CSCE 608 Database System
Final Review Sheet

N/A

May 3, 2018

1 Note 1 to 6

Basic SQL reviews

2 Note 7

Memory structure review - won't be in the final

3 Note 8

For a given database program P, how to understand it?

1. Break the program P into "words" (lexical analysis);
2. Use the language grammar to "parse" the program P;
3. Make a parse tree (A tree structure that shows how the program P follows the grammar rules) for the program P.

4 Note 9

Relational algebra: S = the elements are tables (i.e., relations). Operations = selection σ, projection π, set operations ($\cap$, $\cup$, $\setminus$), renaming ρ, joins ($\times$, $\bowtie$, $\bowtie_C$), extended operations (δ - duplicate elimination, γ - sorting, τ - grouping, outer-joins)

Expressions in relational algebra are presented by expression trees. See HW1 p1 for details.

General execution order of SELECT: FROM - WHERE - SELECT (bottom-up approach when **only simple relations and conditions are involved**)

5 Note 10

Handle subqueries in logic plan conversion:

1. Convert the subquery into a logic plan;
2. Collect the attributes for the condition;
3. Make a cross product with the "main" relations;
4. Apply the conditions in the cross product.

6 Note 11

View (virtual table) processing: for each CREATE VIEW statement, construct its parse tree. In the parse tree for a statement that uses the view, replace the view name with the parse tree for the view.

6.1 Improve logical query plan via logical laws

Move selections σ so they can be applied early (σ reduces table size. Specifically: the size of the intermediate relations).

$\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$

$\sigma_C(R \cap S) = \sigma_C(R) \cap \sigma_C(S) = \sigma_C(R) \cap S = R \cap \sigma_C(S)$

$\sigma_C(R - S) = \sigma_C(R) - \sigma_C(S) = \sigma_C(R) - S$

$\sigma_{CandD}(R) = \sigma_c(\sigma_D(R)) = \sigma_D(\sigma_C(R))$

$\times$, $\bowtie$, $\bowtie_D$: σ_C can only be pushed to the arguments that have all attributes in C. It is not advised to do so.

7 Note 12

Dealing with projections π

General rule: a projection can be added anywhere as long as the eliminated attributes neither are used by any operation above the projection nor appear (explicitly/implicitly) in the final expression.

e.g. $\pi_L(\sigma_C(R)) = \pi_L(\sigma_C(\pi_{L,L(C)}(R)))$

Dealing with duplicate elimination δ

δ can be pushed in join operations

$\delta(R \times S) = \delta(R) \times \delta(S), \delta(R \bowtie S) = \delta(R) \bowtie \delta(S), \delta(R \bowtie_C S) = \delta(R) \bowtie_C \delta(S)$

δ can swap with selections: $\delta(\sigma_C(R)) = \sigma_C(\delta(R))$ **BUT NOT projections**

δ can be pushed in set operations and bag-intersection $\cap_B$, **BUT NOT other bag operations**

$\delta(R \cup_S S) = \delta(R) \cup_S \delta(S), \delta(R \cap_S S) = \delta(R) \cap_S \delta(S), \delta(R -_S S) = \delta(R) -_S \delta(S), \delta(R \cap_B S) = \delta(R) \cap_B \delta(S)$

Dealing with grouping γ and sorting τ: more case dependent.

Grouping reduces table size, but could be expensive if the table is unorganized.

Sorting unchanges table size, and is expensive. But it can be extremely helpful for later computation.

General remarks on LQP optimization: No transformation is always good, except pushing selections down.

Commutative operator: R * S = S * R

Associative operator: (R * S) * T = R * (S * T)

$\cap, \cup, \times, \bowtie$: both commutative and associative (but not $\bowtie_D$ and -). Group each commutative and associative binary operator into a "group".

8 Note 13

8.1 Improve logical query plan via relation size

1. Collect size parameters for stored relations: T(R), B(R), V(R,A) (the # of different values on attribute A)
2. Set up estimation rules for size parameters on relational algebraic operators;
3. Using logic laws to convert a logic query into the one that minimizes the (estimated) sizes of intermediate relations.

8.1.1 Estimate size parameters T: (# of tuples)

$\pi, \tau, \times$: size parameters can be calculated precisely

$\delta : T(\delta(R)) = min\{T(R)/2, \prod_A V(R, A)\}$ ($\prod_A V(R, A$: products of all attribute values from R and A)

$\gamma : T(\gamma(R)) = min\{T(R)/2, \prod_{grouping_A} V(R, A)\}$

$\sigma_{A=c} : T(\sigma_{A=c}(R)) = T(R)/V(R, A)$

$\sigma_{A<c} : T(\sigma_{A<c}(R)) = T(R)/3$

$\cap : T(R \cap S) = T(S)/2$ (assume S is smaller)

$\cup : T(R \cup S) = T(R) + T(S)/2$ (assume S is smaller)

$- : T(R - S) = T(R) - T(S)/2$

$\bowtie$: $T(R \bowtie S) = T(R)T(S)/max\{V(R, A), V(S, A)\}$

$\bowtie_C$: $T(R \bowtie_C S) = T(\sigma_C(R \times S))$

8.1.2 Estimate size parameters B: (# of blocks)

If we know the schemas of relations R and S, we will also know the schema of the relation W obtained by applying an operation on R and/or S, from which we know how much space a tuple in W will take. Therefore, the value B(W) can be computed from the value T(W).

$B(W) = T(W)/\#tuples - per - block$

8.1.3 Estimate size parameters V: (# of distinct values)

$\pi, \tau, \times$: size parameters can be calculated precisely
$\delta : V(\delta(R), A) = V(R, A)$
$\gamma : T(\gamma(R), A) = V(R, A)$ (A is a grouping attribute)
$\sigma_{A=c} : V(\sigma_{A=c}(R), A) = 1, V(\sigma_{A=c}(R), D) = V(R, D)$
$\sigma_{A<c} : V(\sigma_{A<c}(R), D) = V(R, D), V(\sigma_{A=c}(R), A) = V(R, A)/3$
$\cap : V(R \cap S, A) = V(S, A)/2$ (assume V(R,A) $\geqslant$ V(S,A))
$\cup : V(R \cup S, A) = V(R, A) + V(S, A)/2$ (assume V(R,A) $\geqslant$ V(S,A))
$- : V(R - S, A) = V(R, A) - max\{R(R, A)/2, T(S, A)/2\}$
$\bowtie$: $V(R \bowtie S, A) = min\{V(R, A), V(S, A)\}$ (A is a shared attribute)
$V(R \bowtie S, D) = max\{V(R, D), V(S, D)\}$ (D is non-shared)
$\bowtie_C$: $V(R \bowtie_C S, A) = V(\sigma_C(R \times S), A)$
Similar to that for the parameter T

9 Note 14

9.1 Using logic laws to convert a logic query into the one that minimizes the (estimated) sizes of intermediate relations.

