Database Management Systems

Lecture 8

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)

  - 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

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)

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: 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*

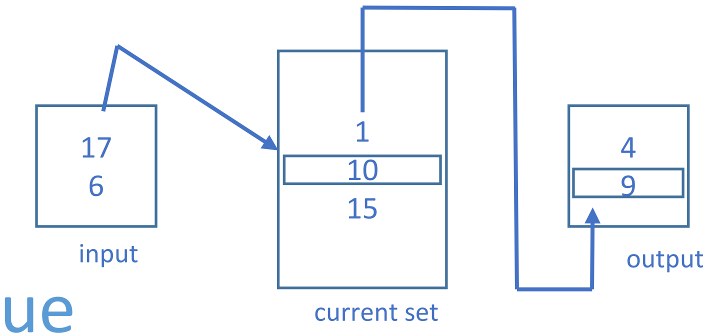
Projection Based on Sorting

- 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

* <u>minimize the number of runs</u> - optional

- external merge sort
  - N pages in the file, B buffer pages => $\lceil N/B \rceil$ runs of B pages each
- <u>improvement</u>
  - algorithm known to produce <u>runs</u> of approximately <u>2*B pages</u> (on average)
  - use 1 page as an input buffer, 1 page as an output buffer
  - the remaining buffer pages are collectively referred to as the *current set*

- <u>example</u> - sort file in ascending order on some key *k*:
  - repeatedly pick record *r* in the current set and append it to the output buffer
  - keep output buffer sorted: *r*'s *k* value >= largest *k* value in the output buffer
  - multiple such records in current set - choose the smallest one



* minimize the number of runs - optional

- example - sort file in ascending order on some key *k*:
  - use the extra space in the current set to bring in the next tuple from the input buffer
  - process all tuples in the input buffer, then read in the next page of the file
  - when the output buffer fills up, write it to disk (add its content to the run that is currently being built)
  - the current run is completed when every *k* value in the current set is < the largest *k* value in the output buffer; when this happens, the output buffer is written out (its content becomes the last page in the current run), and a new run is started

Projection Based on Sorting

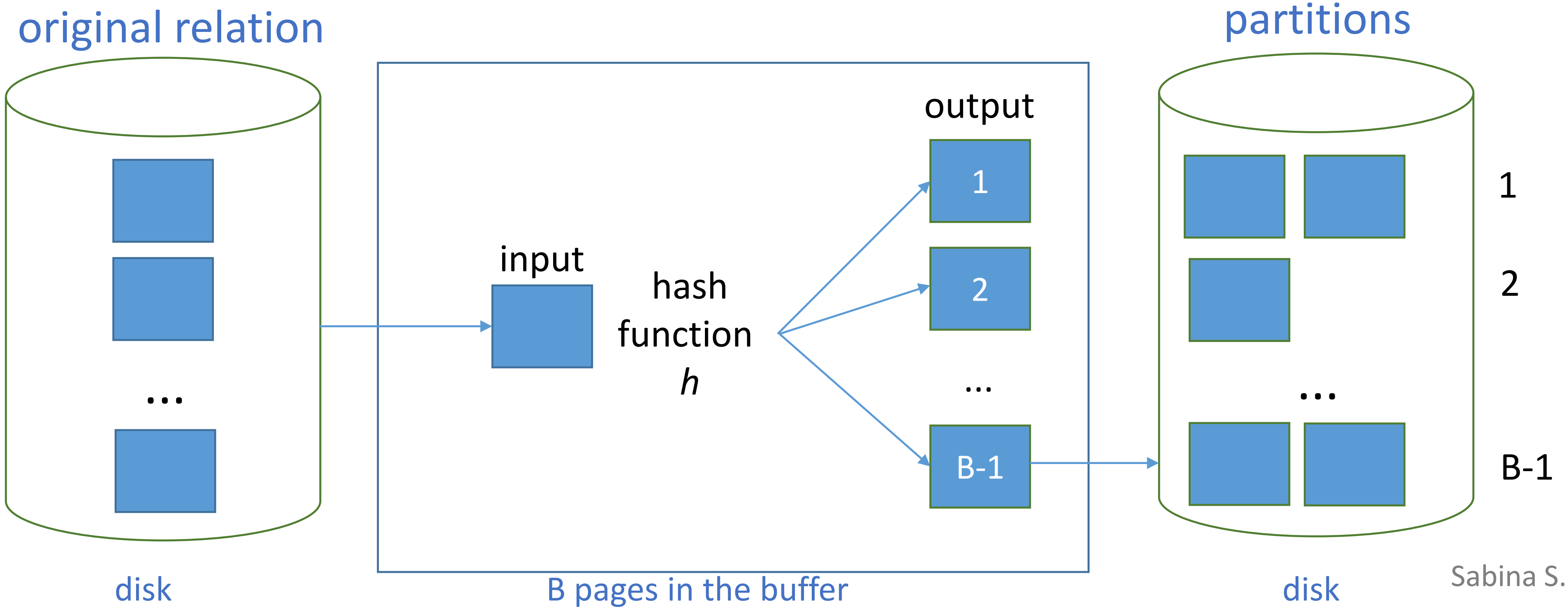
- improvement

  * 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



Sabina S. CS

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

Projection Based on Hashing

- duplicate elimination phase:
    - 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

