<u>Indexing</u>

- An **index** on a file speeds up selections on the **search key fields** for the index.
  - contains a collection of data entries, and supports efficient retrieval of all data entries k* with a given search key value k.
- **Tree-structured indexing** techniques support both **range** searches and **equality** searches.
- In a **B+ tree** index leaf pages contain data entries
  - **Inserts/deletes** keep tree height-balanced. **Log_FN cost** (F = fanout, N = # leaf pages).
  - Are ideal for range searches, also good for equality searches
  - High fanout (F) means depth rarely more than 3 or 4
- Given: equality search query using B+tree index, time required to answer the query?
  - ((Average) time to read or write disk page)*(Height of B-tree built on relation)
- **Log Structured Merge Trees**
  - Store key-value pairs
  - Put(key, new value)
  - Get(key) -> current value
- **Hash-based** indexes are best for equality selections. Cannot support range searches
  - Index is a collection of **buckets**
    - **Bucket** = primary page plus zero or more overflow pages
    - Buckets contain data entries
- **Hashing function h**: h(k) = bucket of data entries of the search key value k
  - **h(k) mod N** = bucket to which data entry with key k belongs. k1≠k2 can lead to the same bucket
  - **Static**: # buckets (N) fixed
- **Long overflow chains** can develop and degrade performance. **Extendible Hashing**: a dynamic technique to fix this problem.
- **Extendible Hashing**
  - **Global depth of directory**: Max # of bits needed to tell which bucket an entry belongs to
  - **Local depth of a bucket**: # of bits used to determine if an entry belongs to this bucket
  - **Delete**: removal of data entry from bucket
- **Alternatives for Data Entries**
  - **Alternative 1:** Index structure is a file organization for data records (instead of a Heap file or sorted file)
    - Alternative 1 implies clustered
  - **Alternatives 2 and 3**: Data entries, with search keys and rid(s), typically much smaller than data records
    - Alternatives 2 and 3 are clustered only if data records are sorted on the search key field
  - So, better than Alternative 1 with large data records, especially if search keys are small.
- Primary index vs. secondary index:

    - If the search key contains the primary key, then it is called the primary index.
    - Other indexes are called secondary indexes
    - **Unique index**: Search key contains a candidate key. No data entries can have the same value
- **Clustered vs. unclustered**: If order of data records is the same as (or `close to'), order of data entries, then it's a clustered index

	Clustered	Unclustered
Selective *(10% retrieved)*	good	not worth it
Not selective *(95% retrieved)*	useful, small improvement	definitely not worth it

Composite Search Keys

- To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.
- If condition is: 20<*age*<30 AND 3000<*sal*<5000:
    - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- If condition is: *age*=30 AND 3000<*sal*<5000:
    - Clustered <*age,sal*> index much better than <*sal,age*> index!
- Composite indexes are larger, updated more often.

- **Index Selection Guidelines**
    - **Exact match** condition suggests **hash index**.
    - **Range query** suggests **tree index**.
        - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates
- **Comparing File Organizations**
    - Heap files (random order; insert at end of file)
    - Sorted files, sorted on <age, sal>
    - Clustered B+ tree file, Alternative (1), search key <age, sal>
    - Heap file with unclustered B + tree index on search key <age, sal>
    - Heap file with unclustered hash index on search key <age, sal>

Cost of Operations

	Scan	Equality	Range	Insert	Delete
Heap File	BD	.5BD	BD	2D	Search + D
Sorted File	BD	$D\log_2 B$	$D(\log_2 B$ + #matching pages)	Search + BD	Search + BD
Clustered Tree Index	1.5BD	$D\log_F 1.5B$	$D(\log_F 1.5B$ + #matching pages)	Search + D	Search + D
Unclustered Tree Index	BD(R+.15)	$D(1+\log_F .15B)$	$D(\log_F .15B$ + #matching recs)	Search + 3D	Search + 3D
Unclustered Hash Index	BD(R+.125)	2D	BD	4D	4D

B: The number of data pages **R:** Number of records per page **D:** (Average) time to read or write disk page

