Database Management Systems

Lecture 9

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

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

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

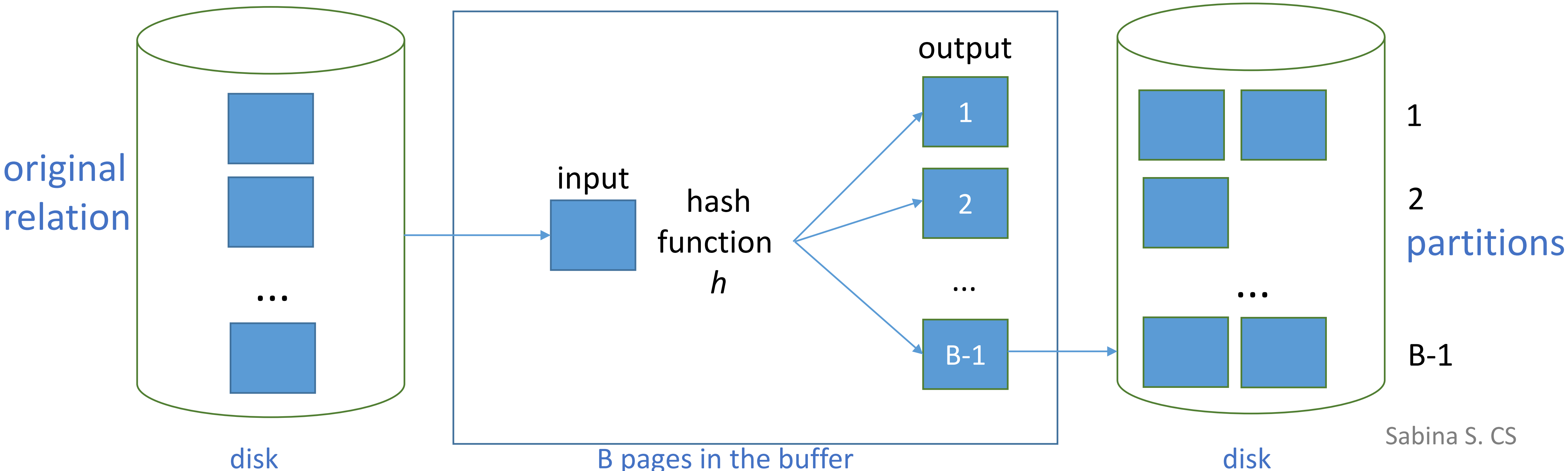
- partitioning phase:
  - there are B pages available in the buffer:
    - use one page as the input buffer page
    - and the remaining B-1 pages as output buffer pages

