Disks, Files, Buffers

Disks

Disk Anatomy

- Disk is composed of stacked platters spinned by a central spindle.
- Arm assembly is moved in or out to position a **head** on a desired **track**
  - Tracks under heads make a "cylinder"
- Block / page size is a multiple of (fixed) **sector** size.

Disk Access

1. Seek time (moving arms to position disk head on track): ~2-3 ms
2. Rotational delay (waiting for block to rotate under head) : ~0-4 ms
3. Transfer time (moving data to/from disk surface) : ~0.25 ms / 64KB page

SSD / Flash memory

- Issues in current generation (NAND)
  - Fine grain reads (4-8K reads), coarse-grain writes (1-2MB writes)
  - Only 2k-3k erasures before failure, so we need to keep moving hot write units around to spread storage wear level.
  - Write amplification: big units, need to reorg for wear & garbage collection
- Performance:
  - Read is fast and predictable. Not much difference between random and seq. reads.
  - Write is slower for random writes (about 4x slower compared to seq writes)
- Expect 10-100x bandwidth for non-sequential read compared to magnetic disk.

Blocks / Pages

Block Level Storage: Interface for read and write **large chunks of sequential bytes**

Block / Page: Unit of transfer for disk read / write , usually 64-128KB these days

Several techniques to maximize usage of data per R/W:

1. Cache popular blocks
2. Pre-fetch several blocks at a time / large consecutive blocks.
3. Buffer writes to sequential blocks. Arrange file pages by 'next' on disk to minimize seek and rotational delay.

'Next' block concept on a disk:

- sequential blocks on same track, followed by
- blocks on same cylinder, followed by
- blocks on adjacent cylinder

Disk Space Management (lowest layer)

Purpose: **Map** pages → locations on disk, **load** pages from disk → memory, **save** pages back to disk. Ensure that sequential read/write is fast.

Implementation

Common: Run over filesystem (FS)

- Allocate single large "contiguous" file on a nice empty disk, and assume sequential/nearby byte access are faset
- Most FS optimize disk layout for sequential access
- Problem: DMBS "file" may span multiple FS files on multiple disks/machines. So, Disk Space Management provides "file" abstraction over details of FS files and devices.

Files as Pages of Records

Tables are stored in logical **DB files**: a collection of pages, each containing a collection of records.

Pages are managed on disk by disk space manager, and in main memory by the buffer manager. Each DB file could span multiple OS files and even machines.

API for higher layers of DBMS:

- Insert/delete/modify record
- Fetch a particular record by **record id**: a pointer encoding pair of (**pageID, location** on page)
- Scan all records

Unordered Heap Files

How do we find a page that can fit a record efficiently? Use a page directory structure.

- The directory is a collection (say, linked list) of header pages. The head of the page directory is stored in the database catalogue.
- The directory entries include a pointer to the data page and the number of free bytes on the referenced page.

Pages

Header may contain:

- number of records
- how much free space is on the page
- pointer to next page
- bitmaps / slot tables to find free space on the page

Pages with Fixed-Length Record

1. Packed : (-) need to repack on deletion and update all record id pointers
2. Unpacked: Use number of slots field + a bitmap to denote "slots" status (valid / deleted records). Record id: <page id, slot number>. Insertion: Find first free slot. Deletion: Toggle bitmap "slot" status.

Pages with Variable-Length Record

Changes from pages with fixed-length record

- Metadata is put at the footer.
- Introduce slot directory in footer
    - number of slots in slot directory (to keep track if we need to add more slots)
    - pointer to start of free space
    - each entry stores length + pointer to beginning of record (arranged in reverse order)

Implementation Details:

- Record ID = location in slot table (from right)
- Deletion: Set slot directory pointer to null
- Insertion:
    1. Place record in free space on page
    2. Create pointer/length pair in next open slot in slot directory
    3. Update free space pointer.
    4. (lazily) reorganize data on page and update all slot directory pointers.

Record Formats

Assume the schema and field types is stored in **System Catalog**. The goal is to have fast access to fields and compact records both in memory and in disk format.

Variable Length Fields

Idea: Move all variable length fields to end to enable fast access
Implementation: Introduce a record header with pointers to variable length fields.

Buffer Management (2nd from bottom)

Files and Index Management expects data to be in RAM.
Buffer Management is very, very similar to cache.

- Buffer Manager manages a **buffer pool**, large range of RAM allocated for DBMS on server boot time.
- Buffer Pool is partitioned into page-sized partitions called **frames**.
- Metadata: Table of <frameid, pageid, dirty?, pin count> pairs are maintained.
- Pin count represent the number of queries that is using the page. +1 on every request, and caller must immediately unpin page on query completion.

Page Replacement Policy

- Least Recently Used (LRU), usually approximated by Clock policy.
- Clock:
  - Clock hand points to next page being considered.
  - Additionally, the metadata table maintain reference bit information (which is 1 for recently referenced pages)
  - When a page is requested, set pin and set reference bit to 1
  - When figuring out which page to throw out, move the clock hand around. If the page is pinned, skip. If the page has ref bit, clear ref bit. Otherwise, we have found the page to kick.
- Most Recently Used (MRU): outperform LRU/Clock on sequential scans.

Additional Topics

1. Improvements for sequential scan: Prefetch
   - Amortize random I/O overhead
   - Allow computation while I/O continues in the background
2. Hybrid page replacement policy: LRU for random access, MRU for certain joins. Two general approaches:
   1. Use DBMS information to hint to BufMgr:
      - For big queries: predict I/O patterns from query processing algos
      - For simple lookups: Use LRU.
   2. Fancier stochastic policies: 2Q, LRU-2, ARC
3. DBMS vs OS Buffer Cache
   Issues:
   1. Portability: different FS, different behaviour
   2. OS limitations: DBMS requires ability to force pages to disk (required for recovery)
   3. OS limitations: DBMS can predict its own page reference patterns, affects both page replacement and prefetching.

Indexes

An index is data structure that enables fast **lookup** (equality, 1-d range, 2-d region search) and **modification** of **data entries** (items stored in the index : pair of keys and heap file pointers) by **search key** (any subset of cols in the relation). If search key is a candidate key, the index is called **unique**:

Things to consider when choosing an index:

- search performance
- storage overhead
- update performance

Tree Based Indexes

Based on **sorting of search key**.

