Database Management Systems

Lecture 9

Evaluating Relational Operators

Query Optimization (III)

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

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, 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 - 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

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 / non-clustered)

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

Selection

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

- B+ tree index on *attr*

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

- 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 - techniques
    - 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

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

- scan Exams: 1000 I/O
- 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: 2500 I/Os

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

Projection Based on Sorting

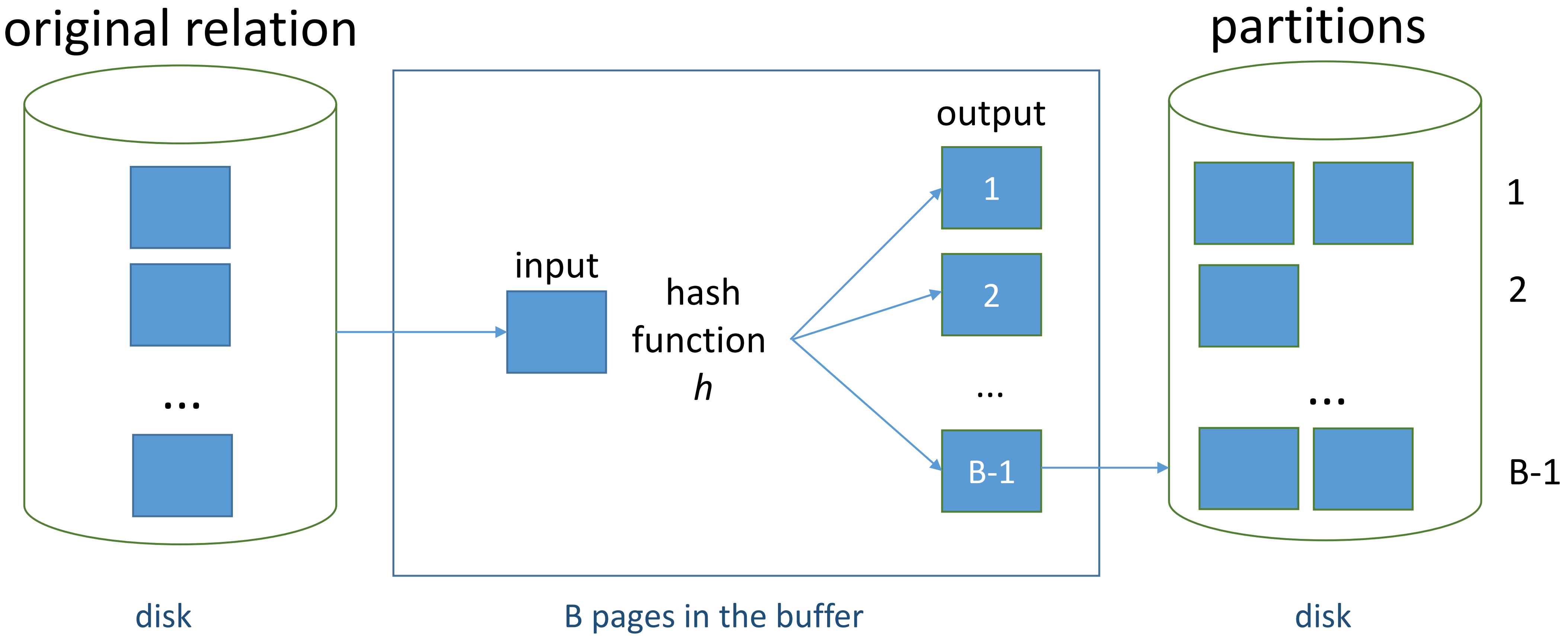
- improvement
  - modify the sorting algorithm to do projection with duplicate elimination
    - modify pass 0 of external sort - eliminate unwanted columns
      - read in B pages from E
      - write out (T/M) * B internally sorted pages of E'
        - more aggressive approach: write out 2*B internally 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
  - 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)
  - pass 1
    - read in all runs – cost: 250 I/O
    - merge runs
  - total cost : 1500 I/O

