四川大學
SICHUAN UNIVERSITY

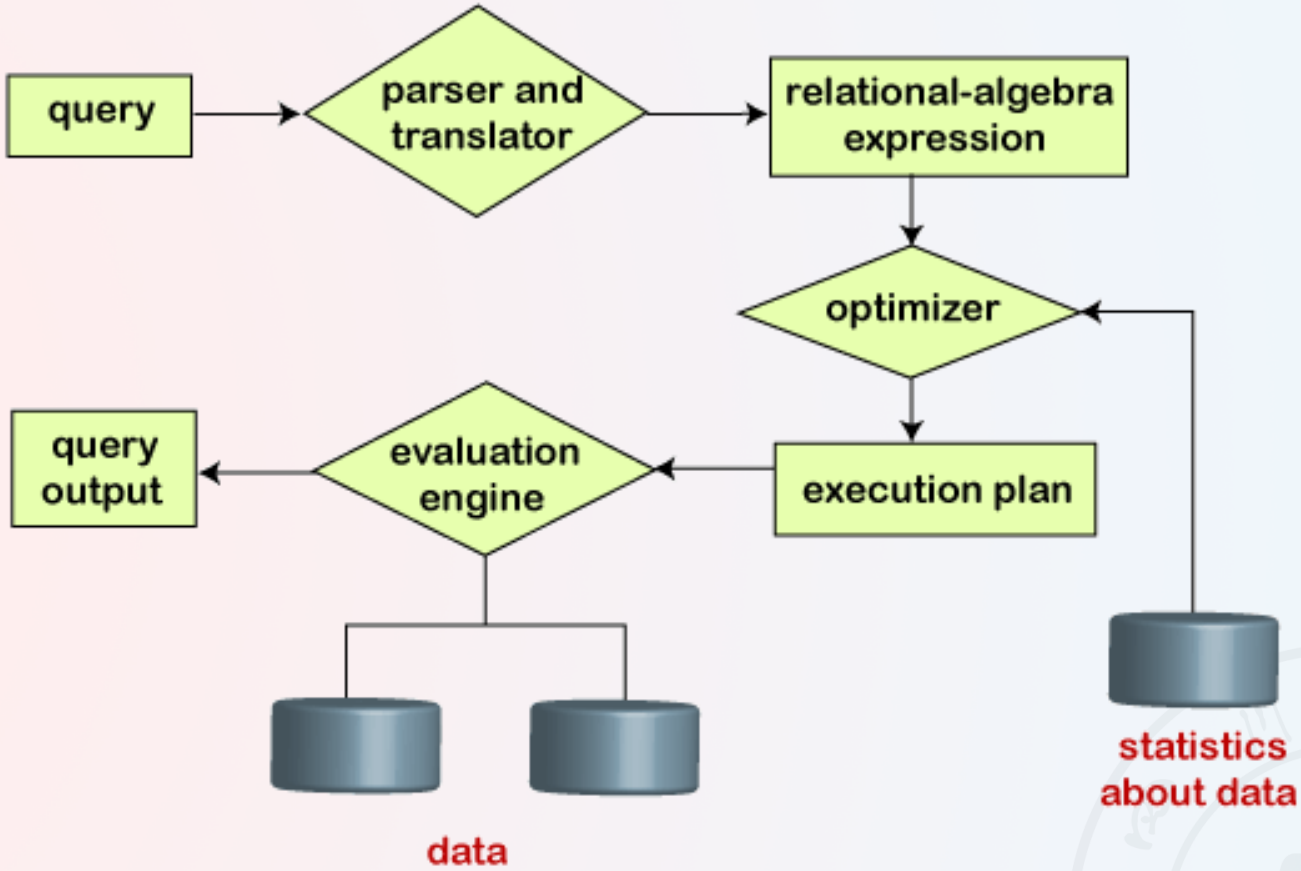
Database System Concepts

Query Processing and Optimization

伍元凯
College of Computer Science (Software), Sichuan University
wuyk0@scu.edu.cn

2023/03/29

1. Parsing and translation: translate the query into its internal form. This is then translated into relational algebra. Parser checks syntax, verifies relations
2. Optimization: Amongst all equivalent evaluation plans choose the one with lowest cost.
3. Evaluation: The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.



Steps in query processing

SQL is the best suitable choice for **humans**. **Relational algebra** is well suited for the internal representation of a **query**.

SQL query

```sql
SELECT name, age
FROM students
WHERE major = 'Computer Science' AND age > 20;
```

Relation algebra

$$\pi_{\text{name, age}}(\sigma_{\text{major='Computer Science'} \wedge \text{age} > 20}(\text{students}))$$

Evaluation plan

$\pi_{\text{name, age}}$

|

$\sigma_{\text{major='Computer Science'} \wedge \text{age} > 20}$; use Index 1

|

students

- A relational algebra expression may **have many equivalent expressions**:

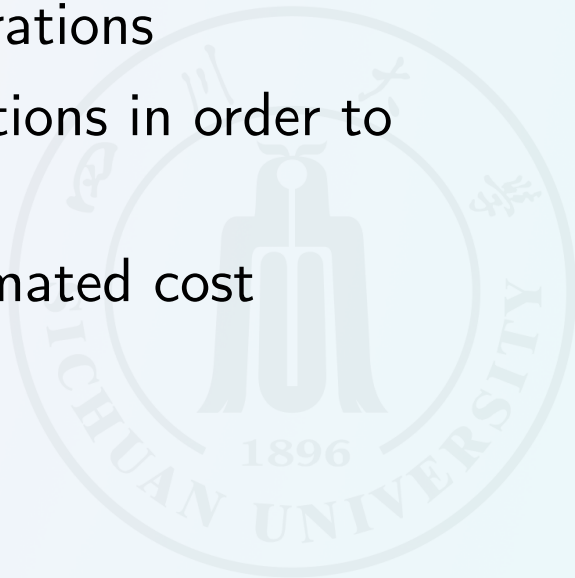
$$\begin{aligned}
&\pi_{\text{name, age}}(\sigma_{\text{major='Computer Science'} \wedge \text{age}>20}(\text{students})) \\
=&\sigma_{\text{major='Computer Science'}}(\sigma_{\text{age}>20}(\pi_{\text{name, age}}(\text{students}))) \\
=&\sigma_{\text{age}>20}(\sigma_{\text{major='Computer Science'}}(\pi_{\text{name, age}}(\text{students})))
\end{aligned}$$

- Each relational algebra operation can be evaluated using one of 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 (**Index or complete scan**).

Query Optimization:

Amongst all equivalent evaluation plans choose the one with lowest cost. Cost is estimated using statistical information from the database catalog. e.g. number of tuples in each relation, size of tuples, etc.

