Database Development and Design (CPT201)

Lecture 4b:
Query Evaluation - Selction

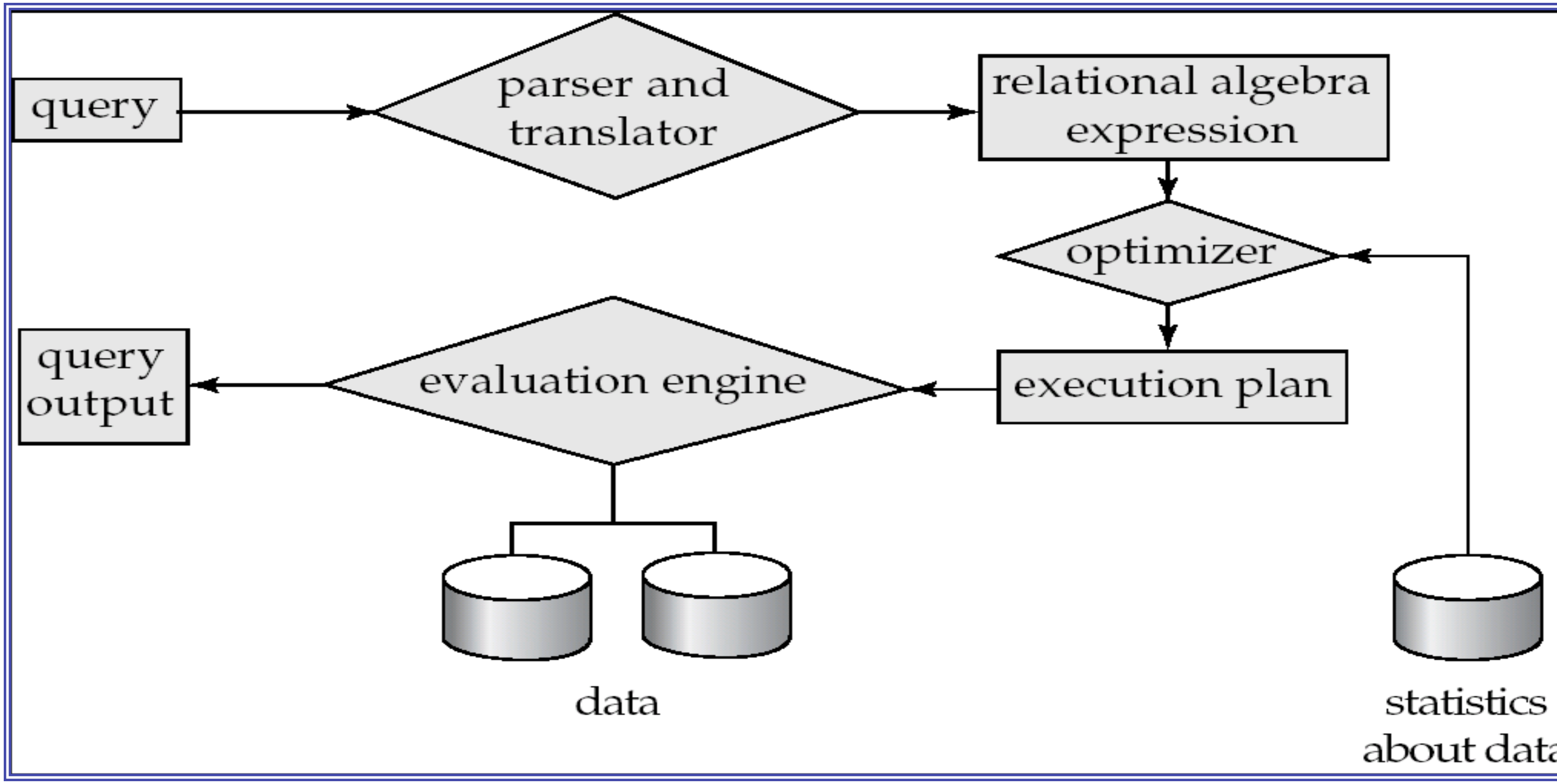
Dr. Wei Wang

Department of Computing

Learning Outcomes

- Basic steps in query processing
- How to measure query costs
- Algorithms for evaluating relational algebra operations (Selection and Projection)
- External Merge Sort

Basic Steps in Query Processing



Parsing and translation

- Translate the query into its internal form.
- This is then translated into relational algebra.
  - (Extended) relational algebra is more compact, and differentiates clearly among the various different operations
- Parser checks syntax, verifies relations
- This is a subject for *compilers*

Evaluation

- The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.
  - The bulk of the problem lies in how to come up with good evaluation plans!