Relational Operations

- We will consider how to implement:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Join* ($\bowtie$) Allows us to combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY
  - *Order By* Returns tuples in specified order.
- After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

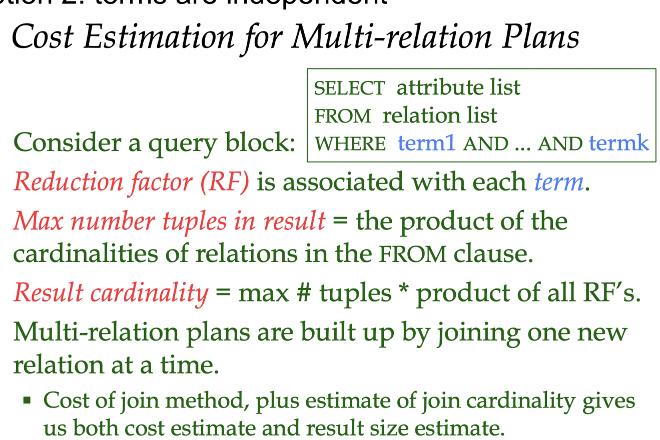
- **Two-Way External Merge Sort**
  - Cost: $2N((\log_2 N) + 1)$; N is # of pages
- **External Merge Sort**
  - Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
  - Cost = 2N * (# of passes)
  - E.g., with 5 buffer pages, to sort 108 page file:
    - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
    - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
    - Pass 2: 2 sorted runs, 80 pages and 28 pages
    - Pass 3: Sorted file of 108 pages
- **B+ Trees**
  - Worse case I/O: pN
    - p: # records per page N: # pages in file
  - **Clustered B+ tree** is good for **sorting**; unclustered tree is usually very bad
- Algorithms for evaluating relational operators use some simple ideas extensively:
  - **Indexing**: Can use WHERE conditions to retrieve small set of tuples (selections, joins)
  - **Iteration**: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - **Partitioning**: By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs

- **Simple Nested Loops Join**
  - **Cost:** M + pR * M * N
    - M and N are number of pages; p_R is # of tuples
- **Page-Oriented Nested Loops Join**
  - **Cost**: M + M * N
- **Block Nested Loops**
  - **Cost**: Scan of outer + #outer blocks * scan of inner
  - $M + N * \lceil M / B\text{-}2 \rceil$
- **Sort-Merge Join**
  - **Cost:** $O(M\log_B M) + O(N\log_B N) + (M+N)$
    - B is buffer size
- **Hash-Join**
  - **Cost**: 3(M+N)
- **Two Approaches to General Selections**
  - **Option 1**: Consider day<8/9/94 AND bid=5 AND sid=3.
    - A B+ tree index on day can be used; then, bid=5 and sid=3 must be checked for each retrieved tuple.
    - A hash index on could be used; day<8/9/94 must then be checked on the fly
  - **Option 2**: Consider day<8/9/94 AND bid=5 AND sid=3. If we have a B+ tree index on day and an index on sid, we can:
    - retrieve rids of records satisfying day<8/9/94 using the first, rids of records satisfying sid=3 using the second,
    - intersect these rids,
    - retrieve records and check bid=5.
- Algorithms: single relation **sorting** or **hashing** based on all remaining attributes.

Relational Query Optimization

- **Basics of Query Optimization**
  - Convert selection conditions to conjunctive normal form (CNF):
  - " (day<8/9/94 OR bid=5 OR sid=3) AND (rname='Paul' OR sid=3)
  - Interleave FROM and WHERE into a plan tree for optimization.
  - Apply GROUP BY, HAVING, DISTINCT and ORDER BY at the end, pretty much in that order
    - ❖ Statistics about each relation (R) and index (I):
      - Cardinality: # tuples (NTuples) in *R*.
      - Size: # pages (NPages) in *R*.
      - Index Cardinality: # distinct key values (NKeys) in *I*.
      - Index Size: # pages (INPages) in *I*.
      - Index height: # nonleaf levels (IHeight) of *I*.
      - Index range: low/high key values (Low/High) in *I*.
      - More detailed info. (e.g., histograms). More on this later…
- **Query Evaluation plan:**
  - **Query evaluation plan** is an extended RA tree, with additional annotations

