Database Management System Implementation
Paolo Papotti

---

1. Introduction

A **database** is a large and integrated collection of data that models a real-world enterprise. It contains *entities* and *relationships*. For example, a university database might contain information about the following:

- Entities: students, faculty, courses and classrooms.
- Relationships: students' enrollment in courses and use of rooms for courses.

A **database management system**, or **DBMS**, is a software designed to store and manage databases: provide efficient and shared access to persistent data.

A **data model** is a collection of high-level concepts for describing data hiding many low-level storage details. Most DBMS today are based on the *relational data model*: relation, basically a table with columns and rows.

A **schema** is a description of data in terms of data model: every relation has a schema describing columns.

Example: University Database

- Entities: Students, Courses, Professors
- Relationships: Who takes what, Who teaches what
- Logical Schema:
  - Students (sid: *string*, name: *string*, gpa: *float*)
  - Courses (cid: *string*, cname: *string*, credits: *int*)
  - Enrolled (sid: *string*, cid: *string*, grade: *string*)

The data in a DBMS is described at three levels of abstraction, consisting of a schema at each of these levels:

- External (views): describes how users see the data. A view is like a relation but not stored explicitly.
- Conceptual (logical): describes all the relations stored in the database (unsorted, index on column).
- Physical: describes how previous relations are stored on secondary storage devices (disks, tapes).

Queries are questions that can be asked by a user in order to obtain information from a database. Those questions are asked in a specialized language provided by the DBMS called **Query Language**.

- SQL: Structure Query Language
- DDL: Data Definition Language
- DML: Data Manipulation Language

Databases vs. File Systems: why databases are better?

1. Support of multiple views
   - Different users viewing data in different ways
   - A view is a virtual subset of the database (no need for multiple copies of data)

2. Insulation between program and data
    - The schema can be modified without a change in a program
3. Data abstraction
    - A data model describing relationships without concern for physical details

Data independence

- Logical data independence: easily update views when there are changes in logical structure of data
- Physical data independence: hiding how data is being stored on storage

BDMS challenges:

1. Fast and declarative: use of indexes for faster scan
2. Limited memory: use of buffers
3. Concurrent transactions: transactions are a sequence of database actions (r/w) that guarantee:
    a. Atomicity: an action either completes entirely or not at all
    b. Consistency: an action results in a state that conforms all integrity constraints
4. Crash: if there is a crash during a transaction, no changes should persist

2. Database Design

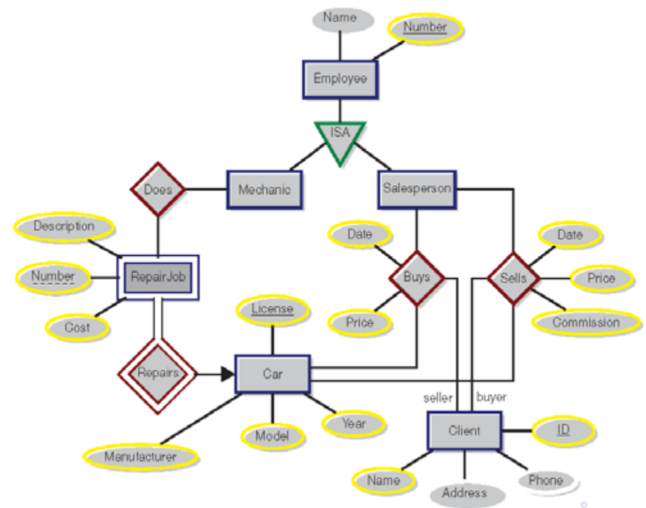
Why? Agree on structure of database before deciding on a particular implementation.
Consider issues such as: what entities to model, how entities are related, what constraints exist on domains, how to achieve good designs.
→ we discuss E/R diagrams.

Database design process:

1. Requirements analysis: what is stored, how it is used, who access the data.
2. Conceptual design: high-level description of db, balanced precision to be understood by technical and non-technical people.
3. More:
    a. Logical db design: mapping a conceptual schema
    b. Physical DB design: choice of storage structure (B-trees, hashed files linked structures)
    c. Security design.

The E/R model:
An **entity** is an object of the real world that is distinguishable from other objects and described by a set of **attributes**. A collection of similar entities (having the same attributes) is called entity set.
A **key** is minimal set of attributes that uniquely identify an entity.
A **relationship** is an association between two entities (e.g. a subset of a cross-product). It can also have attributes.
We can always use a new entity instead of a relationship to allow multiple instances of each entity combination.