  - 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*

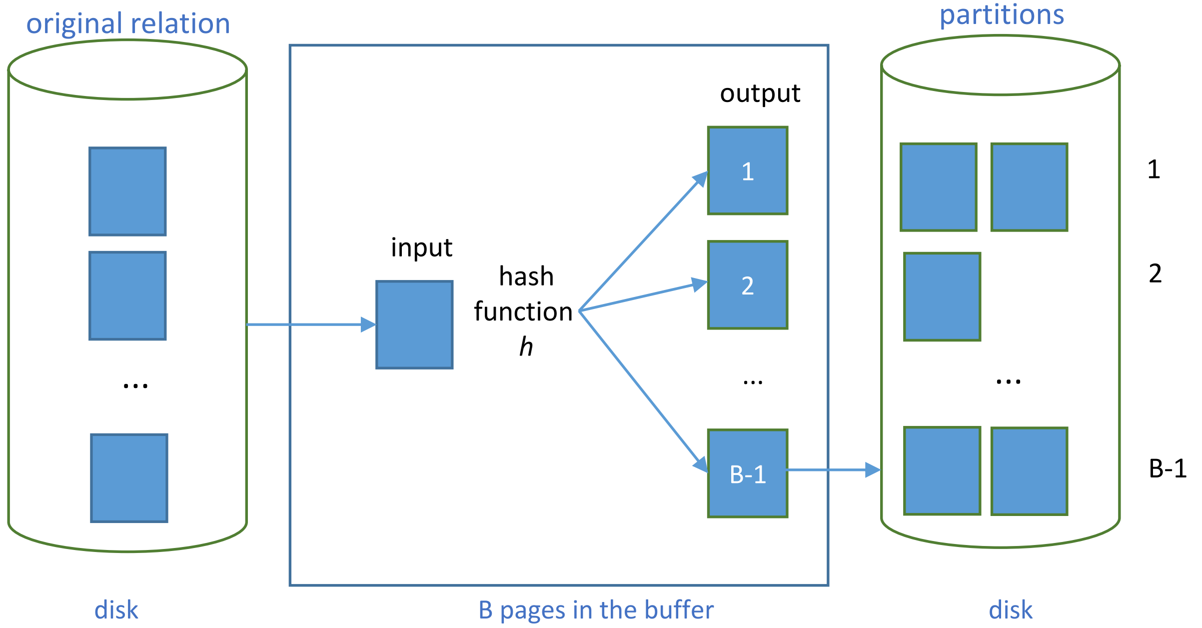
Hash Join

- 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



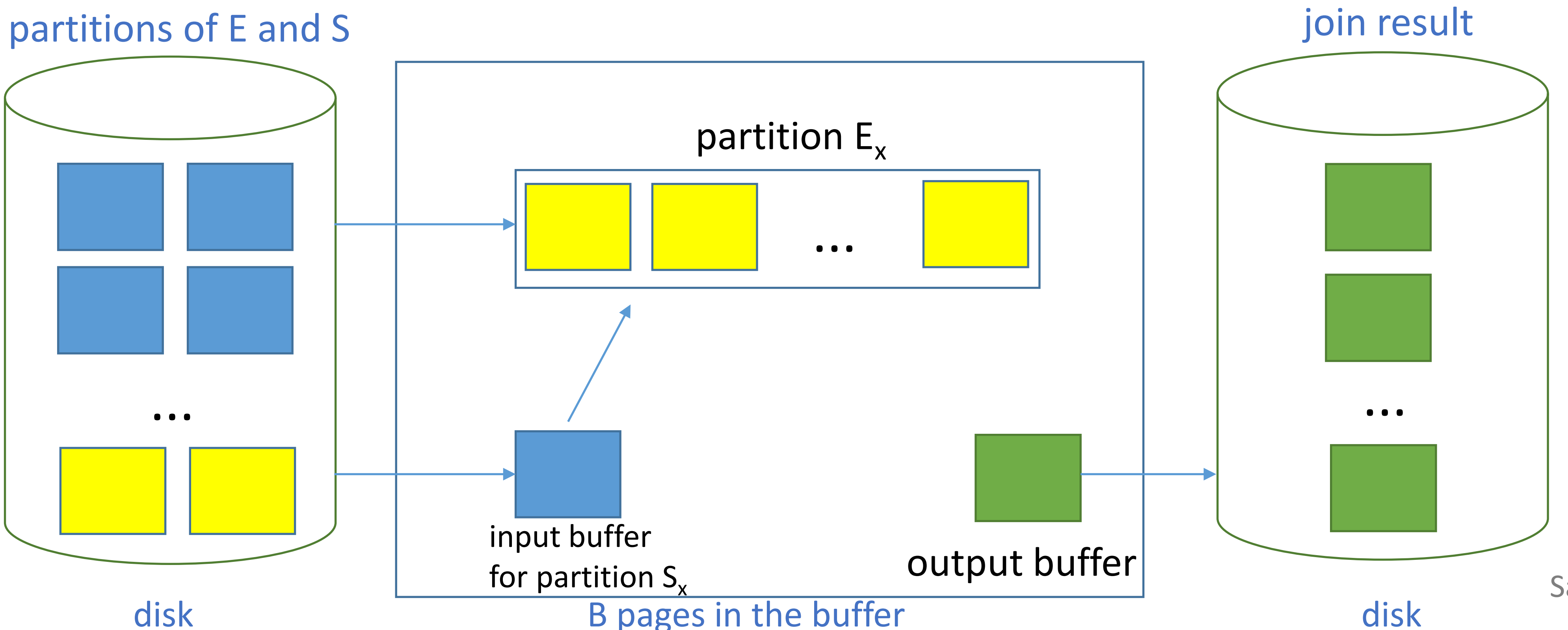
Hash Join

- 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*)



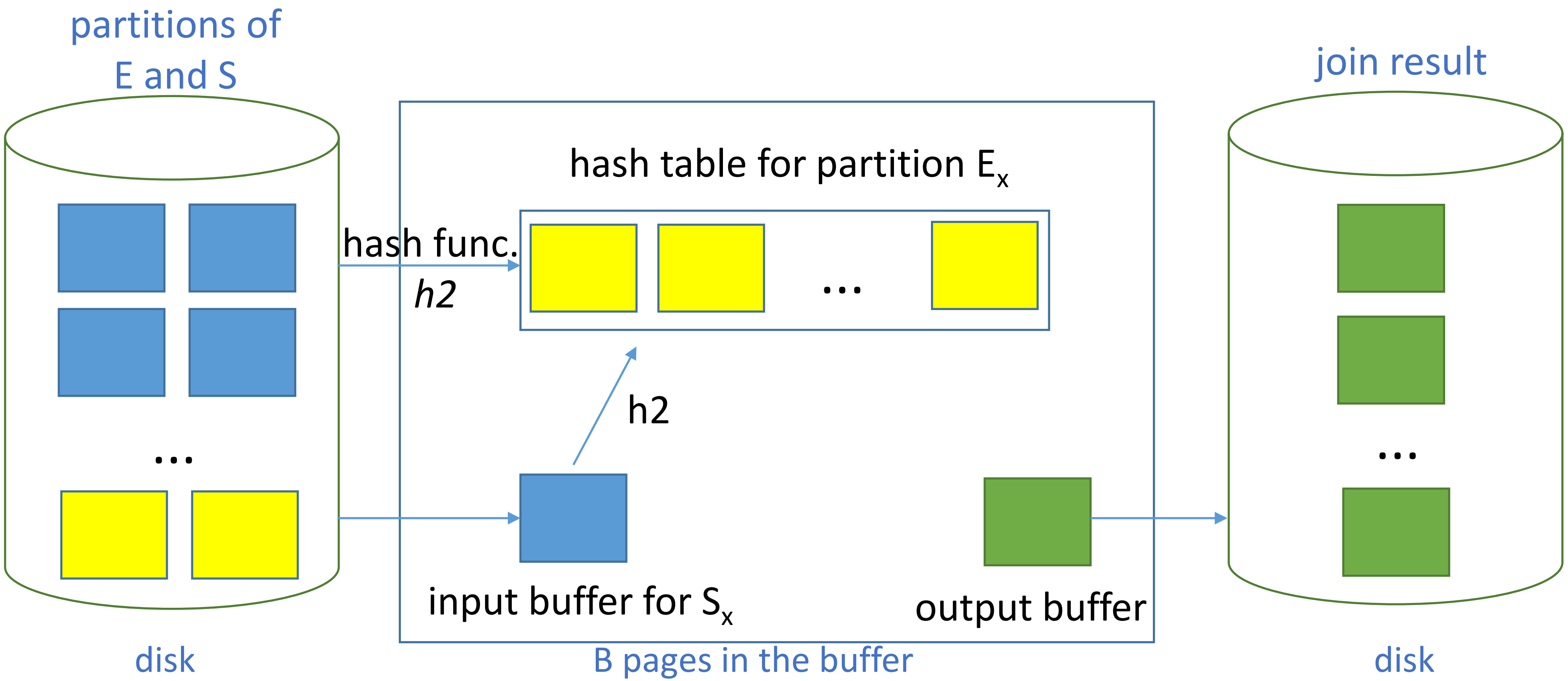
Hash Join

- 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



Hash Join

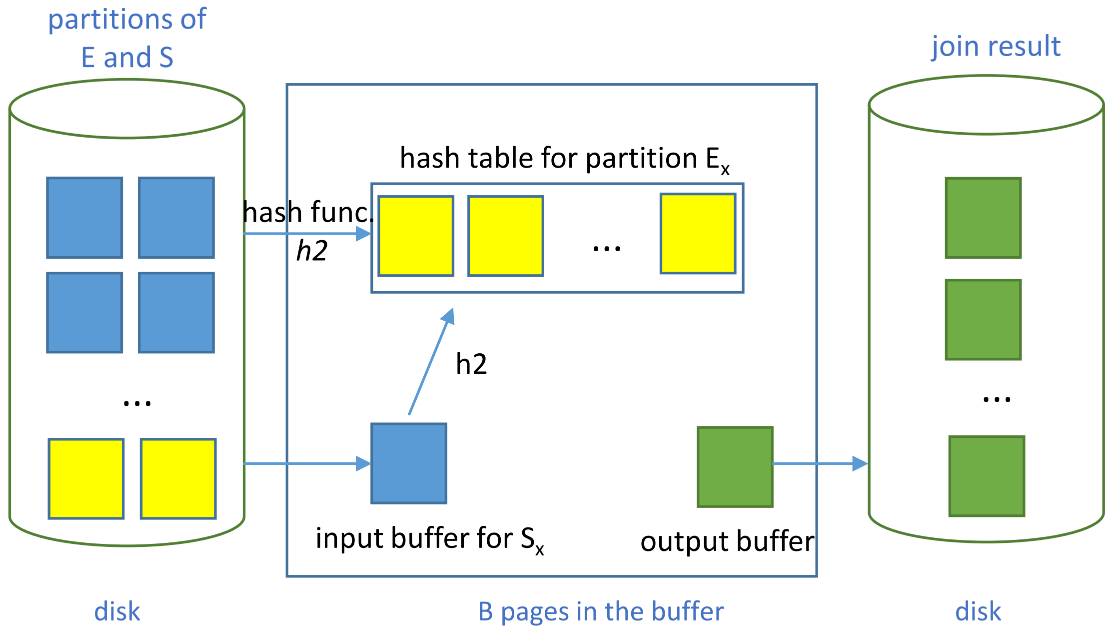
- probing phase:
  - in practice, to reduce CPU costs, an in-memory hash table is built, using a different function *h2*, for the E partition



