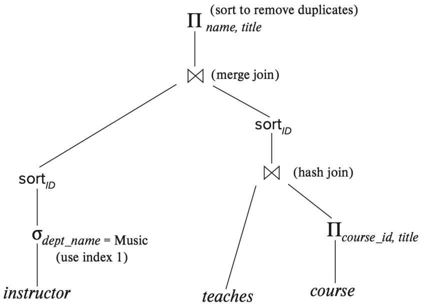
Query Processing (I)

Spring, 2024

DBMS: Operator execution

Execute a dataflow by operation on tuples and files.



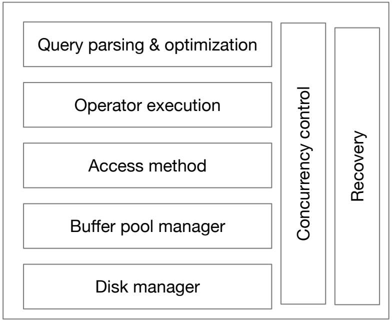
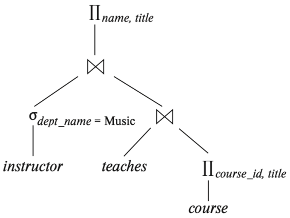
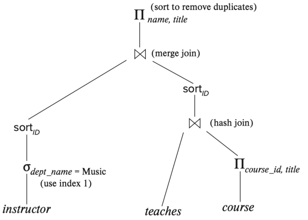


Figure: DBMS architecture

Query processing overview

```sql
SELECT name, title
FROM instructor natural join teaches
     natural join course
WHERE dept_name ='Music';
```





- Each node of a logical plan is a relational operator.
- Each node of a physical plan represents an operator algorithm.
- Data flows from the leaves of the physical plan tree up towards the root.

Notations

- Tables: R, S
- Tuples: t_r, t_s
- Number of tuples: $|R|$, $|S|$
- Number of pages: $P(R)$, $P(S)$
- Number of available buffer pool pages: B
- Cost metric: number of I/O's

Sequential scan

- Scan table R sequentially and process the query
  - Selection over R
  - Projection of R without duplicate elimination
- I/O cost: $P(R)$
- Not counting the cost of writing the result out
  - Maybe not needed – results may be pipelined into another operator
  - Same for the algorithms discussed later

Sorting

Why sorting

- Tuples in a table have no specific order.
- Query may require output be sorted.
  - E.g., `SELECT * from student ORDER BY credit DESC;`
- Several relational operators can be implemented efficiently with sorting.
  - E.g., duplication elimination, aggregation, merge join, set operations.
- External sorting is required when data cannot fit in memory.

External merge sorting

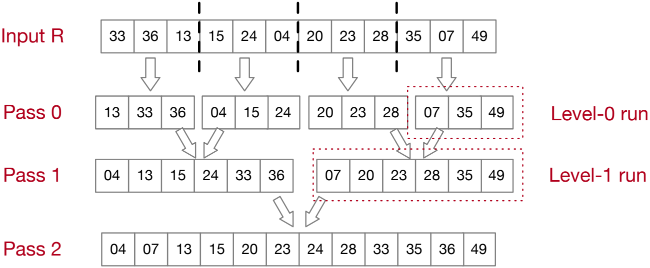
A divide-and-conquer approach to sort a large relation R that cannot fit in memory.

Recall that we have B pages available in the buffer pool.

- Pass 0: read B pages of R each time, sort them, and write out a level-0 run.
- Pass 1: merge $B - 1$ level-0 runs each time, and write out a level-1 run.
- Pass 2: merge $B - 1$ level-1 runs each time, and write out a level-2 run.
- $\cdots$
- Final pass produces one sorted run.

External merge example

- $B = 3$, i.e., 3 pages available in buffer pool.
- Each page hols only one tuple.
- In pass 0, all 3 pages are used for sorting.
- In pass i, where $i \geqslant 1$, $B-1=2$ pages are used for input, and 1 page for output.



