# Basics

* Hard disk slow (compared to stuff you do in memory)
* Reading many pages sequentially good (better than random IO)
* Must read into memory any page to do anything

# Hard Disk Types

Magnetic, cheap

SSD – Solid State, expensive

ioping:

us 🡪 microseconds 1/1 000 000

ms 🡪 milliseconds 1/ 1 000

# Magnetic disk access:

Reading a portion of the disk:

Seek time (move the disk head)

+ Rotational latency (more or less fixed)

+ Data transfer time (getting faster with denser disks)

## Terms

## Sector

(hardware minimal unit of disk)

## < Page

(logical combination of sectors, single unit of data we can read) 🡪 Block

## < Track

(a circle on the platter, divided into sectors )

### Surface

A disk track is on a surface of the disk (can be on both sides)

## < Cylinder

(tracks stacking together, can be read from both sides)

A page: a single unit o data I can read.

E.g.:

12 TB

RPM: 7200

Read and write speed: 243 MB/s, 236 MB/s

512 bytes/ sector

8 platters

Seek Time:

Rotational latency: 60 000 / (2\*7200) 6.46ms

Transfer time: (number of consecutive pages) \* transfer time (0.03 ms/page)

Random IO:

Cost to read 100 random pages: 100 \* (6.46 + 4.17 + 0.03) = 1000ms

Sequential IO:

Cost to read 100 sequential pages: 6.46 + 4.17 + 100 \* (0.03) = 13ms

What is the cost of answering this query?

SELECT COUNT(\*) FROM playedonradio where station = ‘mai’;

* Cost: number of pages I have to read
* Sequential scan: read every page the ‘playedonradio’ is stored in and check for station and increment the counter
* Create an index on playedonradio(station)

Each index entry: key value (station) and pointer to a tuple

* Index only scan for

Select COUNT(\*) from playedonradio where station = ‘mai’

* Read all pages for the index until the end of the station ‘mai’

SELECT COUNT(distinct songid) from playedonradio where station = ‘mai’

* Read all index pages to find all tuples for station = ‘mai’

+ read all found tuples to find songids

Disks are much slower than memory

Cost of a query: # of disk pages read (and written)

# RAID

(Redundant Arrays of Inexpensive Disks)

1. Speed of read/write operations
2. Fault tolerance

## RAID – 0

(4 disks, distribute the data to multiple disks)

D1 p1 p5 p9 p13

D2 p2 p6 p10 p14

D3 p3 p7 p11 p15

D4 p4 p8 p12 p16

Read/Writes faster

No redundancy

## RAID – 1

(4 disks, distribute the data to multiple disks)

D1 p1 p3 p5 p7

D2 p2 p4 p6 p8

D3 p1 p3 p5 p7

D4 p2 p4 p6 p8

Reads are fast (though not as fast as RAID-0), writes are slower than RAID-0.

If any disk goes down, no data loss and can continue to operate without a slowdown

## RAID – 4

(4 disks, distribute the data to multiple disks)

D1 p1 p5 p9 p13

D2 p2 p6 p10 p14

D3 p3 p7 p11 p15

D4 p4 p8 p12 p16

D5 p1-4 p5-8 p9-12 p13-16

D5 is called the parity disk, which stores the (XOR) of all other 4 disks

On updates, the parity must be updated simultaneously

Reads are much faster, writes are slower (every write to disks 1-4 requires a change to parity disk), parity disk is a bottleneck.

RAID-4 does not loose any data for 1 disk failure

It is possible to construct the lost disk from the others, but this operation is slow(the RAID is degraded!)

