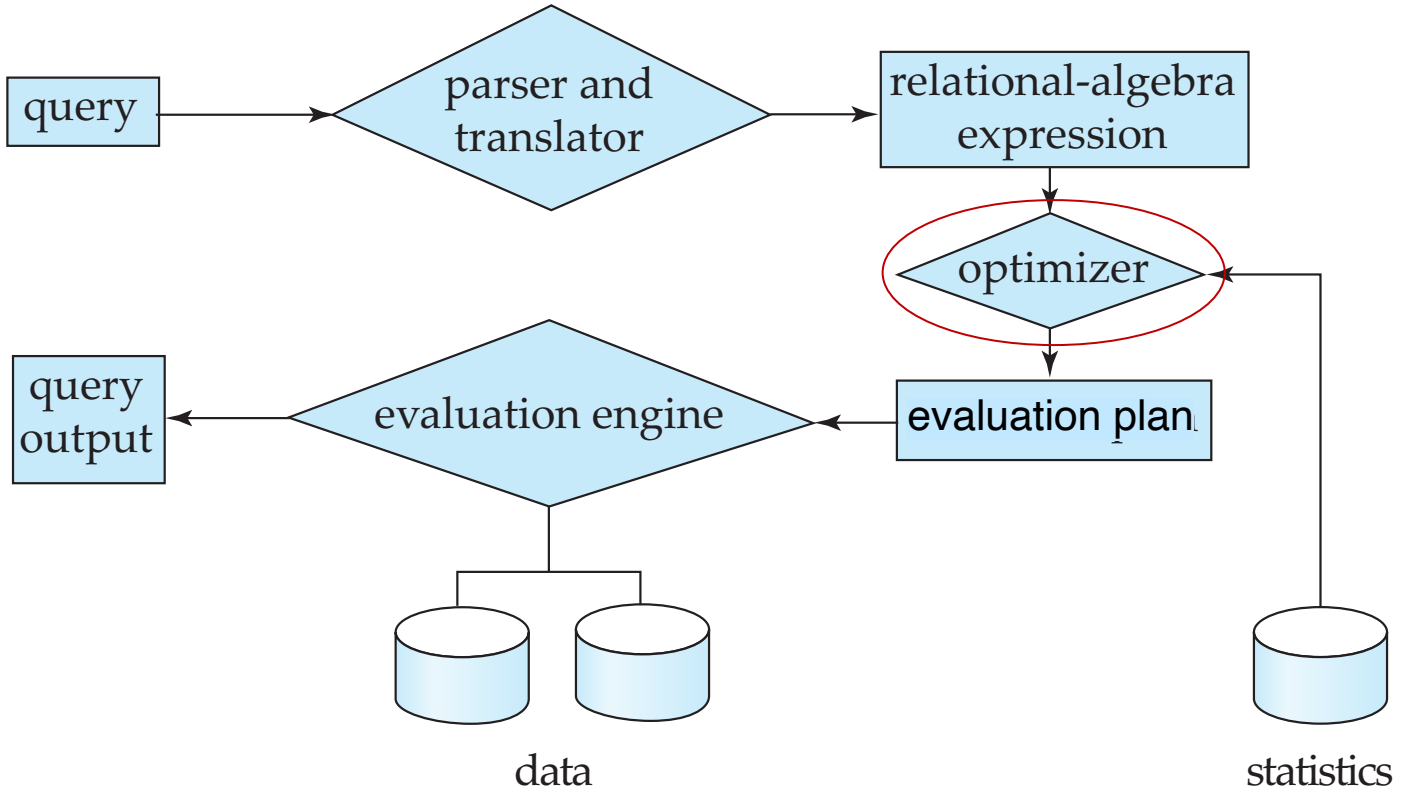
Query Processing

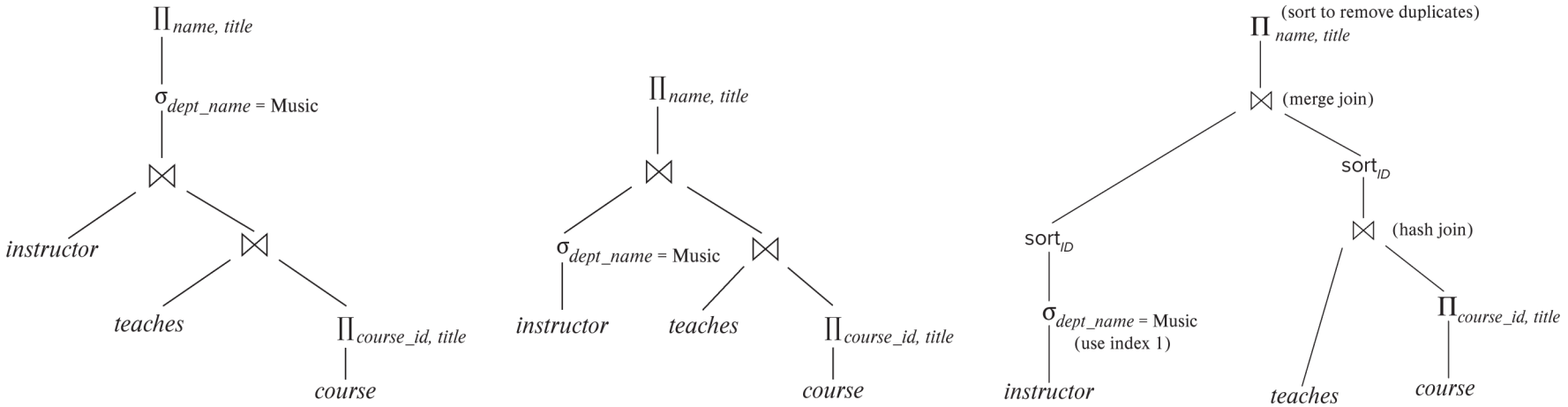
Basic Steps in Query Processing

1. Parsing and translation
   - Parser to check syntax, verify relation names, etc.
   - Translate query into internal form (~ relational algebra)
2. Optimization
   1. Equivalent expression trees
   2. Best evaluation plan
3. Evaluation



