DBMS Internals
How does it all work?

May 3rd, 2004

Agenda

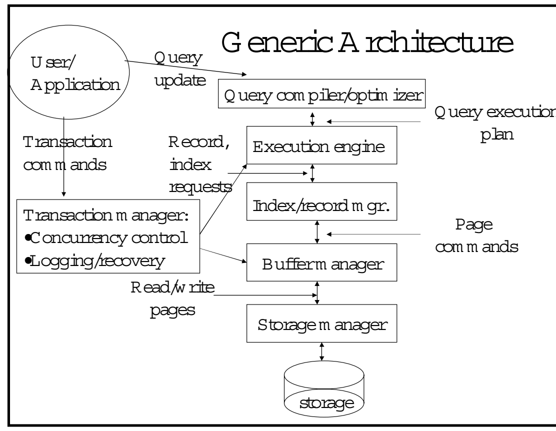
- Comments on phase 2 of the project
- HW 3 is out.
- Today: DBMS internals part 1 –
  - Indexing
  - Query execution
- Next week: query optimization.

What Should a DBMS Do?

- Store large amounts of data
- Process queries efficiently
- Allow multiple users to access the database concurrently and safely.
- Provide durability of the data.

- How will we do all this??

Generic Architecture



Main Points to Take Away

- I/O model of computation
  - We only count accesses to disk.
- Indexing:
  - Basic techniques: B+-tree, hash indexes
  - Secondary indexes.
- Efficient operator implementations: join
- Optimization: from what to how.

The Memory Hierarchy

Main Memory

- Volatile
- limited address spaces
- expensive
- average access time:
  10-100 nanoseconds

Cache:
access time 10 nano's

Disk

- 5-10 MB/S transmission rates
- Gigs of storage
- average time to access a block: 10-15 msecs.
- Need to consider seek, rotation, transfer times.
- Keep records "close" to each other.

Tape

- 1.5 MB/S transfer rate
- 280 GB typical capacity
- Only sequential access
- Not for operational data

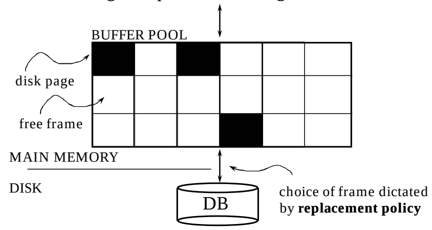
Main Memory

- Fastest, most expensive
- Today: 512MB-2GB are common on PCs
- Many databases could fit in memory
  - New industry trend: Main Memory Database
  - E.g TimesTen
- Main issue is volatility

Secondary Storage

- Disks
- Slower, cheaper than main memory
- Persistent !!!
- Used with a main memory buffer

Buffer Management in a DBMS



- Data must be in RAM for DBMS to operate on it!
- Table of <frame#, pageid> pairs is maintained.
- LRU is not always good.

Buffer Manager

Manages buffer pool: the pool provides space for a limited number of pages from disk.

Needs to decide on page replacement policy.

Enables the higher levels of the DBMS to assume that the needed data is in main memory.

Why not use the Operating System for the task??

- DBMS may be able to anticipate access patterns
- Hence, may also be able to perform prefetching
- DBMS needs the ability to force pages to disk.

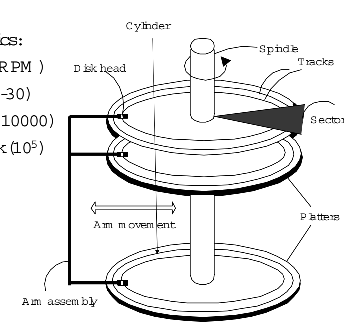
Tertiary Storage

- Tapes or optical disks
- Extremely slow: used for long term archiving only

The Mechanics of Disk

Mechanical characteristics:

- Rotation speed (5400RPM)
- Number of platters (1-30)
- Number of tracks (<=10000)
- Number of bytes/track (10^5)



Disk Access Characteristics

- Disk latency = time between when command is issued and when data is in memory
- Is not following Moore's Law!

