COP 3540: Introduction to Database Structures

Fall 2017

Query Optimization

M. Rathod

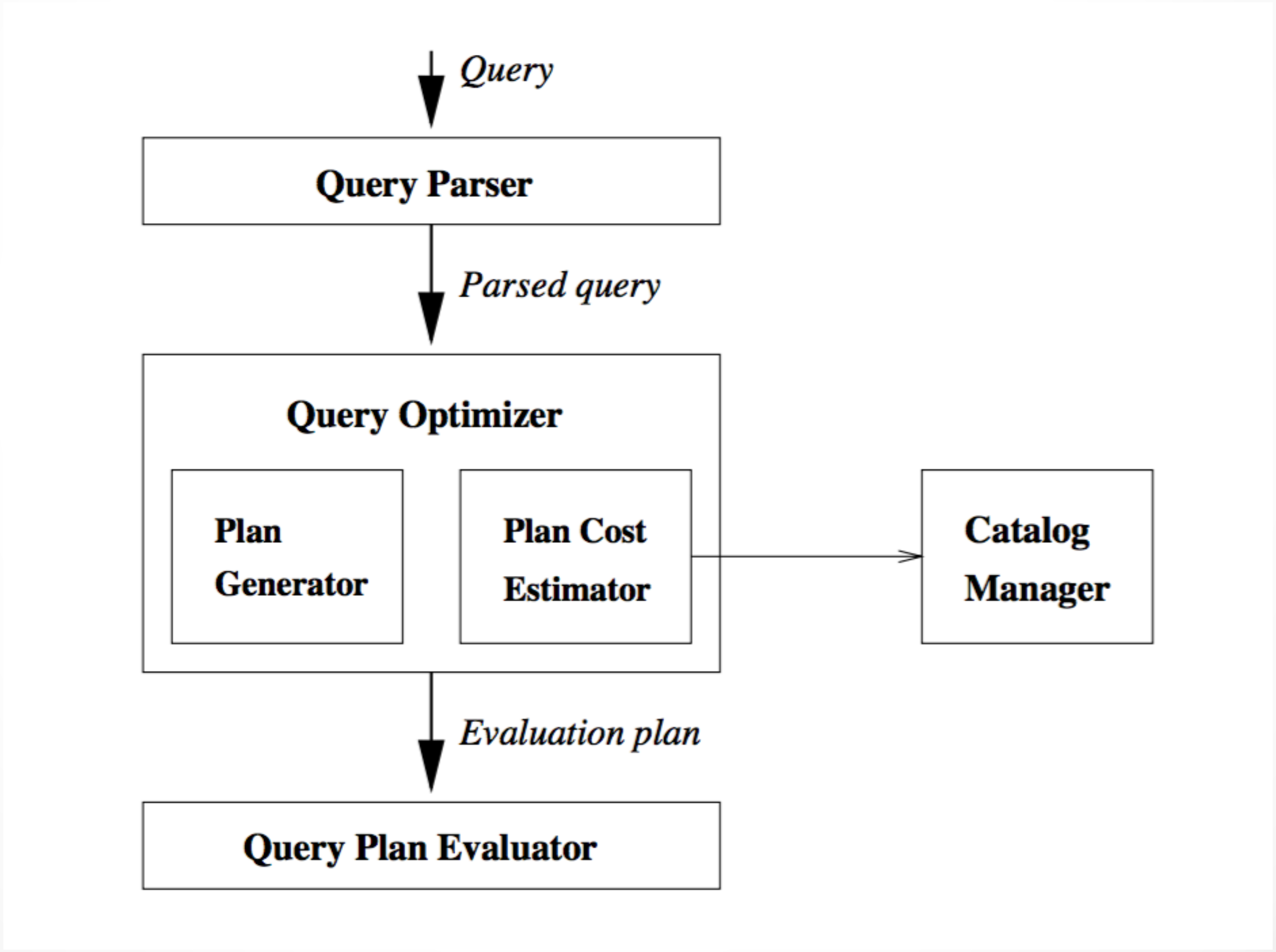
Query Optimization

- Queries consist of a few basic operators
- Operator implementation should be optimized for performance
- Queries are parsed and then presented to a query optimizer

query optimizer:

- identifies an efficient execution plan for evaluating the query
- generates alternative plans
- chooses the plan with the least estimated cost

Query Parsing, Optimization and Execution



Query Optimization

query evaluation plan consists of an extended relational algebra tree

```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
        AND R.bid = 100 AND S.rating > 5
```

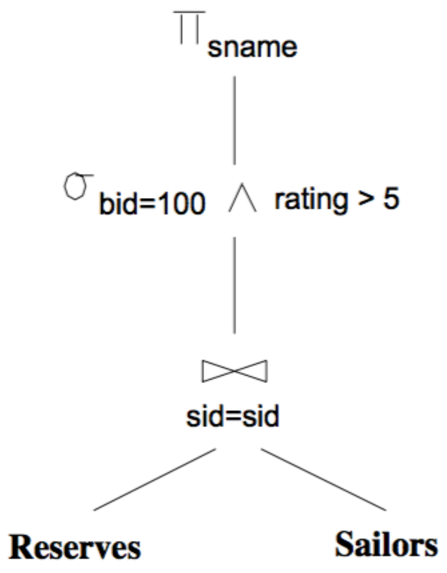
above query can be represented as:

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$

Query Optimization

query evaluation plan consists of an extended relational algebra tree

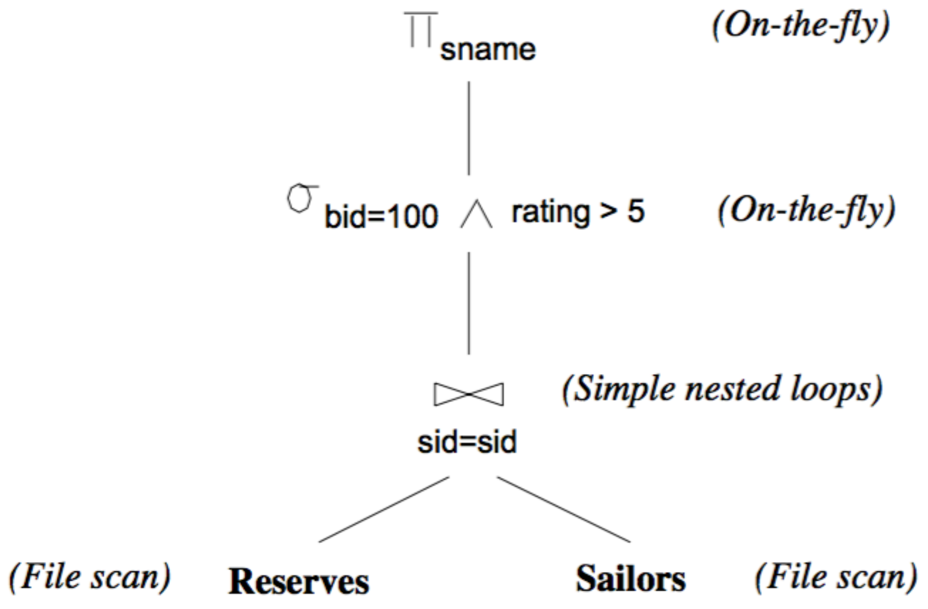
$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$



Query Optimization

To obtain a fully specified evaluation plan, we must decide on an implementation for each of the algebra operations involved

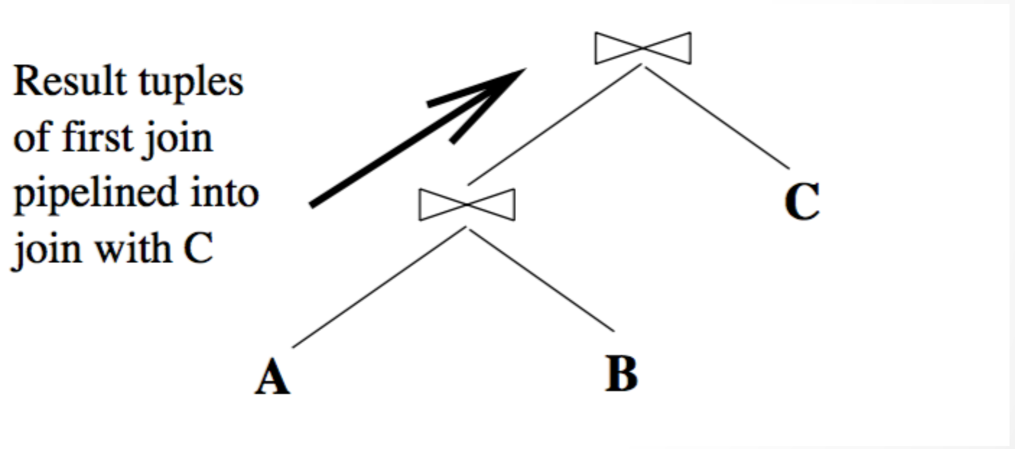
$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$



Query Optimization

If query is composed of several operators, the result of one operator is sometimes **pipelined** to another operator (without creating a temporary relation to hold the intermediate result)

$$(A \bowtie B) \bowtie C$$

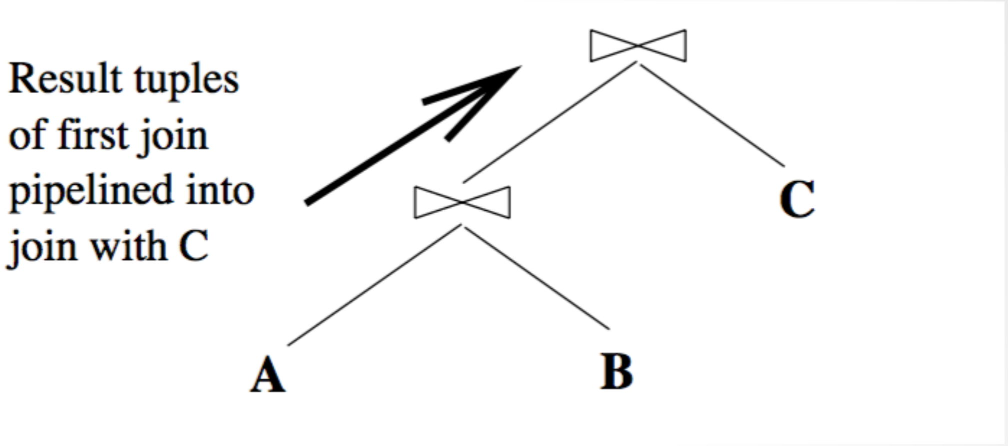


Pipelining the output of an operator into the next operator saves the cost of writing out the intermediate result and reading it back in, and the cost savings can be significant

Query Optimization

pipelining

$(A \bowtie B) \bowtie C$



- evaluation is initiated from the root (node joining A and B produces tuples as and when they are requested by its parent node)
- root node gets a page of tuples from its left child (the outer relation)
- all the matching inner tuples are retrieved (using either an index or a scan) and joined with matching outer tuples;
- current page of outer tuples is then discarded
- request is then made to the left child for the next page of tuples, and the process is repeated

Query Optimization

If the output of an operator is saved in a temporary relation for processing by the next operator, we say that the tuples are **materialized**

Pipelined evaluation has lower overhead costs than materialization and is chosen whenever the algorithm for the operator evaluation permits it

- When the input relation to a unary operator (selection or projection) is pipelined into it, we sometimes say that the operator is applied **on-the-fly**

Query Optimization

relational operators that form the nodes of a plan tree (to be evaluated using pipelining) typically support a uniform **iterator** interface

iterator interface for an operator includes:

- **open**: function that initializes the state of the iterator by allocating buffers for its inputs and output and used to pass in arguments such as selection conditions that modify the behavior of the operator
- **get_next**: function calls the get_next function on each input node and calls operator-specific code to process the input tuples, output tuples generated by the processing are placed in the output buffer of the operator, and the state of the iterator is updated to keep track of how much input has been consumed
- **close**: function is called (by the code that initiated execution of this operator) to deallocate state information

Query Optimization

iterator interface

If the algorithm implemented for the operator allows input tuples to be processed completely when they are received, input tuples are not materialized and the evaluation is pipelined

If the algorithm examines the same input tuples several times, they are materialized.

also used to encapsulate access methods such as B+ trees and hash-based indexes

Query Optimization

System Catalog in Relational DBMS

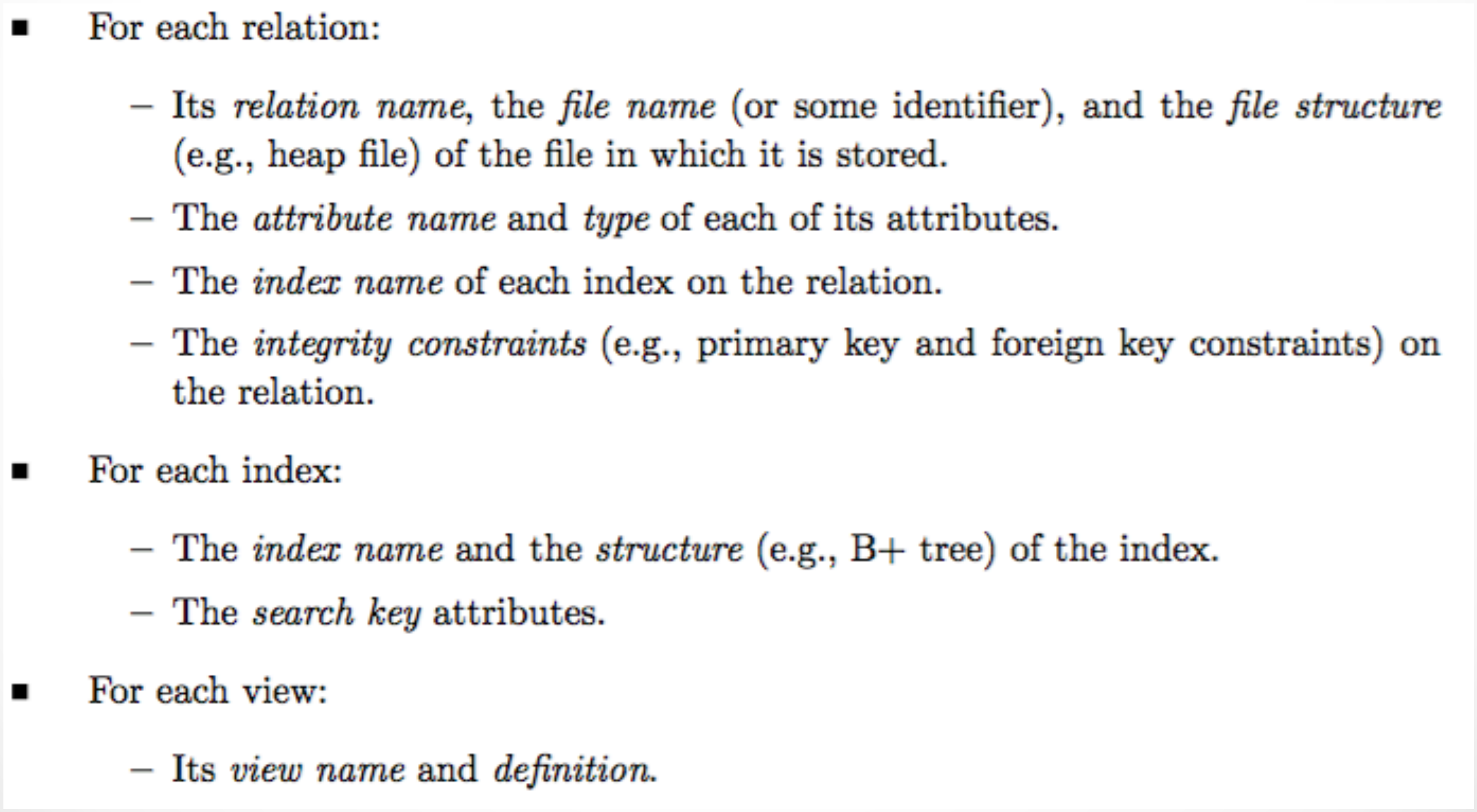
attr_name	*rel_name*	*type*	*position*
attr_name	Attribute_cat	string	1
rel_name	Attribute_cat	string	2
type	Attribute_cat	string	3
position	Attribute_cat	integer	4
sid	Students	string	1
name	Students	string	2
login	Students	string	3
age	Students	integer	4
gpa	Students	real	5
fid	Faculty	string	1
fname	Faculty	string	2
sal	Faculty	real	3

- information is stored in a collection of relations, maintained by the system, called the **catalog relations**
- also known as system catalog, the catalog, the data dictionary and is sometimes referred to as metadata