Basically it means to combine the logical laws and relation sizes estimation as mentioned in the previous sections to estimate the cost. **Goal: minimize the cost.**

9.2 Logic Plan Improvement for Join via Size

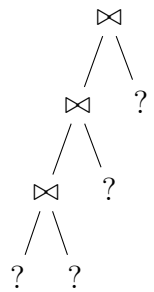
Two techniques:

- Estimating sizes of immediate relations
- Consider different order of an operation

  left-deep tree

  dynamic programming

9.2.1 Left-deep join tree



Rule: when two relations R, S are natural joined with each other via a common attribute y, the cost formula would be $T(R(X, y) \bowtie S(y, Z)) = T(R) * T(S)/max\{V(R, y), V(S, y)\}$.

Three relations? $T(R(X,y,a) \bowtie S(y,Z,a)) = T(R) * T(S)/(max\{V(R,y), V(S,y)\} * max\{V(R,a), V(S,a)\})$ and so on...

It is fine to have decimal place when estimating the intermediate relation size. That simply means that after joining on so many attributes the resulting set is really small.

Only needs to estimate intermediate relation sizes.

Left-deep tree: general algorithm
Input: n relations $R_1, R_2, , R_n$
Output: the best left-deep join of $R_1, R_2, , R_n$

1. Construct a left-deep tree T of n leaves;
2. For each P of the permutations of the n relations $R_1, R_2, , R_n$ Do
   assign the n relations to the leaves of T in order of P;
   evaluate the cost of the plan;
3. Pick the plan with the permutation that gives the minimum cost.

9.2.2 Dynamic Programming

Scope: expands to all tree structures
How to find the best join? Consider all possible ways and calculate the cost!

Dynamic programming: general algorithm
Input: n relations $R_1, R_2, , R_n$
Output: the best join of $R_1, R_2, , R_n$

1. FOR each R_i DO cost(R_i) = 0; size(R_i) = 0;
2. FOR each pair of R_i and R_j DO cost(R_i, R_j) = 0; compute size($R_i \bowtie R_j$);
3. FOR k = 3 TO n DO
   FOR any k relations $S_1, S_2, , S_k$ of $R_1, R_2, , R_n$ DO
   FOR each partition P = $\{(S_{i1}, , S_{ij}), (S_{ij+1}, , S_{ik})\}$ of $S_1, S_2, , S_k$ DO
   cost(P) = cost($S_{i1}, , S_{ij}$) + size($S_{i1}, , S_{ij}$) + cost($S_{ij+1}, , S_{ik}$) + size($S_{ij+1}, , S_{ik}$);
   let cost($S_1, S_2, , S_k$) be the smallest cost(P) among the above partitions;
   compute size($S_1, S_2, , S_k$) (and remember this partition P);
4. Return cost($R_1, R_2, , R_n$).

RLIN Note: Mostly, left deep tree is preferable as it produced less cost and it is friendly for one pass algorithm. But this is not absolute. Still needs to use dynamic programming to calculate costs under different join scenarios to find the best match. (See the slides for examples)

10 Note 15

10.1 Construction of Physical Query Plan

Input: an optimized LQP T, and main memory constraint M

1. Replacing each leaf R of T by "scan(R)";
2. Combining the "scan's" with other operations;
3. Replacing each internal node v of T by a proper algorithm;
4. For each edge e in T, decide if e should be "materialized"; (i.e. create a temporary relation and write back to the disk)
5. Cut all materialized edges;
6. Each subtree is a call to the subroutine at the root of the subtree. The order of the calls follows the bottom-up order in the structure.

Comment: This produces an executable code for the input DB program.

10.2 Physical Query Plan: Summary

- Replacing internal nodes of a LQP by proper algorithms;
- Deciding if a subroutine call should be pipelined or materialized;
- Many optimization techniques are involved here;
- In practice, heuristic optimization techniques are used to construct good physical query plans;
- The resulting physical query plan is an executable code.

10.3 Index Structures: B+ Tree

Main purpose: to speed up the search process. Indexes can quickly identify blocks that contain desired tuples, instead of looping through the whole relation.

Characteristics of B+ tree:

- Support fast search
- Support range search
- Support dynamic changes
- Could be either dense (pointing to all records) or sparse (one pointer per block)

Typically, the order n of B+ tree is high (around 100 to 200).
Each node has **n** keys and **n+1** pointers
Basic rule: use at least one half of the pointers.

10.3.1 B+ tree rules

1. All leaves are at same lowest level (balanced tree)
2. Pointers in leaves point to records except for the "sequence pointer" (connecting the two leaf nodes).
3. Number of keys/pointers in nodes:

	Max. # pointers	Max. # keys	min. # pointers	min. # keys
Non-leaf	n+1	n	Ceiling[(n+1)/2]	Ceiling[(n+1)/2] -1
Leaf	n+1	n	Floor[(n+1)/2] + 1	Floor[(n+1)/2]
Root	n+1	n	2	1

10.3.2 Search in B+ Tree

In general: start from the root, search the leaf block, and there might not be a need to go back to the data file.

Algorithm:

Search(ptr, k);

\\ search a record of key value k in the subtree rooted at ptr

\\ assume the B+tree is a dense index of order n

Case 1. ptr is a leaf \\p_{n+1} is the sequence pointer

IF $(k = k_i)$ for a key k_i in *ptr THEN return(p_i);

ELSE return(Null);

Case 2. ptr is not a leaf

find a key k_i in *ptr such that $k_{i-1} \leqslant k < k_i$;

return(Search(pi, k));

11 Note 16

11.0.1 Range Search in B+ Tree

Algorithm:
Range-Search(ptr, k_1, k_2):
Call Search(ptr, k_1);
Follow the sequence pointers until the search key value is larger than k_2.

11.0.2 Insert into B+ tree

Could be really complicated, especially at the case of non-leaf overflow

General idea: when there is no space for a new child (all pointers are in use), split the node into two nodes, with both at least half full.

Pseudo code: see below.

Insert(*pt*, (p, k), (p', k')); \\ technically, the smallest key k_{min} in *pt* is also returned
\\ (p, k) is a pointer-key pair to be inserted into the subtree rooted at *pt*; p' is a new sibling
\\ of *pt*, if created, and k' is the smallest key value in p' ;

Case 1. *pt* = $(p_1, k_1, ..., p_i, k_i, --, p_{n+1})$ is a leaf \\ p_{n+1} is the sequence pointer

IF $(i < n)$ **THEN** insert (p, k) into *pt*; return($p' =$ Null, $k' = 0$);

ELSE re-arrange (p_1, k_1), ..., (p_n, k_n), and (p, k) into $(\rho_1, \kappa_1, ..., \rho_{n+1}, \kappa_{n+1})$;

create a new leaf p''; put $(\rho_{r+1}, \kappa_{r+1}, \ldots, \rho_{n+1}, \kappa_{n+1}, --, p_{n+1})$ in p''; \\ $r = \lfloor (n+1)/2 \rfloor$

put $(\rho_1, \kappa_1, ..., \rho_r, \kappa_r, --, p'')$ in *pt*; \\ p_{n+1} and p'' are sequence pointers in p'' and *pt*.