- Disk latency = seek time + rotational latency
  - Seek time = time for the head to reach cylinder
    - 10ms – 40ms
  - Rotational latency = time for the sector to rotate
    - Rotation time = 10ms
    - Average latency = 10ms/2
- Transfer time = typically 10MB/s
- Disks read/write one block at a time (typically 4kB)

The I/O Model of Computation

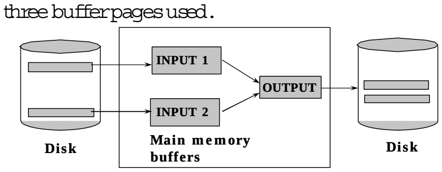
- In main memory algorithms we care about CPU time
- In databases time is dominated by I/O cost
- Assumption: cost is given only by I/O
- Consequence: need to redesign certain algorithms
- Will illustrate here with sorting

Sorting

- Illustrates the difference in algorithm design when your data is not in main memory:
  - Problem: sort 1Gb of data with 1Mb of RAM.
- Arises in many places in database systems:
  - Data requested in sorted order (ORDER BY)
  - Needed for grouping operations
  - First step in sort-merge join algorithm
  - Duplicate removal
  - Bulk loading of B+-tree indexes.

2-Way Merge-sort: Requires 3 Buffers

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, …, etc.:
  - three buffer pages used.



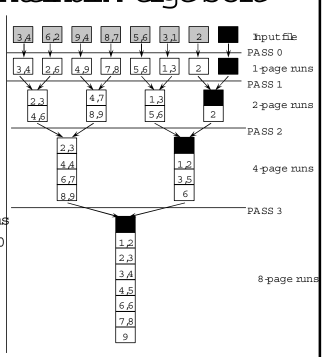
Two-Way External Merge Sort

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

$$= \lceil \log_2 N \rceil + 1$$

- Improvement: start with larger runs
- Sort 1GB with 1MB memory in 10 passes

$$2N \left(\lceil \log_2 N \rceil + 1 \right)$$



Can We Do Better?

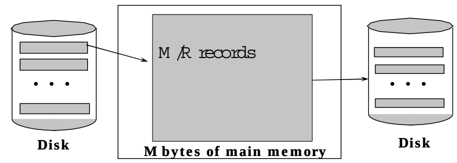
- We have more main memory
- Should use it to improve performance

Cost Model for Our Analysis

- B : Block size
- M : Size of main memory
- N : Number of records in the file
- R : Size of one record

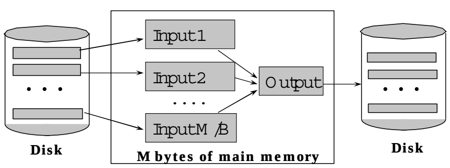
External Merge-Sort

- Phase one: load M bytes in memory, sort
  - Result: runs of length M/R records



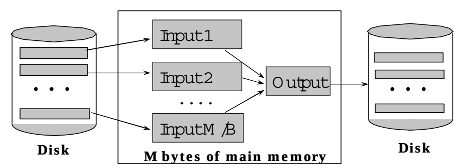
Phase Two

- Merge M/B – 1 runs into a new run
- Result: runs have now M/R (M/B – 1) records



Phase Three

- Merge M/B – 1 runs into a new run
- Result: runs have now M/R $(M/B - 1)^2$ records



Cost of External Merge Sort

- Number of passes: $1+\lceil \log_{M/B-1} \lceil NR/M \rceil \rceil$
- Think differently
  - Given B = 4KB, M = 64MB, R = 0.1KB
  - Pass 1: runs of length M/R = 640000
    - Have now sorted runs of 640000 records
  - Pass 2: runs increase by a factor of M/B – 1 = 16000
    - Have now sorted runs of 10,240,000,000 = 10^{10} records
  - Pass 3: runs increase by a factor of M/B – 1 = 16000
    - Have now sorted runs of 10^{14} records
    - Nobody has so much data !
- Can sort everything in 2 or 3 passes !