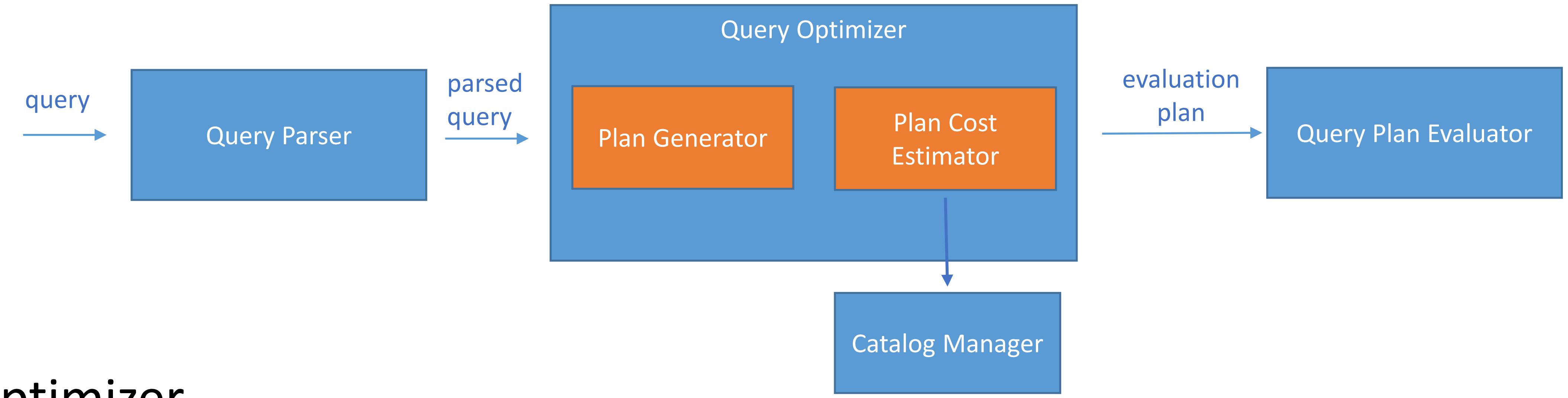
Aggregate Operations

- 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 Q
  - generates alternative plans for Q, estimates their costs, and chooses the one with the least estimated cost
  - uses information from the system catalogs

