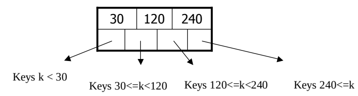
Introduction to Database Systems

CSE 444

Lecture #14
Feb 26 2001

Review: B+ Tree Node Structure



2

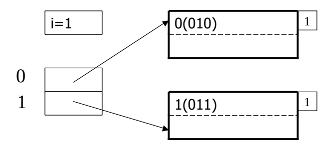
B+ Tree and Indexes

- ⌘Index on composite (concatenated) key: (last name, first name)
  - ⌧What's the impact of ordering?
- ⌘Index AND-ing or OR-ing
  - ⌧Age between [40, 50] and Salary between [100,200]
  - ⌧Obtain the pointers (record identifiers) to data file for each qualifying leaf node
  - ⌧Sort and intersect (union)

3

Extensible Hash Table

- ⌘E.g. i=1, n=2, k=4

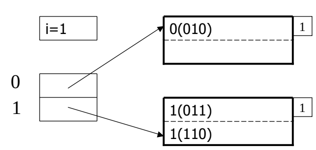


- ⌘Note: we only look at the first bit (0 or 1)

4

Insertion in Extensible Hash Table

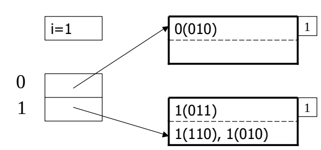
- ⌘Insert 1110



5

Insertion in Extensible Hash Table

- ⌘Now insert 1010

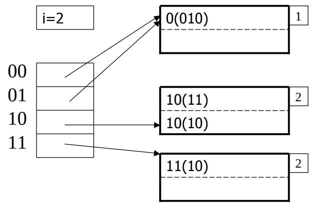


- ⌘Need to extend table, split blocks
- ⌘i becomes 2

6

Insertion in Extensible Hash Table

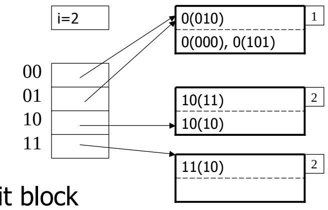
⌘Now insert 1110



7

Insertion in Extensible Hash Table

⌘Now insert 0000, then 0101

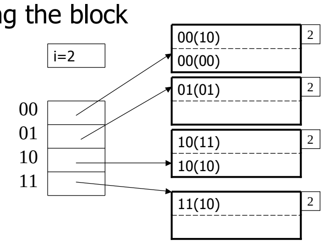


⌘Need to split block

8

Insertion in Extensible Hash Table

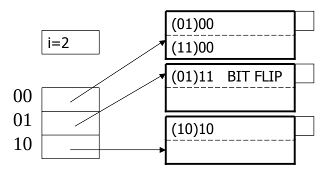
⌘After splitting the block



9

Linear Hash Table Example

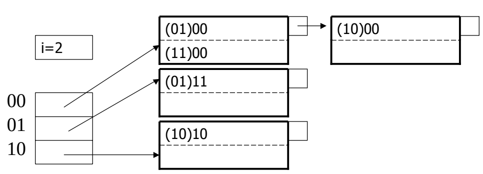
⌘N=3



10

Linear Hash Table Example

⌘Insert 1000: overflow blocks…



11

Linear Hash Tables

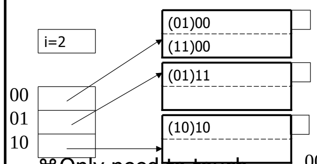
⌘Key parameters
- I #of discriminating bits, N #of buckets, R # of records
- *Capacity Threshold* = R/N

⌘Extension:
- when capacity threshold exceeds (say) 80%
- independent on overflow blocks

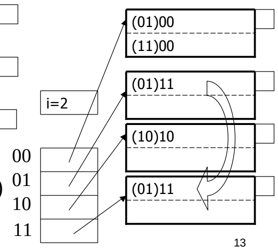
12

Linear Hash Table Extension

⌘From n=3 to n=4



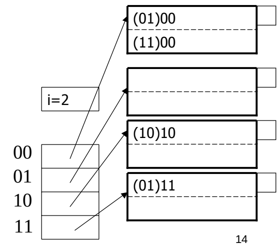
⌘Only need to touch one block (which one ?)