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)

Hash Join

- cost:
  - 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 *

Hash Join

- *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 applied
    - hash join
      - not applied
    - 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
    - $a < 100 \vee b = 5$
    - 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

Projection

- $\Pi_{\text{SID, CID}}$(Exams)

```
SELECT DISTINCT E.SID, E.CID
FROM Exams E
```

- to implement projection:
  - eliminate:
    - unwanted columns
    - duplicates
- projection algorithms - *partitioning* technique:
  - sorting
  - hashing

Projection Based on Sorting

- step 1
  - scan E => set of tuples containing only desired attributes (E')
  - cost:
    - scan E: M I/Os
    - write temporary relation E': T I/Os
      - T depends on: number of columns and their sizes, T is O(M)
- step 2
  - sort tuples in E'
  - sort key: all columns
  - cost: O(TlogT)   (also O(MlogM))
- step 3
  - scan sorted E', compare adjacent tuples, eliminate duplicates
  - cost: T
- total cost: O(MlogM)

<u>Projection Based on Sorting</u>

* example

```
SELECT DISTINCT E.SID, E.CID
FROM Exams E
```

- scan Exams: 1000 I/Os
- size of tuple in E': 10 bytes

=> cost of writing temporary relation E': 250 I/Os

- available buffer pages: 20
  - E' can be sorted in 2 passes
  - sorting cost: 2 * 2 * 250 = 1000 I/Os

- final scan of E' - cost: 250 I/Os

=> total cost: 1000 + 250 + 1000 + 250 = 2500 I/Os

* E – record size = 40 bytes * * 1000 pages * * 100 records / page*

Projection Based on Sorting

* example

```
SELECT DISTINCT E.SID, E.CID
FROM Exams E
```

- scan Exams: 1000 I/Os
- size of tuple in E': 10 bytes

=> cost of writing temporary relation E': 250 I/Os

- available buffer pages: 257
  - E' can be sorted in 1 pass
  - sorting cost: 2 * 1 * 250 = 500 I/Os

- final scan of E' - cost: 250 I/Os

=> total cost: 1000 + 250 + 500 + 250 = 2000 I/Os

* E – record size = 40 bytes *   * 1000 pages *   * 100 records / page*

Projection Based on Sorting

- improvement
  - adapt the sorting algorithm to do projection with duplicate elimination
    - modify pass 0 of External Merge Sort: eliminate unwanted columns
      - read in B pages from E
      - write out (T/M) * B internally sorted pages of E'
        - refinement: write out 2*B internally sorted pages of E' (on average)
      - tuples in runs - smaller than input tuples
    - modify merging passes: eliminate duplicates
      - number of result tuples is smaller than number of input tuples

Projection Based on Sorting

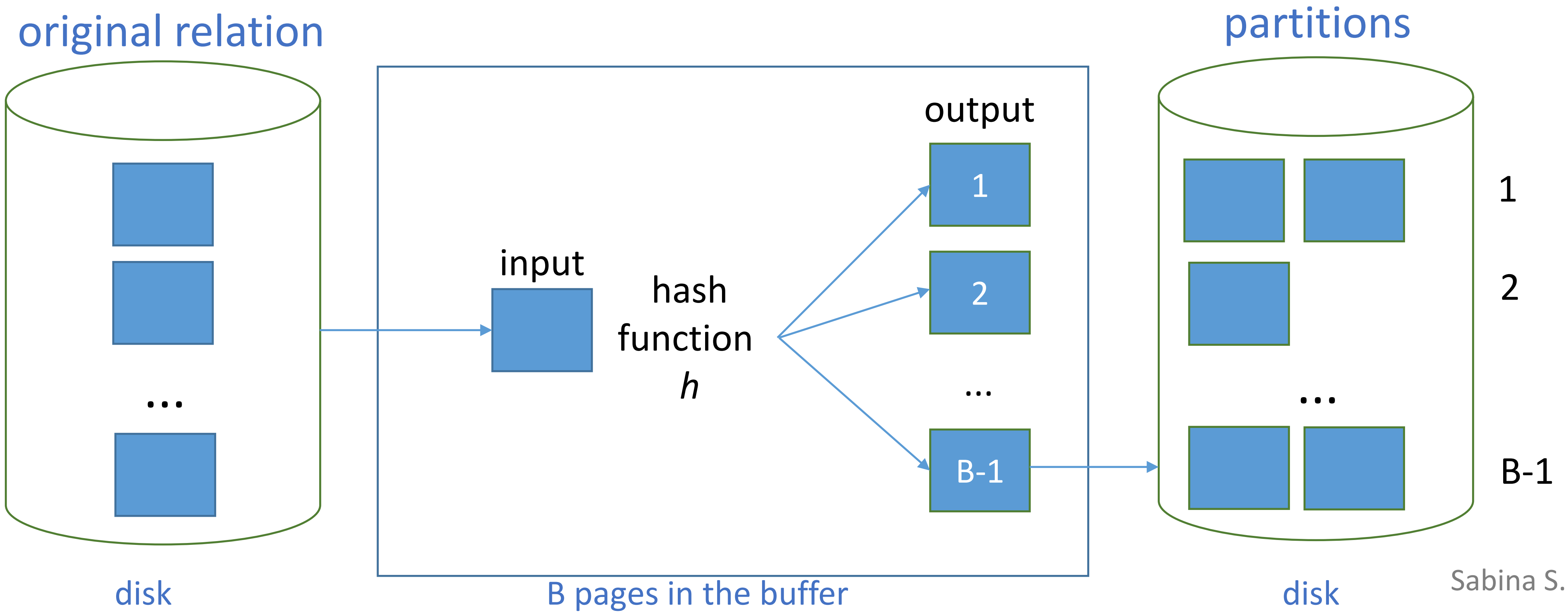
- improvement

  * example

  - pass 0:
    - scan Exams: 1000 I/Os
    - write out 250 pages:
      - 20 available buffer pages
        - 250 pages => 7 sorted runs about 40 pages long (except the last one, which is about 10 pages long)
  - pass 1:
    - read in all runs – cost: 250 I/Os
    - merge runs
  - total cost : 1000 + 250 + 250 = 1500 I/Os

Projection Based on Hashing

- phases: partitioning & duplicate elimination
- partitioning phase:
  - 1 input buffer page – read in the relation one page at a time
  - hash function *h* – distribute tuples uniformly to one of B-1 partitions
  - B-1 output buffer pages – one output page / partition



Projection Based on Hashing

- partitioning phase:
  - read the relation using the input buffer page
  - for each tuple *t*:
    - discard unwanted fields => tuple *t'*
    - apply hash function *h* to *t'*
    - write *t'* to the output buffer page that it is hashed to by *h*

  => B-1 partitions
  - partition:
    - collection of tuples with:
      - common hash value
      - no unwanted fields
  - tuples in different partitions are guaranteed to be distinct