IF *pt* is the root **THEN** create a new root with children *pt* and p'' (and key κ_{r+1})

ELSE return($p' = p''$, $k' = \kappa_{r+1}$);

Case 2. *pt* is not a leaf

find a key k_i in *pt* such that $k_{i-1} \le k < k_i$;

Insert(p_i, (k, p), (k'', p''));

IF ($p'' =$ Null) **THEN** return($k' = 0$, $p' =$ Null);

ELSE IF there is room in *pt*, **THEN** insert (k'', p'') into *pt*; return($k' = 0$, $p' =$ Null);

ELSE re-arrange the content in *pt* and (k'', p'') into $(\rho_1, \kappa_1, ..., \rho_{n+1}, \kappa_{n+1}, \rho_{n+2})$;

create a new node p''; put $(\rho_{r+1}, \kappa_{r+1}, \ldots, \rho_{n+1}, \kappa_{n+1}, \rho_{n+2}, --)$ in p''; \\ $r = \lceil (n+1)/2 \rceil$

leave $(\rho_1, \kappa_1, ..., \rho_{r-1}, \kappa_{r-1}, \rho_r, --)$ in *pt*;

IF *pt* is the root **THEN** create a new root with children *pt* and p'' (and key κ_r)

ELSE return($p' = p''$, $k' = \kappa_r$).

67

11.0.3 Delete in B+ tree

May involved in leaf/non leaf coalesce and key redistribution.

Note: Often, coalescing is not implemented - Too hard and not worth it!

Pseudo code: see below.

Delete(*pt*, (*k*,*p*), belowmin); \\ technically, the smallest key k_{min} in **ptr* is also returned
\\ (*k*,*p*) is the data record to be deleted from the subtree rooted at *pt*; belowmin = *true* if
\\ after deletion, *pt* has fewer than the required min # of pointers;

Case 1. *pt* is a leaf
  delete (*k*,*p*) in *pt*;
  IF *pt* has at least $\lfloor (n+1)/2 \rfloor$ data pointers OR *pt* is the root
  THEN return (belowmin = *false*) **ELSE** return (belowmin = *true*);

Case 2. *pt* is not a leaf
  find a key k_i in *pt* such that $k_i \leq k < k_{i+1}$;
  Delete(p_i, (*k*, *p*), belowmin′);
  IF (*not* belowmin′) **THEN** return(belowmin= *false*);
  ELSE IF p_i has an adjacent sibling p' that has more than the required min # of pointers
  THEN move one key-pointer pair from p' to p_i;
  ELSE combine p_i with an adjacent sibling of p_i into a single node;
    IF *pt* is the root with only one pointer p_i
    THEN *pt* = p_i; return(belowmin= *false*);
    IF *pt* has at least $\lceil (n+1)/2 \rceil$ pointers OR *pt* is the root
    THEN return(belowmin= *false*) **ELSE** return(belowmin= *true*);

151

12 Note 17

12.1 Index Structures: B Tree

Summary: B tree has no duplicate keys. It has record pointers in non-leaf nodes, therefore eliminated the need of having sequence pointers.

Trade off:
✓B trees have faster lookup than B+trees
× in B tree, non-leaf & leaf different sizes
× in B tree, insertion and deletion are more complicated
B+ tree is prefered!

12.2 Index Structures: Hash Tables

Hash the search keys and put them into different buckets.

Hashed values in the buckets could be direct (address from the hash function) or indirect (indexes in a directory).

Rule of thumbs: Try to keep the space utilization between 50% to 80%. Utilization = $\frac{\#keys-used}{total-\#-keys-that-fit}$

To cope with growth: **Overflows/reorganizations** and **Dynamic hashing** are important.

12.2.1 Extensible Hashing

Note: extensible hashing uses the **higher order** bits (from the left) to find the elements. Basic ideas:

1. Fix a hash function h that can handle a very large hash table, but also use h for smaller hash tables (so the hash value can be used even when the hash table is changed)
2. When the set is small, use only part of the value h(x).
3. If local population is even sparser, we use even fewer bits of h(x). Adjacent elements in the hash table can be merged.
4. Use a directory.

Constructing an Extensible Hashing for A Given Set S

Input: A set S of elements, hash function h into b bits (b is large),

1. make S a group G_0 with group index 0;
   \\ each group G_k is associated with a unique string s_k of k bits such that all
   \\ elements in G_k have their hash values with the first k bits equal to s_k
2. WHILE there is a group G_k that cannot fit in a single block DO
   split G_k into two groups $G^{(1)}_{k+1}$ and $G^{(2)}_{k+1}$ using
   the $(k+1)$st bit in the hash values of the elements;
3. FOR each group G_k DO
   put all elements in G_k into a block, with a block index k;
4. the largest block index is the hashing index i;
5. make a directory $H[\overbrace{00\cdots0}^{i} .. \overbrace{11\cdots1}^{i}]$ of size 2^i with H[r] pointing to the block for group G_k, if the first k bits of r is equal to the group string s_k.

48

13 Note 18

13.0.1 Search in Extensible Hashing

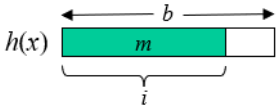
input: a search key x
\\ h is the hash function, H is the directory, i is the current hash index.

1. m = the first i bits of h(x);
2. read in the disk block B with the address H[m].

13.0.2 Insertion in Extensible Hashing

input: a tuple t with search key x

\\ h is the hash function, H is the directory, i is the current hash index.

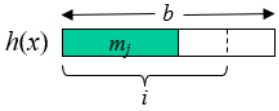


1. m = the first i bits of $h(x)$;
2. let the block with the address $H[m]$ be B;
3. IF B has room THEN add t in B
4. ELSE let j be the block index of B

   IF $i = j$ THEN

   {double the size of H to 2^{i+1}, $i = i + 1$; let the pointers in the new $H[2k]$ and $H[2k+1]$ both equal to that in the old $H[k]$, $0 \le k \le 2^i$; }

   split $B \cup \{t\}$ into B_0 and B_1 (with block index j+1) using the j+1st bit, let $H[m_j0^{**}]$ point to B_0 and $H[m_j1^{**}]$ point to B_1.



Note: Still need overflow chains, otherwise inserting duplicate keys can cause an issue.

13.0.3 Deletion in Extensible Hashing

Two cases: Either no merging of blocks, or merge blocks and cut directory if possible (basically to reverse the insertion). It's more complicated and should be done at the time when the server is not busy.

13.0.4 Summary of Extensible Hashing

✓Can handle growing files...
- with less wasted space
- with no full reorganizations

× Indirection (Not bad if directory in memory)
× Directory doubles in size (Now it fits, now it does not) - could be too sparse

13.0.5 Linear Hashing

Note: linear hashing uses the **lower order** bits (from the right) to find the elements. Similar to extensible hashing, it uses the same hash function h that is accountable for really large b.

Difference: Hash table size n grows linearly; use overflow blocks.

If $h(x)_i \geqslant n$: put x in $h(x)_i - 2^{i-1}$ (which replaces the leading bit 1 by 0)

13.0.6 Search in Linear Hashing

input: a search key x
\\ h is the hash function, n is the current upper bound, $i = |n|$

1. m = the last i bits of $h(x)$;
2. IF $m \geq n$ THEN $m = m - 2^{i-1}$;
3. read in the disk block(s) with the address m
   \\ you should check overflow blocks in the address m.

13.0.7 Insertion in Linear Hashing

input: a tuple t with search key x
\\ h is the hash function, n is the current upper bound, $i = |n|$

1. m = the last i bits of $h(x)$;
2. IF $m \geq n$ THEN $m = m - 2^{i-1}$;
3. insert t in the disk block B with the address m;
   \\ If B is full, you need to use an overflow block.

