Database Management Systems

Lecture 7

Evaluating Relational Operators

Query Optimization

Sabina S. CS

- running example - schema
  - Students (SID: integer, SName: string, Age: integer)
  - Courses (CID: integer, CName: string, Description: string)
  - Exams (SID: integer, CID: integer, EDate: date, Grade: integer, FacultyMember: string)

- Students
  - every record has 50 bytes
  - there are 80 records / page
  - 500 pages of Students tuples
- Courses
  - every record has 50 bytes
  - there are 80 records / page
  - 100 pages of Courses tuples
- Exams
  - every record has 40 bytes
  - there are 100 records / page
  - 1000 pages of Exams tuples

External Merge Sort

- Simple Two-Way Merge Sort: buffer pages are not used effectively
  - for instance, if 200 buffer pages are available, this algorithm still uses only 2 input buffers for passes 1, 2, …
- generalize the Two-Way Merge Sort algorithm to effectively use the available main memory and minimize the number of passes

- input file to be sorted: N pages
- B buffer pages are available
- pass 0:
  - use B buffer pages
  - read in B pages at a time and sort them in memory

    => $\left\lceil \frac{N}{B} \right\rceil$ runs of B pages each (except for the last one, which may be smaller)

External Merge Sort

- consider again the input file in the previous example:

3, 4	6, 2	9, 4	8, 7	5, 6	3, 1	2
page 1	page 2	…				

- N = 7 (number of pages in the file)
- B = 4 (there are 4 available buffer pages)
- <u>pass 0</u> produces $\left\lceil \frac{N}{B} \right\rceil = \left\lceil \frac{7}{4} \right\rceil = 2$ runs:
    - use all 4 buffer pages
    - read in 4 pages:

3, 4	6, 2	9, 4	8, 7

    - sort the pages in memory, write to disk a run that is 4 pages long:

2, 3	4, 4	6, 7	8, 9

    - read in remaining 3 pages:

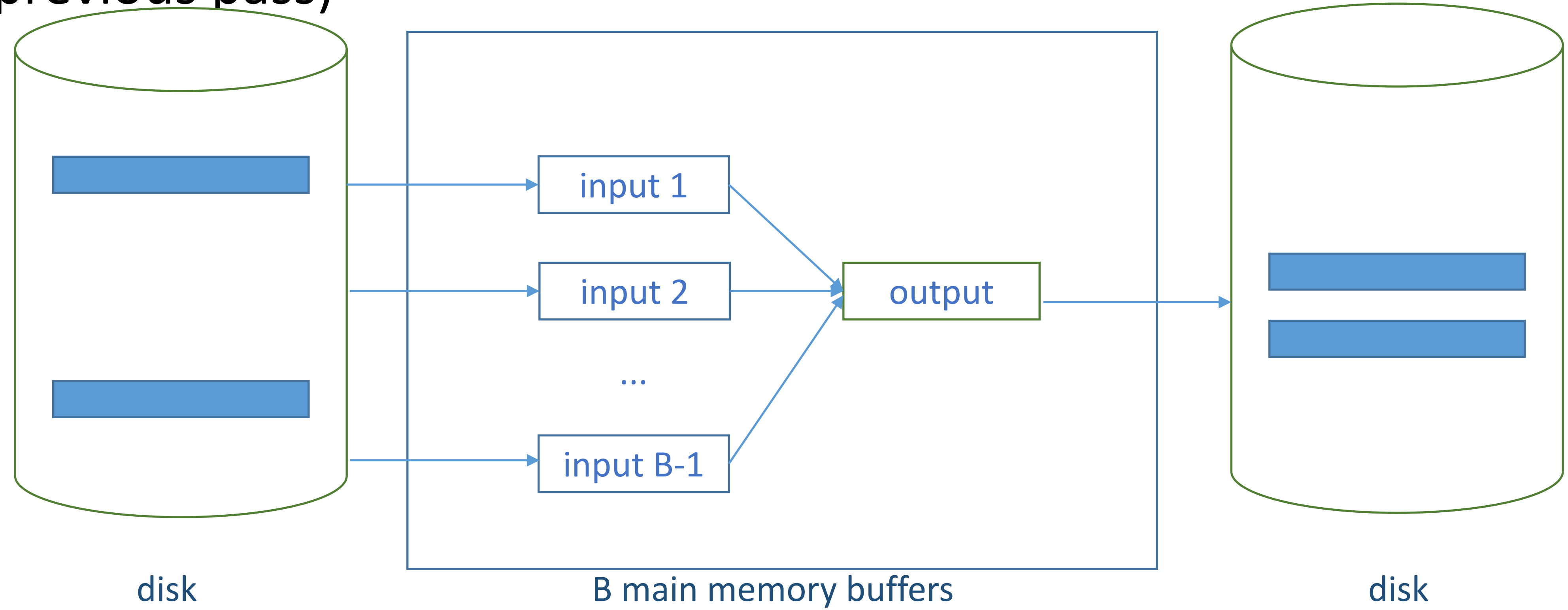
5, 6	3, 1	2

    - sort pages in memory, write to disk run of 3 pages:

1, 2	3, 5	6

External Merge Sort

- input file to be sorted: N pages
- B buffer pages are available
- pass 1, 2 ...:
  - use B-1 pages for input, and one page for output
  - perform a (B-1)-way merge in each pass (i.e., merge B-1 runs from the previous pass)



External Merge Sort

- runs at the end of pass 0:

2, 3	4, 4	6, 7	8, 9

1, 2	3, 5	6

- pass 1
  - read in & merge the first B-1 = 4-1 = 3 runs from pass 0
  - pass 0 produced only 2 runs in this example; read in and merge these 2 runs:

    => run

1, 2	2, 3	3, 4	4, 5	6, 6	7, 8	9

External Merge Sort

- another example:
  - 5 buffer pages B = 5
  - sort file with 108 pages N = 108

- pass 0
  - use all 5 buffer pages
  - read in the first 5 pages of the file, sort them in memory, write the resulting run to disk (5 pages long)
  - read in the next 5 pages of the file, sort them in memory, write the resulting run to disk (5 pages long)
  - …
  - read in the remaining 3 pages of the file, sort them in memory, write the resulting run to disk (3 pages long)
  - 21 runs are 5 pages long; 1 run is 3 pages long

External Merge Sort

- another example: B = 5, N = 108
- pass 0
  - at the end of pass 0 there are $\left\lceil \frac{N}{B} \right\rceil = \left\lceil \frac{108}{5} \right\rceil$ = 22 runs
- pass 1
  - use B-1 = 5-1 = 4 pages for input, and one page for output
  - do a 4-way merge: read in and merge 4 runs from the previous pass
  - read in the first 4 runs from pass 0 (each run into an input buffer)
    - merge the runs and write to the output buffer
    - write the output buffer to disk one page at a time

  => a run that is 20 pages long (4 runs from pass 0 times 5 pages per run)
  - read in the next 4 runs from pass 0; merge the runs and write to the output buffer; write the output buffer to disk one page at a time

  => another run (20 pages long)

  …

External Merge Sort

- another example: B = 5, N = 108
- pass 0
  - at the end of pass 0 there are 22 runs

- pass 1
  - read in the last 2 runs from pass 0 (one has 5 pages, the other one has 3 pages)
    - merge the runs and write to the output buffer; write the output buffer to disk one page at a time

  => the last run (8 pages long)
  - at the end of pass 1 there are $\left\lceil \frac{22}{4} \right\rceil$ = 6 runs
  - 5 runs are 20 pages long; 1 run is 8 pages long

External Merge Sort

- another example: B = 5, N = 108
- pass 1
  - at the end of pass 1 there are 6 runs

- pass 2
  - 4-way merge
  - read in the first 4 runs from pass 1
    - merge the runs and write to the output buffer; write the output buffer to disk one page at a time

  => a run that is 80 pages long (4 runs from pass 1 times 20 pages per run)
  - read in the remaining 2 runs from pass 1 (20 and 8 pages, respectively)

  => a run that is 28 pages long
  - at the end of pass 2 there are $\left\lceil \frac{6}{4} \right\rceil$ = 2 runs

External Merge Sort

- another example: B = 5, N = 108
- pass 2
    - at the end of pass 2 there are 2 runs

- pass 3
    - read in the 2 runs from pass 2 and merge them
      => a run that is 108 pages long, representing the sorted file

