CS-300: Data-Intensive Systems

Query Processing with Relational Operations (part 2)

(Chapters 15.5.3 - 15.5.6 & 15.6.3 & 15.6.5)

Prof. Anastasia Ailamaki, ***Prof. Sanidhya Kashyap***

EPFL

Recap

Simple selection:

Operation of choosing or filtering rows from a relation based on specific criterion

- Of the form: $\sigma_{\text{R.attr } \mathbf{op} \text{ value}}$ (R)

- Best approach to implement depends on:
  - Available index / access paths
  - Expected size of the result (# tuples / # pages)

```
SELECT *
  FROM Reserves R
  WHERE R.rname < 'C%'
```

- Size of result approximated as

  size of R * *reduction factor*

  - "Reduction factor" also known as **selectivity**
  - Selectivity estimate is based on statistics

General selection conditions

Selection condition is a combination of conditions using AND / OR

```
(day < 8/9/94 AND rname=`Paul') OR bid=5 OR sid=3
```

1. First convert to conjunctive normal form (CNF) (collection of ANDs) →

```
(day < 8/9/94 OR bid=5 OR sid=3) AND (rname=`Paul' OR bid=5 OR sid=3)
```

1. Match conditions to indexes

- A B+Tree index **works best** for prefix (start) of the indexed attributes
  - Index on `<a, b, c>` `matches a=5 AND b=3`, but not `b=3`
- Hash index must have all attributes in search keys; only helpful in super-fast equality lookups

Selection: 1st approach: Example

- Consider: **(day < 8/9/94 AND rname=`Paul') OR bid=5 OR sid=3**

Questions: What happens if we have indexes with different attributes orders

- a B+Tree on <rname, day>?
  - First sort by **rname** and then **day**
  - Good choice as equality on **rname**, and then range scan on **day** using B+tree index
- A B+Tree on <day, rname>?
  - First range scan on day
  - Cannot do equality lookup on rname (scan all tuples for day and then filter rname)
- A hash index on <day, rname>?
  - A bad choice here as we have to generate all possible rnames

Selection: 1st approach: Example

- Consider: **(day < 8/9/94 AND rname=`Paul') OR bid=5 OR sid=3**

Questions: What happens if we have indexes with different attributes orders

- CNF: **(day < 8/9/94 OR bid=5 OR sid=3) AND (rname=`Paul' OR bid=5 OR sid=3)**

Q. a B+Tree on <rname, day>?

Selection: 1st approach: Example

- Consider: **(day < 8/9/94 AND rname=`Paul') OR bid=5 OR sid=3**

Questions: What happens if we have indexes with different attributes orders

- CNF: **(day < 8/9/94 OR bid=5 OR sid=3) AND (rname=`Paul' OR bid=5 OR sid=3)**

Q. a B+Tree on <rname, day>?

- Clause 1: **day < 8/9/94 OR bid=5 OR sid=3**
  - No condition on rname for first clause
  - Index scans all rname values
- Clause 2: **rname=`Paul' OR bid=5 OR sid=3**
  - index can find all entries where rname=Paul

Selection: 1st approach: Example

- Consider: **(day < 8/9/94 AND rname=`Paul') OR bid=5 OR sid=3**

Questions: What happens if we have indexes with different attributes orders

- CNF: **(day < 8/9/94 OR bid=5 OR sid=3) AND (rname=`Paul' OR bid=5 OR sid=3)**

Q. a B+Tree on <day, rname>?

- Range scan on day in first clause but not helpful for the second clause

Today's focus

- Joins
  - Avoid enumeration using index/partition
    - Index nested loop
    - Hash join
    - Sort-merge join
- General joins and aggregates

Joins

- Combines two relations: R⋈S
- Combine multi-joins using a pair-wise joins from left to right
- At a high-level: combination of relation product followed by a selection
  - Inefficient as large intermediate results
- Join operator algorithms:
  - One pass algorithms
  - Index-based algorithms
  - Two-pass algorithms
- Join techniques:
  - Nested-loop joins, sort-merge join, hash join

Simple nested loops join

- For each tuple in the **outer** relation **R**, scan the entire **inner** relation **S**

```
foreach tuple r in R do
  foreach tuple s in S do
    if r_i==s_j then add<r,s> to result
```