Some rules:
The arrowhead is drawn at the “one” end of the relationship type: $X \to Y$ means that there exists a function mapping from X to Y.
Entity sets are **weak** when their key (dashed underline) comes from other classes to which they are related called the owner.



3. SQL

→ **S**tructured **Q**uery **L**anguage, or SQL, is a standard language for querying and manipulating data. SQL is a declarative very high-level programming language.
→ **D**ata **D**efinition **L**anguage, or DDL, to define relational schemata (create/alter/delete tables). Compiling DDL results in a *Data Dictionary* that stores metadata.
→ Data Manipulation Language, or DML, to insert/delete/modify tuples in tables.

Relation or Table – Attribute or Column – Tuple or Row
Atomic types: characters, numbers, money, datetime, etc. but not lists and sets.
Schema of table: name + attributes + types
Key: minimal subset of attributes that acts like a unique identifier for tuples.
SQL queries: selection (rows), projection (columns), join (tables), aggregation
Semantics of a join: cross product → selection/condition → projection
Semantics of group by: from/where → group by → having → select

$$CREATE\ TABLE\ tablename\left(att_1\ type_1, \dots, att_n\ type_n, PRIMARY\ KEY\ (att_i), FOREIGN\ KEY\ (att_j)\right)$$

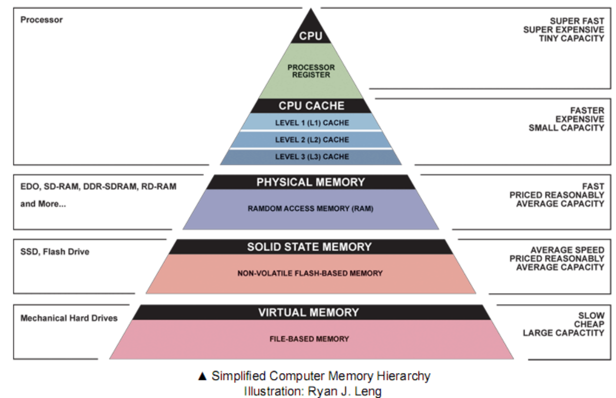
$$INSERT\ INTO\ tablename\ [attribute\ list]\ values\ (val_1, \dots, val_k)$$

4. Data Storage

Computer memory issues:

- Storage capacity
- Random access speed
- Bandwidth capacity (sequential access)
- Cost
- Persistence

Memory	Typical Access Time	Typical Sizes
Registers	$1cyc$	$16 \times 8B$
L1	$4cyc$ ($\times$ 1)	$16 \times 32B$
L2	$10cyc$ ($\times$ 2.5)	$256KB$
L3	$60cyc$ ($\times$ 15)	$8MB$
Main Memory	$60ns$ ($\times$ 45)	$16GB$
Flash/Hard Disk	$5ms$ ($\times$ 3.7M)	$2TB$



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

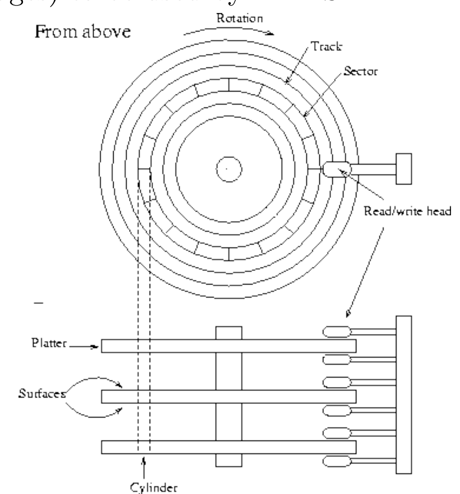
Data lives in *secondary storage* (as blocks) but must be in *memory* (as pages) to be used by DBMS.

Hard Disks

→ Slow random access
→ Durable: once on disk, data is safe.
→ Cheap

Time for a disk block access or disk I/O = seek time + rotational delay + transfer time

1. Seek time: access specific sector: $3.5ms$ to $15ms$
2. Rotational delay: access specific sector: avg. ½ revolution: $2ms$ to $7ms$ (depend on rpm)
3. Transfer time: block size / transfer rate



SSD

→ Robust, no moving parts
→ Non-volatile.
→ $\times$ 100 faster random access than disks and slower random read
→ More expensive than HDD.