External Merge Sort

- cost
  - N – number of pages in the input file, B – number of available pages in the buffer
  - in each pass: read / process / write each page
  - number of passes: $\lceil log_{B-1}\lceil N/B\rceil\rceil + 1$
  - total cost: $2 * N * \left(\left\lceil log_{B-1}\left\lceil\frac{N}{B}\right\rceil\right\rceil + 1\right)$ I/Os

- previous example: B = 5 and N = 108, with 4 passes over the data
  - cost:
    2 * 108 * 4 = 864 I/Os
  - $2 * 108 * \left(\left\lceil log_{5-1}\left\lceil\frac{108}{5}\right\rceil\right\rceil + 1\right) = 216 * (\lceil log_4 22\rceil + 1) =$
    $216 * 4 = 864$ I/Os

- B buffer pages
- sort file with N pages

Simple Two-Way Merge Sort

pass 0 => N runs

number of passes = $\lceil log_2 N \rceil + 1$

External Merge Sort

pass 0 => $\left\lceil \frac{N}{B} \right\rceil$ runs

number of passes = $\left\lceil log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil + 1$

- External Merge Sort – reduced number of:
  - runs produces by the 1st pass
  - passes over the data
- B is usually large => significant performance gains

<u>External Merge Sort</u> – number of passes for different values of N and B

N	B = 3	B = 5	B = 9	B = 17	B = 129	B = 257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Sort-Merge Join

- equality join, one join column: $\mathrm{E} \otimes_{i=j} S$ (i^{th} column's value in E = j^{th} column's value in S)
- sort E and S on the join column (if not already sorted):
  - for instance, by using External Merge Sort

  => *partitions* = groups of tuples with the same value in the join column
- merge E and S; look for tuples *e* in E, *s* in S such that $e_i = s_j$:
  - while *current* $e_i <$ *current* s_j
    - advance the scan of E
  - while *current* $e_i >$ *current* s_j
    - advance the scan of S
  - if *current* $e_i =$ *current* s_j
    - output joined tuples <*e, s*>, where *e* and *s* are in the current partition (i.e., they have the same value in the i^{th} and j^{th} column, respectively)
    - there could be multiple tuples in E with the same value in the i^{th} column as the current tuple *e* (same is true for S)

Sort-Merge Join

- partitions are illustrated on tables Students and Exams below (join column SID in both tables):

SID	SName	Age
20	Ana	20
30	Dana	20
40	Dan	20
45	Daniel	20
50	Ina	20

SID	CID	EDate	Grade	FacultyMember
30	2	20/1/2018	10	Ionescu
30	1	21/1/2018	9.99	Pop
45	2	20/1/2018	9.98	Ionescu
45	1	21/1/2018	9.98	Pop
45	3	22/1/2018	10	Stan
50	2	20/1/2018	10	Ionescu

Sort-Merge Join

- during the merging phase, E is scanned once; every partition in S is scanned as many times as there are matching tuples in the corresponding partition in E

E

…	i^{th} column	…
	1	
	3	
	3	
	3	
	8	
	…	

S

…	j^{th} column	…
	2	
	3	
	3	
	4	
	…	
	…	

partition P

  - for instance, partition P in the above table S is scanned 3 times, once per matching tuple in the corresponding partition in E
  - there are 6 output joined tuples <*e, s*> for partition P
- this algorithm avoids the enumeration of the cross-product: tuples in a partition in E are compared only with the S tuples in the same partition!

Sort-Merge Join

- cost:
  - sorting E
    - cost: O(MlogM)
  - sorting S
    - cost: O(NlogN)
  - cost of merging: M + N I/Os, assuming partitions in S are scanned only once
    - worst-case scenario: O(M * N) I/Os (when all records in E and S have the same value in the join column)

* E - M pages; S - N pages*

Sort-Merge Join (Exams$\otimes_{Exams.SID=Students.SID}$ Students)

- 100 buffer pages
  - sort Exams
    - 2 passes => cost: 2 * 2 * 1000 = 4000 I/Os
  - sort Students
    - 2 passes => cost: 2 * 2 * 500 = 2000 I/Os
  - merging phase
    - cost: 1000 + 500 = 1500 I/Os
  - total cost: 4000 + 2000 + 1500 = 7500 I/Os
    - similar to the cost of Block Nested Loops Join
- 35 buffer pages, 300 buffer pages – cost remains unchanged (need 2 passes to sort Exams, 2 passes to sort Students)
    - ex: compute cost of BNLJ and compare

