Principles of Database Systems (CS307)

Lecture 12: Query Processing

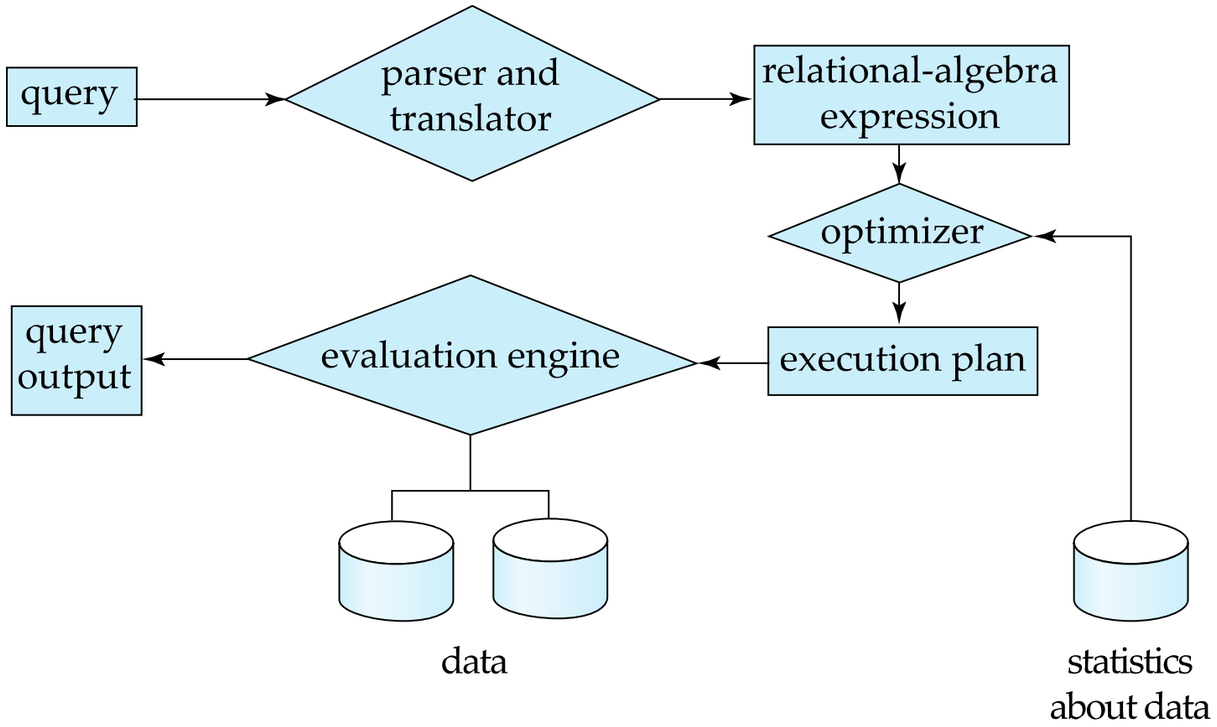
Yuxin Ma

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.

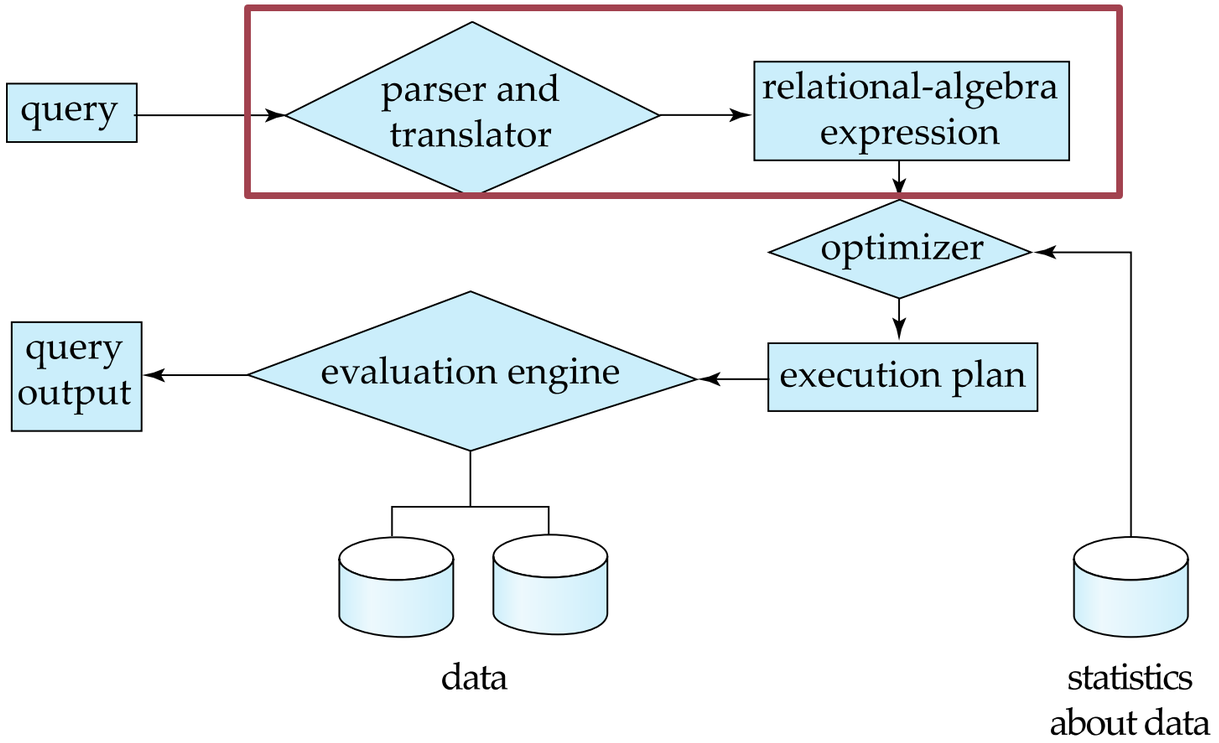
Basic Steps in Query Processing

- Parsing and Translation
- Optimization
- Evaluation



Basic Steps in Query Processing

- Parsing and Translation
  - Translate the query into its internal form
    - The internal form is then translated into relational algebra
  - Parser checks syntax and verifies relations