13.0.8 Deletion in Linear Hashing

input: a tuple t with search key x

\\ h is the hash function, n is the current upper bound, $i = |n|$

1. m = the last i bits of $h(x)$;
2. IF $m \geq n$ THEN $m = m - 2^{i-1}$;
3. Delete t in the disk block B with the address m;
   \\ you may need to check overflow blocks.

13.0.9 Linear Hashing: Increasing Hash Table Size

Motivation: if R ($\frac{needed-space}{available-space}$) is greater than the threshold (says 80%), then it's necessary to increase the hash table size.

input: the current upper bound n

\\ h is the hash function, $i = |n|$

1. read in the disk block(s) B of address $n - 2^{i-1}$;
2. split (properly) the tuples in B and put them in the block B and the block B' with address n;
3. $n = n + 1$;

7

13.0.10 Linear Hashing: Shrinking Hash Table Size

Motivation: if R ($\frac{needed-space}{available-space}$) is less than the threshold (says 60%), then it's necessary to decrease the hash table size.

input: the current upper bound n

\\ h is the hash function, $i = |n|$

1. $n = n - 1$;
2. move the tuples in the block(s) of address n to the block(s) of address $n - 2^{i-1}$

   (here i is the length of the new n).

Basic idea: whenever expanding or shrinking the hash table, just need to check the blocks with address $n - 2^{i-1}$.

13.0.11 Summary of Linear Hashing

✓Can handle growing files...
- with less wasted space
- with no full reorganizations

✓No indirection like extensible hashing
× Overflow chains

14 Note 19

To model the performance in DBMS: count the number of Disk I/Os.

Assumptions:
inputs are on disk (so must be read in)
But output is not written back to disk (may not have to; hard to estimate output size, which also does not depend on the adopted algorithms)

14.1 Operations requiring (almost) no M

Tuple-based operations: $\pi, \sigma, \cup_B$, table-scan

General framework:

1. Read in a block;
2. Process;
3. Send to the output.

Memory: M = 2
Cost:
$\pi(R), \sigma(R)$, table-scan(R): B(R)
$\cup_B(R, S)$: B(R)+B(S)

14.2 One-pass Algorithms

Condition: the main memory M is sufficiently large

General framework for unary operations:

1. Read in an entire relation R;
2. Process R;
3. Read in the other relation S block by block;
4. Sent the results to an output block.

For unary operations $\gamma(R), \delta(R), \tau(R)$:
Memory: M $\geqslant$ B(R)
Cost: B(R)

General framework for binary operations:

1. Read in an entire relation R_{small};
2. Process R_{small};
3. Read in the other relation R_{large} block by block;
4. Sent the results to an output block.

For binary operations $\cup_S, \cap_S, -_S, \cap_B, -_B, \times, \bowtie, \bowtie_C$:
Memory: M $\geqslant B(R_{small})$
Cost: $B(R_{small}) + B(R_{large})$

Algorithms piece by piece:

1. $R_{large} \cup_S R_{small}$
   (a) Sort R_{small}
   (b) For each tuple t in R_{large} DO
   IF t is not in R_{small}
   THEN put t to the output;
   (c) Send R_{small} to the output.
2. $R_{large} \cap_S R_{small}$

(a) Sort R_{small}

(b) For each tuple t in R_{large} DO
IF t in R_{small}
THEN put t to the output;

3. $R_{large} -_s R_{small}$ (Note that $-_S$ is not commutative)

(a) Sort R_{small}

(b) For each tuple t in R_{large} DO
IF t is not in R_{small}
THEN put t to the output;

4. $R_{small} -_s R_{large}$

(a) Sort R_{small}

(b) For each tuple t in R_{large} DO
IF t is in R_{small}
THEN remove t from R_{small}

(c) Send R_{small} to the output.

5. $R_{large} \cap_B R_{small}$

(a) Make R_{small} a balanced tree;

(b) FOR each tuple t in R_{large} DO
IF t is in R_{small} THEN
output t; and remove a copy of t from R_{small}

6. $R_{large} -_B R_{small}$ (Note that $-_B$ is not commutative)

(a) Make R_{small} a balance tree;

(b) FOR each tuple t in R_{large} DO
IF t is not in R_{small} THEN output t
ELSE remove a copy of t from R_{small};

7. $R_{small} -_B R_{large}$

(a) Make R_{small} a balance tree;

(b) FOR each tuple t in R_{large} DO
IF t is in R_{small} THEN remove a copy of t from R_{small};

(c) Output R_{small}

8. $R_{large} \times R_{small}$

FOR each tuple t in R_{large} DO
cross join t and each tuple in R_{small} and send to the output.

9. $R_{large} \bowtie_C R_{small}$

   FOR each tuple t in R_{large} DO

   cross join t and each tuple in R_{small} ;

   IF the join satisfies C THEN send to the output.

10. $R_{large} \bowtie R_{small}$

    (a) sort R_{small} by join attributes A;

    (b) FOR each tuple t in R_{large} DO

    find the tuples in R_{small} with the same A-value;

    join them with t and put in the output block

14.3 Nested Loop Algorithms

When the relations are too large to fit into the main memory, one pass algorithm cannot be used.

Nested loop: A generic algorithm for **binary operations**

General framework: (Nested loop R $\square$ S)
FOR each tuple t_R in R DO
FOR each tuple t_S in S DO
Apply the operation $\square$ on t_R and t_S

Algorithms piece by piece:

1. $R \cup_S S$

   (a) \\ in the first execution of the t_R-loop
   output t_S

   (b) \\ in an execution of the t_S-loop
   IF $t_R = t_S$ THEN mark t_R;

   (c) \\ at the end of the t_R-loop
   IF t_R is unmarked THEN output t_R.

2. $R \cap_S S$

   (a) \\ in an execution of the t_R-loop
   IF $t_R = t_S$ THEN mark t_R;

   (b) \\ at the end of the t_R-loop
   IF t_R is marked THEN output t_R.

3. $R -_S S$ (Note: $S -_S R$ cannot use this algorithm as set difference is not commutative)

(a) \\ in an execution of the t_R-loop
IF $t_R = t_S$ THEN mark t_R;

(b) \\ at the end of the t_R-loop
IF t_R is unmarked THEN output t_R.

4. For $R \cup_B S$ and $R \cap_B S$: Nested-loop does not seem to be effective since we cannot mark S. So it's not applicable here.

5. R joins S

   IF t_R and t_S are joinable THEN

   Join t_R and t_S;

   IF the join is $\times$ or $\bowtie$ THEN output the join;

   ELSE \\ the join is $\bowtie_C$

   output the join if it satisfies C

Memory: M $\geqslant$ 2
Cost: T(R)*T(S) + T(R) (Really bad)

Optimization 1: change the inner loop - instead of looping through t_S, now loop through B_S.
New cost: T(R)*B(S) + T(R) (Still large)

Optimization 2: in addition to optimization 1, also change the outer loop - instead of looping through t_R, now loop through B_R.
New cost: B(R)*B(S) + B(R) (Better)

Optimization 3: in addition to optimization 2, try to fit the main memory with the maximum numbers of blocks.
New cost: $B(R) * B(S)/M$ + B(R) (Very good if B(R) or B(S) is only slightly larger than M. The second term B(R) suggests that we should pick the **smaller** relation for the outer loop.)

