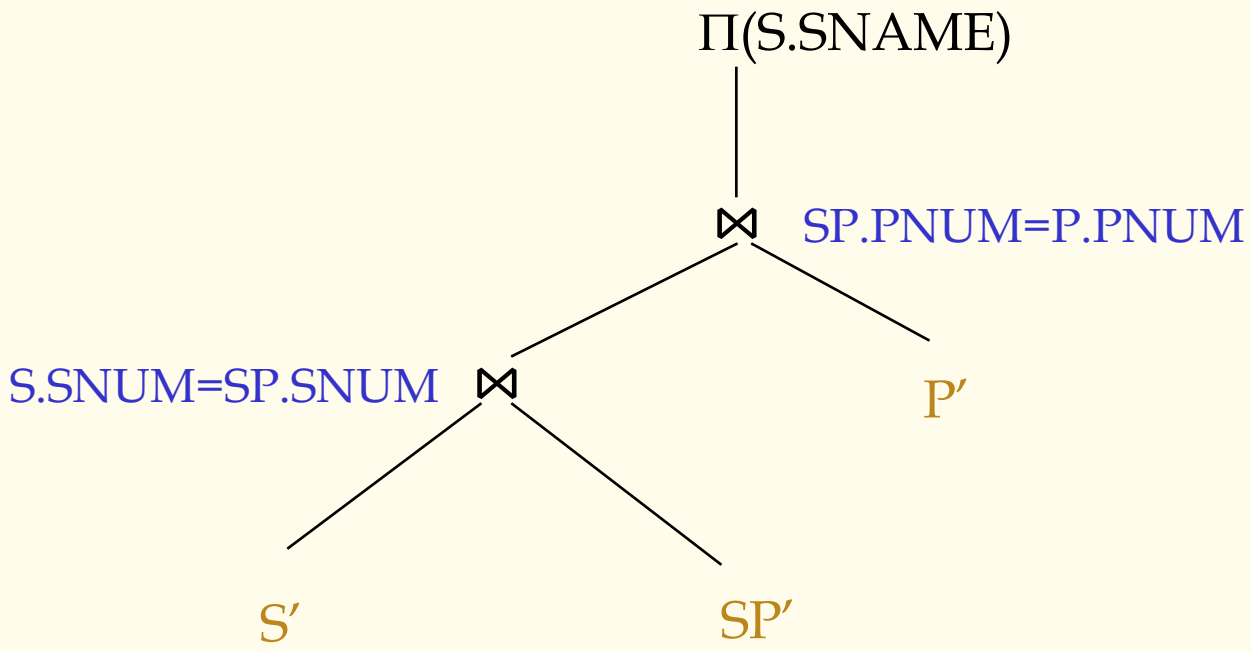


Π1= Π(S.SNUM,S.SNAME)

Π2= Π(SP.SNUM,SP.PNUE)

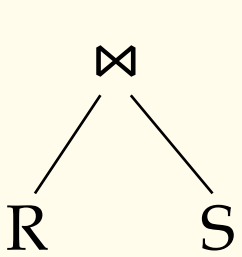
Π3= Π(P.PNUM)

The result of equivalent transform



The operation optimization of the tree :

- Decide the order of two joins
- For every join operation, there are many computing method:



Nest Loop, R as O, S as I
Nest Loop, S as O, R as I
Nest Loop, use possible index on R or S
Merge Scan
Hash Join
⋮