        - access method for each relation and implementation method for each relational operator.
- An **access method (path)** is a method of retrieving tuples:
    - File scan, or index scan with the search key matching a selection in the query
- **Pipelined Evaluation**
    - **Materialization:** Output of an op is saved in a temporary relation for use (namely, multiple scans) by the next operator
    - **Pipelining**: No need to create a temporary relation. Avoid the cost of writing it out and reading it back. Can occur in two cases
        - **Unary operator**: when the input is pipelined into it, the operator is applied on-the-fly, e.g. selection on-the-fly, project on-the-fly.
        - **Binary operator**: e.g., the outer relation in indexed nested loops join.
- All left-deep join trees: all the ways to join the relns one-at-atime, with the inner reln in the FROM clause. " Considering all permutations of N relns, N factorial
- **Reduction factor** (RF) or **Selectivity** of each term:
    - Assumption 1: uniform distribution of the values!
    - Term col=value: RF = 1/NKeys(I), given index I on col
    - Term col>value: RF = (High(I)-value)/(High(I)-Low(I))
    - Term col1=col2: RF = 1/MAX(NKeys(I1), NKeys(I2))
        - NKeys = number of keys
- Max. number of tuples in result = the product of the cardinalities of relations in the FROM clause
- **Result cardinality** = Max # tuples(add up all the tuples) * product of all RF's
    - Assumption 2: terms are independent



- **Enumeration of Left-Deep Plans**
    - Enumerate using N passes (if N relations joined):
    - **Pass 1**: Find best 1-relation plan for each relation. Include index scans available on "sargable" predicates.
    - **Pass 2**: Find best ways to join result of each 1-relation plan (as outer) to another relation. (All 2-relation plans.)
    - **Pass N**: Find best ways to join result of a (N-1)-relation plan (as outer) to the N'th relation. (All N-relation plans.)

- **System R Limitations**
  - **Uniform distribution of values**:
    - Term col=value has RF 1/NKeys(I), given index I on col
    - Term col>value has RF (High(I)-value)/(High(I)-Low(I))
  - **Predicates are independent**:
    - Result cardinality = max # tuples * product of Reduction Factors of matching predicates
- **Nested Queries With No Correlation**
  - a query that appears as an operand of a predicate of the form “expression operator query”.
  - the nested block does not contain a reference to tuple from the outer.
- **Nested Queries With Correlation**
  - the nested block contains a reference to a tuple from the outer