Quick review:

Operations requiring almost no space: π, σ, $\cup_B$, table-scan	One-pass Algorithms: γ, δ, τ, $\cup_S$, $\cap_S$, $-_S$, $\cap_B$, $-_B$, $\times$, $\bowtie$, $\bowtie_C$	Nested-loop Algorithms For binary operations: $\cup_S$, $\cap_S$, $-_S$, $\times$, $\bowtie$, $\bowtie_C$
Memory: $M = 2$ Cost: π (R), σ(R), table-scan(R): B(R) $\cup_B$(R, S): B(R) + B(S)	**Unary:** Memory: $M \geq B(R)$ Cost: B(R) **Binary:** Memory: $M \geq B(R_{small})$ Cost: $B(R_{small}) + B(R_{large})$	Memory: $M \geq 2$ Cost: B(R)*B(S)/M + B(R)

15 Note 20

15.1 Two-pass Algorithms

Conditions: large relations that cannot fit into the main memory M, but they are not extremely large.

Basic ideas:

1. Sort: Break relations into smaller pieces that fit in the main memory, make them more organized, and store them back to disk;
2. Merge: Apply operation based on "merging" the blocks from the smaller and more organized pieces.

Two types: sorted based or hashed based.

15.2 Two-pass sort-based algorithms

General framework:

1. Repeat (Phase 1. making sorted sublists)

   Fill the main memory with tuples of a relation;

   Make them a sorted sublist;

   Write the sorted sublist back to disk.

2. Repeat (Phase 2. Merging)

   Bring in a block from each of the sorted sublists;

   apply operation by "merging" the sorted blocks;

Note: at the merging phase, if a sublist is exhausted, we will go back to the disk and grab the next sublist. If all sublists from a block are exhausted, then we would ignore that block and continue to work on the remaining sublists/blocks that are not finished yet. (May need a heap structure in the main memory to effectively handle searching)

Algorithms piece by piece **for unary operations:**

1. $\delta(R)$
   (a) Remove all minimums except one;
   (b) Output the minimum.
2. $\gamma(R)$ (sublists are sorted by the grouping attributes)
   (a) Group all tuples with the minimum grouping attributes;
   (b) Calculate the aggregation value;
   (c) Output a grouping tuple.
3. $\tau(R)$: sorting has already been handled during the first phase.

Memory: M $\geqslant \sqrt{B(R)}$
Cost: 3B(R) (read in, output to disk, read in again)

Algorithms piece by piece **for binary operations:**

1. $R \cup_S S$: REPEAT
   IF $R_{min} = S_{min}$ THEN
   send one copy to output; delete both;
   ELSE
   send the smaller to output, and delete it.
2. $R \cap_S S$: REPEAT
   IF $R_{min} = S_{min}$ THEN
   send one copy to output; delete both;
   ELSE
   delete the smaller.

   Note: The same algorithm works for R $\cap_B$ S!
3. $R -_S S$: REPEAT
   IF $R_{min} = S_{min}$ THEN
   delete both;
   ELSE IF $R_{min} > S_{min}$ THEN

delete S_{min};

ELSE \\$R_{min} < S_{min}$

send R_{min} to output, and delete R_{min}.

Note: The same algorithm works for R $-_B$ S!

4. R $\times$ S: Sorted sublists do not seem to help; each tuple of R should be joined with all tuples of S. Use nested loop instead.

5. R $\bowtie_C$ S: Same as R $\times$ S above, sorted sublist does not help. Use nested loop instead.

6. R $\bowtie$ S: in extreme case, cannot use the sort-based algorithm. But normally it works.

   \\ sublists are sorted by the join attributes.

   REPEAT

   IF $R_{min} = S_{min}$ THEN

   collect all R_{min}-tuples and all S_{min}-tuples, and send their join to the output; delete all R_{min} and S_{min} tuples;

   ELSE delete the smaller;

Memory: M $\geqslant \sqrt{B(R) + B(S)}$
Cost: 3(B(R)+B(S)) (read in, output to disk, read in again)

Comments: Applicable unless the relations are extremely large, in that case we can extend the method to multiway pass. The output is also sorted.

16 Note 21

16.1 Two-pass hash-based algorithms

General framework: (RLIN: assuming a good hash function so that each bucket is around the same size)

1. Phase 1. making hash buckets

   Hash tuples into M buckets (using one block for each bucket); write the buckets back to disk.

2. Phase 2. bucketwise operation

   apply the operation based on buckets.

Algorithms piece by piece **for unary operations:**

1. $\delta(R)$

   \\ All duplicates are in the same bucket.

   Call one-pass algorithm on each bucket.

   Note: M must be large enough to hold an entire bucket.

2. $\gamma(R)$

   Use the same algorithm for $\delta(R)$, assuming that hash is based on the grouping attributes.

3. $\tau(R)$: Hash-based algorithm does not apply for sorting

Memory: M $\geqslant \sqrt{B(R)}$
Cost: 3B(R) (read in, output to disk, read in again)

Algorithms piece by piece **for binary operations:**

1. $R \cup_S S$: (Same algorithm works for $\cap_S, -_S, \cap_B, -_B$)

   FOR each bucket index i DO

   call the one-pass algorithm on the R_i-bucket and S_i-bucket

2. R $\times$ S: Hash-based algorithm does not work. Use nested loop instead.

3. R $\bowtie_C$ S: Hash-based algorithm does not work. Use nested loop instead.

4. R $\bowtie$ S: hash based on join attributes

   FOR each bucket index i DO

   call the one-pass algorithm on the R_i-bucket and S_i-bucket

Memory: M $\geqslant \sqrt{B(R_{small})}$
Cost: 3(B(R)+B(S)) (read in, output to disk, read in again)

Comments: Memory use is better than sorted-based; the output is not sorted; required a good hash function.

16.1.1 Trick to save some disk I/Os?

1. Use fewer buckets to leave some free M space;

2. Use the free M space to hold some buckets so they are not written back to disk (so save Disk I/O's);

3. Operation on these buckets are performed directly.

e.g. Say the memory size M = 5, keep k = 2 buckets in M, only write D = M k = 3 buckets back to disk. 2 disk I/Os are saved!

How many (k) buckets should be left in M though??
Answer: Doesn't matter. More critical is the amount of M space used for holding these buckets.

As k grows, the bucket size would be smaller. Since the amount of data stays the same, there would be more buckets in total. Thus, more blocks in the main memory need to be

reserved for those bucket blocks. Since the size of the main memory is fixed, less M space would be available for holding the buckets...

$\therefore$ **k should be as small as possible: k=1.**

How many disk I/O's we have saved? (In general)

For binary operations, assumed that the memory meets the requirement $M \geqslant \sqrt{B(S)/c(1-c)}$. S has a smaller size.