## RAID – 5

Stripe the parity disk

D1 p1 p6 p11 p16 p17-20

D2 p2 p7 p12 p13-16 p17

D3 p3 p8 p9-12 p13 p18

D4 p4 p5-8 p9 p14 p19

D5 p1-4 p5 p10 p15 p20

Reads are still fast, writes are faster than RAID-4, and still can recover from 1 failure

# B-Trees

An index page stores: some header info, key values (indexed attribute/ attributes) and a pointer to a disk page

For example: a disk page 8KB

Suppose you index an integer (4B)

And a disk page address is 12B

Each entry is 16B

The capacity of a single page is: 8\*1024/16 (disregarding the header)

🡪order of the B-Tree (n value)

## Given a B-Tree of order n:

1. Each node of a B-tree is a disk page
2. Each node can store at most n key values (n+1 pointers) and at least n/2 values  
   (each node is at least half full, with the exception of root)
3. A leaf node will store key values and pointers to tuple in the relation

A leaf node can address at most n tuples and has n+1 pointers (1 pointer to the next leaf node in the B-tree)

1. Internal nodes point to index nodes at the level below. An internal node can have at most n key values and n+1 pointers
2. Root can have anywhere between 2 to n+1 pointers and (1 to n key values)

## Estimate the size of a B-tree:

A B-tree on R(A,B) where each node can store at most 200 **entries** (key+pointer)

(min 100, except for root)

What is the size of the B-tree for a relation with 10 million tuples

Leaf level: 10000000/200 = 50,000 nodes at the leaf level

Next internal level: 50,000/200 = 250 nodes

Next internal level: 250/200 = 2 nodes

Next level: 1 node (root)

number of levels is log\_n(number of tuples)

What if every node was half full?

Leaf level: 10000000/100 = 100,000 nodes at the leaf level

Next internal level: 100,000/100 = 1,000 nodes

Next internal level: 1000/100 = 10 nodes

Next level: 1 node (root)

Insertion/Deletion

Insert: search for the value

if there is space, insert

else: create a new node, distribute the values and insert a

pointer to the new recursively at the parent

Secondary storage continued

B-trees

Each value in the intermediate level stores the decision value that distinguished two different leaves

E.g.

Create index “playedonradio-Idx” on playedonradio(station, songid)

This will create an index sorted on station first and then songid

Each node will be (station, songid)

This will be extremely efficient on search of station for its sorted first on station, but not as efficient searching on songid.

Half full:

Leaf: 2 tuples minimum (4 tuples max)

Internal: 2 tuples minimum (5 tuples max)

Index on R(X,Y) can be used for many types of searches (to different levels of effectiveness)

Effectiveness = how much of the leaf level do I need to scan

E.g.

X=1 and Y=5 OR X=1 and Y>=5 and Y<=10

X=1 OR X<=4 and Y=2 (Y condition is mostly irrelevant )

Y=2 (Y condition is irrelevant, scan the whole index)

B-trees (points, any geometric objects)

Balanced tree

PostgreSQL: GIST trees

Unbalanced trees: Quadtrees and k-d trees

PostgreSQL: sp-GIST trees

Inverted index:

W1 -> t1:2,4 t2:4,6

Secondary index vs primary index

——

Select \* from playedonradio where station = ‘mai’;

Secondary indices does not change how the relation is stored on disk, just create look up reference.

Primary indices change how to store the data.

Hashing improves the equality search incredibly.

Secondary storage continued!

-------------------------------

Lecture exercise today

Still no exam results, sorry about that.

create index playedonradio1\_idx on playedonradio(station,songid);

create index playedonradio2\_idx on playedonradio(station);

select songid from playedonradio where station = 'mai' ;

select station from playedonradio where songid = 1234 ;

B-trees are super useful

----------------------------

Index on R(X,Y) can be used for many types of searches (to different

levels of effectiveness)

effectiveness = how much of the leaf level do I need to scan

X=1 and Y=5 or X=1 and Y>=5 and Y<=10

X=1 or X>=1 and X<=4

X>=1 and X<=4 and Y=2 (Y condition is mostly irrelevant)

Y=2 (Y condition is irrelevant, scan the whole index)

But wait, are there other types of indices?