Query optimization

- Different algorithms can be used to evaluate the same relational algebra operation
  - Evaluation primitive: A relational algebra operation annotated with instructions on how to evaluate it
  - **Evaluation plan**: Sequence of primitives to evaluate a query
    - Feed to the *query-evaluation engine*



Query Processing

- Query processing
  - Measures of Query Cost
- Algorithms for:
  - Select Operation
  - Sorting
  - Join Operation
  - Other Operations
- Choosing an Evaluation Plan
- Evaluating a Plan

Query Processing: Measures of Cost

- **Query Optimization**: Among all *equivalent evaluation plans*, choose the one with lowest cost.
  Cost is estimated using statistical info. from DB catalog
- **Cost** in terms of *disk access (I/O), CPU*, and network *communication*
  With HDD, I/O cost dominated CPU cost. Not longer the case with SDDs and large RAM.
- For simplicity, we only annotate I/O cost:
  Real systems do use CPU cost, and, in parallel DBs, network costs are basic
  - **No. blocks transferred** (b): Assuming write cost=read cost
    - t_T: block-transfer time
  - **No. random I/O accesses** (S): disk seek
    - t_S: block-access time (disk seek time + rotational latency)
  - Cost of an operation that transfer *b* blocks and performs *S* accesses:

$$b * t_T + S * t_S$$

Estimated on DBMS installation or *specified* by users:
HDD: t_S = 4 ms ; t_T = 0.1 ms
SSD: t_S = 20-90 µs ; t_T = 2-10 µs

ALGORITHMS

Algorithms for Select Operation

- **File scans**: search algorithms to explore relation, and locate and retrieve records fulfilling a selection condition.

- **Linear search** [Algorithm **A1**] Scan each file block and check the selection condition on all the records.
  - Cost estimate = $b_r * t_T + 1 * t_S$
    - b_r denotes no. blocks containing records from relation *r*
    - Assumes contiguous blocks (otherwise, no. accesses (S) > 1)
  - If selection is on a key attribute, stop once record is found
    - Cost estimate = $(b_r/2) * t_T + 1 * t_S$
  - Linear search can be ***applied to any file***

Algorithms for Select Operation

- Retrieve record(s) that satisfies an equality condition, $\sigma_{A=V}(r)$
- **Search with clustering index** (equality) **[A2]**
  - $Cost = h_i * (t_T + t_S) + b * t_T + t_S$
    where b is no. blocks (contiguous) with records with the specified search-key
    and h_i is the depth of the index tree ($h_i=1$ to denote that the index is in memory)

In both cases:
If index's search-key is a candidate *key of the relation*, result is a single record:
$Cost = (h_i + 1) * (t_T + t_S)$

- **Search with secondary index** (equality) **[A3]**
  - $Cost = (h_i + b) * (t_T + t_S)$ >>> Can be very expensive! <<<
    where b is no. blocks (non-contiguous) with records with the specified search-key
    and h_i is the depth of the index tree ($h_i=1$ to denote that the index is in memory)

Algorithms for Select Operation

- Retrieve record(s) that satisfies a comparison, $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$
- **Search with clustering index** (range) **[A4]**
  - $\sigma_{A \geq V}(r)$: use index to find first **index entry** $\geq v$ and go on sequentially on the **file**
  - $\sigma_{A \leq V}(r)$: scan **file** sequentially until a record $> v$.
    - $Cost = h_i * (t_T + t_S) + b * t_T + t_S$ (same as before)

Index not used!!

- **Search with secondary index** (range) **[A5]**
  - $\sigma_{A \geq V}(r)$: use index to find first **index entry** $\geq v$, scan **index** sequentially from there, and retrieve pointers to records
  - $\sigma_{A \leq V}(r)$: scan index's leaves until first entry $> v$, to find pointers to records and retrieve pointed records
    - $Cost = (h_i + b) * (t_T + t_S)$ (same as before)