<u>Projection Based on Hashing</u>

- <u>duplicate elimination phase</u>:
    - process all partitions:
        - read in partition P, one page at a time
            - build in-memory hash table with hash function *h2* ($\neq$ *h*) on all fields:
                - if a new tuple hashes to the same value as an existing tuple, compare them to check if they are distinct
                - eliminate duplicates
            - write duplicate-free hash table to result file
            - clear in-memory hash table

    - partition overflow
        - apply hash-based projection technique recursively (subpartitions)

Projection Based on Hashing

- cost
  - partitioning:
    - read E: M I/Os
    - write E': T I/Os
  - duplicate elimination:
    - read in partitions: T I/Os

  => total cost: M + 2*T I/Os

- Exams:
  - 1000 + 2*250 = 1500 I/Os

Set Operations

- intersection, cross-product
    - special cases of join (join condition for intersection - equality on all fields, no join condition for cross-product)
- union, set-difference
    - similar

- union: R ∪ S
    - sorting
        - sort R and S on all attributes
        - scan the sorted relations in parallel; merge them, eliminating duplicates
        - refinement
            - produce sorted runs of R and S, merge runs in parallel

Set Operations

- union: R ∪ S
  - hashing
    - partition R and S with the same hash function *h*
    - for each S-partition
      - build in-memory hash table (using *h2*) for the S-partition
      - scan corresponding R-partition, add tuples to hash table, discard duplicates
      - write out hash table
      - clear hash table

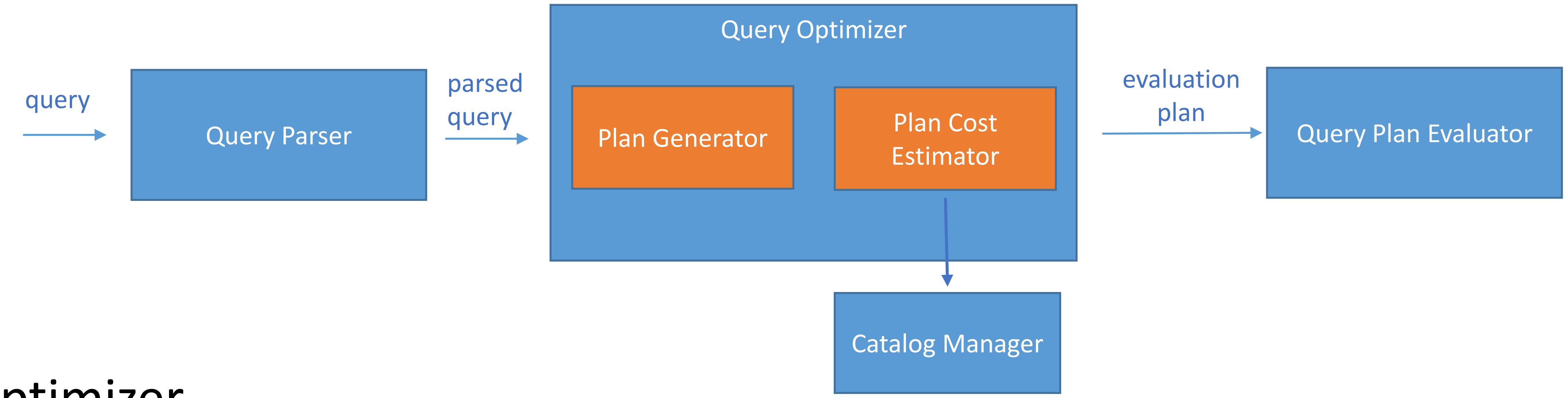
Aggregate Operations

- without grouping
  - scan relation
  - maintain *running information* about scanned tuples
    - COUNT - count of values retrieved
    - SUM - *total* of values retrieved
    - AVG - <*total, count*> of values retrieved
    - MIN, MAX - smallest / largest value retrieved
- with grouping
  - sort relation on the grouping attributes
  - scan relation to compute aggregate operations for each group
  - improvement: combine sorting with aggregation computation
  - alternative approach based on hashing

Aggregate Operations

- using existing indexes
  - index with a search key that includes all the attributes required by the query
    - work with the data entries in the index (instead of the data records)
  - attribute list in the GROUP BY clause is a prefix of the index search key (tree index)
    - get data entries (and records, if necessary) in the required order (i.e., avoid sorting)

Query Optimization



- optimizer
  - objective
    - given a query Q, find a good evaluation plan for a Q
  - generates alternative plans for Q, estimates their costs, and chooses the one with the least estimated cost
  - uses information from the system catalogs

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

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

Query Evaluation Plans

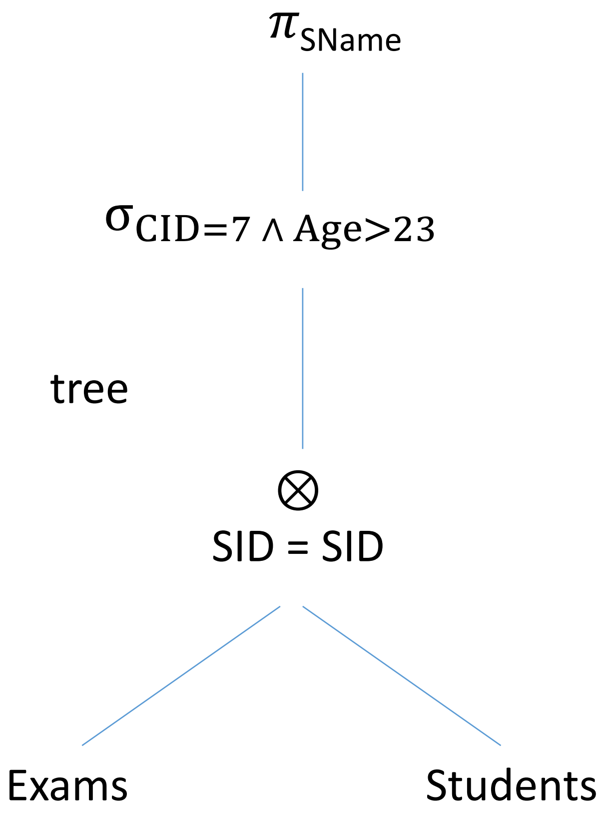
```
SELECT S.SName
FROM Exams E, Students S
WHERE E.SID = S.SID AND E.CID = 7
     AND S.Age > 23
```