- How to measure query costs
- Algorithms for evaluating relational algebra operations
- How to combine algorithms for individual operations in order to evaluate a complete expression
- How to find an evaluation plan with lowest estimated cost



Cost is generally measured as total elapsed time for answering query:
Many factors contribute to time cost (disk accesses, CPU, or even network communication)
Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account

- Number of seeks *average-seek-cost
- Number of blocks read *average-block-read-cost
- Number of blocks written *average-block-write-cost

Cost to write a block is greater than cost to read a block (data is read back after being written to ensure that the write was successful)

postgresql.conf

```
# - Planner Cost Constants -
seq_page_cost = 1.0                     # Sequentially fetched disk
    page
random_page_cost = 4.0                  # the cost of reading a
    single disk page randomly
cpu_tuple_cost = 0.01                   # the cost of processing a
    single row of data on the CPU
cpu_index_tuple_cost = 0.005            #  the cost of processing a
    single index entry on the CPU
cpu_operator_cost = 0.0025              #  the cost of executing a
    single operator (such as addition or comparison) on the CPU
effective_cache_size = 4GB              # the amount of memory
    available for caching data
```

postgresql.conf

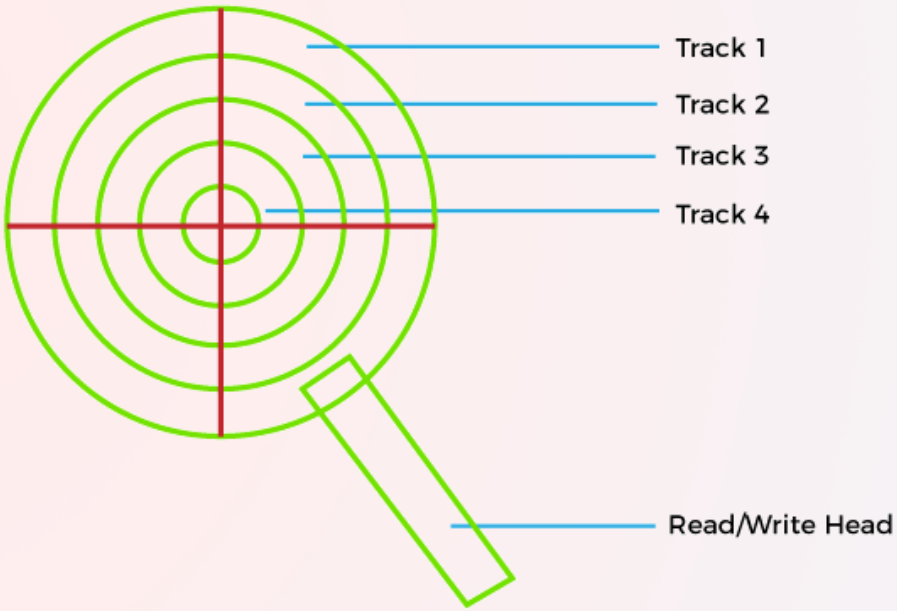
```
1 test=# explain(analyze, timing, verbose, buffers,costs) select max(
      sum) from ( select count(*) as sum from tgroup group by point)
      as t;
2 --
      ----------------------------------------------------------------


3  Aggregate  (cost=235363.15..235363.16 rows=1 width=8) (actual time
       =4898.900..4898.900 rows=1 loops=1)
4    Output: max((count(*)))
5    Buffers: shared hit=12770 read=49578
6    ->  HashAggregate  (cost=235363.04..235363.09 rows=5 width=4) (
         actual time=4898.890..4898.891 rows=5 loops=1)
7          Output: count(*), tgroup.point
8          Group Key: tgroup.point
9          Buffers: shared hit=12770 read=49578
10         ->  Seq Scan on public.tgroup  (cost=0.00..177691.36 rows
               =11534336 width=4) (actual time=0.045..1643.984 rows
```

```
                =11534336 loops=1)
11                Output: tgroup.id, tgroup.age, tgroup.point
12                Buffers: shared hit=12770 read=49578
13  Planning time: 0.170 ms
14  Execution time: 4898.982 ms
15 (12 rows)
16 Time: 4899.758 ms
17 test=#
```

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 (0.1 milliseconds, 2 microseconds fof SSD)
- t_S: time for one seek (4 milliseconds, 60 microseconds for SSD)
- Cost for b block transfers plus S seeks: $bt_T + St_S$

We ignore CPU costs for simplicity (**disk I/O costs are often the most significant factor**). We do not include cost to writing output to disk in our cost formulae



"Seek time" refers to the duration of waiting for your desired sushi to be prepared. "Transfer time" is the time it takes to pick up the sushi and bring it to your table for consumption. A "block" is a sequence of sushi that you desire.

Response time $\neq$ Query cost estimation

- **Measurement units**: Response time is measured in units of time, such as seconds or milliseconds, while query cost is typically measured in units of I/O operations or CPU cycles.
- **Granularity:** Response time measures the total time it takes to execute a query, while query cost estimation breaks down the cost of executing a query into individual operations or steps.
- **Use case:** Query cost estimation is typically used by query optimizers to compare different query plans and select the most efficient one. Response time, on the other hand, is more commonly used by end users to evaluate the performance of a query in real-world scenarios.

目录

- Algorithm A1 (linear search). Scan each file block and test all records to see whether they satisfy the selection condition.

$$C = b_r t_T + t_S$$

  b_r denotes number of blocks containing records from relation r.

- If selection is on a key attribute, can stop on finding record

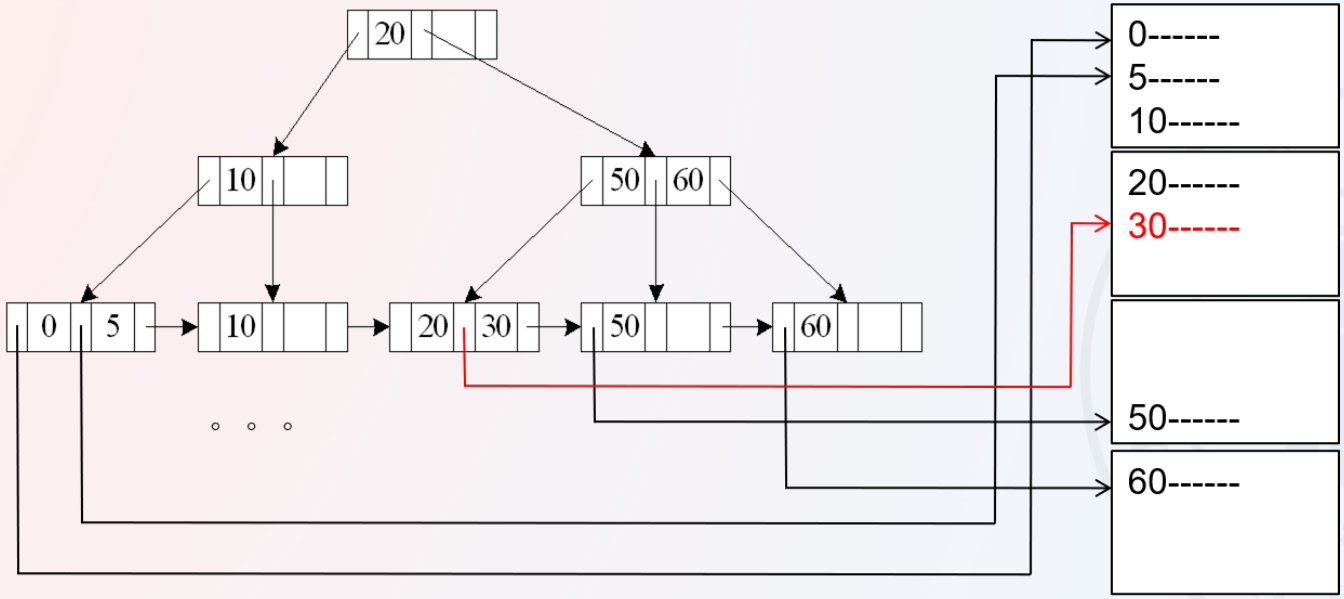
$$C = \frac{b_r}{2} t_T + t_S$$

- Linear search can be applied regardless of selection condition or ordering of records in the file, or availability of indices

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

$$C = (h_I + 1)(t_T + t_S)$$

h_I is the height of the index. Traverse the height of the tree plus one I/O to fetch the record, each of these I/O operation requires a seek and a block transfer.

