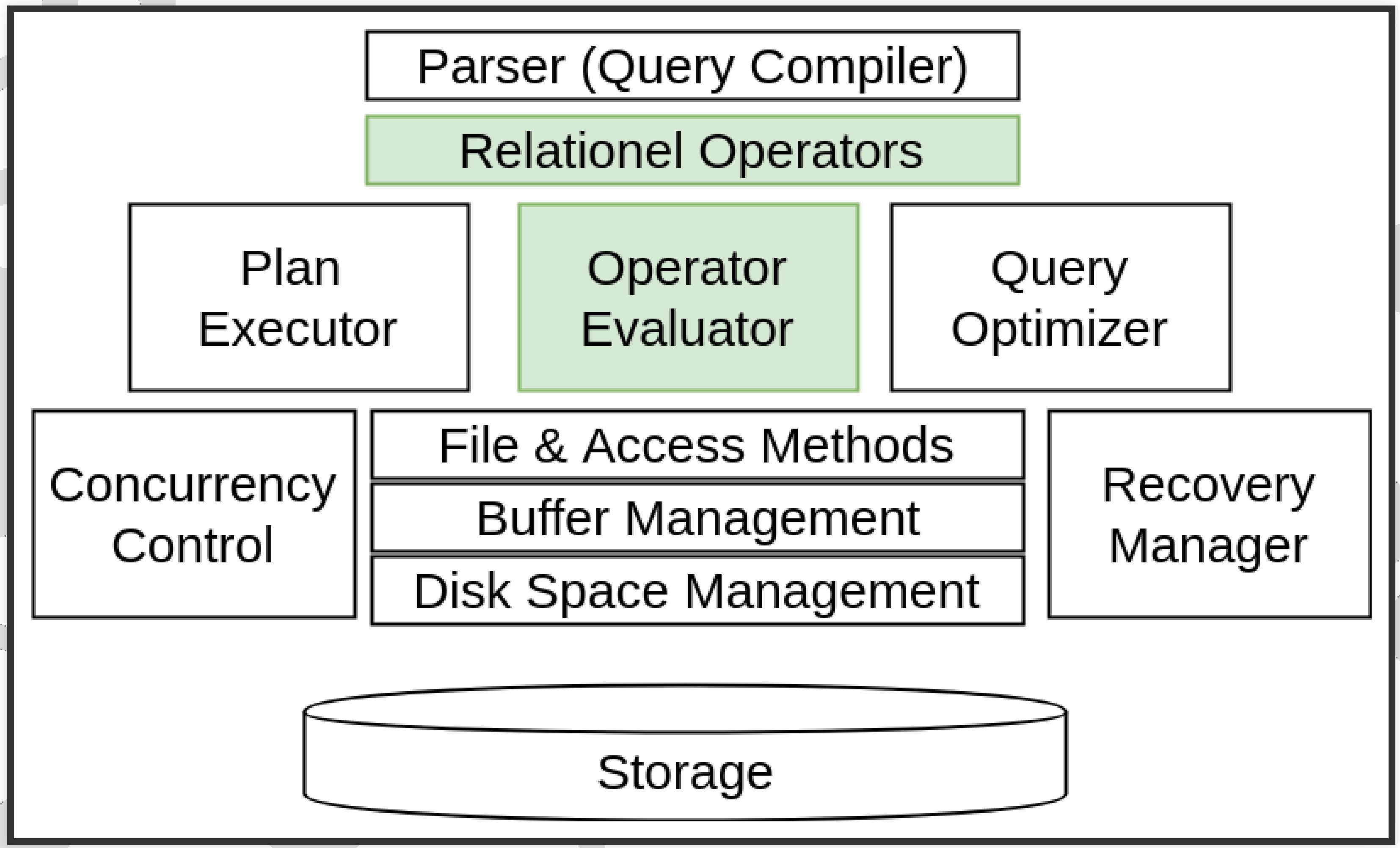
EVALUATING AN SQL QUERY

CHAPTER 12 AND 14

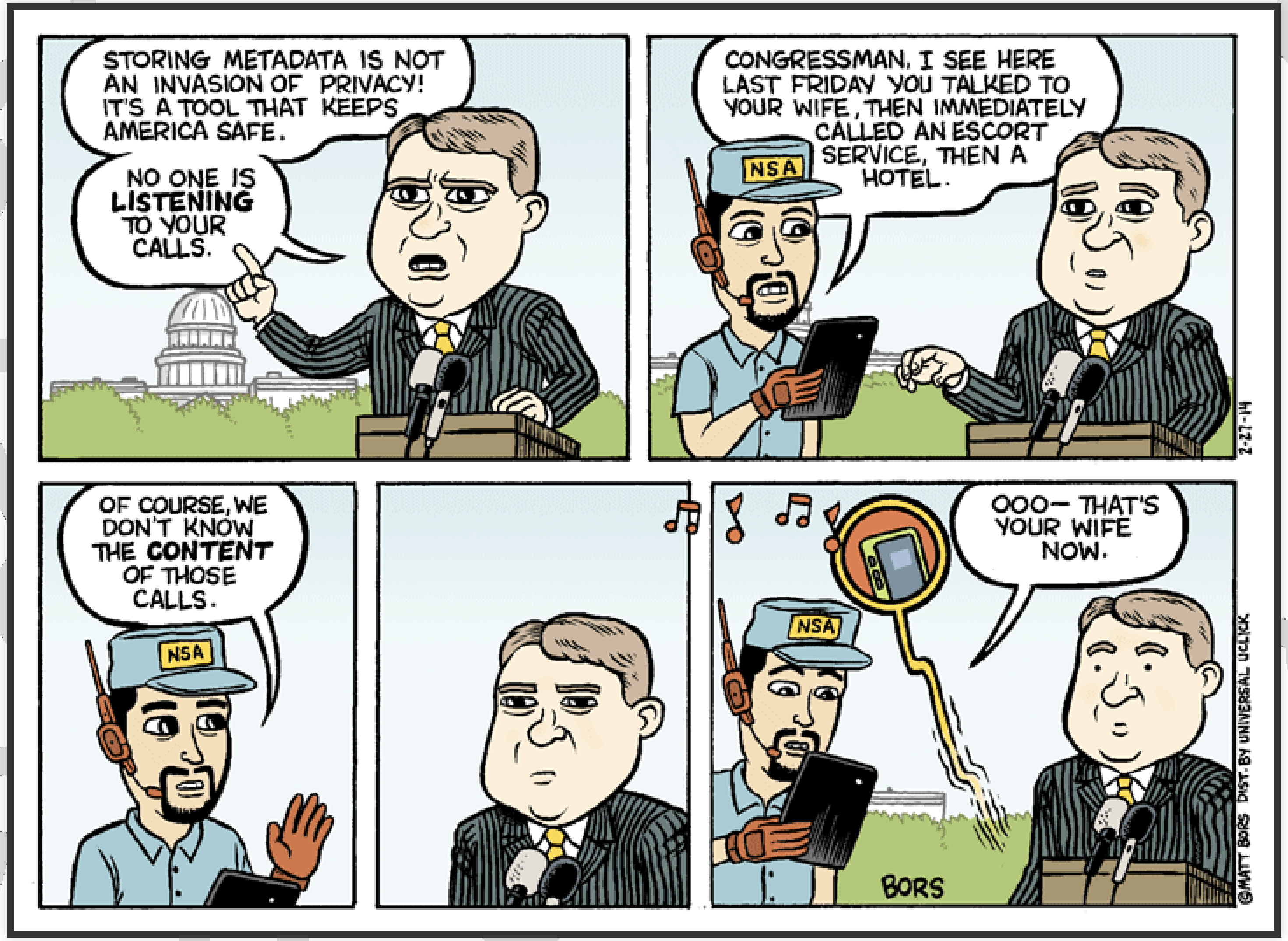
Query Evaluation and Relational Operators



GOALS

- System Catalog
- Algorithms for algebra operations
- Query evaluation plans and representations
- Intro to query optimization

THE SYSTEM CATALOG



INFORMATION IN THE CATALOG

For each table:

- *Table Name*, *file name* and *file structure*
- *Attribute name* and *type* of all attributes
- *index name* of each index
- *integrity constraints* (primary/foreign key constraints)

INFORMATION IN THE CATALOG

For each index:

- *Index name* and its structure
- *Search key attribute*

For each view:

- *View name*
- *Definition*

INFORMATION IN THE CATALOG

Some general statistics:

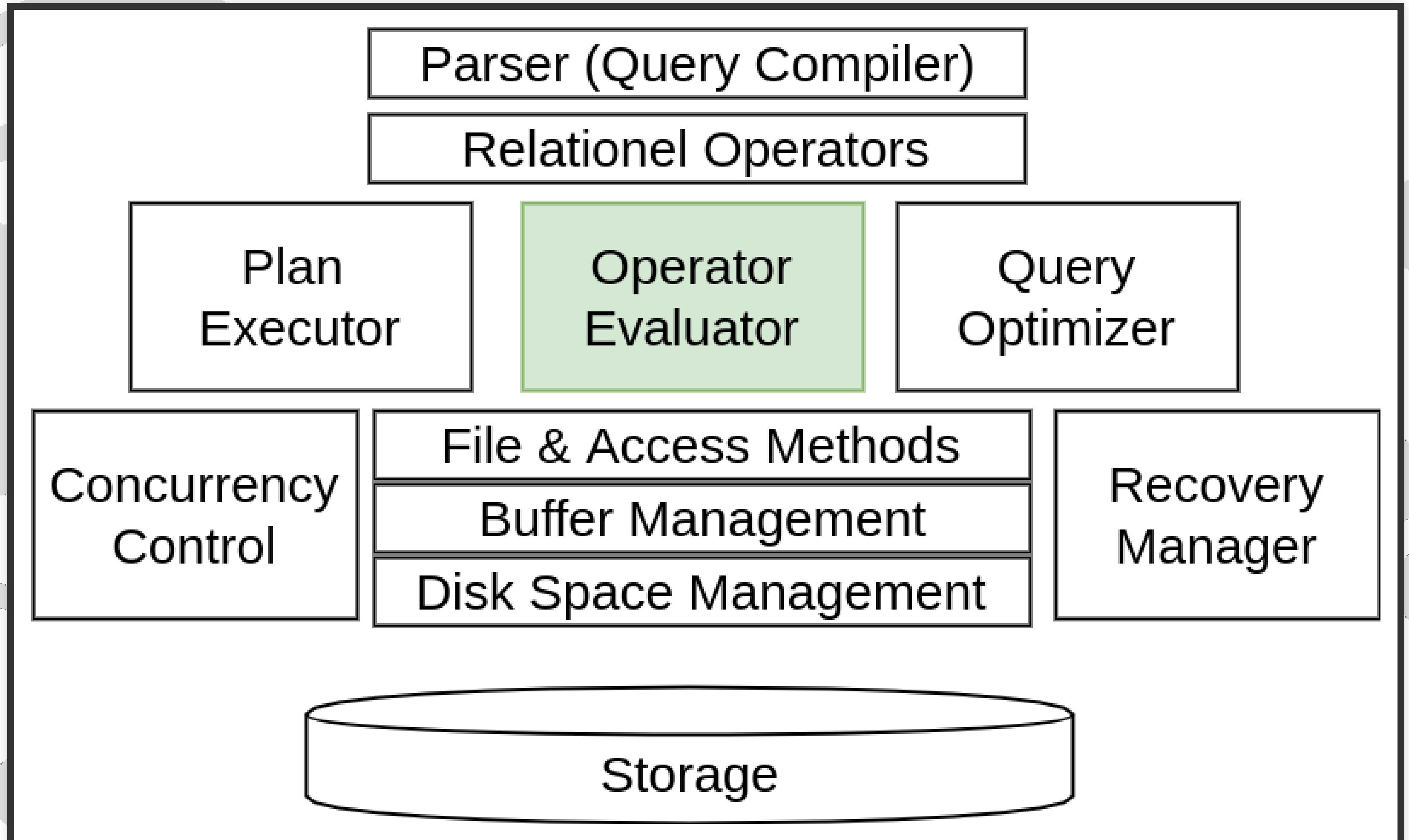
- *Cardinality* - number of tuples in each releation
- *Size* - number of pages for each relation
- *Index Cardinality* - Number of distinct keys in index
- *Index size* - number of pages used for index
- *Index Height* - number of non-leaf levels
- *Index range* - Min and Max key

HOW CATALOGS ARE STORED

Catalogs are themselves stored as a relation

attr_name	rel_name	type	position
attr_name	Att_Catalog	String	1
rel_name	Att_Catalog	String	2
type	Att_Catalog	String	3
position	Att_Catalog	int	4
sid	Sailor	int	1
…	…	…	…

INTRO TO OPERATOR EVALUATION (AND QUERY OPTIMIZATION)



OPERATORS

- *Selection* (σ) Select a subset of rows.
- *Projection* (π) Remove unwanted columns.
- *Join* ($\bowtie$) Combine two relations.
- *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
- *Intersection* ($\cap$) Tuples in both reln. 1 and reln. 2.
- *Union* ($\cup$) Tuples in rel 1 and/or in reln 2.