Input R: 33 36 13 | 15 24 04 | 20 23 28 | 35 07 49

Pass 0: 13 33 36 | 04 15 24 | 20 23 28 | 07 35 49 — Level-0 run

Pass 1: 04 13 15 24 33 36 | 07 20 23 28 35 49 — Level-1 run

Pass 2: 04 07 13 15 20 23 24 28 33 35 36 49

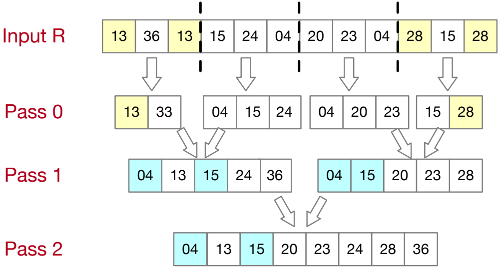
Cost analysis

#(Passes)	$\log_{B-1}\lceil P(R)/B \rceil + 1$
#(Read Pages)	$P(R) * (\log_{B-1}\lceil P(R)/B \rceil + 1)$
#(Write Pages)	$P(R) * \log_{B-1}\lceil P(R)/B \rceil$
Total cost	$2P(R) * \log_{B-1}\lceil P(R)/B \rceil + P(R)$

- Pass 0: read B pages of R each time, sort them, and write out a level-0 run.
- Pass i: merge $(B-1)$ level-$(i-1)$ runs each time, and write out a level-i run.
- Each pass read the entire relation and write it once.
- We do not include the output cost of the final pass as we have discussed.

Sort-based duplication elimination

1. Perform external merge sort.
2. Eliminate duplicates during sort and merge.
3. Cost: same cost as sorting.



Input R: 13 36 13 | 15 24 04 | 20 23 04 | 28 15 28

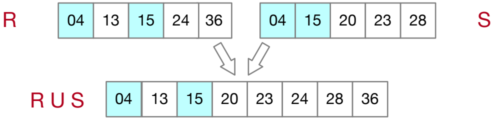
Pass 0: 13 33 · 04 15 24 · 04 20 23 · 15 28

Pass 1: 04 13 15 24 36 · 04 15 20 23 28

Pass 2: 04 13 15 20 23 24 28 36

Sort-based set operations

- $R \cup S$, $R \cap S$, $R - S$ requires duplication elimination by default.
- Sort R and S in the same order.
- Scan the sorted R and S to produce the desired results and eliminate duplicates.
- Both R and S require only one pass of scan.
- Cost: sorting + $P(R) + P(S)$



Sort-based aggregation

- Sort the tuples on the GROUP BY attributes
- Perform a sequential scan over the sorted data to compute the aggregation.
  - This can be fused into the final pass of sorting.
- Apply partial aggregation on the fly.
- The output will be sorted on the attributes.
- Cost: same cost as sorting.

```
SELECT dept_name, AVG(salary)
FROM instructor
GROUP BY dept_name
```

Agg	Running value
MIN	min
MAX	max
COUNT	count
SUM	sum
AVG	(count, sum)

Join

Naive nested loop join

```
for each tuple $t_r$ in $R$ do
    for each tuple $t_s$ in $S$ do
        if $\theta(t_r, t_s)$ then
            add $t_r \bowtie t_s$ to the result
```

Figure: Algorithm for $R \bowtie_\theta S$

- The most basic join algorithm to compute join $R \bowtie_\theta S$.
- R: the outer table, S: the inner table.
- Require no indices and can be used with any kind of join conditions.

Cost analysis

- Cost: $P(R) + |R| * P(S)$
- Buffer pool requirement: $B = 3$
  - Two buffer pool pages for input, and one for output.

Example

A	B
10	a
20	b
20	c
40	d

R

A	C
50	e
20	f
20	g
30	h
40	i
50	j

S

A	B	C
20	b	f
20	b	g
20	c	f
20	c	g
40	d	i

$R \bowtie S$

- $|R| = 4$, $|S| = 6$, $P(R) = 2$, $P(S) = 3$.
- If R is the outer table, then the cost is 14.
- If S is the outer table, then the cost is 15.