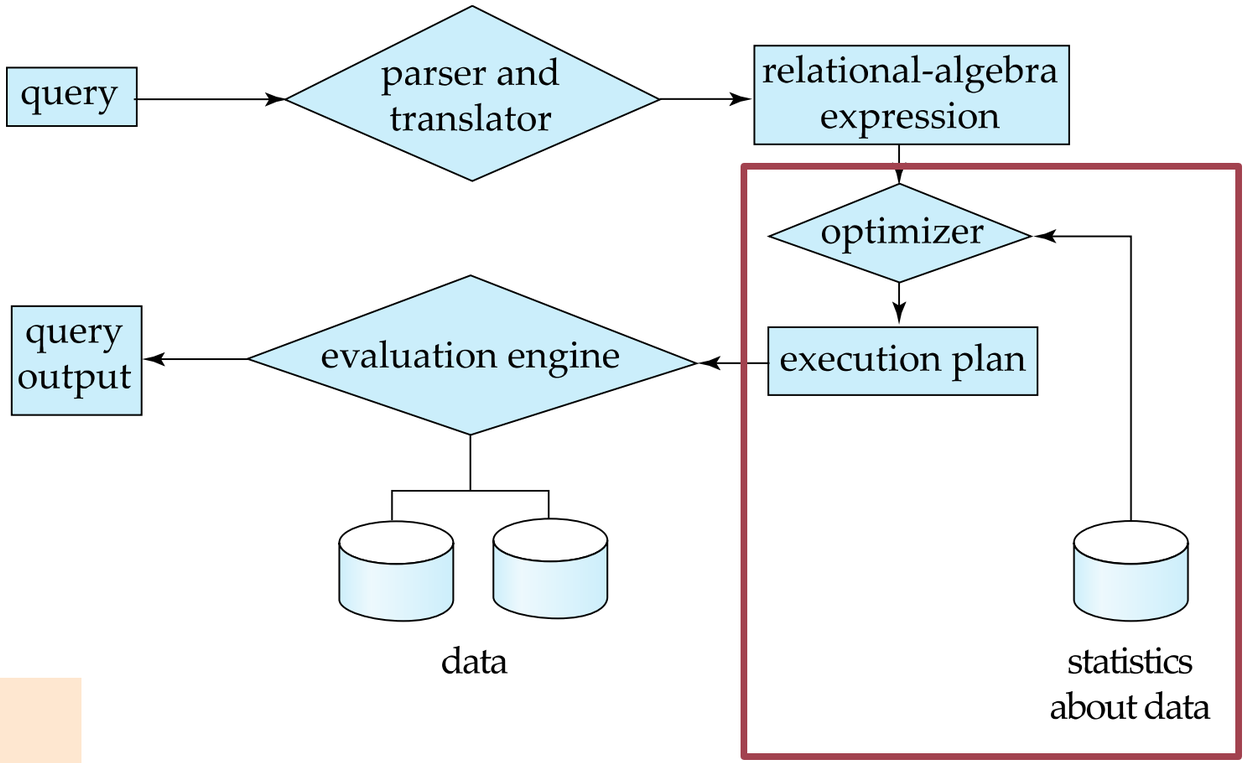
Basic Steps in Query Processing

- Optimization
  - A relational algebra expression may have many equivalent expressions
  - E.g.,

$$\sigma_{salary<75000}(\prod_{salary}(\text{instructor}))$$

is equivalent to

$$\prod_{\text{salary}}(\sigma_{\text{salary}<75000}(\text{instructor}))$$



But the number of rows involved in the projection operation may be (significantly) smaller in the second expression

Basic Steps in Query Processing

- Optimization
  - A relational algebra expression may have many equivalent expressions
    - … and each relational algebra operation can be evaluated using one of several different algorithms
  - *Correspondingly, a relational-algebra expression can be evaluated in many ways*

