Midterm Review

CMPSCI 445
Spring 2018

Exam Thursday

- Please arrive promptly (or early if you can).
- Sit closer to door if you think you might finish early.
- Don’t sit with your friends.
- No notes, books, electronic devices.
- No blue books.
- Just bring a pen or pencil.

A note on sample exam

- The SQL questions on the exam spend more time on basic SQL features, but ignore advanced stuff (data cubes, window functions, json)
- That advanced stuff is fair game. Expect some questions on it (like Homework 4)

Topics on the exam

Topic	Textbook Reference
Relational algebra	Ch 4
SQL review, SQL advanced	Ch 5 + slides from class
XML, JSON, XPath queries	27.6 + slides
Indexes and access methods	Ch 8, 9, 10, 11
Sorting and join algorithms	Ch 12, 13, 14

Review topics

- Relational algebra
- Nested queries
- Advanced SQL
- XML/XPath, JSON
- Access methods
- Sorting and Join Algorithms

Relational Algebra

Relational Algebra

- Operates on relations, i.e. *sets*
  - Later: we discuss how to extend this to *bags*
- Five basic operators:
  - Union: $\cup$
  - Difference: -
  - Selection: σ
  - Projection: Π
  - Cartesian Product: $\times$
- Derived or auxiliary operators:
  - Intersection, complement
  - Joins (natural, equi-join, theta join)
  - Renaming: ρ

Query equivalence

Definition: **Query Equivalence**

Two queries Q and Q' are equivalent if:

for all databases D, Q(D) = Q'(D)

Query Optimization Is Based on Algebraic Equivalences

- Relational algebra has laws of commutativity, associativity, etc. that imply certain expressions are **equivalent**.
- They may be different in cost of evaluation!

$$\sigma_{c \wedge d}(R) \equiv \sigma_c(\ \sigma_d(R)\ )$$ cascading selection

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T)$$ join associativity

$$\sigma_c (R \bowtie S) \equiv \sigma_c(R) \bowtie S$$ pushing selections

- Query optimization finds the most efficient representation to evaluate (or one that's not bad)

Nested queries

Nested queries

- A **nested query** is a query with another query embedded within it.
- The embedded query is called the **subquery**.
- The subquery usually appears in the WHERE clause:

```
SELECT   S.sname
FROM     Sailors S
WHERE    S.sid IN ( SELECT R.sid
                    FROM Reserves R
                    WHERE R.bid = 103 )
```

(Subqueries also possible in FROM or HAVING clause.)

Correlated subquery

- If the inner subquery depends on tables mentioned in the outer query then it is a **correlated subquery.**
- In terms of conceptual evaluation, we must recompute subquery for each row of outer query.

Correlation

```
SELECT    S.sname
FROM      Sailors S
WHERE     EXISTS (   SELECT *
                     FROM    Reserves R
                     WHERE  R.bid = 103
                     AND R.sid = S.sid )
```

Example

What does this query compute?

```
SELECT  S.sid, S.name
FROM    Sailors S
WHERE   S.rating >= ALL (SELECT S2.rating
                         FROM Sailors S2 )
```

- Find the sailors with the highest rating

exercise

- Emp(<u>eid</u>, ename, age, salary)
- Works(<u>eid, did</u>)
- Dept(<u>did</u>, dname, budget, managerid)

```
SELECT DISTINCT D.managerid
FROM Dept D
WHERE 1000000 < ALL (SELECT D2.budget
                     FROM Dept D2
                     WHERE D2.managerid = D.managerid )
```

Describe in English the result of this query

Example schema

- **Sales(item-name, color, size, number)**
  - **item-name**: {skirt, dress, shirt, pant}
  - **color**: {dark, pastel, white}
  - **size**: {small, medium, large}
  - **number**: number of units sold
- **Measure attributes**: measure some value; can be aggregated.
- **Dimension attributes**: define the dimensions on which measure attributes, and summaries of measure attributes, are viewed.

Cross Tabulation

or Cross-Tab

size: **all**

color

item-name

	dark	pastel	white	Total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pant	20	2	5	27
Total	62	54	48	164

- Values for one of the dimension attributes form the **row** headers
- Values for another dimension attribute form the **column** headers
- Other dimension attributes are listed on top (here, just *size*)
- Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.

