Database Management Systems

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Sai University

Lecture 17, 25 October 2023

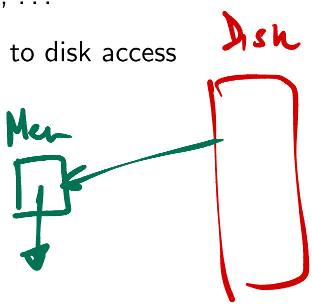
Query processing

- Translate the query from SQL into relational algebra
- Evaluate the relational algebra expression
- Challenges
  - Many equivalent relational algebra expressions

    $\sigma_{salary<75000}(\pi_{salary}(instructor))$ vs $\pi_{salary}(\sigma_{salary<75000}(instructor))$
  - Many ways to evaluate a given expression
- Query plan
  - Annotate the expression with a detailed evaluation strategy key values
    - Use index on *salary* to find instructors with $salary < 75000$
    - Or, scan entire relation, discard rows with $salary \geq 75000$

Query optimization

- Choose plan with lowest cost
- Maintain database catalogue — number of tuples in each relationn, size of tuples, ...
- Assess cost in terms of disk access and transfer, CPU time, ...
- For simplicity, ignore in-memory costs (CPU time), restrict to disk access
- Disk accesses
  - Relation r occupies b_r blocks
  - Disk seeks — time t_S per seek
  - Block transfers — time t_T per transfer
- Other factors — buffer management etc



Selection

(A1) Linear search

(A2) Clustering index, equality on key — index height h_i

(A3) Clustering index, equality on nonkey

(A4) Secondary index (key, non-key)

(A5) Clustering index, comparison — sorted on A

(A6) Clustering index, comparison — not sorted on A

(A7) Conjunctive selection using one index

(A8) Conjunctive selection using composite index

(A9) Conjunctive selection using intersection of pointers

(A10) Disjunctive selection by union of pointers

(Neg) Negation

Sorting

- In-memory sorting vs sorting on disk

Sorting

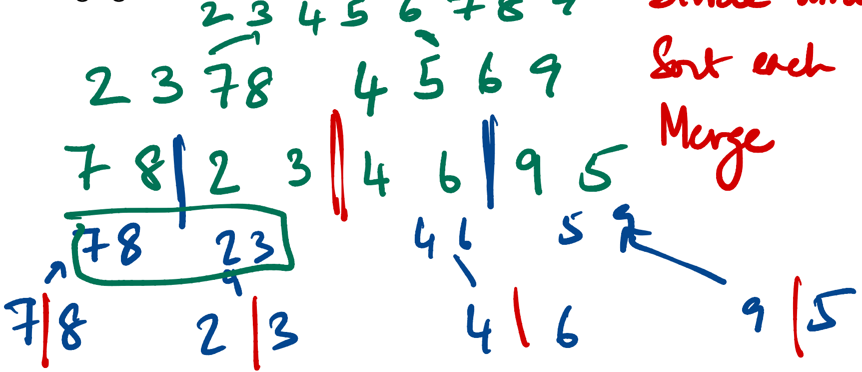
- In-memory sorting vs sorting on disk
- Merging sorted lists — varieties

Merge Sort

Divide array in 2 halves

Sort each half

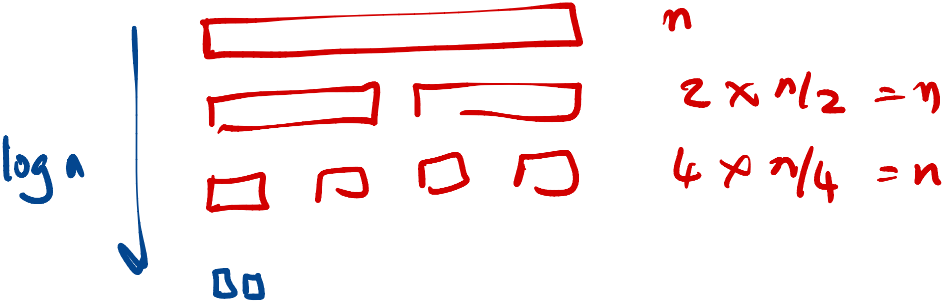
Merge



Sorting

- In-memory sorting vs sorting on disk
- Merging sorted lists — varieties

Merge is $O(n)$ Merge Sort is $O(n \log n)$



Sorting

- In-memory sorting vs sorting on disk
- Merging sorted lists — varieties

Merge has many versions

- Simple merge of merge sort
- Discard duplicates — "Union" of lists
- Detect duplicates & keep — "Intersection"
- Set/list difference

Sorting

- In-memory sorting vs sorting on disk
- Merging sorted lists — varieties
- Traditional merge sort