Case study: calculate access time for $8KB$ data block

Device	Rotational Speed	Avg. Seek Time	Transfer Rate	Access Time
Seagate Cheetah	$15K$ rpm	$3.4ms$	$163MB/s$	$3.4ms + 2ms + 0.05ms = 5.45ms$
Seagate Barracuda	$7.5K$ rpm	$9ms$	$103MB/s$	$9ms + 4ms + 0.07ms = 13.07ms$
Seagate Pulsar (SSD)	–	–	$370MB/s$	$0ms + 0ms + 0.3ms = 0.3ms$

Sequential read: read 1000 blocks of size $8KB$
→ Random access

HDD = $1000 \times 5.45ms = 5.45s$
SSD = $1000 \times 0.3ms = 0.3s$

→ Seq. read adjacent blocks

HHD = seek + rot. + 1000tr + 16 track-to-track = $3.4 + 2 + 50 + 3.2 = 58.6ms$
SSD = $(1000 \times 8/128) \times 0.3 = 18.9ms$

Cost of writing is similar to reading unless we want to verify, add (full) rotation + block size / transfer rate

---

Random Access Memory or **Main Memory**
→ Fast: $\times 10$ faster for sequential access and $\times 100{,}000$ faster for random access
→ Volatile: data can be lost if crash occurs, power goes out, etc.
→ Expensive: for $70€$, $16GB$ of RAM vs. $2TB$ of disk.

Optimization

- Disk scheduling algorithm: with multiple requests, choose the one that requires the smallest arm movement (e.g. elevator algorithm).
- Track skewing: align sector 0 of each track to avoid rotational delay during longer sequential scan.
- Pre-fetch, double buffering: while processing data in buffer 1 by program, read data to be processed next in buffer 2 so buffering time is $R + nP$ instead of $n(R + P)$.
- RAID: data is copied on multiple disks
    - RAID0: data is not duplicated by spread across 2 disks (disk failure → half of data lost) but access data is faster.
    - RAID1: data is copied on more than one disk (faster read)
    - RAID5: spread data across multiple disks and store *parity* used to rebuild data if failure
    - RAID10 (1+0): spread AND copy data across (at least 4) disks (fast & fault tolerant)

5. Buffers

Page: main memory representation of a block.
Disk vs. Main Memory:

- Main memory: fast but limited capacity, volatile.
- Disk: slow but large capacity, durable

How we effectively utilize both?
A **buffer** is a region of physical memory (main memory) used to store temporary data: intermediate data between disk and main memory because reading/writing to disk is slow (need to cache data).
How? Disk pages are brought into buffer *pool* as needed and loaded into memory *frames*.
Hight-level code requests (pins) pages from the buffer and releases (unpins) pages after use.
A *replacement policy* decides which page to evict when full.
Read (from disk to buffer), Flush (evict from buffer & write to disk), Release (evict from buffer wo writing)

BDMS vs. OS
The BDMS knows more about access patterns (can predict order of access & avoid random access).
The ability to pin and flush to disk.

The Buffer Manager: pins & unpins pages, handles & executes replacement policy (LRU, MRU, Clock)

- $pin(p\#)$: Request $p\#$ from the buffer manager, load it to memory, mark page as clean
  Return a reference to the frame containing $p\#$
- $unpin(p\#, dirty)$: Release $\#p$ making it a candidate for eviction
  Must set dirty to $True$ if page was modified

The effectiveness of a buffer manager depends on the replacement policy: how to choose an unpinned page to flush? what to keep in memory?
Temporal Locality: small distance in time for same address. E.g. $P1, \boldsymbol{P7}, P42, \boldsymbol{P7}, P2$
Spatial Locality: small distance in time for similar address. I.e. $dist(P_i, P_j) = |i - j| < t$

Replacement Policies:

- Least Recently Used (LRU): evict the page whose latest unpin in longest ago.
    - Need extra space, time and maintenance (ordered list).
- LRU-K: same as LRU but consider last K unpin calls.
    - Given a reference, $r4 = p2$ means the fourth page we accessed is page2.
    - $B(r, p, k) = x$ means that page p was accessed at time $T - x$ and $k - 1$ other time in $r[T - x, T]$. The next victim is $v = \underset{p \in P}{\operatorname{argmax}} B(r, p, k)$
    - E.g. given $r = p7, p8, p2, p4, p8, p7, p2, p5, p6, p7, p1$, $B(p1,2) = \infty, B(p2,2) = 8, \ldots$
- Most Recently Used: evict the page that has been unpinned most recently.
- Random: pick a victim randomly.
- Clock: scan periodically and use a second-chance bit.