------------------------------------------------

R-tree (points, any geometric object)

balanced tree

Postgresql -> GIST trees

Unbalanaced trees -> Quadtrees and k-d trees

Postgresql -> Sp-GIST trees

create extension pg\_trgm;

create table rs as select \* from rollingstonetop500 ;

alter table rs add review tsvector ;

update rs set review = to\_tsvector(critic) ;

create index review\_idx on rs using gin(review) ;

Inverted index:

W1 -> t1:2,4 t4:4,6

select a.name, r.album

from artists a, rs r

where r.artistid = a.id

and r.review @@ to\_tsquery('(masterpiece | avantgarde) & (comtemporary | modern)');

--------------

Secondary Index vs. Primary Index

-----------------------------------------

select \* from playedonradio where station = 'mai';

Secondary indices do not change how the relation is stored on disk

Just creates look up structures

Primary indices change how to store the data

Hashing is a way to create a primary index

Hashing is useful for equality searches

Hash on R(A) is useful for queries: A=x

Primary index structures are useful for some queries but

need to be maintained to remain effective

# Query Processing

Cost of queries: Number of disk pages read/written

Each relational algebra operator implements the same interface

## Iterator interface:

Initialize: Allocate memory space to the operator, initialize relations to be read

Get\_Next: Do processing needed to produce 1 block of output

Close: Deallocate memory space and release any resources

For each operator: let M is the number of blocks of memory allocated

Relations R 🡪

PAGES(R) is the number of pages for relation R

TUPLES(R) is the number of tuples in the relation R

**Often, TUPLES(R) >> PAGES(R)**

## Single pass operations:

Sequential scan (R)

PAGES(R) and only needs M=1 blocks

Duplicate Removal (R)

Group by (R)

-Depends on whether M-1 blocks is sufficient to hold the

temporary results in memory

R union all S

R union S

R except all S

R except S

R intersect all S

R intersect S

-Depends on whether M-1 blocks is sufficient to hold the

temporary results in memory

See costs in hand written notes

# Index Scan

Relation R

PAGES(R) = 2,000

TUPLES(R) = 20,000

(How many tuples of R per page on average? )

## Index I1 on R(C)

with 3 levels (root, internal, leaf) and 400 nodes at leaf level

( Each leaf node of I1 on average indexes how many tuples? )

TUPLES(R.C=10) = 120

### Q1: SELECT A,B FROM R WHERE C=10;

Cost of query Q1:

Sequential scan: Cost = 2,000 pages

Using index scan with I1 (M=2):

From index = Root + Internal + 3 or 4 leaf nodes = 5 or 6 pages (nodes)

From relation = 120 pages in the worst case

(each tuple is in a different page)

Total = 126 (or 125)

PAGES(R) = 2,000

TUPLES(R) = 20,000

(How many tuples of R per page on average? 10 )

## Index I2 on R(B,C)

with 3 levels (root, internal, leaf)

and 800 nodes at leaf level

(Each leaf node of I2 on average indexes how many tuples? -20000/800= 25)

TUPLES(R.C=10) = 120

### Q1: SELECT A,B FROM R WHERE C=10;

Cost of Q1 using I2:

Index scanning cost + Relation scanning cost

2+ 800 + 120 (worst case) = 922

Sequantial scan = 2000

Using I1 = 126

Using I2 = 922

## Index I3 on R(C,A,B)

with 3 levels (root, internal, leaf)

and 1200 nodes at leaf level

(Each leaf node of I3 on average indexes how many tuples? -20000/1200= 16 tuples)

### Q1: SELECT A,B FROM R WHERE C=10;

120 tuples are stored in how many leaf nodes? 120/16= 8 approx

Cost of Q1 using I3:

Index scanning cost + Relation scanning cost

2 + 8 + 0 (index already stores A and B)

Index types:

1. Selective index for a condition C: only a few tuples match condition C

(very low relation scan cost)

2. Indices that enable index only searches (storing attributes needed

for a query)

### Q2: SELECT A,B FROM R WHERE C=10 AND D > 5;

Index I3 on R(C,A,B)

Scan index I3 using C=10, then read matching tuples to check on D>5

Cost of answering Q2 using I3?

