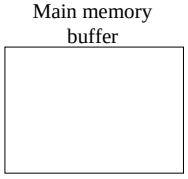
CS411
Database Systems

10: Indexing 2
11: Query Execution

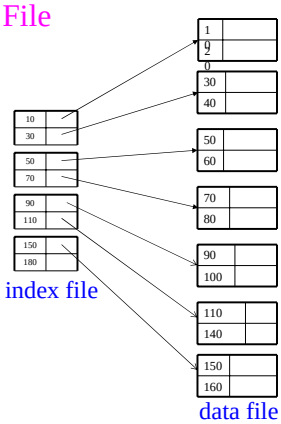
Kazuhiro Minami

Revisiting Sequential Indexes on a Sequential Data File

Main memory buffer

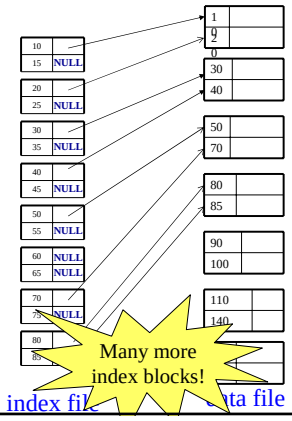


Q: how many disk I/O's do we need to get a record with key value'150'?
Q: If we want to avoid a binary search on index blocks, what can we do?



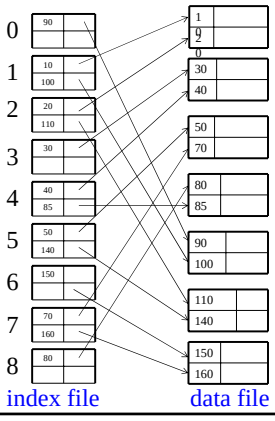
Direct Addressing Approach

- Suppose that a key value is a multiple of 5
- We add an entry for every possible key value in index
- If we look up a record with key '50',
- then, we can figure out that we should look up the 5th index block
- Q: How many disk I/O's do we need in this scheme?
- Q: Is there any problem?

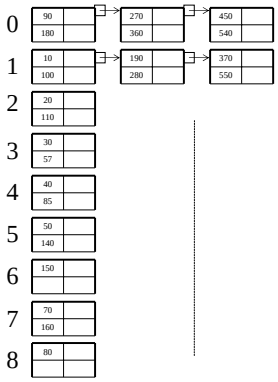


Hashing-based Approach

- Consider a hash function
  h(v) = v mod 9
- Pointer for value *v* goes to *h(v)th* index block
- Note that we only store only pointers to existing records
- Q: How many index blocks do we need?
- Q: How many disk I/O's do we need to find a record with value '50'?
- Q: Any other observations?



However, as we have more records, we need overflow blocks



index file

Hash Tables

- Secondary storage hash tables are much like main memory ones
- Recall basics:
  - There are n *buckets*
  - 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

Extensible Hash Table

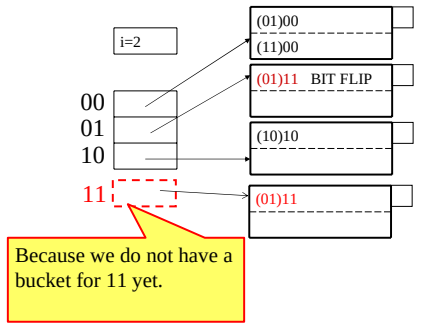
- Allows hash table (i.e., #buckets) to grow, to avoid performance degradation
- Assume a hash function h that returns numbers in {0, …, $2^k - 1$}
- Instead of using a different hash function for each i = 1,…,k, we use the same hash function h
- How?
- The trick is to only look at first i most significant bits $2^i << 2^k$ where 2^i is #buckets n

Linear Hash Table

- Idea: extend only one entry at a time
- Use the *i* bits at the end of a hash value as a bucket ID
- Problem: #buckes *n* = no longer a power of 2
- Let *i* be #bits necessary to address *n* buckets; that is,
  - $2^{i-1} < n <= 2^i$
- We don't have a bucket for hash value *v* where $n <= v < 2^i$
- If $n <= k$, change most significant bit of *k* from 1 to 0
  - if i = 3, n = 5, k = 110 (= 6), entries for *k* go to the bucket for 010 (=2).