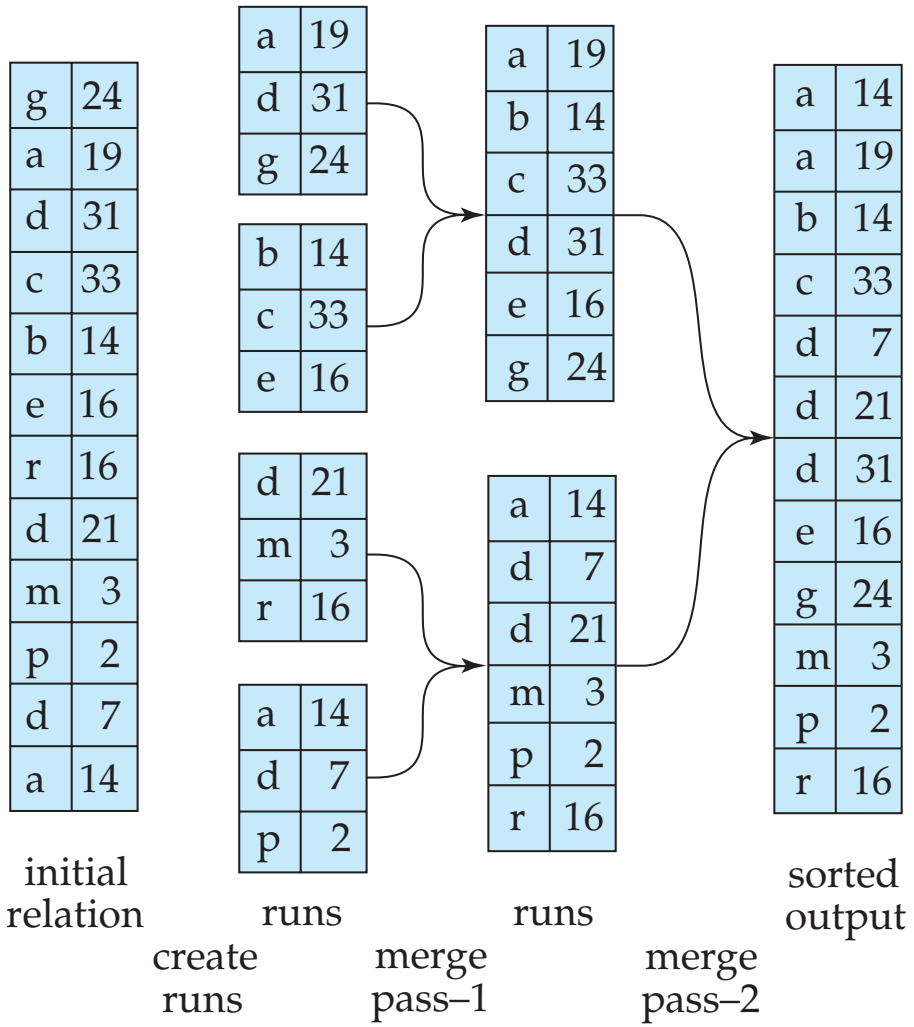
>>> Can be very expensive! <<<

Algorithms for Sorting

- **Sorting** is important because:
  - Queries can be requested to be ordered
  - Certain operations benefit from ordered input relations
- Alternative: build an index on the sorting attribute, and use index to read the relation in sorted order.
  - Inefficient: logical order, but not physical order!
    Might lead to one disk access per tuple
- Two situations:
  - Relations that fit in memory: use standard sorting methods
    E.g., Quicksort
  - If relation don't fit in memory, use **external sorting**:
    - **External sort-merge**, most commonly used technique

External Sort-Merge Sorting

- Two steps:
  1. Divide the relation into small ordered pieces (called **runs**)
  2. Combine (merge) the runs in a sorted way



External Sort-Merge Sorting

1. Create sorted runs:
Repeat until the end of the relation (i=1..N):

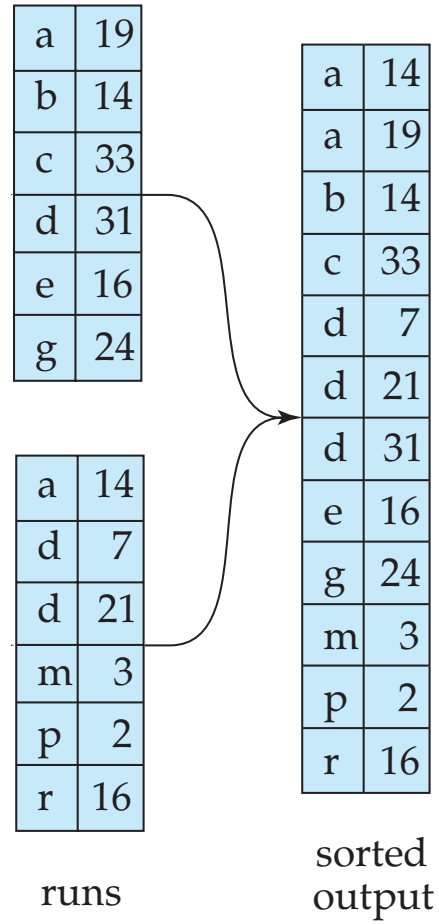
1. Read M blocks of relation into memory
2. Sort the in-memory blocks
3. Write sorted data to *run* R_i

$M \equiv$ no. blocks (buffer size)