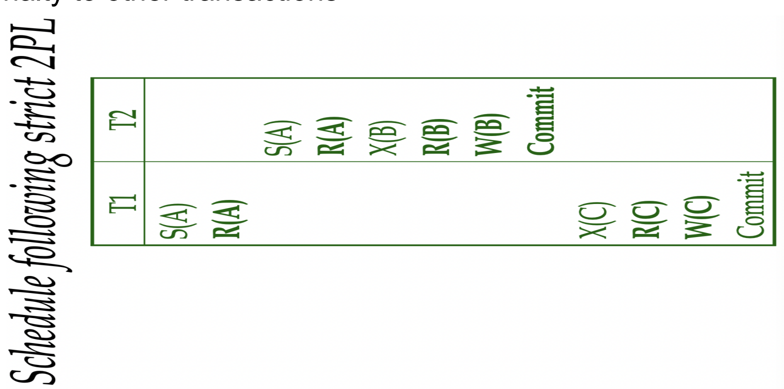
Concurrency

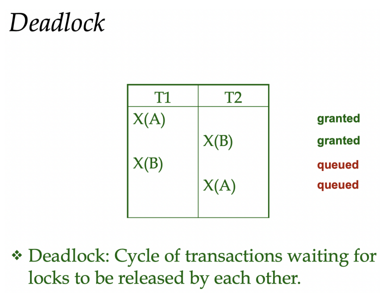
- Database systems ensure the **ACID** properties:
  - **Atomicity**: all operations of transaction reflected properly in database, or none are.
  - **Consistency**: each transaction in isolation keeps the database in a consistent state (this is the responsibility of the user).
  - **Isolation**: should be able to understand what’s going on by considering each separate transaction independently.
  - **Durability**: updates stay in the DBMS!!!
- A transaction is seen by DBMS as sequence of reads and writes
  - read of object O denoted R(O)
  - write of object O denoted W(O)
  - must end with Abort or Commit\
- **Scheduling Transactions**
  - **Serial schedule:** Schedule that does not interleave the actions of different transactions.
  - **Equivalent schedules**: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule
  - **Serializable schedule**: A schedule that is equivalent to some serial execution of the transactions.
- **Two schedules are conflict equivalent if**:
  - Involve the same actions of the same transactions
  - Every pair of **conflicting actions** (of committed trans) are ordered the same way.
  - Alternatively: S can be transformed to S’ by swaps of non-conflicting actions.
- A schedule is **conflict serializable** if and only if its dependency graph doesn’t have a loop

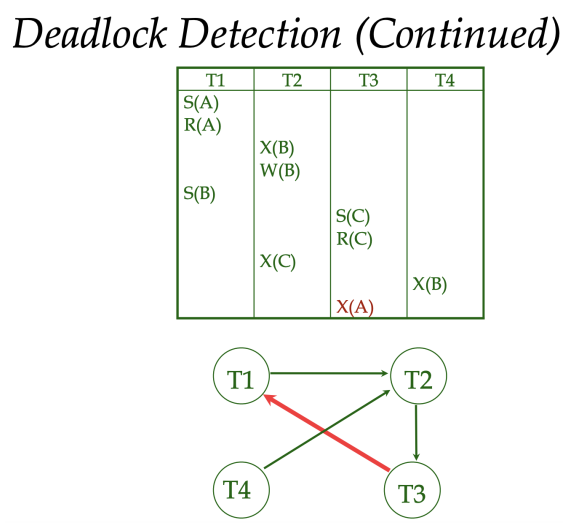
Transaction state

- **Active**: a transaction is active while executing.
- **Partially committed**: after the final statement has executed
- **Failed**: after discovery that normal execution cannot proceed
- **Aborted**: transaction has been rolled-back and database restored to prior state.
- **Committed**: after successful completion.

- **Recoverable schedule**: For any transactions Ti and Tj: if Tj reads data written by Ti, then Ti commits before Tj commits
- **Cascading Rollback**: a single transaction failure leading to a series of rollbacks
- **Cascadeless schedule**: For any transactions Ti and Tj: if Tj reads data written by Ti, then Ti commits before read operation of Tj
- A schedule is **strict** if:
  - A value written by a transaction T is not read or overwritten by other transactions until T either aborts or commits
  - **Strict schedules are recoverable and cascadeless**
- When a transaction requests a **lock**, it must wait (block) until the **lock** is granted
- **Two-Phase Locking (2PL)**
  - Each Xact must obtain a S (shared) **lock** on object before **reading**, and an X (exclusive) **lock** on object before **writing**
- **Strict Two-phase Locking Protocol**:
  - Same as 2PL, but a transaction can not request additional locks once it releases any locks.
  - All X (exclusive) locks acquired by a transaction must be held until completion (commit/abort).
- 2PL ensures **conflict serializability**
- Strict 2PL ensures **conflict serializable** and **cascadeless** schedules
- 2PL reduces concurrency because
  - Holding locks unnecessarily
  - Locking too early
  - Penalty to other transactions







- **Deadlock Prevention**
  - **Wait-Die**: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
  - **Wound-wait**: If Ti has higher priority, Tj aborts; otherwise Ti waits
- **Lock-based schemes** resolve conflicting schedules by **blocking** and **aborting**
- **Anomaly**: two consistency-preserving committed transactions that lead to an inconsistent state