ISAM (Indexed Sequential Access Method)

- Data entries in **sorted** (by index key) heap file
- High fan-out **static** tree index → only leaf pages modified
- Fast search + good locality (things that are sorted tgt are stored tgt)
- Insert into overflow pages → linked list of pages (not good!, lots of insertion degrades performance into linear search)

B+ Tree

[B+ Tree Visualization (usfca.edu)]

- Similar to ISAM: same interior node structure, same search routine.
- Leaf nodes stored in **sorted** data entries.
- Each data record has an entry in the leaf node **(dense index)**
- **Dynamic** Tree Index : always balanced, support efficient insertion & deletion (grows at root not leaves)
- B+ tree differs from B-tree because it stores data entries in leaves only (and this enables fast range search)

Property:

- Height-balanced **(search and update efficient)**
- Each interior node (except root) is at least partially full → **(storage efficient)**
  - d <= #records <= 2d
  - d: order of the tree (max fan-out = 2d + 1)
  - Typical B+ tree have order 1600, with 67% fill rate. Height 2 can store 10GB of data.
- Data pages at bottom need not be sequential pages. Next leaf pointer to chain up the leaf nodes **(efficient range search)**

Insert

Algorithm:

1. Find the correct leaf L
2. Put data entry onto L
    - If L has enough space, done!
    - Else, must split L (into L and a new node L2)
        - Redistribute entries evenly, **copy up** middle key
        - Insert index entry pointing to L2 into parent of L
3. Step 2 can happen recursively (index node can be full)
    - To split index node, redistribute entries evenly, but **push up** middle key.

Deletion

- In practice, occupancy invariant often not enforced
- Just delete leaf entries and leave space for future inserts.
- Space are reclaimed only when a page / node is completely empty.

Bulk Loading

Inserting one by one (esp if data is not sorted) causes poor cache efficiency as we are modifying random pages.

Instead, we can:

1. Sort the input records by key
2. Fill leaf pages to some fill factor, while updating parent pages.
3. If the parent page is full, we split the parent node into 2 nodes with d entries each and moving up the middle element.

Characteristics of an index

Query support

- Basic Selection → equality, range selection
    - B+ Trees support both equality & range
    - Linear hash indexes provide only equality
- Exotic selection → 2-d box, 2-d circle, k-nearest-neighbor queries, regex match, etc.

Search Key and Ordering

- We can index on any ordered subset of columns, but the order determines the queries supported!
- Suppose a composite search key is defined on columns $(c_1, c_2, \dots c_n)$. A query is **compatible** with the index only if the query is a *conjunction* (AND) of $m \geq 0$ equality clauses on $(c_1, c_2, \dots c_m)$ and at most 1 additional range clause on c_{m+1}.

- Reason: The index is looked-up and scan in lexicographic order. First, find start-of-range, then do a scan of **contiguous** data entries until the condition fails.

Data entry storage

Basic alternatives for data entries in any index:

1. by value
2. by reference (< key, matching record id >)
3. by list of references (< key, list of matching record ids >).

Note: By-reference indexing (alternative 2 or 3) is needed to support multiple index per table.

Clustered index

Index in which heap file records are kept mostly ordered according to search keys in index (order need not be perfect).

Pros	Cons
Efficient for range searches	More expensive to maintain: need to periodically update heap file order either on the fly or "lazy" reorgs
Potential locality benefits: sequential disk access, prefetching, etc	Heap file usually only packed to 2/3 to accomodate inserts
Support certain types of compression	

Variable-length key tricks

How do we handle variable length keys like strings?

Observation: Order (d) makes little sense with variable length entries

- Different nodes can hold different number of entries
- Index pages often hold many more entries than leaf pages

We use a physical criterion in practice : at-least half-full (in bytes)

How can we get more keys on a page?

1. Prefix Key Compression
    - might result in a slightly different layout compared to if we copy the whole key, but it's okay

2. Suffix Key Compression : Move common prefix to header and leave only (prefix-compressed) suffix next to the pointers.
    - This is especially useful in composite keys!

Hash-based Indexes

- Key k, hash function h, h(k) returns the pageID that stores record with *search key* k.
- Best for equality selections, inefficient for range searches (depends on hash function used).
- Performance degenerate for skewed data distributions
- Buckets consists of 1 primary data page, and 0+ chain of overflow pages.
- Each data entry may contain **the records directly** or pointers to data records.

Static Hashing

- Data stored in M (a fixed number) buckets

Dynamic Hashing

Linear Hashing

```
0 1 ... next - 1 | next ... N_i - 1 | N_i ... N_i + next - 1
uses h_(i + 1)       uses h_i            uses h_(i + 1)
```

- Hash file grows linearly (**1 bucket at a time**, i.e. B_i should be split before B_j if $i < j$).
    - At the end of each **splitting round**, the size of hash file would have doubled, and the hash function should be changed.
    - Instead of doubling the number of buckets at once, splitting only 1 bucket at a time minimize the number of record redistributions that is needed, thus the operation that triggers this splitting is minimally affected.
- Use the last $\lceil \log N(i) \rceil + 1$ bits of h(k) to split records between B_j and $B_{N(i)+j}$
- Maintain a "next" pointer to indicate which bucket to split next.
- When to split a bucket? Split whenever **some** bucket overflows
    - This will increment the "next" pointer (or increment level and set "next" ← 0)
    - It might be the case that the current bucket we split is not overflowing.
- When to delete a bucket? Only when the last bucket is empty.
    - Decrement "next" (or decrement level and set "next" ← last bucket in prev level)

On average for uniformly distributed data, 1.2 disk I/O is needed.

Extendible Hashing

-

Sorting and Hashing

Sorting is used in many places:

1. `DISTINCT`, `GROUPBY` (rendezvous match: similar things are grouped together), `ORDERBY`
2. First step in bulk-loading tree indexes

Problem: It is difficult to sort 100GB of data with 1GB of RAM
Solution: Out-of-core algorithm (out of RAM)

1. Single-pass streaming data thru RAM
2. Divide (into RAM-sized chunks) and Conquer

Single-pass streaming data

```
INPUT --- Input Buffers --- f(x) --- Output Buffers --- OUTPUT
```

1. Read a chunk from INPUT to an Input Buffer
2. Write f(x) for each item into an Output Buffer
3. When input buffer is consumed, read another chunk
4. When output buffer fills, write to Output