EXAMPLE QUERY SCHEMA

Sailors

Sailors(sid: integer, sname: string, rating: integer, age: real)

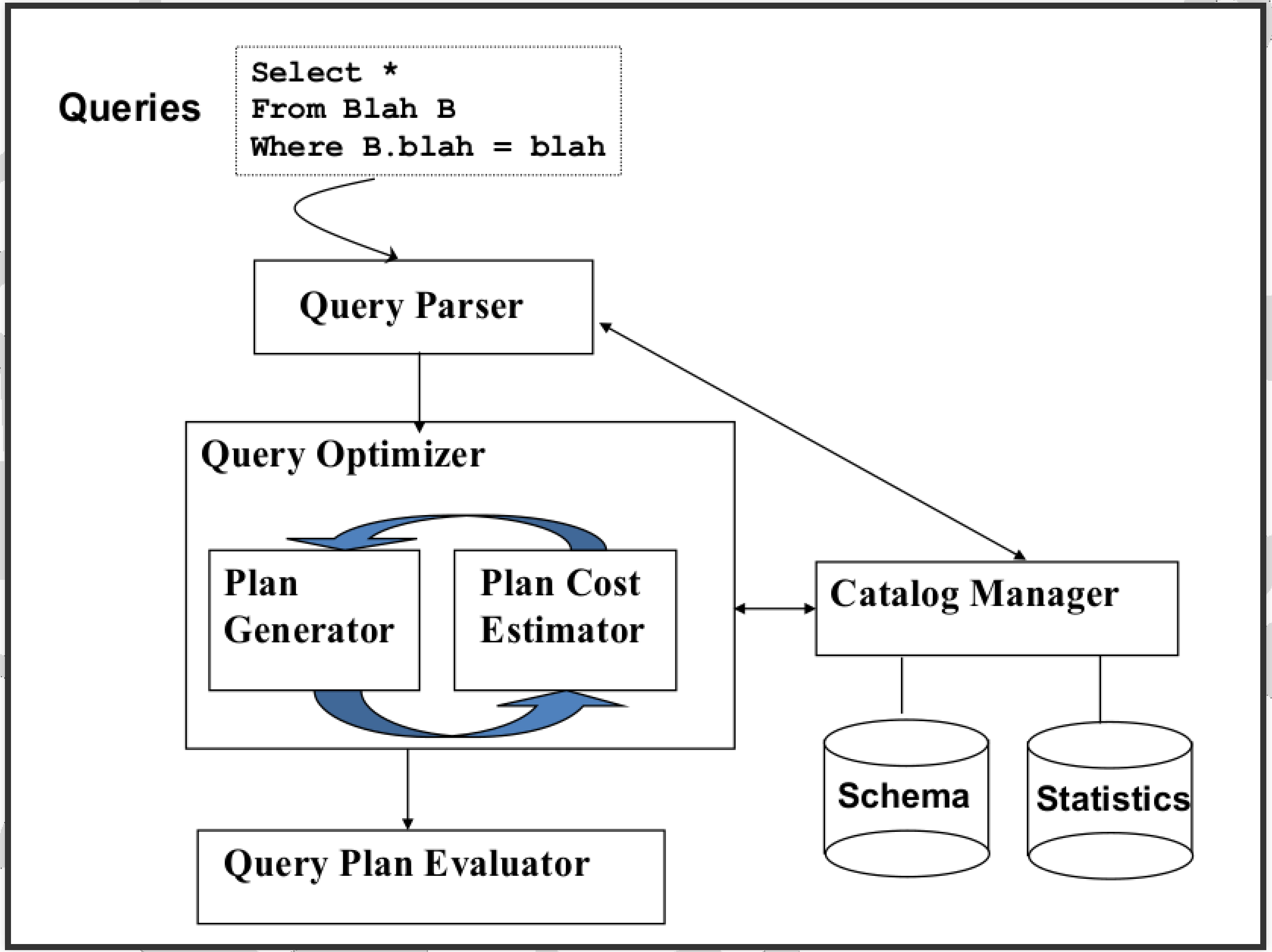
- Each tuple is 50 bytes long, 80 tuples per page, 500 pages

Reserves

Reserves(sid:integer, bid:integer, day: date, rname: string)

- Each tuple is 40 bytes long, 100 tuples per page, 1000 pages

QUERY SUB-SYSTEM



QUERY BLOCKS: UNITS OF OPTIMIZATION

- An SQL query is parsed into a collection of query blocks
  - optimize one block at a time.