Recovery

- Types of failure:
  - Transaction failure
    - partially-executed transaction cannot commit
      ➡ changes must be removed: ROLLBACK
  - System failure
    - volatile memory lost
      ➡ updates of committed Xact persist
      ➡ updates of aborted or partial Xacts removed
  - Media failure
    - corrupted storage media
      ➡ database brought up-to-date using backup
- **UNDO**: removing effects of incomplete or aborted transaction (for atomicity)
- **REDO**: re-instating effects of committed transactions (for durability)
- **STEAL** (why enforcing Atomicity is hard)
  - To steal frame F: Current page in F (say P) is written to disk; some Xact holds lock on P
- **NO FORCE** (why enforcing Durability is hard)
- **Log**: An ordered list of REDO/UNDO actions
- Each log record has a unique **Log Sequence Number (LSN)**
- Each data page contains a pageLSN
- **Transaction Table**: One entry per active Xact.
  - Contains XID, status (running/committed/aborted), and lastLSN.
- **Dirty Page Table**: One entry per dirty page in buffer pool.

- Contains recLSN -- the LSN of the log record which first caused the page to be dirty

Parallel Data Processing

Final Review

Homework 5:

- Cache miss rate remains above zero when the buffer is as large as the total number of data pages

- Buffer will begin empty and be considered as misses

- For nested loops workload, performance of MRU is good because more likely that one will remain in buffer at any given time

Homework 6:

- B+ tree index, it is possible to evaluate query with an index-only plan because all required fields are present in index

- Equality unclustered hash index

- Range clustered B+ tree index

- Small Range clustered B+ tree index or unclustered B+ tree index (# matching pages and # matching record end up being relatively similar)

- Inequality heap file (require full scan)

Homework 7:

- Runs in first pass = HIGH(# of pages / # buffer pages)

- Passes to sort file = 1 + HIGH(logB-1(N/B))

- Total I/O cost of sorting = 2N * # of passes

- Cost of block nested loops join = M + N * (M / B-2) = Scan of Outer + Scan of Inner * # Outer Blocks

- Cost of sort-merge join = Cost of Sorting R + Cost of Sorting S + Cost of Merge = 2N * # of passes + (M + N)

- Cost of hash join If B > sqrt(M) Cost = 3(M+N)

Homework 8:

- Cardinality = N*(count item / total counts) * (months / total months) *** INCORRECT

- Assumption that distribution is uniform

- Information needed about relation for query optimizer statistics such as size of tables and high and low index values, distribution of items, what indexes exist on fields

- System-R only allows for left-deep join

- B+ tree index on join operations, clustered index on range searches to filter

Homework 9

(a) T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Commit
(b) T1:R(X), T2:W(X), T1:W(X), T2:Abort, T1:Commit
(c) T1:R(X), T2:W(X), T1:W(X), T2:Commit, T1:Commit
(d) T1:R(X), T2:R(X), T1:W(X), T1:Commit, T2:W(X), T2:Commit
(e) T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Abort
(f) T1:W(X), T2:R(Y), T1:R(Y), T2:R(X), T1:Commit, T2:Commit
(g) T1:R(X), T2:R(X), T1:Commit, T2:W(X), T2:Commit

	Conflict Serializable	Recoverable	Cascadeless	Strict
(a)				
(b)	✓	✓	✓	
(c)		✓	✓	
(d)		✓	✓	✓
(e)	✓			
(f)	✓	✓		
(g)	✓	✓	✓	✓

- Conflict Serializable: no conflicting operations (R/W or W/W) among committed transactions. Ignore aborted transactions
- Recoverable: For any transactions Ti and Tj, if Tj reads data written by Ti, then Ti commits before Tj commits
- Cascadeless: for any transactions Tj and Tj, if Tj reads data written by Ti, then Ti commits before read operation of Tj

- Strict: if value written by a transaction is not read or overwritten by another transaction until the first either aborts or commits
- Strict 2PL conflict seralizability and no cascading rollback
- Deadlock is possible because the lock releases occur after the commits
    - Not possible when unlocks are in middle of transactions
- Cascading rollback is possible because unlocks occur in middle of transaction and previous transaction could read a new modification