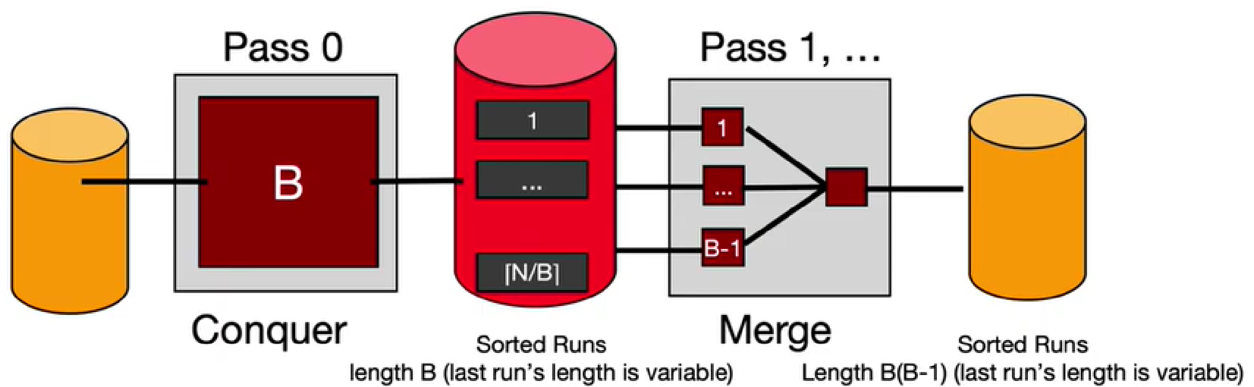
Double Buffering Optimization

Thread 1 runs f(x) on 1 pair of I/O bufs, while Thread 2 drains/fills unused I/O bufs in parallel. When Thread 1 finish, swap the two buffers

```
Thread 1: INPUT --- Input_Buf_1              Output_Buf_1 --- OUTPUT
Thread 2:           Input_Buf_2 --- f(x) --- Output_Buf_2
```

This can "hide" the latency of I/O behind CPU work.

External Sort



Sort a file with N pages using B buffer pages.

1. Pass 0: (Internal) sort each page in RAM. Produce $\lceil N/B \rceil$ sorted runs of B pages each.
2. Pass 1, ... : merge $B-1$ runs at a time (one buffer is used as the output buffer) using streaming

Number of passes: $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$
Cost: $2N \times$ number of passes. With big values of B, this cost is easily linear time.

Memory requirement : $\sim \sqrt{N}$ memory to sort N pages.

External Hashing

Sometimes, we don't require order, we just need to rendezvous matches (e.g. removing duplicates, forming groups)

1. Divide (Streaming partition)
    - Use a hash fn h_p to stream records to disk partitions. Use 1 input buffers, and $B-1$ output buffers.
    - All matches ended up in the same partition. Each partition contains a mix of values.
    - We ended up with $B-1$ hash partitions of size $\sim \frac{N}{B-1}$.
    - If a partition ended up having size $> B$, recursively "External Hash" that partition using another hash fn h_{p1} for the divide step.
2. Conquer (Rehash)
    - Motivation: same values from different pages ended up in 1 partition after partitioning, but not contiguous.
    - Read partitions into RAM hash table one at a time, using hash fn h_r
    - Then, read out the RAM hash table buckets and write to disk, ensuring that duplicate values are contiguous.

Note: For very frequent / duplicate key (e.g. gender column) → we ended up with the same thing even after a recursive partition. We need another way to recognise this

and just put it back into disk.

Cost: $2N\times$ number of passes. 4N I/O's (if we assume $N < B^2$).
Memory requirement: $\sim \sqrt{N}$ memory to sort N pages assuming hash function distributes records evenly.

Sorting & Hashing pros

Sorting and hashing **duality**:

- Sorting: Conquer & Merge
- Hashing: Divide & Conquer

Sorting pros:

1. Great if we need output to be sorted
2. Not sensitive to duplicates or "bad" hash functions

Hashing pros:

1. Easy to shuffle equally in parallel case
2. For duplicate elimination, hashing scales with number of **distinct values** (not number of items when using sorting)

External Sorting Optimization Consideration

On internal sort algorithm

- We can use replacement selection instead to further reduce the number of passes (and generate longer runs than memory size)
- In practice the run length of replacement selection is generally $2B$.

Replacement Selection

- Read B blocks into memory. For each iteration, move the smallest record to output buffer then read in a new record. If the new record < last record in output buffer, freeze block. Keep looping until all blocks are freezed.

Is minimizing pass always optimal?

Not always. For example in the merge step, assuming we have B input buffer pages and a run fits in a track, we have 2 alternatives.

1. Merging B runs of N pages at once will incur a cost of a seek every time we read a page from a run. Total number of seek: BN
2. We will need 2 steps. If in the first step, we were to merge $\frac{B}{K}$ runs of N pages at a time (K times), we can read K pages at once, thus for each run we only need $\frac{N}{K}$ seeks. As we have $\frac{B}{K}$ runs and we need to repeat this merging K times, we

ended up with $\frac{BN}{K}$ seeks for the first step. In the second step, we merge K runs of $\frac{BN}{K}$ pages, we can read $\frac{B}{K}$ pages at once, thus for each run, we only ended up with N seeks. We ended up with KN seeks for the second step. In total this is only $\frac{BN}{K} + KN$ seeks, smaller than BN.

Using B+-tree for sorting

A clustered B+-tree index is good for sorting, while unclustered tree index in general performs worse than external sort (on reasonably sized buffer page), it can be useful in scenario where initial fast response is needed (e.g. to upstream operator).

Evaluating Operations

Notation:

- $|R|$: the number of pages of relation R
- p_R : tuples per page

Joins

Setting: equality join on 1 column.

Always try to make the outer loop smaller in size then the inner loop. :D

Simple (Tuple-based) Nested Loop Join

```
for tuple r in R do
    for tuple s in S do
        if r.sid == s.sid then add <r,s> to result
```

Cost: $|R| + (|R| \times p_R) \times |S|$

Block Nested Loop Join

```
for page P_R of R do
    for page P_S of S do
        for tuple r of P_R do
            for tuple s of P_S do
                if r.sid == s.sid then add <r,s> to result
```

Cost: $|R| + |R| \times |S|$
Memory: 3 pages.