- If we assume:
  - M pages in R, p_R tuples per page
  - N pages in S, p_S tuples per page
- IO (read) cost: M + ((M * p_R) * N)

  (Assume: No CPU cost and output writing cost)

- An index on the join column on one relation (**inner S**) can now exploit the index

→ IO (read) cost: M + ((M * p_R) * **cost of finding matching S tuples**)

Simple nested loops join

- IO (read) cost: M + ((M * p_R) * cost of finding matching S tuples)

- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree

- **Clustering** determines S tuples cost (asume Alt. (2) or (3) for data entries):

→ **Clustered index: 1 I/O per page of matching S tuples**

→ **Unclustered: can be as high as 1 I/O per matching S tuple**

Schema for examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Similar to old schema; *rname* added for variations.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages
  - N=500, p_S=80
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages
  - M=1000, p_R=100

```
foreach tuple r in R do
  foreach tuple s in S do
    if r_i==s_j then add<r,s> to result
```

```
SELECT *
FROM Reserves R
WHERE R1.sid=S1.sid
```

Examples of index-nested loops (1/2)

```
SELECT *
FROM Reserves R
WHERE R1.sid=S1.sid
```

- Hash-index (Alt. 2) on *sid* of Sailors (inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples
  - For each Reserves tuple:
    - 1.2 I/Os to get data entry in index (hash index overhead)
    - plus 1 I/O to get (the exactly one) matching Sailors tuple (data page retrieval)

- Cost = M + (($M*p_R$) * cost of finding S tuples)
  - Cost: 1000 + 1000 * 100 * 2.2 = 221,000 I/Os
  - At 10 msec/IO: **36 mins**

Examples of index-nested loops (2/2)

```
SELECT *
FROM Reserves R
WHERE R1.sid=S1.sid
```

- Hash-index (Alt. 2) on *sid* of Reserves (inner):
  - Scan Sailors: 500 page I/Os, 80*500 tuples
  - For each Sailors tuple:
    - 1.2 I/Os to find index page with data entries (hash lookup overhead)
    - Plus cost of retrieving matching Reserves tuples
    - Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000).
    - Retrieval cost: 1 I/O (clustered) or 2.5 I/Os (unclustered)

- Cost = M + ((M*p_S) * cost of finding R tuples)
  - Cost: 500 + 500 * 80 * (1.2 + 2.5) = 148,500 I/Os
  - At 10 msec/IO: **24 mins**

Page-oriented nested loops join

- Process data **page-by-page** than tuple-by-tuple

```
foreach page b_R in R do
 foreach page b_S in S do
  foreach tuple r in b_R do
   foreach tuple s in b_S do
    if r_i==s_j then add<r,s> to result
```

- For each page of outer **R**
  - Scan all the pages of inner **S**
  - For each pair of pages (one from **R** and one from **S**), match tuples
  - If tuples match (<**r**,**s**>: **r** is in **R**-page and **s** is in **S**-page), output them

- IO Cost: M + M * N
  - M: Reading each page from **R** exactly once
  - M*N: Reading all pages from **S** for every page of **R**

Block nested loop joins: use available buffers

- Page-oriented nested loop does not use extra buffers
- Buffer allocation: 1 page for `S(inner)`, 1 for output and the remaining for `R(outer)`($block_R$)
- Steps:
  - Read multiple pages of R into memory
  - For this block R, scan each page of S and find matching tuples
  - Output matched tuple

```
foreach block block_R in R do
 foreach page b_S in S do
  foreach tuple r in block_R do
   foreach tuple s in b_S do
    if r_i==s_j then add<r,s> to result
```

- IO Cost = M + (# outer blocks * N (inner scan))
  - # outer blocks = ⌈# outer pages / blocksize⌉