Number of Passes of External Sort

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

B: number of frames in the buffer pool; N: number of pages in relation.

Data Storage and Indexing

Representing Data Elements

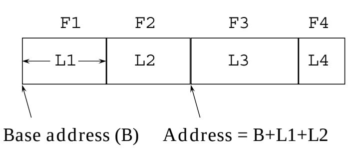
- Relational database elements:

```
CREATE TABLE Product (

   pid INT PRIMARY KEY,
   name CHAR(20),
   description VARCHAR(200),
   maker CHAR(10) REFERENCES Company(name)
)
```

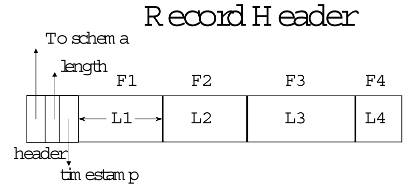
- A tuple is represented as a record

Record Formats: Fixed Length



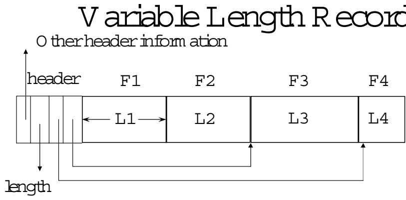
- Information about field types same for all records in a file; stored in system catalogs.
- Finding i'th field requires scan of record.
- Note the importance of schema information!

Record Header



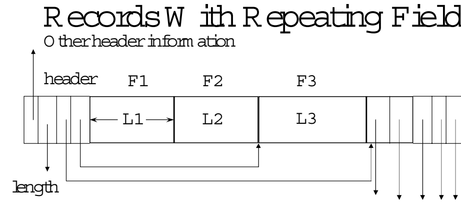
Need the header because:
- The schema may change
  for a while new+old may coexist
- Records from different relations may coexist

Variable Length Records



Place the fixed fields first: F1, F2
Then the variable length fields: F3, F4
Null values take 2 bytes only
Sometimes they take 0 bytes (when at the end)

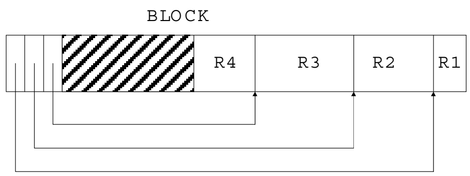
Records With Repeating Fields



Needed e.g. in Object Relational systems,
or fancy representations of many-many relationships

Storing Records in Blocks

- Blocks have fixed size (typically 4k)



Storage and Indexing

- How do we store efficiently large amounts of data?
- The appropriate storage depends on what kind of accesses we expect to have to the data.
- We consider:
  - primary storage of the data
  - additional indexes (very very important).

Cost Model for Our Analysis

* As a good approximation, we ignore CPU costs:
  - B: The number of data pages
  - R: Number of records per page
  - D: (Average) time to read or write disk page
  - Measuring number of page I/O's ignores gains of pre-fetching blocks of pages; thus, even I/O cost is only approximated.
  - Average-case analysis; based on several simplistic assumptions.

File Organizations and Assumptions

- Heap Files:
  - Equality selection on key; exactly one match.
  - Insert always at end of file.
- Sorted Files:
  - Files compacted after deletions.
  - Selections on sort field(s).
- Hashed Files:
  - No overflow buckets, 80% page occupancy.
- Single record insert and delete.

Cost of Operations

	Heap File	Sorted File	Hashed File
Scan all recs			
Equality Search			
Range Search			
Insert			
Delete			

Indexes

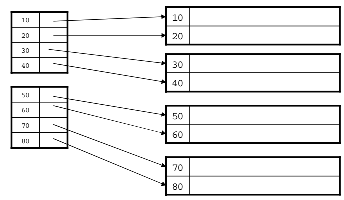
- An <u>index</u> on a file speeds up selections on the search key fields for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - Search key is not the same as key (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of data entries, and supports efficient retrieval of all data entries with a given key value k.