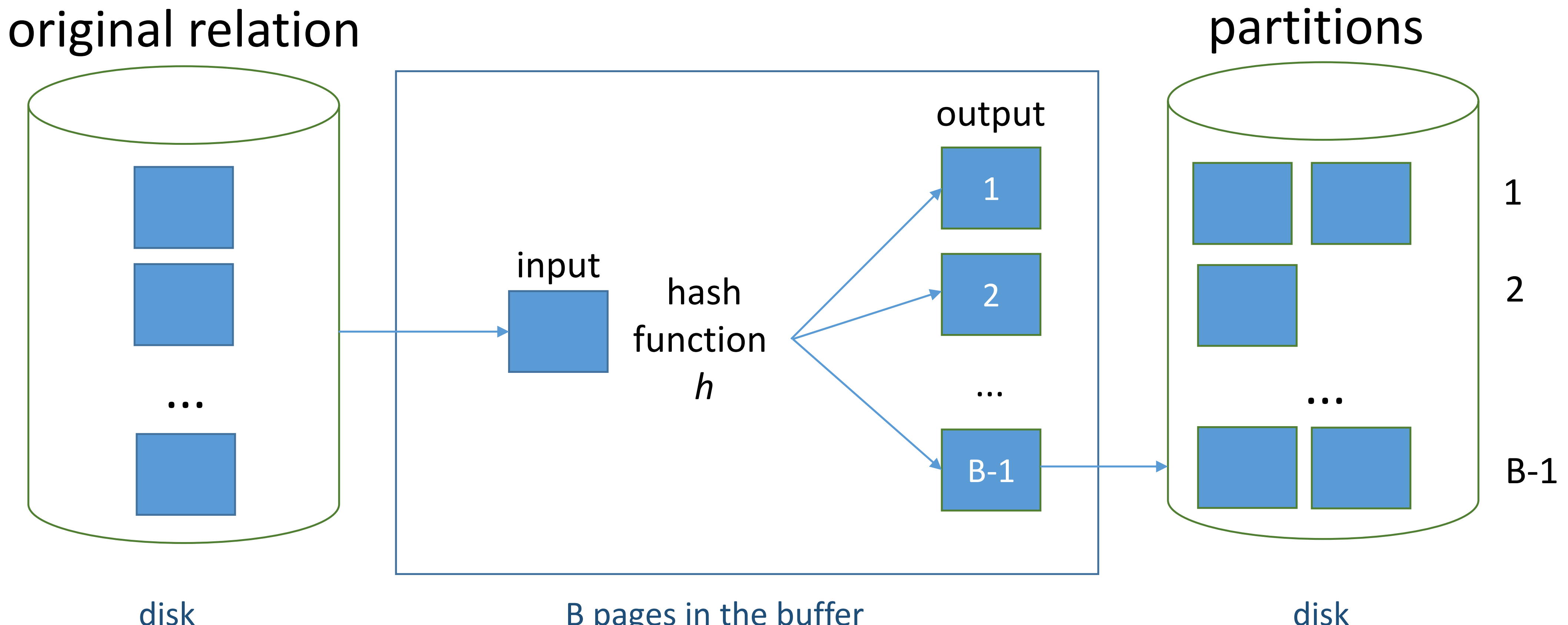
Projection Based on Hashing

- phases
  - partitioning
  - duplicate elimination



Projection Based on Hashing

- 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
        - apply hash function *h* to the combination of all remaining attributes
        - write *t* to the output buffer page that it is hashed to by *h*

    => B-1 partitions
    - partition
        - collection of tuples
            - common hash value
            - no unwanted fields
    - 2 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 for P 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 as they are detected
      - 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 (i.e., join condition for intersection - equality on all fields, no join condition for cross-product)
- union, set-difference
  - similar

- union: $R \cup S$
  - 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
    - index-only scan
  - 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

* balanced merge sort*

- table T, |T| = 3100 records
- 1 run – at most 100 records
- runs distribution – 4 files

- initial runs – distribution
    - about half of the files
- merge runs, write runs to remaining files
- continue until a single run is produced

- notation
    - x^y
        - y runs with relative length = 1
    - run with relative length 1
        - produced with an internal sorting algorithm

* balanced merge sort*

F1	F2	F3	F4	obs
1^{16}	1^{15}	-	-	runs distribution
-	-	2^{8}	$2^{7}1^{1}$	merging – alternate F3 and F4; copy one run
4^{4}	$4^{3}3^{1}$	-	-	merging – alternate F1 and F2
-	-	8^{2}	$8^{1}7^{1}$	merging – alternate F3 and F4
16^{1}	15^{1}	-	-	merging – alternate F1 and F2
-	-	31^{1}	-	merging, final result obtained in F3

* polyphase merge sort *

- determine an initial configuration for the distribution of runs
  - one file F - empty
- merge runs from all files into F until a file F' is empty
- continue until a single run is produced