Optimisation

- A relational algebra expression may have many equivalent expressions, e.g.,
  - $\sigma_{balance<2500}(\prod_{balance}(account))$
  - $\prod_{balance}(\sigma_{balance<2500}(account))$
- Each relational algebra operation can be evaluated using several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**, e.g.,
  - Plan 1: use an index on *balance* to find accounts with balance < 2500,
  - Plan 2: perform linear scan and discard accounts with balance ≥ 2500
- **Query Optimisation**: Amongst all equivalent evaluation plans choose the one with the lowest cost.
  - Cost is estimated using statistical information from the database catalog, e.g. number of tuples in each relation, size of tuples, number of blocks in a relation, etc.

Xi'an Jiaotong-Liverpool University 西交利物浦大學

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering a query
  - Many factors contribute to time cost, such as *disk accesses, CPU*, or even network *communication*
- For simplicity, we just use *number of block transfers* from/to disk and *number of seeks* as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for b block transfers plus s seeks: $b * t_T + s * t_S$
- We do not include cost to writing final output to disk in the cost formulae (with some exceptions, will see later)
- We ignore CPU costs for simplicity, as they tend to be much lower
- Evaluating the cost of an algorithm in terms of block transfers and seeks is substantially different from that in term of number of steps.

Access Paths

- Possible access paths for selections:
  - Scan the file records.
  - Scan the index
- Selectivity of an access path is the number of block I/Os needed to retrieve the tuples satisfying the desired condition.
  - Obviously, we want to use the most selective access path (with the fewest page/block I/Os).
- Access path using index may not be the most selective!
  - Why not? ($\sigma_{bid \neq 10}$ Reserves, non-clustering index on bid).

Selection Operation

- Notation: $\sigma_p(r)$
- p is the selection predicate
- Defined by:

$$\sigma_p(\mathbf{r}) = \{t \mid t \in r \textbf{ and } p(t)\}$$

in which p is a formula of propositional calculus of terms connected by: $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
Each term is of the form:

<attribute> *op* [<attribute> or <constant>]

where *op* can be one of: $=, \neq, >, \geq. <. \leq$

- Selection example:

$\sigma_{\textit{branch-name='Perryridge'}}(\textit{account})$

Evaluation of Selection Operation

- **File scan** – search algorithms that scan files and retrieve records that fulfill a selection condition.
  - blocks of a relation are stored contiguously
- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.

Algorithm for Selection Operation (File scan: *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 (Linear Search, Equality on Key)
    - Average cost = (b_r/2) block transfers + 1 seek
  - This linear search can be always applied, regardless of:
    - selection condition or
    - ordering of records in the file, or
    - availability of indices

Xi'an Jiaotong-Liverpool University 西交利物浦大學

Algorithms for Selection Operation (File scan: *binary search*)

- Applicable only if the selection is an equality comparison on the attribute on which file is ordered.
  - Cost estimate (number of block transfers and seeks):
    - cost of locating the first tuple by a binary search on the blocks
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - If there are multiple records satisfying selection
      - Add transfer cost of the number of blocks containing records that satisfy selection condition
      - Will see how to estimate this cost later
  - If b_r is not too big, then most likely binary search doesn't pay.
    - Note that t_T is several (say, 40) times smaller than t_S
    - Which of the two algorithms needs to be wisely chosen for a specific query at hands.

Selections Using Indices (*primary index on candidate key, equality*).

- Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$

  where h_i denotes the height of the index
- Recall that the height of a B+-tree index is at most $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$, where n is the number of pointers per node and K is the number of search keys.
  - E.g. for a relation r with 1,000,000 different search key, and with 100 index entries per node, $h_i = 4$
  - Unless the relation is really small, this algorithms always "pays" when indexes are available.

Selections Using Indices (*primary index on nonkey, equality)*

- Retrieve possibly 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 (*equality on key and non-key of secondary index)*

- 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 at most is: $(h_i + n) * (t_T + t_S)$
  - Can be very expensive if n is big! Note that it multiplies the time for seeks by n.

Comparative Selections $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$

- Using a linear file scan or binary search just as before
- Using primary index, comparison
  - 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 $> v$;
    - Using the index would be useless, and would requires extra seeks on the index file.
- Using secondary 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 for each record (a lot!)
    - Linear file scan may be much cheaper!!!!