Basic Steps in Query Processing

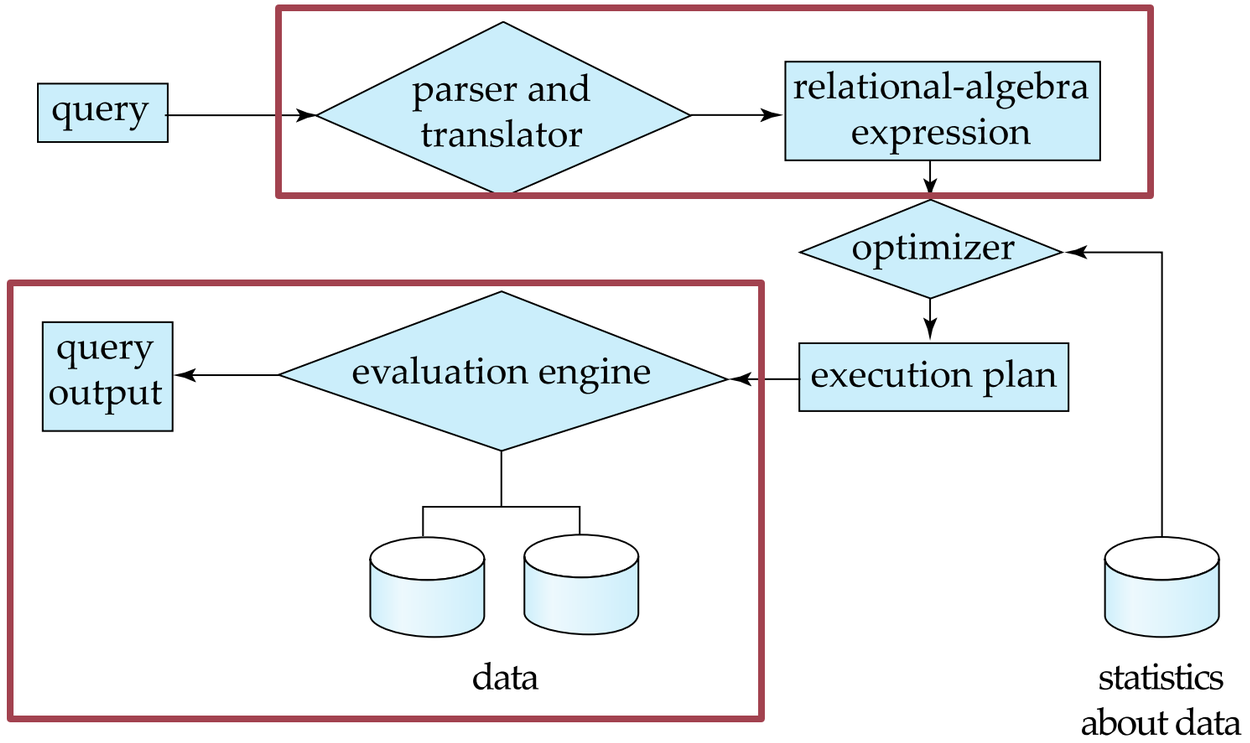
- Optimization
  - **Evaluation Plan**: Annotated expression specifying detailed evaluation strategy
  - E.g.,:
    - Use an index on `salary` to find instructors with `salary<75000`
    - Or perform complete relation scan and discard instructors with `salary<75000`

Query Optimization: Choose the one with the lowest cost among all equivalent evaluation plans

  - Cost can be estimated using statistical information from the database catalog
    - E.g., Number of tuples in each relation, size of tuples, etc.