External merge sort

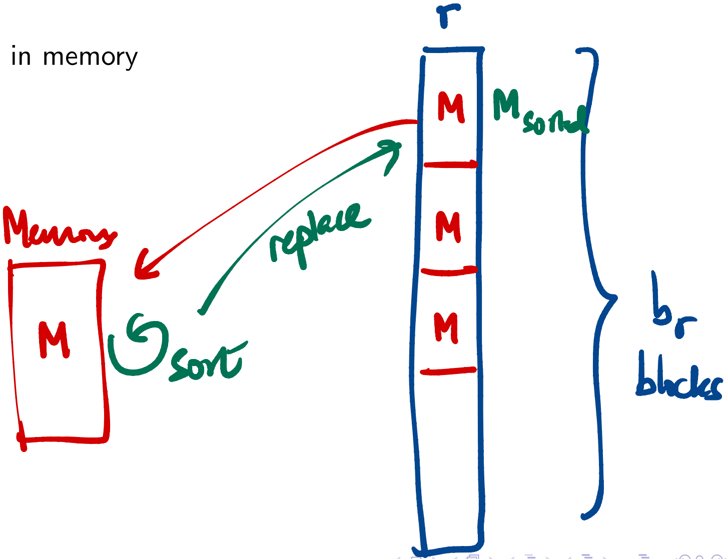
- N records, b_r blocks, M blocks in memory

$b_r >> M$

N / br records per block

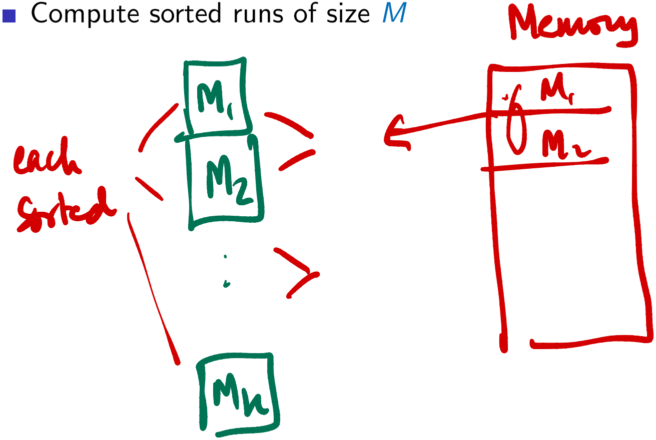
External merge sort

- N records, b_r blocks, M blocks in memory
- Compute sorted runs of size M



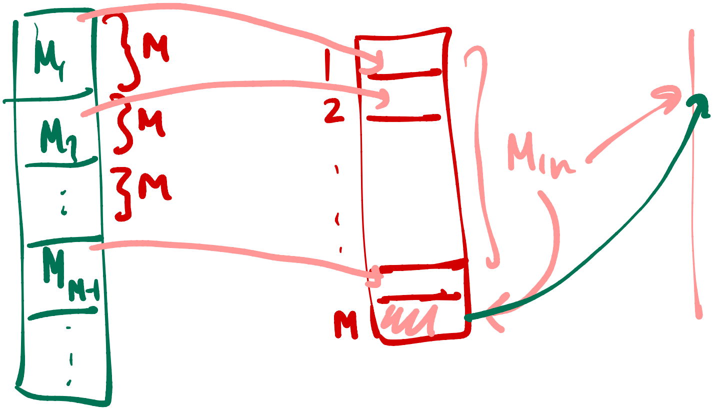
External merge sort

- N records, b_r blocks, M blocks in memory
- Compute sorted runs of size M



External merge sort

- N records, b_r blocks, M blocks in memory
- Compute sorted runs of size M



External merge sort

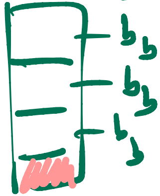
- N records, b_r blocks, M blocks in memory
- Compute sorted runs of size M
- Merge sorted runs, 1 block per run vs b_b blocks per run

— Optimizes seeks

Merge M-1 at a time

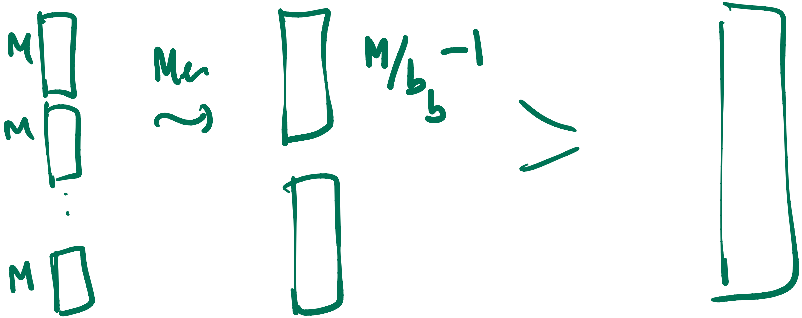
Merge $(M/b_b)-1$ at a time

Memory M



External merge sort

- N records, b_r blocks, M blocks in memory
- Compute sorted runs of size M
- Merge sorted runs, 1 block per run vs b_b blocks per run