Cross-tab as relation

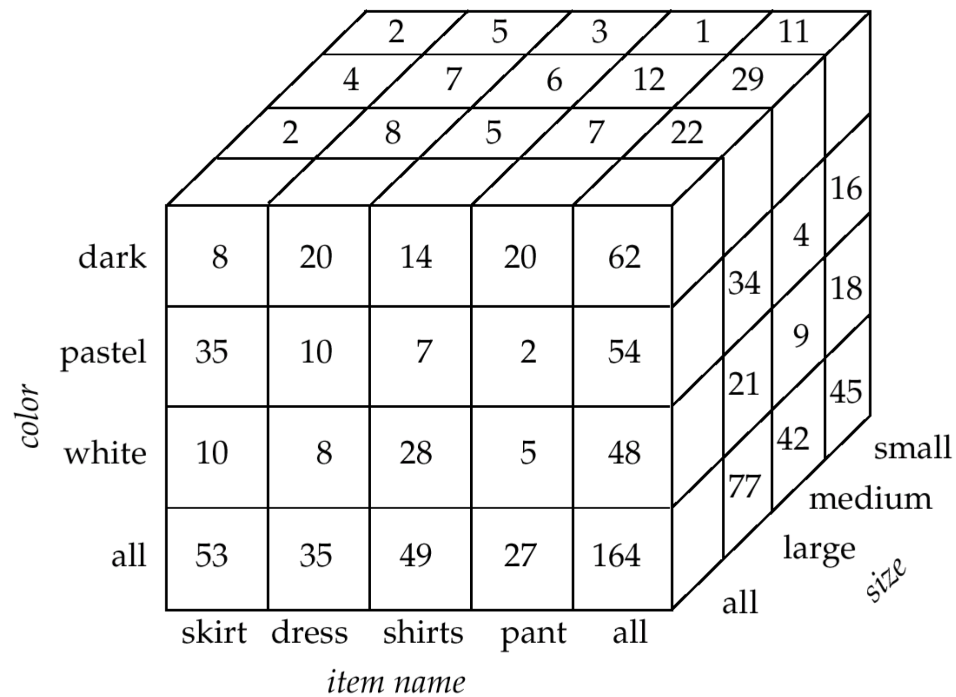
size: **all**

	color			
item-name	dark	pastel	white	Total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pant	20	2	5	27
Total	62	54	48	164

item-name	*color*	*number*
skirt	dark	8
skirt	pastel	35
skirt	white	10
skirt	**all**	53
dress	dark	20
dress	pastel	10
dress	white	5
dress	**all**	35
shirt	dark	14
shirt	pastel	7
shirt	white	28
shirt	**all**	49
pant	dark	20
pant	pastel	2
pant	white	5
pant	**all**	27
all	dark	62
all	pastel	54
all	white	48
all	**all**	164

- SQL queries can only output relations.
- Cross-tabs can be represented as relations

Data cube



- A data cube is a multidimensional generalization of a cross-tab
- In general, a data “cube” can have n dimensions; 3 shown above
- Cross-tabs can be used as views on a data cube

Window functions

- A **window function** performs a calculation across a set of table rows that are somehow related to the current row.
- This is related to the type of calculation that can be done with an aggregate function, but…
- Unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row. The rows retain their separate identities.
- Behind the scenes, the window function is able to access more than just the current row of the query result.

From Postgres: window functions

Window function syntax

- A window function call always contains an OVER clause directly following the window function's name and argument(s).
  - This is what syntactically distinguishes it from a regular function or aggregate function.
  - The OVER clause determines exactly how the rows of the query are split up for processing by the window function.
  - The PARTITION BY list within OVER specifies dividing the rows into groups, or partitions, that share the same values of the PARTITION BY expression(s).
  - For each row, the window function is computed across the rows that fall into the same partition as the current row.