1. Let bucket size = c * M, $0 < c < 1$. Number of buckets = $(1-c)M + 1 \approx (1c)M$; $\backslash\backslash B(S) = c(1-c)M^2$, c is the larger root of $c^2 - c + B(S)/M^2$
2. S saves 2c * M = 2c * B(S) * (M/B(S)) disk I/O's
3. R saves 2B(R)/(1-c)M = 2c * B(R) * (M/B(S)) disk I/O's
4. Original cost without these savings: 3(B(R) + B(S))
5. So now the cost is $(3 - 2c$ * M/B(S))(B(R) + B(S))

For unary operations, the same trick applies. The cost can be reduced to 3B(R) - M.

Quick review:

Summary

Operations requiring almost no space: π, σ, $\cup_B$, table-scan	One-pass Algorithms: γ, δ, τ, $\cup_S$, $\cap_S$, $-_S$, $\cap_B$, $-_B$, $\times$, $\bowtie$, $\bowtie_C$	Nested-loop Algorithms For binary operations: $\cup_S$, $\cap_S$, $-_S$, $\times$, $\bowtie$, $\bowtie_C$
Memory: M = 2 Cost: π (R), σ(R), table-scan(R): B(R) $\cup_B$(R, S): B(R) + B(S)	**Unary:** Memory: M ≥ B(R) Cost: B(R) **Binary:** Memory: M ≥ $B(R_{small})$ Cost: $B(R_{small}) + B(R_{large})$	Memory: M ≥ 2 Cost: B(R)*B(S)/M + B(R)

Two-pass sort-based algorithms: γ, δ, τ, $\cup_S$, $\cap_S$, $-_S$, $\cap_B$, $-_B$, $\bowtie$	Two-pass hash-based algorithms: γ, δ, $\cup_S$, $\cap_S$, $-_S$, $\cap_B$, $-_B$, $\bowtie$	Two-pass hybrid hash-based: γ, δ, $\cup_S$, $\cap_S$, $-_S$, $\cap_B$, $-_B$, $\bowtie$
Unary: Memory: M ≥ $\sqrt{B(R)}$ Cost: 3B(R) **Binary:** Memory: M ≥ $\sqrt{B(R) + B(S)}$ Cost: 3(B(R) + B(S))	**Unary:** Memory: M ≥ $\sqrt{B(R)}$ Cost: 3B(R) **Binary:** Memory: M ≥ $\sqrt{B(R_{small})}$ Cost: $3(B(R_{small}) + B(R_{large}))$	**Unary:** Memory: M ≥ $2\sqrt{B(R)}$ Cost: 3B(R) − M **Binary:** Memory: M ≥ $\sqrt{B(R_{small})/c(c-1)}$ Cost: $(3 - 2c \cdot M/B(R_{small})) \cdot (B(R_{small}) + B(R_{large}))$ (c is the larger root of $c^2-c+B(S)/M^2$)

65

π, σ, $\cup_S$, $\cap_S$, $-_S$, $\cup_B$, $\cap_B$, $-_B$, γ, δ, τ, table-scan, $\times$, $\bowtie$, $\bowtie_C$

17 Note 22

System may fail due to various reasons, but data should be accurate/correct all the time.

Data in consistent state: Data that satisfy all (logic/realistic) constraints
Consistent database: Database that contains data in consistent state
Challenge: a database cannot always be consistent!

What is a transaction?
A sequence of actions that we want to be done as a single "logic step"

What is a logical step (what do we want a transaction to be)?
ACID
Atomicity: either all steps of a transaction happen, or none happen
Consistency: a transaction transforms a consistent DB into a consistent DB
Isolation: execution of a transaction is isolated from that of other transactions
Durability: if a transaction commits, its effects persist.

What other things can affect transaction ACID?

- Accident system failures may destroy Atomicity and Durability, thus Consistency;
- Concurrent executions of transactions (which is highly desired) may affect Isolation, thus Consistency.

To cope with system failures, the logging strategy must be carefully designed to ensure Atomicity and Durability.

17.1 Undo-logging

Main idea: When log record <T, commit> reaches disk, **all changes** made by T have reached disk.

Rules:
The log record <T, V, a_{old} > reaches disk before the change of V reaches disk;
The log record <T, commit> reaches disk after all changes made by T reach disk.

18 Note 23

Algorithms for undo-logging:

Log recording

1. When T writes on A, add a log-record <T, A, a_{old}> (in memory-log);
2. Before output(A), make sure the memory-log is flushed to disk-log at least once;
3. Add log-record <T, commit> to memory-log only when all output(A) related to T are done.

System recovery with disk-log

1. Scan the disk-log backwards;
2. Ignore T if see <T, commit>;
3. For each T with no <T, commit>
   For each log-record <T, A, a_{old}>
   Undo the action by restoring the value of A back to a_{old};
   Add <T, abort> to disk-log.

Basically, logging is just inserting actions in the normal execution steps within a transaction.

Remarks:

- To execute the recovery algorithm, we need to
  - Read the disk-log into the main memory;
  - FOR each <T, A, a_{old} > in step 3 DO
    write(A, a_{old}); output(A);
- What if the system crashes again during Undo?
  - The disk-log remains;
  - We can simply repeat steps above to execute the recovery algorithm;
  - Some <T, A, a_{old} > may have been undone in step 3 before the second crash, but re-Undo them would not hurt.

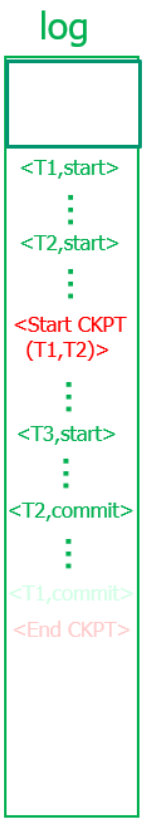
Checkpoint for Undo-logging

- The disk-log file could be extremely large: costing disk space and recovery time;
- We may use checkpoint to shorten disk-log:
  * Place a log record <Start CKPT(T_1, …, T_k)>, where T_1, …, T_k are the currently active transactions;

  \\ at the meanwhile, the system can continuously accept new
  \\ transactions (non-quiescent)

  * When all T_1, …, T_k complete, place a log record <End CKPT>.

  \\ at this point, all log records before <Start CKPT(T_1, …, T_k)>
  \\ can be ignored
  \\ if crash occurs between <Start CKPT(T_1, …, T_k)> and
  \\ <End CKPT>, then trace back to the earliest <T_i, start>
  \\ among the list (T_1, …, T_k) .



Drawbacks of undo-logging:

- <T, commit> reaches disk very late

  All "real works" of T may have been done much earlier. If a crash occurs before <T, commit> reaches disk, all these done works are wasted.

- The memory log may need to be flushed often (in order to satisfy U1)

  This takes disk I/O's.

19 Note 24

19.1 Redo-logging

Format: similar to undo logging, however, each block in the log has their new values associated to it.

Main idea: If the Redo-log record <T, commit> has not reached disk, then **no changes** made by T have reached disk.

Rule:

The entire Redo-log for T (including <T, commit>) reaches disk before any change by T reaches disk

Remarks:

1. Log-record of all results of T reaches disk before T's costly disk I/O's;
2. One flush-log is sufficient for the entire T-log;
3. Drawback: needs main memory space to hold intermediate results

Algorithms for redo-logging:

Log recording

1. Add <T, start> to memory-log;
2. When T writes on A, add <T, A, a_{new}> in memory-log (no output(A) is allowed yet);
3. Add <T, commit> to memory-log;
4. Flush memory-log to disk;
5. Perform output(A) for disk elements A that were written by T.