Index scanning cost + Relation scanning cost

2 + 8 （or 9） + 120 (worst case, all tuples for C=10)

Query processing

# Multi-pass operations

## Block Nested Loop Join

M Blocks of memory

Assume M = 2:

1 Block for R and 1 block for S:

For each Page in R, we need to read all the pages in S to complete one cycle of the join operation

Block Nested Loop Join M > 2:

M-1 block for R and 1 block for S

## External Sort

M<Pages(R)

2 phases 🡪 assume M is fixed throughout

### Phase 1:

Read🡪Sort🡪Write temporary

Number of sorted chunk

### Phase 2:

Read the smallest part of each sorted group into memory

Find the smallest tuple and output

## Benefits of sorting

Distinct: All duplicates are adjacent to each other

Group By: Similar to distinct

# Hashing

Apply to each bucket separately

Remove duplicates

Group by

Union/ Difference/ Intersection

# Query Optimization

## Query optimization procedure🡪

1. Parse query and construct a relational algebra equivalent

2. Sanity check: logic and syntax check

(select \* from r where a = 1 and a = 2;) [return 0 on impossible queries]

Enumerate many options for:

3. Construct a query tree from your query

4. Find an implementation for each operator

5. Find its expected cost (before running the query)

Choose the cheapest one!

- Finding equivalent query trees

## Relational Algebra equivalences

R join S = S join R

R join (S join T) = (R join S) join T

### Pushing selections down a join:

select\_{C} (R join S) = select\_{C} (R) join select\_{C} (S)

If condition C only applies to R, then (\*\*\*):

select\_{C} (R join S) = select\_{C} (R) join S

Advantage: joins may be cheaper because we are joining fewer tuples

that span fewer pages.

### Pushing projections down:

project\_ {A,B } (R join S) = project\_{A,B} (project\_{A,B,C1} R) join (project\_{A,B,C2} S)

where C1 and C2 are attributes from R and S respectively that are

needed for the join.

Advantage: the size of tuples is reduced so more can be stored in the

same amount of memory, and reduce the cost of joins (and other

operations).

## Finding expected size of output of different operations

(Before I run the query)

Statistics to keep for each table:

Number of tuples in the table (TUPLES(R))

For each attribute A in the table,

**VALUES(R.A): number of distinct values stored for R.A (DISTINCT(R.A))**

**MINVAL(R.A) MAXVAL(R.A) 🡪current min and max value stored for R.A**

(Assumption is each value for R.A is equally like and is distributed

uniformly between min/max values)

### Equality query:

This assumes each class spreads evenly

### Range query:

### NOT query:

### AND query:

### OR query:

### Join query:

select \* from r,s where r.a = s.b

select \* from r,s where r.a = c1 -> 1/values(R.A)

select \* from r,s where s.b = c2 -> 1/values(S.B)

Tuples = TUPLES(R) \* TUPLES(S) \* Sel(R.A=S.B)

## Example

Table: Students(RIN, Class, Major, Age)

Tuples(Students) = 8,000

Values(RIN) = 8,000

Values(Class) = 4

Values(Major) = 40

Values(Age) = 10 Minval(Age) = 15, Maxval(Age) = 32

Table: Transcript(rin, crn, grade)

Tuples(Transcript) = 200,000

Values(RIN) = 7,695

Values(CRN) = 300

### Q6: SELECT \* FROM Students S, Transcript T WHERE S.rin = T.rin;

Sel(S.rin = T.rin) = 1/{8000,7695}

Tuples = 8000 \* 200,000 \* 1/8000 = 200,000

### Q1: select \* from students where class = 'senior';

Selectivity(C) = the percentage of tuples that are expected to pass

condition C

Tuples(Q) = Tuples(R)\* Selectivity(C)

Selectivity(class = 'senior') = 1/4

Tuples(Q1) = 8,000 \* 1/4 = 2,000

### Q2: select \* from students where major = 'Math' ;

Selectivity (major = 'Math') = 1/40

Tuples(Q2) = 8,000 \* 1/40 = 200 students

### Q2': select \* from students where major = 'CSCI' ;

Same expected values, but may be way off!