External merge sort

- N records, b_r blocks, M blocks in memory
- Compute sorted runs of size M
- Merge sorted runs, 1 block per run vs b_b blocks per run
- Complexity
  - b_r/M sorted runs, $\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_r/M) \rceil$ merge passes

Initial

External merge sort

- N records, b_r blocks, M blocks in memory
- Compute sorted runs of size M
- Merge sorted runs, 1 block per run vs b_b blocks per run
- Complexity
  - b_r/M sorted runs, $\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_r/M) \rceil$ merge passes
  - Block transfers — $b_r\ (2\lceil \log_{\lfloor M/b_b \rfloor - 1}(br/M) \rceil + 1)$ +2 −1
    - Why not $b_r\ (2\lceil \log_{\lfloor M/b_b \rfloor - 1}(br/M) \rceil + 2)$?

Initially

Read r & create M sized sorted groups

$b_r + b_r = 2b_r$

Exclude last round of write

init, merge: $b_r(2 + 2\log b_r/M)$

External merge sort

- N records, b_r blocks, M blocks in memory
- Compute sorted runs of size M
- Merge sorted runs, 1 block per run vs b_b blocks per run
- Complexity
  - b_r/M sorted runs, $\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_r/M) \rceil$ merge passes
  - Block transfers — $b_r\ (2\lceil \log_{\lfloor M/b_b \rfloor - 1}(br/M) \rceil + 1)$
    - Why not $b_r\ (2\lceil \log_{\lfloor M/b_b \rfloor - 1}(br/M) \rceil + 2)$?
  - Block seeks — $2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil\ (2(\lceil \log_{\lfloor M/b_b \rfloor - 1}(br/M) \rceil - 1)$

Computing joins

- Running example
  - $Student \bowtie Takes$

Computing joins

- Running example
  - *Student* $\bowtie$ *Takes*
  - *Student* — 5000 rows, 100 blocks
  - *Takes* — 10000 rows, 400 blocks

Nested-loop join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)

r s

r for each row in Student

s for each row in Takes

Check join condition

Nested-loop join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_\theta s$ — r is outer relation, s is inner relation

Nested-loop join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_\theta s$ — r is outer relation, s is inner relation
  - Block transfers: $b_r + n_r \cdot b_s$

read outer relation once

for each row in r, one pass of s

Nested-loop join

- (5000 rows, 100 blocks) *Student* ⋈ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_{\theta} s$ — r is outer relation, s is inner relation
  - Block transfers: $b_r + n_r \cdot b_s$
  - Block seeks: $b_r + n_r$ — inner relation read sequentially

Nested-loop join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_\theta s$ — r is outer relation, s is inner relation
  - Block transfers: $b_r + n_r \cdot b_s$
  - Block seeks: $b_r + n_r$ — inner relation read sequentially
  - Special case: smaller relation fits in memory

Block nested-loop join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)

for each block A in r

for each block B in s

compare all rows in A vs all rows in B

Block nested-loop join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_\theta s$ — r is outer relation, s is inner relation

Block nested-loop join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_\theta s$ — r is outer relation, s is inner relation
  - Block transfers: $b_r + b_r \cdot b_s$

$b_r + n_r \cdot b_s$

$5000 \times b_s$

vs $100 \times b_s$

Block nested-loop join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_\theta s$ — r is outer relation, s is inner relation
  - Block transfers: $b_r + b_r \cdot b_s$
  - Block seeks: $b_r + b_r = 2b_r$

$b_r + n_r$

outer

one seek of s for each block in r

Use catalogue to check which relation to use as outer / inner

Indexed nested-loop join

- (5000 rows, 100 blocks) *Student* ⋈ *Takes* (10000 rows, 400 blocks)

Python – list intersection of l_1 & l_2

```
for x in l1:
  for y in l2:
    if x==y
      ~
```

```
d1 = {}
for x in l1:
   d1[x] = True

for y in l2:
   if y in d1:
```

Access of d1 is "efficient"

Indexed nested-loop join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_\theta s$ — r is outer relation, s is inner relation

for each row in r

check r vs index in s

Indexed nested-loop join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_\theta s$ — r is outer relation, s is inner relation
  - Total cost: $b_r(t_T + t_S) + n_r \cdot c$
    - c is cost of single selection on s

processing r

one lookup in s for each row in r

Merge join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)

