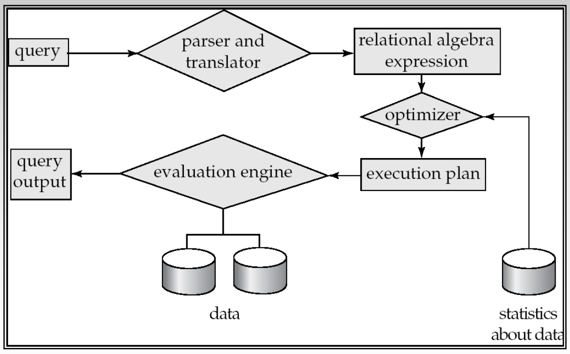
12. Query Processing

- Query Processing Steps
  1. parsing & translation: SQL → Internal relational algebra
  2. optimization: pick amongst several the best plan
  3. evaluation or execution of the selected plan



Query Optimization

- Amongst all equivalent plans choose the one with lowest cost
- Cost is measured as total time for answering query
- Factors that contribute to time cost
  - disk accesses (predominant)
  - CPU and network communication
- Disk access cost in terms of
  - Number of seeks X average-seek-cost
  - Number of blocks read X average-block-read-cost
  - Number of blocks written X average-block-write-cost
    - note that a write of a block requires a re-read after being written to ensure correctness

Measures of Query Cost (Cont.)

- We use the number of block transfers *from disk and the* number of seeks
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for b block transfers plus the number S of seeks
    $b * t_T + S * t_S$
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- Some algorithms can reduce disk IO by extra buffer space
  - Size of real buffer memory available depends on other concurrent queries
    - known only during execution
  - We use worst case estimates, assuming minimum amount of memory needed for the operation
  - Concurrent queries may share blocks needed avoiding disk I/O
    - Impossible to measure this in our cost estimation (during optimization)

292

Cost Parameters

- Statistical information maintained in the system's catalog

n_r = number of tuples in the relation r
b_r = number of blocks containing tuples of relation r
s_r = average size of a tuple of relation r
f_r = blocking factor of r, I.e. the number of r tuples that fit in a block

Note: $b_r = \frac{n_r}{f_r}$

V(A,r) = number of distinct values of attribute A in r
= n_r if A is a key

min(A,r) = minimum value of attribute A in r
max(A,r) = maximum value of attribute A in r

- Two important computations
  - I/O cost of each operation
    - Number of blocks accessed
    - Number of seeks
  - the size of the result

293

Types of Access & Algorithms

1. Linear Scan (simple predicate)
   - Single value
   - Range of values
2. Index Scan (simple predicate)
   - Single value
   - Range of values
3. Complex Predicates
4. Merge-Sort
5. Join Algorithms
   - Nested Loop
   - Indexed
   - Merge-Sort
   - Hash
6. Other Operations (outerjoins, group by, duplicate elimination, etc)
7. Expressions

1. Linear Scan: Selection/Projection

simple predicate: R.A = v

- A1: search for equality on a non key-unsorted: R.A = v [or R.A ≤ v]

$$Cost = t_S + b_r * t_T$$ 1 seek + b_r transfers

- projection on attribute A without duplicate elimination: cost as above

$$Cost = t_S + b_r * t_T$$

- A1: R.A=v and R.A is a primary key – sorted (we stop early when the key is found)

$$Cost = t_S + \frac{b_r}{2} * t_T$$

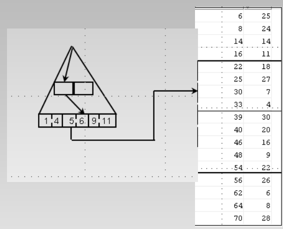
- A1: If the relation is sorted (primary key), we can stop earlier for R.A ≤ v

$$Cost = t_S + \frac{b_r}{2} * t_T$$

2. Index Scan: Selection

simple predicate: R.A = v

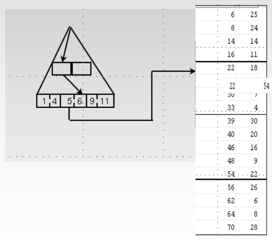
- **Index scan – search uses an index to find records (selection condition must be on search-key of index)**