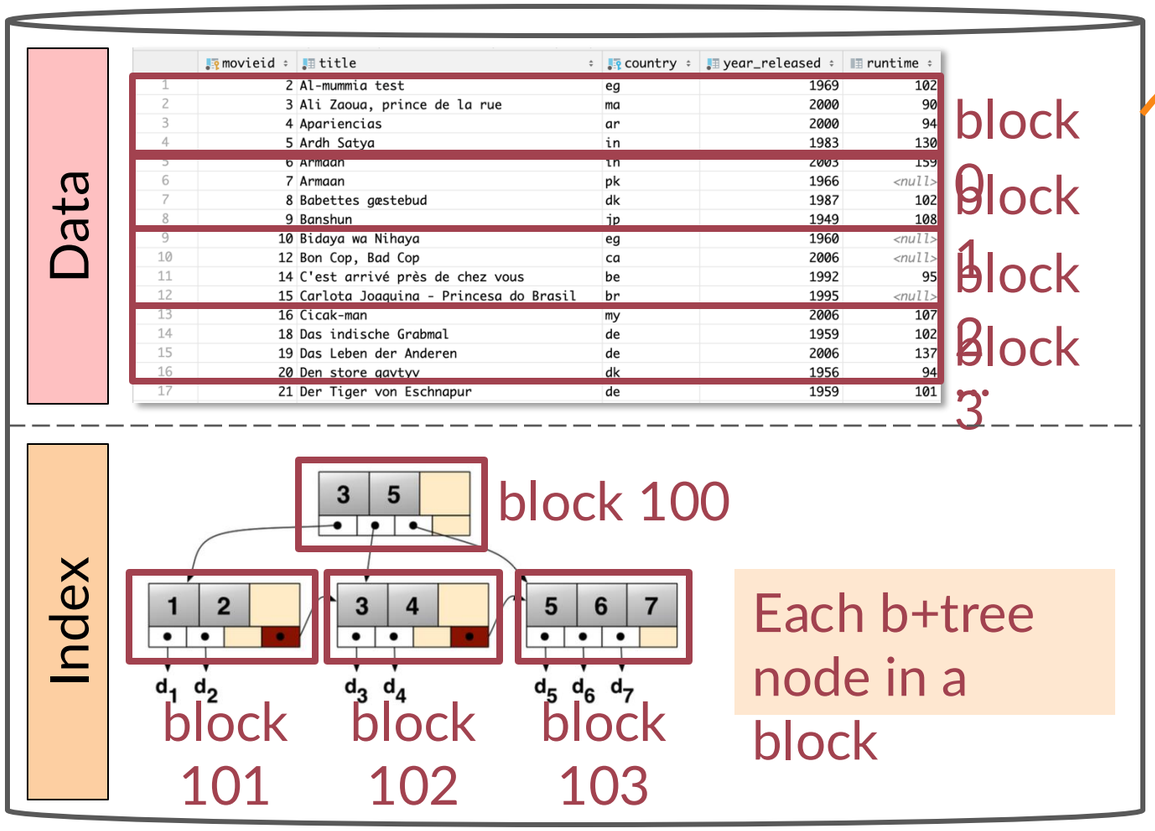
Basic Steps in Query Processing

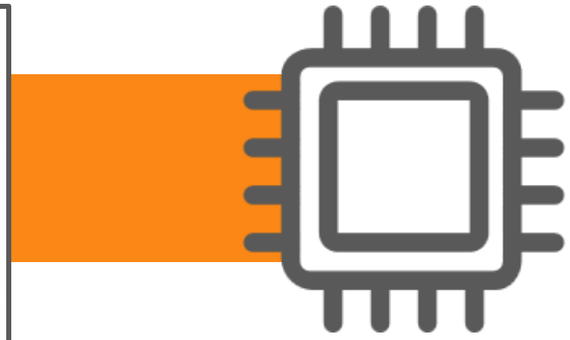
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query



Prerequisite

- Storage model



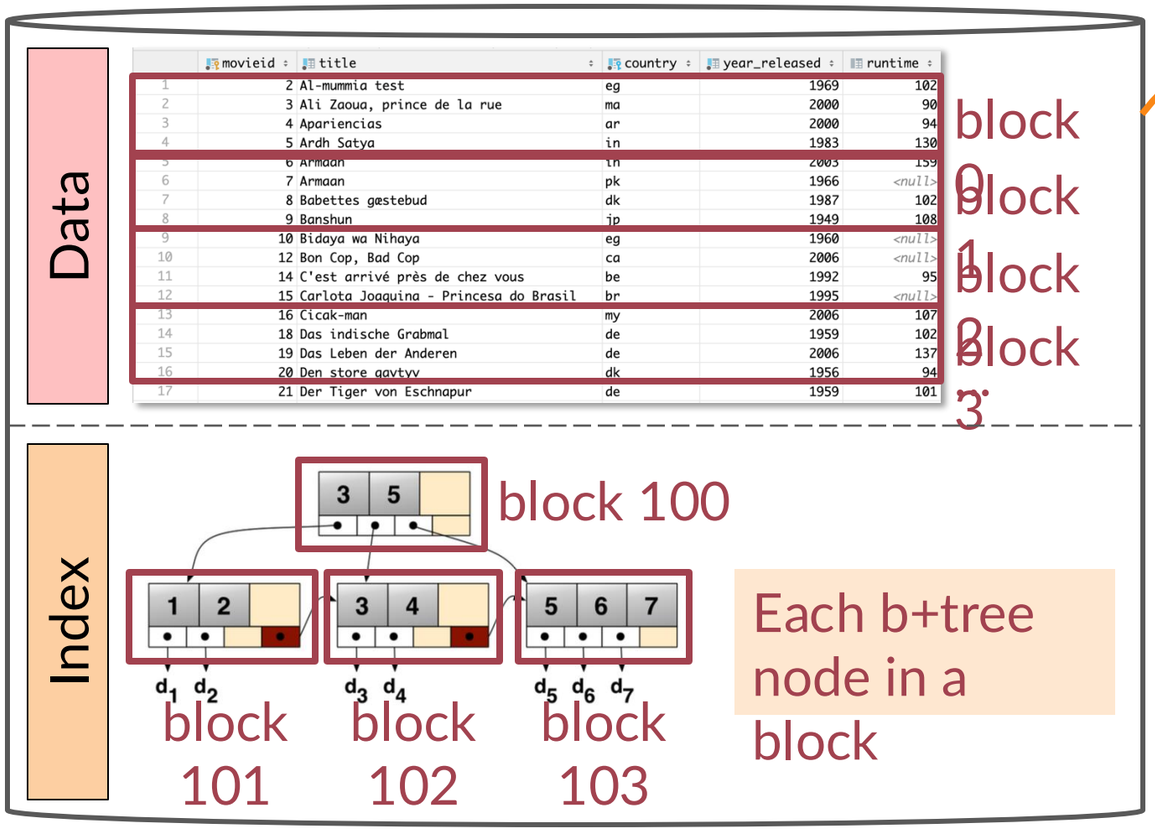


Prerequisite

- Storage model

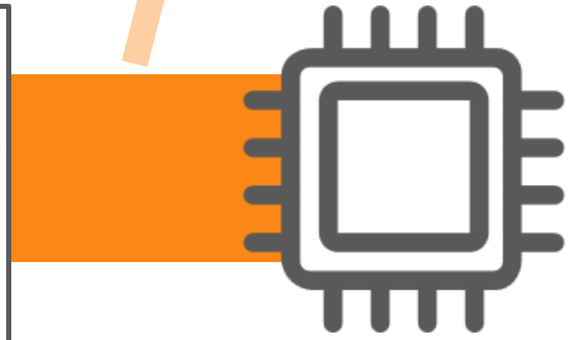
- Relatively small bandwidth
  - 100MB/s ~ <10GB/s
- High latency
  - Millisecond-level

- Very large bandwidth
  - 94GB/s (for DDR4 2933*)
- Very low latency
  - Nanosecond-level



HDD/SSD

Main Memory



CPU (Cache, Register)

*
https://www.intel.com/content/www/us/en/support/articles/000056722/processors/intel-core-processors.html

Prerequisite

- Measuring query cost
  - Disk cost can be estimated as:
    - Number of seeks * average-seek-cost
    - Number of blocks read * average-block-read-cost
    - Number of blocks written * average-block-write-cost
  - For simplicity, we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
    - t_T – time to transfer one block
      - Assuming for simplicity that write cost is same as read cost
    - t_S – time for one seek
  - E.g., cost for b block transfers plus S seeks
    $b * t_T + S * t_S$