Nested loops: IO Cost

- `R(outer)` is Reserves and `S(inner)` is Sailors
  - M = 1000, p_R = 100
  - N = 500, p_S = 80
- Indexed nested loop (does not use extra pages)
  Cost = M + ((M*p_R) * cost of finding S tuples)
  Cost: 500 + 500 * 80 * (1.2 + 2.5) = 148,500 I/Os, **24 mins**
- Blocked nested loop with blocksize = 100:
  Cost of scanning R is 1000 I/Os; a total of 10 *blocks*
  Per block of R, we scan Sailors (S); 10*500 = 5000 I/Os
  Total cost: 6000 I/Os, ***1 minute!***
- Page-oriented nested loop
  Cost = M + M * N = 501,000 I/Os, ***1.5 hours!!***
- Simple nested loop
  Cost = M + (p_R * M) * N = 40,000,500 I/Os, ***111 hours!!***

```
SELECT *
FROM Reserves R
WHERE R1.sid=S1.sid
```

When to use index nested loop?

- Great for queries with highly selective predicates

  *select * from sailors, reserves*

  where sailors.sid = reserves.sid and sailors.sid = 100

- Looking only for one **SID** => Scan filters out most tuples
- Assume unsorted Sailors table, hash index on Reserves.sid
- **Index nested loop** = 503.7 I/Os (5s)

  500 (scan Sailors – worst case) + 1.2 (index lookup) + 2.5 (Reserves lookup) = 503.7 I/Os

- **Blocked nested loop** (bsize = 100): 1500 I/Os (55s)

  500 (scan Sailors – worst case) + 1 (#matching blocks)*1000 = 1500 I/Os

Today's focus

- Joins
  - Avoid enumeration using index/partition
    - Index nested loop
    - Hash join
    - Sort-merge join
- General joins and aggregates

Hash join

- Hash join: R⋈S

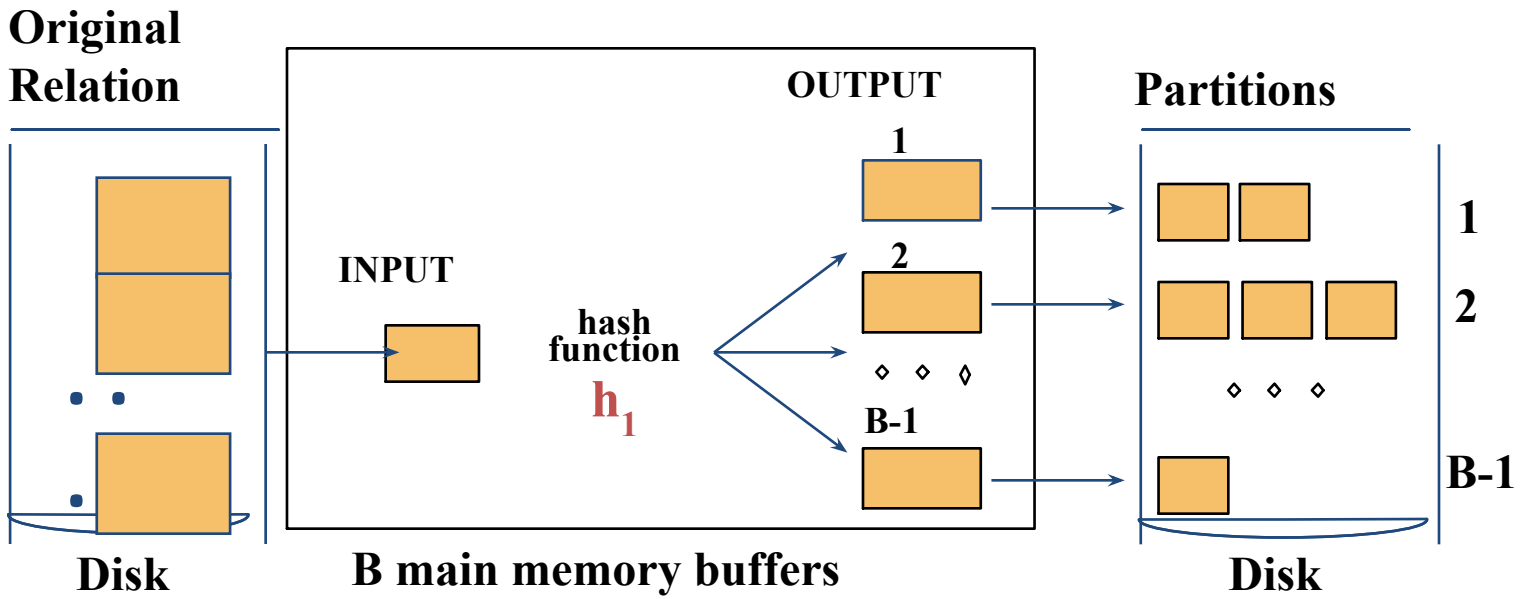
```
Read S in memory and build a hash index on it
  foreach tuple r in R do
    Use the hash index on S to find tuples (S.a = r.a)
```