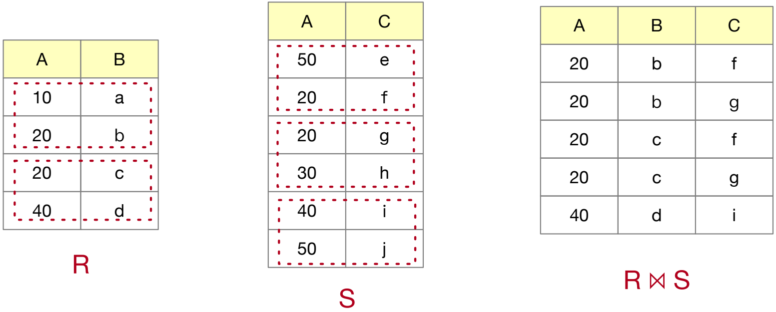
Blocked nested loop join

- Naive nested loop join is costly since for every tuple in the outer table R, we must do a sequentially scan of the inner table S.
- To maximize the utilization of buffer pool, we can process tables on a per-page basis, rather than on a per-tuple basis.

```
for each page $P_r$ in $R$ do
    for each page $P_s$ in $S$ do
        for each tuple $t_r$ in $P_r$ do
            for each tuple $t_s$ in $P_s$ do
                if $\theta(t_r, t_s)$ then
                    add $t_r \bowtie t_s$ to the result
```

Figure: Improved algorithm for $R \bowtie_\theta S$

Example



A	B
10	a
20	b
20	c
40	d

R

A	C
50	e
20	f
20	g
30	h
40	i
50	j

S

A	B	C
20	b	f
20	b	g
20	c	f
20	c	g
40	d	i

$R \bowtie S$

- $|R| = 4$, $|S| = 6$
- $P(R) = 2$, $P(S) = 3$.
- If R is the outer table, then the cost is 8.
- If S is the outer table, then the cost is 9.

Cost analysis

- Cost: $P(R) + P(R) * P(S)$
- Buffer pool requirement: $B = 3$

Optimization: If B pages are available in the buffer pool for the join operation, then

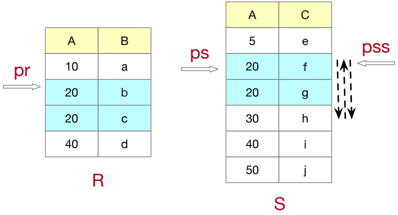
- $B - 2$ pages for scanning the outer table R
- One page for inner table scan
- The rest page for buffering the output
- Total cost: $P(R) + \lceil P(R)/(B-2) \rceil * P(S)$

Merge join

- Require equality predicate, e.g., equi-joins or natural joins.
- If R or S is not sorted by the join attributes, then sort it first.
- All tuples with the same value on the joined attributes are in consecutive order.
- Merge scan the sorted tables and emit tuples that match.

Merge join

```
/* ps/pr points to the first tuple of R/S */
while pr! = EOF & ps! = EOF do
  while $t_{pr}[A] < t_{ps}[A]$ do ++pr;
  while $t_{pr}[A] > t_{ps}[A]$ do ++ps;
  while $t_{pr}[A] = t_{ps}[A]$ do
    pss := ps; /* set pss to the first match */
    while $t_{pr}[A] = t_{pss}[A]$ do
      add $t_{pr} \bowtie t_{pss}$ to result;
      ++pss;
    ++pr;
  ps := pss; /* all matches processed, advance ps */
```



A	B
10	a
20	b
20	c
40	d

R

A	C
5	e
20	f
20	g
30	h
40	i
50	j

S

A	B	C
20	b	f
20	b	g
20	c	f
20	c	g
40	d	i

R ⋈ S

Cost analysis

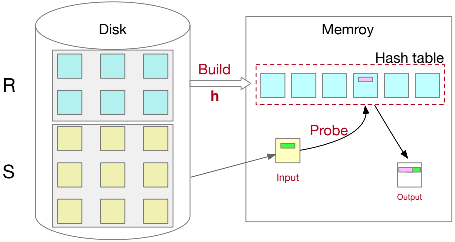
- Most cases: Sorting + $P(R) + P(S)$.
- Assumption: Every set of match candidates in S can fit in buffer pool.

- Worst case: Sorting + $P(R) + P(R) * P(S)$
- Assumption: Everything joins and $B = 3$.