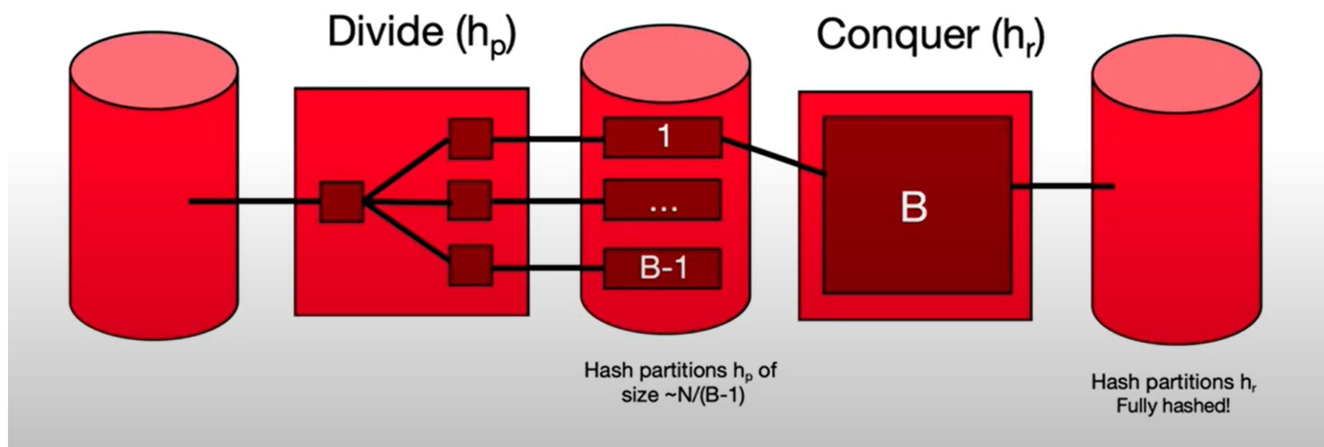
We can improve on this if we have more than 3 pages.
Cost: $|R| + \left\lceil \frac{|R|}{B-2} \right\rceil \times |S|$
Memory: B pages

Sort-Merge Join

1. Sort R and S on the join column → $2|R| \times (1 + \lceil \log_{B-1} \lceil |R|/B \rceil \rceil)$ each
2. Scan them to do a "merge" on join column and output result tuples. → $(|R| + |S|)$

Cost: Sort cost + Merge cost

Grace Hash Join



1. Partition phase: number of partition passes $\times 2 \times (|R| + |S|)$.
    1. Partition relation R (on join attribute) using hash fn h. ← at most $B-1$ partition.
    2. Partition relation S using the same hash fn h
    3. R tuples in partition i will only match S tuples in partition i
2. Join phase: $|R| + |S|$
    1. Read in a partition of R → partition should fit in $\leq B-2$ pages.
    2. Build a hash table for the partition using hash fn h2
    3. Scan matching partition of S, search for matches (using hash fn h2)

Need to check whether $\min(|R|, |S|) \leq (B-1)(B-2)$. If not, apply the hash-join partition step recursively.

Cost: partition cost + join cost.

Index Nested Loops Join

Precondition: There is an index on the join column of one relation (S) → put it as the inner relation to exploit the index.

```
for tuple r in R do
    search index of S on sid using S_{search-key} = r.sid
    for each matching key
        retrieve s
        add (r,s) to result
```

Cost: $|R| + (|R| \times p_R) \times \text{tuple search cost}$

Tuple search cost:

1. B+-tree:
    - Clustered index: tree height + 1 I/O (typical)
    - Unclustered: tree height + upto 1 I/O per matching S tuple
2. Hash index: 1.2 + cost of finding S tuples (1 I/O per matching tuple)

General Join

Multiple equality conditions

- SMJ, HJ → sort / partition on combination of the join columns
- Index NJ → use existing index on one of the columns, or build a new index on both columns

Inequality conditions

- SMJ, NJ → fine
- HJ → not suitable

General Selections

Terminology

- **Selectivity** = size of result / $(|R| \times p_R)$. Selectivity of access path: num of pages accessed. Most selective → lowest selectivity
- **Access path**: ways of accessing data records/entries
    - table scan
    - index-only scan
    - index-search → might be followed by data records lookup
    - index intersection → combine result of multiple index traversal
- An index I is a **covering index** of a query Q if Q can be performed using index-only scan on I, i.e. all attributes referenced in Q is in I.
- Index I **matches** predicate p if p have equality conditions on a prefix of I (order matters, no attribute skipped), and at most 1 range condition for a B+-tree index.

Strategy

1. Find most selective access path, retrieve tuples using it, then apply remaining conditions that don't match the index
2. If we have more than 1 matching index. Get set of rids using both matching index, intersect, retrieve records, then apply remaining conditions.

Projection (DISTINCT)

Sort-based approach

Optimized step:

1. Create sorted runs with attribute L
2. Merge sorted runs while removing duplicates

$|\pi_L^*(R)| = |R|$ x size of L / original size of tuple
Unoptimized cost: $|R| + |\pi_L^*(R)|$ + cost of sorting $|\pi_L^*(R)|$ pages + $|\pi_L^*(R)|$
Optimized cost: $|R| + |\pi_L^*(R)|$ (first phase) + $|\pi_L^*(R)| \times \lceil \log_{B-1} \lceil |\pi_L^*(R)|/B \rceil \rceil$ (second phase)

Hash-based approach

1. Partitioning phase: Project out unwanted attribute: t → t', then apply hash function similar to hash join
2. Joining phase: Read in 1 partition. For each tuple t in partition, insert t into hash table only if it doesn't exist yet. Write out tuples in hash table.

To avoid partition overflow, we need $\frac{|\pi_L^*(R)| \times f}{B-1} < B$, where f is a fudge factor for the hash table.

Cost: $|R| + |\pi_L^*(R)|$ (partitioning) + $|\pi_L^*(R)|$ (duplicate elimination phase)

Comment: Both hash-based and sort-based has same cost if $B > \sqrt{|\pi_L^*(R)| \times f}$

Using index

- If the index covers the projection, we can replace table scan with index scan
- If the index is ordered (e.g. B+-tree index), we can scan data entries, and remove adjacent duplicates.

Set Operations

- Intersection and Cross Product are special cases of join
- Union (Distinct) and Difference are similar
  - Sort based approach similar to external sort.
  - Hash based approach similar to external hashing / hash join.

Aggregates

Without grouping

Generally involves table scan. If there is a covering index, we can replace with index-only scan.

With Grouping

- Sort/Hash on groupby-attributes, then compute aggregates for each group
- If there is a **covering tree** index, we can do an index-only scan.
    - If group-by attributes form prefix of index search key, we can retrieve data entries/records in group-by order.

Parallel Query Execution

- Metric: We want to keep the throughput by increasing hardware as we increase our workload.
- Parallel architectures: 1. Shared Memory, 2. Shared Disk, 3. Shared-Nothing

Kinds of parallelism

1. Inter Query: Each query runs on one thread (no parallelism within query), require parallel-aware concurrency control
2. Intra Query: Within a query
    1. Inter Operator: pipeline parallelism, bushy tree parallelism (parallelize execution of different branches of a tree)
    2. Intra Operator: partition parallelism

Partitioning (partition table across disks / machines)

1. range partitions: good for equijoins, range queries, group by
2. hash partitions: good for equijoins, group by
3. round robin: good for spreading load → might need broadcast insert / lookup

Parallel sorting and **Parallel Sort Merge Join**:

1. Shuffle data across machines as it is streaming out of the disk: split on range of value (taking care to ensure the same number of pages for each machine)
2. Do local sorting on each machine.