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.age IN
    (SELECT  MAX (S2.age)
     FROM  Sailors S2
     GROUP BY  S2.rating)
```

Outer block   *Nested block*

- Nested blocks are usually treated as calls to a subroutine,
  - called once per outer tuple.

TRANSLATING SQL TO RELATIONAL ALGEBRA

```
SELECT S.sid, MIN (R.day)
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = "red"
GROUP BY S.sid
HAVING COUNT (*) >= 2
```

For each sailor with at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.

TRANSLATING SQL TO RELATIONAL ALGEBRA

```
SELECT S.sid, MIN (R.day)
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = "red"
GROUP BY S.sid
HAVING COUNT (*) >= 2
```

$$\pi_{\text{S.sid, MIN(R.day)}}\Big(\text{HAVING}_{\text{COUNT(*)>2}}\big(\text{GROUP BY}_{\text{S.Sid}}\big(\sigma_{\text{B.color = "red"}}\big(\text{Sailors} \bowtie_{\text{S.sid = R.sid}} \text{Reserves} \bowtie_{\text{R.bid = B.bid}} \text{Boats}\big)\big)\big)\Big)$$

TRANSLATING SQL TO RELATIONAL ALGEBRA

$$\pi_{\text{S.sid, MIN(R.day)}}$$

$$|$$

$$\text{HAVING}_{\text{COUNT(*)>2}}$$

$$|$$

$$\text{GROUP BY}_{\text{S.Sid}}$$

$$|$$

$$\sigma_{\text{B.color = "red"}}$$

$$|$$

$$\text{Sailors} \bowtie_{\text{S.sid = R.sid}} \text{Reserves} \bowtie_{\text{R.bid = B.bid}} \text{Boats}$$

THE ITERATOR INTERFACE

Relational operators are implemented as `iterators`:

```
interface iterator {
    void init();
    tuple next();
    void close();
    iterator &inputs[];
    // additional state goes here
}
```

Note:

- Edges in the graph are specified by inputs (max 2, usually)
- Any iterator can be input to any other!

EXAMPLE - SORT

```
class Sort extends iterator {
    void init();
    tuple next();
    void close();
    iterator &inputs[1];
    int numberOfRuns;
    DiskBlock runs[];
    RID nextRID[];
}
```

EXAMPLE - SORT

`init():`

- generate the sorted runs on disk (passes 0 to n-1)
- Allocate runs[] array and fill in with disk pointers.
- Initialize numberOfRuns
- Allocate nextRID array and initialize to first RID of each run

EXAMPLE - SORT

`next():`

- nextRID array tells us where we're "up to" in each run
- find the next tuple to return based on nextRID array
- advance the corresponding nextRID entry
- return tuple (or EOF — "End of Fun" — if no tuples remain)

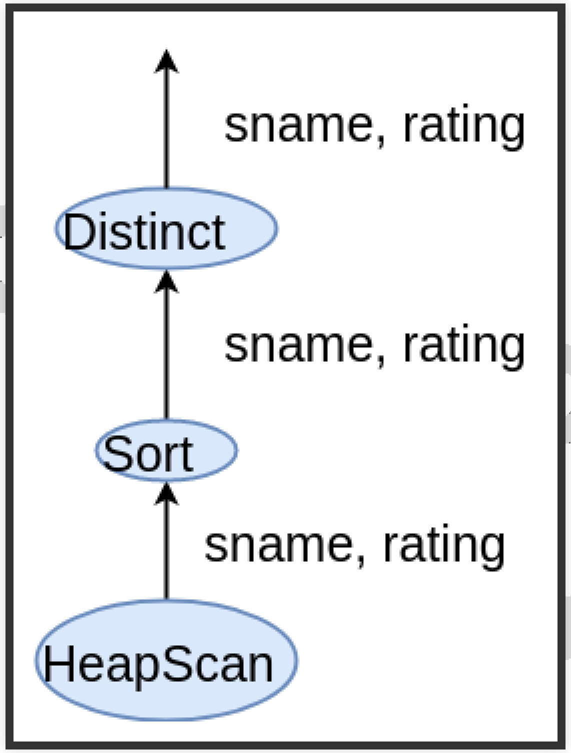
EXAMPLE - SORT

`close():`

- deallocate the runs and nextRID arrays

QUERY EXECUTION FRAMEWORK