2. Merge the runs: when $N < M$,

$N \equiv$ no. runs

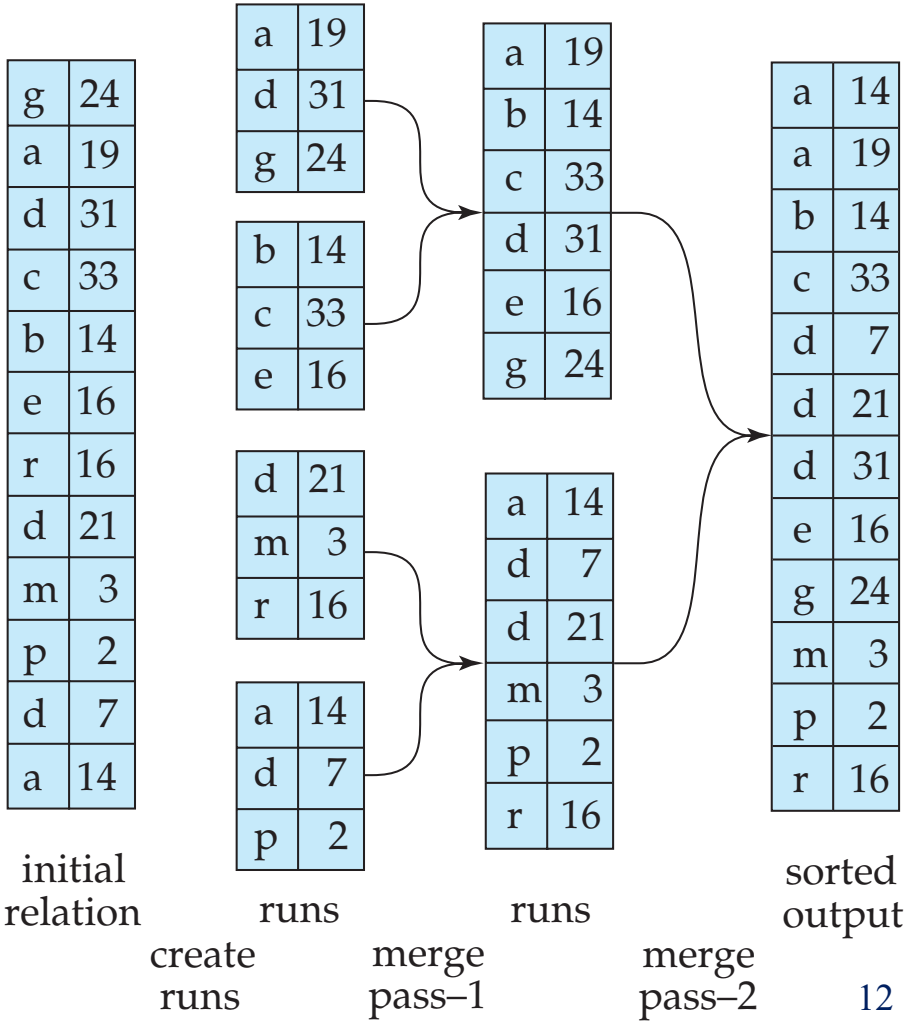
1. Use N buffer blocks for input runs, and 1 block for output
2. Read the first block of each run into its buffer block
3. **Repeat until** all input buffer blocks are empty
   1. Select from buffer the *first record in sort ordering*
   2. Delete it from its buffer input block, and write it to the output buffer
   3. **If** *output buffer* is full **then** write it to disk and free output buffer.
   4. **If** an *input buffer* block is empty **then** read next block (if any) from its run



External Sort-Merge Sorting

2. Merge the runs: when $N \geq M$, need several merging passes:

1. In each pass, contiguous groups of *M-1* runs are merged.
   - Each pass reduces the number of runs by a factor of *M-1*.
     E.g. If *M=11*, and there are 90 runs, first pass only reduces the number of runs to 9
2. Passes are repeatedly performed until only one run is left.



Cost of External Sort-Merge

- Cost of *block transfers*:
  - First step needs $2b_r$ block transfers (b_r: no. blocks in relation)
    - Load all blocks and write them into runs. Initial no. runs: $\lceil b_r/M \rceil$
  - Use buffers of b_b blocks per run to reduce no. accesses
    - Merges $\lfloor M/b_b \rfloor - 1$ runs in each pass
  - No. passes required: $\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_r/M) \rceil$
  - Total no. block transfers:

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

- Cost of *seeks*:
  - First step: 2 random I/O accesses per run (read and write), $2\lceil b_r/M \rceil$
  - Second step: Need $2\lceil b_r/b_b \rceil$ seeks per merge pass
  - Total no. random I/O accesses :

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

Algorithms for Join Operation

- To implement ***equi-joins*** ($r \bowtie_{r.A=s.B} s$), several algorithms:
  1. Nested-loop join
  2. Block nested-loop join
  3. Indexed nested-loop join
  4. Merge-join
  5. Hash-join
- We use cost estimates to choose an algorithm

Running example: *student* $\bowtie$ *takes*

	student	*takes*
No. records:	5,000	10,000
No. blocks:	100	400

Algorithms for Join: 1. Nested-Loop

- Compute *theta join*: $r \bowtie_\theta s$

Natural join: also drops repeated attribute(s)

```
for each tuple t_r in r
   for each tuple t_s  in s
     if pair (t_r, t_s) satisfies condition θ then
        result +=  (t_r • t_s)                    >> Concatenate t_r and t_s
     end if
   end for
end for
```

- *r* and *s* are called **outer** and **inner relation**, resp.
- No index required.
- Works with any kind of join condition, θ.
- Checks every pair of tuples: Expensive!!!