- **A2 (primary key index, equality).**
  R.A=v retrieves a single record

$$Cost = (h_i + 1)(t_S + t_T)$$

- **A3 (clustering index, equality on nonkey)**
  ***R.A=v* retrieves multiple records**
  (clustering index- records are on consecutive blocks):



$$Cost = h_i * (t_S + t_T) + t_T * b$$

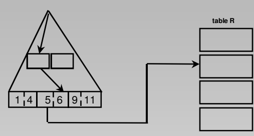
Where

$$b = \frac{\frac{n_r}{V(A,r)}}{f_r}$$

Index Scan: Selection

simple predicate: R.A = v

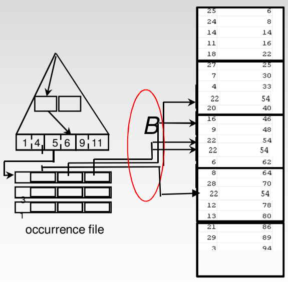
- **A4 (secondary index, equality on candidate key)**
  R.A=v retrieves a single record:



$$Cost = (h_i + 1)(t_S + t_T)$$

(same as A2)

- **A4 (secondary index, equality on nonkey)**
  R.A=v retrieves multiple records (each of which may be on a different block)



$$Cost = (h_i + 1 + B)(t_S + t_T)$$

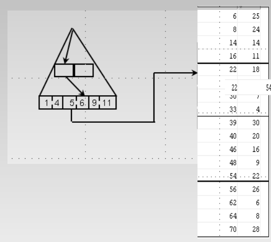
where $B = \frac{n_r}{V(A,r)}$

Can be very expensive!

Index Scan: Range Selection

predicate: R.A >= v

- A5 (clustering index, range). R.A≥v (Relation is sorted on A)
  - Use the index to find first tuple ≥ *v* and scan the relation sequentially from that point on



$$Cost = h_i * (t_S + t_R) + t_S + t_R * B$$

w/ a known value v

$$B = b_r * \frac{\max(A,r) - v}{\max(A,r) - \min(A,r)}$$

w/ v unbounded

$$B = \frac{b_r}{2}$$

- For R.A ≤ v
  - do not use the index
  - scan the relation sequentially till first tuple > v

w/ a known v

$$Cost = b_r * \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$$

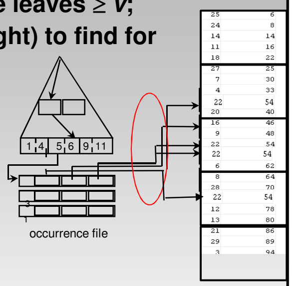
w/ v unbounded

$$Cost = \frac{b_r}{2}$$

Index Scan: Range Selection

predicate: R.A >= v

- A6 (secondary index, range). R.A≥v (Relation is not sorted on A)
  - Use the index to find first index entry in the B+Tree leaves ≥ *v*; then scan the *B+TreeLeaves* sequentially (to the right) to find for each entry its pointer to the sub-leaf level records



$$Cost = (h_i + \frac{B + TreeLeaves}{2} + C)(t_S + t_R)$$

w/ a bounded v

$$C = (1 + \frac{n_r}{V(A,r)})(\max(A,r) - v)$$

w/ v unbounded

$$C = \frac{n_r}{2}$$

- For R.A≤v Start at the first (left most) index entry and scan until you find one greater than v

w/ a bounded v

$$Cost = (\frac{B + TreeLeaves}{2} + C)(t_S + t_R)$$

$$C = (1 + \frac{n_r}{V(A,r)})(v - \min(A,r))$$

w/ v unbounded

$$C = \frac{n_r}{2}$$

  - tuples with the same value are scattered
  - can be a lot more expensive than a linear scan

3. Complex Selection Predicates

Conjunction: $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$

- A7 (conjunctive selection using one index).
  - Select one of θ_i and algorithms A1 through A6 that results in the least cost for $\sigma_{\theta i}(r)$.
  - Test other conditions on tuple after fetching it into memory buffer.
- A8 (conjunctive selection using composite index).
  - Use appropriate composite (multiple-key) index if available.
- A9 (conjunctive selection by intersection of TIDs).
  - Requires multiple indices
  - Use corresponding index for each condition, and take intersection of all the obtained sets of TIDs.
  - Then fetch records from file
  - For those attributes that do not have indices, apply the predicate tests in memory