```
SELECT DISTINCT sname, rating
FROM Sailors
```



THREE COMMON TECHNIQUES

- Indexing
- Iteration
- Partitioning

ACCESS PATHS

An **access path** is a way of retrieving tuples from a table and consists of either

1. File scan or
2. Index plus matching selection condition

CONJUNCTIVE NORMAL FORM (CNF)

For a selection condition: *attr* **op** *value* where **op** is one of **<, < =, =, ! =, >, >=** it is called a **conjunct**

Conjunctive Normal Form (CNF) is where all the conjuncts are `AND`ed together

MATCHING A SELECTION CONDITION

An index **matches** a selection condition if it can be used to retrieve just the tuples that satisfy the condition.

- A hash index matches a CNF selection if there is a term of the form `attr = value` in the selection for each attribute in the index's search key
- A tree index **matches** a CNF selection if there is a term of the form `attr op value` for each attribute in a *prefix* of the index's search key

An index can match some subset of the conjuncts in a selection condition in CNF. Those are the **primary conjuncts**

MATCHING A SELECTION CONDITION

query: day < 8/9/94 AND bid=5 AND sid=3

1. Can B+tree index on day be used?
2. How about a B+tree on <rname,day>?
3. How about a B+tree on <day, rname>?
4. How about a hash index on <bid, sid>?
5. How about a Hash index on <day, rname>?

SELECTIVITY OF ACCESS PATHS

> The **selectivity** of an access path is the number of pages retrieved (index pages plus data pages) if we use this access path to retrieve all desired tuples.

If a table contains an index that matches a given selection there are at least 2 access paths:

1. The index
2. Scan the whole file

Sometimes we can scan the index only

MOST SELECTIVE ACCESS PATHS

💡 The **most selective access path** is the one that retrieves the fewest pages.

Using this will minimize the cost of data retrieval

REDUCTION FACTOR

The fraction of tuples that satisfy a given conjunct is called **reduction factor**

When there are several primary conjuncts, the fraction that satisfy them all can be estimated by multiplying their reduction factors

ALGORITHM FOR RELATIONAL OPERATIONS

We considered Sort last week - lets dive into all the others

SELECTION

The selection operation is a simple retrieval of tuple from a table and implementation follows the lines of access path.

If we have $\sigma_{R.attr \text{ OP } value}(R)$

```
SELECT *
FROM Reserves R
WHERE R.rname < 'C%'
```

SELECTION - PERFORMANCE

If **no appropriate index exists**: Must scan the whole relation

cost = |R|

For “reserves” = 1000 I/Os.

SELECTION - PERFORMANCE

With **index on selection attribute**:

1. Use index to find qualifying data entries
2. Retrieve corresponding data records

Total cost = cost of step 1 + cost of step 2

For “Reserves”, if selectivity = 10% (100 pages, 10000 tuples):

- If clustered index, cost is a little over *100 I/Os*
- If unclustered, could be *up to 10000 I/Os!* … unless …

REFINEMENT FOR UNCLUSTERED INDEXES

1. Find qualifying data entries.
2. Sort the rids of the data records to be retrieved.
3. Fetch rids in order.

Each data page is looked at just once

1. though # of such pages likely to be higher than with clustering

SELECTION

Best performance depends on:

- What indexes available
- Expected size of result

Size of result: **(size of R) * selectivity**

Rule of thumb: It is probably cheaper to simply scan the entire table instead of using an unclustered index if over 5% of the tuples are to be retrieved

GENERAL SELECTIONS - APPROACH 1

1. Find the **most selective access path**
2. Retrieve tuples using it
3. Apply any remaining terms that don’t **match** the index

GENERAL SELECTIONS - APPROACH 1

query: **day < 8/9/94 AND bid=5 AND sid=3**

Some options:

- B+tree index on day; check bid=5 and sid=3 afterward.
- hash index on <bid, sid>; check day < 8/9/94 afterward.

GENERAL SELECTIONS - APPROACH 2

use 2 or more matching indexes.

1. From each index, get set of rids
2. Compute intersection of rid sets
3. Retrieve records for rids in intersection
4. Apply any remaining terms

GENERAL SELECTIONS - APPROACH 2

query: **day < 8/9/94 AND bid=5 AND sid=3**

Suppose we have an index on day, and another index on sid.

- Get rids of records satisfying day<8/9/94.
- Also get rids of records satisfying sid=3.
- Find intersection, then retrieve records
- Then check bid=5.

PROJECTION

- Drop certain fields of the input - quite easy

PROJECTION

If `DISTINCT` is not in the `SELECT` clause of the SQL, it is trivial to drop the columns of the output that is not needed.

The issue is removing duplicates

```
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```

Typically, we use a partitioning technique.