Conjunctive Selections $\sigma_{\theta 1 \wedge \theta 2 \wedge \cdots \wedge \theta n}(r)$

- *using one index*
  - Select θ_i and previous algorithms that results in the least cost for $\sigma_{\theta i}(r)$.
  - Test other conditions on tuple after fetching it into memory buffer.
  - In this case the choice of the first condition is crucial!
    - One must use estimates to know which one is the best.
- *using multiple-key index*
  - Use appropriate composite (multiple-key) index if available.

Disjunctive Selections

$\sigma_{\theta 1 \vee \theta 2 \vee \cdots \vee \theta n}(r)$

- Union of identifiers
  - Use linear scan.
  - Or index scan if *some* conditions have available indices.
    - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
    - Then fetch records from file

Selections With Negation: $\sigma_{\neg\theta}(r)$

- Use linear scan on file
- If an index is applicable to θ
  - Find satisfying records using index
  - $\sigma_{\neg\theta}(r)$ is simply the set of tuples in r that are not in $\sigma_{\theta}(r)$

Duplicate elimination

- Duplicate elimination can be implemented via hashing or sorting.
  - On sorting, duplicates will come adjacent to each other, duplicates can be deleted.
    - *Optimisation:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge (details come later)
  - Hashing is similar – duplicates will come into the same bucket.

Evaluating Projection

- Projection drops columns not in the selected attribute list.
- The expensive part is removing duplicates.

Sorting

- Sorting algorithms are important in query processing at least for two reasons:
  - The query itself may require sorting (**order by** clause)
  - Some algorithms for other operations, like join, set operations and aggregation, require that relations are previously sorted and duplicates removed.
- To sort a relation:
  - We may build an index on the relation, and then use the index to read the relation in sorted order.
    - This only sorts the relation logically, not physically
    - May lead to one disk block access for each tuple.
  - For relations that fit in memory sorting algorithms that you've studied before, like quicksort, can be used.

External Sort-Merge

- For relations that don't fit in memory, special algorithms that take into account the measures in terms of block transfers and seeks, are required.
- Let *M* denote memory size: the number of disk blocks whose contents can be buffered in main memory.
- **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 file R_i;
  (d) Increment *i*.
  Let the final value of *i* be *N* (that is we have *N* run files)
- *Next step: merge the runs (next slide).....*

External Sort-Merge cont'd

Merge the runs (N-way merge). We assume (for now) that $N < M$.

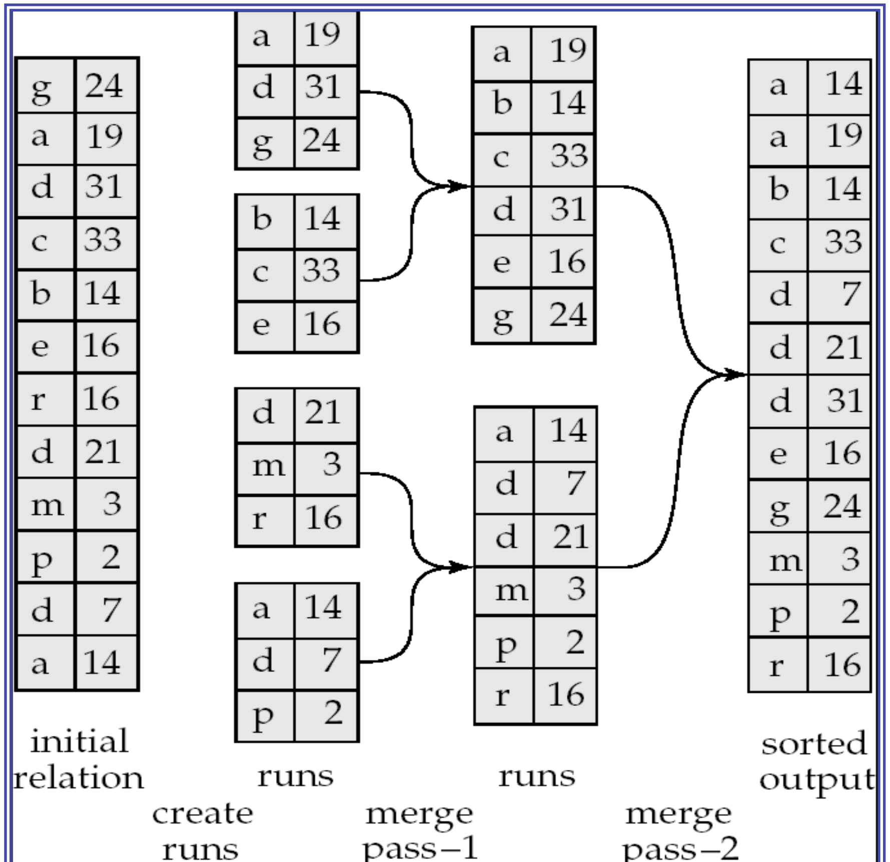
1. Use N blocks of memory to buffer input runs (1 for each of the N run files), and 1 block to buffer output. Read the first block of each run into memory.
2. **repeat**
    1. Select the first record in sort order (i.e., the smallest) among all buffer pages
    2. Write the record to the output buffer. 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.
    - For example, 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.