Prerequisite

- Measuring query cost
  - t_S and t_T depend on where data is stored. With 4 KB blocks:
    - High end magnetic disk: t_S = 4 millisec and t_T =0.1 millisec
    - SSD: t_S = 20-90 microsec and t_T = 2-10 microsec for 4KB
  - Required data may be buffer resident already, avoiding disk I/O
    - But hard to take into account for cost estimation
  - Worst case estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
    - But more optimistic estimates are used in practice
  - We ignore CPU costs for simplicity
    - Real systems do take CPU cost into account
    - Network costs must be considered for parallel systems

Overview

- Selection
- Joining

Selection Operation

- Let’s start from this simple query:

```
select * from movies where [CONDITION];
```

  - If you are the designer of the database engine, what do you think is the best way to fulfill this requirement?
  - Two factors to consider:
    - What comparison is it in the CONDITION (equality / comparison)?
    - Does the column involved in the CONDITION have an index?

Basic Linear Scan

- Linear Search (displayed as `Seq Scan` in PostgreSQL)
  - 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*
- Linear search can be applied regardless of
  - Selection condition
  - Ordering of records in the file
  - Availability of indices

Basic Linear Scan

- However, a full-table linear scan on extremely-large tables can be a disaster
  - E.g., billions of records in database
  - That’s why we need other optimized ways

Index Scan

- **Index scan** – Search algorithms that use an index
  - Selection condition must be on search-key of index

```
select * from movies where movieid = 125;
```

We have a B+ tree index on `movieid`

- Plan: `Index Scan`

```
select * from movies where runtime = 100;
```

We don't have any index on `runtime`

- Plan: `Seq Scan`

Index Scan

- **Index scan** – Search algorithms that use an index
  - Selection condition must be on search-key of index

- Unlike linear scan, we need to talk about different types of indexes and CONDITIONs
  - Clustered / Non-clustered index (Primary / Secondary index)
  - Equality / Comparison test

Index Scan

h_i: height of the B+-tree

Clustered index, equality on key

- Retrieve a single record that satisfies the corresponding equality condition
  - *key => no duplicated values*
  - *Cost* = $(h_i + 1) * (t_T + t_S)$

Clustered index, equality on non-key

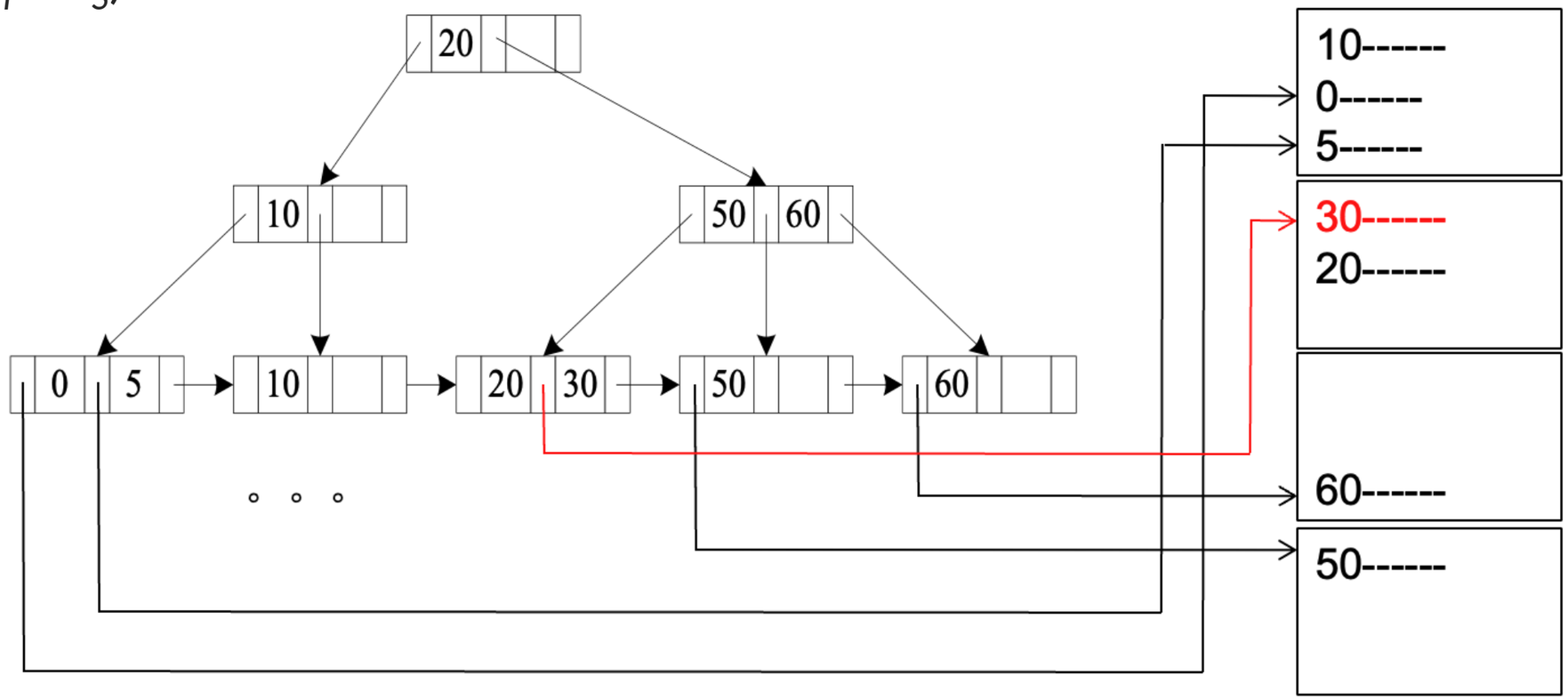
- Retrieve multiple records
  - *non key attributes => possible to have duplicated values*
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - *Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$