Index Classification

- Primary/secondary
- Clustered/unclustered
- Dense/sparse
- B+ tree / Hash table / …

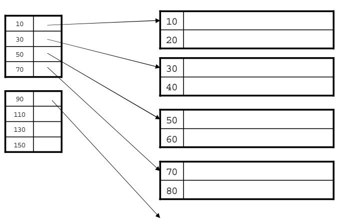
Primary Index

- File is sorted on the index attribute
- Dense index: sequence of (key,pointer) pairs



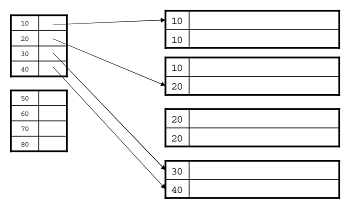
Primary Index

- Sparse index



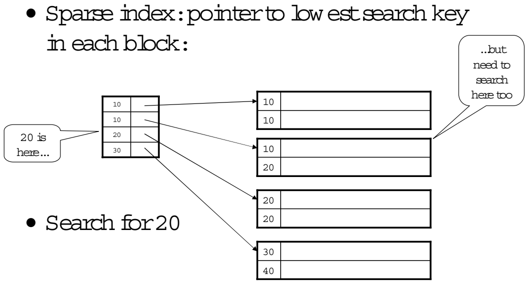
Primary Index with Duplicate Keys

- Dense index:



Primary Index with Duplicate Keys

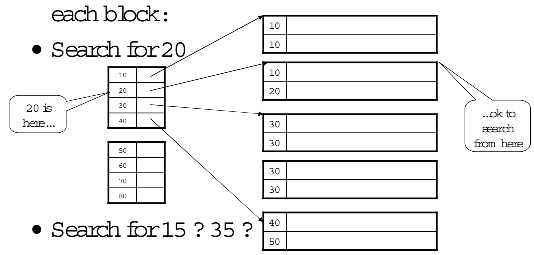
- Sparse index: pointer to lowest search key in each block:



- Search for 20

Primary Index with Duplicate Keys

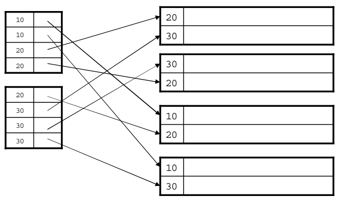
- Better: pointer to <u>lowest new search key</u> in each block:
- Search for 20



- Search for 15 ? 35 ?

Secondary Indexes

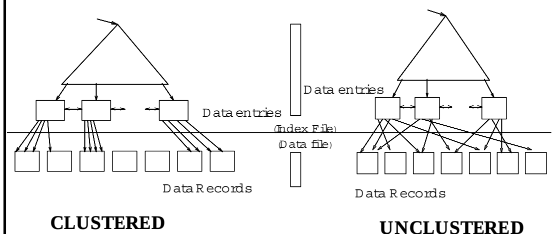
- To index other attributes than primary key
- Always dense (why?)



Clustered/Unclustered

- Primary indexes = usually clustered
- Secondary indexes = usually unclustered

Clustered vs. Unclustered Index



Secondary Indexes

- Applications:
  - index other attributes than primary key
  - index unsorted files (heap files)
  - index clustered data

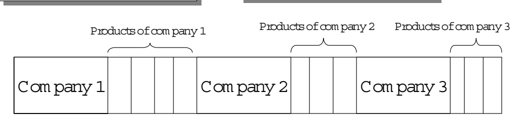
Applications of Secondary Indexes

- Clustered data

  Company(name, city), Product(pid, maker)

```
Select city
From Company, Product
Where name=maker
   and pid="p045"
```

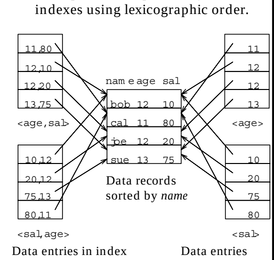
```
Select pid
From Company, Product
Where name=maker
   and city="Seattle"
```



Composite Search Keys

- Composite Search Keys: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal=75
  - Range query: Some field value is not a constant. E.g.:
    - age =20; or age=20 and sal > 10

Examples of composite key indexes using lexicographic order.