- Cost: M (outer scan) + N (inner scan)
- Assumptions:
  - Enough memory to hold S (N pages) + hashtable
  - Smaller relation used to build the hash table for lower memory usage

What if we do not have enough memory?

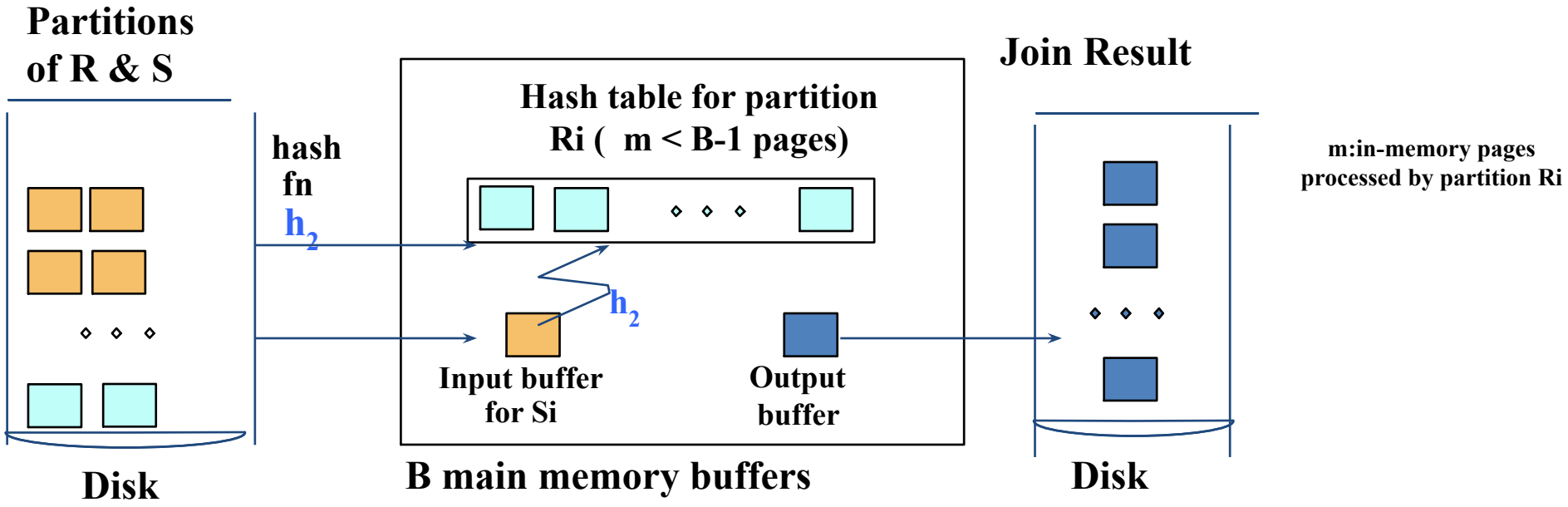
Two-pass hash join: Phase 1

- Read the relation *R* block by block, partition it with $h_1(a)$
  - Create one partition for each possible value of $h_1(a)$
  - Write the partitions to disk
  - R gets partitioned into R_1, R_2, …, R_k where k = B - 1
- Similarly, read and partition S



Two-pass hash join: Phase 2

- Read whole R1, and build a hash index on it with $h_2(a)$
- Read S1 page by page, and use the hash index to find matches.
- Repeat for R2， S2, and so on.