Index Scan

h_i: height of the B+-tree

Secondary index, equality on key/non-key

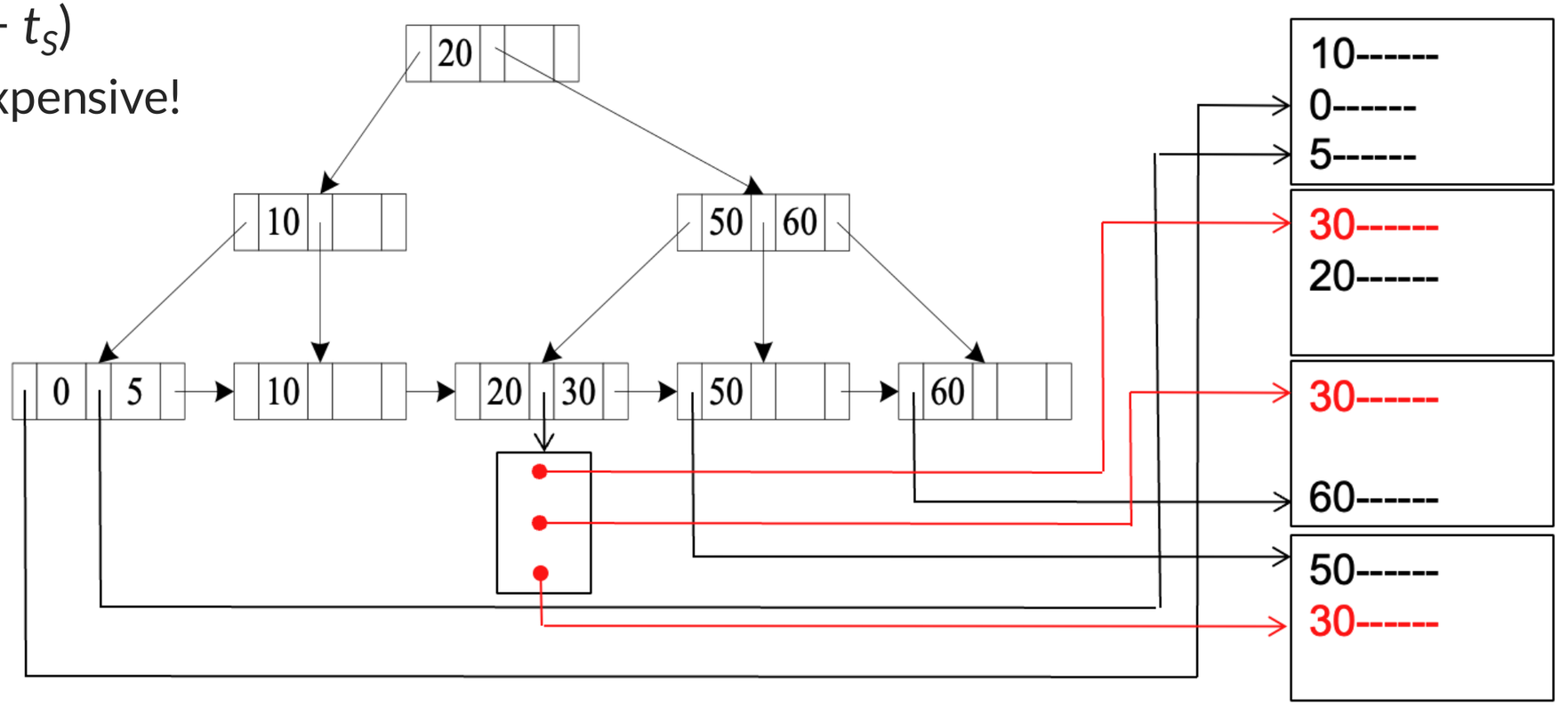
- Retrieve a single record if the search-key is a candidate key
  - *Cost* = $(h_i + 1) * (t_T + t_S)$