PROJECTION USING SORTING

1. Scan R, extract only the needed attributes
2. Sort the resulting set
3. Remove adjacent duplicates

Cost:

Reserves with size ratio 0.25 = 250 pages and there are 20 buffer pages

With 20 buffer pages can sort in 2 passes, so:

Total cost = 1000 + 250 + 2 * 2 * 250 + 250 = 2500 I/Os

PROJECTION - IMPROVED

Modify the external sort algorithm:

- Modify Pass 0 to eliminate unwanted fields.
- Modify Passes 1+ to eliminate duplicates.

Cost:

Reserves with size ratio 0.25 = 250 pages.

With 20 buffer pages can sort in 2 passes

1. Read 1000 pages
2. Write 250 (in runs of 40 pages each)
3. Read and merge runs

Total cost = 1000 + 250 + 250 = 1500 I/Os.

OTHER PROJECTION TRICKS

If a B+Tree index search key prefix has all wanted attrs:

- **Scan index in order**
- Compare adjacent tuples on the fly (no sorting required!)

If an index search key contains all wanted attrs:

- **Do index-only scan**
- Apply projection techniques to data entries (much smaller!)

PROJECTION BASED ON HASHING

Idea:

- Many of the things we use sort for don't exploit the order of the sorted data
  - removing duplicates in DISTINCT
  - finding matches in JOIN

Often good enough to match all tuples with equal values

Hashing does this!

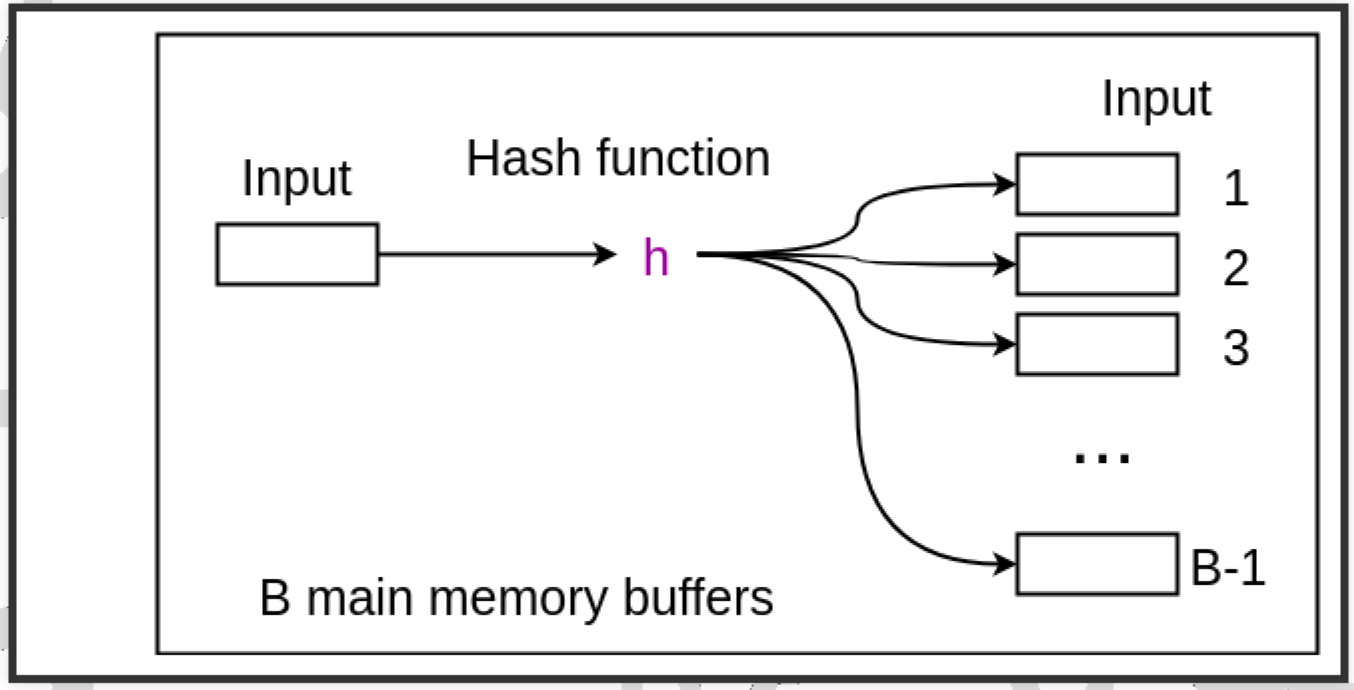
- And may be cheaper than sorting! (Hmmm...!)

