A *database* is a large collection of structured data

A *Database Management System (DBMS)* is software that stores, manages and/or facilitates access to databases.

What we want: API contracts, simple efficient queries, efficient scalable bulk processing,

*Time-space Rendezvous* - in the same place (RAM) at the same time

Divide and conquer:

Phase 1 - “streamwise” *divide* into N/(B-2) megachunks, *conquer* each and write to disk

Phase 2 - a streaming algorithm over *conquered megachunks,* the streaming must ensure *rendezvous*

A lot of databases still use magnetic disks - No “pointer derefs”. Instead, an API read/write, but both API calls are expensive

Time to access (read/write) a disk block: seek time (moving arms to position disk head on track) + rotational delay (waiting for block to rotate under head) + transfer time (actually moving data to/from disk surface)

Arranging pages: ‘Next’ block concept: blocks on same track, followed by blocks on same cylinder, followed by blocks on adjacent cylinder. Arrange file pages sequentially on disk to minimize seek and rotational delay. For a sequential scan, pre-fetch several pages at a time!

Double Buffering: Main thread runs *f(x)* on one pair I/O bufs, 2nd “I/O thread” fills/drains unused I/O bufs, Main thread ready for a new buf? Swap!

Sorting and Hashing

Given: A file *F*: containing a multiset of records *R* and consuming **N** blocks of storage, Two “scratch” disks each with >> N blocks of free storage, A fixed amount of space in RAM memory capacity equivalent to **B** blocks of disk

Sorting: Produce an output file *FS* with contents *R* ***stored in order by a given sorting criterion***

Hashing: Produce an output file *FH* with contents *R,* ***arranged on disk so that no 2 records that are* incomparable** (i.e. “equal” in sort order) ***are separated by a greater or smaller record.***

Two way sort: Pass 0 (conquer): read a page, sort it, write it. only one buffer page is used, a repeated “batch job” Pass 1, 2, 3, …, etc. (merge): requires **3 buffer pages,** *merge pairs* of runs into runs twice as long

Each pass we read + write, each page in file. N pages in the file So, the number of passes is: So total cost is: 

General external merge sort: To sort a file with N pages using B buffer pages: Pass 0: use B buffer pages, Produce sorted runs of B pages each. Pass 1, 2, …, etc.: merge B-1 runs.

Number of passes: Cost = 2N \* (# of passes)

How big of a table can we sort in two passes? Each “sorted run” after Phase 0 is of size B, Can merge up to B-1 sorted runs in Phase 1. Answer: B(B-1)

Tournament sort: 2 heaps, removemin, if read input < min put in other heap. **Average run length = 2(B-2)**

Hash: Streaming Partition (divide): Use a hash function *hp* to stream records to disk partitions, All matches rendezvous in the same partition. ReHash (conquer): Read partitions into RAM hash table one at a time, using different hash *hr* ,Then go through each bucket of this hash table to achieve rendezvous in RAM. **4\*N**

How big of a table can we hash in two passes? B-1 “partitions” result from Pass 1, Each should be no more than B pages in size. Answer: B(B-1). We can hash a table of size N pages in about sqrt(N) space. Note: assumes hash function distributes records evenly! Have a bigger table? Recursive partitioning!

Sorting pros: Great if input already sorted (or almost sorted) w/heapsort, Great if need output to be sorted anyway, Not sensitive to “data skew” or “bad” hash functions

Hashing pros: For duplicate elimination, scales with # of values, Not # of items! We’ll see this again. Can simply conquer sometimes! (Think about that)

SELECT [DISTINCT] *<column expression list>* FROM *<single table>* [WHERE *<predicate>*] [GROUP BY *<column list>* [HAVING *<predicate>*] ] [ORDER BY *<column list>*]

From->Where->Select->Group By->Having->Distinct

Sailors as s-> needed w/ ambiguity. ‘\_’ stands for any one character and ‘%’ stands for 0 or more arbitrary characters.

From t1 join t2 on \_\_\_\_

Natural/inner: equi-join for each pair of attributes with the same name

Left outer: Returns all matched rows, plus all unmatched rows from the table on the left of the join clause

(use nulls in fields of non-matching tuples)

Right Outer Join returns all matched rows, plus all unmatched rows from the table on the right of the join clause

Full Outer Join returns all (matched or unmatched) rows from the tables on both sides of the join clause

Named queries: CREATE VIEW *view\_name* AS *select\_statement*

GRANT *privileges* ON *object* TO *users* [WITH GRANT OPTION]

Integrity constraints: conditions that every legal instance of a relation must satisfy. Types of IC’s: Domain constraints, primary key constraints, foreign key constraints, general constraints.