Hash join

- Applicable for equi-joins and natural joins, e.g., $R \bowtie_{R.A=S.B} S$.
- If $t_1 \in R$ and $t_2 \in S$ can join, then they have the same value on the join attributes.
- Use a hash function h to partition both relations.
- Compute the join results on each partition.

Basic in-memory hash join



- Build phase: scan the outer table R and construct a hash table using a hash function h on the join attributes.
- Probe phase: scan the inner table S and use h on each tuple $t \in S$ to jump to the location in the hash table and find a matching tuple.
- Cost: $P(R) + P(S)$.
- Buffer pool requirement: $B \geqslant P(R) + 2$ or roughly the outer table R can fit in memory.

Hash join: partition phase

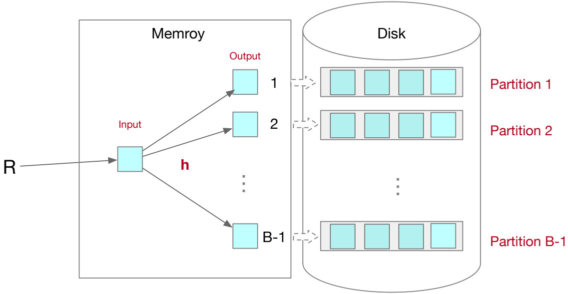
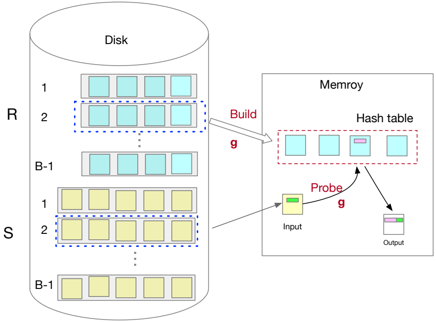


Figure: Partition R with h (need to do the same for S)

- Partition both R into $B-1$ partitions, using a hash function h on the join attributes.
- A buffer block/page is reserved as the output buffer for each partition.
- Partition table S in the same way.

Hash join: build & probe phase



- Read each partition R_i of R and build a hash table using another hash function g.
  - The hash functions g and h must be different. Why?
- Read the corresponding partition S_i of S in a per-page basis; then probe and join.
- R is the build relation and S is the probe relation.

Cost analysis

Assumption

- Partition phase divides table R into $(B-1)$ partitions evenly. That is, each partition of R has $\lceil P(R)/B-1 \rceil$ pages.
- Build & probe requires $\lceil P(R)/B-1 \rceil \leqslant B-2$, i.e., every partition of R fits into memory.
- $P(R) \leqslant (B-1)(B-2) \approx B^2$. Thus roughly $B \geqslant \sqrt{P(R)}$.
- We have no size requirement for the probe relation S.
  — Use the smaller input as the build relation R.

Cost: $3(P(R)+P(S))$

Question. What if a partition is too large for memory?

Hash-based algorithms

- Union, intersection, difference.
  - More or less like hash join.
- Duplicate elimination.
  - Eliminate duplicates within each partition.
- Group by aggregation.
  - (i) Apply the hash functions to the group-by columns.
  - (ii) Tuples in the same group will end up in the same partition.

Indexed nested loop join

```
for each tuple $t_r$ in $R$ do
    for each tuple $t_s$ in Index($t_r.A$) do
        add $t_r \bowtie t_s$ to the result
```

Figure: Algorithm for $R \bowtie_{R.A=S.B} S$, using an index of S on attribute B

- Cost analysis: $P(R) + |R| * C$.
- C is the I/O cost of an index lookup, which is $2 \sim 4$ I/O's typically.
- If both R and S support index lookup, better pick the smaller one as the outer relation.

Join algorithms (recap)

Algorithms	I/O costs
Naive Nested Loop Join	$P(R) + \|R\| * P(S)$
Block Nested Loop Join	$P(R) + P(R) * P(S)$
Indexed Nested Loop Join	$P(R) + \|R\| * C$
Merge Join	$P(R) + P(S)$
In-memory Hash Join	$P(R) + P(S)$
Hash Join	$3 * (P(R) + P(S))$

Table: Algorithms for $R \bowtie S$