* E - M pages, p_E records / page * * 1000 pages * * 100 records / page*

* S - N pages, p_S records / page * * 500 pages * * 80 records / page *

Hash Join - equality join, one join column: $\mathrm{E} \otimes_{i=j} \mathrm{S}$

- phases: partitioning (building phase) & probing (matching phase)

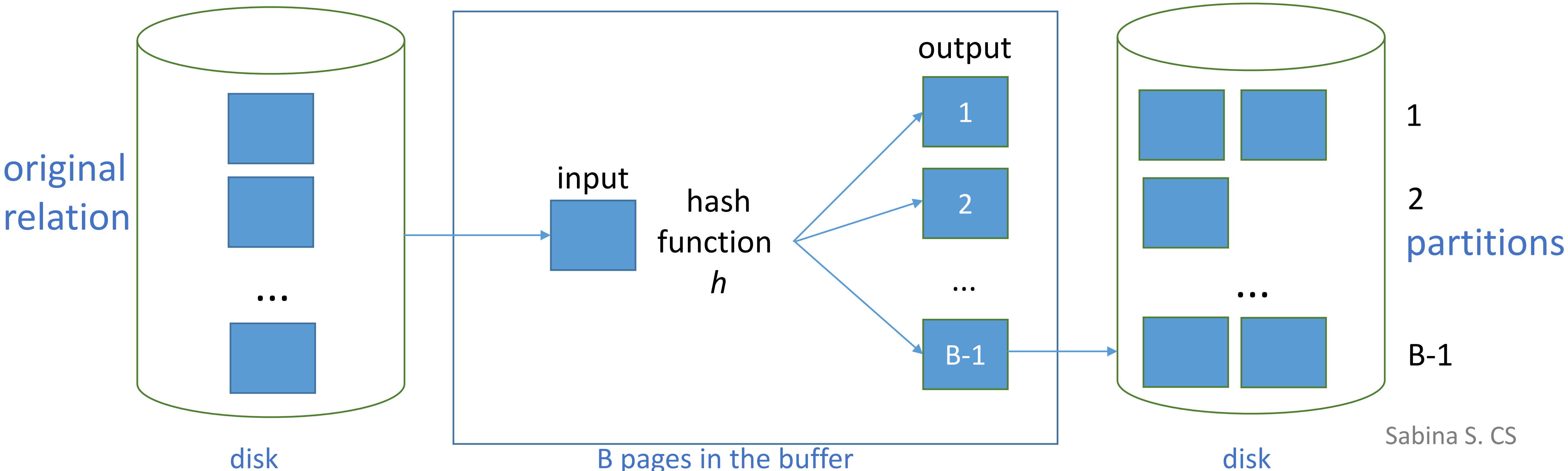
- partitioning phase:
  - there are B pages available in the buffer:
    - use one page as the input buffer page
    - and the remaining B-1 pages as output buffer pages

  - choose a hash function *h* that distributes tuples uniformly to one of B-1 partitions

  - hash E and S on the join column (the i^{th} column of E, the j^{th} column of S) with the same hash function *h*