Two-pass hash join: Cost

- First pass creates B-1 partitions, each of size R_i = M/(B-1)
- Need each R_i < B-1 in order to fit in memory for 2nd pass
  - 1 page for S_i, 1 page for output, B-2 pages for R_i

→ Need f * M/(B-1) ≤ B-2

… or, roughly: B > $\sqrt{f * M}$

where f is fudge factor → accounts for uneven distribution (data skew)

- **IO Cost: 3 (M + N)**
  - In partitioning phase, read+write both relns; 2(M+N)
  - In matching phase, read both relns; M+N I/Os

Nested loops: IO Cost

- `R(outer)` is Reserves and `S(inner)` is Sailors
  - M = 1000, p_R = 100; N = 500, p_S = 80
- Two-pass hash join (also called *Grace Join)*
  - Cost = 3 (M+N) = 3 * 1500 = 4500 I/Os = ***45 secs!!!***
- Simple nested loop
  - Cost = M + (p_R * M) * N = 40,000,500 I/Os, ***111 hours!!***
- Page-oriented nested loop
  - Cost = M + M * N = 501,000 I/Os, ***1.5 hours!!***
- Indexed nested loop (does not use extra pages)
  - Cost = M + ((M*p_R) * cost(index lookup)) = 148,500 I/Os, **24 mins**
- Blocked nested loop with block size = 100:
  - Total cost: 6000 I/Os, ***1 minute!***

```
SELECT *
FROM Reserves R
WHERE R1.sid=S1.sid
```

Hash join issues

- Requires R partition to fit in memory: B > $\sqrt{f * M}$
- Not enough memory?
  - *Recursive Partitioning*
  - Rarely used, but can be done

- Hash function is bad?
  - Assume *h1(a) = a % 100;* And all *values of a* are multiple of 100
    - So *h1(a) is always = 0*

→ **Called *hash-table overflow***

- Overflow avoidance: Use a good hash function
- Overflow resolution: Repartition using a different hash function

Hybrid hash join algorithm

- Two phase hash join partitions both relations and writes them directly to disk
- Hybrid hash join minimizes the reads/writes to disk

- Idea: If extra memory available, **keep one partition in memory during Phase 1**
  - Hybrid of *one-pass hash join* (for one partition) and *two-pass hash join* for the remaining partitions

Hybrid hash join algorithm

Suppose

… **`B > f * M / k`**

- Create k partitions of S
- During partitioning process, B – (k + 1) pages not in use
  (k pages for partitions, 1 extra page for input)

… and **`B – (k + 1) > f * M / k`**

- Enough memory to hold one whole partition

Phase 1 (partitioning phase)

- Partition S, but do not write out first partition
- Partition R, directly join first partition R1 with S1

Phase 2 (probing phase)