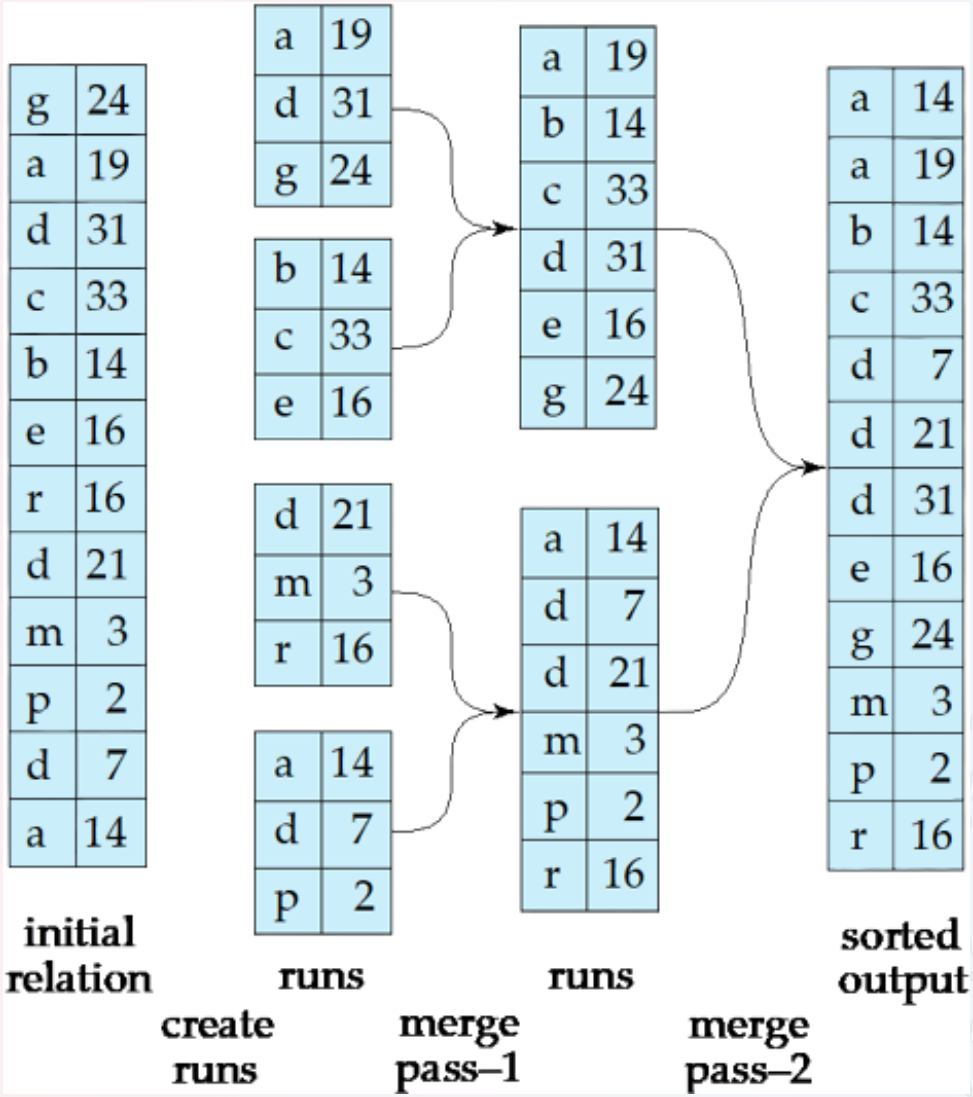


目录

Conjunction: $\sigma_{\theta 1} \wedge \theta_2 \cdots \theta_n(r)$

conjunctive selection using one index: Select a combination of θ_i that results in the least cost for $\sigma_{\theta i}(r)$.

- 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.


g 24
a 19
d 31
c 33
b 14
e 16
r 16
d 21
m 3
p 2
d 7
a 14
a 19
d 31
g 24
b 14
c 33
e 16
d 21
m 3
r 16
a 14
d 7
p 2
a 19
b 14
c 33
d 31
e 16
g 24
a 14
d 7
d 21
m 3
p 2
r 16
a 14
a 19
b 14
c 33
d 7
d 21
d 31
e 16
g 24
m 3
p 2
r 16
initial relation
create runs
runs
merge pass–1
runs
merge pass–2
sorted output

$N < M$, single merge pass is required(如果归并段少于可用内存页)

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**
3: Select the first record (in sort order) among all buffer pages
4: Write the record to the output buffer. If the output buffer is full write it to disk.
5: Delete the record from its input buffer page
6: **if** the buffer page becomes empty **then**
7: read the next block (if any) of the run into the buffer.
8: **end if**
9: **until** all input buffer pages are empty

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.

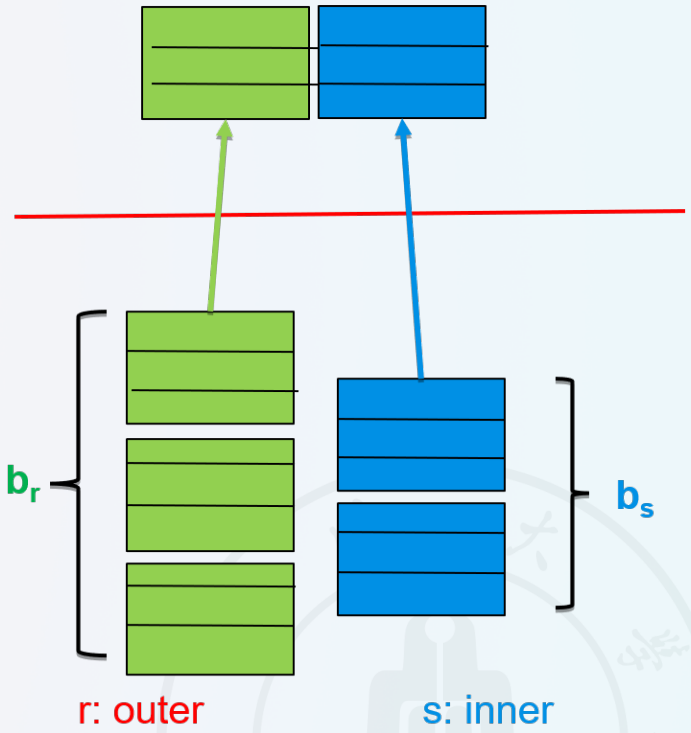
Cost analysis: (**simple version**)

- Total number of runs : $\{b_r/M\}$.
- Total number of merge passes required : $\{log_{M-1}(b_r/M)\}$
- Block transfers for initial run creation as well as in each pass is: $2b_r$
- Total: $b_r(2log_{M-1}(b_r/M)+1)$

1. Pass 0: $108/5 = 22$ sorted runs of 5 pages each
2. Pass 1: $22/4 = 6$ sorted runs of 20 pages each
3. Pass 2: $6/4 = 2$ sorted runs, 80 pages and 28 pages
4. Pass 3: Sorted file of 108 pages

To compute the theta join $r \bowtie_\theta s$

1: **for** each tuple t_r in r do begin **do**
2: **for** each tuple t_s in s do begin **do**
3: test pair (t_r, t_s) to see if they satisfy the join condition θ
4: **if** TRUE **then**
5: add t_r, t_s to the result
6: **end if**
7: **end for**
8: **end for**



In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

$$(n_r b_s + b_r)t_T + (n_r + b_r)t_S$$

Suppose we have 5000 students $n_r = 5000$ with 100 blocks $b_r = 100$, and we have 10000 takes $n_s = 10000$ with 400 blocks $b_s = 400$.
with student as outer relation: $5000 \times 400 + 100 = 2,000,100$ block transfers, 5100 seeks.
with takes as outer relation: $10000 \times 100 + 400 = 1,000,400$ block transfers, $10,400$ seeks.