Recovery with Redo-log

1. Bring disk-log to memory;
2. Scan the log forwards;
3. If <T, commit> is not in log
   Then Ignore T;
   \\but write <T, abort> to disk-log
4. For T with <T, commit> in log
   For each <T, A, a_{new}> in log
   Redo by assigning a_{new} to A;

Checkpoint for Redo-logging

- Place a log record <Start CKPT(T_1, …, T_k)>, where T_1, …, T_k are the currently active transactions;
- Copy all elements in main memory that were written by committed transactions to disk;
  \\ at the meanwhile, the system can continuously
  \\ accept new transactions (non-quiescent)
- When step 2 is done, place a log record <End CKPT>.
  \\ at this point, all log records before the earliest
  \\ <T_i, start> among (T_1, …, T_k) can be ignored
  \\ if crash occurs between <Start CKPT(T_1, …, T_k)>
  \\ and <End CKPT>, then look for a previous
  \\ <End CKPT>

19.2 Undo/redo Logging

Simultaneously record both old and new values on a block in a transaction

Main idea: Increase flexibility and compromise undo logging and redo logging (at the expense of using more space).

Undo: helps guarantee Atomicity, but transactions commit later, more flush of log files

Redo: helps guarantee Durability, but requires more main memory space

Rule:

An undo/redo log record must reach disk before the corresponding change reaches disk. (When the commit arrives doesn't matter anymore)

Algorithms for undo/redo logging:

Undo/Redo log recording

1. Add <T, start> to memory-log;
2. When T writes on A, add
   <T, A, a_{old}, a_{new}> in memory-log;
\\ output(A) can be performed now,
\\ either before or after step 3
3. Add <T, commit> to memory-log.

Recovery with Undo/Redo log

1. Bring disk-log to memory;
2. Scan the log forwards;
3. For T with <T,commit> in log
   For <T, A, a_{old}, a_{new}> in log
   Redo by assigning a_{new} to A;
4. Scan the log backwards;
5. For T without <T,commit> in log
   For <T, A, a_{old}, a_{new}> in log
   Undo by assigning a_{old} to A.

Basically: redo if see a commit in log, undo if don't see a commit.

Checkpoint for Undo/Redo logging

- Place a log record <Start CKPT(T_1, …, T_k)>, where T_1, …, T_k are the currently active transactions;
- Copy all "dirty data" in main memory to disk, this can be due to either committed or incomplete transactions;
  \\ at the meanwhile, the system can continuously
  \\ accept new transactions (non-quiescent)
- When step 2 is done, place a log record <End CKPT>.
  \\ at this point, nothing beyond <Start CKPT(…)>
  \\ needs Redone, but Undo still has to trace back to
  \\ the earliest <T_i, start> among (T_1, …, T_k).
  \\ if crash occurs between <Start CKPT(T_1, …, T_k)>
  \\ and <End CKPT>, then look for a previous
  \\ <End CKPT>

Summary: logging can handle accidental system failures.

20 Note 25

In ACID, isolation and consistency are the other concern that reflect in concurrency control.

Serial schedule: In a schedule of a collection of transactions, for every transaction T, the actions of T are scheduled consecutively without interlacing with actions from other transactions. (i.e. Every step in T_1 executes first, then $T_2, ...T_n$)

Serial schedule guarantees that isolation is met.

if a schedule is "equivalent" to a serial schedule, then it ensures Isolation. (two schedules S_1 and S_2 are "equivalent" if starting with any initial DB state, they always result in the same DB state.)

! Sometimes, a non-serial schedule may acheive the same state as the serial schedule does by plugging in special values. However, this does not mean that they are equivalent. The equivalence should hold true regardless of what values are being used in the transactions.

We want good schedules regardless of: *initial state* and *transaction semantics*.

Only look at the order of reads and writes. (e.g. Schedule C = $r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)$ $w_1(B)r_2(B)w_2(B)$)

Serializability: A schedule is serializable if it is equivalent to a serial schedule.

Conflict-Serializability: A sufficient condition for serializability. That is, if a schedule is conflict-serializable, then it must be serializable. However, serializable schedule need not to be conflict-serializable.

Notes on swapping between two actions:

- IF the swapping occurs within the same transaction: it is NOT allowed. It will violate this transaction's logic.
- IF the swapping occurs between two transactions:
  - IF the swapping is on two different data elements: (say $r_1(A)$ and $w_2(B)$): no restriction on swapping.
  - IF the swapping is on the same data element: (say $r_1(A)$ and $w_2(A)$): swapping only allowed IF the two actions are both **Read**. Any actions that involve **Write** cannot be swapped. (Think about data hazards in computer architecture: RAW, WAW, WAR)
- in short: Two actions **conflict** if either they are by the same transaction, or they are by two transactions on the same element and at least one of the actions is write.

⋆ Swapping two non-conflicting actions gives an equivalent schedule.

⋆ A schedule S1 is **conflict-serializable** if it can be converted into a serial schedule by swapping non-conflicting actions.

Testing Conflict-Serializability: precedence graph
Input: a schedule S

1. Build a directed graph G_S;
2. Each transaction is a node in G_S;
3. If two actions a_1 and a_2 conflict, where a_1 and a_2 are by T_1 and T_2, resp., and a_1 is before a_2 in S, then add an edge from T_1 to T_2 in GS;
4. S is conflict-serializable if and only if G_S contains no cycle.

The algorithm above works because one can always rearrange the graph without loop to make it serializable.

Enforcing Conflict-Serializability: simple locking protocol
Two new actions: lock $l_i(A)$ (transaction T_i exclusively locks the element A), and unlock $u_i(A)$ (transaction T_i releases the lock on the element A)

Rules:
♣ A transaction can lock an unlocked element, and must release (unlock) it later.
♣ A transaction cannot access a locked element until it is released by the transaction holding the lock.

Nonetheless, locking is not enough for conflict-serializability though. (See slides 86 as an example)

Solution: Two phase locking (2PL) (for transactions)
$T_i : ...l_i(A)...l_i(B).....u_i(B)...u_i(A)...$
In a transaction, all lockings precede all unlockings.

2PL enforces conflict-serializability!
Theorem. The execution of a 2PL schedule is equivalent to the serial schedule in which transactions are ordered by the time when they start to unlock.

Proof. Let S be a 2PL schedule in which transaction T_1 starts its unlocks the latest. Let *a* be any action of T_1. Suppose that there is an action *a'* by another transactions T_k that conflicts *a* and appears after *a* in the schedule S. Thus, S = …… *a* …… *a'* ……

Since *a* and *a'* conflict, they are on the same element A. Both T_1 and T_k need to lock A before their corresponding actions *a* and *a'* can be executed. Since *a* appears in S before *a'*, T_1 must lock A before T_k does. So the schedule S looks like S = … l_1(A) … *a* … l_k(A) … *a'*...