query

$$\pi_{SName}(\sigma_{CID=7 \wedge Age>23}(Exams \otimes_{SID=SID} Students))$$

relational algebra expression

tree

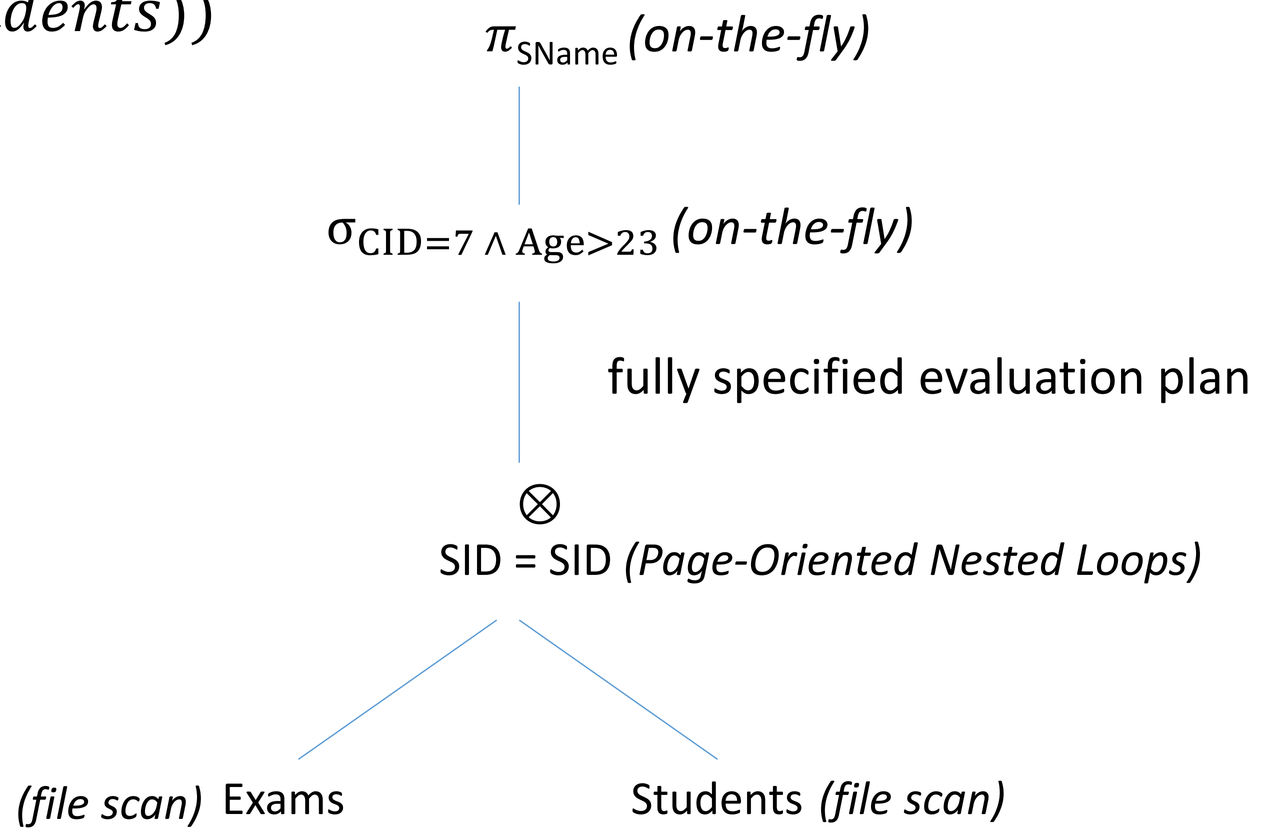


Query Evaluation Plans

```
SELECT S.SName
FROM Exams E, Students S
WHERE E.SID = S.SID AND E.CID = 7
     AND S.Age > 23
```

$\pi_{SName}(\sigma_{CID=7 \wedge Age>23}(Exams \otimes_{SID=SID} Students))$

- query evaluation plan
  - extended relational algebra tree
  - node – annotations
    - relation
      - access method
    - relational operator
      - implementation method

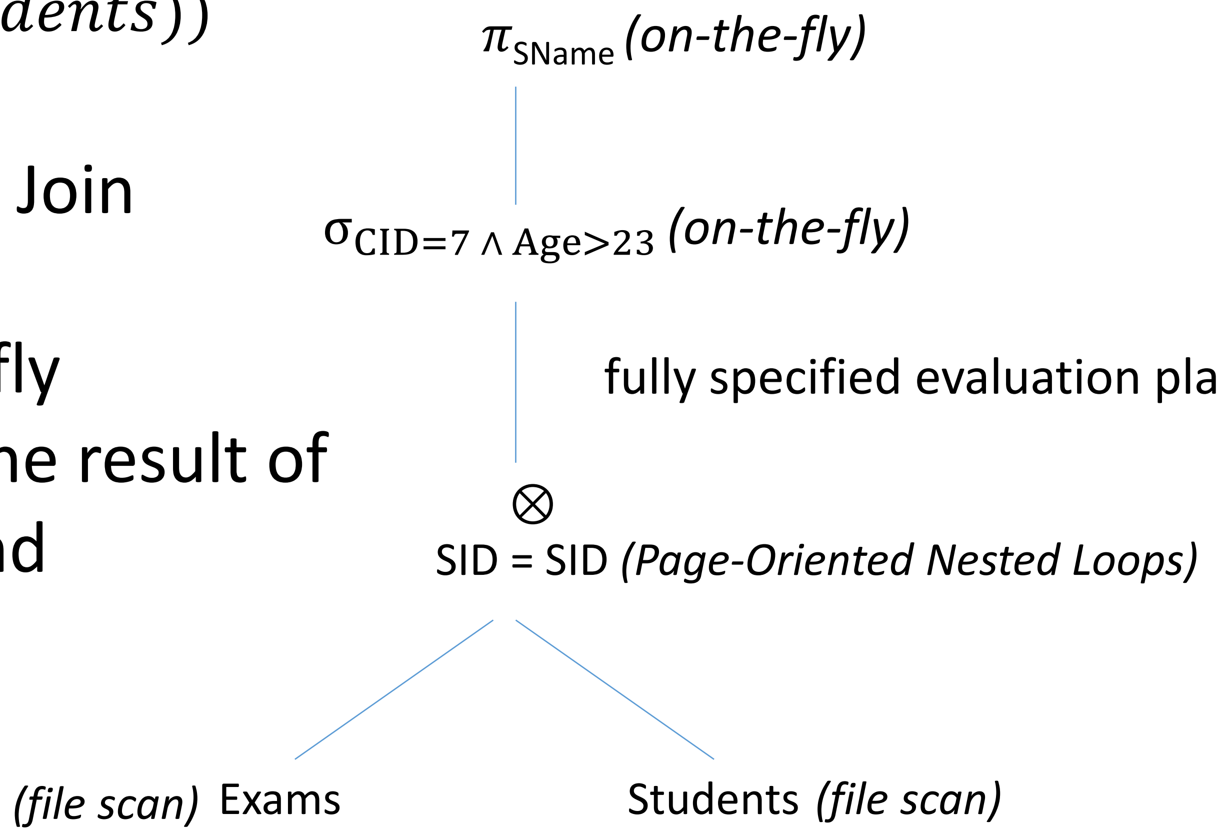


Query Evaluation Plans

```
SELECT S.SName
FROM Exams E, Students S
WHERE E.SID = S.SID AND E.CID = 7
     AND S.Age > 23
```

$\pi_{SName}(\sigma_{CID=7 \wedge Age>23}(Exams \otimes_{SID=SID} Students))$

- page-oriented Simpled Nested Loops Join
  - Exams – outer relation
- selection, projection applied on-the-fly
  to each tuple in the join result, i.e., the result of the join (before applying selection and projection) is not stored



fully specified evaluation plan

Pipelined Evaluation

```
SELECT *
FROM Exams
WHERE EDate > '1-1-2020' AND Grade > 8
```