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

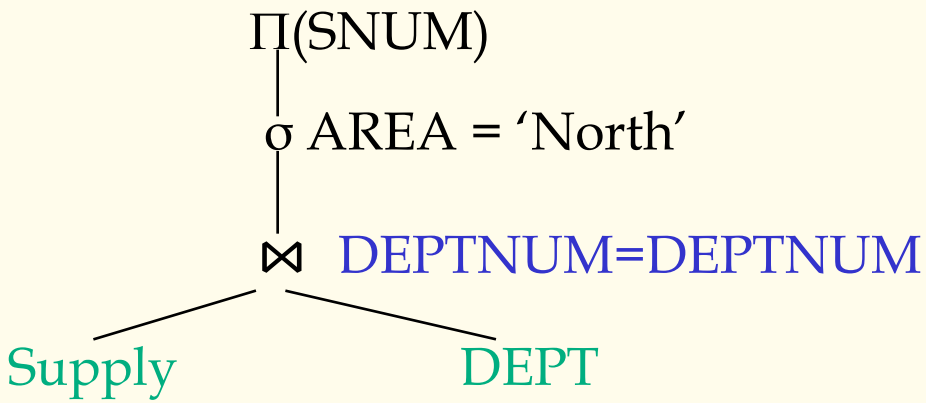
4.4.2 The Equivalent Transform of a Query

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

(1) Query tree

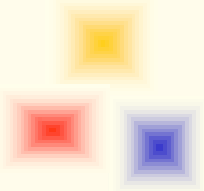
For example: $\Pi_{SNUM}\sigma_{AREA='NORTH'}(SUPPLY \bowtie_{DEPTNUM} DEPT)$



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

(2) The equivalent transform rules of relational algebra

1) Exchange rule of $\bowtie/\times$: $E1 \times E2 \equiv E2 \times E1$
2) Combination rule of $\bowtie/\times$: $E1\times(E2\times E3)\equiv(E1\times E2)\times E3$
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: $\sigma_{F1}(\sigma_{F2}(E))\equiv\sigma_{F1\wedge F2}(E)$
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 $\sigma_F(E1 \times E2) \equiv \sigma_{F2}(\sigma_{F1}(E1) \times E2)$

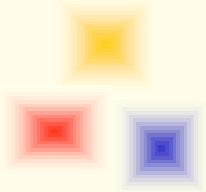
7) $\sigma_F(E1 \cup E2) \equiv \sigma_F(E1) \cup \sigma_F(E2)$

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) $\Pi_{A1 \ldots An}(E1 \cup E2) \equiv \Pi_{A1 \ldots An}(E1) \cup \Pi_{A1 \ldots An}(E2)$



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 $\sigma_F(E1 \times E2) \equiv \sigma_{F2}(\sigma_{F1}(E1) \times E2)$

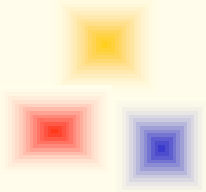
7) $\sigma_F(E1 \cup E2) \equiv \sigma_F(E1) \cup \sigma_F(E2)$

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) $\Pi_{A1\ldots An}(E1 \cup E2) \equiv \Pi_{A1\ldots An}(E1) \cup \Pi_{A1\ldots An}(E2)$

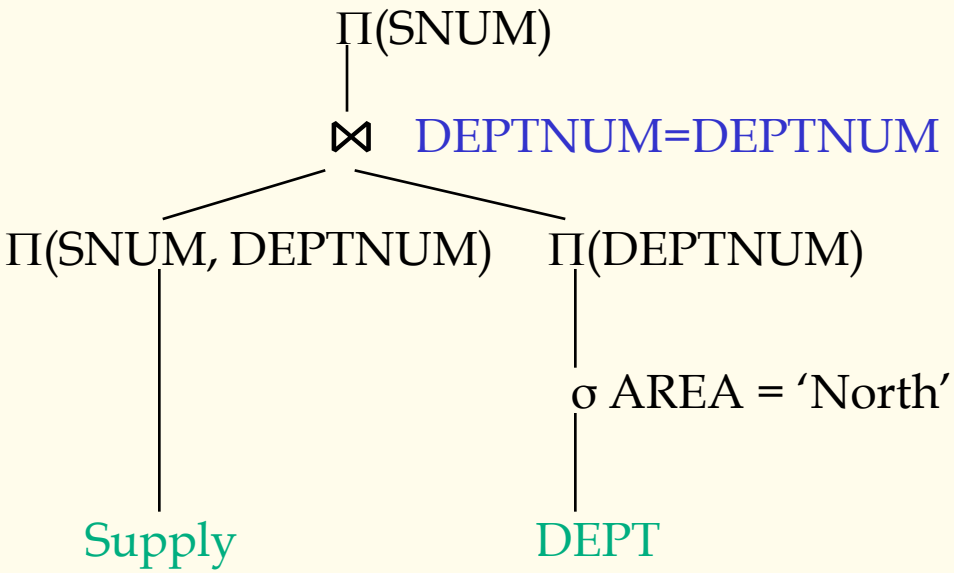


```
SELECT  S.sname
FROM    Sailors S
WHERE   EXISTS  (SELECT  *
                 FROM  Reserves R
                 WHERE  R.bid=103 AND S.sid=R.sid)
```