Keys are a way to associate tuples in different relations

A set of fields is a superkey if: No two distinct tuples can have same values in all key fields

A set of fields is a key for a relation if it is *minimal*: It is a superkey, No subset of the fields is a superkey

what if >1 key for a relation? One of the keys is chosen (by DBA) to be the primary key. Other keys are called candidate keys.

*Foreign key*: a “logical pointer” Ex. FOREIGN KEY (sid) REFERENCES Students

Theta join: R ⋈θ S: all pairs {r,s} where θ(r,s)

A common case: EquiJoin i.e., θ is an equality test

Cost Notation

[R] : the number of pages to store R

pR: number of records per page of R

|R| : the number of records in R (*cardinality)*

Note: pR*\**[R] = |R|

Simple nested loops join:

foreach record r in R do

foreach record s in S do

if θ(ri, sj) then add <r, s> to result

Cost = (pR\*[R])\*[S] + [R] =

Page-Oriented NestLoop Join:

foreach page bR in R do

foreach page bS in S do

foreach record r in bR do

foreach record s in bSdo

if θ(ri,sj) then add <r, s> to result

Cost = [R]\*[S] + [R]

Chunk nested loops join: use chunks to exploit buffers

cost = [outer] + (outer-chunks \* [inner]), #outer chunks **=** 

Index nested loops join: need index on S

foreach tuple r in R do

foreach tuple s in S where ri == sj do

add <r, s> to result

Cost = [R] + ([R]\*pR)

Sort-merge join:

* 1. Sort R on join attr(s)
  2. Sort S on join attr(s)
  3. Scan sorted-R and sorted-S in tandem, to find matches

Cost: Sort R + Sort S + ([R]+[S]). But in worst case, last term could be [R]\*[S]

Optimized: Do the join during the final merging pass of sort! Cost = 3\*[R] + 3\*[S]

Hash join: build hash table, then stream S and probe

* Partitioning phase: read+write both relations ⇒ 2([R]+[S]) I/Os
* Matching phase: read both relations, write output ⇒ [R]+[S] + [output] I/Os
* Total cost of 2-pass hash join = 3([R]+[S])+[output]

Block = Page, Relation = Table, Tuple = Row = Record, Attribute = Column = Field

Unordered (Heap) Files

Collection of records in no particular order. As file shrinks/grows, disk pages (de)allocated. To support record level operations, we must: keep track of the *pages* in a file, keep track of *free space* on pages keep track of the *records* on a page

A Heap file allows us to retrieve records: by specifying the *rid,* or by scanning all records sequentially. Nice to fetch records *by value*, Indexes: file structures for efficient value-based queries

Fixed length record: Field types same for all records in a file

Variable length: Fields Delimited by Special Symbols or Array of Field Offsets

*Record id = <page id, slot #>*

In bitmap, moving records for free space management *changes rid*

Slotted page: Can move records on page without changing rid!

When a Page is Requested, Buffer pool information table contains: <frame#, pageid, pin\_count, dirty>. If requested page is not in pool: Choose a frame for *replacement. Only “un-pinned” pages are candidates!* If frame “dirty”, write current page to disk and Read requested page into frame. *Pin* the page and return its address.

Requestor of page must eventually: *unpin* it, indicate whether page was modified via *dirty* bit.

*Index* : disk-based data structure for fast lookup by value

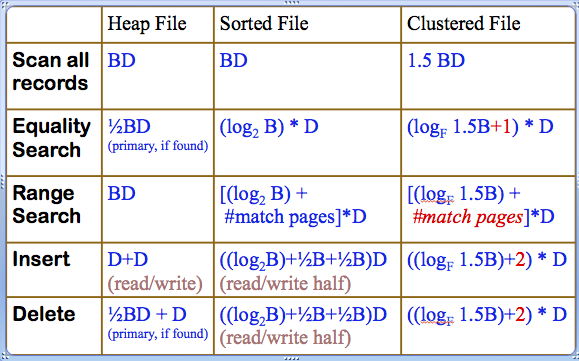
*Search key:*  any subset of columns in the relation.

*Search key* need not be a *key* of the relation

I.e. There can be multiple items matching a search key

Index contains a collection of *data entries*

Clustered Pros: Efficient for range searches, Potential locality benefits, Support certain types of compression. Clustered Cons: More expensive to maintain



ISAM: Index file, *Index entries:* <search key value, page id>. static

B+: Insert/delete at log F N cost; but keep tree *height-balanced*. F = fanout, N = # leaf pages, dynamic

Tree is not width-balanced: varying fanouts per node. BUT: minimum 50% node occupancy (except for root). Each node contains *m entries where*  d <= *m* <= 2d entries. “d” is called the *order* of the tree.