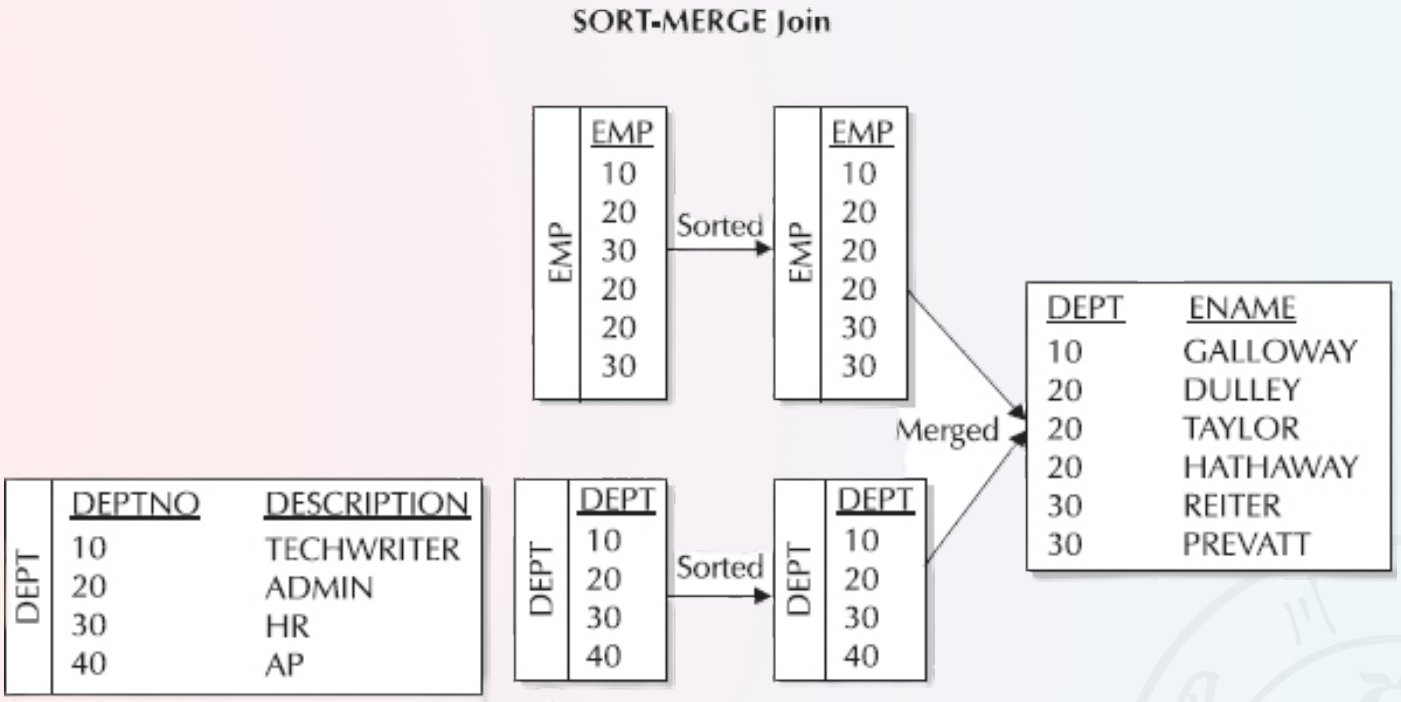
目录

```
1:  for each block B_r in r do begin do
2:      for each block B_s in s do begin do
3:          for each tuple t_r in B_r do begin do
4:              for each tuple t_s in B_s do begin do
5:                  test pair (t_r, t_s) to see if they satisfy the join condition θ
6:                  if TRUE then
7:                      add t_r, t_s to the result
8:                  end if
9:              end for
10:         end for
11:     end for
12: end for
```

Merge Join | 目录

The merge-join algorithm is a popular algorithm for combining two **sorted** datasets.

1. The two datasets to be merged are sorted in ascending order based on a common key.
2. Two pointers are initialized to the first record in each dataset.
3. If the key value in the first table is less than the key value in the second table, the pointer for the first dataset is moved forward by one record. Same for the second table.
4. If the key values in both tables are equal, the records are combined into a single record and output.
5. This process is repeated until one or both of the datasets have been completely processed.



Assuming b_b buffer blocks are allocated to each relation.

$Cost = (b_r/b_b + b_s/b_b)t_S + (b_r + b_s)t_T + sortTime$

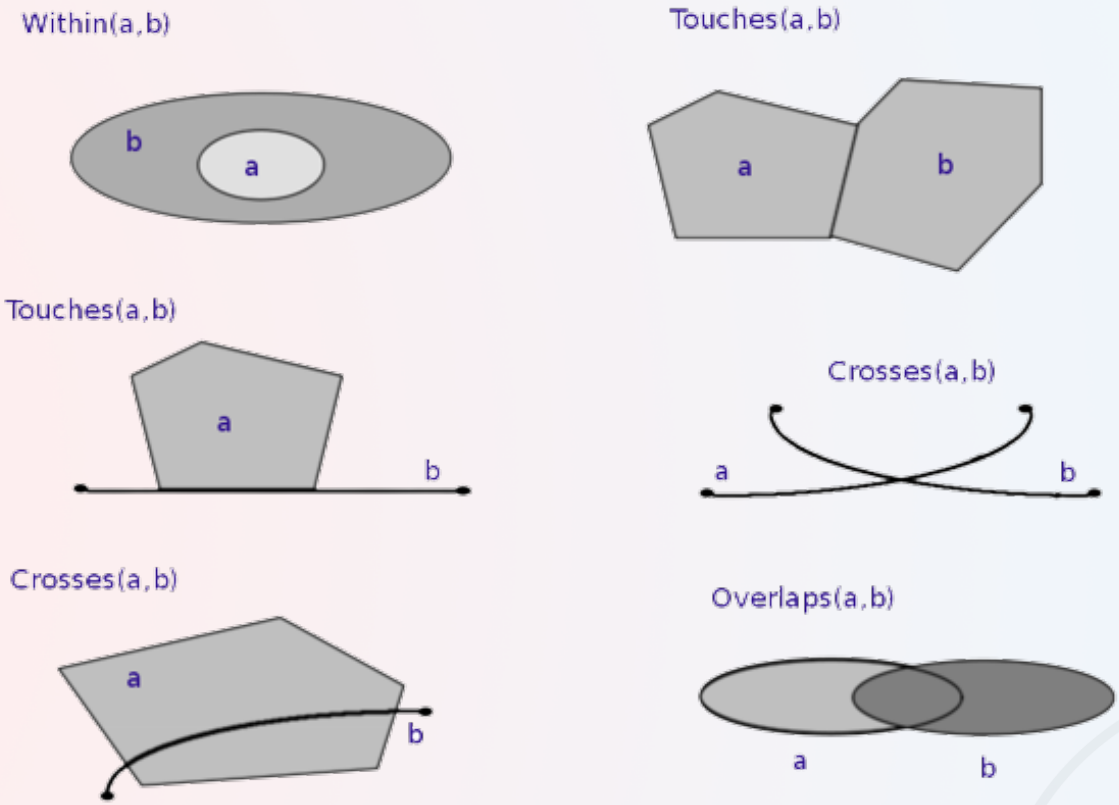
Join with conjunctive conditions $r \bowtie_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n} s$:

- 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

Join with conjunctive conditions $r \bowtie_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n} s$:

- Either use nested loops/block nested loops, or
- Compute as the union of the records in individual joins $r \bowtie_{\theta_i} s$: $r \bowtie_{\theta_1} s \cup r \bowtie_{\theta_2} s \cdots \cup r \bowtie_{\theta_n} s$

目录



spatial join

```
1 SELECT name, city
2 FROM citizens
3 JOIN city
4 on ST_CONTAINS(city.geom, citizens.geom);
```

ID	Name	Geom
1	John Doe	POINT(-74.0059 40.7128)
2	Jane Smith	POINT(-87.6298 41.8781)
3	Bob Johnson	POINT(-118.2437 34.0522)

Citizens Table

ID	Name	
1	New York	POLYGON((-74.047285 40.6796
2	Chicago	POLYGON((-87.940101 41.6442
3	Los Angeles	POLYGON((-118.668404 33.70365

City

Name	City
John Doe	New York
Jane Smith	Chicago
Bob Johnson	Los Angeles

Result of Spatial Join Query

Nested loops join can always be used.