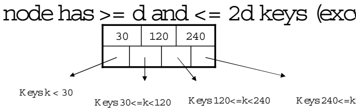


B+ Trees

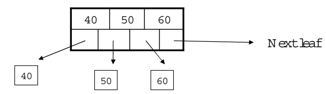
- Search trees
- Idea in B Trees:
  - make 1 node = 1 block
- Idea in B+ Trees:
  - Make leaves into a linked list (range queries are easier)

B+ Trees Basics

- Parameter d = the <u>degree</u>
- Each node has >= d and <= 2d keys (except root)

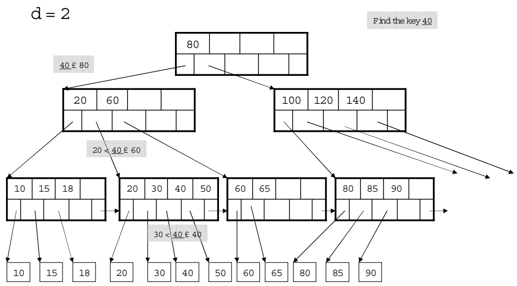


- Each leaf has >=d and <= 2d keys:



B+ Tree Example

d = 2



B+ Tree Design

- How large d ?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 byes
- 2d x 4 + (2d+1) x 8 <= 4096
- d = 170

Searching a B+ Tree

- Exact key values:
  - Start at the root
  - Proceed down, to the leaf

```
Select name
From people
Where age = 25
```

- Range queries:
  - As above
  - Then sequential traversal

```
Select name
From people
Where 20 <= age
  and age <= 30
```

B+ Trees in Practice

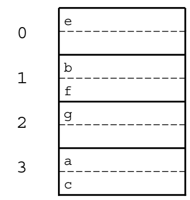
- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Hash Tables

- Secondary storage hash tables are much like main memory ones
- Recall basics:
  - There are n <u>buckets</u>
  - A hash function f(k) maps a key k to {0,1,… ,n-1}
  - Store in bucket f(k) a pointer to record with key k
- Secondary storage: bucket = block, use overflow blocks when needed

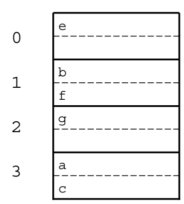
Hash Table Example

- Assume 1 bucket (block) stores 2 keys + pointers
- h(e)=0
- h(b)=h(f)=1
- h(g)=2
- h(a)=h(c)=3



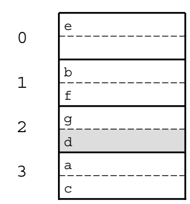
Searching in a Hash Table

- Search for a:
- Compute h(a)=3
- Read bucket 3
- 1 disk access



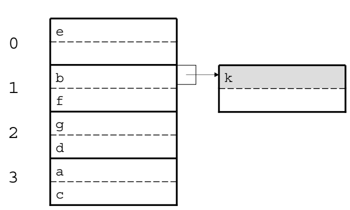
Insertion in Hash Table

- Place in right bucket, if space
- E.g. h(d)=2



Insertion in Hash Table

- Create overflow block, if no space
- E.g. h(k)=1



- More over-flow blocks may be needed

Hash Table Performance

- Excellent, if no overflow blocks
- Degrades considerably when number of keys exceeds the number of buckets (I.e. many overflow blocks).
- Typically, we assume that a hash-lookup takes 1.2 I/Os.

Where are we?

- File organizations: sorted, hashed, heaps.
- Indexes: hash index, B+-tree
- Indexes can be clustered or not.
- Data can be stored in the index or not.

- Hence, when we access a relation, we can either scan or go through an index:
  - Called an access path.

Current Issues in Indexing

- Multi-dimensional indexing:
  - how do we index regions in space?
  - Document collections?
  - Multi-dimensional sales data
  - How do we support nearest neighbor queries?

- Indexing is still a hot and unsolved problem!