Query Optimization

System Catalog in Relational DBMS

information in the system catalog is used extensively for query optimization

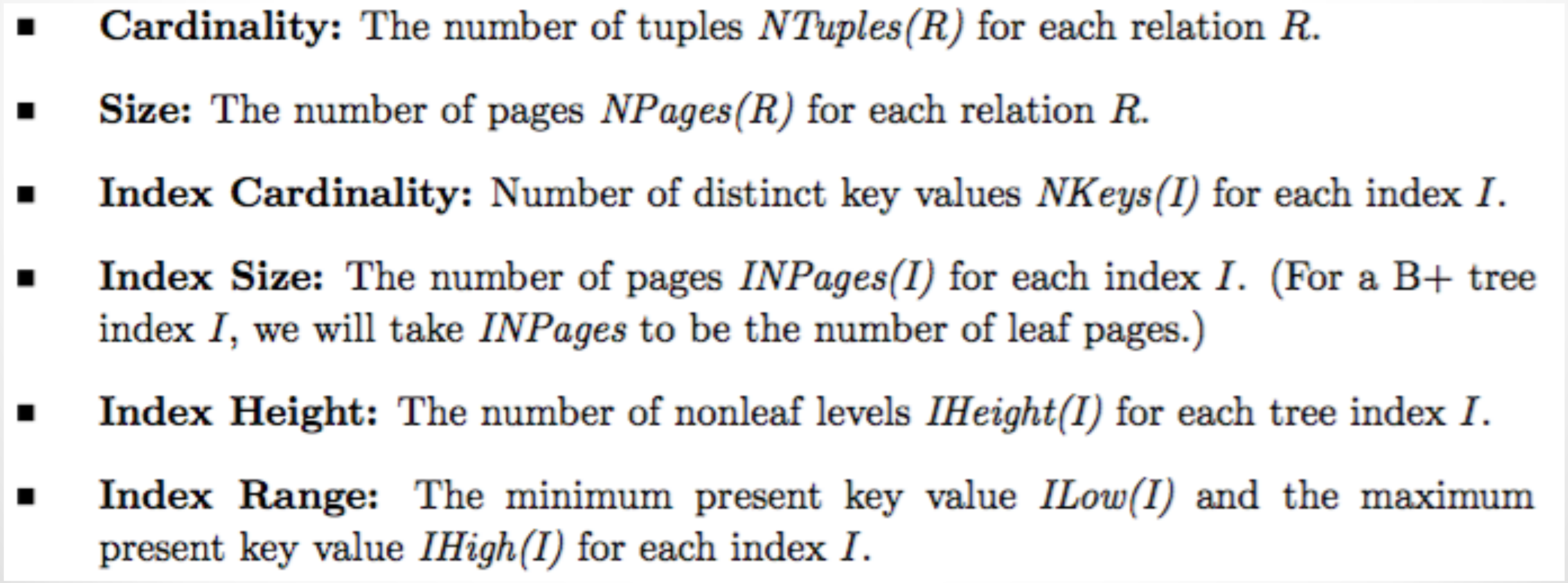


- For each relation:
  - Its *relation name*, the *file name* (or some identifier), and the *file structure* (e.g., heap file) of the file in which it is stored.
  - The *attribute name* and *type* of each of its attributes.
  - The *index name* of each index on the relation.
  - The *integrity constraints* (e.g., primary key and foreign key constraints) on the relation.
- For each index:
  - The *index name* and the *structure* (e.g., B+ tree) of the index.
  - The *search key* attributes.
- For each view:
  - Its *view name* and *definition*.

Query Optimization

System Catalog in Relational DBMS

information in the system catalog is used extensively for query optimization

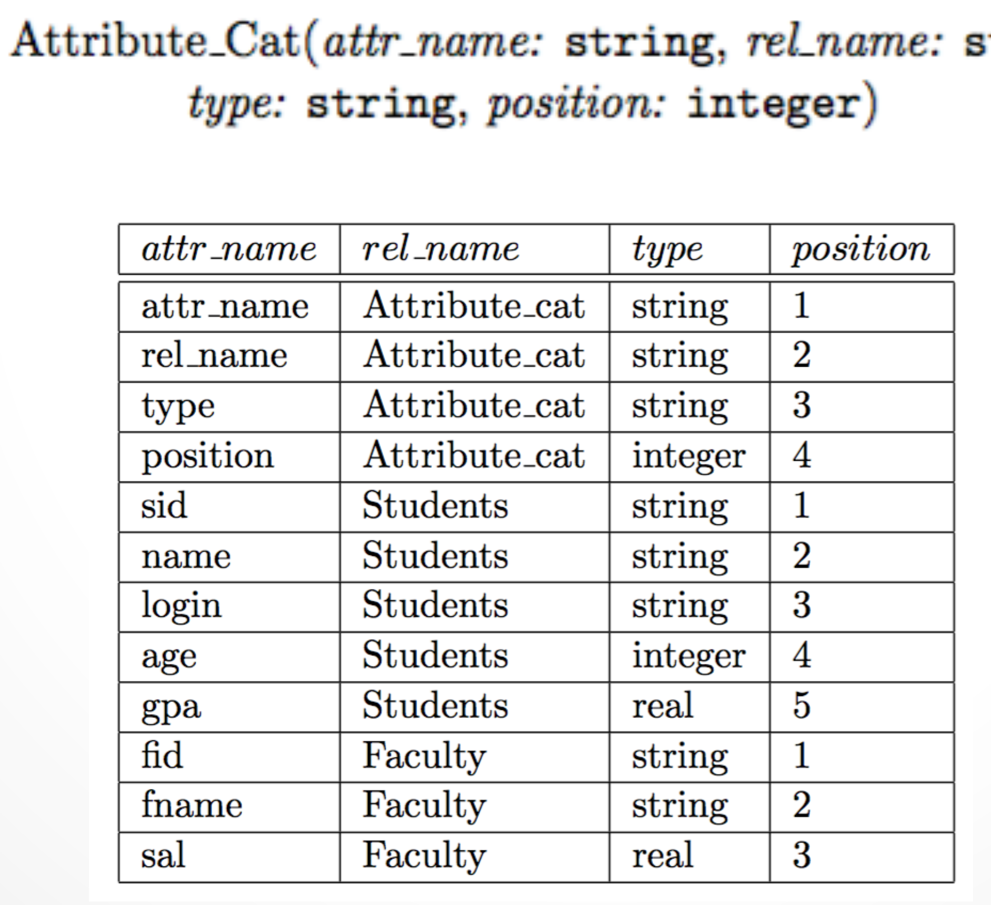


- **Cardinality:** The number of tuples *NTuples(R)* for each relation *R*.
- **Size:** The number of pages *NPages(R)* for each relation *R*.
- **Index Cardinality:** Number of distinct key values *NKeys(I)* for each index *I*.
- **Index Size:** The number of pages *INPages(I)* for each index *I*. (For a B+ tree index *I*, we will take *INPages* to be the number of leaf pages.)
- **Index Height:** The number of nonleaf levels *IHeight(I)* for each tree index *I*.
- **Index Range:** The minimum present key value *ILow(I)* and the maximum present key value *IHigh(I)* for each index *I*.