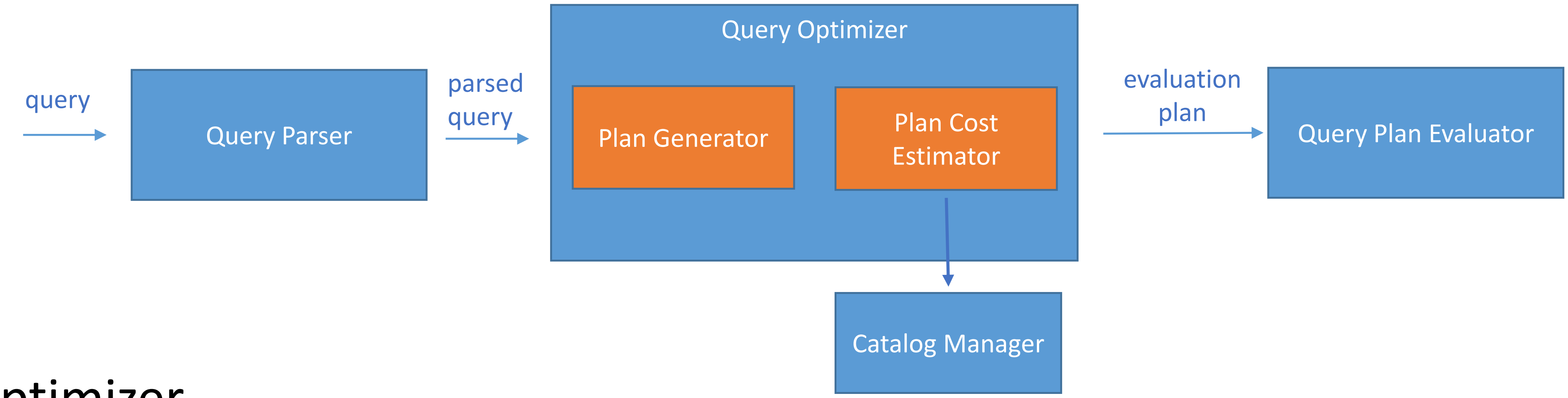
* polyphase merge sort *

F1	F2	F3	F4	obs
-	1^7	1^{11}	1^{13}	runs distribution
3^7	-	1^4	1^6	merge in F1 until F2 is empty
3^3	5^4	-	1^2	merge in F2 until F3 is empty
3^1	5^2	9^2	-	merge in F3 until F4 is empty
-	5^1	9^1	17^1	merge in F4 until F1 is empty
31^1	-	-	-	merge, final result obtained in F1

* polyphase merge sort * - initial configuration

F1	F2	F3	F4	obs
1	-	-	-	the run in F1 is obtained from the other files
-	1	1	1	the run in F4 is obtained from the other files
1	2	2	-	runs in F3 are obtained from the other files
3	4	-	2	runs in F2 are obtained from the other files
7	-	4	6	runs in F1 are obtained from the other files
-	7	11	13	runs in F4 are obtained from the other files
13	20	24	-	runs in F3 are obtained from the other files
...	...	...	...	...
c_n	b_n	a_n	-	$a_n >= b_n >= c_n$
$c_n + a_n$	$b_n + a_n$	-	a_n	

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</u>, EDate: date, Grade: integer, FacultyMember: string)

- 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

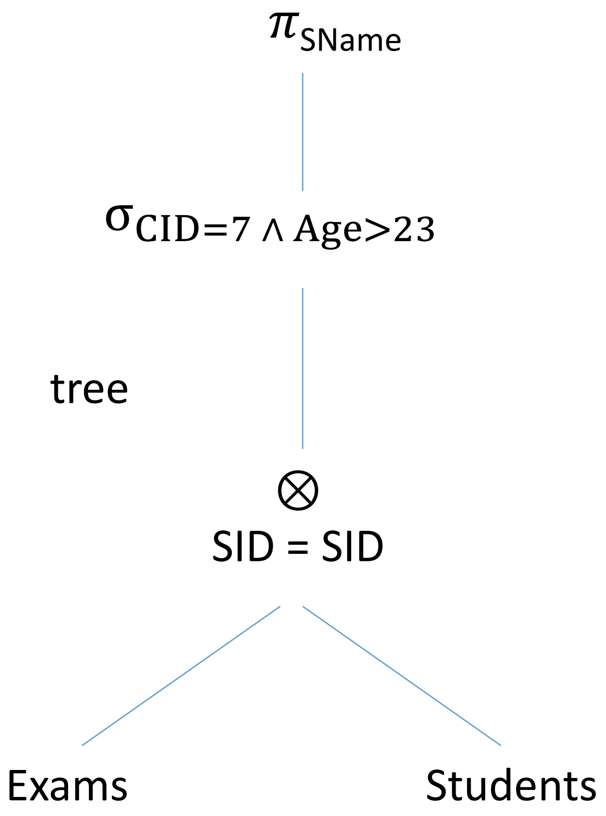
```
SELECT S.SName
FROM Exams E, Students S
WHERE E.SID = S.SID AND E.CID = 7
     S.Age > 23
```