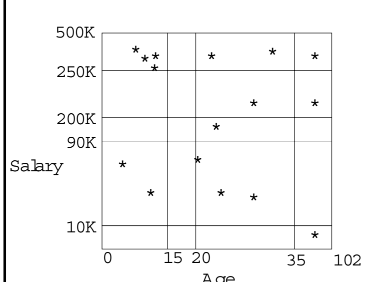
Multidimensional Indexes

- Applications: geographical databases, data cubes.
- Types of queries:
  - partial match (give only a subset of the dimensions)
  - range queries
  - nearest neighbor
  - Where am I? (DB or not DB?)
- Conventional indexes don't work well here.

Indexing Techniques

- Hash like structures:
  - Grid files
  - Partitioned indexing functions
- Tree like structures:
  - Multiple key indexes
  - kd-trees
  - Quad trees
  - R-trees

Grid Files

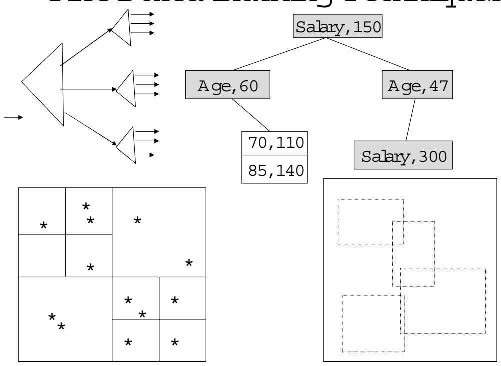


- Each region in the corresponds to a bucket.
- Works well even if we only have partial matches
- Some buckets may be empty.
- Reorganization requires moving grid lines.
- Number of buckets grows exponentially with the dimensions.

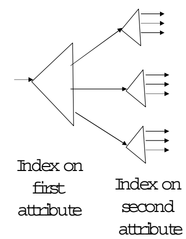
Partitioned Hash Functions

- A hash function produces k bits identifying the bucket.
- The bits are partitioned among the different attributes.
- Example:
  - Age produces the first 3 bits of the bucket number.
  - Salary produces the last 3 bits.
- Supports partial matches, but is useless for range queries.

Tree Based Indexing Techniques



Multiple Key Indexes

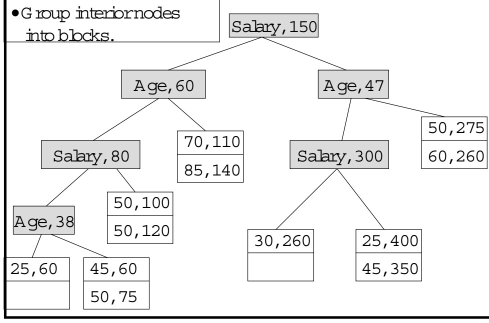


- Each level as an index for one of the attributes.
- Works well for partial matches if the match includes the first attributes.

KD Trees

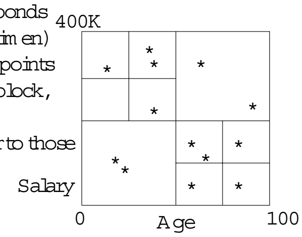
Adaptation to secondary storage:

- Allow multiway branches at the nodes, or
- Group interior nodes into blocks.



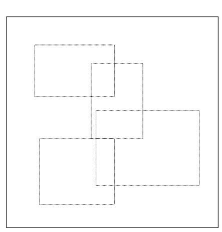
Quad Trees

- Each interior node corresponds to a square region (or k-dimen)
- When there are too many points in the region to fit into a block, split it in 4.
- Access algorithms similar to those of KD-trees.