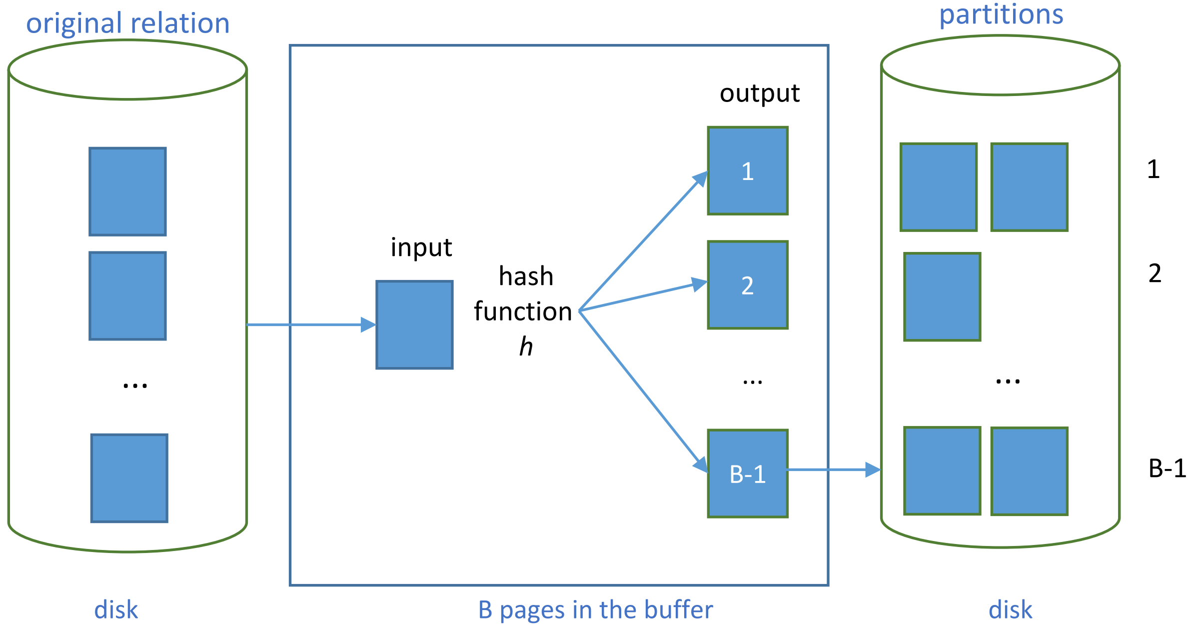
Hash Join

- hash E on the join column with hash function *h* (similarly for S):
  - for each tuple *e* in E, compute $h(e_i)$
    (e_i: the value of the i^{th} column in tuple *e*)
  - add tuple *e* to the output buffer page that it is hashed to by *h* (buffer page $h(e_i)$)
  - when an output buffer page fills up, flush the page to disk