Algorithms for Join: 1. Nested-Loop

- Cost:
  - **Worst case**: only one block per relation fits in memory
    no. block transfers: $n_r * b_s + b_r$; no. I/O accesses: $n_r + b_r$
    where n_r and b_r are the no. records and no. blocks of relation r (same for s)
    - E.g., with *student* as **outer relation**:
      5000 ∗ 400 + 100 = 2,000,100 block transfers
      5000 + 100 = 5100 I/O accesses
      with *takes* as **outer relation**:
      1,000,400 block transfers
      10,400 I/O accesses

Order matters!
To decide the order, consider times t_T & t_S

  - **Best case**: both relations fit completely in memory
    no. block transfers: $b_s + b_r$; no. I/O accesses: *2*
    - If **only one** relation fits in memory, **use it as inner relation** to obtain *best-case* cost
    - E.g., If *student* fits in memory:
      500 block transfers
      2 I/O accesses

Algorithms for Join: 2. Block Nested-Loop

- If no relation fits in memory, use a per-block basis:
  - Every block of *inner relation* is paired with every block of *outer relation*

```
for each block B_r of r
    for each block B_s of s
        for each tuple t_r in B_r
            for each tuple t_s in B_s
                if pair (t_r, t_s) satisfies condition θ then
                    result += (t_r • t_s)          >> Concatenate t_r and t_s
                end if
            end for
        end for
    end for
end for
```

Variant of *nested-loop join*:
Shares most of its properties

Algorithms for Join: 2. Block Nested-Loop

- Cost:
  - **Worst case**: only one block per relation in memory
    no. block transfers: $b_r * b_s + b_r$; no. I/O accesses: $2 * b_r$
    - More efficient if the *smaller relation* is used as *outer relation*
    - E.g., with *student* as outer relation:
      100 ∗ 400 + 100 = 40,100 block transfers
      2 ∗ 100 = 200 I/O accesses
  - **Best case**: at least the smaller relation fits completely in memory
    no. block transfers: $b_s + b_r$; no. I/O accesses: 2
    - E.g., If *student* fits in memory: 500 block transfers + 2 I/O accesses

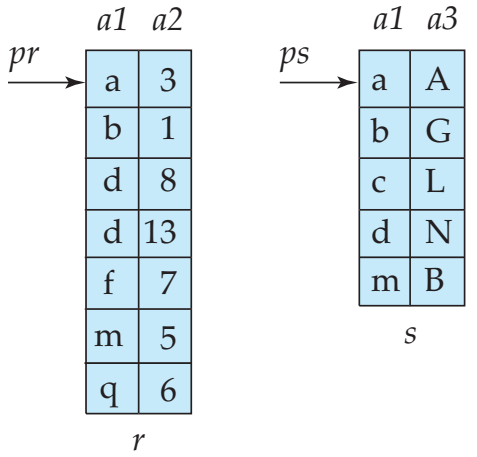
Algorithms for Join: 1-2 (Block) Nested-Loop

- Improvements:
  - If the *join attribute is a key* on inner relation, *stop* inner loop on first match of the join condition
  - If the *joint attribute* has *an index* on inner relation, *use efficient index lookups*
  - Scan inner loop forward and backward alternately, to make *efficient* use of the blocks remaining in buffer

Algorithms for Join: 3. Merge-Join

1. *Sort* both relations on their *join attribute* (if required)
2. *Join* them by merging the sorted relations
   Similar to merge stage of sort-merge algorithm
   1. Use a pointer on each relation's tuples, p_r and p_s
   2. For each value a_v of the joint attribute A (as ordered in s), repeat:
      1. Collect in S_s all tuples t_s in s with value $t_s[A]=a_v$
         This set must fit in memory!
      2. For any tuple t_r in r with $t_r[A]=a_v$
         For any tuple t_s in S_s
         result += $(t_r \bullet t_s)$

- Can be used for *equi-joins* and *natural joins*



Algorithms for Join: 3. Merge-Join

- Cost:
  - Each block needs to be read only once
    If S_s fits always in memory
    - No. block transfers: $b_r + b_s$
    - No. I/O accesses: $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$
      where b_b is the no. buffer blocks for each relation
  - Add cost of sorting (if required)
  - If S_s for any a_v don't fit in memory, slightly higher cost
    - *Block nested-loop join* can be performed for such sets

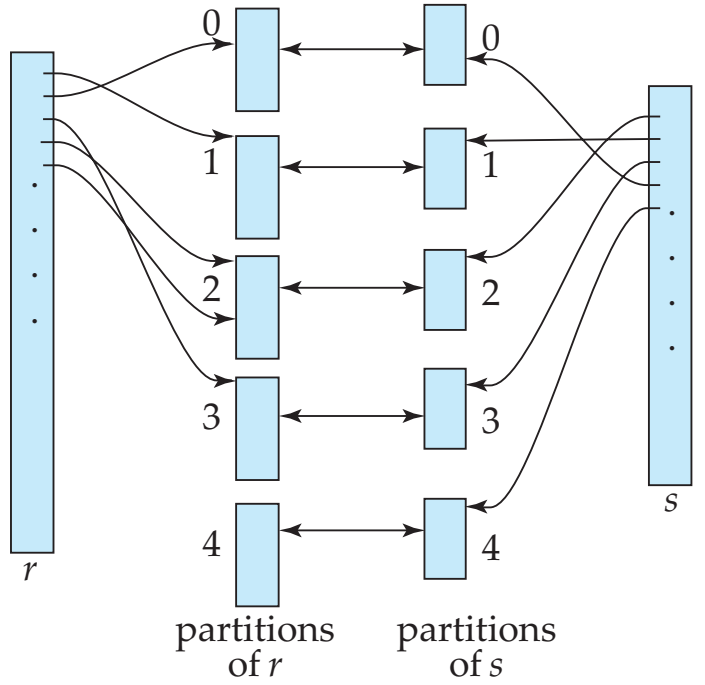
  - **Worst case** (b_b=*1*) of *student* ⋈ *takes,* with *ID* as *join attribute*
    - No. block transfers: *400 + 100 = 500*
    - No. I/O accesses: *400/1 + 100/1 = 500*