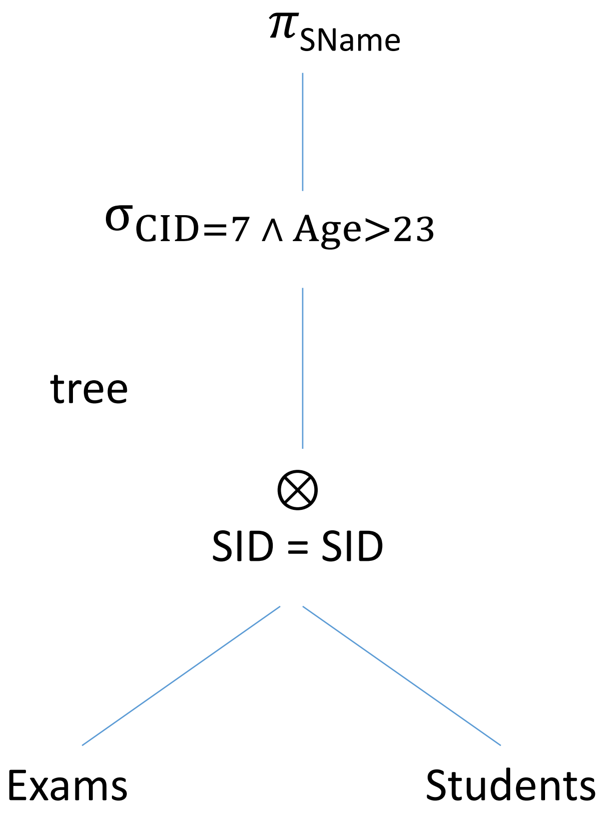
Query Evaluation Plans

```
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



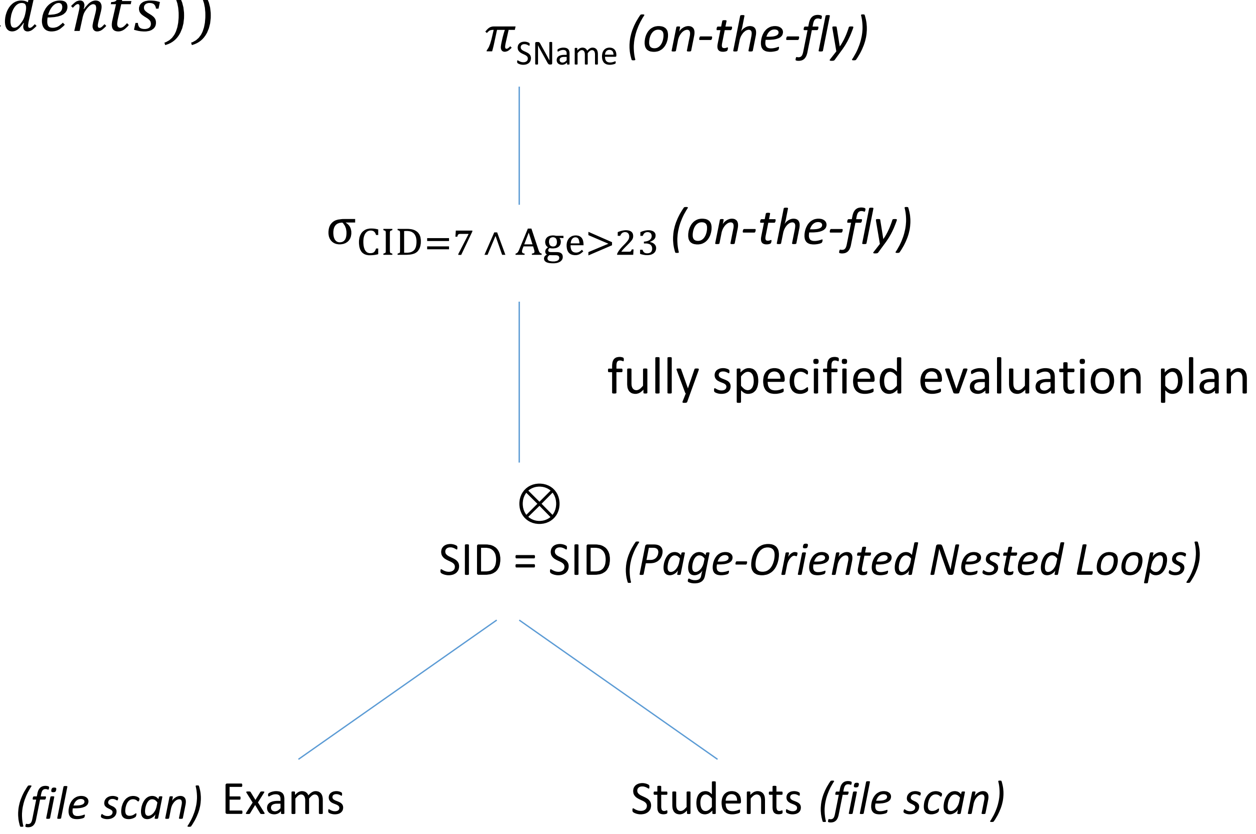
tree

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

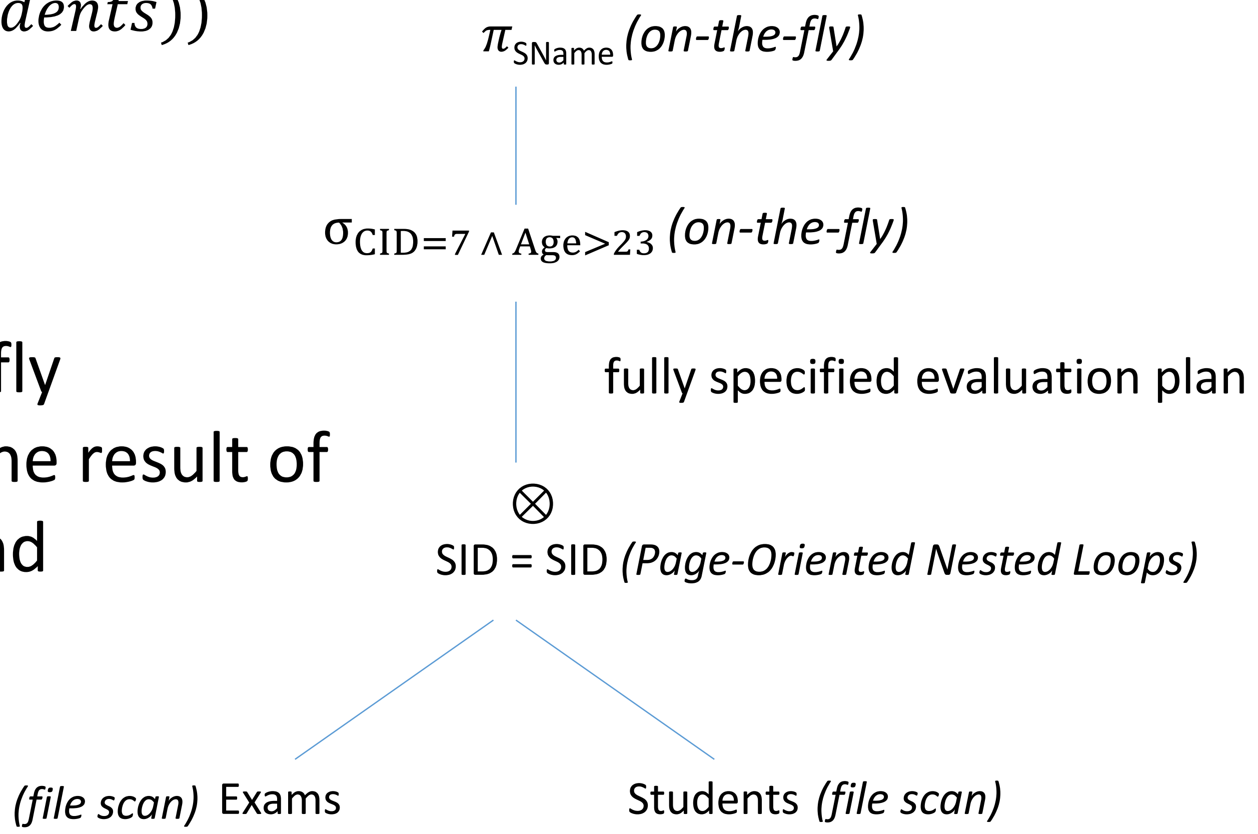


Query Evaluation Plans