Linear Hash Table Extension

⌘From n=3 to n=4 finished

⌘Extension from n=4 to n=5 (new bit)



BitMap Indexes (Reading: 5.4.1-5.4.3)

⌘Bit Vector for every distinct value in the column

⌘As many bits as there are records in the data

⌘R1:25, R2:50 R3:25 R4: 50 R5: 50 R6: 70 R7:70 R8:25

⌘25: 10100001; 50: 01011000 70: 00000110

⌘Easy Index OR-ing (score = 25 or score = 50)

⌘Easy Index AND-ing (last score = new score)

Compressed BitMaps: Run Length Encoding

⌘Represent sequence of I 0-s followed by 1 as a binary encoding of I

⌘Concatenate codes for each run together
- ⍓But, must be able to recover runs

⌘Scheme
- ⍓B_I = #of bits in binary encoding of I
- ⍓Represent as B_I – 1 1-s followed by 0 and then binary encoding of I

Example

⌘13 0-s followed by 1. 4 bits to represent 13. Hence represent as (11101101)

⌘Decode: (11101101001011)

⌘Run-Length: (13,0,3)

⌘00000000000001100001

⌘Note: Trailing 0-s not recovered

Index AND-ing and OR-ing

⌘Decode and then do Index AND and OR

⌘Can do stepwise
- ⍓Decode one run at a time
- ⍓Read Example 5.26

Query Execution

Required Reading: 2.3.3-2.3.5, 6.1- 6.7
Suggested Reading: 6.8, 6.9

An Algebra for Queries

- Logical operators
  - *what* they do
- Physical operators
  - *how* they do it

20

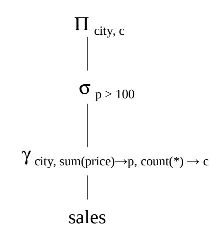
Logical Operators in the Algebra

- Union, intersection, difference
- Selection σ
- Projection Π
- Join ⋈
- Duplicate elimination δ
- Grouping γ
- Sorting τ

21

Example

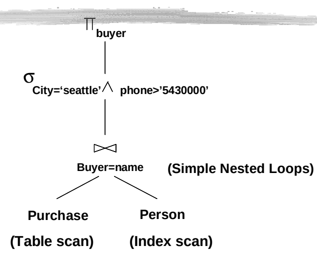
Select city, count(*)
From sales
Group by city
Having sum(price) > 100