Algorithms for Join: 4. Hash-Join

- Define a **hash function** *h* on the *join attribute(s)*, *A*

  h: {values$_A$} ⟶ {0, 1, ..., *n*}

- Both *r* and *s* relations are partitioned into *n* subsets
- Tuples in r_i need to be compared only with tuples in s_i as:
  - A tuple pair $<t_r , t_s>$ that satisfies join condition has the same value for *A*: $t_r[A]=t_s[A]$
  - Thus, in both cases, h(A)=*i*
    - t_r will be sent to r_i , and t_s to s_i



- Needs *n* output buffers in memory
- Applicable for *equi-joins* and *natural joins*.

Algorithms for Join: 4. Hash-Join

- The **hash-join** of *r* and *s* is computed as follows:
  1. *Partition* relations *r* and *s* into *n* parts using hashing function *h*.
  2. For each *partition,* perform a *nested-loop join*.

- Select value *n* and hash function *h* such that resulting partitions s_i fit in memory:

  Partitions r_i don't need to fit in memory

$$n \geq \lceil b_s/M \rceil$$

Algorithms for Join: 4. Hash-Join

- **Cost**:
  - No. block transfers: $3(b_r + b_s) + 4n$
  - No. I/O accesses: $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil + n)$

  - Better to choose the smaller relation as *s*

- E.g., *instructor* ⋈ *teaches,* with *M=20* blocks, *n=5*
  - *instructor*: 5 partitions of 20 blocks. *teaches*: 5 partitions of 80 blocks
  - Cost (ignoring cost of writing partially filled blocks):
    - No. block transfers: 3(100 + 400) = 1500 blocks
    - No. I/O accesses: $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) = 336$ accesses

Algorithms for Join Operation

- *Nested-loop* and *block nested-loop joins* can test any join condition.
- Other join techniques only test simple conditions (natural or equi-joins)
- Join with a **conjunctive condition**:

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

  - Compute the result of a join $r \bowtie_{\theta_i} s$. Then, as tuples in this join are produced, check 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**:

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

  - Compute all individual joins $r \bowtie_{\theta_i} s$ and use their union:

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

Other operations

- **Duplicate elimination**:
  - Using *merge-sort*: Duplicates can be deleted as *runs* are built (before writing them to disk) *and* at merge.
  - Using *hashing*: after bucket partition, a new hashing is built in memory.
    - At that point, duplicates are easily identifiable.
  - SQL doesn't do it by default due to its high cost
    - Needs to be requested

- **Projection** (select subset of columns):
  - Projection on each tuple + *duplicate elimination* (if required)

- **Aggregation**: aggregate somehow the values of a column.
  - Similar implementation to duplicate elimination

CHOOSING AN EVALUATION PLAN

Choosing an Evaluation Plan

- A **query evaluation plan** defines, over an expression tree, the algorithms chosen for each operation.

- **Cost-based optimizers** explore the space of *all* equivalent query-evaluation plans and chooses that with the lowest cost.
  - Exhaustive search is expensive!
    E.g., finding the best **join order** for $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$.
    - No. orders = $(2n-2)!/(n-1)!$ ** Commutativity and associativity
      With $n = 7$, no. orders = 665.280 ; with $n = 10$, no. orders $\sim 2 \cdot 10^{10}$!
  - Use *dynamic programming*, to calculate the lowest-cost order for any subset of $\{r_1, r_2, \ldots, r_n\}$ only once.

Choosing an Evaluation Plan

- We want to generate all possible evaluation plans
  (as we did with expressions)
- **Physical equivalence rules** transform a logical query plan into a physical query plan by giving an algorithm for each operation.