Ranking

```
SELECT depname,
       empno,
       salary,
       rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;
```

depname	empno	salary	rank
develop	8	6000	1
develop	10	5200	2
develop	11	5200	2
develop	9	4500	4
develop	7	4200	5
personnel	2	3900	1
personnel	5	3500	2
sales	1	5000	1
sales	4	4800	2
sales	3	4800	2

Rank within current row’s partition

Order By is important

XML, XPath, and JSON

Relations converted to XML

STUDENT

sid	name	gender
1	Jill	F
2	Bo	M
3	Maya	F

Takes

sid	cid
1	445
1	483
3	435

COURSE

cid	title	sem
445	DB	F08
483	AI	S08
435	Arch	F08

One possible representation:

```
<students>
     <student>
          <name>Jill</name>
          <gender>F</gender>
          <courses>
               <course cid=445>
                    <title>DB</title>
                    <sem>F08</sem>
               </course>
               <course cid=483>
                    <title>AI</title>
                    <sem>S08</sem>
               </course>
     </student>
     <student>
     ...
     </student
</students>
```

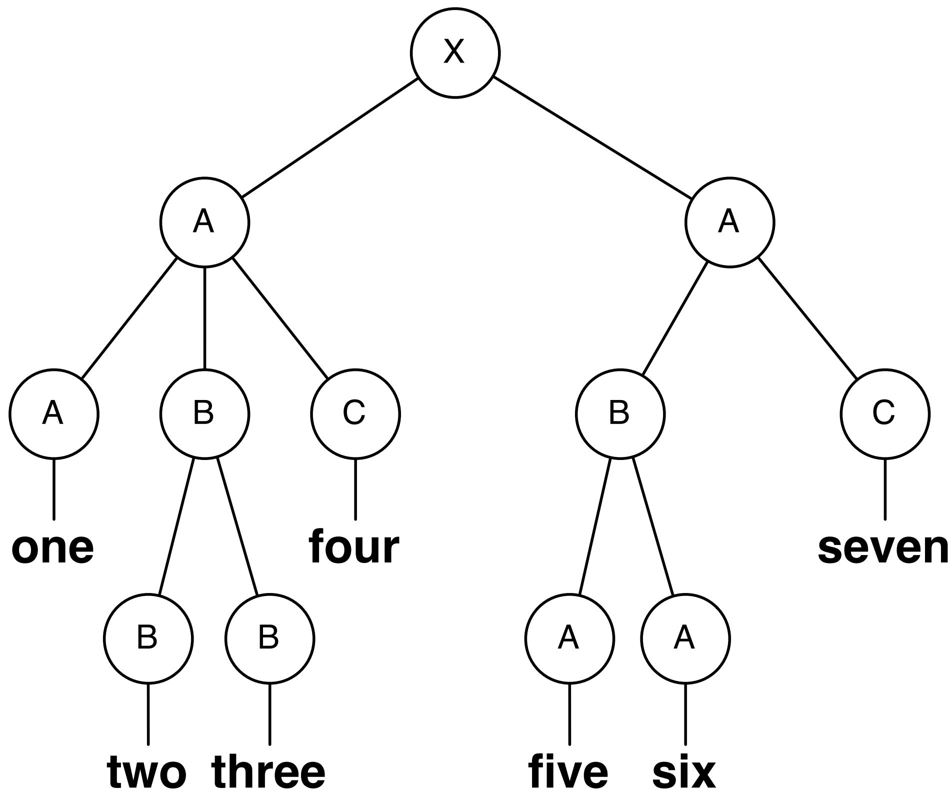
Xpath: Summary

bib	matches a bib element
*	matches any element
/	matches the root element
/bib	matches a bib element under root
bib/paper	matches a paper in bib
bib//paper	matches a paper in bib, at any depth
//paper	matches a paper at any depth
paper | book	matches a paper or a book
@price	matches a price attribute
bib/book/@price	matches price attribute in book, in bib
bib/book[./@price<“55”]/ author/lastname	matches…

I-clicker exercise

What does the following XPath expression return on the XML document below

//A[A]/*/*/text()