$\underbrace{\texttt{EDate > '1-1-2020'}}_{T1}$ $\underbrace{\texttt{Grade > 8}}_{T2}$

$$\sigma_{Grade>8}(\sigma_{EDate>'1-1-2020'}(Exams))$$

- index *I* matches *T1*
- v1 - *materialization*
  - evaluate *T1*
  - write out result tuples to temporary relation *R*, i.e., tuples are *materialized*
  - apply the 2nd selection to *R*
  - cost: read and write *R*

Pipelined Evaluation

```
SELECT *
FROM Exams
WHERE EDate > '1-1-2020' AND Grade > 8
```

$\underbrace{\texttt{EDate > '1-1-2020'}}_{T1}$ AND $\underbrace{\texttt{Grade > 8}}_{T2}$

- v2 – *pipelined evaluation*
  - apply the 2nd selection to each tuple in the result of the 1st selection as it is produced
  - i.e., 2nd selection operator is applied *on-the-fly*
  - saves the cost of writing out / reading in the temporary relation *R*

Query Blocks – Units of Optimization

- parse Q => collection of query *blocks* -> passed on to the optimizer
- optimizer:
  - optimize one block at a time
- query *block* - SQL query:
  - without nesting
  - with exactly: one SELECT clause, one FROM clause
  - with at most: one WHERE clause, one GROUP BY clause, one HAVING clause
    - WHERE condition - CNF

Query Blocks – Units of Optimization

- query Q:

```
SELECT S.SID, MIN(E.EDate)
FROM Students S, Exams E, Courses C
WHERE S.SID = E.SID AND E.CID = C.CID AND C.Description = 'Elective' AND
                  S.Age = (SELECT MAX(S2.Age)
                          FROM Students S2)
GROUP BY S.SID
HAVING COUNT(*) > 2
```

nested block

- decompose query into a collection of blocks without nesting

```
SELECT S.SID, MIN(E.EDate)
FROM Students S, Exams E, Courses C
WHERE S.SID = E.SID AND E.CID = C.CID AND C.Description = 'Elective' AND
                  S.Age = Reference to nested block
GROUP BY S.SID
HAVING COUNT(*) > 2
```

Query Blocks – Units of Optimization

* block optimization

- express query block as a relational algebra expression

```
SELECT S.SID, MIN(E.EDate)
FROM Students S, Exams E, Courses C
WHERE S.SID = E.SID AND E.CID = C.CID AND C.Description = 'Elective' AND
                  S.Age = Reference to nested block
GROUP BY S.SID
HAVING COUNT(*) > 2
```

$\pi_{S.SID,\ MIN(E.EDate)}($
$HAVING_{COUNT(*) > 2}($
$GROUP\ BY_{S.SID}($
$\sigma_{S.SID = E.SID \land E.CID = C.CID \land C.Description = 'Elective' \land S.Age = value\_from\_nested\_block}($
$\quad Students \times Exams \times Courses\ ))))$

- GROUP BY, HAVING – operators in the extended algebra used for plans
- argument list of projection can include aggregate operations

Query Blocks – Units of Optimization

- query Q treated as a $\sigma\, \pi \times$ algebra expression
- the remaining operations in Q are performed on the result of the $\sigma\, \pi \times$ expression

```
SELECT S.SID, MIN(E.EDate)
FROM Students S, Exams E, Courses C
WHERE S.SID = E.SID AND E.CID = C.CID AND C.Description = 'Elective' AND
                  S.Age = Reference to nested block
GROUP BY S.SID
HAVING COUNT(*) > 2
```

$\pi_{S.SID,\ E.EDate}($

$\sigma_{S.SID\ =\ E.SID\ \wedge\ E.CID\ =\ C.CID\ \wedge\ C.Description\ =\ 'Elective'\ \wedge\ S.Age\ =\ value\_from\_nested\_block}($

$\quad Students \times Exams \times Courses\ ))$

- attributes in GROUP BY, HAVING are added to the argument list of projection
- aggregate expressions in the argument list of projection are replaced by their argument attributes

Query Blocks – Units of Optimization

* block optimization

- find best plan P for the $\sigma\ \pi \times$ expression
- evaluate P => result set RS
- sort/hash RS => groups
- apply HAVING to eliminate some groups
- compute aggregate expressions in SELECT for each remaining group

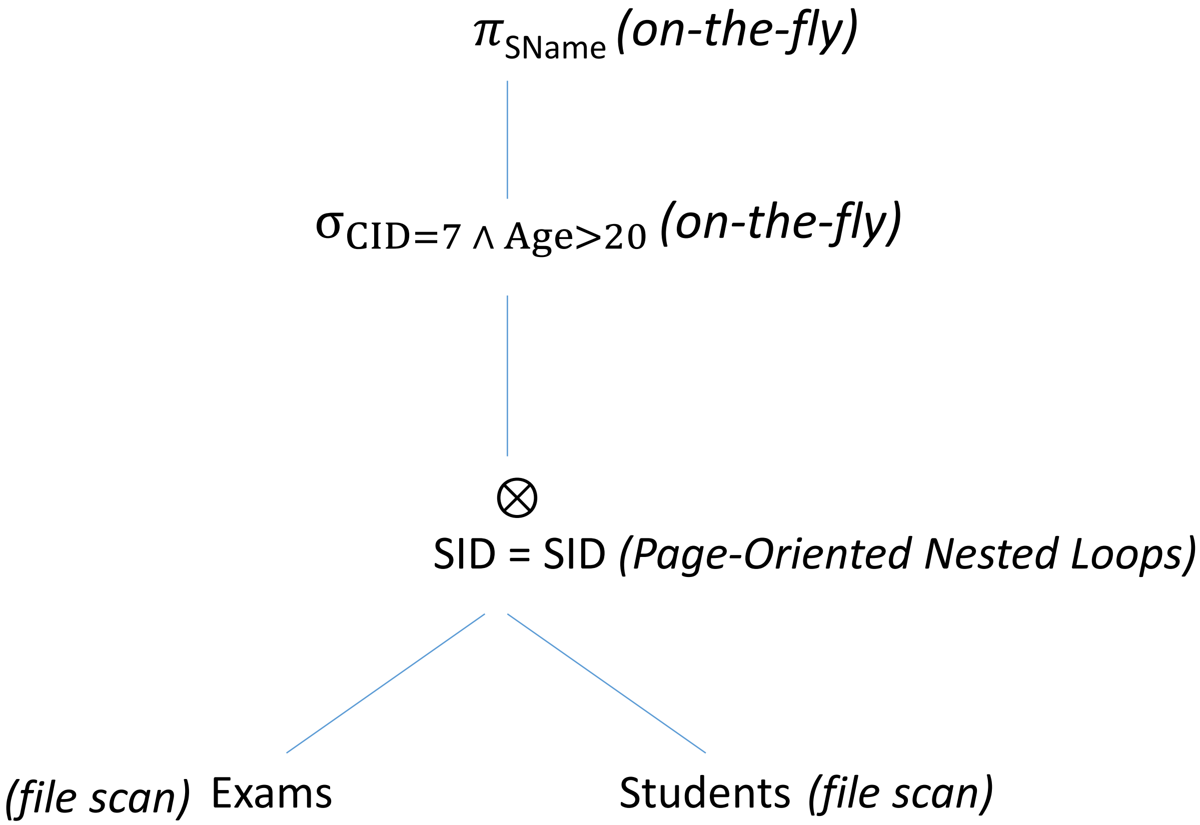
$\pi_{S.SID,\ MIN(E.EDate)}($
$HAVING_{COUNT(*)\ >\ 2}($
$GROUP\ BY_{S.SID}($
$\pi_{S.SID,\ E.EDate}($
$\sigma_{S.SID\ =\ E.SID\ \land\ E.CID\ =\ C.CID\ \land\ C.Description\ =\ 'Elective'\ \land\ S.Age\ =\ value\_from\_nested\_block}($
$Students \times Exams \times Courses\ )))))$