300

Complex Selection Predicates

Disjunction: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$

- A10 (disjunctive selection by union of TIDs).
  - Applicable if *only all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets TIDs
  - Then fetch records from file
- Negation: $\sigma_{\neg\theta}(r)$
  - Use linear scan on file

301

4. External Sorting with Sort-Merge

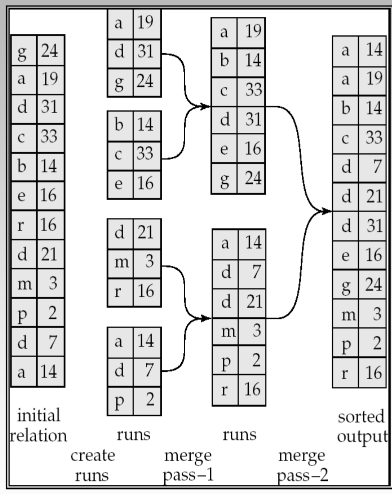
- external vs. internal sorting: relation/file does not fit in memory: $b_r >> M$
- create runs phase using M buffers:

```
repeat until done
   read M  blocks of the relation (or rest if <=M)
   internal sort using any sort method, e.g
      QuickSort(M)
   write the sorted tuples into a run R data file
end
```

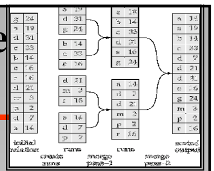
- merge-runs phase:

```
read one block from each run;
merge tuples on the result;
advance the pointer from the run you appended last;
if the block of a run is empty, read the next one until
      all blocks of all runs are done
```

- this assumes that a block from each run can be kept in main memory. If not, then the same algorithm has to be applied in multiple passes



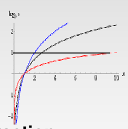
Type equation here.External Merge Cost



- Cost analysis:
  - Initial number of runs: b_r / M (above example 12/3=4)
  - Number of merge passes needed: $\lceil \log_{M-1}(b_r/M) \rceil$ ($\log_2 4=2$)
  - Block transfers for each run creation is $b_r+b_r=2b_r$
    - for final pass, we don't count write cost (pipelined to the display or to follow up operator in a rel. expression)
    - Thus total number of block transfers for external sorting:

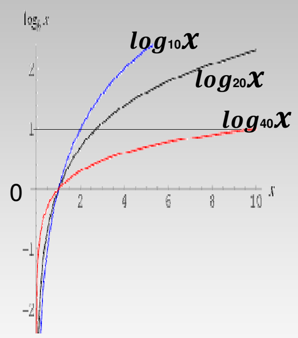
$$2\, b_r (\lceil \log_{M-1}(b_r / M) \rceil) + b_r = b_r (2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$$ 12*(2*2+1)=60

Reads&Writes for each pass — Final Merge Reads



  - If $M \geq \lceil b_r/M \rceil$ (only one pass is required) the expression $\lceil \log_{M-1}(b_r/M) \rceil = 1$
    total cost = $3b_r$
    fraction
  - However, if $M > b_r$ then the expression $\lceil \log_{M-1}(b_r/M) \rceil = 0$
    total cost = b_r fits in memory
    negative
  - Seeks = $2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{M-1}(b_r / M) \rceil - 1)$ 2*12/3+12*3=44

$\log_{M-1}$



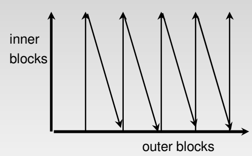
5. Join Algorithms: Nested Loop

- **tuple-oriented:**

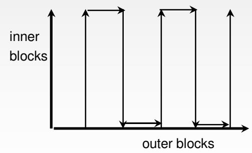
```
for each tuple t(r) in r do begin
   for each tuple t(s) in s  do begin
       join(t(r),t(s)) and append the result to the output
   end
 end
```

- **block-oriented:**

```
for each  block b_r  in  r  do begin
   for each block  b_s  in s  do begin
       join(b_r ,b_s and append the result to the output
   end
 end
```