(3) Basic principles

The target of algebra optimization is to make the scale of the operands which involved in binary operations be as small as possible :

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



4.4.3 The Operation Optimization

How to find a "good" access strategy to compute the query improved by algebra optimization is introduced in this section:

- Optimization of select operation
- Optimization of project operation
- Optimization of set operation
- Optimization of join operation
- Optimization of combined operations

Optimization of join operation

- 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>=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:

  $$b_R + \lceil b_R/(n_B-1) \rceil \times b_S$$

Optimization of join operation

- 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>=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:

  $$b_R + \lceil b_R/(n_B-1) \rceil \times b_S$$

Optimization of join operation

- 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 ⋈ S can be computed based on the hash file.

Optimization of join operation

- 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 ⋈ S can be computed based on the hash file.

4.5 Recovery

4.5.1 Introduction

The main roles of recovery mechanism in DBMS are:

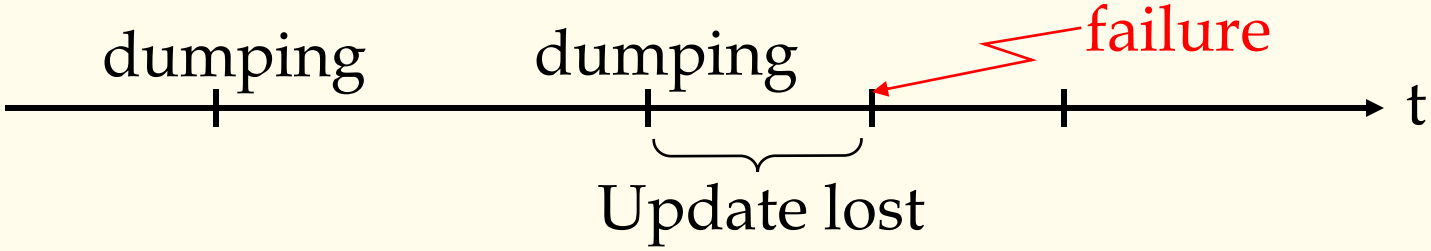
(1) Reducing the likelihood of failures (prevention)

(2) Recover from failures (solving)

Restore DB to a consistent state after some failures.

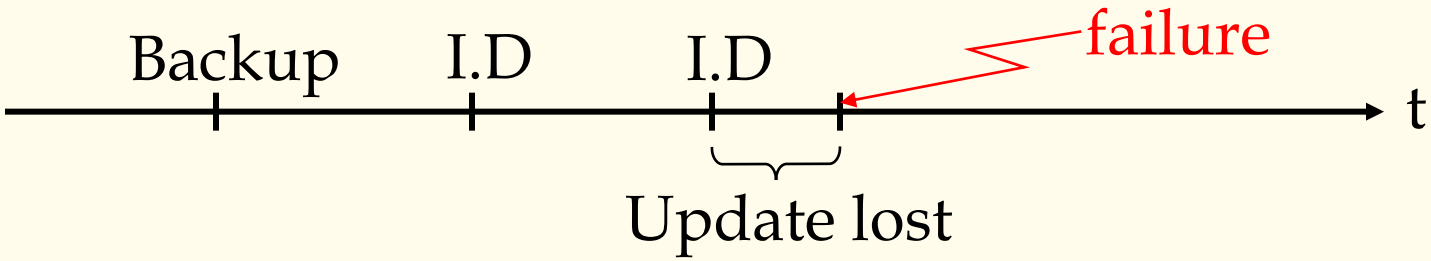
- Redundancy is necessary.
- Should inspect all possible failures.
- General method:

1) Periodical dumping



- Variation : Backup + Incremental dumping

  I.D --- updated parts of DB



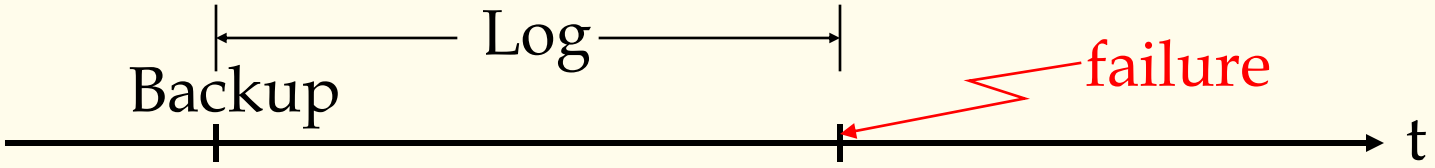
This method is easy to be implemented and the overhead is low, but the update maybe lost after failure occurring. So it is often used in file system or small DBMS.

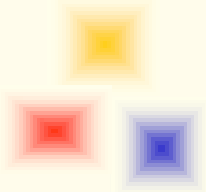
2) Backup + Log

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) } Recorded into Log

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

4.5.2 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 */

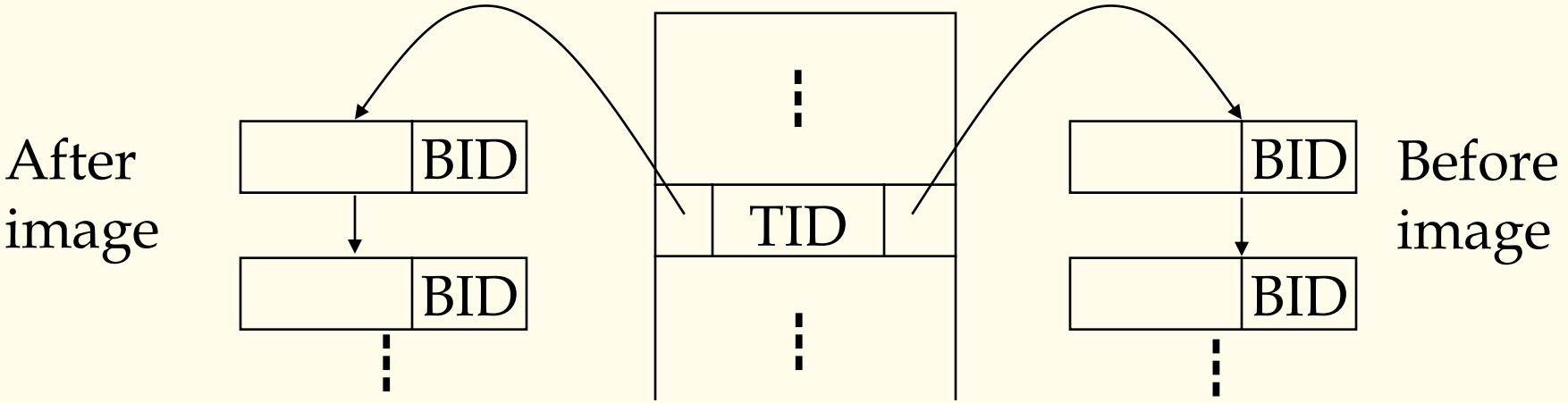
Rollback --- abnormal termination. (Nothing)

Commit --- normal termination. (All)

4.5.3 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 :



4.5.4 Commit Rule and Log Ahead Rule

1) Commit Rule

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

2) Log Ahead Rule

If A.I is written to DB before commit then B.I must first written to log.

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)

4.5.4 Commit Rule and Log Ahead Rule

1) Commit Rule

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

2) Log Ahead Rule

If A.I is written to DB before commit then B.I must first written to log.