### Q0: select \* from students where rin = 660012345

Selectivity =1/8000

Tuples = 8000 \* 1/8000 = 1

### Q3: select \* from students where age >= 18 and age < 21;

Selectivity = 3/(32-15) = 3/17

Tuples = 8,000 \* 3/17

### Q3': select \* from students where age >= 29 and age < 40;

(Must estimate in range)

Selectivity = (32-29)/(32-15) = 3/17

### Q4: select \* from students where major = 'CSCI' and class = 'senior';

sel = 1/40 \* 1/4

tuples = 8000/160

### Q5: select \* from students where age = 18 and class = 'senior';

## -Enumerate all possible implementations

---> Find all possible two-way joins (if joining R,S,T,W)

R,S

R,T

S,T

R,W

S,W

T,W

Find cheapest ways to implement the join + keep a number

of interesting query plans with the two relations that may provide

cheaper plans later

-> Take each 2-way join and add a third relation to join

(R,S) + T

(R,T) + W

...

Prune options that are too expensive.

Continue adding joins until the whole query is implemented, and choose

the cheapest.

-------------------------------------------

PAGES(R) = 100, TUPLES(R) = 20,000

PAGES(S) = 500, TUPLES(S) = 200,000

Suppose each attribute is 4 bytes long.

VALUES(R.A)= 1,000

VALUES(R.E)= 2,000

VALUES(S.F)= 1,800

VALUES(R.B)= 400

VALUES(R.C)= 5,000 between 0 (minval) - 10,000 (maxval)

SELECT

R.D

FROM

R

WHERE

R.A = 5

AND R.B in ('a','b')

AND R.C <= 100 AND R.C > 500 ;

Sel( R.A = 5) = 1/1000

(a or b logic)

Sel (R.B in ('a','b')) = 1- (1-1/400)\*(1-1/400)

Sel( R.C <= 100 AND R.C > 500) = 400/10000

Sel (Query) = 1/1000 \* (1- (1-1/400)\*(1-1/400)) \* (400/10000)

SELECT

R.D, S.G, count(\*)

FROM

R,S

WHERE

R.E = S.F

AND R.A = 5

GROUP BY

R.D, S.G ;

Sel(R.E=S.F) = 1/max{2000,1800} = 1/2000

Sel(R.A) = 1/1000

Sel = 1/1000 \* 1/2000

Tuples = 20,000 \* 200,000 \* 1/1000 \* 1/2000

-------

select

avg\_width

, histogram\_bounds

from

pg\_stats

where

tablename = 'songs'

and attname = 'decade';

# Concurrency

Serial execution of transactions are accepted as correct.

## Goal of concurrency

Goal of a concurrency is to ensure that even through operations are executed concurrently, the end result o the transaction is equal to a serial execution

Abstract transaction T[i] as sequence of

ri(x), wi(i), zi(7) wi(y)

## schedules

a sechedule is what took places in the db, as a combination of read/write operations of different transactions.

Schedule S1: r1(x), r2(X), w2(x), r2(Y),w2(y),W1(x),r1(y),w1(Y)

Schedule S1: r1(x), w1(X), r1(x), w1(Y),r2(y),W2(x),r2(y),w2(Y)

Schedule S1: r2(x), w2(X), r2(x), w2(Y),r1(y),W1(x),r1(y),w1(Y)

Two schedules are going to produce the same result in the DB if they read the same values and write values in the same order.

Example

S1: r1(x), r2(y), w2(y), w1(x),r2(x),w2(x)

S2: r2(x), w2(y), r1(y), w1(x),r2(x),w2(x)

These two are the same because all the values are read are guaranteed to be the same, and written in the same order

S3: r1(x), w1(x), w2(x)

S4: r1(x), w2(x), w1(x)

The order of write is swapped, so S3 and S4 may not produce the same result

To check for this, we will look at only the conflicting operations between two different transactions:

## Conflicts:

Two operations by two different transactions is a conflict if they are on the same item and one of the two is a write operation

1. Different transaction (tr1 and tr2)
2. Same item (on X for example)
3. One write

r1(x) r2(x) is not a conflict

w1(x) r2(x) is a conflict

r1(x) w2(x) is a conflict

w1(x) w2(x) is a conflict

if the order is changed for:

r1(x) w2(x)

w1(x) r2(x)

then the value read will be different

If the order is changed for:

w1(x), w2(x)

then the final value written will be different

Two schedules are conflict equivalent (i.e. guaranteed to produce the same result) if all conflict in both occur in the same order

## serializable

A schedule is serializable if it is guaranteed to be equivalent to a serial schedule.