Motivating Example

* E - 1000 pages *

* S - 500 pages *

```
SELECT S.SName
FROM Exams E, Students S
WHERE E.SID = S.SID AND E.CID = 7
     AND S.Age > 20
```



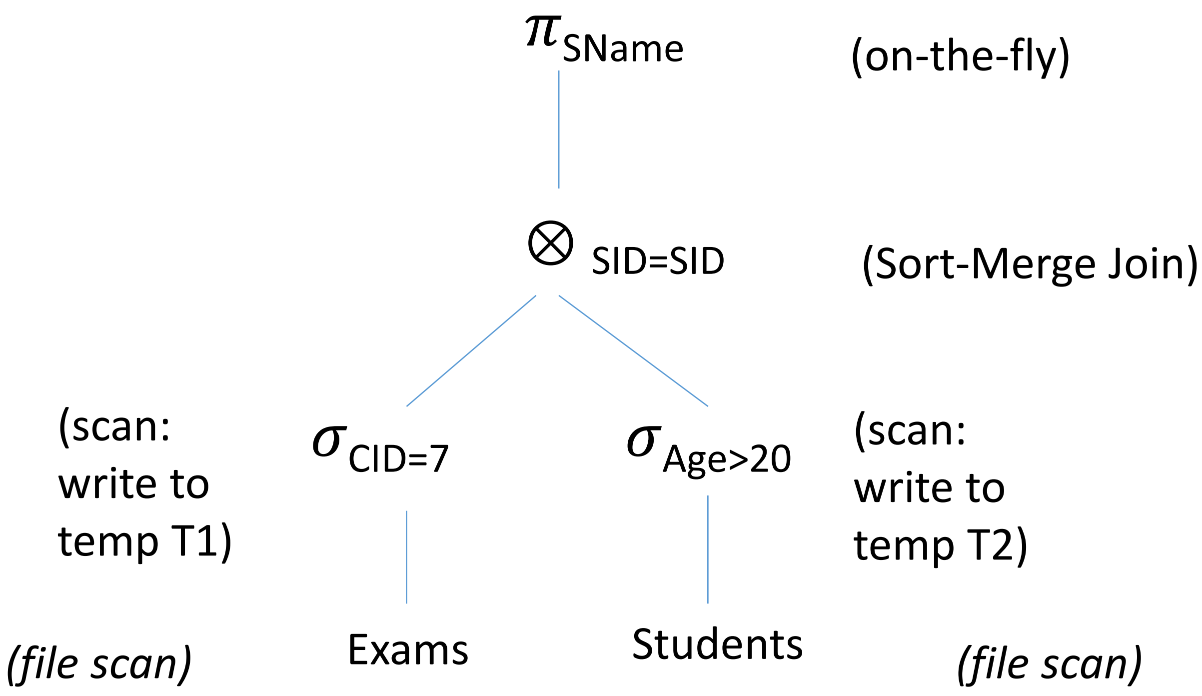
- σ, π – on-the-fly
- cost of plan – very high:
  - 1000 + 1000 * 500 = 501,000 I/Os

Motivating Example

* optimizations

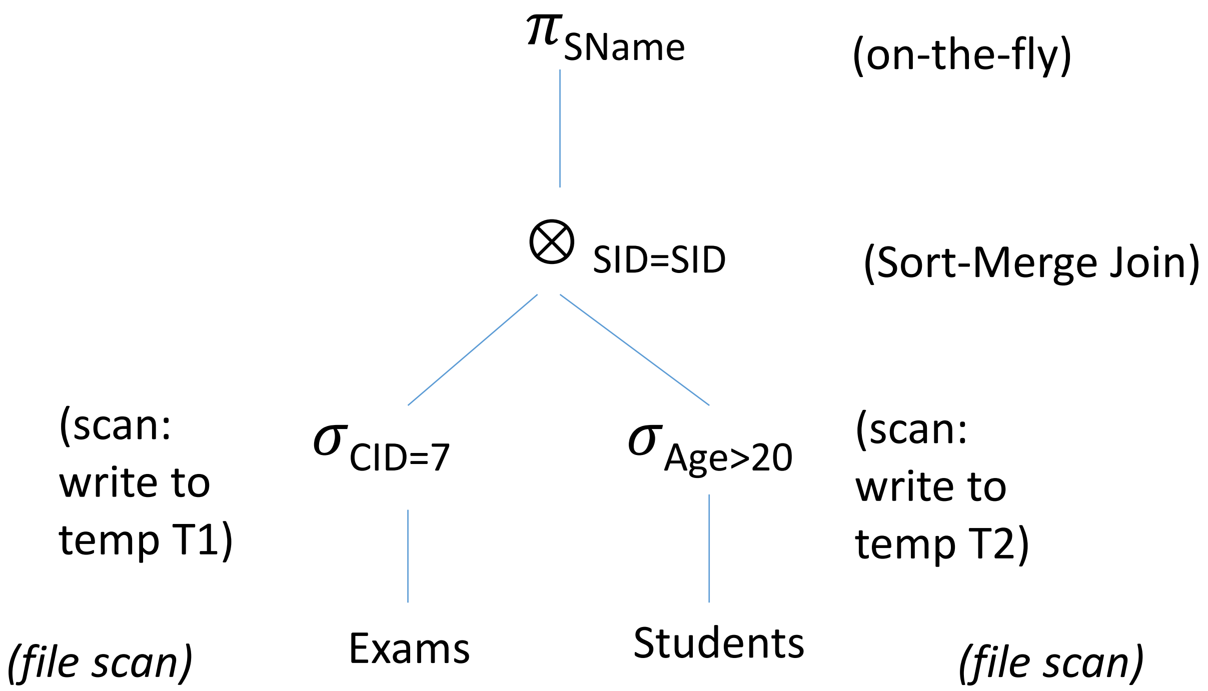
- reduce sizes of the relations to be joined
  - push selections, projections ahead of the join

- alternative plans
  - push selections ahead of joins
- selection
  - file scan
  - write the result to a temporary relation on disk
- join the temporary relations using Sort-Merge Join