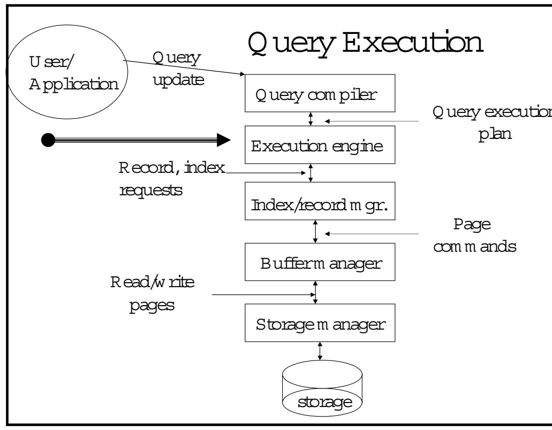


R-Trees

- Interior nodes contain sets of regions.
- Regions can overlap and not cover all parent's region.
- Typical query:
  - Where am I?
- Can be used to store regions as well as data points.
- Inserting a new region may involve extending one of the existing regions (minimally).
- Splitting leaves is also tricky.

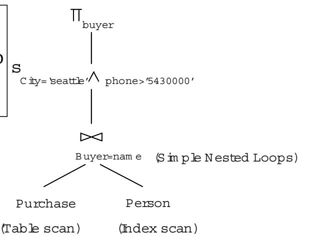


Query Execution



Query Execution Plans

```
SELECT  S.sname
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
      Q.city='seattle' AND
      Q.phone > '5430000'
```



Query Plan:
- logical tree
- implementation choice at every node
- scheduling of operations.

Some operators are from relational algebra, and others (e.g., scan, group) are not.

The Leaves of the Plan: Scans

- Table scan: iterate through the records of the relation.
- Index scan: go to the index, from there get the records in the file (when would this be better?)
- Sorted scan: produce the relation in order. Implementation depends on relation size.

How do we combine Operations?

- The iterator model. Each operation is implemented by 3 functions:
  - Open: sets up the data structures and performs initializations
  - GetNext: returns the the next tuple of the result.
  - Close: ends the operations. Cleans up the data structures.
- Enables pipelining!
- Contrast with data-driven materialize model.
- Sometimes it's the same (e.g., sorted scan).

Implementing 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 Tuples in reln. 1 and in reln. 2.
  - Aggregation (SUM, MIN, etc.) and GROUP BY

Schema for Examples

Purchase (*buyer*:string, *seller*: string, *product*: integer),

Person (*name*:string, *city*:string, *phone*: integer)

- Purchase:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages (i.e., 100,000 tuples, 4MB for the entire relation).
- Person:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages (i.e, 40,000 tuples, 2MB for the entire relation).

Simple Selections

```
SELECT  *
FROM Person R
WHERE   R.phone < '543%'
```

- Of the form $\sigma_{R.attr\ op\ value}(R)$
- With no index, unsorted: Must essentially scan the whole relation; cost is M (#pages in R).
- With an index on selection attribute: Use index to find qualifying data entries, then retrieve corresponding data records. (Hash index useful only for equality selections.)
- Result size estimation:

  (Size of R) * reduction factor.

  More on this later.

Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records.
  - In example, assuming uniform distribution of phones, about 54% of tuples qualify (500 pages, 50000 tuples). With a clustered index, cost is little more than 500 I/Os; if unclustered, up to 50000 I/Os!
- Important refinement for unclustered indexes:
  1. Find sort the rid's of the qualifying data entries.
  2. Fetch rids in order. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).

Two Approaches to General Selections

- <u>First approach</u>: Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don't match the index:
  - Most selective access path: An index or file scan that we estimate will require the fewest page I/Os.
  - Consider city="seattle AND phone<"543%" :
    - A hash index on city can be used; then, phone<"543%" must be checked for each retrieved tuple.
    - Similarly, a b-tree index on phone could be used; city="seattle" must then be checked.

Intersection of Rids

- <u>Second approach</u>
  - Get sets of rids of data records using each matching index.
  - Then intersect these sets of rids.
  - Retrieve the records and apply any remaining terms.

Implementing Projection

```
SELECT DISTINCT
       R.name,
       R.phone
FROM Person R
```

- Two parts:
  (1) remove unwanted attributes,
  (2) remove duplicates from the result.
- Refinements to duplicate removal:
  - If an index on a relation contains all wanted attributes, then we can do an index-only scan.
  - If the index contains a subset of the wanted attributes, you can remove duplicates locally.