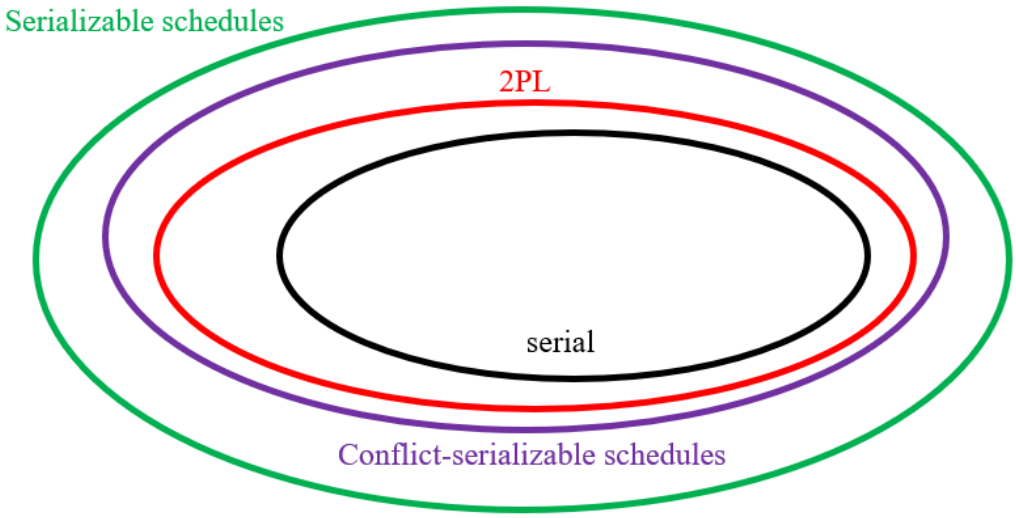
T_k cannot lock A before T_1 unlocks A. So S = … l_1(A) … *a* … u_1(A) … l_k(A) … *a'*... . Since S is 2PL, the first unlock of T_k must appear after l_k(A) , thus after u_1(A). But this contradicts the assumption that T_1 starts its unlock the latest.

Thus, in the schedule S, there is no action by transactions other than T_1 that conflicts an action of T_1 and appears after the conflicting action of T_1. So, we can swap the actions in S to move all actions of T_1 to the tail of the schedule. The new schedule has a form $S_1 = \alpha_1 T_1$.

The new schedule S_1 is equivalent to S.

Using the same argument, we can move all actions of the transaction T_2 that started its unlocks the second latest, to construct a schedule of the form $S_2 = \alpha_2 T_2 T_1$ that is also equivalent to S. Repeating this process, we will eventually get a serial schedule that is equivalent to S. ■

Containment Relations



Serializable schedule: equivalent to a serial schedule	Conflict-serializable schedule: can be swapped into a serial schedule	2PL: transactions locks before any unlock.	Serial schedule: transactions are executed one after the other

21 Note 26

21.1 Different Locking Techniques

Update the simple locking above:
Read-lock (shared-lock $sl_k(A)$) by T_k on A: friendly to reads, but block writes
Write-lock (exclusive-lock $xl_k(A)$) by T_k on A: block any other actions.
Compatibility matrix for lock relations:

	Lock requested		
Lock held		shared	exclusive
	shared	YES	NO
	exclusive	NO	NO

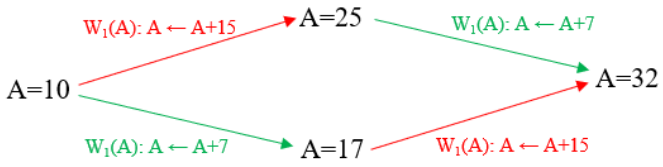
To handle the situation where a transaction would like to read and write on an element: it can either 1) request an exclusive lock that allows both read and write, or 2) exclusively lock an element shared-locked by itself.

Assumption: a single unlock works for both the shared lock and the exclusive lock.

Theorem. 2PL still works for shared/exclusive locks.

Lock Types Beyond Shared/Exclusive Locks

1. Increment lock (further encouragement of concurrency):
   Allowing sharing simple write locks (for writes that increase an element by a constant): $w_i(A)$: A ← A+20):



	shared	exclusive	increment
shared	YES	NO	NO
exclusive	NO	NO	NO
increment	NO	NO	YES

2. Update lock (reducing deadlocks):
   Shared locks are probably too flexible that may cause deadlocks. Update locks make it a bit more difficult.

T_1:	T_2:
$ul_1(A)$;	
$xl_1(A)$;	$ul_2(A)$;
	$sl_2(A)$;

Will be blocked so a deadlock is avoided.

	shared	exclusive	update
shared	YES	NO	YES
exclusive	NO	NO	NO
update	NO	NO	NO

A transaction can obtain an exclusive lock *xl* on A only if it already holds an update lock *ul* on A

Cannot completely avoid deadlocks

How does locking work?

- Scheduler inserts lock actions for transactions
- Lock table

Definition of deadlocks: A deadlock is a situation in which there is a collection $\mathscr{C}$ of transactions such that each transaction in $\mathscr{C}$ is waiting for a resource held by another transaction in $\mathscr{C}$ and no transaction in $\mathscr{C}$ can progress.

System wait-for Graph G. Each transaction is a vertex in G. There is an edge from T_i to T_k if T_i is waiting for a resource held by T_k.

Theorem. There is a deadlock if and only if the system wait-for graph contains a cycle.

Deadlock detection: Using the system wait-for graph.

Deadlock resolving: Rollback a transaction in a cycle in the system wait-for graph (which one?)

Deadlock prevention:

- Using the system wait-for graph.
  - Dynamically updating the system wait-for graph G;
  - When a transaction T is about to introduce a cycle, rollback T.
  - (dynamically manipulating the system wait-for graph can be expensive.)
- Using timeout.
  - Put a time limit for transaction waiting time, when the waiting time exceeds the limit, rollback the transaction.
  - (It may cause many unnecessary rollbacks.)
- Using a resource ordering.
  - Assign database elements a (fixed) order;
  - A transaction T can only lock an element that is larger than the elements it holds. (e.g. T has requested 1, 3, 5, 7. T cannot get 7 until 1, 3, 5 have been collected.)
  - Proof of this preventing deadlocks: If there is a deadlock cycle $T_1, T_2, , T_n$, where each $T_{i-1}, 2 \leqslant i \leqslant n$, is waiting for a resource A_i held by T_i, and T_n is waiting for a resource A_1 held by T_1, then $A_1 < A_2 < ... < A_n < A_1$: impossible.
  - (Problem: Ordered lock requests are not realistic in most cases)
- Using a deadlock timestamp
  - Assign each transaction T a (deadlock) timestamp ts(T);
  - Earlier transactions (i.e., with smaller timestamp) win.
    - Wait-die: Suppose that transaction T_1 needs to wait for a lock held transaction T_2:
      IF T_1 is earlier than T_2 (i.e., ts(T_1) < ts(T_2))
      THEN T_1 waits ELSE rollback T_1.
    - Wound-wait: Suppose that transaction T_1 needs to wait for a lock held transaction T_2:
      IF T_1 is earlier than T_2 (i.e., ts(T_1) < ts(T_2))
      THEN rollback T_2 ELSE rollback T_1 wait.
  - There are no deadlocks since either the earlier transactions need to wait for the later transactions (wait-die), or vice versa (wound-wait)
  - There is no starvation because transaction with smallest ts never dies, and a transaction that dies will eventually have smallest ts and will complete.