- Join remaining k -1 partitions

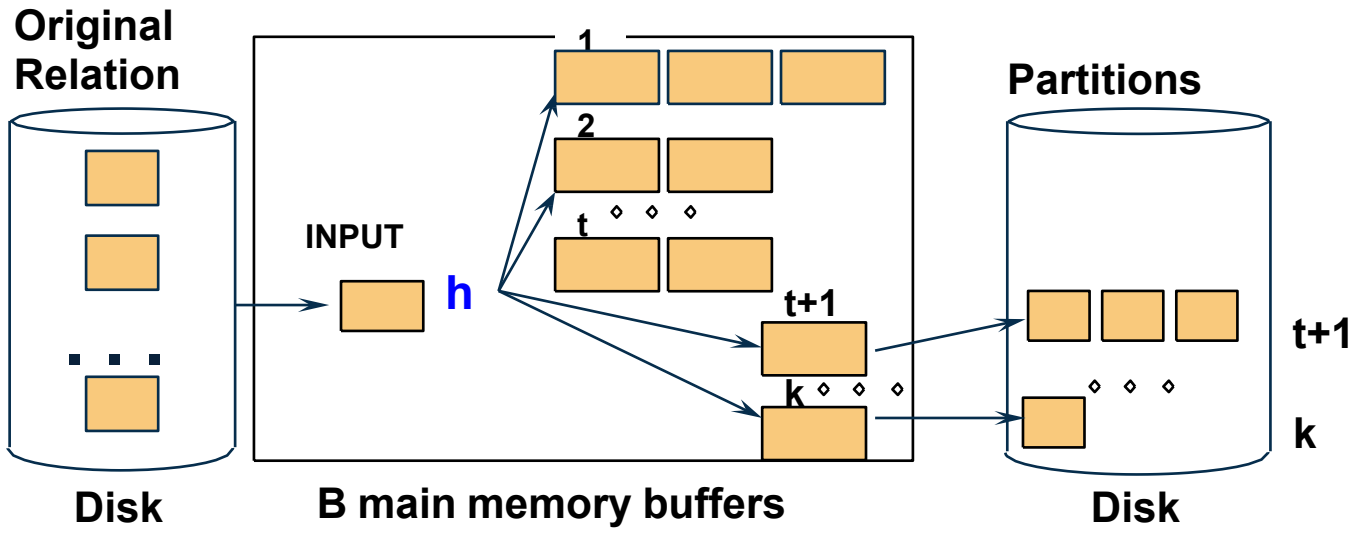
Hybrid hash join algorithm

Generalize:

1. Partition S into k buckets
   - t buckets S_1 , …, S_t stay in memory
   - Rest k-t buckets S_{t+1}, …, S_k to disk
2. Partition R into k buckets
   - First t buckets join immediately with S
   - Rest k-t buckets go to disk
3. Finally, join k-t pairs of buckets:
   - (R_{t+1},S_{t+1}), (R_{t+2},S_{t+2}), …, (R_k,S_k)

Step	What Happens?	Benefit
1	Partition S (smaller relation)	Keep **t buckets** in memory
2	Partition R (larger relation)	Join **t buckets** immediately
3	Join remaining (k-t) buckets from disk	Fewer disk operations overall

Hybrid hash join algorithm



How to choose k and t?

Hybrid hash join algorithm

- How to choose k and t?
  - Choose k large so that

One page/bucket in memory

k <= B

Hybrid hash join algorithm

- How to choose k and t?
  - Choose k large so that
  - Choose t/k large so that

One page per bucket in memory

k <= B

First t buckets in memory

t/k * N <= B

Hybrid hash join algorithm

- How to choose k and t?
  - Choose k large so that

One page per bucket in memory

$k <= B$

  - Choose t/k large so that

First t buckets in memory

$t/k * N <= B$

  - Together:

One page per disk partition

$t/k * N + k - t <= B$

Hybrid hash join algorithm

- How to choose k and t?
  - Choose k large so that

    One page per bucket in memory

    $k <= B$
  - Choose t/k large so that

    First t buckets in memory

    $t/k * N <= B$
  - Together:
    - Assuming $t/k * N >> k - t$:

      One page per disk partition

      $t/k = B/N$

      $t/k * N + k - t <= B$

Hybrid hash join algorithm

- Even better: adjust t dynamically

- Start with t = k: all buckets are in main memory
- Read blocks from S, insert tuples into buckets
- When out of memory:
  - Send one bucket to disk
    - t := t-1

- Worst case:
  - All buckets are sent to disk (t=0)
  - Hybrid Join becomes Grace Join

Hybrid hash join: Cost

- Cost of Hybrid Join:

- Grace Join: 3M + 3N

- Hybrid Join:
  - Saves 2 I/Os for t/k fraction of buckets
  - Total 2t/k(M+N) I/O
  - Cost:
    - (3-2t/k)(M+N) = (3-2B/N)(M+N)

Hybrid hash join: Benefit

- It degrades gracefully when S larger than B:

- When N <= B
  - Main memory hash-join has cost M + N

- When N > B
  - Grace-join has cost 3M + 3N
  - Hybrid join has cost (3-2t/k)(M + N)

Hybrid hash join: Example

Example: Assume B = 300

- Sailors (N)= 500, Reserves (M) = 1000

- Phase 1
  - Partition Sailors into S1, S2. Keep S1 in memory
    - Cost = 500 + 250 = 750
  - Partition Reserves into R1, R2. Probe with R1. Write R2.
    - Cost = 1000 + 500 = 1500
- Phase 2:
  - S1, R1 join is done
  - Scan S2, R2 and join
  - Cost: 250 + 500 = 750
- Hybrid hash join cost = 750 + 1500 + 750 = 3000 I/Os
- Hash join cost = 3 (1500 + 750) = 4500 I/Os