```
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 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



Pipelined Evaluation

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

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

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

- index *I* matches *T1*
- v1 - *materialization*
  - evaluate *T1*
  - write out result tuples to temporary relation *R*, i.e., tuples are *materialized*
  - apply the 2^{nd} 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
```

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

- 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

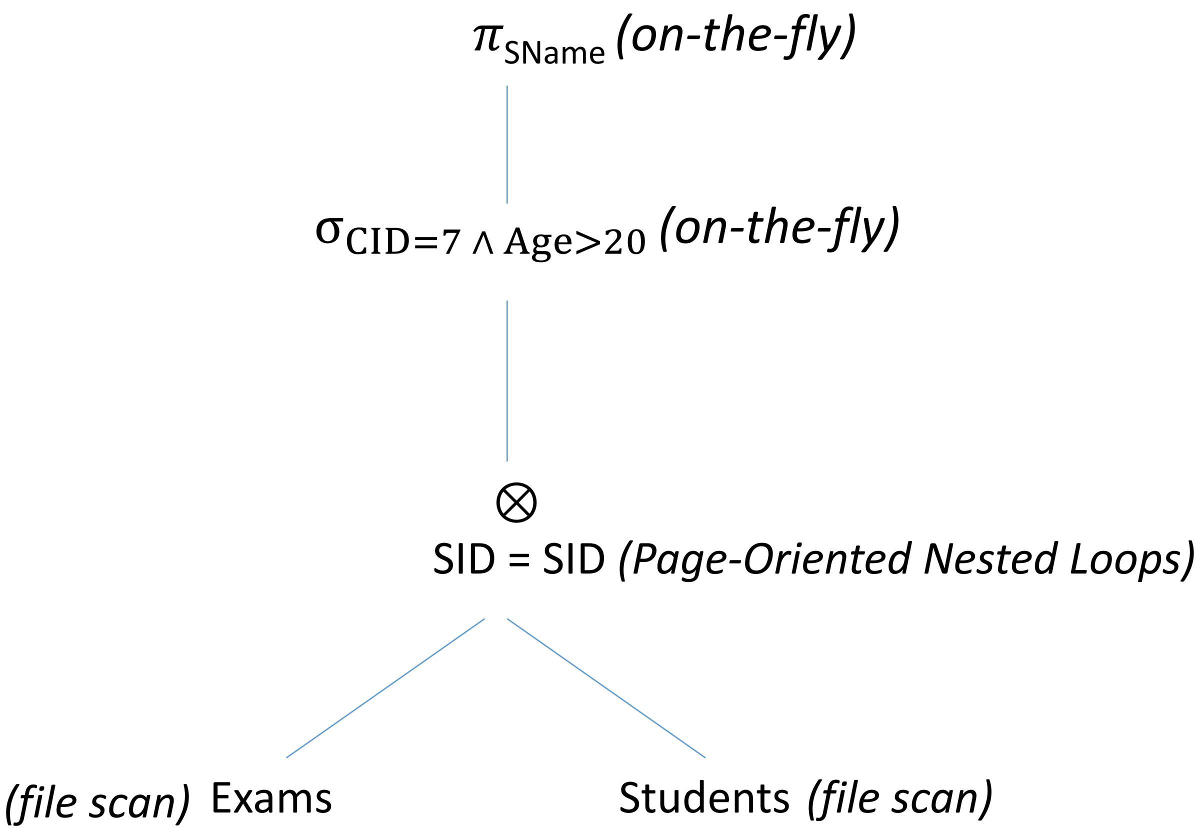
$$\pi_{S.SID,\ MIN(E.EDate)}($$
$$HAVING_{COUNT(*)\ >\ 2}($$
$$GROUP\ BY_{S.SID}($$
$$\pi_{S.SID,\ E.EDate}($$
$$\sigma_{S.SID\ =\ E.SID\ \wedge\ E.CID\ =\ C.CID\ \wedge\ C.Description\ =\ 'Elective'\ \wedge\ S.Age\ =\ value\_from\_nested\_block}($$
$$Students \times Exams \times Courses\ )))))$$

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
```