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.

Aggregation: combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values

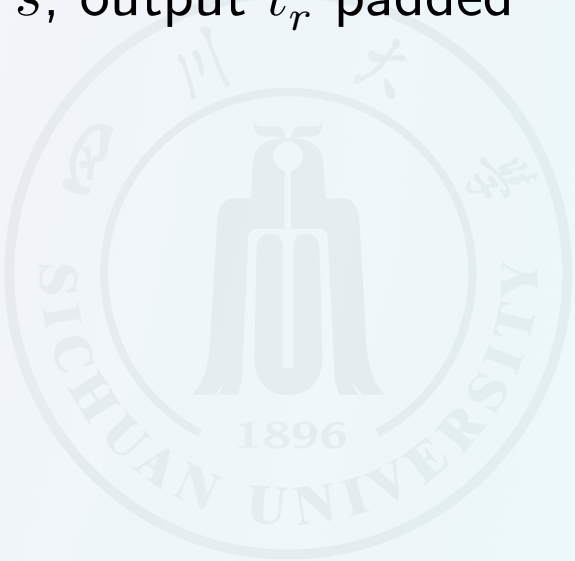
- For count, min, max, sum: keep aggregate values on tuples found so far in the group. When combining partial aggregate for count, add up the aggregates
- For avg, keep sum and count, and divide sum by count at the end

Set operations ($\cup$, $\cap$ and $-$): can either use variant of merge-join after sorting, or variant of hash-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 left outer join: During merging, for every tuple t_r from r that do not match any tuple in s, output t_r padded with nulls.



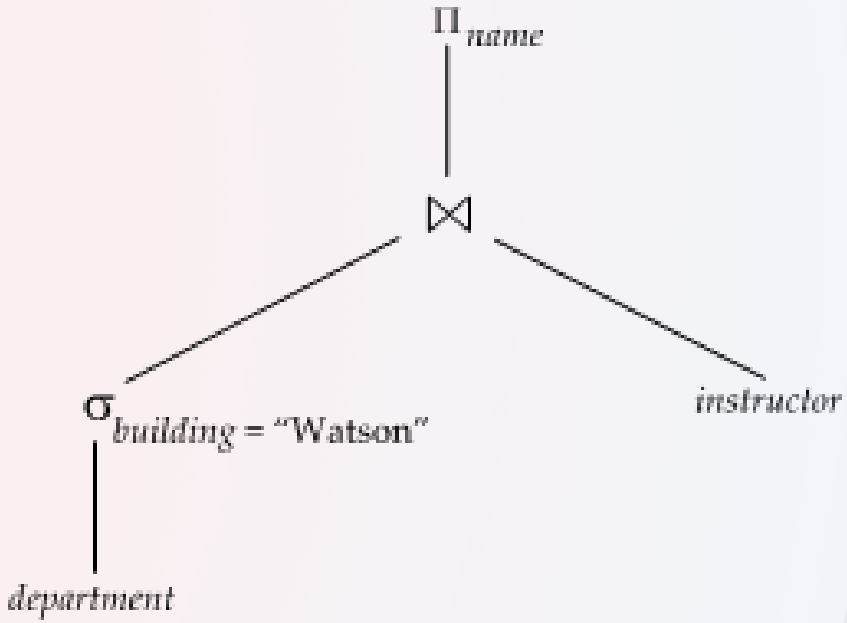
So far: we have seen algorithms for individual operations
Alternatives for evaluating an entire expression tree

- Materialization: generate results of an expression whose inputs are relations or are already computed, materialize (store) it on disk. Repeat.
- Pipelining: pass on tuples to parent operations even as an operation is being executed.

Materialization 目录

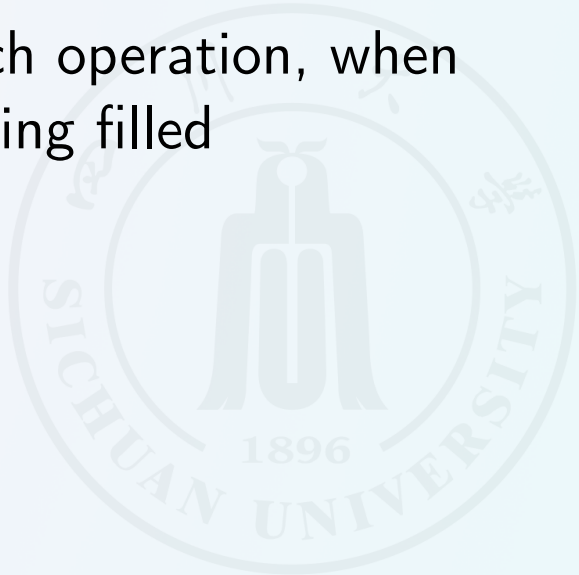
Materialized evaluation: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.

$$\Pi_{name}\left(\sigma_{building='watson'}(department) \bowtie instructor\right)$$



compute and store $\sigma_{building='watson'}(department)$ (in a buffer, they are written in disk when the buffer is full). then compute the store its join with instructor, and finally compute the projection on name.

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
- Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- Double buffering: use two output buffers for each operation, when one is full write it to disk while the other is getting filled



- Pipelined evaluation : evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of $\sigma_{building='watson'}(department)$, instead, pass tuples directly to the join. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: demand driven and producer driven

In demand driven or lazy evaluation

- system repeatedly requests next tuple from top level operation
- Each operation requests next tuple from children operations as required, in order to output its next tuple
- In between calls, operation has to maintain "state" so it knows what to return next

In producer-driven or eager pipelining

- Operators produce tuples eagerly and pass them up to their parents
- Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
- if buffer is full, child waits till there is space in the buffer, and then generates more tuples
- System schedules operations that have space in output buffer and can process more input tuples

Pipeline	Materialization
It does not use any temporary relations for storing the results of the evaluated operations.	It uses temporary relations for storing the results of the evaluated operations. So, it needs more temporary files and I/O.
It is a more efficient way of query evaluation as it quickly generates the results.	It is less efficient as it takes time to generate the query results.
It requires memory buffers at a high rate for generating outputs. Insufficient memory buffers will cause thrashing.	It does not have any higher requirements for memory buffers for query evaluation.
Poor performance if trashing occurs.	No trashing occurs in materialization. Thus, in such cases, materialization is having better performance.
It optimizes the cost of query evaluation. As it does not include the cost of reading and writing the temporary storages.	The overall cost includes the cost of operations plus the cost of reading and writing results on the temporary storage.