Parallel hashing and **Parallel Grace Hash Join**:

1. Shuffle data across machines as it is streaming out of the disk: Use another hash fn h_n. This will make same values ended up in the same machine.
2. As values stream in, each machine starts doing partitioning phase (phase 1) of external hashing.
3. Wait until all machines finish phase 1, then do rehashing / joining phase (phase 2) on each machine.

Comment: near-perfect speed-up, scale-up! Only waiting is for phase 1 to end.

Parallel Aggregation:

For each aggregate function, we need to compute the local aggregate, and then combine these aggregates into a global aggregate.

Parallel Groupby:

1. Local aggregation: in hash table keyed by group key k_i, keep local aggregates for each key.
2. Shuffle local aggregates to receiver nodes using a hash function $h_p(k_i)$
3. Compute global aggregates for each key k_i.

Symmetric (Pipelined) Hash Join:

Problem: Grace Hash Join can't start probing until the hashtable is built

Solution:

1. Each node allocates two hash tables, one for each side.
2. Upon arrival of tuple of R: insert tuple into R hashtable, and probe S hashtable for any matches, and output any that are found.
3. Upon arrival of tuple of S: symmetric to R.

Why does it works? Each output tuple is generated exactly once, i.e. when the second part arrives.

Comment:

- This join is a single phase, streaming algorithm, and hence is very responsive.
- We can easily extend this to its parallelized version
- Out of core symmetric hash join: X-join.
- Non-blocking SMJ: Progressive Merge Join.

Other join patterns

So far we have discussed symmetric shuffle patterns. Others:

1. Asymmetric (One-sided) shuffle: When one of the table has been partitioned, we only need to shuffle the other table.
2. Broadcast join: When one of the table is quite small, say a few hundred rows, we can replicate (or broadcast) this table to every other machine containing the bigger table and let them do joins locally.

Query Optimizer

Intro

When a query is passed in,

1. Query Parser: checks correctness, authorization; generates a parse tree
2. Query Rewriter: converts queries to canonical form, e.g. flatten views, subqueries into fewer query blocks

3. "Cost-based" Query Optimizer: Optimizes 1 query block at a time, and uses catalog stats to find least-"cost" plan per query block
4. Passed the final query plan to the query executor.

Orthogonal concerns in query optimization:

1. plan space: Based on relational equivalences and different implementation choice
2. cost estimation: based on cost formulas and size estimation (based on catalog information and selectivity)
3. search strategy: how do we "search" in the "plan space" to find the lowest cost option

(Realistic) Goal: find the plan with least estimated cost (and try to avoid really bad actual plans)

Plan Space

Algebra (Logical) Equivalences

Relational Algebra Equivalences

What about $\sigma_{p1 \vee p2 \ldots \vee pn}(R)$?

- **Cascading of selections**: $\sigma_{p_1 \wedge p_2 \wedge \cdots \wedge p_n}(R) \equiv \sigma_{p_1}(\sigma_{p_2}(\cdots(\sigma_{p_n}(R))\cdots))$
- **Commutativity of selections**: $\sigma_{p_1}(\sigma_{p_2}(R)) \equiv \sigma_{p_2}(\sigma_{p_1}(R))$
- **Cascading of projections**: $\pi_{L_1}(R) \equiv \pi_{L_1}(\pi_{L_2}(\cdots(\pi_{L_n}(R))\cdots))$, where $L_i \subseteq L_{i+1}$ for $i \in [1, n)$
- **Commutativity of cross-products**: $R \times S \equiv S \times R$
- **Associativity of cross-products**: $R \times (S \times T) \equiv (R \times S) \times T$
- **Commutativity of joins**: $R \bowtie S \equiv S \bowtie R$
- **Associativity of joins**: $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$
- **Others**: $R \cup S = S \cup R$, $R \cap S = S \cap R$, $R \cup (S \cup T) = (R \cup S) \cup T$, $R \cap (S \cap T) = (R \cap S) \cap T$, etc.

Relational Algebra Equivalences

- $\pi_L(\sigma_p(R)) \equiv \sigma_p(\pi_L(R))$ if σ involves only attributes retained by π
- $R \bowtie_p S \equiv \sigma_p(R \times S)$
- $\sigma_p(R \times S) \equiv \sigma_p(R) \times S$ if σ refers to attributes only in R but not in S
- $\sigma_p(R \bowtie S) \equiv \sigma_p(R) \bowtie S$ if σ refers to attributes only in R but not in S
- $\pi_L(R \times S) \equiv \pi_{L_1}(R) \times \pi_{L_2}(S)$ if $L_1 = L \cap attr(R)$ & $L_2 = L \cap attr(S)$
- $\pi_L(R \bowtie_p S) \equiv \pi_{L_1}(R) \bowtie_p \pi_{L_2}(S)$ if $L_1 = L \cap attr(R)$, $L_2 = L \cap attr(S)$, & every attribute in p also appears in L
- Others: $\sigma_p(R \cup S) = \sigma_p(S) \cup \sigma_p(R)$, etc.

Physical Equivalences

What algorithms can we swap in

- Base table access with single table selections and projections: heap scan, index scan (if available on ref columns)
- Equijoins: BLNJ, Indexed NLJ (often good if 1 is relatively small, and the other is indexed properly), SMJ (good with small memory, equal-size tables), Grace Hash Join (better than sort if 1 table is small).

Heuristics

1. Selection cascade and pushdown: apply selections as soon as you have the relevant columns
2. Projection cascade and pushdown: keep only the columns you need to evaluate downstream

3. Avoid cartesian products (not always optimal, e.g. when you have 2 very small tables)

Note: Pushing a selection into the inner loop (right branch) of a nested loop doesn't save IOs, because it still needs to scan the entire right table on every iteration. That's where materialization might come in handy.

Summary

For a SQL query, full plan space include all equivalent relational algebra expressions (join orders, operation orders), and all mixes of physical implementations of those algebra expressions (join algs).

We might prune this space by:

1. Applying heuristics (selection / projection pushdown, avoiding cartesian products)
2. Considering left-deep trees only (a System R heuristics)
3. Taking into account about physical properties (how the data is grouped), because downstream ops may depend on them, and enforcing them later might be expensive. E.g. SMJ groups data based on the group key, BLNJ preserves outer table ordering, hashing groups similar data together, etc.