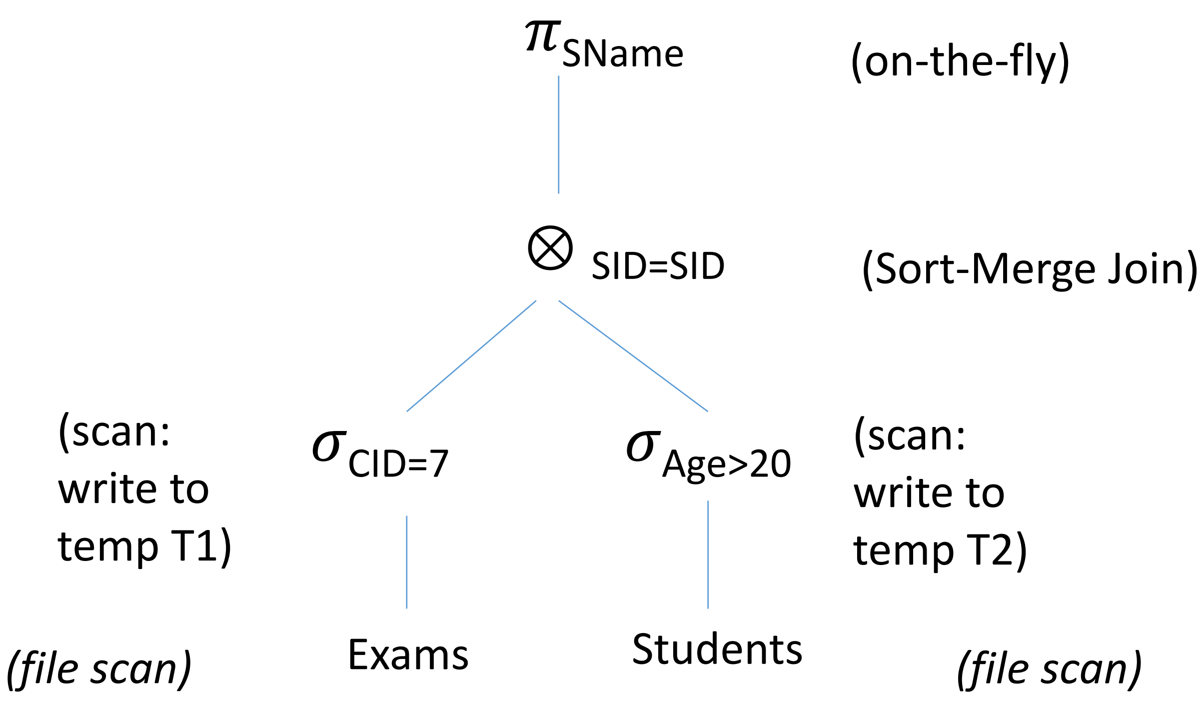


- σ, π – on-the-fly
- cost of plan – very high:
  - 1000 + 1000 * 500 = 501,000 I/Os