Query Optimization

System Catalog in Relational DBMS

system catalog is itself a collection of relations



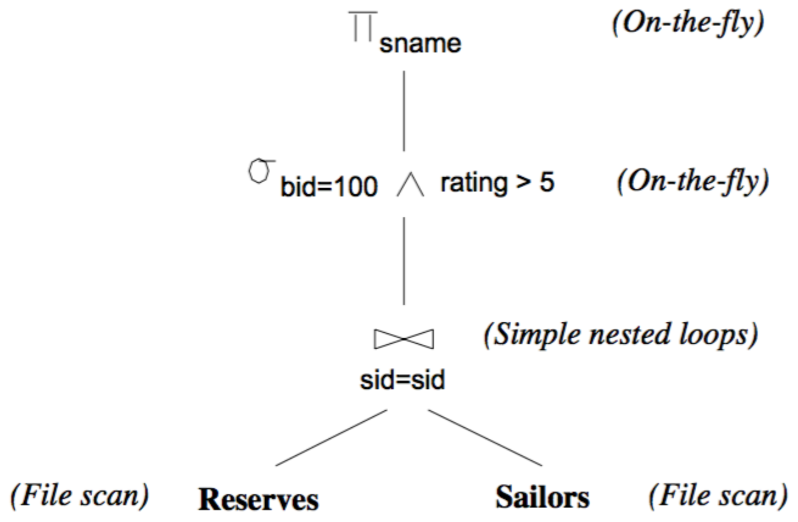
Attribute_Cat(*attr_name*: `string`, *rel_name*: `string`, *type*: `string`, *position*: `integer`)

attr_name	*rel_name*	*type*	*position*
attr_name	Attribute_cat	string	1
rel_name	Attribute_cat	string	2
type	Attribute_cat	string	3
position	Attribute_cat	integer	4
sid	Students	string	1
name	Students	string	2
login	Students	string	3
age	Students	integer	4
gpa	Students	real	5
fid	Faculty	string	1
fname	Faculty	string	2
sal	Faculty	real	3

Query Optimization

Alternative Plans

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$



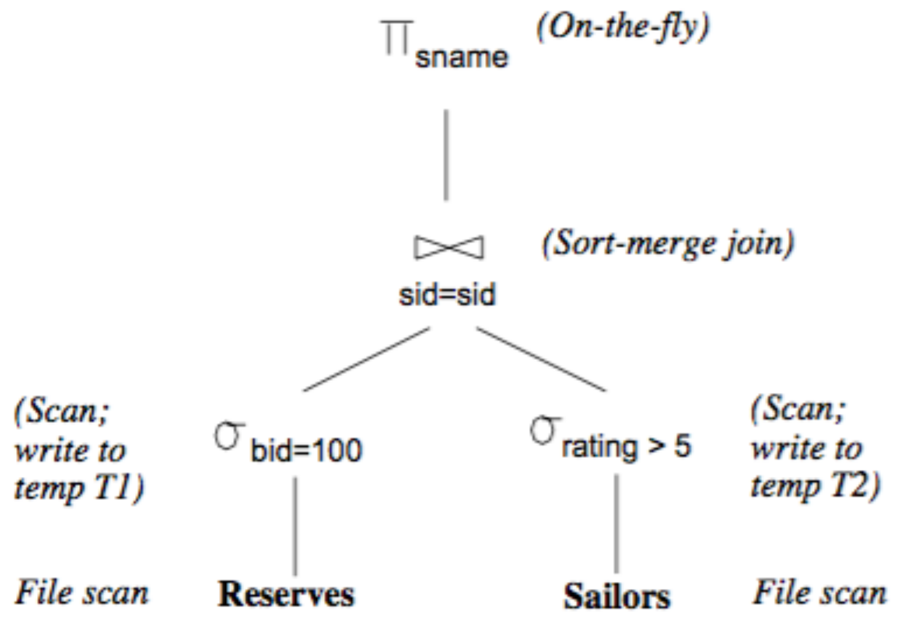
- cost of the join $1,000 + 1,000 * 500 = 501,000$ page I/Os
- selections and the projection are done on-the-fly and do not incur additional I/O

Query Optimization

Alternative Plans

Pushing Selections:

- apply selections early (pushed ahead of the join)
- selection bid=100 involves only the attributes of Reserves
- selection rating> 5 involves only attributes of Sailors

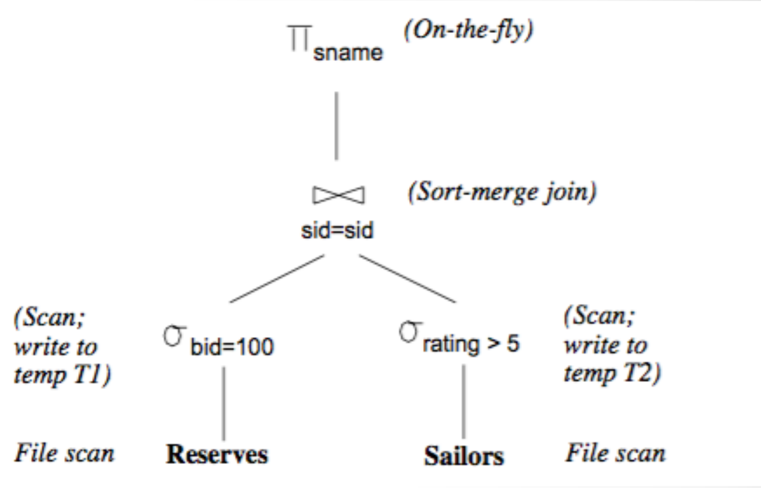


Query Optimization

Alternative Plans

Pushing Selections:

- assume that five buffer pages are available cost of applying bid=100 to Reserves is cost of scanning Reserves (1,000 pages) + cost of writing the result to a temporary relation (T1)



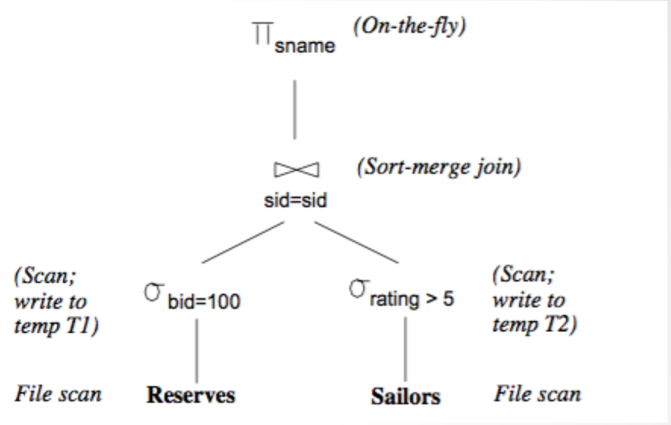
- assume that the number of pages in T1 is 10
- cost of applying rating> 5 to Sailors is cost of scanning Sailors (500 pages) + cost of writing out the result to a temporary relation (T2)
- Assume ratings are uniformly distributed over the range 1 to 10 and estimate the size of T2 as 250 pages
- sort-merge join of T1 and T2 -two relations are first completely sorted and then merged

Query Optimization

Alternative Plans

Pushing Selections:

- assume that five buffer pages are available
- 5 buffer pages are available, we can sort T1 (10 pages) in 2 passes



- 2 runs of 5 pages each are produced in the first pass and these are merged in the second pass
- For each pass, we read and write 10 pages and cost of sorting T1 is $2 * 2 * 10 = 40$ page I/Os
- 4 passes to sort T2 (250 pages), cost is $2 * 4 * 250 = 2{,}000$ page I/Os
- To merge of sorted T1andT2, scan these relations, cost is10 + 250 = 260
- final projection is done on-the-fly , cost is ignored
- Final cost is sum of the cost of the selection (1,000+10+500+250 = 1,760) and cost of the join (40+2,000+260 = 2,300) which is 4,060 page I/Os (reduce by different join?)

Query Optimization

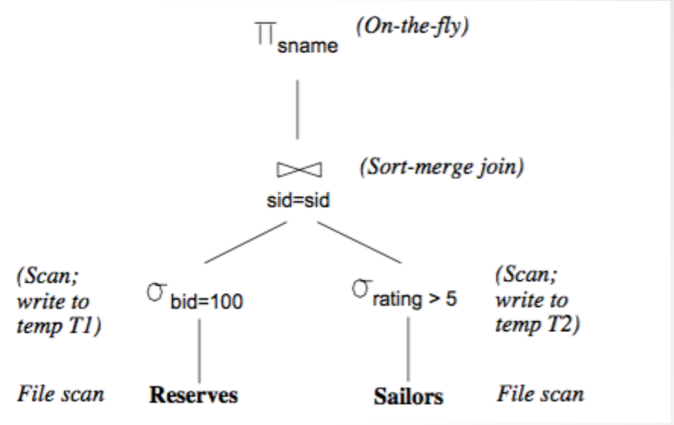
Alternative Plans

Pushing Projections:

- only the sid attribute of T1 and

sid and sname attributes of T2

are really required



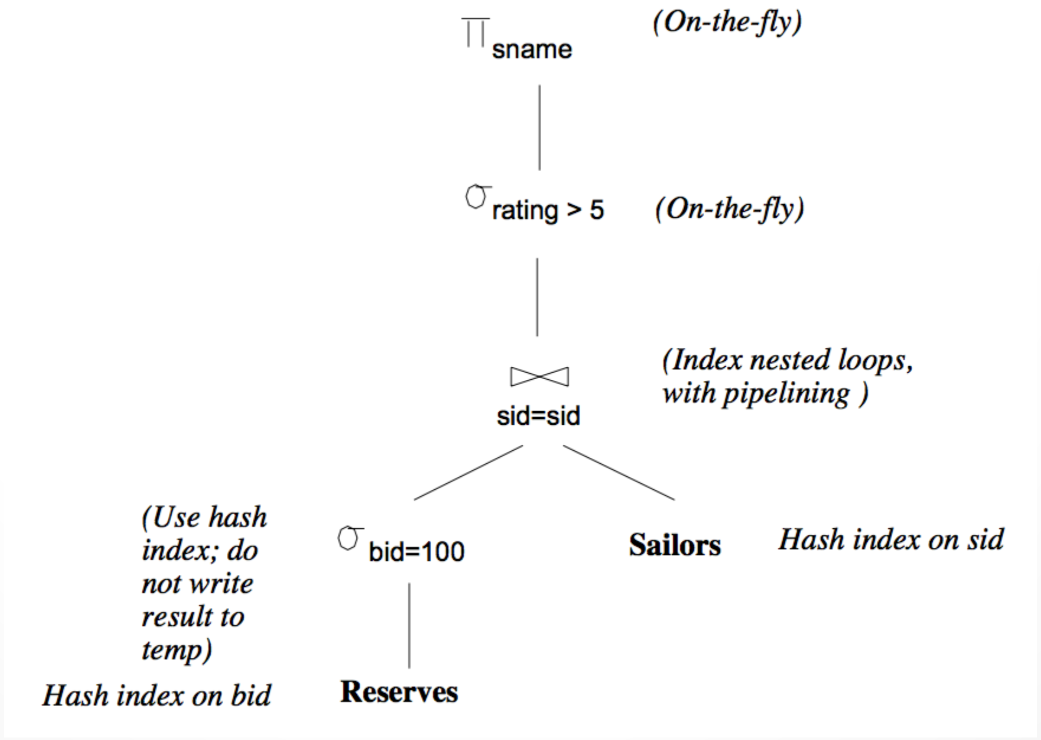
- as we scan Reserves and Sailors to do the selections, we can eliminate unwanted columns
- on-the-fly projection reduces the sizes of the temporary relations T1 and T2
- reduction in the size of T1 is significant - only an integer field is retained (T1 will now fit within three buffer pages!)
- cost of the join step drops to under 250 page I/Os, and the total cost of the plan drops to about 2,000 I/O