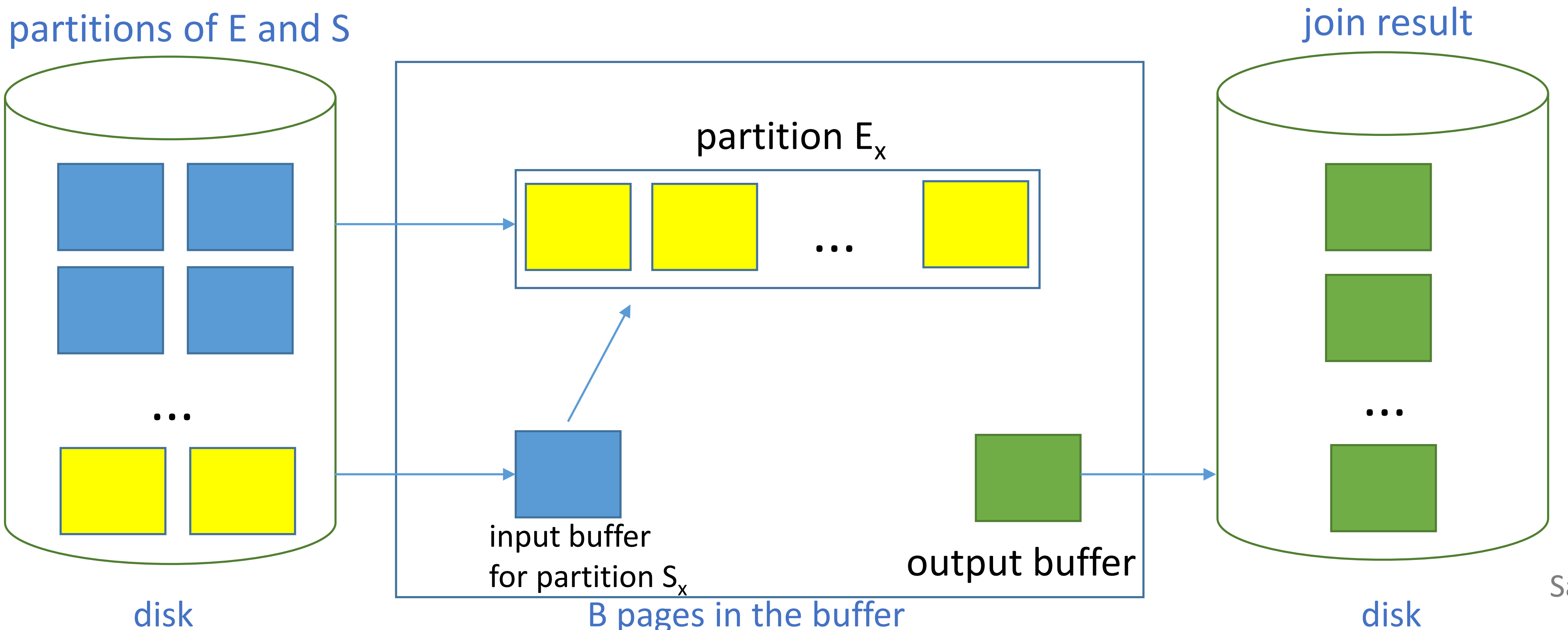
Hash Join

- partitioning phase => *partitions* of E (E_1, E_2, etc.) and S (S_1, S_2, etc.) on disk
- partition = collection of tuples that have the same hash value
- tuples in partition E_1 can only join with tuples in partition S_1 (they cannot join with tuples in partitions S_2 or S_3, for instance, since these tuples have a different hash value)
- so to compute the join, we need to scan E and S only once (provided any partition of E fits in main memory)
- when reading in a partition E_k of E, we must scan only the corresponding partition S_k of S to find matching tuples (compare tuples *e* in E_k with tuples *s* in S_k to test the join condition *value of* i^{th} *column in E = value of* j^{th} *column in S*)



Hash Join

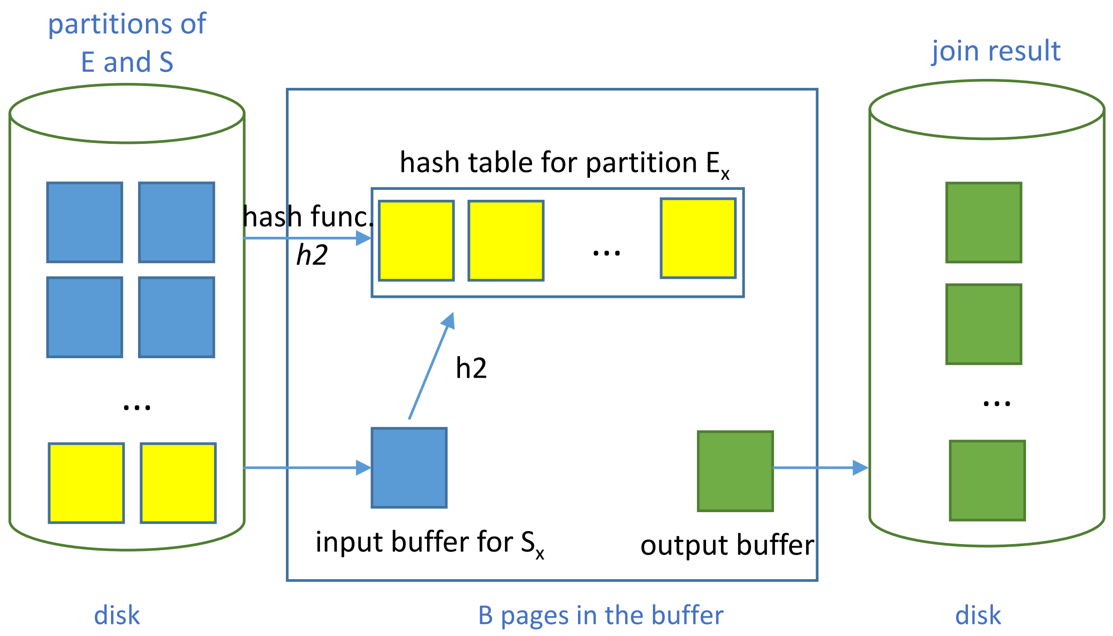
- probing phase:
  - read in a partition of the smaller relation (e.g., E) and scan the corresponding partition of S for matching tuples
  - use one page as the input buffer for S, one page as the output buffer, and the remaining pages to read in partitions of E



Sabina S. CS

Hash Join

- probing phase:
  - in practice, to reduce CPU costs, an in-memory hash table is built, using a different function *h2,* for the E partition



- consider a partition E_x of E
- build in-memory hash table for E_x using hash function *h2* (the function is applied to the join column of E)
- for each tuple *s* in partition S_x, find matching tuples in the hash table using the hash value $h2(s_j)$
- result tuples <e, s> are written to output buffer
- once partitions E_x and S_x are processed, the hash table is emptied (to prepare for the next partition)