Today's focus

- Joins
  - Avoid enumeration using index/partition
    - Index nested loop
    - Hash join
    - Sort-merge join
- General joins and aggregates

Sort-merge join

- Sort-merge join: R ⋈ S
  - Scan R and sort
  - Scan S and sort
  - Merge R and S

- Useful if
  - one or both inputs are already sorted on join attribute(s)
  - output is required to be sorted on join attributes(s)

One-pass sort-merge join

- *Merge Steps:*
  - Scan relation looking for qualifying/matching tuples
  - Join R and S tuples in partition
  - Advance scan to next partition & repeat

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

One-pass sort-merge join: Cost

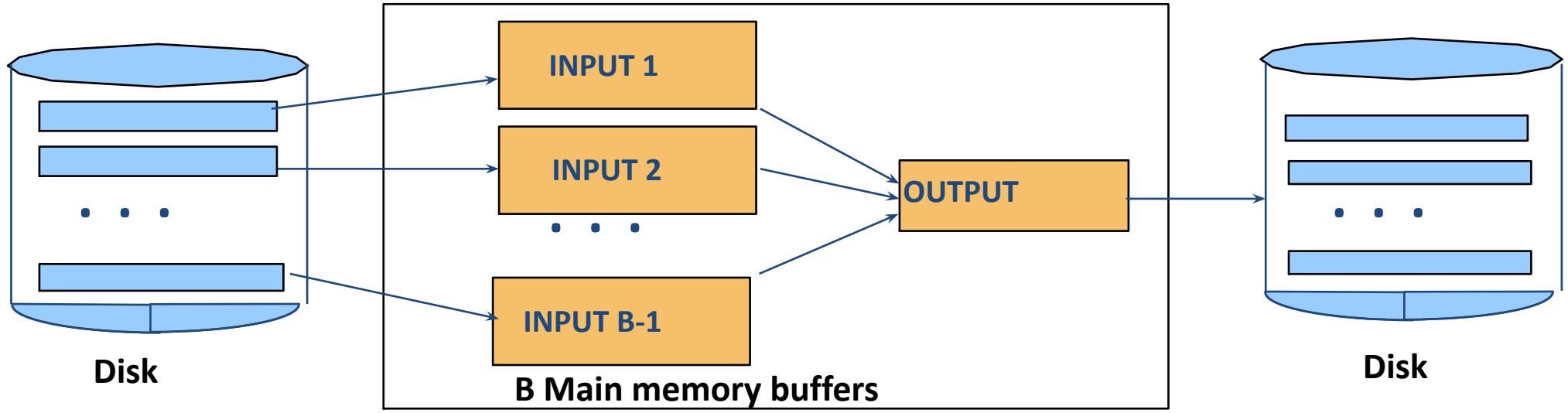
- Sort-merge join: R ⋈ S
  - Scan R and sort in main memory
  - Scan S and sort in main memory
  - Merge R and S
- Cost: M (outer scan) + N (inner scan)

- One-pass assumption
  - Enough memory to store M + N
- But, this is typically NOT a one pass algorithm
  - Available memory usually < M + N

- Use external merge sort for sorting

General external merge-sort: Recap

- To sort a file with *N* pages using *B* buffer pages:
  - Pass 0: use *B* buffer pages. Produce $\lceil N / B \rceil$ sorted runs of *B* pages each.
  - Pass 1, 2, …, etc.: merge *B-1* runs.



Number of passes: $1+\lceil \log_{B-1} \lceil N / B \rceil \rceil$

Cost = 2N * (# of passes)

Multi-pass merge-sort join: Cost & example

Cost: Sort R + Sort S + (M+N)

- With 100 buffer pages
  - Can sort both Reserves and Sailors in 2 passes
  - Cost of sorting Reserves = 2M * 2 = 2 * 2 * 1000 = 4000
  - Cost of sorting Sailors = 2N * 2 = 2 * 2 * 500 = 2000
  - Cost of merging = 1000 (read sorted Reserves) + 500 (read sorted Sailors) = M + N
  - Total cost = 5 (M + N) = 7500 I/Os