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

⋮

commit: 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:

Commit list	Active list	
	✓	Undo, delete TID from active list
✓	✓	delete TID from active list
✓		nothing to do

b) A.I→DB after commit

TID →active list

A.I →Log (Commit Rule)

⋮

commit { TID →commit list

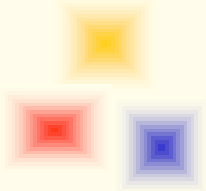
A.I →DB

delete TID from active list

The recovery after failure in this situation

Check two lists for every TID while restarting after failure:

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



c) A.I→DB concurrently with commit

TID →active list

A.I, B.I →Log (Two Rules)

A.I →DB (partially done)

⋮

TID →commit list

commit: A.I →DB (completed)

commit: delete TID from active list

c) A.I→DB concurrently with commit

TID →active list

A.I, B.I →Log (Two Rules)

A.I →DB (partially done)

⋮

TID →commit list

commit: A.I →DB (completed)

commit: delete TID from active list

The recovery after failure in this situation

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

Conclusion :

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

?

4.6 Concurrency Control

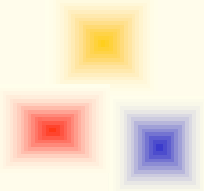
4.6.1 Introduction

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

- Why concurrency?

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

- 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.

4.6.2 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 → 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 $T_A \rightarrow T_B \rightarrow T_C$.

4.6.3 Locking Protocol

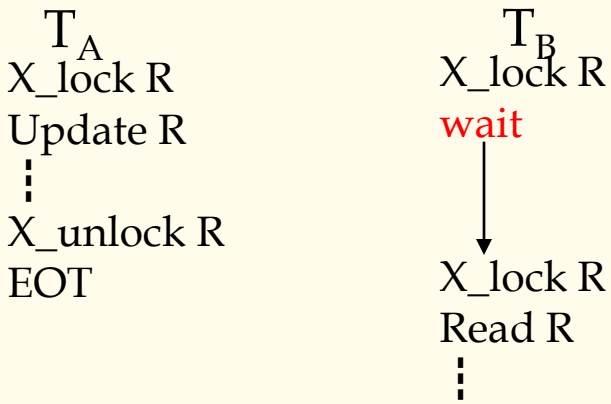
Locking method is the most basic concurrency control method. There maybe many kinds of locking protocols.

(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

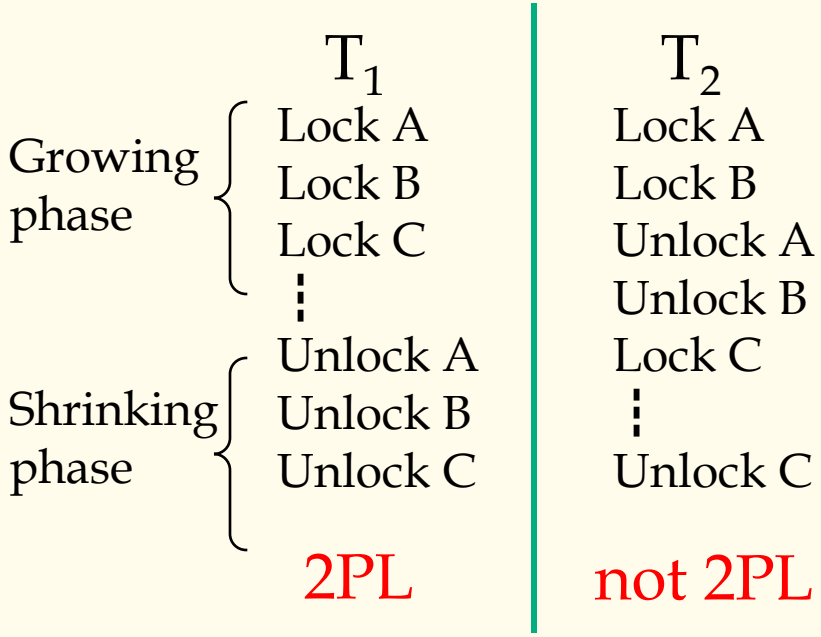
	NL	X
NL	Y	Y
X	Y	N



*Two Phase Locking

- *Definiton1:* 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.