Query Optimization

Alternative Plans

Using Indexes:

- clustered static hash index on the bid field of Reserves
- hash index on the sid field of Sailors

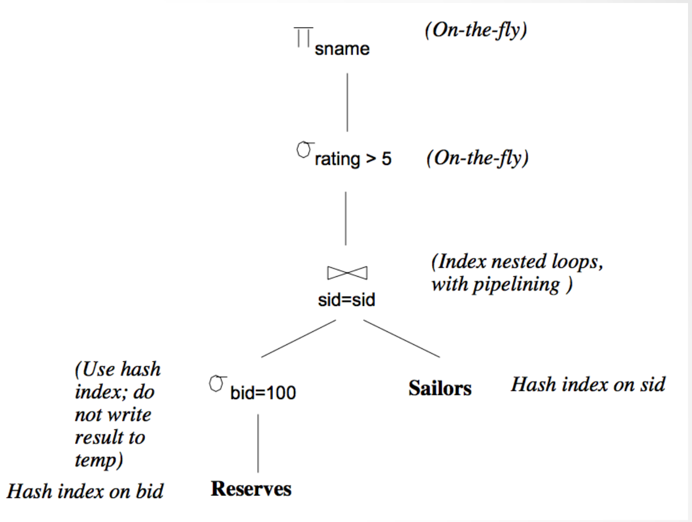


Query Optimization

Alternative Plans

Using Indexes:

- selection bid=100 is performed on Reserves using hash index to retrieve only matching tuples
- we can estimate the number of selected tuples to be 100,000/100 = 1,000
- cost is 10 page I/Os (index on bid is clustered so1,000 tuples appear consecutively within the same bucket)
- for each selected tuple, we retrieve matching Sailors tuples using the hash index on the sid field
- selected Reserves tuples are not materialized
- the join is pipelined



Query Optimization

Alternative Plans

Using Indexes:

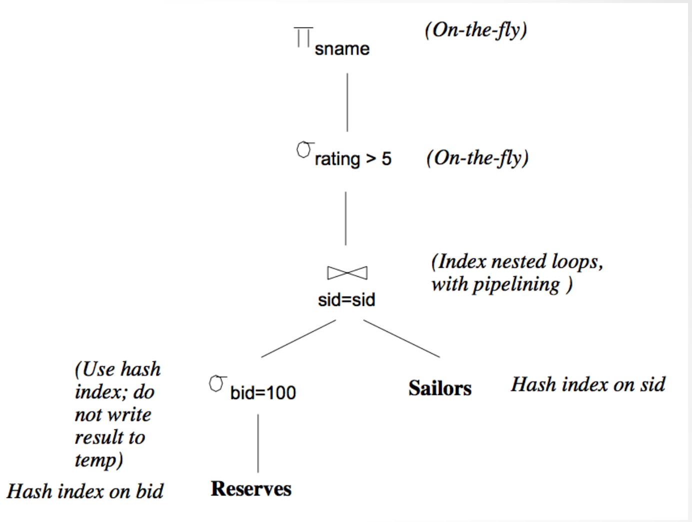
- each tuple in the result of the join,

perform the selection rating>5 and projection of sname on-the-fly

- for a hash index, 1.2 page I/Os

(on average) is a good estimate of the cost for retrieving a data entry

- selection of Reserves tuples costs 10 I/Os
- for 1,000 such tuples the cost of finding the matching Sailors tuple is 1.2 I/Os so cost of the join = 1,200 I/Os
- selections and projections are performed on-the-fly
- total cost of the plan is 1,210 I/Os.



Query Optimization

Alternative Plans

Using Both:

```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
        AND R.bid = 100 AND S.rating > 5
        AND R.day = '8/9/94'
```