- **reverse inner loop**
  similar to above but for even outer blocks we scan the inner relation in reverse



5.1 Cost of Block-Oriented Nested Loop

Buffer size M+1

- cost depends on the number of buffers and the buffer replacement strategy
  - fasten 1 block from the outer relation, allocate M for the inner, and LRU

    $Cost = b_r + b_r b_s$ block transfers assuming that $b_s > M$
    (wastes M-1 buffers and is the same as if M=2)

    $Seeks = 2b_r$ 1 to get each outer block+1 to get to the 1st inner block

  - fasten M blocks from the outer relation, allocate 1 for the inner
    1: read M from the outer — cost: M blocks
    2: for each block of s join 1 X M blocks — cost: b_s blocks
    3: repeat with the next M blocks of r until all done — repeated b_r/M times

$$Cost = \frac{(M + b_s) b_r}{M} = b_r + \frac{b_r b_s}{M}$$

$$Seeks = \lceil \frac{2b_r}{M} \rceil$$

- which relation should be the outer?

5.2 Indexed Nestet-Loop Join (equi-join)

- inner relation has an index (clustering or not) on the joining attribute
  - Consider building one just for doing the join

```
for each  block b_r  in  r  do begin
   for each tuple t(r)   in b_r   do begin
       search the index B on s with the value t.A of the joining attr. A
       and join(t(r),t.A)
   end
end
```

- cost = $b_r (t_T + t_S) + n_r{}^* Cost(\sigma(S.A=c))$

where Cost(σ(S.A=c)) is as computed for indexed selection (A2-A6)

5.3 Sort-Merge-Join

- two phases
  - sorting phase: sort both relations (this can be done in parallel)
  - merging phase: join tuples during the merge

sort R on joining attribute
sort S on joining attribute
merge(sorted-R,sorted-S)

- cost with M buffers

$$\text{cost} = b_r\,(2\lceil \log_{M-1}(b_r/M)\rceil + 1) + b_r + b_s\,(2\lceil \log_{M-1}(b_s/M)\rceil + 1) + b_s + b_r + b_s$$

Sorting R | Write sorted R | Sorting S | Write sorted S | Merge

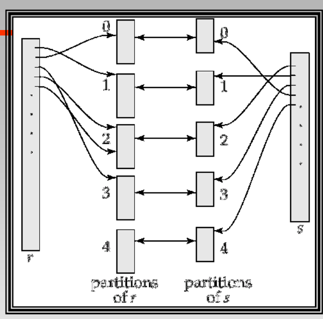
if one pass is required the expressions $\lceil \log_{M-1}(b_r/M)\rceil = \lceil \log_{M-1}(b_s/M)\rceil = 1$
so the total cost is $3*b_r + b_r + b_r + 3*b_s + b_s + b_s = 5*b_r + 5*b_s$

However, if M> b_r then the expression evaluates to $3*b_r + 5*b_s$)

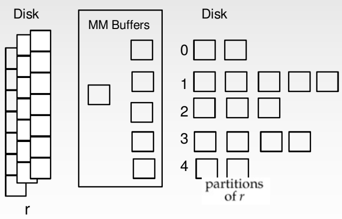
5.4 Hash-Join (equi-joins)

- two phases
  - hash phase: hash both relations into hashed partitions (this can be done in parallel)
  - bucket-wise join phase: join tuples of the same partitions only

hash R on the joining into H(R) partitions
hash S on the joining into H(S) partitions
nested-loop join of corresponding partitions Hj(R),Hj(S)
or main-memory hash index join of corresponding partitions



- Number of partitions n?
  - Choose n large enough to make each partition of one of the relations to fit in the buffer memory $|s_i| \leq B$
  - During partitioning, one block of memory is reserved as the output buffer for each partition
  - each partition consists of several blocks



Hash-Join Algorithm Details

The hash-join of *r* and *s* is computed as follows:

1. **Partition the relation *s* using hashing function *h*.**
   When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. **Partition *r* similarly.**
3. **For each *i*:**
   (a) **Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one *h*.**
   (b) **Read the tuples in r_i from the disk one by one. For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.**

Relation *s* is called the **build input** and
r is called the **probe input.**

310

Hash-Join Number of Partitions

- **The number *n* of partitions and the hash function *h* is chosen such that each s_i should fit in memory.**
  - **Typically we choose $n_h = \lceil b_s/M \rceil * f$ where f is a "fudge factor", typically around 1.2**
  - **The probe relation partitions r_i need not fit in memory**

311

Cost of Hash-Join

- cost of hash join is
  $3(b_r + b_s) + 4 * n_h$ block transfers
  $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) + 2 * n_h$ seeks
  the red font factors account for the partially filled blocks in the partitions and b_b is the number of blocks you read into the buffer every time
- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$.