Example: External Sorting Using Sort-Merge

Suppose that Memory holds at most three blocks, only one tuple fits in a block. The relation needs 12 blocks to store.



External Merge Sort (Cost analysis)

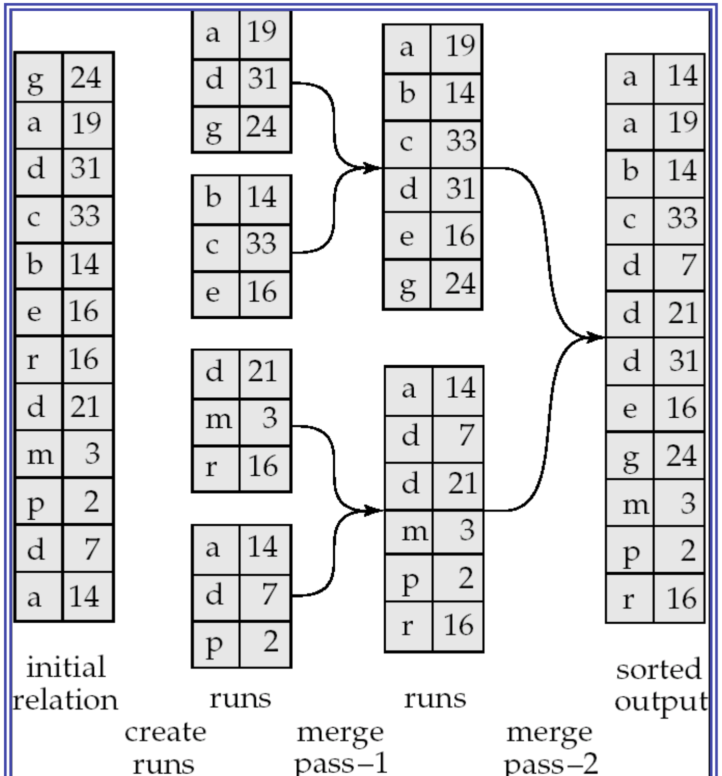
- Assume relation is stored in b_r blocks, M is the memory size, so the number of run file b_r/M.
- Buffer size b_b (read b_b blocks at a time from each run and b_b blocks for writing).
- Cost of Block Transfer
  - Total number of merge passes required: $\lceil \log_{\lfloor M/bb \rfloor -1}(b_r/M) \rceil$. Each time can merge $\lfloor (M-b_b)/b_b \rfloor$.
  - Block transfers for initial run creation as well as in each pass is $2b_r$ (read/write all b_r blocks)
    - for final pass, we don't count write cost
    - Thus total number of block transfers for external sorting:

      $$2b_r + 2b_r \lceil \log_{\lfloor M/bb \rfloor -1}(b_r / M) \rceil - b_r = b_r (2 \lceil \log_{\lfloor M/bb \rfloor -1}(b_r / M) \rceil + 1)$$
- 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
    - Total number of seeks:

      $$2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{\lfloor M/bb \rfloor -1}(b_r / M) \rceil -1)$$

Example: External Sorting Using Sort-Merge

Suppose that Memory hold at most three blocks, only one tuple fits in block. The relation needs 12 blocks to store, m= 3, *assume* b_b=1.



Total block transfer = ?

Total seeks = ?

End of Lecture

- Summary
  - Basic Steps in Query Processing
  - Query costs
  - Algorithms for evaluating relational algebra operations
    - Selection
    - Projection
    - Merge-sorting
- Reading
  - 6th edition, Chapters 12.1, 12.2., 12.3, 12.4
  - 7th edition, Chapters 15.1, 15.2., 15.3, 15.4