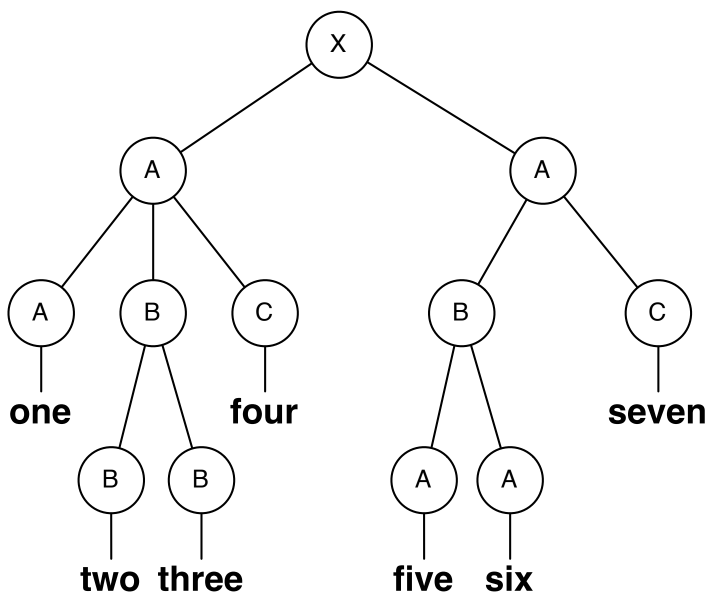
A. [empty set]
B. one two three four
C. two three
D. one four seven
E. two three five six

answer on next slide

I-clicker exercise

What does the following XPath expression return on the XML document below

//A[A]/*/*/text()



A. [empty set]

B. one two three four

C. two three

D. one four seven

E. two three five six

Comparing File Organizations

- Heap files (random order; insert at eof)
- Sorted files, sorted on <*age, sal*>
- Clustered B+ tree file, Alternative (1), search key <*age, sal*>
- Heap file with unclustered B + tree index on search key <*age, sal*>
- Heap file with unclustered hash index on search key <*age, sal*>

Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

- **B:** The number of data pages
- **R:** Number of records per page
- **D:** (Average) time to read or write disk page
- Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.
- Average-case analysis; based on several simplistic assumptions.

☛ *Good enough to show the overall trends!*

Operations to Compare

- Scan: Fetch all records from disk
- Equality search
- Range selection
- Insert a record
- Delete a record

Assumptions in Our Analysis

- Heap Files:
  - Equality selection on key; exactly one match.
- Sorted Files:
  - Files compacted after deletions.
- Indexes:
  - Alt (2), (3): data entry size = 10% size of record
  - Hash: No overflow chains.
    - 80% page occupancy => File size = 1.25 data size
  - B+Tree:
    - 67% occupancy (typical): implies file size = 1.5 data size
    - Balanced with fanout F (133 typical) at each non-level

Assumptions (contd.)

- Scans:
  - Leaf levels of a tree-index are chained.
  - Index data-entries plus actual file scanned for unclustered indexes.
- Range searches:
  - We use tree indexes to restrict the set of data records fetched, but ignore hash indexes.