Cost Estimation

For each plan considered, we must estimate total cost

- estimate **cost** of each operation in plan tree, which depends on input cardinalities
- estimate **size of result** of each operation in tree
  - because it determines downstream input cardinalities
  - We use information about the input relations
  - Typical assumptions: **uniform distribution** of data, **independence** of predicates, inclusion assumptions for equijoins.

Statistics and Catalogs

Statistics and Catalogs provide information on relations and indexes involved.

Typically it contains at least:

1. NTuples or $||R||$: Number of tuples in a table (cardinality)
2. NPages or $|R|$: Number of disk pages in a table
3. Low / High: Min / Max value in a column
4. Nkeys: Number of distinct values in a column.
5. IHeight: Height of an index
6. INPages: Number of disk pages in an index.

Catalogs are updated periodically, and modern systems usually keep more detailed statistical information on data values, e.g. histograms.

Size estimates

Result cardinality = Max number of tuples x product of all selectivity

Assumption 1: Uniformly distributed over all distinct values in a column.

1. col = value → sel = 1 / NKeys(I)
2. col1 = col2 (handy for joins too) → sel = 1 / Max(NKeys(I1), NKeys(I2))
3. col > value → sel = (High(I) - value) / (High(I) - Low(I) + 1)

Assumption 2: Independent Predicates

- Selectivity of AND = product of selectivities of predicates
- Selectivity of OR = sum of selectivities of predicates - product of selectivities of predicates
- Selectivity of NOT = 1 - selectivity of predicates

Assumption 3: Preservation of value sets

For joins $U = R1(A, B) \bowtie R2(A, C)$, then

- NKeys(U, A) = min { NKeys(R1, A), NKeys(R2, A) }
- NKeys(U, B) = min { || U || , NKeys(R1, B) }.

Histograms

Types:

- Equidepth: each bucket has approximately same number of records
- Equiwidth: each bucket has (almost) equal number of values (same range)

Assumption: Within each bucket, records are uniformly distributed across the range of the bucket

Comment: In reality, most modern DB only maintains histogram for the top 10% of the distinct values, and treat the other buckets as uniformly distributed.

Search Strategy

Exhaustive search

Calculate cost for every plan

Greedy Algorithms

Based on heuristic, e.g. smallest relation next, smallest result next

Randomized techniques

Randomly move to neighboring states until it reaches either a local minimum / global minimum using local optimization.

Moves are usually swaps between two relation, or cycles of three relations.

Comparison between exhaustive, greedy, and randomized algorithms:

- Search space: Exhaustive (largest, i.e. all), Randomized (has the potential to be large if rules is reasonable), Greedy (smallest)
- Plan Quality: Exhaustive (best), Randomized (probably second, because searching a larger space)
- Optimization overhead: Greedy (fastest), Randomized (allows end user to determine how much optimization by controlling how much time the optimizer get to run or controlling how many plans are compared before deciding that the plan is a local minimum based on the number of joins in query, e.g. more joins, more comparison)

Dynamic Programming (System R)

The algorithm proceeds by considering increasingly larger subset of the set of all relations.

- Builds a plan bottom-up (beginning from 1 table, then to 2, and so on)
- Plans for a set of cardinality i are constructed as extensions of the **best** plan for a set of cardinality $i-1$.

DP may maintain multiple plans per subset of relations to capture interesting orders of physical data, e.g. using SMJ ensures that the data is has been sorted, NLJ preserves orders, hashing groups data, etc.

Time complexity: For left deep trees → $2^k - 1$ entries, for bushy trees → $O(3^k)$.

Concurrency Control

Why?

- Increase throughput by increasing processor/disk utilization, e.g. can use CPU while another transaction is writing to disk.
- Increase latency since multiple transactions can run at the same time, so one transaction's latency need not be dependent on another unrelated transaction.

Anomalies

1. Dirty read: (WR conflicts) T2 reads an object that has been modified by T1 (which might abort later)

2. Unrepeatable read: (RW conflicts) T2 updates an object that T1 has just read while T1 is still in progress. T1 could get a different value if it reads the object again.
3. Lost update: (WW conflicts) T2 overwrites the value of an object that has been modified by T1 while T1 is still in progress. As a result, T1's update is lost.

Transactions

- **Transaction** is a sequence of multiple **actions** (reads and writes of database objects) to be executed as an **atomic** unit (batch of work must commit or abort altogether).
- **Transaction manager** controls execution of transactions.

ACID Properties of transaction

1. Atomicity: Either execute all its actions or none of them.
2. Consistency: If the DB starts out consistent, it ends up consistent. Consistent = follows declarative integrity constraints, e.g. in `CREATE TABLE` statements.
3. Isolation: Execution of each transaction is not affected by (isolated from) other transactions
4. Durability: The effects of a commited transaction must survive failures.

Concurrency control provides isolation property, e.g. via 2PL
Recovery provides atomicity and durability properties, e.g. via WAL

Schedules

- A **schedule** is a sequence of actions on data from 1 or more transactions
- A schedule is a **serial** schedule if each transaction runs from start to finish without any intervening actions from other transactions
- 2 schedules are **equivalent** if
    1. involve the same transactions
    2. each individual transaction's actions are ordered the same
    3. both schedules leave the DB in the same final state.
- Schedule S is **serializable** if S is equivalent to *some* serial schedule.

Conflict Serializable Schedules

- Two operations **conflict** if they are by different transactions, acting on the same object, and at least one of them is a write.
- 2 schedules are **conflict equivalent** iff:
    1. They involve the same actions of the same transactions
    2. Every pair of conflicting actions is ordered the same way.
- Schedule S is **conflict serializable** if S is conflict equivalent to *some* serial schedule.

- Equivalently, a schedule S is conflict serializable if you are able to transform S into a serial schedule by swapping *consecutive non-conflicting* operations of different transactions.
- To rigorously prove serializability, we can construct a **dependency graph** with 1 node per transaction, and an edge from Ti to Tj if an earlier operation Oi of Ti conflicts with an operation Oj of Tj.

Theorem. Schedule is conflict serializable iff its dependency graph is acyclic.

Remark: conflict serializable schedules are a subset of all serializable schedules.

View Serializable Schedules

- Schedules S1 and S2 are **view equivalent** if:
    1. Same initial reads: if Ti reads the initial value of A in S1, then Ti also reads the initial value of A in S2.
    2. Same dependent reads: if Ti reads the value of A written by Tj in S1, then Ti also reads the value of Tj in S2.
    3. Same winning writes: If Ti writes final value of A in S1, then Ti also writes final value of A in S2.
- Schedule S is **view serializable** if S is view equivalent to *some* serial schedule.
- View serializable schedules are the set of all conflict serializable schedules, which also allows "blind writes".