The five-minute rule
If a page is accessed more often, keep in memory; otherwise, remain on disk and read when needed.
Breakeven point: $CD = CM \Rightarrow \frac{\$D}{X \times I} = \frac{\$M}{P} \Rightarrow X = \frac{\$D \times P}{\$M \times I} = \frac{\$2000 \times 128}{\$15 \times 64} \approx 5\ mn.$

6. Files, Pages and Records

Files

Page = main memory representation of a block
Disk space manager: takes care of the (de)allocation of pages within a database
File system model:

- A file is one or more pages
- A page contains one or more records
- A record corresponds to one tuple

Heap file: the most important type of files in a DB, stores records in no particular order
Record IDs (RID): used as record address, identifies the page containing the record

Heap file interface: create/destroy hf, insert/delete/get record, initiate a sequential scan
Which page to pick to insert new record? Best Fit, First Fit, Next Fit

Pages

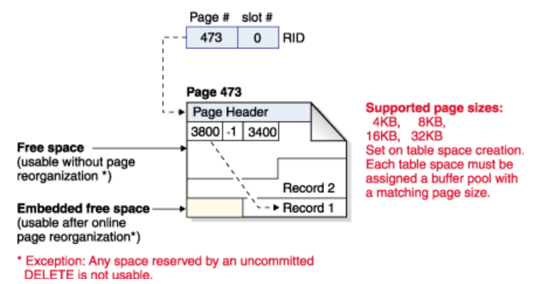
All data sit in pages, pages have IDs, a page is a collection of slots for records

$$RID =< PID, Slot\ N° >$$

Page Header: contains # of records and bitmap for each slot (e.g. 0101101, 0 for empty & 1 for full)

Fixed-length components: for static or append-only data (logs)

Variable-length components: for variable-length



Records

Representing data with bytes (1 byte = 8 bits)

- Integer (short): 2 bytes: 35 = 00000000 00100011
- Real, floating point: n bits for mantissa, m for exponent, ...
- Characters: e.g. UTF-8: A = 1000001, a = 1100001, 5 = 0110101
- Boolean: $TRUE$ = 1111 1111, $FALSE$ = 0000 0000
- Date/Time: e.g. Integer: timestamp, characters: YYYYMMDD

Fixed Format Record	Variable Format Record
For a record with 3 integer fields A, B and C A at start address B at start address + 4 C at start address + 20	2 | 5 I 64 | 4 S 4 FORD Number of fields Field code + Type (+ Length) + Value

Unspanned (records within one page/block) vs. Spanned (essential id record size > block size)

7. Layouts

Layouts

Row Stores	Column Stores
+ Easy to implement + Good for transactional workload + Good for single row access	+ Good for single to few column access + Good for analytical workload (SELECT avg(c))
− Bad for analytics − Bad for wide tables	− Bad when accessing multiple attributes − Bad for transactional workload

Best layout? Depends on the query!

PAX (Partition Attributes Across) Layout: horizontal partitioning (each chunk in page). Number of horizontal partitions: $1 < i < N$ (tuples per partition $\lceil N/i \rceil$). Higher numbers of tuples per partition → Closer to a row layout (better update and worse read)

Advantages:

- Improves locality for single attributes
- Data values are reorganized inside a page only
- Record reconstruction is cheap

! Not the best solution for analytics

PAX PAGE

PAGE HEADER | 0962 | 7658
3859 | 5523
Jane | John | Jim | Susan
30 | 52 | 45 | 20

Compression

Why? Less storage space, less bandwidth

Goal: decompress + read compressed < read uncompressed

Dictionary Compression: e.g. replace city by cityID and add a city dictionary

Advantages: string converted to integer; single row access still possible

Disadvantages: extra joins to dictionary

8. File Organization

Different file organizations:

1. Files of randomly ordered records (heap file)
2. Files sorted on some record fields
3. Files hashed on some record fields

Hash function maps a record r onto a page of the file

- $h(r)$ = bucket in which r belongs
- file is collection of buckets, h does not determine placement in page

For example, h can use lower 3 bits of the first INT field to compute bucket:

$h(< 42, true, "Nice" >) = 2 \ (42 = 101\mathbf{010})$

$h(< 14, true, "Paris" >) = 6 \ (14 = 1\mathbf{110})$

$h(< 26, false, "Lyon" >) = 2 \ (26 = 11\mathbf{010})$

Fill page to 80% of its capacity to avoid overflow.