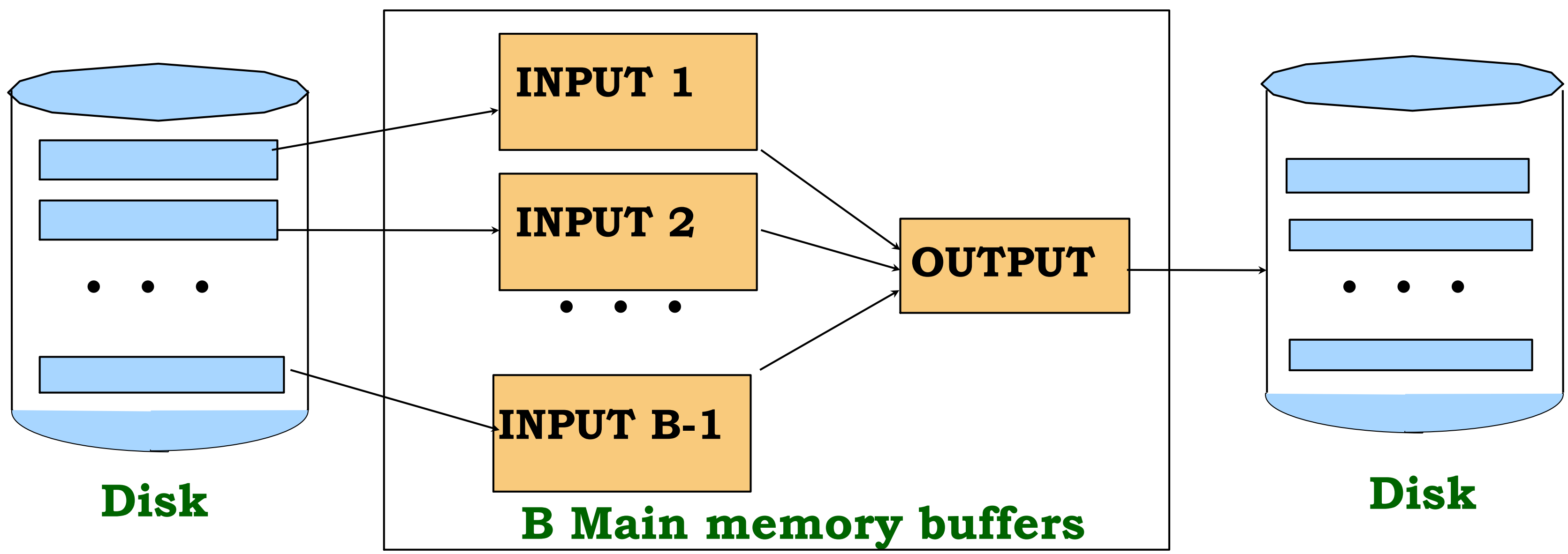
Cost of Operations

	Scan	Equality	Range	Insert	Delete
Heap File	BD	.5BD	BD	2D	Search + D
Sorted File	BD	$D\log_2 B$	$D(\log_2 B$ + #matching pages)	Search + BD	Search + BD
Clustered Tree Index	1.5BD	$D\log_F 1.5B$	$D(\log_F 1.5B$ + #matching pages)	Search + D	Search + D
Unclustered Tree Index	BD(R+.15)	$D(1+\log_F .15B)$	$D(\log_F .15B$ + #matching **recs**)	Search + 3D	Search + 3D
Unclustered Hash Index	BD(R+.125)	2D	BD	4D	4D

☛ *Several assumptions underlie these (rough) estimates!*

General External Merge Sort

- ***More than 3 buffer pages. How can we utilize them?***
  - 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 2, …, etc.: merge $B\text{-}1$ runs.

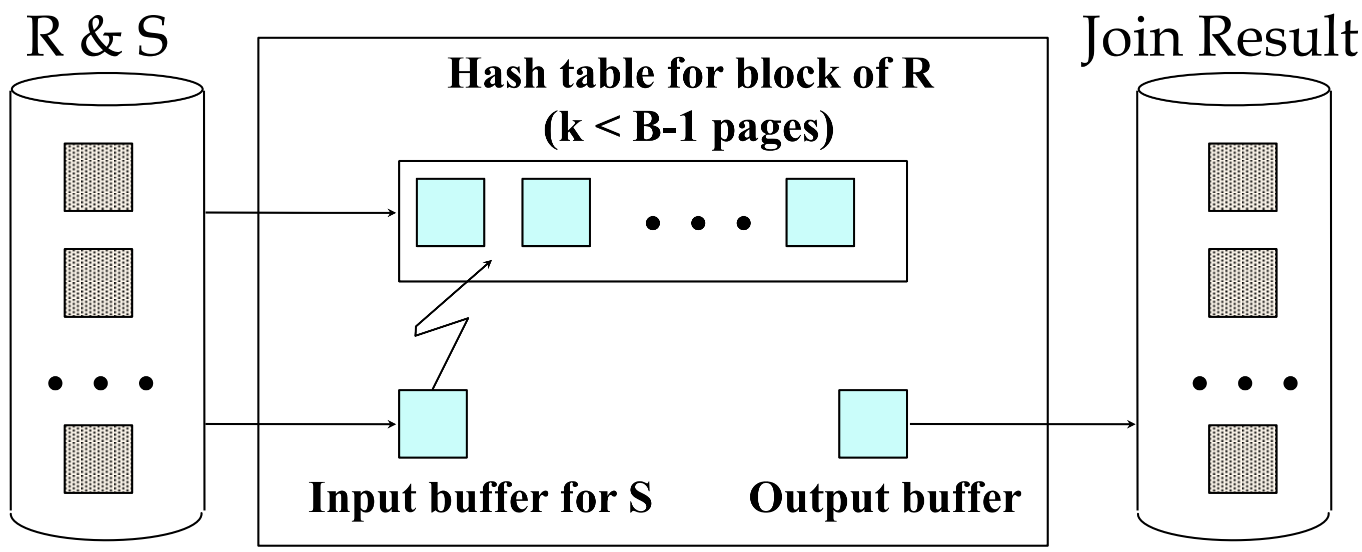


Cost of External Merge Sort

- Number of passes: $1+\lceil \log_{B-1} \lceil N/B \rceil \rceil$
- Cost = 2N * (# of passes)
- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil 108/5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22/4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Block Nested Loops Join