Remark:

- conflict serializability is more commonly used because it can be enforced efficiently.
- serial $\subset$ conflict serializable $\subset$ view serializable $\subset$ serializable schedules.

Recoverable Schedules

For correctness, if Ti has read from Tj, then Ti must abort if Tj aborts. This recursive aborting phenomenon is known as **cascading aborts**

- A schedule S is **recoverable** if for each transaction Ti in S, Ti commits after Tj if Ti reads from Tj.
- A schedule S is **cascadeless** if a transaction Ti only reads from a committed transaction Tj.
- A schedule S is **strict** if for every $W_i(O)$ in S, O is not read nor written by another transaction until Ti either aborts or commits.

Remark:

- Recoverable schedules prevent undoing committed transactions, but still suffer from cascading aborts.

- Cascadeless schedules allow writing to an object previously written by another uncommitted transaction.
- serial ⊂ strict ⊂ cascadeless ⊂ recoverable schedules.

Pessimistic CC: Locking-based Protocol

The most common scheme for enforcing conflict serializability is 2 phase locking. However, it is a bit pessimistic compared to alternative schemes, e.g. Optimistic or Timestamp-Ordered or Multiversion Concurrency Control.

2 Phase Locking (2PL)

Rules

1. Well formed: Transaction must obtain a S (shared) lock before reading and an X (exclusive) lock before writing
2. Legal: Scheduler respects the lock compatibility matrix
3. Two-phase: Transaction cannot get new locks after releasing any lock.

Lock compatibility matrix

	S	X
S	T	-
X	-	-

2PL guarantees conflict serializability, but does not prevent cascading aborts.

Why 2PL guarantees conflict serializability?

When a commiting transaction has reached the end of its acquisition phase, call this the "lock point". At this point, it has everything it needs locked and any conflicting transactions either started release phase before this point, or are blocked waiting for this transaction. Two conflicting transactions will see data in the order that their locked points occur. So, the order of lock points gives us an equivalent serial schedule.

Strict 2 PL

As noted previously, 2PL suffers from cascading aborts.

Strict 2PL is similar to 2PL, except all locks are released together when transaction completes, i.e. either transaction has committed (all writes durable) OR aborted (all writes have been undone). As a result, the transaction will be atomically visible in the database, and there is no need for cascading aborts.

Theorem. Strict 2PL schedules are strict and conflict serializable, but might suffer from deadlock.

Comment: Googling says, this is called rigorous 2PL. Strict 2PL only delays releasing X locks.

Lock Management

Lock and unlock requests are handled by Lock Manager. LM maintains a hashtable, keyed on names of objects currently being locked, where each entry contains

- granted set: set of transactions currently granted access to the lock
- lock mode: type of lock held
- wait queue: queue of conflicting lock requests.

Lock Granularity

What are the objects that we lock?

We have a dilemma between locking large objects (e.g. relations) which require few locks but result in low concurrency, and locking small objects (e.g. tuples, fields) which require more locks but result in higher concurrency.

Multiple Locking Granularity

- Allows us to not have to make same decision for all transactions, as we allow data items to be of various sizes.
- Define a hierarchy of data granularities, and view it as a tree (DB → tables → pages → records)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendants in the same mode.

Warning Protocol (Jim Gray)

Idea: We allow transactions to lock at any level, but it must have proper intent locks on all its ancestors in the granularity hierarchy before getting S or X locks.

Intention lock modes:

1. IS: Intent to get S lock(s) at finer granularity
2. IX: Intent to get X lock(s) at finer granularity
3. SIX: Like S and IX at the same time. Useful for typical SQL update command, e.g. `UPDATE employees SET salary = 1000 WHERE name = 'Bob';`

Intention locks allow a higher level node to be locked in S or X mode without having to check all descendant nodes.

How it works:

- Each transaction starts from the root of hierarchy
- To get S or IS on a node, must hold IS or IX on parent node.
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.
- Release locks in bottom-up order.
- Enforce 2 phase and lock compatibility matrix rules.

Lock compatibility matrix

	IS	**IX**	**S**	**SIX**	**X**
IS	T	T	T	T	-
IX	T	T	-	-	-
S	T	-	T	-	-
SIX	T	-	-	-	-
X	-	-	-	-	-

Comment: Why IX and SIX is not compatible? If SIX is currently hold, every child node implicitly holds S lock. If I allow IX, the future X request on a child node will be granted as the S lock is only hold implicitly. However, S and X lock are incompatible, so I should not allow IX lock.

Examples:

1. T1 scans R and updates a few tuples: gets an SIX lock on R, and get X lock on updated tuples
2. T2 reads only part of R: gets an IS lock on R, and repeatedly gets an S lock on tuples of R
3. T3 reads all of R: gets an S lock on R OR proceed like T2 with lock escalation when too many low level locks are acquired.

Deadlock

Deadlocks are cycle of transactions waiting for locks to be released by each other.

Lock upgrade requests should be prioritized to avoid deadlocks. Even then, multiple lock upgrades would still end up in deadlocks.

Common techniques of deadlock prevention fails:

1. using timeout: what if transactions holds locks for a long time because it is busy doing some computation?
2. resource ordering: we cannot impose an order on the tuples or pages in our files.

Deadlock Avoidance

Assign priorities to transactions, e.g. based on age: now - start_time.

Say A wants a lock that B holds. Two possible policies

1. **Wait-Die**: If A has higher priority, A waits; else A aborts
2. **Wound-Wait**: If A has higher priority, B aborts; else A waits

Details:

- These schemes guarantee no deadlocks because there is no cycle of waiting.
- To prevent starvation, if a transaction restarts, it should get its old timestamp.
- We can use other priority schemes, such as measures of resource consumption like number of locks acquired.

Deadlock Detection

1. Create and maintain a "waits-for" graph
2. Periodically check for cycles in a graph.
3. Breaks the deadlock by aborting a transaction in cycle using some priority scheme (random, most "connected", workload/time-based, etc.)

Phantom Reads

In the above discussion, we have mechanisms to prevent the dirty read, unrepeatable read and lost update problems. However, it assumes that we are working with a static database. When insert / delete actions are accounted, we might face the phantom read anomaly.

Phantom Read occurs when a transaction *re-executes* a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently committed transaction.

A simple solution would be to lock the logical range that satisfy the search condition.

In practice, we set locks in indexes cleverly, so called "next key locking".

Optimistic CC: Validation-based Protocol

This protocol is optimistic and lock-free! → no deadlocks!