<u>Hash Join</u>

- <u>cost</u>:
  - partitioning: both E and S are read and written once => cost: 2*(M+N) I/Os
  - probing: scan each partition once => cost: M+N I/Os

  => total cost: 3*(M+N) I/Os
    - assumption: each partition fits into memory during probing
  - 3*(1000 + 500) = 4500 I/Os

* E - M pages, p_E records / page * * 1000 pages * * 100 records / page*

* S - N pages, p_S records / page * * 500 pages * * 80 records / page *

- *partition overflow* – an E partition does not fit in memory during probing: apply hash join technique recursively:
  - divide E, S into subpartitions
  - join subpartitions pairwise
  - if subpartitions don't fit in memory, apply hash join technique recursively

Hash Join

- memory requirements - objective: partition in E fits into main memory (S - similarly)
  - B buffer pages; need one input buffer => maximum number of partitions: B-1
  - size of largest partition: B – 2 (need one input buffer for S, one output buffer)
  - assume uniformly sized partitions => size of each E partition: M/(B-1)

  => M/(B-1) < B-2 => we need approximately B > $\sqrt{M}$

  - if an in-memory hash table is used to speed up tuple matching => need a little more memory (because the hash table for a collection of tuples will be a little larger than the collection itself)

* E - M pages, p_E records / page *   * 1000 pages * * 100 records / page*

general join conditions

- equalities over several attributes
  - E.SID = S.SID AND E.attrE = S.attrS
    - index nested loops join
      - Exams – inner relation:
        - build index on Exams with search key <SID, attrE> (if not already created)
        - can also use index on SID or index on attrE
      - Students – inner relation (similar)
    - sort-merge join
      - sort Exams on <SID, attrE>, sort Students on <SID, attrS>
    - hash join
      - partition Exams on <SID, attrE>, partition Students on <SID, attrS>
    - other join algorithms
      - essentially unaffected

general join conditions

- inequality comparison
    - E.attrE < S.attrS
        - index nested loops join: B+ tree index required
        - sort-merge join: not applicable
        - hash join: not applicable
        - other join algorithms: essentially unaffected

* no join algorithm is uniformly superior to others

- choice of a good algorithm depends on:
    - size(s) of:
        - joined relations
        - buffer pool
    - available access methods

Selection

Q:

```
SELECT *
FROM Exams E
WHERE E.FacultyMember = 'Ionescu'
```

- use information in the selection condition to reduce the number of retrieved tuples
- e.g., |Q| = 4 (result set has 4 tuples), there's a B+ tree index on FacultyMember
  - it's expensive to scan E (1000 I/Os) to evaluate the query
  - should use the index instead

- selection algorithms based on the following techniques:
  - iteration, indexing

* E - M pages, p_E records / page *   * 1000 pages * * 100 records / page*

Selection

- simple selections
  - $\sigma_{E.attr\ op\ val}(E)$

- no index on *attr*, data not sorted on *attr*
  - must scan E and test the condition for each tuple
  - access path: file scan

  => cost: M I/Os = 1000 I/Os

- no index, sorted data (E physically sorted on *attr)*
  - binary search to locate 1st tuple that satisfies condition

    and
  - scan E starting at this position until condition is no longer satisfied
  - access method: sorted file scan

Review lecture notes on *Relational Algebra, Indexes, DB – Physical Structure* (*Databases* course)

Selection

- simple selections
  - $\sigma_{E.attr\ op\ val}(E)$

- no index, sorted data (E physically sorted on *attr)*
  => cost:
  - binary search: $O(\log_2 M)$
  - scan cost: varies from 0 to M
  - binary search on E
    - $\log_2 1000 \approx 10$ I/Os

Selection

- simple selections
  - $\sigma_{E.attr\ op\ val}(E)$

- B+ tree index on *attr*
  * search tree to find 1st index entry pointing to a qualifying E tuple
    - cost: typically 2, 3 I/Os
  * scan leaf pages to retrieve all qualifying entries
    - cost: depends on the number of qualifying entries
  * for each qualifying entry - retrieve corresponding tuple in E
    - cost: depends on the number of tuples and the nature of the index (clustered / unclustered)