22

Physical Operators

```
SELECT S.buyer
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.

23

Scanning Tables

- The table is *clustered* (i.e. blocks consists only of records from this table):
  - Table-scan: if we know where the blocks are
  - Index scan: if we have a sparse index to find the blocks
- The table is unclustered (e.g. its records are placed on blocks with other tables)
  - May need one read for each record

24

Sorting While Scanning

- Sometimes it is useful to have the output sorted
- Three ways to scan it sorted:
  - If there is a primary or secondary index on it, use it during scan
  - If it fits in memory, sort there
  - If not, use multiway merging

25

Estimating the Cost of Operators

- Very important for the optimizer (next week)
- Parameters for a relation R
  - B(R) = number of blocks holding R
    - Meaningful if R is clustered
  - T(R) = number of tuples in R
    - E.g. may need when R is unclustered
  - V(R,a) = number of distinct values of the attribute a

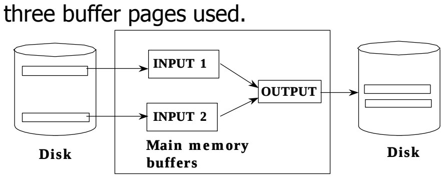
26

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-Phase Merge-sort: Requires 3 Buffers

- Phase 1: Read a page, sort it, write it.
  - only one buffer page is used
- Phase 2: Merge all sorted sublists
  - three buffer pages used.

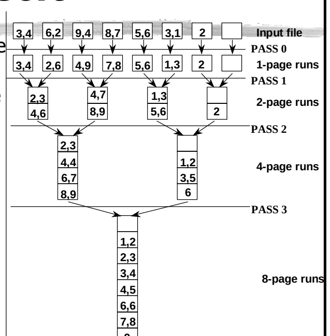


2-Way Merge Sort

- Each pass we read + write each page in file.
- N pages in the file => the number of passes $= \lceil \log_2 N \rceil + 1$
- So total cost is:

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

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



Can We Do Better ?

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

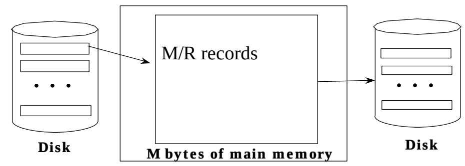
30

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

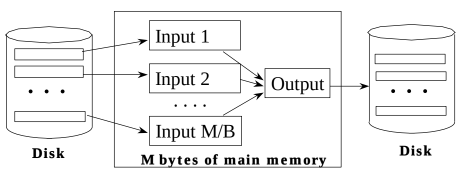
- Phase one: load M bytes in memory, sort
  - Result: runs of length M/R records



32

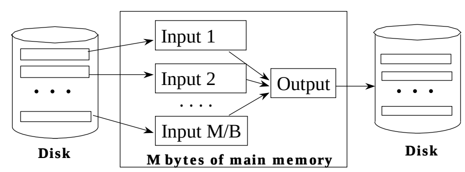
Phase Two

- 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 !

Cost of the Scan Operator

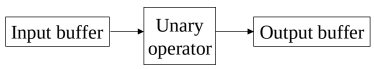
- Clustered relation:
  - Table scan: B(R); to sort: 3B(R)
  - Index scan: B(R); to sort: B(R) or 3B(R)
- Unclustered relation
  - T(R); to sort: T(R) + 2B(R)

36

One-Pass Algorithms

Selection $\sigma(R)$, projection $\Pi(R)$

- Both are *tuple-at-a-Time* algorithms
- Cost: B(R)



37

One-pass Algorithms

Duplicate elimination $\delta(R)$

- Need to keep tuples in memory
- When new tuple arrives, need to compare it with previously seen tuples
- Balanced search tree, or hash table
- Cost: B(R)
- Assumption: $B(\delta(R)) <= M$

38

One-pass Algorithms

Grouping: $\gamma_{city,\ sum(price)}$ (R)

- Need to store all cities in memory
- Also store the sum(price) for each city
- Balanced search tree or hash table
- Cost: B(R)
- Assumption: number of cities fits in memory

39

One-pass Algorithms

Binary operations: R ∩ S, R U S, R – S

- Assumption: min(B(R), B(S)) <= M
- Scan one table first, then the next, eliminate duplicates
- Cost: B(R)+B(S)

40

Nested Loop Joins

- Tuple-based nested loop $R \bowtie S$

For each tuple r in R do
  For each tuple s in S do
    if r and s join then output (r,s)

- Cost: T(R) T(S), sometimes T(R) B(S)

41

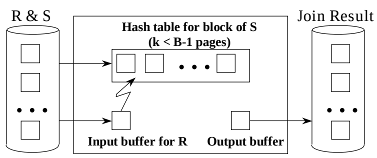
Nested Loop Joins

- Block-based Nested Loop Join

For each (M-1) blocks bs of S do
  for each block br of R do
    for each tuple s in bs
      for each tuple r in br do
        if r and s join then output(r,s)

42

Nested Loop Joins



Nested Loop Joins

- Block-based Nested Loop Join
- Cost:
  - Read S once: cost B(S)
  - Outer loop runs B(S)/(M-1) times, and each time need to read R: costs B(S)B(R)/(M-1)
  - Total cost: B(S) + B(S)B(R)/(M-1)
- Notice: it is better to iterate over the smaller relation first
- $R \bowtie S$: R=outer relation, S=inner relation