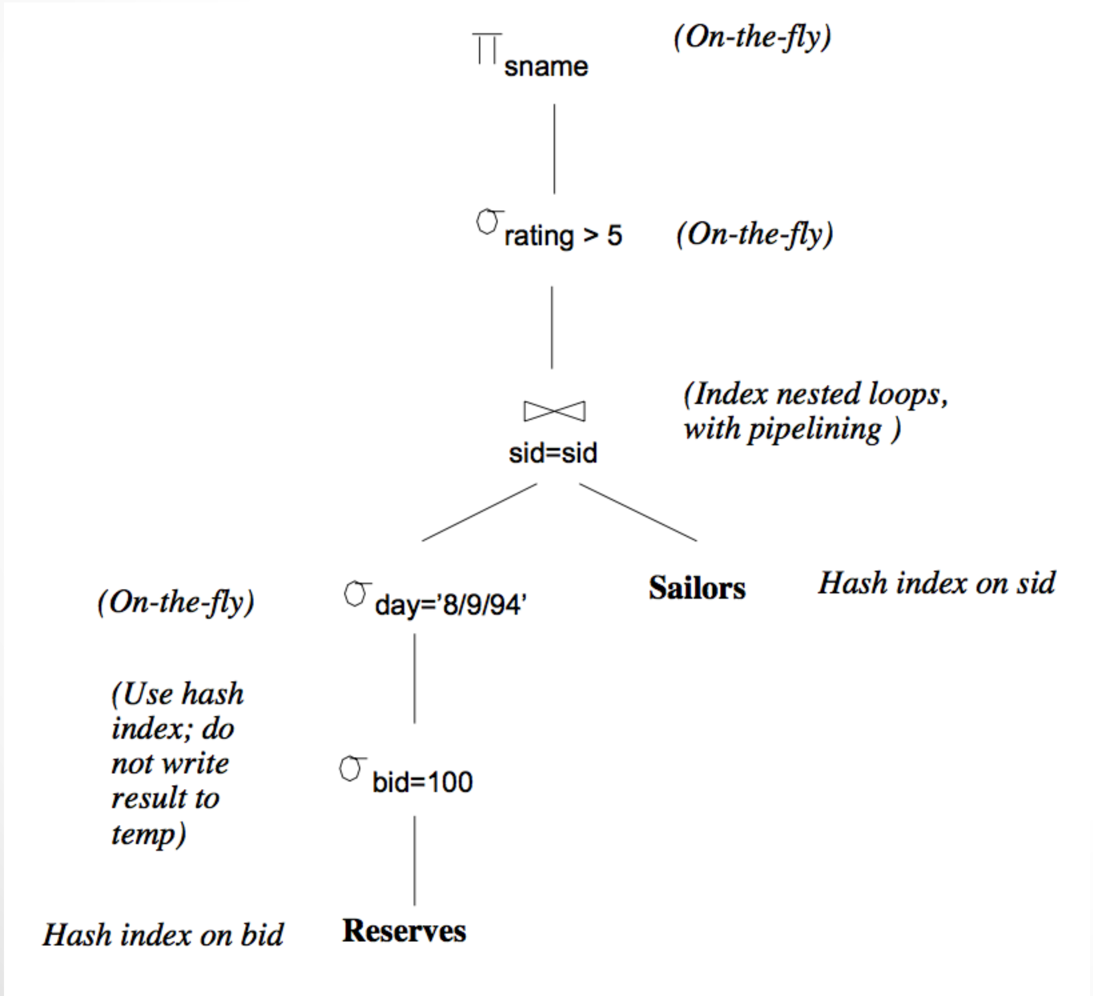
- selection day='8/9/94' is applied on-the-fly to the result of the selection bid=100 on the Reserves relation

Query Optimization

Alternative Plans

Using Both:

```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
        AND R.bid = 100 AND S.rating > 5
        AND R.day = '8/9/94'
```



Query Optimization

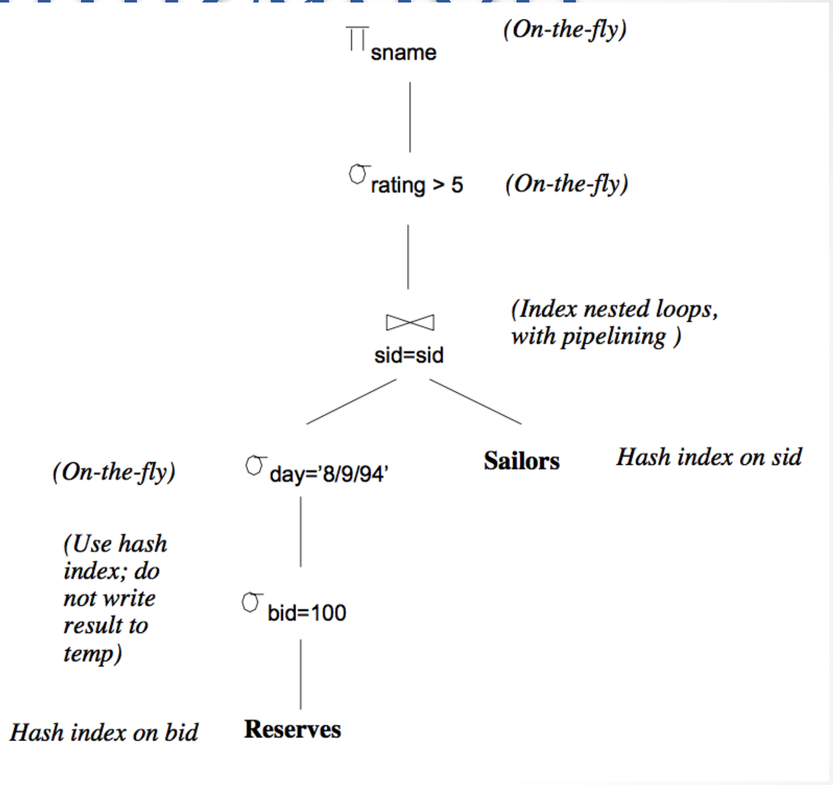
Alternative Plans

Using Both:

- suppose that bid and day form a key for Reserves
- selection bid=100 costs 10 page I/Os

selection day='8/9/94' is applied on-the-fly (eliminating all but (at most) one Reserves tuple)

- at most one matching Sailors tuple is retrieved in 1.2 I/Os
- selection on rating and the projection on sname are then applied on-the-fly (no additional cost)
- total cost 11 I/Os



SUMMARY

- goal of query optimization is usually to avoid the worst evaluation plans and find a good plan
- **query evaluation plan** is a tree with relational operators at the intermediate nodes and relations at the leaf nodes
- results of one operator can be **pipelined** into another operator without **materializin**g the intermediate result
- if the input tuples to a unary operator are pipelined, this operator is said to be applied **on-the-fly**
- uniform **iterator** interface with functions **open**, **get next**, and **close**
- DBMS maintains **metadata** information about the data in a special set of relations called the **catalog**
- alternative plans can differ substantially in their overall cost