- Note: cost of merge could be as bad as M * N
  - All tuples in R and S have same value of join attribute (unlikely)

Sort-merge join refinement

- Original approach: Sort R; Sort S; Merge sorted R and S (one final pass)
- **Idea: Combine final sorting pass** of both R and S with the **join (merge)** step

- Do the join during final merging pass of sort
  - Read R and write out sorted runs (pass 0)
  - Read S and write out sorted runs (pass 0)
  - Merge R-runs and S-runs, while finding R ⋈ S matches

- 2-pass cost = 3 * R + 3 * S = 3000 + 1500 = 4500
- Need 1-page from each R and S in memory
  - Requires B >= √R + √S

Sort-merge join vs hash join

- Given a minimum amount of memory both have a cost of 3(M+N) I/Os
- Hash Join Pros:
  - Superior if relation sizes differ greatly
  - Shown to be highly parallelizable*
- Sort-Merge Join Pros:
  - Less sensitive to data skew
  - Result is sorted (may help “upstream” operators)
  - Goes faster if one or both inputs already sorted

* Hash join parallelization is beyond the scope of this class.

Today's focus

- Joins
  - Avoid enumeration using index/partition
    - Index nested loop
    - Hash join
    - Sort-merge join
- **General joins and aggregates**

General join conditions

Equality over several attributes (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):

- For Index NL, build index on *<sid, sname>* (if S is inner); or use existing indexes on *sid* or *sname*
- For Sort-Merge and Hash Join, sort/partition on combination of the two join columns
- Other joins unaffected

Inequality conditions (e.g., *R.rname < S.sname*):

- For Index NL, need (clustered!) B+ tree index
  - Range probes on inner; # matches likely to be much higher than for equality joins
- Hash Join not applicable!
- Block NL quite likely to be the best join method here

Set operations (union/intersection/difference)

- Intersection and cross-product special cases of join
- Union (Distinct) and Difference similar; we'll do union:
- Sorting based approach to union:
  - Sort both relations (on combination of all attributes)
  - Scan sorted relations and merge them
  - *Alternative*: Merge runs from Pass 0 for *both* relations

- Hash based approach to union:
  - Partition R and S using hash function *h*
  - For each S-partition, build in-memory hash table (using *h2*), scan corresponding R-partition and add tuples to table while discarding duplicates

Aggregate operators (AVG, MIN …) 1/2

- Without grouping:
  - In general, requires scanning the relation
  - If index exists (search key on all attributes in the SELECT or WHERE clauses): index-only scan

- With grouping (i.e., GROUP BY clause): *compute aggregate on each group separately*
  - Sorting-based approach:
    - Sort based on group-by attributes
    - Scan sorted data and compute aggregate for each group
    - Can improve this by combining sorting and aggregate computation
  - Similar approach for hashing on group-by attributes
    - Partition relation using a hash function based on the group-by attributes
    - For each partition, compute aggregate in memory → handles groups by keeping each group together

Aggregate operators (AVG, MIN …) 2/2

- Without grouping:
  - In general, requires scanning the relation
  - If index exists (search key on all attributes in the SELECT or WHERE clauses): index-only scan

- With grouping (i.e., GROUP BY clause): *compute aggregate on each group separately*
  - Tree-index:
    - Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, we can do index-only scan
    - If group-by attributes form prefix of the search key → retrieve data entries/tuples in group-by order

Impact of buffering

- Concurrent operations and buffer estimation:
  - Estimating the number of available buffer pages is guesswork
    - Resource use is dynamic and changes with workloads

- Repeated access patterns interact with buffer replacement policy
  - E.g., Inner relation is scanned repeatedly in Nested Loop Join
    - MRU is best, LRU is worst (*sequential flooding*)

- Implementation choice impacts buffer management
- Optimizing access path can help improve cache use
  - Index nested loop can be better by sorting outer relation

Summary

- A virtue of relational DBMSs:
  - Queries are composed of a few basic operators
  - Implementation of operators can be carefully tuned
- Many alternative implementations for each operator
  - No universally superior technique for most operators
- Must consider alternatives for each operation in a query and choose best one based on system statistics…
  - Part of the broader task of optimizing a query composed of several operations