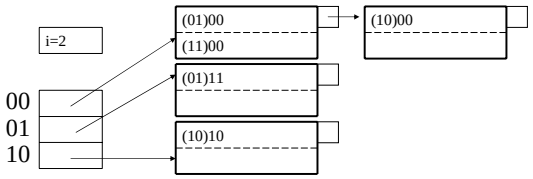
Linear Hash Table Example

- N=3



Linear Hash Table Example

- Insert 1000: overflow blocks…



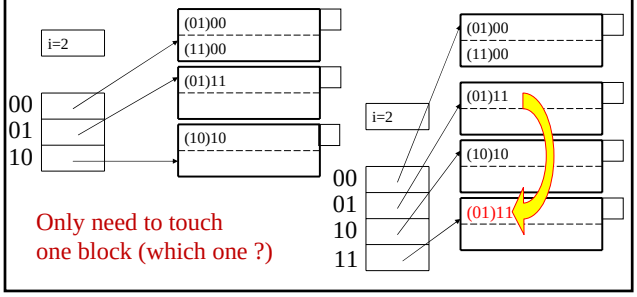
Linear Hash Tables

- Extension: independent on overflow blocks
- Extend n:=n+1 when average number of records per block exceeds (say) 80%

Linear Hash Table Extension

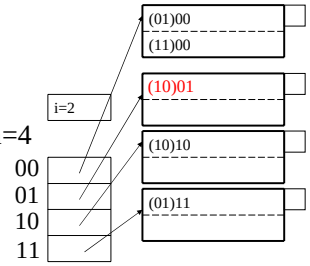
- From n=3 to n=4,

Current number of records r <= 1.6 * n.



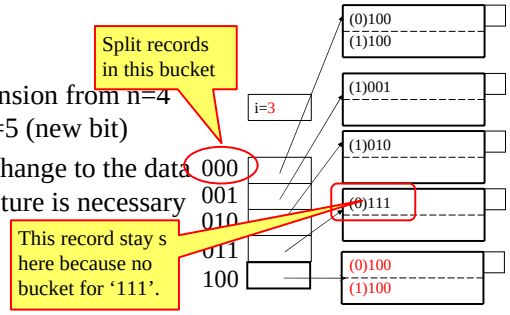
Linear Hash Table Extension

- From n=3 to n=4 finished
- Insert 1001
- Need extension from n=4 to n=5 (new bit)

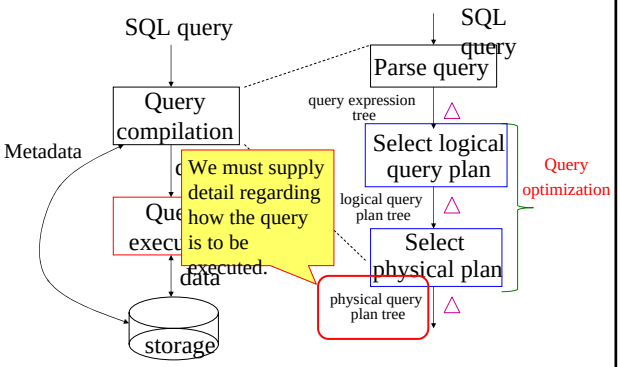


Linear Hash Table Extension

- From n=3 to n=4 finished
- Extension from n=4 to n=5 (new bit)
- No change to the data structure is necessary



Components of Query Processor



Outline

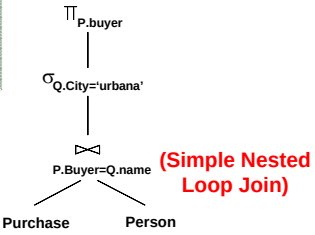
- Logical/physical operators
- Cost parameters
- One-pass algorithms
- Nested-loop joins
- Two-pass algorithms based on sorting

Logical v.s. Physical Operators

- Logical operators
  - *what* they do
  - e.g., union, selection, project, join, grouping
- Physical operators
  - *how* they do it
  - Principal methods: scanning, hashing, sorting, and indexing
  - Consider assumptions as to the amount of available main memory
  - e.g., nested loop join, sort-merge join, hash join, index join

Physical Query Plans

```
SELECT   P.buyer
FROM     Purchase P, Person Q
WHERE  P.buyer=Q.name AND
          Q.city='urbana'
```



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 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