S1: r1(x) w1(x) r2(x) w2(x)

S2: r2(x) w2(x) r1(x) w1(x)

S3: r1(x) r2(x) w2(x) w1(x)

Conflict in S3:

r1(x) w2(x) no, this is swapped in S2

r2(x) w1(x) no, this is swapped in S1

Do these occur in the same order as s1 No

Do these occur in the same order as s2 No

S3 is not serializable

S4: r1(x) r2(x) w1(x) w2(x)

To check whether a schedule is serializable, create a conflict graph:

Each node will be a transaction

Each edge is a conflict: edge from Ti 🡪 Tj exists if there is a conflict of the form ri(x) … wj(x) or wi(x) … rj(x) or wi(x) … wj(x)

S4: r1(x) r2(x) w1(x) w2(x)

r1(x) w2(x)

r2(x) w1(x)

w1(x) w2(x)

T1 🡨🡪 T2

Has a cycle, so not serializable

### Example

r1(x) r3(z) w3(z) r4(k) z2(y) w1(x) w2(y) r3(x) r5(y) r4(z) w3(y) r3(k) w4(k)

w3(z) r4(z)

r2(y) w3(y)

w1(x) r3(x)

w2(y) r3(y)

w2(y) w3(y)

r3(k) w4(k)

(1) 🡪(3) 🡪(4)

^

(2) -|

No cycle, serializable

T1 T2 T3 T4 conflict equivalent

T2 T1 T3 T4 serializable schedules

## Locking based concurrency control

Before any transaction reads or writes an item X< it must get a lock on the item X

🡪If the lock is not available, transaction is suspended and waits until the lock is available

Allow two types of locks:

S 🡪 shared lock (read lock), typically used for reading items

X 🡪 exclusive lock (write lock), typically used for writing item

Existing lock on an item

|  |  |  |  |
| --- | --- | --- | --- |
|  | No lock | S | X |
| S | Allow | Allow | Reject |
| X | Allow | Reject | Reject |

Two Phase Locking protocol (2PL):

In 2PL, a transaction may not get a new lock after it releases a lock

Two phases:

🡪 growing phase: transaction may get new locks

🡪 Shrinking phase: transaction may only release locks (cannot get new locks)

Announcements:

---------------------------------------------

1. Please fill your course evaluations

2. FINAL EXAM on: Thursday, 12/16 6:30pm-9:30pm

Report any remaining conflicts by the end day today

3. Lecture 25 exercise will be either today or thursday depending on

how far we get today.

# Concurrency

Locking based mechanisms: deadlocks are possible!

Resolving deadlocks:

-> Avoid deadlocks

-> Detect deadlocks and resolve them

## Two phase locking (2PL):

Lock with S lock for reading and X lock for writing

and no new locks can be received after releasing one lock.

## Strict two phase locking (S2PL):

Lock with S lock for reading and X lock for writing

and hold all locks until commit, and then release all locks.

2PL and S2PL both guarantee serializability.

## Set transaction isolation levels:

SET ISOLATION LEVEL READ COMMITTED;

SET ISOLATION LEVEL REPEATABLE READ;

-- Two phase locking guarantees this, but

-- can still suffer from phantom updates for tuple based locks.

SET ISOLATION LEVEL SERIALIZABLE ;

What is being locked?

Table

Data Page

Tuple

Suppose we are locking tuples.

T1: select count(\*) from R WHERE A = 5;

lock every tuple with a shared lock

T2: insert into R(A) values(5,..);

Since T1 locks all existing tuples, T2 is going to lead to an

incorrect count and cna make the schedule non-serializable despite

using 2PL. --> phantom update problem

## Multi-level locks:

### IX -> intention to write some objects at levels below

### IS -> intention to read some objects at levels below

### S -> will read all nodes below this one

### X -> will write all nodes below this one

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Req\Exist | IS | IX | S | X |
| IS | True | True | True | False |
| IX | True | True | False | False |
| S | True | False | True | False |
| X | False | False | False | False |

## Durability

(Assume concurrency problem is resolved!)