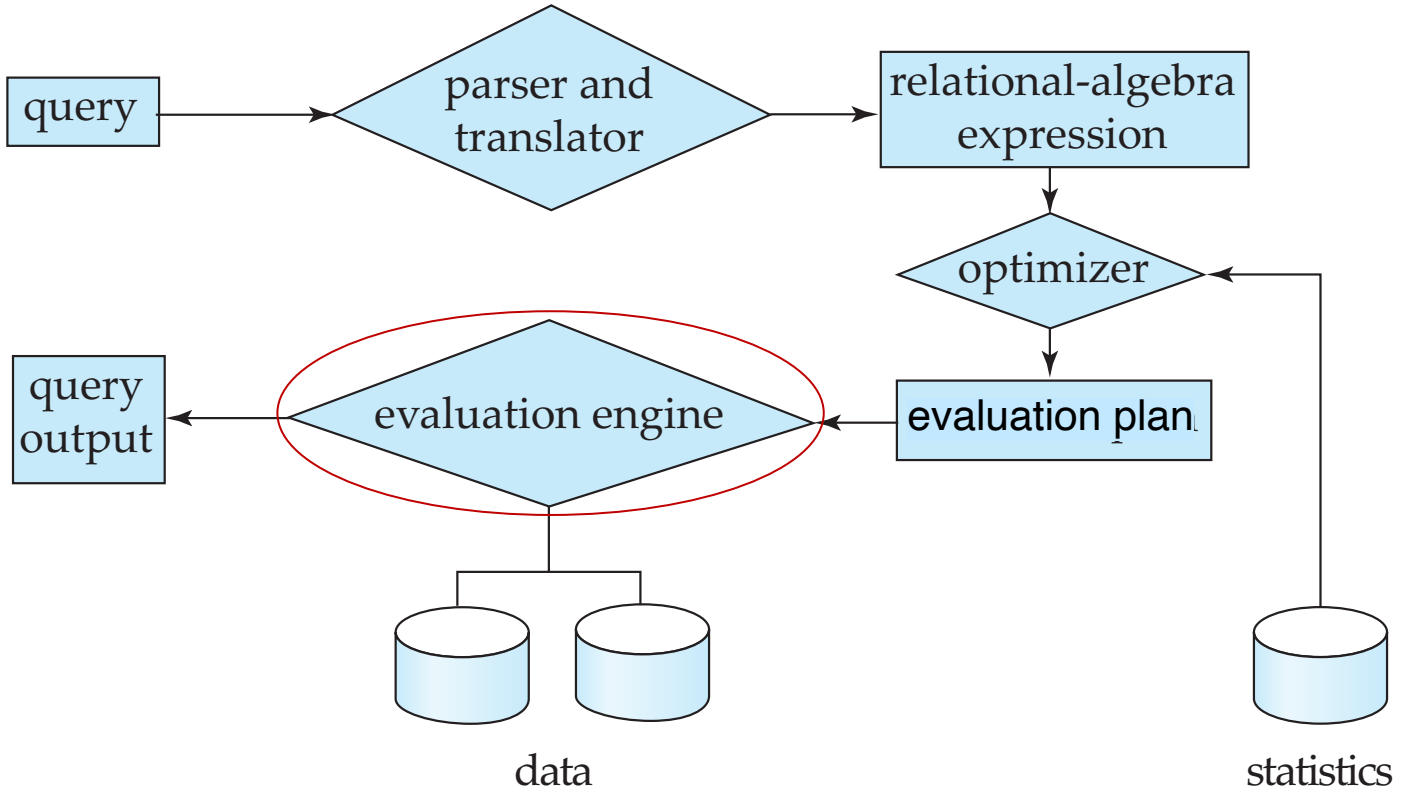
- An efficient cost-based optimizer needs:
  - To combine physical and standard equivalence rules
  - An **expression representation** that avoids multiple copies of subexpressions
  - **Memoization** (dynamic programming): for each optimization call on a subexpression, the best plan is stored
  - Cost-based **pruning** to avoid generating all plans
  - To use heuristic to increase performance

Choosing an Evaluation Plan

- To improve performance…
  - **Heuristics:** general rules which usually perform well:
    - Perform selection *asap*
      - Reduces no. tuples
    - Perform projection *asap*
      - Reduces no. attributes
    - Perform most restrictive selection and join operations (i.e., with smallest result size) before other similar operations
  - **Optimization cost budget** to stop optimization early:
    - Set a budget (dynamically) to, at most, spend within a plan
    - Do not explore unpromising plans
  - **Plan caching**: reuse a plan if query is resubmitted

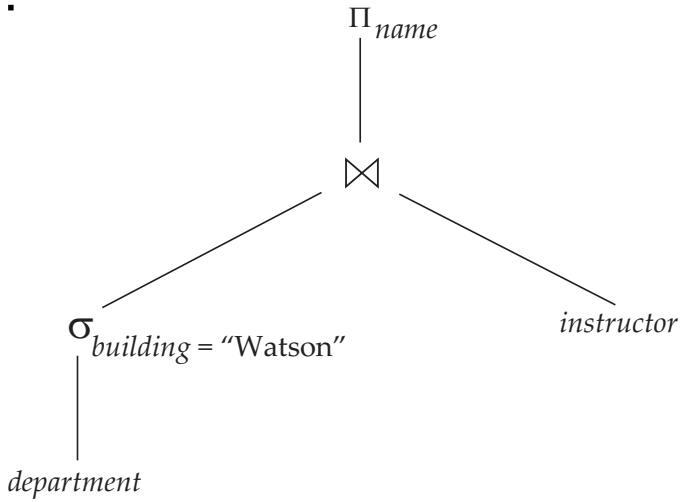
Basic Steps in Query Processing

1. Parsing and translation
   - Parser to check syntax, verify relation names, etc.
   - Translate query into internal form (~ relational algebra)
2. Optimization
3. Evaluation



Evaluation of Plans

- How to evaluate a **plan** containing *multiple operations*
  - **Operator trees**: used to visualize the expression. E.g.:
    1. Compute the *select*
       $\boldsymbol{\sigma}_{\text{building="Watson"}}(\text{department})$
    2. *join* the temporary relation with *instructor*
       temp_sel_d_w ⋈ *instructor*
    3. Project on *name*
       $\boldsymbol{\Pi}_{\text{name}}(\text{temp\_join\_dw\_i})$