Equality Joins With One Join Column

JOIN

```
SELECT *
FROM Person R, Purchase S
WHERE R.name=S.buyer
```

- $R \bowtie S$ is a common operation. The cross product is too large. Hence, performing $R \times S$ and then a selection is too inefficient.
- Assume: M pages in R, p_R tuples per page, N pages in S, p_S tuples per page.
  - In our examples, R is Person and S is Purchase.
- Cost metric: # of I/Os. We will ignore output costs.

Discussion

- How would you implement join?

Simple Nested Loops Join

For each tuple r in R do
    for each tuple s in S do
        if $r_i == s_j$ then add <r, s> to result

- For each tuple in the outer relation R, we scan the entire inner relation S.
  - Cost: M + (p_R * M) * N = 1000 + 100*1000*500 I/Os: 140 hours!
- Page-oriented Nested Loops join: For each page of R, get each page of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - Cost: M + M*N = 1000 + 1000*500 (1.4 hours)

Index Nested Loops Join

foreach tuple r in R do
    foreach tuple s in S where $r_i == s_j$ do
        add <r, s> to result

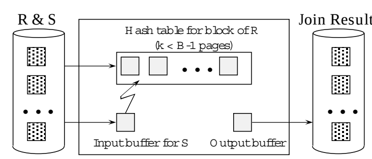
- If there is an index on the join column of one relation (say S), can make it the inner.
  - Cost: M + ((M*p_R) * cost of finding matching S tuples)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: up to 1 I/O per matching S tuple.

Examples of Index Nested Loops

- Hash-index on name of Person (as inner):
  - Scan Purchase: 1000 page I/Os, 100*1000 tuples.
  - For each Person tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Person tuple. Total: 220,000 I/Os. (36 minutes)
- Hash-index on buyer of Purchase (as inner):
  - Scan Person: 500 page I/Os, 80*500 tuples.
  - For each Person tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Purchase tuples. Assuming uniform distribution, 2.5 purchases per buyer (100,000 /40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on clustering.

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



Sort-Merge Join ($R \bowtie_{i=j} S$)

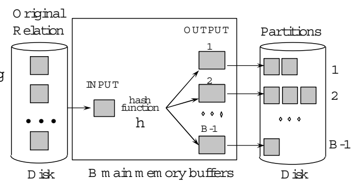
- Sort R and S on the join column, then scan them to do a ``merge'' on the join column.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value and all S tuples with same value <u>match</u>; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.

Cost of Sort-Merge Join

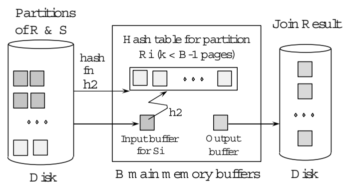
- R is scanned once; each S group is scanned once per matching R tuple.
- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N (unlikely!)
- With 35, 100 or 300 buffer pages, both Person and Purchase can be sorted in 2 passes; total: 7500. (75 seconds).

Hash-Join

- Partition both relations using hash fn h: R tuples in partition i will only match S tuples in partition i.



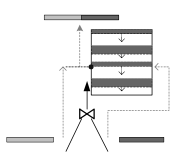
- Read in a partition of R, hash it using **h2 (<> h!)**. Scan matching partition of S, search for matches.



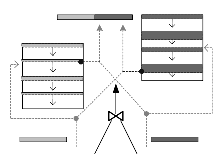
Cost of Hash-Join

- In partitioning phase, read+write both relations; 2(M+N). In matching phase, read both relations; M+N I/Os.
- In our running example, this is a total of 4500 I/Os. (45 seconds!)
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory both have a cost of 3(M+N) I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

Double Pipelined Join (Tukwila)



Hash Join
- 8 Partially pipelined: no output until inner read
- 8 Asymmetric (inner vs. outer) – optimization requires source behavior knowledge



Double Pipelined Hash Join
- 4 Outputs data immediately
- 4 Symmetric – requires less source knowledge to optimize

Discussion

- How would you build a query optimizer?