3 file organizations in 5 disciplines:

1. scan to read all records in a file
2. search with equality test
3. search with range selection
4. insert a given record in a file
5. delete a record

Cost Model

b: # data pages in the file

r: # records in a page

D: time needed to read/write in disk page ($\sim 5 - 15ms$)
C: CPU time to process a record ($\sim 0.1\mu s$)
H: CPU time to apply a function (e.g. hash function) ($\sim 0.1\mu s$)

	Scan cost	Search with =	Range selection	Insertion cost	Deletion cost (given RID)
Heap file	$b(D + rC)$	On PK $1/2b(D + rC)$ Not on PK $b(D + rC)$	$b(D + rC)$	$2D + C$	$D + C + D$
Sorted file	$b(D + rC)$	$D \log_2 b + C \log_2 r$	$D \log_2 b + C \log_2 r$ $+ ceil(n/r)D + nC$	$D \log_2 b + C \log_2 r$ $+ \frac{1}{2}(D + rC + D)$	$D + \frac{1}{2}b(D + rC + D)$
Hashed file	$(100/80)b(D + rC)$	On PK $H + D + 1/2\, rC$ Not on PK $H + D + rC$	$(100/80)b(D + rC)$	$H + D + C + D$	$D + C + D$

There is no single file organization that responds equally fast to all operations.
To search quickly along multiple attributes, we must keep multiple copies of the records each sorted by one attribute. A lot of space needed!

9. Indexing

Index structures offer the advantages of sorted files and support insertions/deletions efficiently.
Instead of sorting the table by a specific field, we maintain an index of each field and search over them (e.g. By_Year_Index, By_Author_Index, etc.)
Why?

- Search: quickly find all records which meet some conditions on the search key attributes
- Insert/Remove entries with low overhead

<u>Clustered index</u>: if the data file is sorted on index search key, the index is said clustered. For range selection, query the *index* once for a record with X=lower, and then *sequentially* scan data file until field X>upper.
<u>Unclustered index</u>: query the index for X=lower then scan index entries to pages *scattered* over file.

NB

- A data file can have at most one clustered index but any number of unclustered indexes.
- For exact search, no difference between clustered and unclustered.
- For range search, big difference between 1 random + R sequential I/O and R random I/O.

<u>Sparse (vs. Dense) index</u>: do not index all the attribute values. To search a record with A=k, locate the largest index entry k'^* such that k' <= k, access the data page and sequentially scan (since clustered) to find records.

<u>Covering index</u>: an index containing all needed attributes (SELECT and WHERE attributes). The query can be answered using the index only.

10. Trees

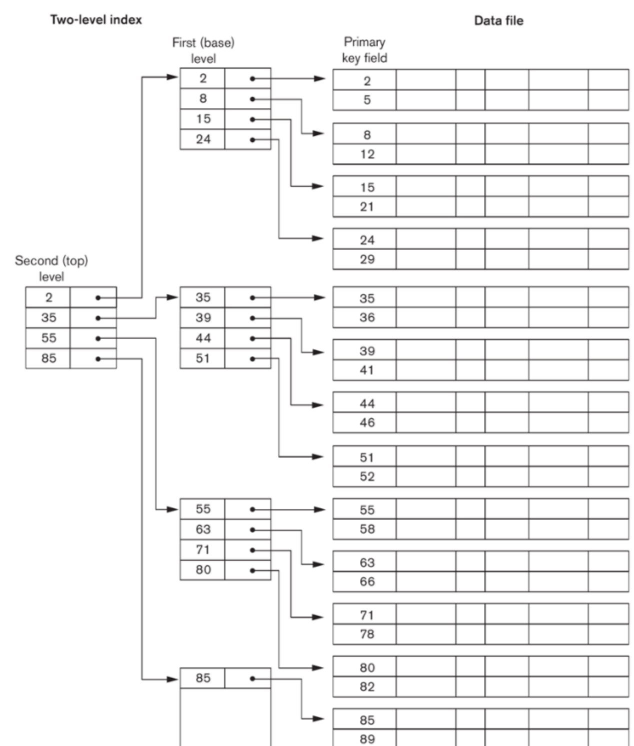
<u>Binary search</u>: SELECT * FROM customers WHERE zipcode between 8880 and 8999
Assume file sorted on zipcode, use binary search to get lower range limit and then sequentially scan.
+ Need to read $\log_2 r$ records during the search phase.
− Need to read as many pages

<u>Index Sequential Access Method</u>: maintain an index file with sorted entries $< k_i, p_i >$ where k_i is the minimal A value on the data file page p_i (k_i is a separator between content of page p_{i-1} and p_i).
+ Access fewer pages comparing to binary search.
− Large data → index search can be slow.

