Department of Computer Science

CS 44800: Introduction To Relational Database Systems

Processing Individual Operations

Prof. Chris Clifton

19 October 2021

Indiana Center for Database Systems ™

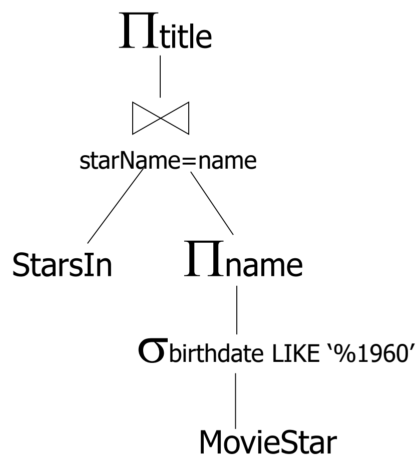
Department of Computer Science

Query Processing

- Process overview
  - Parse to relational algebra
  - Transform to optimized query plan
  - Evaluate steps in plan on the data
- Cost measures
  - Goal: reduce disk blocks read
  - Alternate view: Number of tuples processed, cost of operation
- Pipelined processing
- Processing individual operations
  - Algorithms
  - Use of indexes
- Query transformation
  - Legal transformations to equivalent queries
  - "Always good" transformations
- Cost estimation
  - Expected number of tuples processed
  - Cost of operation
  - Cost of plan
- Putting it all together: Query Optimizer

PURDUE UNIVERSITY
Department of Computer Science

Logical Query Plan



Selection Operation

- **File scan**
- Algorithm **A1** (**linear search**). Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate = b_r block transfers + 1 seek
    - b_r denotes number of blocks containing records from relation *r*
  - If selection is on a key attribute, can stop on finding record
    - cost = (b_r /2) block transfers + 1 seek
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices
- Note: binary search generally does not make sense since data is not stored consecutively
  - except when there is an index available,
  - and binary search requires more seeks than index search

Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
- **A2** (**clustering index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$
- **A3** (**clustering index, equality on nonkey**) Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$

Selections Using Indices

- **A4** (**secondary index, equality on key/non-key**).
  - Retrieve a single record if the search-key is a candidate key
    - $Cost = (h_i + 1) * (t_T + t_S)$
  - Retrieve multiple records if search-key is not a candidate key
    - each of *n* matching records may be on a different block
    - Cost = $(h_i + n) * (t_T + t_S)$
      - Can be very expensive!

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:
- **A5** (**clustering index, comparison**). (Relation is sorted on A)
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index
- **A6** (**clustering index, comparison**).
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
  - In either case, retrieve records that are pointed to
  - requires an I/O per record; Linear file scan may be cheaper!

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$
- **A7** (**conjunctive selection using one index**).
  - Select a combination of θ_i and algorithms A1 through A7 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 identifiers**).
  - Requires indices with record pointers.
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.

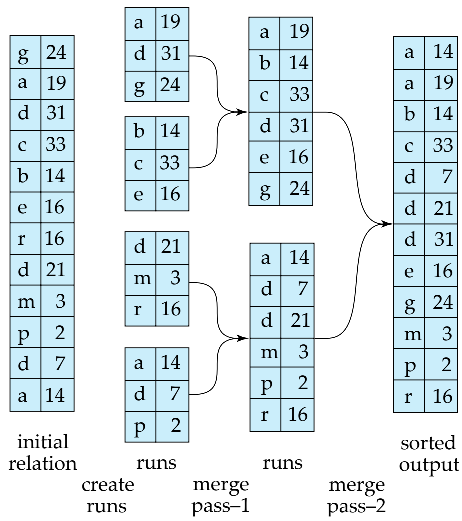
Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$.
- **A10** (**disjunctive selection by union of identifiers**).
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file
- **Negation:** $\sigma_{\neg\theta}(r)$
  - Use linear scan on file
  - If very few records satisfy $\neg\theta$, and an index is applicable to θ
    - Find satisfying records using index and fetch from file

Sorting

- We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple.
- For relations that fit in memory, techniques like quicksort can be used.
  - For relations that don't fit in memory, **external sort-merge** is a good choice.

Example: External Sorting Using Sort-Merge



External Sort-Merge

Let *M* denote memory size (in pages).

1. **Create sorted runs**. Let *i* be 0 initially.
   Repeatedly do the following till the end of the relation:
   (a) Read *M* blocks of relation into memory
   (b) Sort the in-memory blocks
   (c) Write sorted data to run R_i; increment *i.*

Let the final value of *i* be *N*

2. *Merge the runs (next slide)…..*

External Sort-Merge (Cont.)

2. **Merge the runs (N-way merge)**. We assume (for now) that *N* < *M*.
   1. Use *N* blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
   2. **repeat**
      1. Select the first record (in sort order) among all buffer pages
      2. Write the record to the output buffer. If the output buffer is full write it to disk.
      3. Delete the record from its input buffer page.
         If the buffer page becomes empty **then**
         read the next block (if any) of the run into the buffer.
   3. **until** all input buffer pages are empty:

External Sort-Merge (Cont.)