Assumption: after hashing the partitions of r and s have the same size with r and s. Otherwise, we need to add another small fudge factor.

Example of Cost of Hash-Join

instructor ⋈ *teaches*

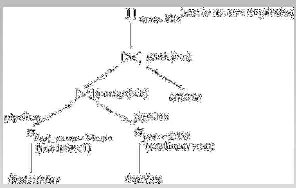
- M=20 blocks $b_{instructor}$= 100 $b_{teaches}$ = 400
- *instructor* is the build input. Partition it into 5 partitions, each of size 20 blocks. This partitioning can be done in one pass
- Partition *teaches* into 5 partitions, each of size 80. This is also done in one pass
- During partitioning, we need to allocate a buffer for input, and a buffer for output in each of the 5 partitions. Therefore, b_b=M/(1+5)=20/6=3
- Total cost, ignoring cost of writing partially filled blocks:
  - 3(100 + 400) +4*5= 1500+20 block transfers
    2($\lceil$100/3$\rceil$+$\lceil$400/3$\rceil$)+2*5 = 336+10 seeks

6. Other Operations

- Outer Joins
  - Left outerjoin easy
  - Right/Full outerjoin (may need some bookkeeping)
- Duplicate elimination
  - Sort at the end and eliminate
  - Hash output and eliminate
- Aggregates
  - Sum, count, min, max easily kept during execution
  - Avg = Sum / count
  - Std = sqrt(ssum/count)
- Set operations ($\cup$, $\cap$ and —): can either use
  - A variant of merge-join after sorting, or
  - A variant of hash-join

7. Evaluation of Expressions

- Relational expressions may contain several operations- evaluation tree



- Alternatives for evaluating an evaluation tree

1. Materialization:
   - generate and store(=materialize) intermediate results of an expression
   - Input materialized intermediate results to compute remaining operators
   - Repeat until done
2. Pipelining:
   - pass on tuples to parent operations even as an operation is being executed
   - No intermediate results

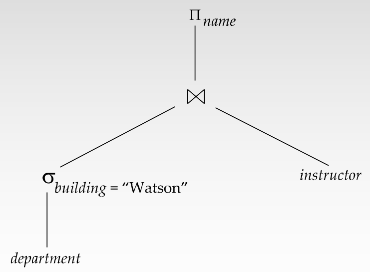
Materialization

- Materialized evaluation: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

$$\sigma_{building="Watson"}(department)$$

then join the stored intermediate result with *instructor,*

and finally compute the projection on *name.*



Materialization (Cont.)

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
- Double buffering:
  - use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

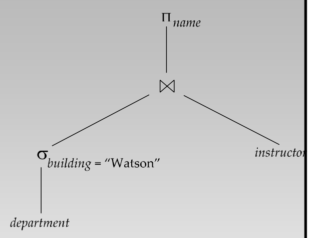
Pipelining

- **Pipelined evaluation: evaluate several operations simultaneously, passing the results of one operation on to the next.**

E.g., do not materialize or use temp store

$$\sigma_{building="Watson"}(department)$$

- **instead, pass tuples directly to the join.**



- **Much cheaper than materialization: no need to store a temporary results**
- **Pipelining may not always be possible – e.g., sort, hash-join.**
- **Pipelines can be executed in two ways: demand driven and producer driven**

Pipelining (Cont.)

- **Implementation of demand-driven pipelining**
  - **Each operation is implemented as an iterator:**
    - **open()**
      - E.g. file scan: initialize file scan
        - state: pointer to beginning of file
      - **next()**
      - E.g. for file scan: Output next tuple, advance, and save pointer as iterator state. .
    - **close()**