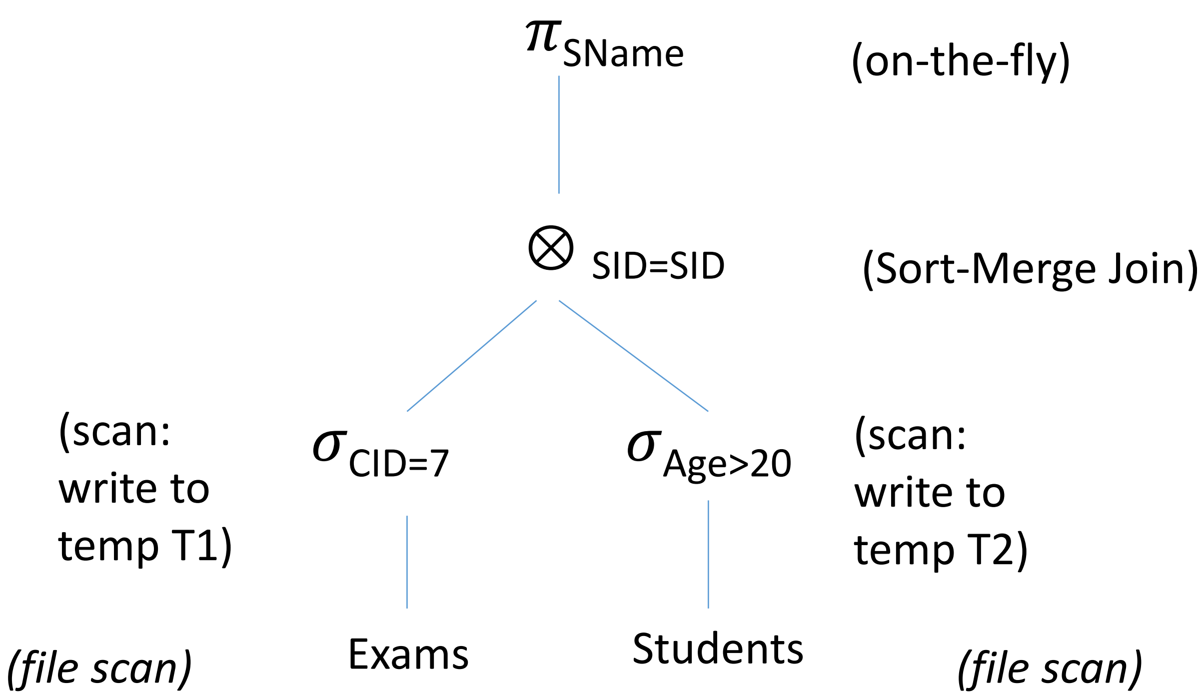
Motivating Example

- 5 available buffer pages
- cost
  - $\sigma_{CID=7}$
    - scan Exams: 1000 I/Os
    - write T1
      - assume exams are uniformly distributed across all courses, i.e., T1 has 10 pages (there are 100 courses)
  - $\sigma_{Age>20}$
    - scan Students: 500 I/Os
    - write T2
      - assume ages are uniformly distributed over the range 19 to 22, i.e., T2 has 250 pages



Motivating Example

- 5 available buffer pages
- cost
  - Sort-Merge Join
    - T1 - 10 pages
      - sort T1: 2 * 2 * 10 = 40 I/Os
    - T2 - 250 pages
      - sort T2: 2 * 4 * 250 = 2000 I/Os
    - merge sorted T1 and T2
      - 10 + 250 = 260 I/Os
  - π - on the fly



=> **total cost**: $\underbrace{1000 + 10 + 500 + 250}_{\text{selection}} + \underbrace{40 + 2000 + 260}_{\text{join}} = \mathbf{4060\ I/Os}$

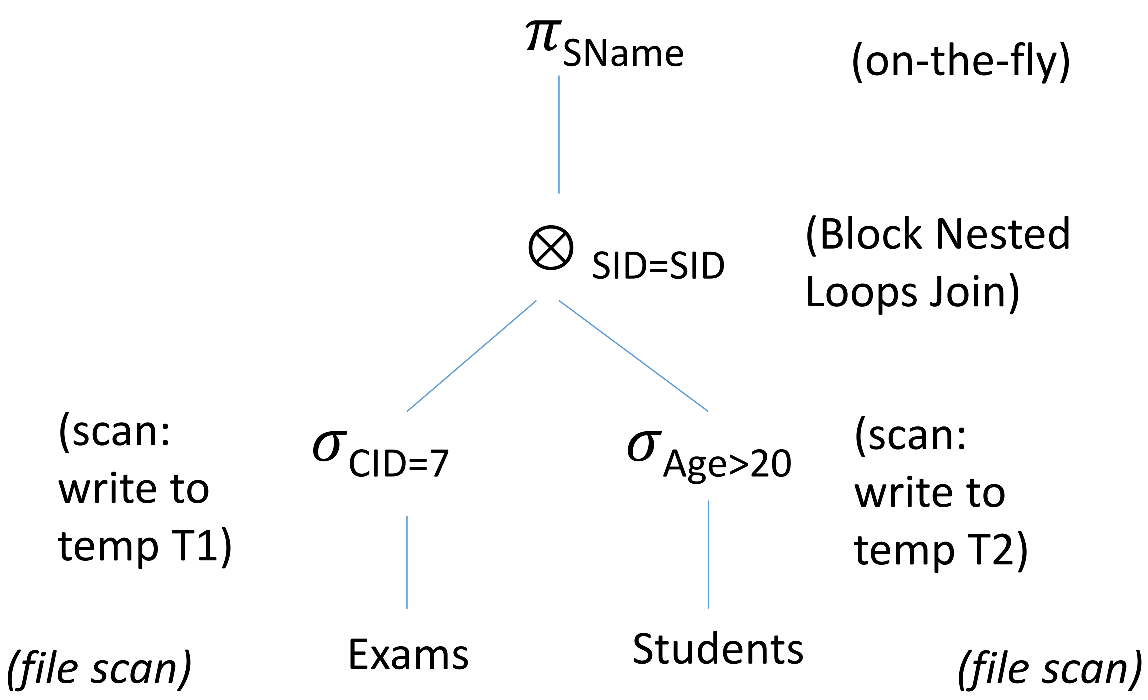
Motivating Example

- 5 available buffer pages
- cost
  - Block Nested Loops Join
    - T1 - 10 pages, T2 - 250 pages
    - T1 - outer relation

      => scan T1: 10 I/Os
    - $\lceil 10/3 \rceil = 4$ T1 blocks

      => T2 scanned 4 times: 4 * 250 = 1000 I/Os
    - BNLJ cost: 10 + 1000 = 1010 I/Os
  - π - on the fly

=> total cost: $\underbrace{1000 + 10 + 500 + 250}_{\text{selection}} + \underbrace{10 + 1000}_{\text{join}} =$ **2770 I/Os**



Motivating Example

- push projections ahead of joins
  - drop unwanted columns while scanning Exams and Students to evaluate selections => T1[SID], T2[SID, SName]
- T1 fits within 3 buffer pages

  => T2 scanned only once

  => **total cost**: about **2000 I/Os**

References

- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8^{th} Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson, 2009
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, 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
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems, http://infolab.stanford.edu/~ullman/fcdb.html