Motivating Example

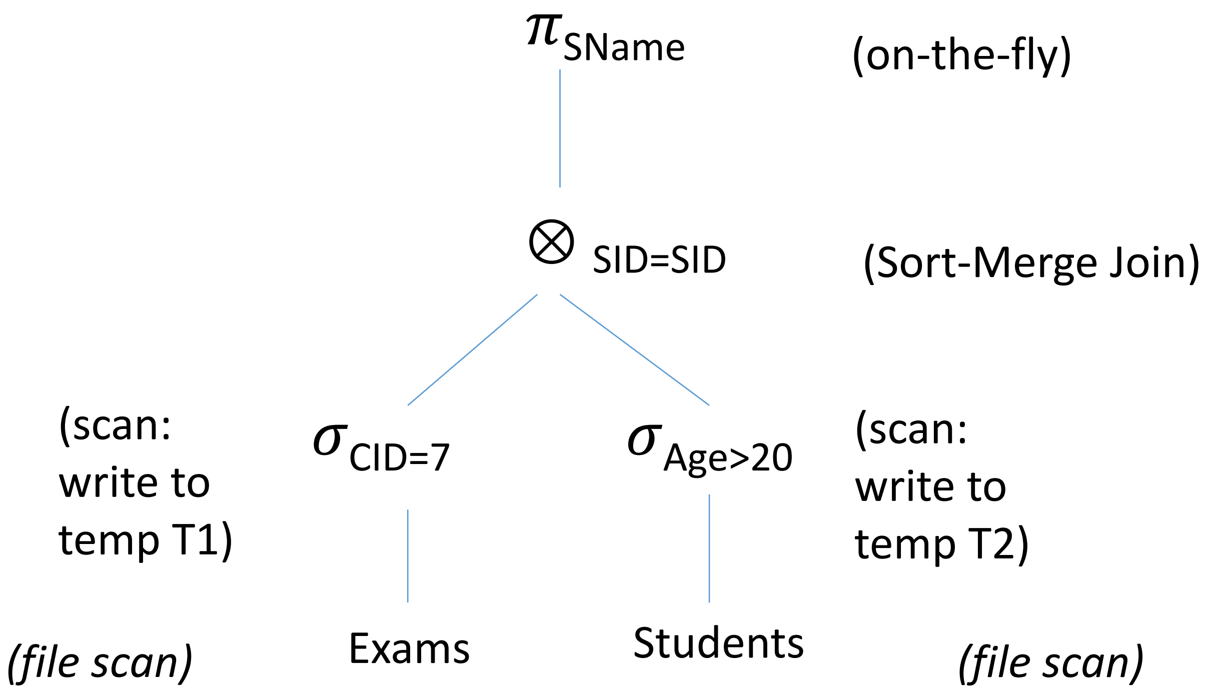
* optimizations

- 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

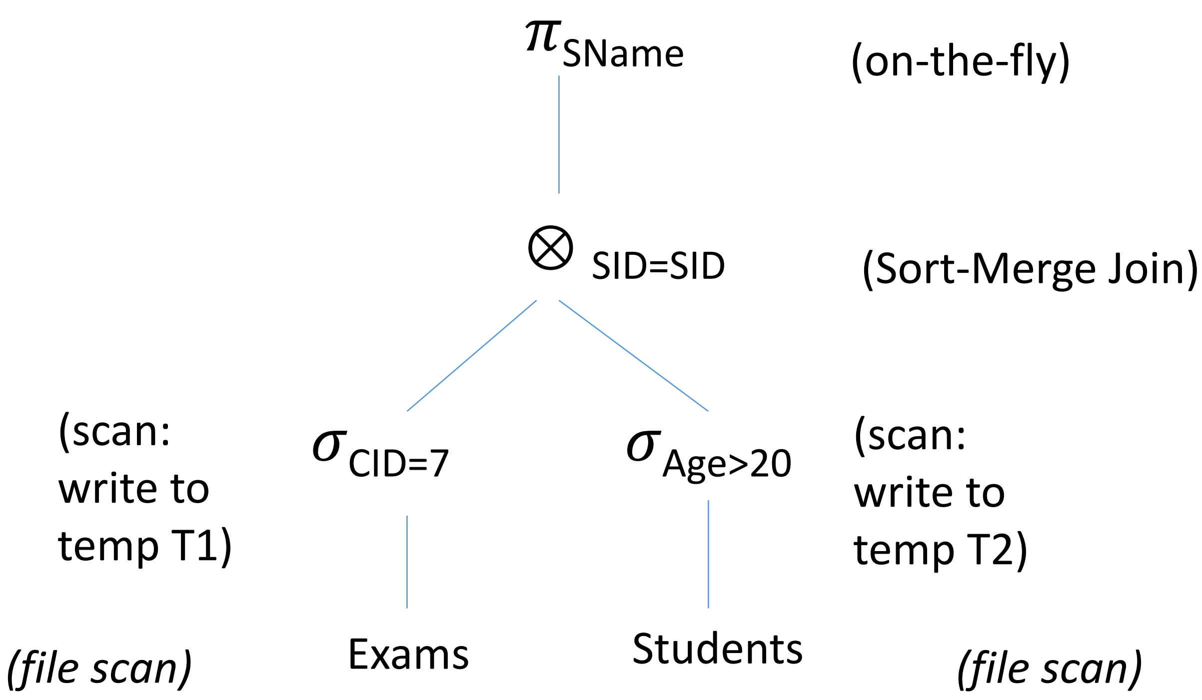
- 5 available buffer pages
- cost
  - $\sigma_{CID=7}$
    - scan Exams: 1000 I/Os
    - write T1
      - assume exams are uniformly distributed across all courses, i.e., T1 has 10 pages (there are 100 courses)
  - $\sigma_{Age>20}$
    - scan Students: 500 I/Os
    - write T2
      - assume ages are 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**: $\underbrace{1000 + 10 + 500 + 250}_{\text{selection}} + \underbrace{40 + 2000 + 260}_{\text{join}} = \mathbf{4060\ I/Os}$