query

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

relational algebra expression

tree

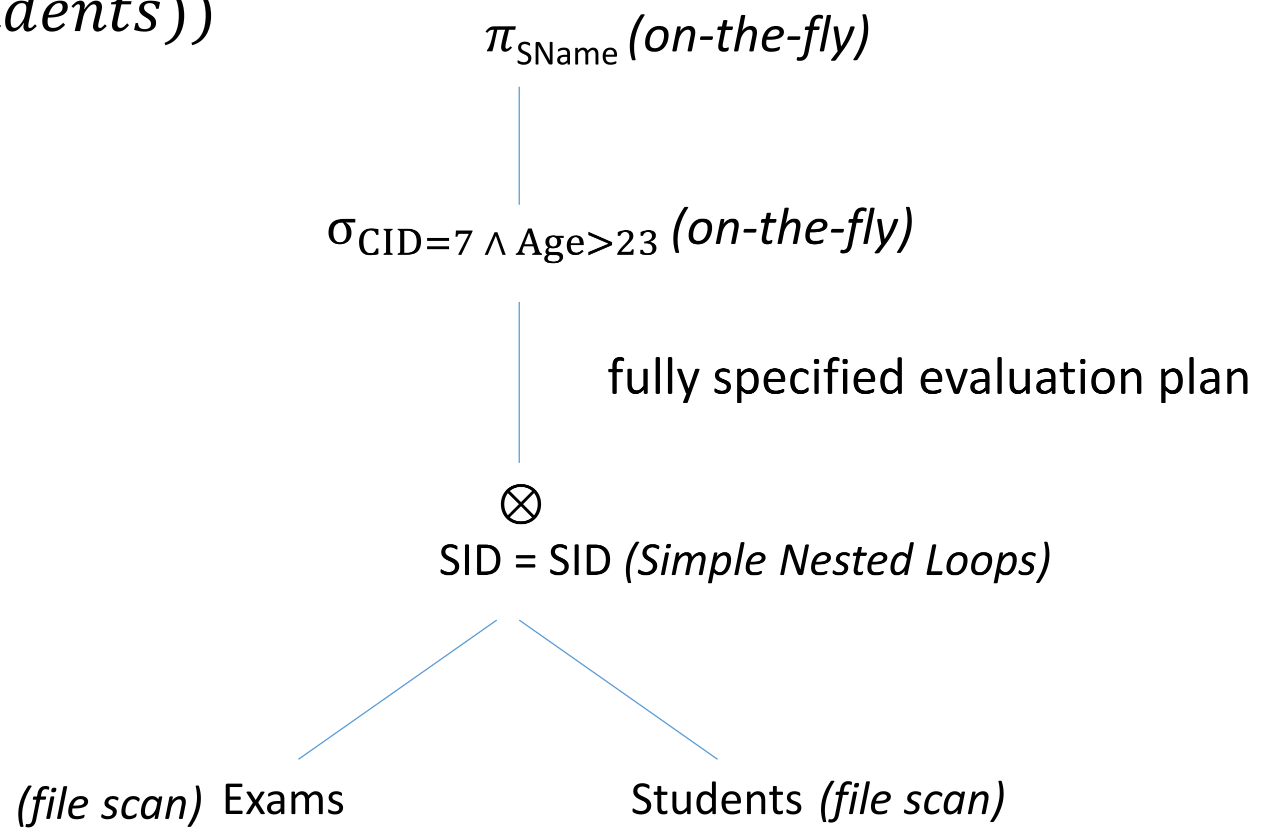


Query Evaluation Plans

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

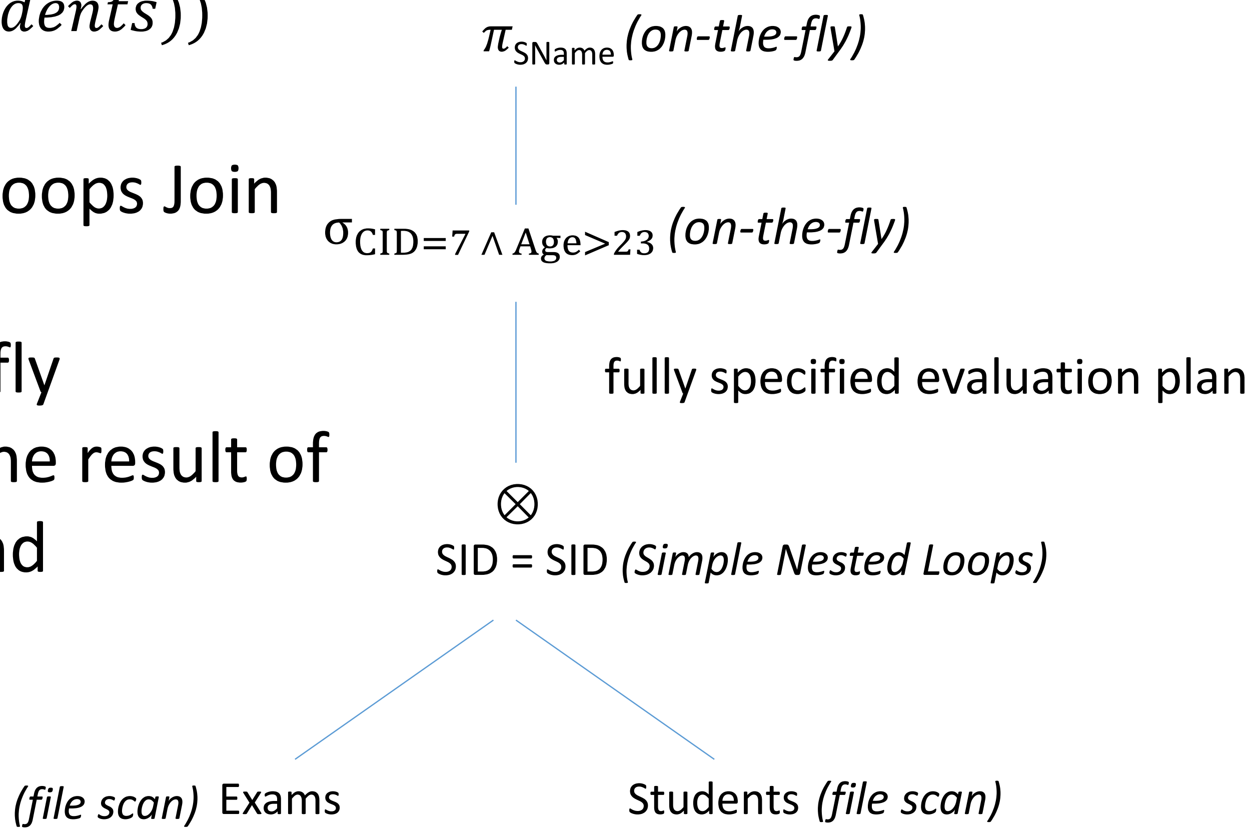


Query Evaluation Plans

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

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

- e.g., 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



Pipelined Evaluation

```
SELECT *
FROM Exams
```

$$\texttt{WHERE } \underbrace{\texttt{EDate > '1-1-2017'}}_{T1} \texttt{ AND } \underbrace{\texttt{Grade > 8}}_{T2}$$

$$\sigma_{Grade>8}$$
$$|$$
$$\sigma_{\mathrm{EDate}>'1-1-2017'}$$
$$|$$
Exams

$$\sigma_{Grade>8}(\sigma_{EDate>'1-1-2017'}(Exams))$$

- 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 > '1-1-2017' AND Grade > 8
```

T1 *T2*

$\sigma_{Grade>8}$

$\sigma_{\mathrm{EDate}>'1-1-2017'}$

Exams

- 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*
- iterator interface
  - combining code for individual operators into an executable plan (functions *open, get_next, close)*
  - supports pipelining

Query Blocks – Units of Optimization

- optimize SQL query Q
  - parse Q => collection of query *blocks*
  - 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:

outer block

```
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
```

$$\pi_{S.SID,\ E.EDate}($$
$$\sigma_{S.SID = E.SID \wedge E.CID = C.CID \wedge C.Description = 'Elective' \wedge S.Age = value\_from\_nested\_block}($$
$$Students \times Exams \times Courses\ ))$$

- 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

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

References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2nd Edition), McGraw-Hill, 2000
- [Da03] DATE, C.J., An Introduction to Database Systems (8th Edition), Addison-Wesley, 2003
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, 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