All write operations are correctly executed:

- If a transaction aborts/rollbacks, then we delete all values

written/changed by them

- If a transaction commits, then all its changes are recorded

correctly.

Even in the presence of power or software failures.

LOG:

LSN Operation PrevLSN

101 T1 update Px A B -

102 T2 update Py C D -

103 T1 commit 101

104 T3 update Pz 10 20 -

105 T3 update Pw E F 104

106 T2 update Px B G 102

107 T2 abort 106

Log records:

Ti update Pj before\_value after\_value

Ti commit

Ti abort

For each data page in memory:

Px LSN of first log entry that changed it that is not yet written to

disk

For each data page on disk:

Px LSN of last log entry that changed that page

-------------------

What do we want?

- If a transaction aborts/rollbacks, then we want all log entries for

values written/changed by them recorded

- If a transaction commits, then all its changes are recorded (and

cannot be lost)

When do I write log back to disk?

### -> Write ahead logging (WAL): The log is always ahead of the data pages.

If you are going to write a data page to disk, first flush all of the

log currently in memory to disk, then the data page can be written to

disk.

When do I write data pages back to disk?

When do I write data for committing transactions?

## Implementing a commit:

### FORCE:

All data pages by a xact must be written to disk before it is

allowed to commit.

Ti wants to commit:

1. Flush the log to disk (WAL)

2. Write all pages modified by Ti to disk

3. Write a commit record to the log, and flush the log

4. Allow Ti to commit

### NO FORCE:

Ti wants to commit:

1. Write a commit record to the log, and flush the log

2. Allow Ti to commit

Force simplifies recovery, but no force allows higher efficiency

by using data pages in memory for multiple transactions.

## STEAL OR NO STEAL

Do we allow data pages written to disk before a transaction commits?

### NO STEAL:

Every transaction have their memory space fixed: all pages that they read in memory stay in memory until they complete!

No changes by an uncommitted transaction are written to disk, so no

undo is needed for recovery

### STEAL:

If a transaction wants to write a page back to disk before it commits, flush the log first.

UNDO may be needed for recovery.

## Verify Force and Steal:

NO FORCE:

Committed but not updated the data

STEAL:

Uncommitted but updated the data

FORCE/NO STEAL:

Not able to verify

PageLSN: the LSN of the last log entry that modified this page at the time it was written to disk

# ARIES Recovery

**assuming there is no FORCE ad there is STEAL**

## Step 1:

Analysis: go through the log from the beginning to find what happened in the database

DPT: (Dirty Page Table):

PageID and LSN of the first entry that modified this page that may not have been written to disk

TT (Transaction Table):

Tid LSN

Tid of all transactions that are active at the time of the crash and LSN of the last action by transaction

## Step 2:

Redo all update by committed transactions from the beginning of the log till the end

(in forward order) – these are all transactions that are not complete in TT

-Read a page update by a transaction, check if PageLSN > LogLSN, then skip, otherwise READ the action/update

Also redo all UNDO records.

## Step 3:

Abort and rollback all active transactions in TT by undoing their actions

* Undo the largest LSN from TT, and replace it with PrevLSN of the same transaction
* Write UNDO record for each action, read the page into memory, if PageLSN >= LogLSN, then UNDO the action

# Checkpointing

is a way to save the current state,

Write the current TT and DPT to log, and flush log to disk.

Step 1: change analysis to start from the most recent checkpoint,

First, record DPT and TT at the time of check point and continue updating DPT and TT from this point on

Step 2: start the REDO phase from the smallest LSN in DPT

# Database Tuning

Workload: All queries + Update and how frequently they run!

What are some things we can do?

* Tune DB to the system: configure memory use, cache use, CPU/multicore use, etc.
* Died up a table to multiple relations

## Normalize

R(A,B,C,D,E)

A🡪 (A,B,C,D,E)

R1(A,B,C)

R2(A,D,E)

A🡪(A,B,C,D,E)

A column store: (A,B) (A,C) (A,D) (A,E)

## Denormalize

Instead of

R(A,B,C) and S(A,D,E)

Where A🡪BC, AD🡪E

Suppose we store RS(A,B,C,D,E) to avoid joins