(2) (S,X) locks

S lock --- if read access is intended.

X lock --- if update access is intended.

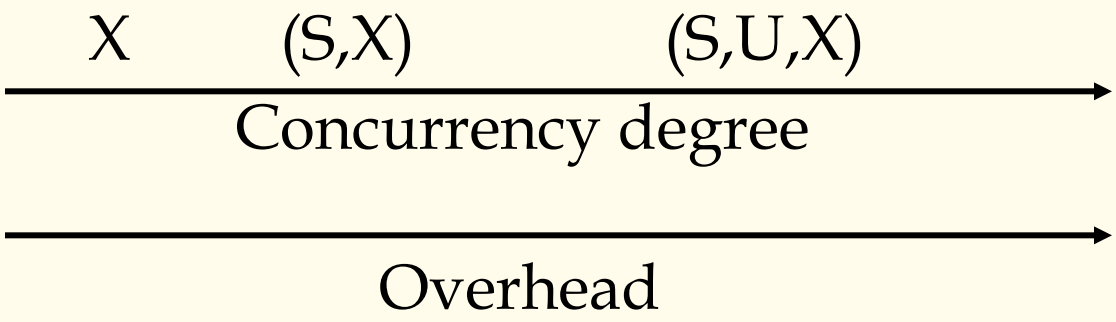
	NL	S	X
NL	Y	Y	Y
S	Y	Y	N
X	Y	N	N

(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.

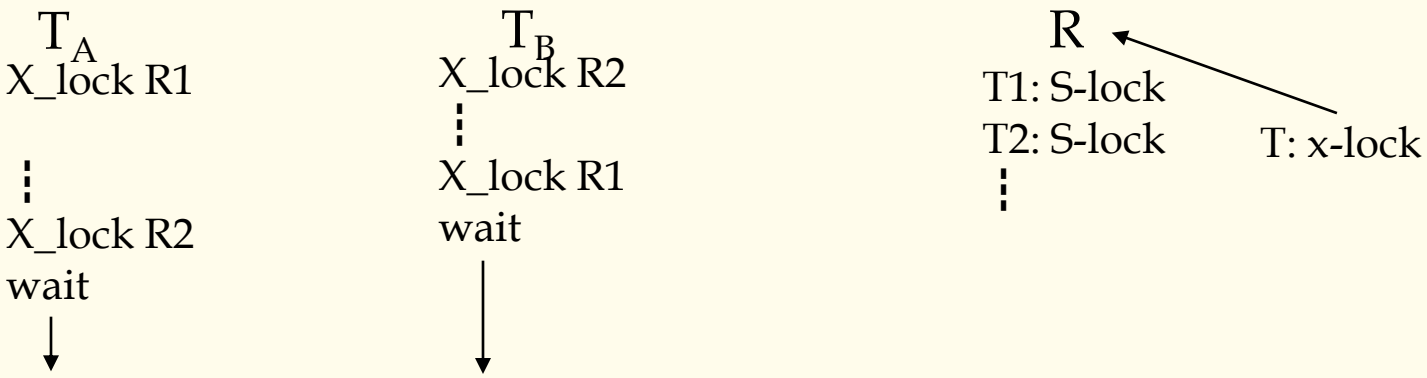
	NL	S	U	X
NL	Y	Y	Y	Y
S	Y	Y	Y	N
U	Y	Y	N	N
X	Y	N	N	N



4.6.4 Deadlock & Live Lock

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.



- 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) Deadlock Detection

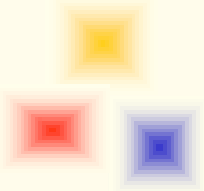
1) Timeout: If a transaction waits for some specified time then deadlock is assumed and the transaction should be aborted.
2) Detect deadlock by wait-for graph G=<V,E>

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

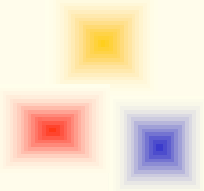


- What to do when detected?

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)

(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



- 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)

(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 → younger or younger → older. It is impossible to occur wait in cycle, so the dead lock is avoided.