Each transaction have 3 phases:

1. Read
    - Read all DB values that needs to be read / written into a temporary local storage
    - Do all updates / writes on local storage
    - Maintains read-set (RS) and write-set (WS)
2. Validate: Check if schedule so far is serializable by ensuring no conflict between RS / WS of different transactions
3. Write: If validate returns ok, write to DB the values of WS. If not, "roll-back".

Details:

- We maintain the timestamps for
    1. start(T): start of read phase of T
    2. validate(T): start of validate phase of T → used as the timestamp of transaction ts(T).
    3. finish(T): end of write phase.
- Transactions are serialized using ts(T)
- The resulting schedule are casecadeless.
- Rolled-back transactions restart with a new timestamp.

Validation rules for transaction A:

For each active transaction B such that ts(B) < ts(A), we must have transaction A to satisfy one of the following conditions:

1. finish(B) < start(A) : nothing to check
2. start(A) < finish(B) < validate(A) : check $RS(A) \cap WS(B) = \emptyset$.
3. validate(A) < finish(B) : check $(RS(A) \cup WS(A)) \cap WS(B) = \emptyset$.

CC for indexes

We don't want to use 2PL on B+ tree pages because the concurrency would suffer badly. Instead, we use in-memory short locks (latches) in a clever way.

Idea: Upper levels of B+ tree just need to direct traffic correctly.
Possible problems:

1. Two transactions / threads modify the contents of the same node at the same time
2. One transaction / thread traverses the tree while another splits / merges the node.

Solution: Latch Coupling / "crabbing". (We have R and W latch)

Suppose we have a latch on a node N, we get a latch on the appropriate child C of N and release latch on N only when we are sure that the node is not going to be splitted / merged when updates happen.

An alternative latching scheme

Whenever we insert/delete, we apply a W latch on the root node at the beginning. One possible solution to increase concurrency would be to acquire only R latch until leaf level, and if the leaf is unsafe, then we restart and revert to the basic latch coupling scheme.

Leaf Traversal

So far, we have discussed tree traversal (top → bottom). If the latch of some child node is latched, the transaction will wait.

However, in some cases, we also need to do leaf traversal, e.g. range queries. Here, waiting can create deadlocks, e.g. imagine when 2 transactions do leaf traversal in opposite direction. Therefore, to avoid deadlocks, we **adopt a "no-wait" scheme**, i.e. any transaction that finds the next leaf locked, should abort and restart.

Recovery

When do transactions abort?

- User / application explicitly aborts
- Failed consistency check, i.e. violation of integrity constraint
- Deadlock
- System failure prior to successful commit

Why do databases crash? Operator error, configuration error, software failure, hardware failure

Buffer Management plays a key role in recovery.

1. Can a dirty page updated by a transaction T be written to disk before T commits?
    - Yes → **STEAL** policy. Need to remember the old value of flushed pages to support UNDOing the write to those pages if transaction aborts or system crashes before the transaction can finish.
    - NO → **NO STEAL** policy. Achieves atomicity without UNDO logging, but can cause poor performance, since pinned pages limit buffer replacement.
2. Must all dirty pages that are updated by transaction T be written to disk when T commits?
    - Yes → **FORCE** policy. Provides durability without REDO logging, but can cause poor performance due to lots of random I/O during commit.
    - No → **NO FORCE** policy. Flush as little as possible (only flush logs). Need to remember the new value to allow REDOing modifications in case of system crash before dirty page is flushed to DB disk.

Log-Based Recovery

Idea: For every update, record info to log to allow REDO/UNDO.

The Log

Log: An ordered list of log records, with a write buffer ("tail") in RAM.
Log records: < transaction ID, pageID, offset, length, old data, new data >

- Each log record has a **Log Sequence Number (LSN)**, which is unique and increasing
- Track **flushedLSN** in RAM (LSN of the last log that we flush to disk)
- Each data page in the database contains a **pageLSN**, i.e. the LSN of the most recent log record that updates that page
- To implement Write Ahead Logging: before page X is flushed to DB, it must satisfy pageLSN_X $\leq$ flushedLSN.

UNDO logging - STEAL/FORCE

For every action, generate undo log record that contains the OLD value

Logging rules:

1. If T modifies X, then <T, X, v> must be written to disk before the dirty page contaning X.
2. If T commits, then dirty pages must be written to disk before <COMMIT T>.

Recovery rules:

- Construct set S of incomplete transactions
- Undo actions of transactions in S in *backward* order.

Remarks: dirty pages are written early, before commit

REDO logging - NO STEAL/NO FORCE

For every action generate redo log record that contains the NEW value

Logging rules:

1. If T modifies X, then both <T, X, v> and <COMMIT T> must be written to disk before the dirty page contaning X.

Recovery rules:

- Construct set S of committed transactions
- Redo actions of transactions in S in *forward* order.

Remarks: dirty pages are written late, after commit

UNDO/REDO logging - STEAL/NO FORCE

Logging rules:

1. Log record flushed before corresponding updated page
2. ALL log records (including commit log) flushed at commit

Recovery process:

- Backward pass
    - Construct set S of committed transactions
    - Undo actions of transactions not in S
- Forward pass: Redo actions of transactions in S

Remarks: No restriction on when to write dirty pages.

Checkpointing

Problem: Recovery can be very, very SLOW because we need to read from the beginning of the log.

Simple Solution: We periodically checkpoint

1. Do not accept new transactions
2. Wait until all (active) transactions to finish
3. Flush all log records to disk (log)
4. Flush all buffers to disk (DB)
5. Write "checkpoint end" record on disk (log)
6. Resume transaction processing

Now, we don't need to examine log records before the most recent checkpoint. However, the database freezes during checkpoint

Non-Quiescent Checkpointing

Undo Log

- Record all active transactions S at <START CKPT>
- Wait until all transactions in S to finish. Undo Logging guarantees that all updates made by transactions in S are reflected on disk (DB).

During recovery, we only need to UNDO all uncommitted (including aborted) transactions after the most recent <START CKPT>.

Redo Log

- Write to disk all dirty pages updated by transactions that committed before <START CKPT>.

During recovery, we only need to REDO all transactions that commit after the most recent <START CKPT>.

Undo/Redo Log

- Flush all dirty buffer pages prior to <START CKPT>.

During recovery, we only need to REDO all committed transactions and UNDO all uncommitted (including aborted) transactions after the most recent <START CKPT>.

Handling Media Failures

Make copies of data

- Triple Modular Redundancy: keep 3 copies on separate disks, and vote for consensus
- DB Dump + log: If active database is lost, restore active database from backup + replay redo entries in log.