<u>Multi-level ISAM</u>: a tree with one data level and many index levels
Upper index levels remain static: not affected by insertion/deletion.
The most efficient order-aware index structure so far.
N: number of pages in the data file.
F: fanout. $F = \#children/index\ nodes = 1000$
As we search down the tree, search space is repeatedly reduced by a factor F: $N \times 1/F \times \ldots \times 1/F$
Index searching ends after s steps when the search space is reduced to 1.

$$N \times \left(\frac{1}{F}\right)^s = 1 \rightarrow s = \log_F N$$

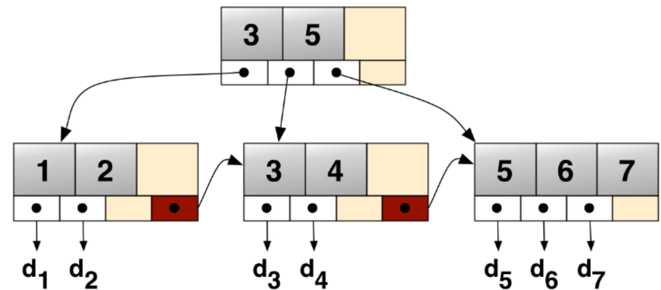
For $F \gg 2$, $\log_F N \ll \log_2 N$: more efficient than binary search.
e.g. for $F = 1000$, a tree of height 3 can index a file of one billion (10^9) pages. 3 I/Os locate any data page.



B-Tree
A self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.
1 node = 1 page

B+ Tree
Make <u>leaves</u> into linked lists for sequential scan (range queries).
d: the order/degree of the tree ($d \leq n \leq 2d$ where n is the number of entries for each node).

- Non-leaf nodes contain pointers to other nodes
- Leaf-nodes contain records or pointer to the previous/next leaf node



Searching a B+ Tree

1. Start at the root
2. Proceed down to the leaf
3. Sequential traversal (for range queries)

How large id d?

Example: key size = 4 bytes, pointer size = 8 bytes, block size = 4096 bytes.

We want each node to fit on a single block/page: $2d \times 4 + (2d + 1) \times 8 \leq 4096 \rightarrow d \leq 170$

In practice

Typical order = 100, typical fill factor = 67%, average fanout = 133

Typical capacities:

- Height = 4: $133^4 = 312{,}900{,}700$ records
- Height = 3: $133^3 = 2{,}352{,}637$ records

Top levels (in buffer pool):

- Level 1: 1 page = 8 Kbytes
- Level 2: 133 pages = 1 Mbytes
- Level 3: 17,689 pages = 133 Mbytes

Simple cost model for search:

f: fanout, N: total number of pages to index, F: fill factor

We need to have room to index N/F pages. With h levels, we can index f^h pages.

So $h = \left\lceil \log_f \frac{N}{F} \right\rceil$

If we have B available buffer pages, levels L_B we can fit in the buffer satisfy: $B \geq \sum_{l=0}^{L_B - 1} f^l$

The IO cost for exact search is $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + 1$

The IO cost for range search is $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + cost(OUT)$

(Read one page in each level except those in buffer, and read the record(s))

Inserting in a B+ Tree

1. Call $\text{search}(k)$ to find the page p to hold the record
2. If p has space ($n < 2d$) then store k^* in p

If no space in p

1. Split into p and p' (can happen recursively and lead to split of root which increases h).
2. Distribute entries of p and new entry k across new p and p'.

Deleting from a B+ Tree

1. Call $\text{search}(k)$ to find the page p containing the record

2. Delete the record
3. If $n \leq d-1$, then entries from a neighbor page migrate to p and update separator for parent node

When the keys are of VARCHAR type, we reduce keys by using only prefixes because index keys (non-leaf) are used only to direct traffic.