Index Scan

Secondary index, equality on key/non-key

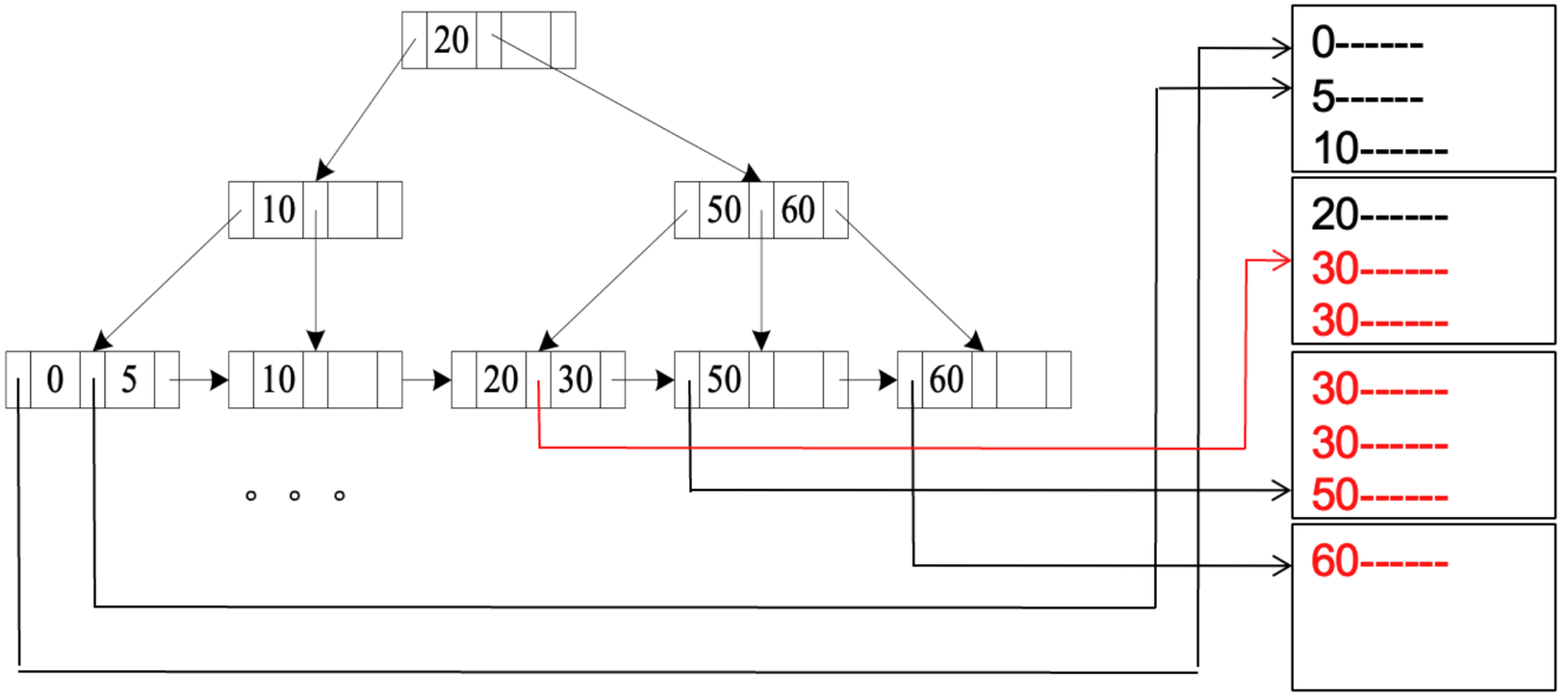
- Retrieve multiple records if search-key is not a candidate key
  - Each of *n* matching records may be on a different block
  - Cost = $(h_i + n) * (t_T + t_S)$
    - Can be very expensive!



Index Scan

Tip: Comparison tests can always be fulfilled with *linear scans*, which is the fallback solution

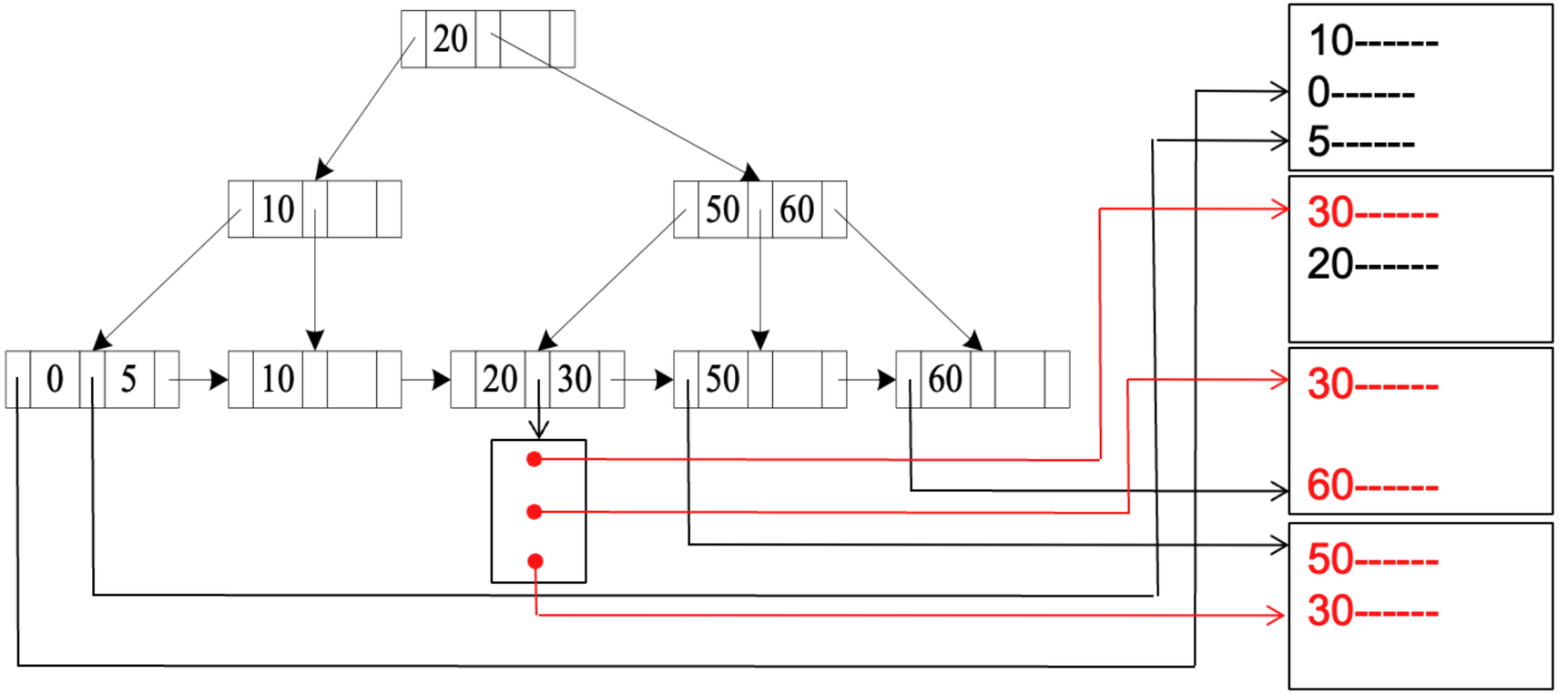
- **Clustered index, comparison** (i.e., Relation is sorted on *A*)
  - For $\sigma_{A \geq V}(r)$, use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$, just scan relation sequentially till first tuple > *v*; do not use index



Index Scan

- **Non-clustered 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, retrieving records that are pointed to requires an I/O per record
- Linear scan may be cheaper!