- Take the smaller relation, say R, as outer, the other as inner.
- Use one buffer for scanning the inner S, one buffer for output, and use all remaining buffers to hold ``block'' of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result.
  - Then read next page in S, until S is finished.



Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil$ # pages of outer / block size $\rceil$
  - Given available buffer size B, block size is at most B-2.
  - M + N * $\lceil$ M / B-2 $\rceil$
- With Sailors (S) as outer, let block be 100 pages of S:
  - Cost of scanning S is 500 I/Os; a total of 5 *blocks*.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
  - Total = 500 + 5 * 1000 = 5,500 I/Os.
  - (a little over 1 minute)

Index Nested Loops Join

foreach tuple r in R do
    foreach tuple s in S where $r_i == s_j$ do
        add <r, s> to result

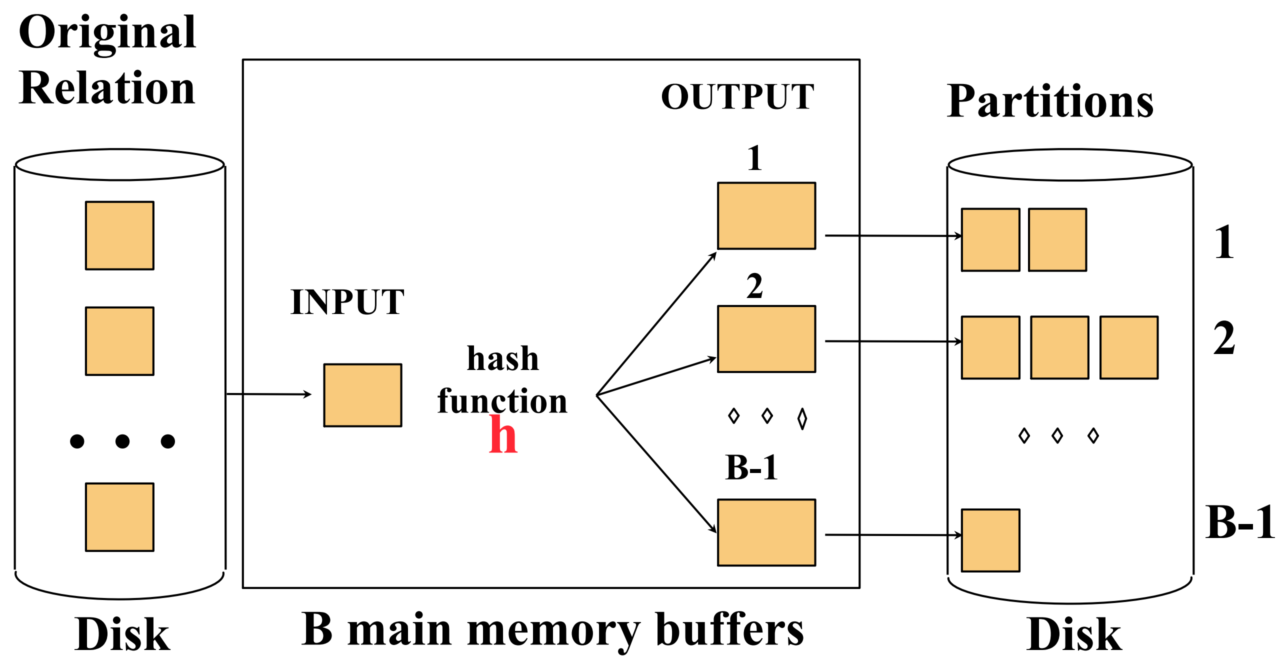
- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: M + ($(M*p_R)$ * cost of finding matching S tuples)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples depends on clustering.
  - Clustered index: 1 I/O (typical).
  - Unclustered: up to 1 I/O per matching S tuple.