Two-Pass Algorithms Based on Sorting

- Recall: multi-way merge sort needs only two passes !
- Assumption: $B(R) <= M^2$
- Cost for sorting: 3B(R)

Two-Pass Algorithms Based on Sorting

Duplicate elimination $\delta(R)$

- Trivial idea: sort first, then eliminate duplicates
- Step 1: sort chunks of size M, write
  - cost 2B(R)
- Step 2: merge M-1 runs, but include each tuple only once
  - cost B(R)
- Total cost: 3B(R), Assumption: $B(R) <= M^2$

Two-Pass Algorithms Based on Sorting

Grouping: $\gamma_{city,\ sum(price)}$ (R)

- Same as before: sort, then compute the sum(price) for each group
- As before: compute sum(price) during the merge phase.
- Total cost: 3B(R)
- Assumption: $B(R) <= M^2$

Two-Pass Algorithms Based on Sorting

Binary operations: R ∩ S, R U S, R – S

- Idea: sort R, sort S, then do the right thing
- A closer look:
  - Step 1: split R into runs of size M, then split S into runs of size M. Cost: 2B(R) + 2B(S)
  - Step 2: merge M/2 runs from R; merge M/2 runs from S; ouput a tuple on a case by cases basis
- Total cost: 3B(R)+3B(S)
- Assumption: $B(R)+B(S) <= M^2$

Two-Pass Join Algorithms Based on Sorting

- Start by sorting both R and S on the join attribute:
  - Cost: 4B(R)+4B(S) (because need to write to disk)
- Read both relations in sorted order, match tuples
  - Cost: B(R)+B(S)
- Difficulty: many tuples in R may match many in S
  - If at least one set of tuples fits in M, we are OK
  - Otherwise need nested loop
  - Total cost: 5B(R)+5B(S)
  - Assumption: $B(R) <= M^2$, $B(S) <= M^2$

49

Two-Pass Algorithms Based on Sorting

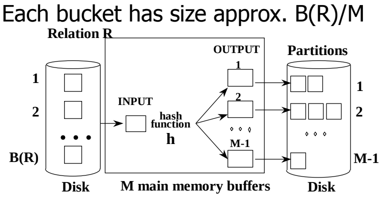
Join $R \bowtie S$

- If the number of tuples in R matching those in S is small (or vice versa) we can compute the join during the merge phase
- Total cost: 3B(R)+3B(S)
- Assumption: $B(R) + B(S) <= M^2$

50

Two Pass Algorithms Based on Hashing

- Idea: partition a relation R into buckets, on disk
- Each bucket has size approx. B(R)/M



- Does each bucket fit in main memory ?
  - Yes if $B(R)/M <= M$, i.e. $B(R) <= M^2$

51

Hash Based Algorithms for δ

- Recall: $\delta(R)$ = duplicate elimination
- Step 1. Partition R into buckets
- Step 2. Apply δ to each bucket (may read in main memory)
- Cost: 3B(R)
- Assumption: $B(R) <= M^2$

52

Hash Based Algorithms for γ

- Recall: $\gamma(R)$ = grouping and aggregation
- Step 1. Partition R into buckets
- Step 2. Apply γ to each bucket (may read in main memory)

- Cost: 3B(R)
- Assumption: $B(R) <= M^2$

53

Hash-based Join

- $R \bowtie S$
- Recall the *main memory hash-based join*:
  - Scan S, build buckets in main memory
  - Then scan R and join

54

Partitioned Hash Join

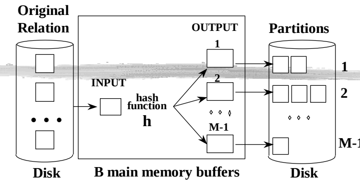
R ⋈ S

- ⌘Step 1:
  - Hash S into M buckets
  - send all buckets to disk
- ⌘Step 2
  - Hash R into M buckets
  - Send all buckets to disk
- ⌘Step 3
  - Join every pair of buckets

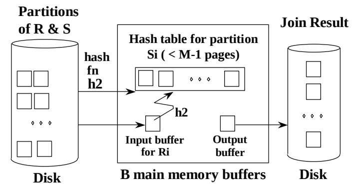
55

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.



Partitioned Hash Join

- ⌘Cost: 3B(R) + 3B(S)
- ⌘Assumption: min(B(R), B(S)) <= M^2

57

Hybrid Hash Join Algorithm

- ⌘Partition S into k buckets
- ⌘But keep first bucket S_1 in memory, k-1 buckets to disk
- ⌘Partition R into k buckets
  - First bucket R_1 is joined immediately with S_1
  - Other k-1 buckets go to disk
- ⌘Finally, join k-1 pairs of buckets:
  - (R_2,S_2), (R_3,S_3), …, (R_k,S_k)

58

Hybrid Join Algorithm

- ⌘How big should we choose k ?
- ⌘Average bucket size for S is B(S)/k
- ⌘Need to fit B(S)/k + (k-1) blocks in memory
  - B(S)/k + (k-1) <= M
  - k slightly smaller than B(S)/M

59

Hybrid Join Algorithm

- ⌘How many I/Os ?
- ⌘Recall: cost of partitioned hash join:
  - 3B(R) + 3B(S)
- ⌘Now we save 2 disk operations for one bucket
- ⌘Recall there are k buckets
- ⌘Hence we save 2/k(B(R) + B(S))
- ⌘Cost: (3-2/k)(B(R) + B(S)) = (3-2M/B(S))(B(R) + B(S))

60