Comparison of Pipeline and Materialization

- Two relational algebra expressions are said to be equivalent if the two expressions generate the same set of tuples on every legal database instance
- In SQL, inputs and outputs are multisets of tuples Two expressions in the multiset version of the relational algebra are said to be equivalent if the two expressions generate the same multiset of tuples on every legal database instance.
- An equivalence rule says that expressions of two forms are equivalent Can replace expression of first form by second, or vice versa

- Conjunctive selection operations can be **deconstructed（分解）** into a sequence of individual selections. $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
- Selection operations are **commutative（可交换的）**. $\sigma_{\theta_2}(\sigma_{\theta_1}(E)) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
- Only the last in a sequence of projection operations is needed, the others can be **omitted（可省略的）**. $\Pi_{L_1}(\Pi_{L_2}(\cdots \Pi_{L_n}(E)\cdots)) = \Pi_{L_1}(E)$
- Selections can be **combined** with Cartesian products and theta joins. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$, $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
- Theta-join operations (and natural joins) are **commutative（可交换的）**. $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$

- Natural join operations are **associative（可结合的）**:
  $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
- Theta joins are **associative** in the following manner:
  $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$, θ_2 involves attributes from only E_2 and E_3
- The selection operation **distributes（分配）** over the theta join operation under the following two conditions: When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined. $\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_0}(E_1) \bowtie_\theta E_2$
- When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E2. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_1}(E_1) \bowtie_\theta \sigma_{\theta_2}(E_2)$

- The projection operation **distributes（分配）** over the theta join operation as follows: if θ involves only attributes from $L_1 \cup L_2$:
  $\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1}(E_1) \bowtie_\theta \Pi_{L_2}(E_2)$
- Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively. Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.
  $\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}\left(\Pi_{L_1 \cup L_3}(E_1) \bowtie_\theta \Pi_{L_2 \cup L_4}(E_2)\right)$

- The set operations union and intersection are **commutative**.
  $E_1 \cup E_2 = E_2 \cup E_1$, $E_1 \cap E_2 = E_2 \cap E_1$.
- Set union and intersection are **associative**
  $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$, $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$
- The selection operation **distributes** over $\cap$, $\cup$ and $-$:
  $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$, and similarly for $\cap$, $\cup$ and $-$.
  $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$, and similarly for $\cap$, but not for $\cup$.
- The projection operation **distributes over union**
  $\Pi_L(E_1 \cup E_2) = \Pi_L(E_1) \cup \Pi_L(E_2)$
- Selection **distributes over aggregation** as below
  $\sigma_\theta({}_G\gamma_A(E)) =_G \gamma_A(\sigma_\theta(E))$

- Full outerjoin is **commutative**. Left and right outerjoin are **not commutative**. E_1 leftjoin E_2 equals to E_2 rightjoin E_1.
- Selection **distributes over left and right outer joins** as below, provided
- Outerjoins can be replaced by inner joins under some conditions. Outerjoins are not **associative**.

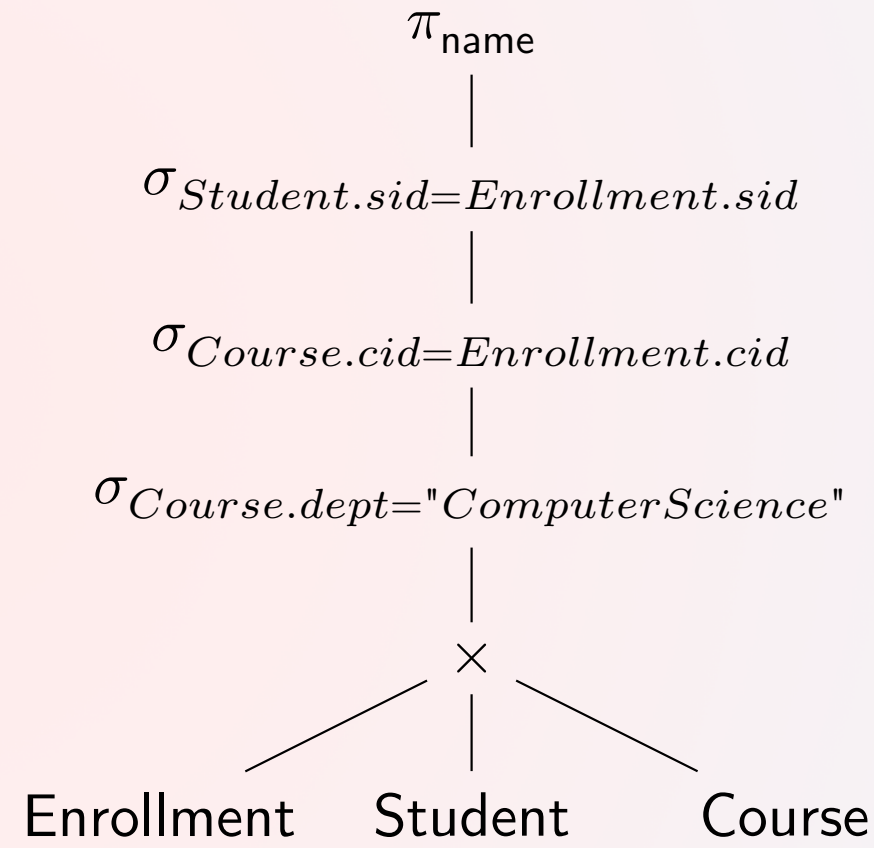
```
1 SELECT DISTINCT Student.name
2 FROM Student, Course, Enrollment
3 WHERE Student.sid = Enrollment.sid
4 AND Course.cid = Enrollment.cid
5 AND Course.dept = 'Computer Science'
```

The original one:

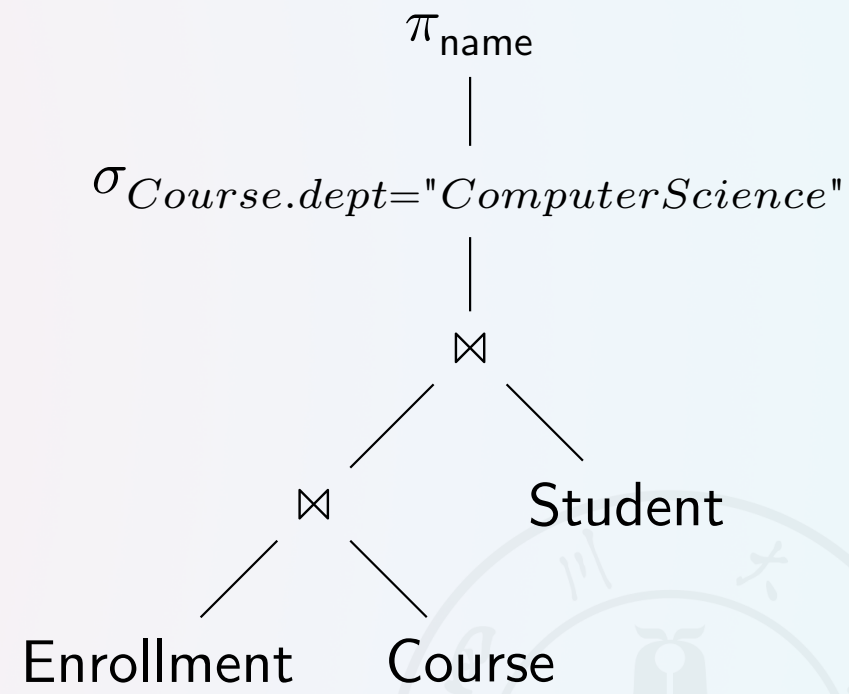
$$\pi_{name}(\sigma_{dept='ComputerScience'}(Student \bowtie Enrollment \bowtie Course))$$

The transformed one:

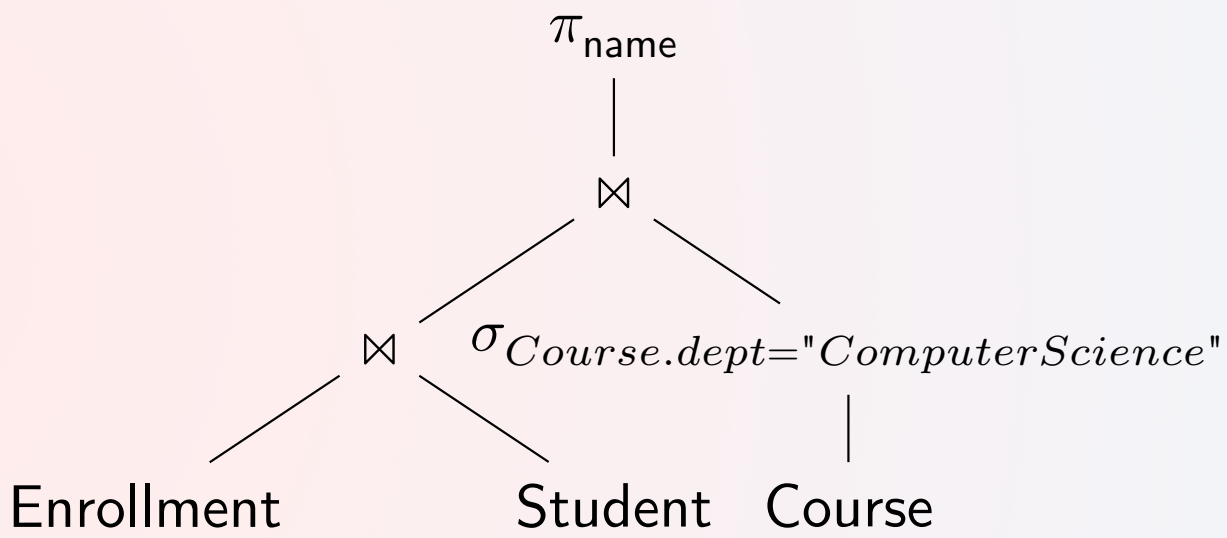
$$\pi_{name}((Student \bowtie Enrollment) \bowtie \sigma_{dept='ComputerScience'}(Course))$$



SQL expression



Join expression



Performing the selection as early as possible

The **commutativity** of the natural join operation allows us to rearrange the order of the tables being joined without changing the result of the query.

$\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$

Consider:

$$\Pi_{\text{name, title}}(\sigma_{\text{dept\_name} = \text{"Music"}}(\text{instructor} \bowtie \text{teaches}) \bowtie \Pi_{\text{course\_id, title}}(\text{course}))$$

When we compute

$$\sigma_{\text{dept\_name} = \text{"Music"}}(\text{instructor} \bowtie \text{teaches})$$

Push projections using equivalence rules; eliminate unneeded attributes from intermediate results to get:

$$\Pi_{\text{name, title}}(\Pi_{\text{name, course\_id}}(\sigma_{\text{dept\_name} = \text{“Music”}}(\text{instructor} \bowtie \text{teaches})) \bowtie \Pi_{\text{course\_id, title}}(\text{course}))$$

Performing the projection as early as possible reduces the size of the relation to be joined.

Join Ordering 目录

Natural join operations are **associative（可结合的）**:
$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$.
if $(E_2 \bowtie E_3)$ is quite large and $(E_1 \bowtie E_2)$ is small, we choose $(E_1 \bowtie E_2) \bowtie E_3$.
so that we compute and store a smaller temporary relation.

```sql
SELECT employees.name
FROM employees
INNER JOIN departments ON employees.department_id = departments.
    department_id
INNER JOIN projects ON employees.employee_id = projects.employee_id
WHERE projects.project_id = 12345 AND departments.location = 'New
    York'
```

The original relational algebra expression for this query would be:

$$\Pi_{\text{name}}\left(\sigma_{\text{location = 'New York'}}\left(\text{departments} \bowtie_{\text{department\_id}} \left(\text{employees} \bowtie_{\text{employee\_id}} \sigma_{\text{project\_id = 12345}}(\text{projects})\right)\right)\right.$$

Transformation using join **associatively**

$$\Pi_{\text{name}}(\sigma_{\text{location='New York' AND project\_id=12345}}(\text{departments} \bowtie_{\text{department\_id}} (\text{employees} \bowtie_{\text{employee\_id}} \text{projects})))$$

1: **repeat**
2: apply all applicable equivalence rules on every subexpression of every equivalent expression found so far
3: add newly generated expressions to the set of equivalent expressions
4: **until** no new equivalent expressions are generated above

The above approach is very expensive in space and time.
Optimized with Heuristics and transformation rules.

n_r : number of tuples in a relation r

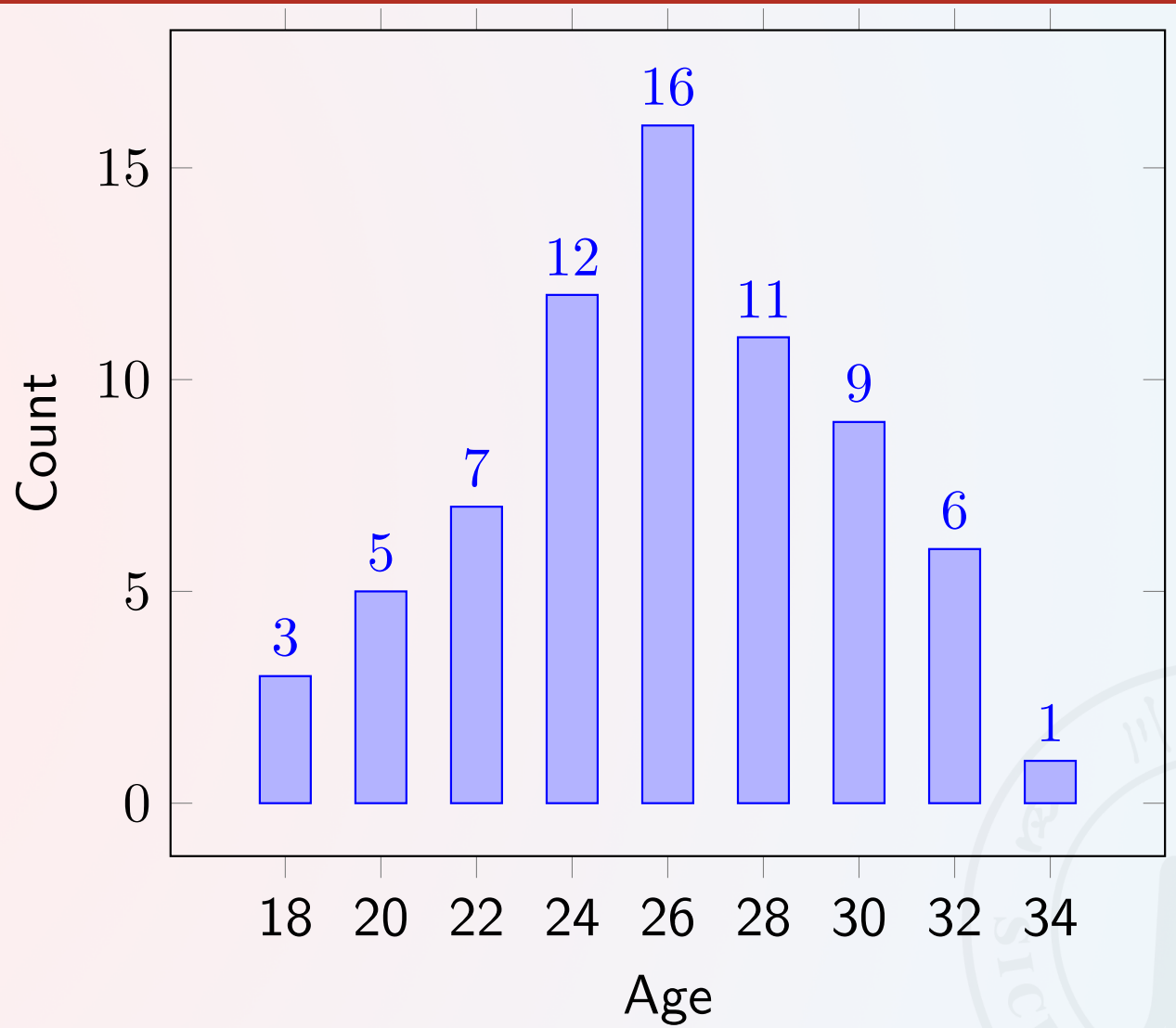
b_r : number of blocks containing tuples of r

l_r : size of a tuple of r

f_r : blocking factor of r —i.e., the number of tuples of r that fit into one block.