Merge join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_\theta s$ — r is outer relation, s is inner relation

Merge join

- (5000 rows, 100 blocks) *Student* ⋈ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_\theta s$ — r is outer relation, s is inner relation
  - Block transfers: $b_r + b_s$

Merge is linear time

Merge join

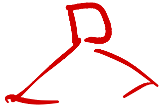
- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_\theta s$ — r is outer relation, s is inner relation
  - Block transfers: $b_r + b_s$
  - Block seeks: $\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$

Requires r & s to be sorted

Merge join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Complexity
  - $r \bowtie_\theta s$ — r is outer relation, s is inner relation
  - Block transfers: $b_r + b_s$
  - Block seeks: $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$
- Hybrid merge join using secondary index

B+ tree on salary



If

Merge r with leaves of B+tree of s

sorted

Not in disk order

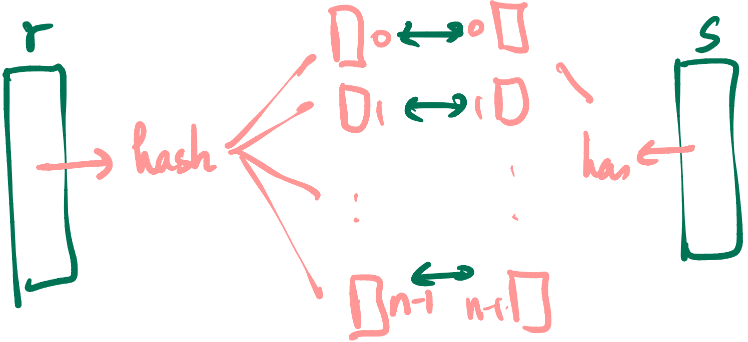
leaves

Sorted by salary

Hash join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)

Hash function h: input keys $\longrightarrow \{0, \ldots, n-1\}$



Hash join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Build input and probe input

Matching block r_i & s_i (i^{th} hash block on both sides)

Build reln || Pick s_i & build an index (in memory)

Probe reln || For each row in r_i, check against s_i index

Hash join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Build input and probe input
- Complexity

Hash join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Build input and probe input
- Complexity
  - $r \bowtie_\theta s$ — s is build relation, r is probe relation

Hash join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Build input and probe input
- Complexity
  - $r \bowtie_\theta s$ — s is build relation, r is probe relation
  - Block transfers: $3(b_r + b_s) + 4n_h$

n_h = # of hash buckets

At most n_h fractional blocks

? — fractional wastage

Ignore output cost

$b_r + b_s$ read & compute hash

$n_h + n_h + b_r + b_s$ to write hash buckets

$n_h + n_h + b_r + b_s$ to build & probe

Hash join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Build input and probe input
- Complexity
  - $r \bowtie_\theta s$ — s is build relation, r is probe relation
  - Block transfers: $3(b_r + b_s) + 4n_h$
  - Block seeks: $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$

Hash join

- (5000 rows, 100 blocks) *Student* $\bowtie$ *Takes* (10000 rows, 400 blocks)
- Build input and probe input
- Complexity
  - $r \bowtie_\theta s$ — s is build relation, r is probe relation
  - Block transfers: $3(b_r + b_s) + 4n_h$
  - Block seeks: $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$
- Recursive partitioning — Hash buckets exceed memory

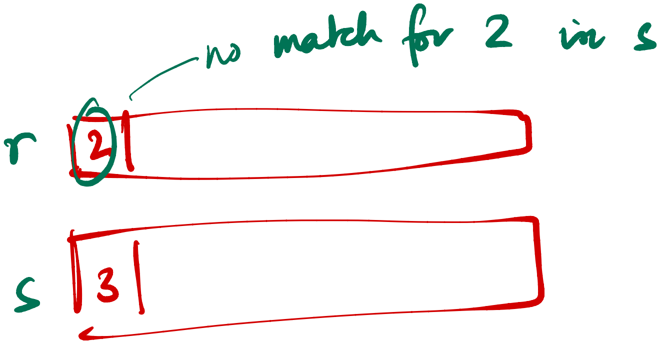
Computing outer joins

r ⟕ s

if a row in r does not match any row in s,
output with default values in s columns

Computing outer joins

- Using merge join



Computing outer joins

- Using merge join
- Using hash join — probe vs build case

probe r_i against index on s_i

Other operations

- Duplicate removal

Other operations

- Duplicate removal
- Aggregate queries with grouping

Other operations

- Duplicate removal
- Aggregrate queries with grouping
  - Aggregate while sorting/hashing

Other operations

- Duplicate removal — merge
- Aggregrate queries with grouping — merge
  - Aggregate while sorting/hashing
- Set theoretic operations — merge