Selection

- simple selections
  - $\sigma_{E.attr\ op\ val}(E)$

- B+ tree index on *attr*
  - assumption
    - indexes use a2 or a3
    - a1-based index => data entry contains the data record => the cost of retrieving records = the cost of retrieving the data entries!
  - access path: B+ tree index
    - clustered index:
      - best access path when *op* is not *equality*
      - good access path when *op* is *equality*

Selection

- simple selections: $\sigma_{E.attr\ op\ val}(E)$
- B+ tree index on *attr*

Q

```
SELECT *
FROM Exams E
WHERE E.FacultyMember < 'C%'
```

- names uniformly distributed with respect to 1st letter

=> |Q| ≈ 10,000 tuples = 100 pages

- clustered B+ tree index on FacultyMember

=> cost of retrieving tuples: ≈ 100 I/Os (a few I/Os to get from root to leaf)

- non-clustered B+ tree index on FacultyMember

=> cost of retrieving tuples: up to 1 I/O per tuple (worst case) => up to 10.000 I/Os

* E - M pages, p_E records / page *   * 1000 pages * * 100 records / page*

Selection

- simple selections: $\sigma_{E.attr\ op\ val}(E)$
- B+ tree index on *attr*

```
SELECT *
FROM Exams E
WHERE E.FacultyMemger < 'C%'
```

- non-clustered B+ tree index on FacultyMember
    - refinement - sort rids in qualifying data entries by page-id
      => a page containing qualifying tuples is retrieved only once
        - cost of retrieving tuples: number of pages containing qualifying tuples (but such tuples are probably stored on more than 100 pages)
- range selections
    - non-clustered indexes can be expensive
    - could be less costly to scan the relation (in our example: 1000 I/Os)

Selection

- general selections
  - selections without disjunctions
- C - CNF condition without disjunctions
  - evaluation options:

  1. use the most selective access path
     - if it's an index I:
       - apply conjuncts in C that match I
       - apply rest of conjuncts to retrieved tuples
     - example
       - *c < 100 AND a = 3 AND b = 5*
         - can use a B+ tree index on *c* and check *a = 3 AND b = 5* for each retrieved tuple
         - can use a hash index on *a* and *b* and check *c < 100* for each retrieved tuple

Selection

- general selections - selections without disjunctions
  - evaluation options:

  2. use several indexes - when several conjuncts match indexes using a2 / a3
     - compute sets of rids of candidate tuples using indexes
     - intersect sets of rids, retrieve corresponding tuples
     - apply remaining conjuncts (if any)
     - example: $c < 100$ *AND* $a = 3$ *AND* $b = 5$
       - use a B+ tree index on c to obtain rids of records that meet condition $c < 100$ (R_1)
       - use a hash index on a to retrieve rids of records that meet condition $a = 3$ (R_2)
       - compute $R_1 \cap R_2 = R_{int}$
       - retrieve records with rids in R_{int} (R)
       - check $b = 5$ for each record in R

Selection

- general selections
  - selections with disjunctions
- C - CNF condition with disjunctions, i.e., some conjunct *J* is a disjunction of terms
  - if some term *T* in *J* requires a file scan, testing *J* by itself requires a file scan
    - example: $a < 100 \vee b = 5$
      - hash index on *b*, hash index on *c*

    => check both terms using a file scan (i.e., best access path: file scan)
  - compare with the example below:
      - $(a < 100 \vee b = 5) \wedge c = 7$
      - hash index on *b*, hash index on *c*

    => use index on *c*, apply $a < 100 \vee b = 5$ to each retrieved tuple (i.e., most selective access path: index)

Selection

- general selections
  - selections with disjunctions

- C - CNF condition with disjunctions
  - every term *T* in a disjunction matches an index

  => retrieve tuples using indexes, compute union
  - example
    - $a < 100 \lor b = 5$
    - B+ tree indexes on *a* and *b*
    - use index on *a* to retrieve records that meet condition $a < 100$ (R_1)
    - use index on *b* to retrieve records that meet condition $b = 5$ (R_2)
    - compute $R_1 \cup R_2 = R$
    - if all matching indexes use a2 or a3 => take union of rids, retrieve corresponding tuples

References

- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8th Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3rd Edition, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si19] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (7th Edition), McGraw-Hill, 2019
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, http://codex.cs.yale.edu/avi/db-book/
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems, http://infolab.stanford.edu/~ullman/fcdb.html