:$V(A, r)$ number of distinct values that appear in r for attribute A; same as the size of $\Pi(A(r))$.

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$



Equi-width histograms: equal-sized ranges

Equi-depth histograms: equal-width

目录

$\sigma_{A=v}(r)$

- $n_r/V(A,r)$: number of records that will satisfy the selection
- Equality condition on a key attribute: 1

$\sigma_{A\geq v}(r)$

Let c denote the estimated number of tuples satisfying the condition. If $min(A,r)$ and $max(A,r)$ are available in catalog

- $c = 0$: if $v < min(A,r)$
- $c = n_r \frac{v-min(A,r)}{min(A,r)-min(A,r)}$

If histograms available, can refine above estimate

In absence of statistical information c is assumed to be $n_r/2$.

The selectivity（中选率）of a condition θ_i is the probability that a tuple in the relation r satisfies θ_i . If s_i is the number of satisfying tuples in r, the selectivity of ☒i is given by s_i/n_r.

Conjunction: $\sigma_{\theta_1 \wedge \theta_2 \cdots \theta_n}(r)$. Assuming independence, estimate of tuples in the result is:

$$n_r \frac{s_1 s_2 \cdots s_n}{n_r^n}$$

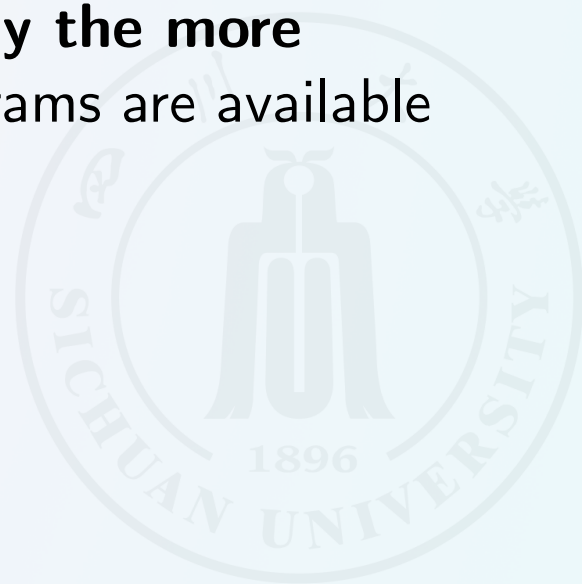
Disjunction: $\sigma_{\theta_1 \wedge \theta_2 \cdots \theta_n}(r)$. Assuming independence, estimate of tuples in the result is:

$$n_r \left(1 - \left(1 - \frac{s_1}{n_r}\right)\left(1 - \frac{s_2}{n_r}\right) \cdots \left(1 - \frac{s_n}{n_r}\right)\right)$$

Negation: $\sigma_{\_\theta}(r)$. Estimated number of tuples:

$$n_r - size(\sigma_\theta(r))$$

- The case for $R \cup S$ being a foreign key referencing S is symmetric.
- If we assume that every tuple t in R produces tuples in $R \bowtie S$, the number of tuples in $R \bowtie S$ is estimated to be: $\frac{n_r n_s}{V(A,s)}$
- If the reverse is true, the estimate obtained will be: $\frac{n_r n_s}{V(A,r)}$
- **The lower of these two estimates is probably the more accurate one.** Can improve on above if histograms are available



Running example: $student \bowtie takes$
Catalog information for join examples:
$n_{student} = 5,000$.
$f_{student} = 50$, which implies that $b_{student} = 5000/50 = 100$.
$n_{takes} = 10000$.
$f_{takes} = 25$, which implies that $b_{takes} = 10000/25 = 400$.
$V(ID, takes) = 2500$, which implies that on average, each student who has taken a course has taken 4 courses.
Attribute ID in takes is a foreign key referencing student.
$V(ID, student) = 5000$ (primary key!)
5000 * 10000/2500 = 20,000, 5000 * 10000/5000 = 10,000. We choose the lower estimate.

Other Operations

目录

- Projection: estimated size of $\Pi_A(r) = V(A, r)$
- Aggregation : estimated size of ${}_A G_f(r) = V(A, r)$
- Set operation: For unions/intersections of selections on the same relation: rewrite and use size estimate for selections (e.g. $\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r) = \sigma_{\theta_1 \wedge \theta_2}(r)$)
- **estimated** size of $r \cup s$ = size of r + size of s
- **estimated** size of $r \cap s$ = minimum size of r and size of s
- **estimated** size of $r - s$ = size of r
- outer join: size of $r \bowtie s$ + size of r or +size of s.

目录

Selections: $\sigma_\theta(r)$, estimate $V(A, \sigma_\theta(r))$

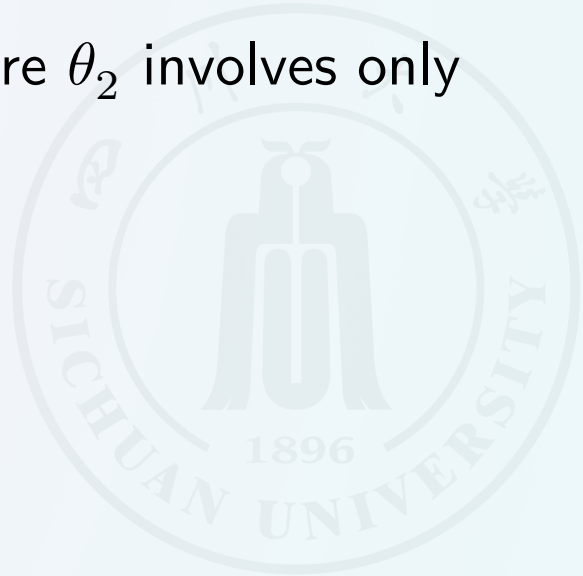
- If θ forces A to take a specified value: $V(A, \sigma_\theta(r)) = 1$
- If θ forces A to take a set of values:
  $V(A, \sigma_\theta(r)) =$ number of specified values
- If the selection condition θ is of the form A comparing v estimated
  $V(A, \sigma_\theta(r)) = sV(A, r)$
- In all the other cases: use approximate estimate of
  $min(V(A, \sigma_\theta(r)), n_{\sigma_\theta(r)})$

15.8 Design sort-based algorithm for computing the relational division operation 设计基于排序的算法来计算关系除法运算。

16.17 Show how to derive the following equivalence by a sequence of transformation using the equivalence rule 展示如何使用等价规则通过一系列变换推导出以下等价式:

a. $\sigma_{\theta_1 \wedge \theta_2 \wedge \theta_3}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(E)))$

b. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2) = \sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$, where θ_2 involves only attributes from E_2



Thanks