- Two alternatives:
  - **Materialization**: evaluate one operation at a time
    - The result of each evaluation is temporarily stored on disk
  - **Pipelining**: evaluate several operations at the same time
    - Results from an evaluation are passed on to the next one without storing

Materialization

- Evaluate one operation at a time, starting at the lowest-level.
- Use intermediate results stored (*materialized*) as temporary relations to evaluate next operation.
- It is **always** applicable
- Cost
  - Cost of the individual operations, **plus**
  - Cost of writing results to disk (ignored so far)
    - no. block transfers: $b_r = n_r / f_r$
    - no. I/O accesses: b_r / b_b

      where n_r is no. records in intermediate relation r, f_r is no. records per block in relation r and b_b is no. buffer blocks

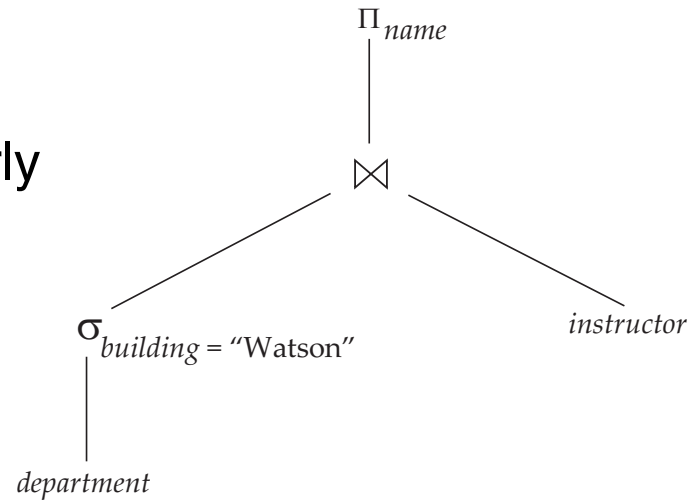
Pipelining

- **Pipelined evaluation**: evaluate several operations simultaneously, passing results on to the next op.
- Final tuples could be output on the fly
  if the root operation is pipelined
- Two strategies:
  - **Demand driven** (**lazy**): System requests next tuple *from top* level operation
    - Each operation modeled as an iterator [*open*(), *next*(), *close*()]
    - To produce next tuple, requests next tuple from its children ops.
    - Need to maintain a *current state* to know what to return next

Pulling tuples

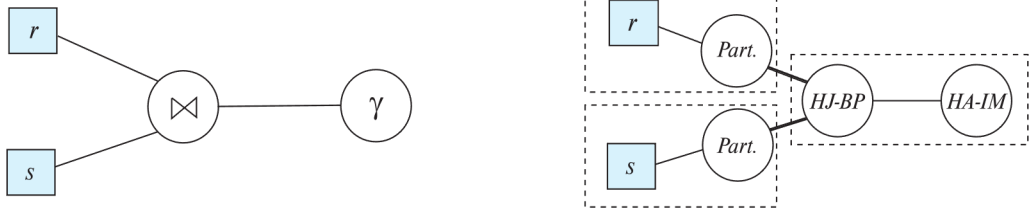
  - **Producer-driven** (**eager**): Operators produce tuples eagerly *from bottom* level op.
    - Each operation, a thread: It writes in an output buffer, and reads from children's output buffers
    - When there is space in its output buffer, it produces next tuple

Pushing tuples



Pipelining

- A *query plan* can be annotated:
  - Non-pipelined edges (aka, **blocking/materialized edges**)
  - **Pipelined edges**
    All ops. connected by pipelined edges can be executed concurrently
- A query plan can be divided into subtrees (**pipeline stage**) such that:
  - Each subtree has only pipelined edges
  - Edges *between* subtrees **are not** pipelined



- Query processor executes one *pipeline stage* at a time (all its operators at once)

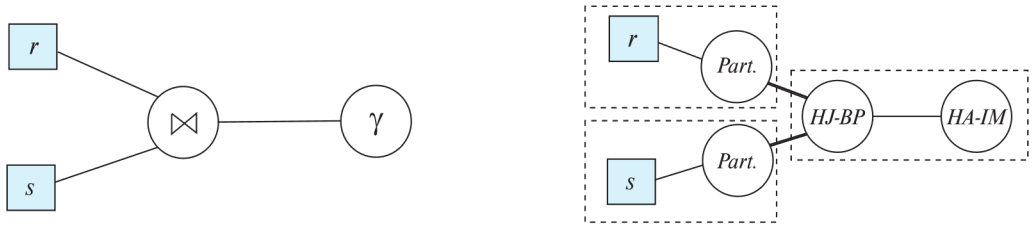
Pipelining

- **Blocking operations**: cannot output any result until all input tuples are visited

  E.g., sorting, aggregation, …

  - However, often, they can consume inputs or produce outputs as generated
  - That is, blocking operations often have *two suboperations* and the blocking actually happens between them

    E.g., *sort*: run creation & merge ; *hash join*: partitioning & compare
  - Treat them as separate operations linked by a non-pipeline edge



  - The external edges (to inputs/outputs) might be pipelined

Exercise: blocking operations

- Suppose you sort relations r and s using sort–merge and then apply merge-join the result with an already sorted relation t.
  - Which part of the expression is materialized and which part can be pipelined?

Query Processing