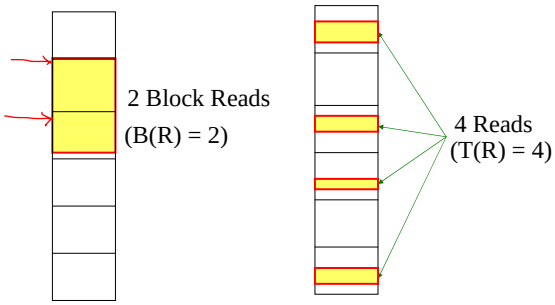
Cost Parameters

- Cost parameters
  - M = number of blocks that fit in main memory
  - B(R) = number of blocks holding R
  - T(R) = number of tuples in R
  - V(R,a) = number of distinct values of the attribute a
- Estimating the cost:
  - Important in optimization (next topic)
  - Compute I/O cost only
  - We consider the cost to *read* the tables
  - We don't include the cost to *write* the result (because pipelining)

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 those of other tables)
  - May need one block read for each record

Scanning Clustered/Uncluserted Tables



Cost of the Scan Operator

- Clustered relation:
  - Table scan: B(R)
  - Index scan: B(R) ignoring the cost for reading a index file
- Unclustered relation
  - T(R)

We assume clustered relations to estimate the costs of other physical operators.

Classification of Physical Operators

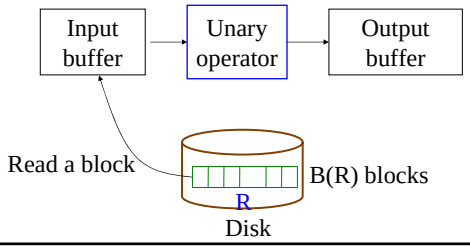
- One-pass algorithms
  - Read the data only once from disk
  - Usually, require at least one of the input relations fit in main memory
- Nested-Loop Join algorithms
  - Read one relation only once, while the other will be read repeatedly from disk
- Two-pass algorithms
  - First pass: read data from disk, process it, write it to the disk
  - Second pass: read the data for further processing

One pass algorithms

One-pass Algorithms

Selection σ(R), projection Π(R)

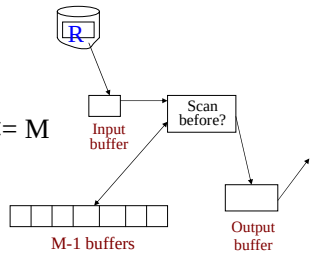
- Both are *tuple-at-a-Time* algorithms
- Cost: B(R)



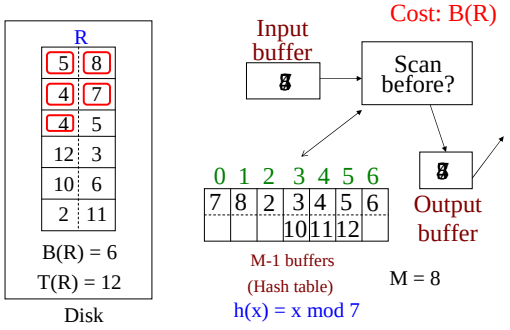
One-pass Algorithms

Duplicate elimination δ(R)

- Need to keep a dictionary in memory:
  - balanced search tree
  - hash table
  - etc
- Cost: B(R)
- Assumption: B(δ(R)) <= M



Duplicate elimination δ(R) when B(δ(R)) <= M



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

- Need to keep a dictionary in memory
- Also store the sum(price) for each city
- Cost: B(R)
- Assumption: number of cities fits in memory

Binary Operations: R U S, R – S

- Assumption: min(B(R), B(S)) <= M
- Scan a smaller table of R and S into main memory, then read the other one block by one
- Cost: B(R)+B(S)
- Example: R ∩ S
  - Read S into M-1 buffers and build a search structure
  - Read each block of R, and for each tuple t of R, see if t is also in S.
  - If so, copy t to the output, and if not, ignore t

Nested loop join

Tuple-based Nested Loop Joins

- Join R ⋉ S

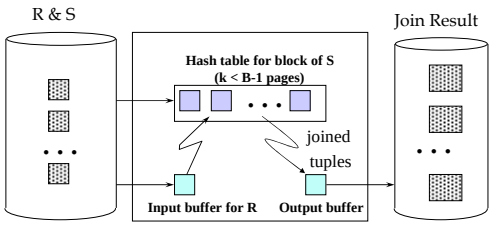
for each tuple r in R do
  for each tuple s in S do
    if r and s join then output (r,s)

- Cost: T(R) T(S), or T(R) B(S) if R is clustered

Block-based Nested Loop Joins

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

Block-based Nested Loop Joins



Block-based Nested Loop Joins

- 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
- S ⋉ R: S=outer relation, R=inner relation