Multi-Dimensional Index

Index intersection: scans in separation, then compute intersection between the two rid lists.
R-Tree: a data structure used for indexing multi-dimensional information (e.g. geographical coordinates)

Hash Index

1. Compute $h(k)$
2. Access primary bucket page with $h(k) \in [0, N-1]$ (N disk pages)
3. Search/insert/delete record on page (or access overflow chain)

Hash function: $h(k) = k \; mod \; N$ (for $N = 2^d$, we consider last d bits of k)
! Some primary buckets can be (almost) empty
Extendible Hashing: Use directory of pointers to buckets, double the number of pointers by doubling directory, split only the bucket that overflow: much cheaper since directory is much smaller

11. Operations

External Merge & Sort

Merge two sorted lists of sizes M and N with 3 buffer pages.

Joins

A join is a subset of cross- product.

Nested Loop Join:

1. Loop over the tuples in R
2. For every tuple in R, loop over all tuples in S
3. Check join conditions
4. Write out

Cost: $p(R) + t(R) \times p(S) + OUT$ or $p(S) + t(S) \times p(R) + OUT$ depending on the outer relation.

Block Nested Loop Join:

1. Load in $B-1$ pages of R
2. For each $(B-1)$-page segment of R, load each page of S
3. Check the join condition
4. Write out

Cost: $p(R) + \frac{p(R)}{B-1} p(S) + OUT$
By loading larger chunks of R, we minimize the number of full disk reads od S. BNLJ is faster.

Index Nested Loop Join: use index to avoid the full cross product

Cost: $p(R) + t(R) \times L + OUT$ where L is the cost of accessing all distinct values in the index (~3)

Sorted Merge Join:

1. Sort R, S on A using external merge sort
2. Scan sorted files and merge

! duplicate join keys
Scan cost: at best $p(R) + p(S)$ reads and at worst $p(R) \times p(S)$ reads.
Cost: $\text{sort}(p(R)) + \text{sort}(p(S)) + p(R) + p(S) + OUT$ where $\text{sort}(N) = 2N\left(\left\lceil \log_B \frac{N}{2(B+1)} \right\rceil + 1\right)$
If already sorted, cost is linear; if $\max\{p(R), p(S)\} < B^2$, cost is $3(p(R) + p(S)) + OUT$

Hash Join:

1. Partition phase: partition R and S into B buckets using a hash function h_B
2. Join tuples in buckets with same hash result

Cost is $2(p(R) + p(S)) + (p(R) + p(S)) + OUT = 3(p(R) + p(S)) + OUT$

Sort to **Union**:

- Sort both relations
- Scan sorted relations and merge them

Hash to **Union**:

- Partition R and S using a hash function
- Build hash table for R using different hash function
- Probe with tuples in S-partition and add tuples to table while ignoring duplicates

Aggregations (w/o grouping): usually require scanning all the table. If index search key includes attributes in SELECT and WHERE, do index-only scan.
Aggregation (with grouping):

- If tree index on all attr. In SELECT, WHERE, GROUPBY: do index-only scan
- If GROUPBY attributes in search key: retrieve in order
- Else: get data entries and use sort/hash aggregate algo, compute aggregate for each group

12. Query Optimization

Recall:
Relation Schema = Relation Name + Attributes + Domains
Relation Instance = set of tuples conforming to the same schema

How does a SQL engine work?
SQL Query → Relational Algebra Plan → Optimized RA Plan → Execution

Operators of Relational Algebra:
→ Basic: Selection $\sigma_c(R)$, Projection $\Pi_{A_1,\ldots,A_n}(R)$, Cartesian Product $R \times S$, Union $R \cup S$, Difference $R - S$.

→ Derived:

- Intersection $R \cap S = R - (R - S)$.
- Natural Join $R \bowtie_c S = \Pi_{A \cup B}\big(\sigma_{C=D}(\rho_{C \to D}(R) \times S)\big)$: a join on equality of shared attributes
- Theta Join $R \bowtie_\theta S = \sigma_\theta(R \times S)$: a join that involves a predicate (any condition θ)
- Equi-Join $R \bowtie_{A=B} S = \sigma_{A=B}(R \times S)$: a join where θ is an equality (most common in practice)
- Semi-Join $\Pi_{A_1,\ldots,A_n}(R \ltimes S)$ where A_i are the attributes of R.

→ Special: Renaming $\rho_{B_1,\ldots,B_n}(R)$

RA Plan Execution

- Join: use memory an I/O cost to pick the best algo (BNLJ, SMJ, HJ)
- Selection: use indexes
- Projection: identify distinct values with hashing or sorting

Optimization

→ Logical: find equivalent plans and choose the one minimizing # of tuples at each step

→ Physical: find algo with lowers I/O cost based on physical params (buffer size) & data size (histograms)

Logical: Always try (if possible) to push down Projections and Selections so they occur as soon as possible.