- If $N \geq M$, several merge *passes* are required.
  - In each pass, contiguous groups of *M* - 1 runs are merged.
  - A pass reduces the number of runs by a factor of *M* -1, and creates runs longer by the same factor.
    - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one.

External Merge Sort (Cont.)

- Cost analysis:
  - 1 block per run leads to too many seeks during merge
    - Instead use b_b buffer blocks per run
      ➔ read/write b_b blocks at a time
    - Can merge $\lfloor M/b_b \rfloor - 1$ runs in one pass
  - Total number of merge passes required: $\lceil \log_{\lfloor M/bb \rfloor - 1}(b_r/M) \rceil$.
  - Block transfers for initial run creation as well as in each pass is $2b_r$
    - for final pass, we don't count write cost
      - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk
    - Thus total number of block transfers for external sorting:
      $b_r (2 \lceil \log_{\lfloor M/bb \rfloor - 1} (b_r / M) \rceil + 1) \lceil$
  - Seeks: next slide

External Merge Sort (Cont.)

- Cost of seeks
  - During run generation: one seek to read each run and one seek to write each run
    - $2 \lceil b_r / M \rceil$
  - During the merge phase
    - Need $2 \lceil b_r / b_b \rceil$ seeks for each merge pass
      - except the final one which does not require a write
    - Total number of seeks:
      $2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{\lfloor M/bb \rfloor - 1}(b_r / M) \rceil - 1)$

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following information
  - Number of records of *student*: 5,000 *takes*: 10,000
  - Number of blocks of *student*: 100 *takes*: 400

Nested-Loop Join

- To compute the theta join $r \bowtie_{\theta} s$

  for each tuple t_r **in** r **do begin**
      for each tuple t_s **in** s **do begin**
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add $t_r \cdot t_s$ to the result.
      end
  end
- *r* is called the **outer relation** and *s* the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is
  $n_r * b_s + b_r$ block transfers, plus $n_r + b_r$ seeks
- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- Assuming worst case memory availability cost estimate is
  - with *student* as outer relation:
    - 5000 ∗ 400 + 100 = 2,000,100 block transfers,
    - 5000 + 100 = 5100 seeks
  - with *takes* as the outer relation
    - 10000 ∗ 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers.
- Block nested-loops algorithm (next slide) is preferable.

Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

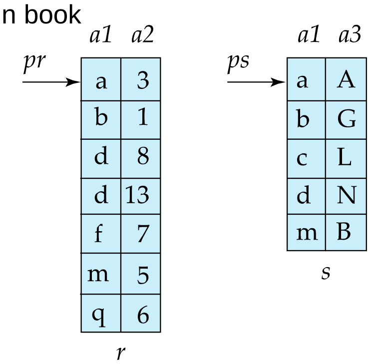
```
for each block Br of r do begin
    for each block Bs of s do begin
        for each tuple tr in Br do begin
            for each tuple ts in Bs do begin
                Check if (tr,ts) satisfy the join condition
                if they do, add tr • ts to the result.
            end
        end
    end
end
```

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join.
- For each tuple t_r in the outer relation *r,* use the index to look up tuples in *s* that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Cost of the join: $b_r (t_T + t_S) + n_r * c$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition.
- If indices are available on join attributes of both *r* and *s,* use the relation with fewer tuples as the outer relation.

Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
   1. Join step is similar to the merge stage of the sort-merge algorithm.
   2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
   3. Detailed algorithm in book



Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory
- Thus the cost of merge join is:

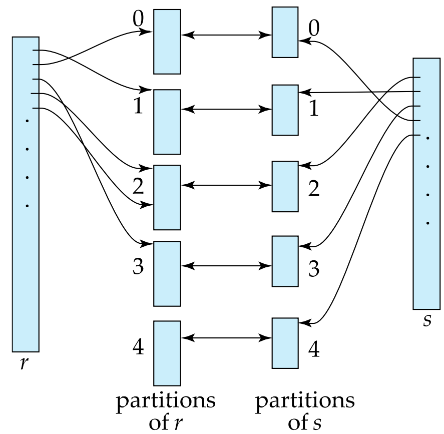
  $b_r + b_s$ block transfers $+ \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks

  + the cost of sorting if relations are unsorted.
- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B^+-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B^+-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - Sequential scan more efficient than random lookup

Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations
- *h* maps *JoinAttrs* values to {0, 1, ..., *n*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.
  - $r_0, r_1, \ldots, r_n$ denote partitions of *r* tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $r_0,, r_1. . ., r_n$ denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.
- *Note:* In book, Figure 12.10 r_i is denoted as H_{ri}, s_i is denoted as H_{si} and *n* is denoted as n_h.