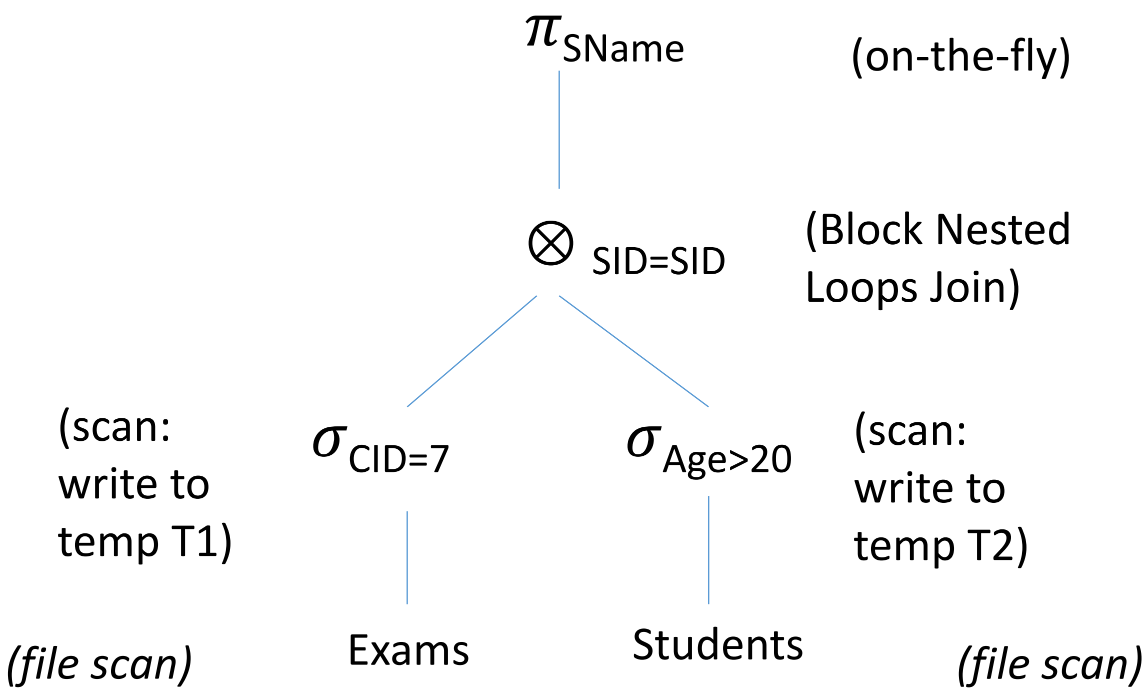
Motivating Example

- 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: $\underbrace{1000 + 10 + 500 + 250}_{\text{selection}} + \underbrace{10 + 1000}_{\text{join}} = \mathbf{2770\ I/Os}$