Physical

1. Estimate the cost for different indexes types

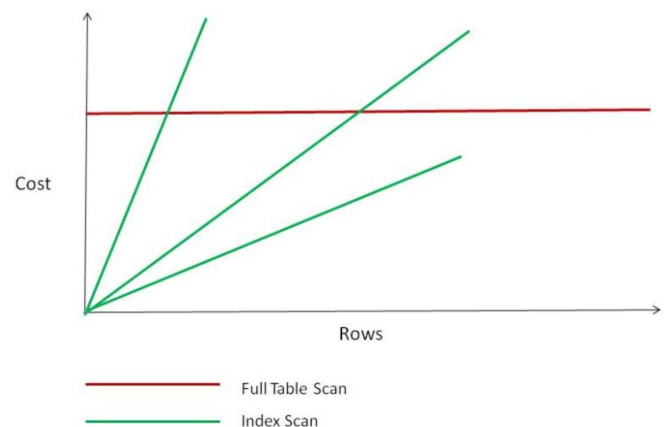
Example: range query for M entries

→ Clustered

- Traverse $h = \log_f 1.5N$
- Scan 1 random I/O + $\left\lceil \frac{M-1}{P} \right\rceil$ sequential I/O

→ Unclustered

- Traverse $h = \log_f 1.5N$
- Scan $\sim M$ random I/O



2. **Histograms** & I/O Cost Estimation

A histogram is a set of value ranges (buckets) and the frequencies of each bucket occurrence.

How to choose the bucket?

- Full: bucket size = 1
- Uniform
- Equi-width: all buckets have the same size
- Equi-depth: all buckets have the same number of items (total frequency)

We must maintain histograms by periodically update them.

Catalogs contain #tuples & #pages for relations; #distinct key values & #pages for indexes; histograms

Enumerate alternative plans and estimate their costs without testing them. We use the catalog information for that.

Estimating query result cardinality
SELECT $\mathrm{attr}_1, \dots, \mathrm{attr}_n$
FROM $\mathrm{R}_1, \dots, \mathrm{R}_k$
WHERE cond_1 AND … AND cond_m

Result Cardinality = $card(\mathrm{R}_1 \times \dots \times \mathrm{R}_k) \times \prod RF(cond_i)$ where RF is reduction factor.

13. Transactions

A **transaction** is a sequence of one ore more operations (read and write) which reflects a single real-world transition (transfer money, purchase products, register for a class)

Motivations

1. Recovery & Durability: make sure that TXNs are either durably stored in full or not at all.
2. Concurrent execution: have the DBMS run several TXNs concurrently, to keep CPU going.

Properties

- **A**tomic: all or nothing (commits or aborts)
- **C**onsistent: tables must always satisfy integrity constraints
- **I**solated: should not be able to observe changes from other TXNs during the run
- **D**urable: the effect of a TXN must persist (committed data must be written to disk)

1. Atomicity and Durability via Logging

The log is a list of modifications that is duplexed and archived on stable disk.
Record information (diff) for every update → ordered list of actions: easy to undo/redo any action.
Why we need that? Partial results of transaction should be written to disk because not enough memory/time to wait for the whole TXN. We need a log to undo these partial results.

Write-Ahead Logging: write log from memory to disk before writing data to disk.

2. Concurrency: parallelizing TXNs without creating conflicts

Interleave transactions to speed up processing (one uses CPU while other uses disk) but this may create conflicts!
When an interleaving schedule (e.g. $T_1(A) \rightarrow T_2(A) \rightarrow T_2(B) \rightarrow T_1(B)$) is different from any serial order (e.g. T_1 then T_2 or T_2 then T_1) we say that this schedule is not serializable.
→ So, a serializable schedule is a schedule that is equivalent (same effect) to some serial execution of the transaction.
Conflict: two actions of different TXNs involve the same variable and at least one of them is write.

3. Conflict Serializability, Locking and Deadlock

Schedule S is conflict serializable if S is conflict equivalent (conflicting actions are in the same order) to some serial schedule.

Theorem: schedule is conflict serializable if and only if its conflict graph is acyclic.

Locking: require each transaction to obtain a lock before accessing a data object to prevent concurrent access.

Also See Deadlock!