Hash-Join (Cont.)



Hash-Join Algorithm

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**.

Hash-Join algorithm (Cont.)

- The value n and the hash function h is chosen such that each s_i should fit in memory.
  - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a "**fudge factor**", typically around 1.2
  - The probe relation partitions s_i need not fit in memory
- **Recursive partitioning** required if number of partitions n is greater than number of pages M of memory.
  - instead of partitioning n ways, use $M – 1$ partitions for s
  - Further partition the $M – 1$ partitions using a different hash function
  - Use same partitioning method on r
  - Rarely required: e.g., with block size of 4 KB, recursive partitioning not needed for relations of < 1GB with memory size of 2MB, or relations of < 36 GB with memory of 12 MB

Handling of Overflows

- Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others
- **Hash-table overflow** occurs in partition s_i if s_i does not fit in memory. Reasons could be
  - Many tuples in s with same value for join attributes
  - Bad hash function
- **Overflow resolution** can be done in build phase
  - Partition s_i is further partitioned using different hash function.
  - Partition r_i must be similarly partitioned.
- **Overflow avoidance** performs partitioning carefully to avoid overflows during build phase
  - E.g., partition build relation into many partitions, then combine them
- Both approaches fail with large numbers of duplicates
  - Fallback option: use block nested loops join on overflowed partitions

Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is
  $3(b_r + b_s) + 4 * n_h$ block transfers +
  $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
- If recursive partitioning required:
  - number of passes required for partitioning build relation *s* to less than M blocks per partition is $\lceil log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil$
  - best to choose the smaller relation as the build relation.
  - Total cost estimate is:
    $2(b_r + b_s)\lceil log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil + b_r + b_s$ block transfers +
    $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)\lceil log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil$ seeks
- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$.

Complex Joins

- Join with a conjunctive condition:

  $$r \bowtie_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n} s$$

  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins $r \bowtie_{\theta i} s$
    - final result comprises those tuples in the intermediate result that satisfy the remaining conditions

  $$\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$$
- Join with a disjunctive condition

  $$r \bowtie_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n} s$$

  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins $r \bowtie_{\theta_i} s$:

  $$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \ldots \cup (r \bowtie_{\theta_n} s)$$

Other Operations

- **Duplicate elimination** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - *Optimization:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
  - Hashing is similar – duplicates will come into the same bucket.
- **Projection**:
  - perform projection on each tuple
  - followed by duplicate elimination.

Other Operations : Aggregation

- **Aggregation** can be implemented in a manner similar to duplicate elimination.
  - **Sorting** or **hashing** can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - Optimization*:* **partial aggregation**
    - combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the partial aggregates
    - For avg, keep sum and count, and divide sum by count at the end

Other Operations : Set Operations

- **Set operations** (∪, ∩ and —): can either use variant of merge-join after sorting, or variant of hash-join.
- E.g., Set operations using hashing:
  1. Partition both relations using the same hash function
  2. Process each partition *i* as follows.
     1. Using a different hashing function, build an in-memory hash index on r_i.
     2. Process s_i as follows
        - $r \cup s$:
          1. Add tuples in s_i to the hash index if they are not already in it.
          2. At end of s_i add the tuples in the hash index to the result.

Other Operations : Set Operations

- E.g., Set operations using hashing:
  1. as before partition *r* and *s,*
  2. as before, process each partition *i* as follows
     1. build a hash index on r_i
     2. Process s_i as follows
        - $r \cap s$:
          1. output tuples in s_i to the result if they are already there in the hash index
        - $r - s$:
          1. for each tuple in s_i, if it is there in the hash index, delete it from the index.
          2. At end of s_i add remaining tuples in the hash index to the result.

Other Operations : Outer Join

- **Outer join** can be computed either as
  - A join followed by addition of null-padded non-participating tuples.
  - by modifying the join algorithms.
- Modifying merge join to compute r ⟕ s
  - In r ⟕ s, non participating tuples are those in $r - \Pi_R(r \bowtie s)$
  - Modify merge-join to compute r ⟕ s:
    - During merging, for every tuple t_r from r that do not match any tuple in s, output t_r padded with nulls.
  - Right outer-join and full outer-join can be computed similarly.

Cache Conscious Algorithms

- Goal: minimize cache misses, make best use of data fetched into the cache as part of a cache line
- For sorting:
  - Use runs that are as large as L3 cache (a few megabytes) to avoid cache misses during sorting of a run
  - Then merge runs as usual in merge-sort
- For hash-join
  - First create partitions such that build+probe partitions fit in memory
  - Then subpartition further s.t. build subpartition+index fits in L3 cache
    - Speeds up probe phase significantly by avoiding cache misses
- Lay out attributes of tuples to maximize cache usage
  - Attributes that are often accessed together should be stored adjacent to each other
- Use multiple threads for parallel query processing
  - Cache misses leads to stall of one thread, but others can proceed