Complex Selections

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

- **Conjunctive selection using single-key index(es)**
  - Select a combination of θ_i and algorithms mentioned above that results in the least cost for $\sigma_{\theta i}(r)$
  - Test other conditions on tuple after fetching it into memory buffer
- *** Conjunctive selection using multi-key index**
  - Use appropriate composite (multiple-key) index if available
- **Conjunctive selection by intersection of identifiers**
  - Requires indices with record pointers (Here, "indices" means the order number of records in the files, like array indices. They are not indexes.)
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers
  - Then fetch records from file

Complex Selections

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

Disjunctive selection by union of identifiers

(Similar to the third way on the previous page)

- Applicable if *all* conditions have available indices
  - Otherwise, just 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

Bitmap Index Scan

- It bridges the gap between using secondary index scan and linear scan
  - Bitmap with 1 bit per page in relation
  - Steps:
    - Index scan used to find record ids, and set bit of corresponding page in bitmap
    - Linear scan fetching only pages with bit set to 1
  - Performance
    - Similar to index scan when only a few bits are set
    - Similar to linear file scan when most bits are set
    - Never behaves very badly compared to best alternative

How `join` Works (with the help of indexes)

- Some widely-used `join` algorithms
  - Nested-loop join
  - Indexed nested-loop join
  - Merge join

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 • t_s to the result.
   end
end
```

- *r* is called the **outer relation** and *s* the **inner relation** of the join
  - Think about the “outer loop” and the “inner loop” in programming

- Requires no index 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

- 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

Actually, we have an even worse solution: For each loop, we read the two corresponding inner and outer blocks from scratch, which is even slower than $n_r * b_s + b_r$

- 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

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.

Indexed Nested-Loop Join

- Worst case: Buffer in memory has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s*

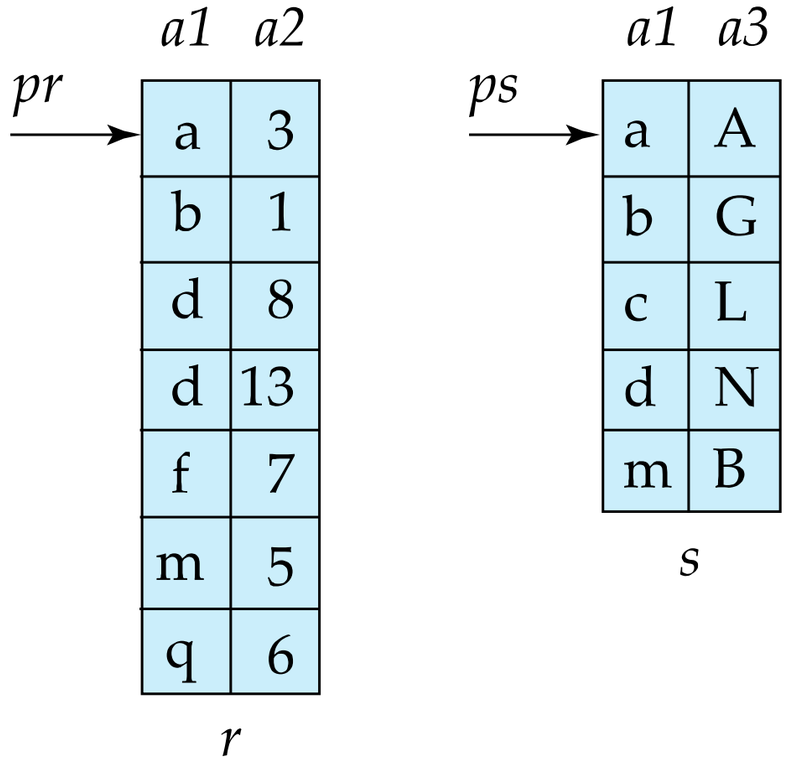
- Cost of the join: $b_r\,(t_T + t_S) + n_r * c$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition

- If indices are available on join attributes of both *r* and *s*, use the relation with fewer tuples as the outer relation

Merge Join

- a.k.a. Sort-merge join
  - Zipper-like joining
- Steps
  - Sort both relations on their join attribute (if not already sorted on the join attributes)
  - Merge the sorted relations to join them

- Join step is similar to the merge stage of the sort-merge algorithm
  - Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched



Merge Join

- 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

- 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

  b_b: the memory buffer size, counted in number of blocks

Hash Join

- Hash join
  - Build a set of buckets for a smaller table to speed up the data lookup

- Procedure:
  - 1. Create a hash table for the smaller table `t1` in the memory
  - 2. Scan the larger table `t2`. For each record `r`,
    - 2.1 Compute the hash value of `r.join_attribute`
    - 2.2 Map to corresponding rows in `t1` using the hash table