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

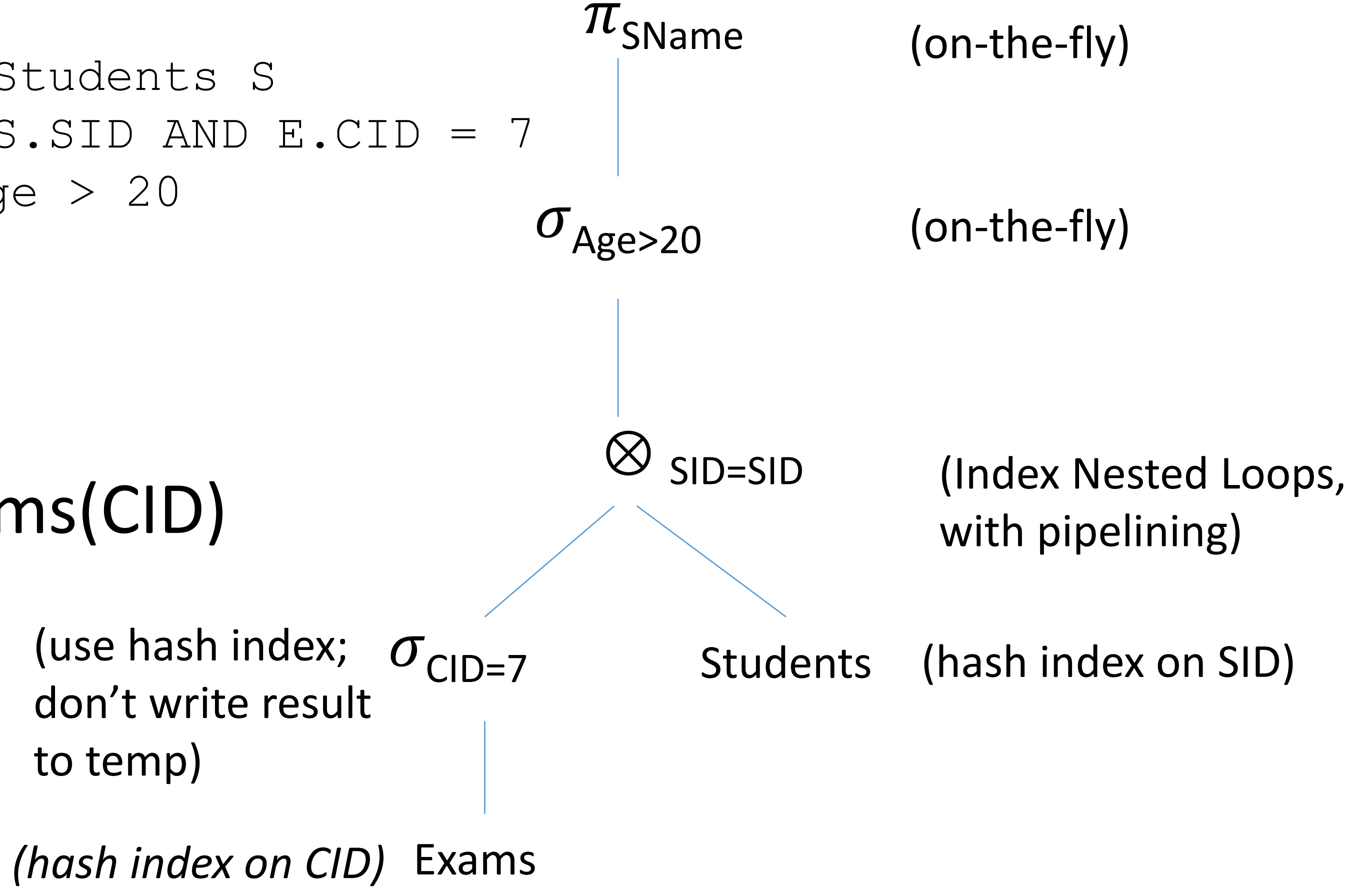
  => T2 scanned only once

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

Motivating Example

* optimizations

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



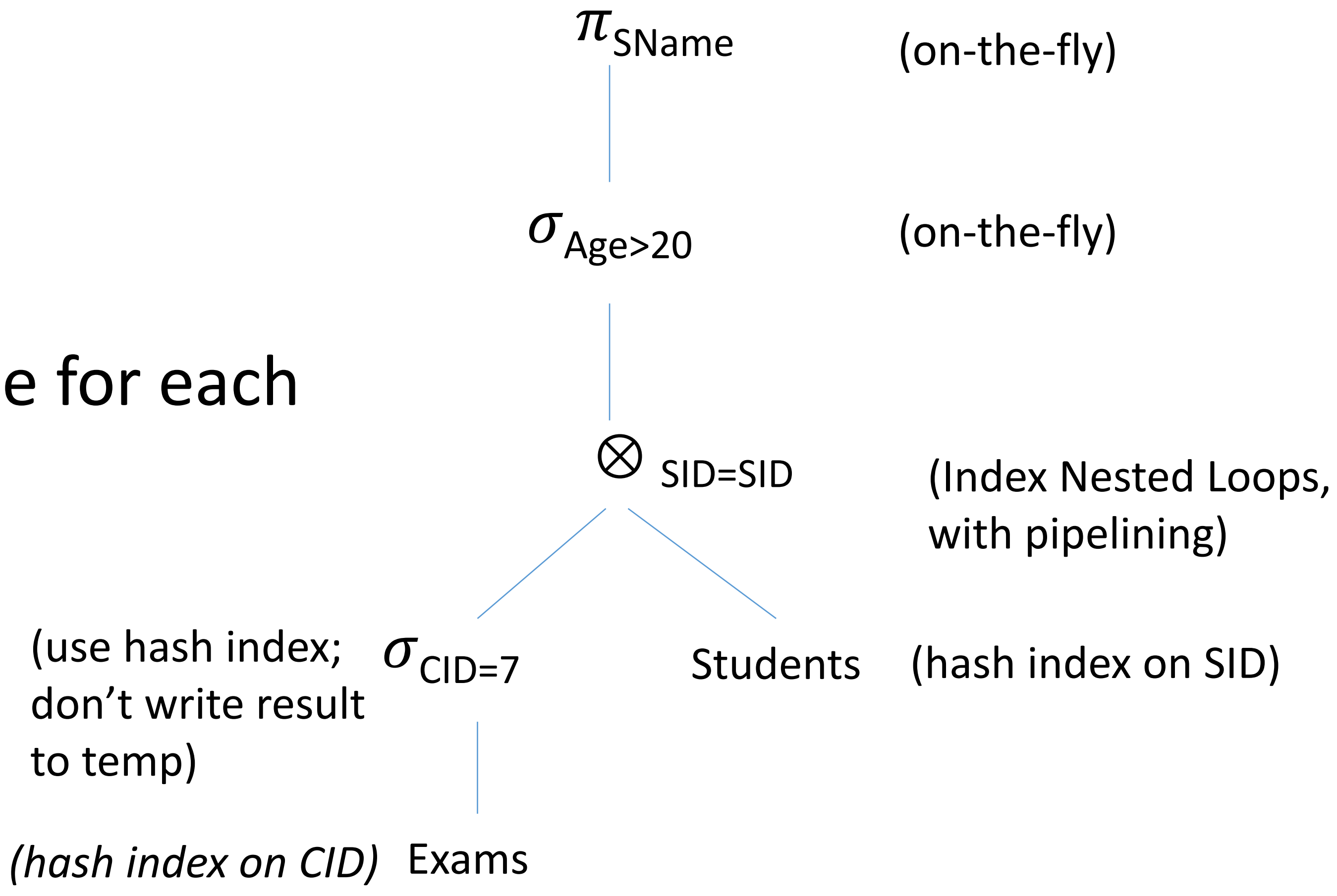
- investigate the use of indexes

- clustered static hash index on Exams(CID)
- hash index on Students(SID)

- cost
  - $\sigma_{CID=7}$
    - assume exams are uniformly distributed across all courses => 100,000 exams / 100 courses => 1,000 exams / course
    - clustered index on CID => 1,000 tuples for course with CID=7 appear consecutively within the same bucket => cost: 10 I/Os
    - the result of the selection is not materialized, the join is pipelined

Motivating Example

- cost
  - Index Nested Loops
    - find matching Students tuple for each selected exam
    - use hash index on SID
      - assume the index uses a1 => cost of 1.2 I/Os (on avg.) per exam



  - σ, π – performed on-the-fly on each tuple in the result of the join

=> **total cost** = $\underbrace{10}_{\sigma\text{ on Exams}} + \underbrace{1000}_{\text{num. of Exams tuples}} * \underbrace{1.2}_{\text{find matching Students tuple (on avg.)}} =$ **1210** I/Os

* can we push the selection *Age>20* ahead of the join?

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