Sort-Merge Join $(R \bowtie_{i=j} S)$

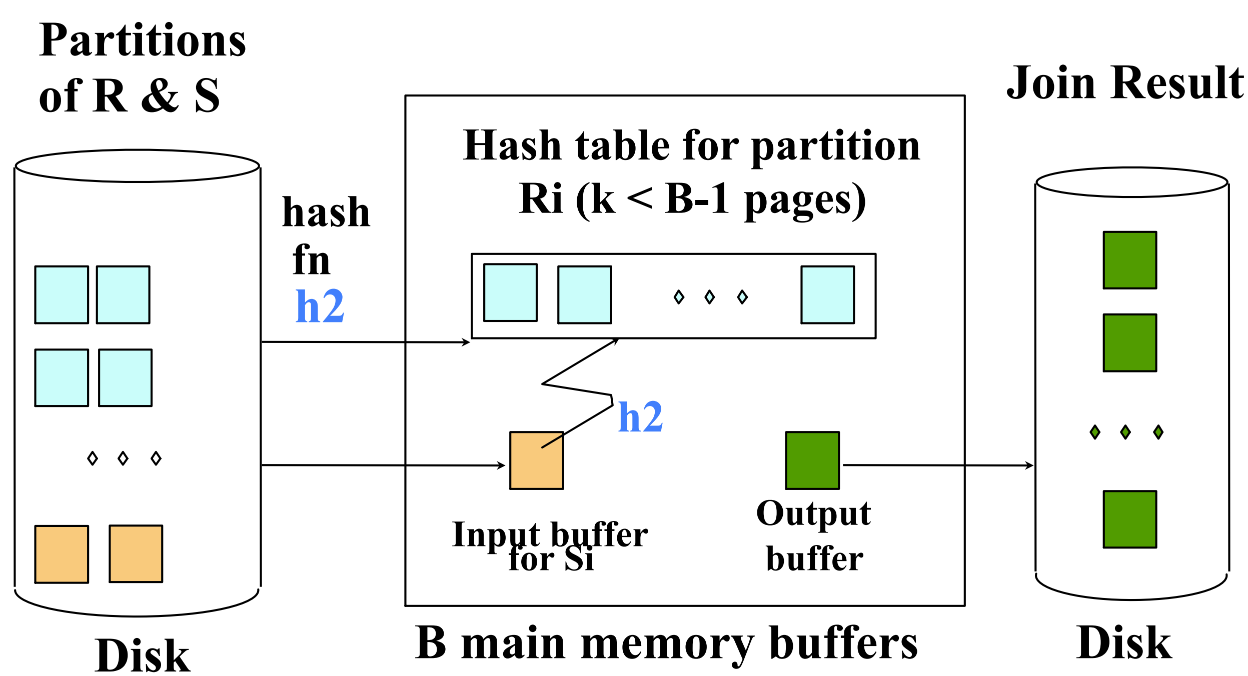
- (1) Sort R and S on the join column, (2) Merge them (on join col.), and output result tuples.
- Merge: repeat until either R or S is finished
  - *Scanning*: Advance scan of R until current R-tuple>=current S tuple, advance scan of S until current S-tuple>=current R tuple; do this until current R tuple = current S tuple.
  - *Matching*: Now all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) match; output <r, s> for all pairs of such tuples.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

Hash-Join

- Partitioning: Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.



- Probing: Read in partition i of R, build hash table on Ri using **h2 (<> h!)**. Scan partition i of S, search for matches.



Observations on Hash-Join

- # partitions ≤ B-1, and size of largest partition ≤ B-2 to be held in memory. Assuming uniformly sized partitions, we get:
  - M / (B-1) < (B-2), i.e., B must be > $\sqrt{M}$
  - Hash-join works if the smaller relation satisfies above.
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If hash function h does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.

Cost of Hash-Join

- Partitioning reads+writes both relns; 2(M+N). Probing reads both relns; M+N I/Os. The total is 3(M+N).
  - In our running example, a total of 4500 I/Os using hash join, less than 1 min (compared to 140 hours w. NLJ).
- Sort-Merge Join vs. Hash Join:
  - With optimizations to Sort-Merge (not discussed in class), the cost is similar ~ 3(M+N)
  - Hash Join superior if relation sizes differ greatly.
  - Hash Join has been shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.