PROJECTION BASED ON HASHING

Duplicate Elimination

1. Apply hash function
2. Look for duplicates in the corresponding bucket
3. If the input buffer is empty, then read in a page and goto 1.

DUPLICATE ELIMINATION USING HASHING



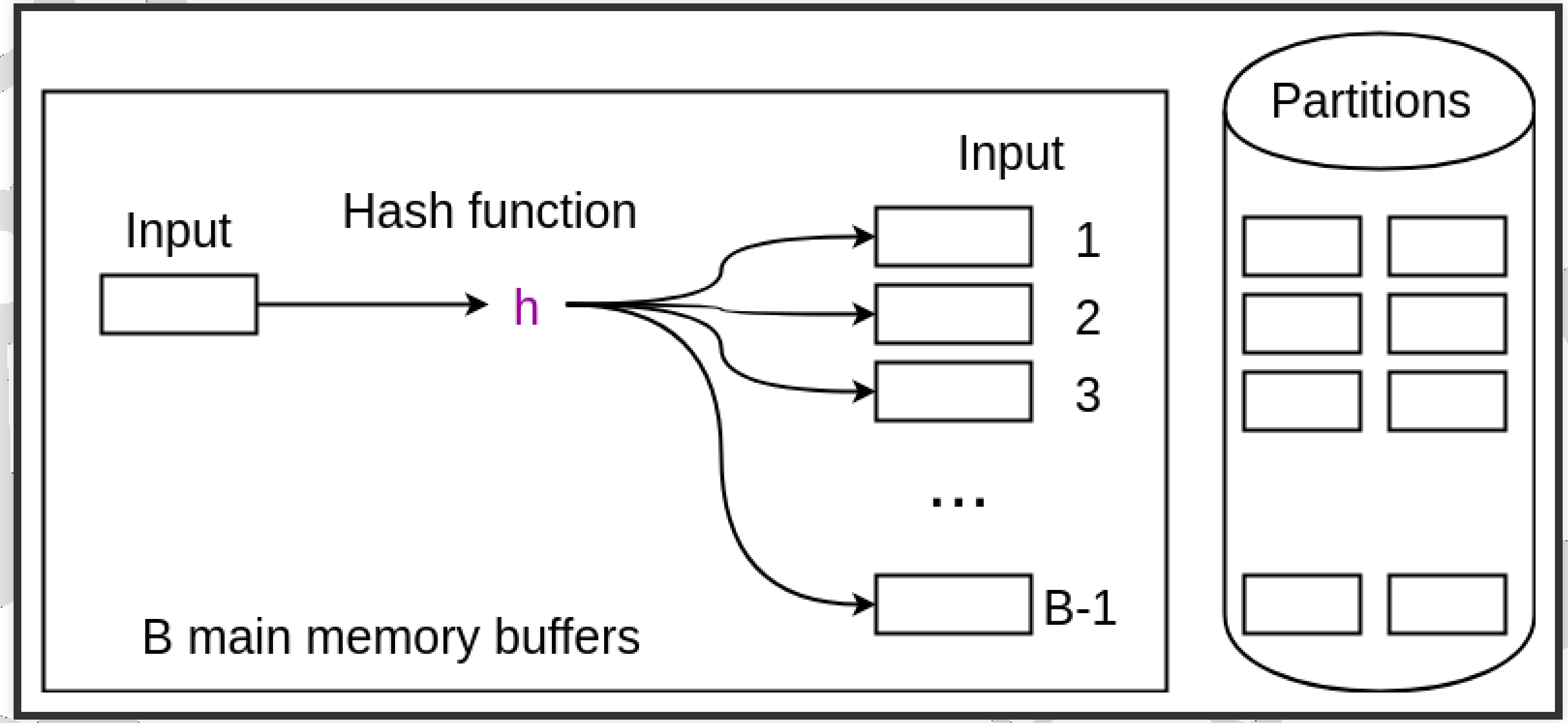
DUPLICATE ELIMINATION USING HASHING

Two phases:

1. **Partition:** use a hash function h to split tuples into partitions on disk.
    - Key property: all matches live in the same partition.
2. **ReHash:** for each partition on disk, build a main-memory hash table using a hash function h2

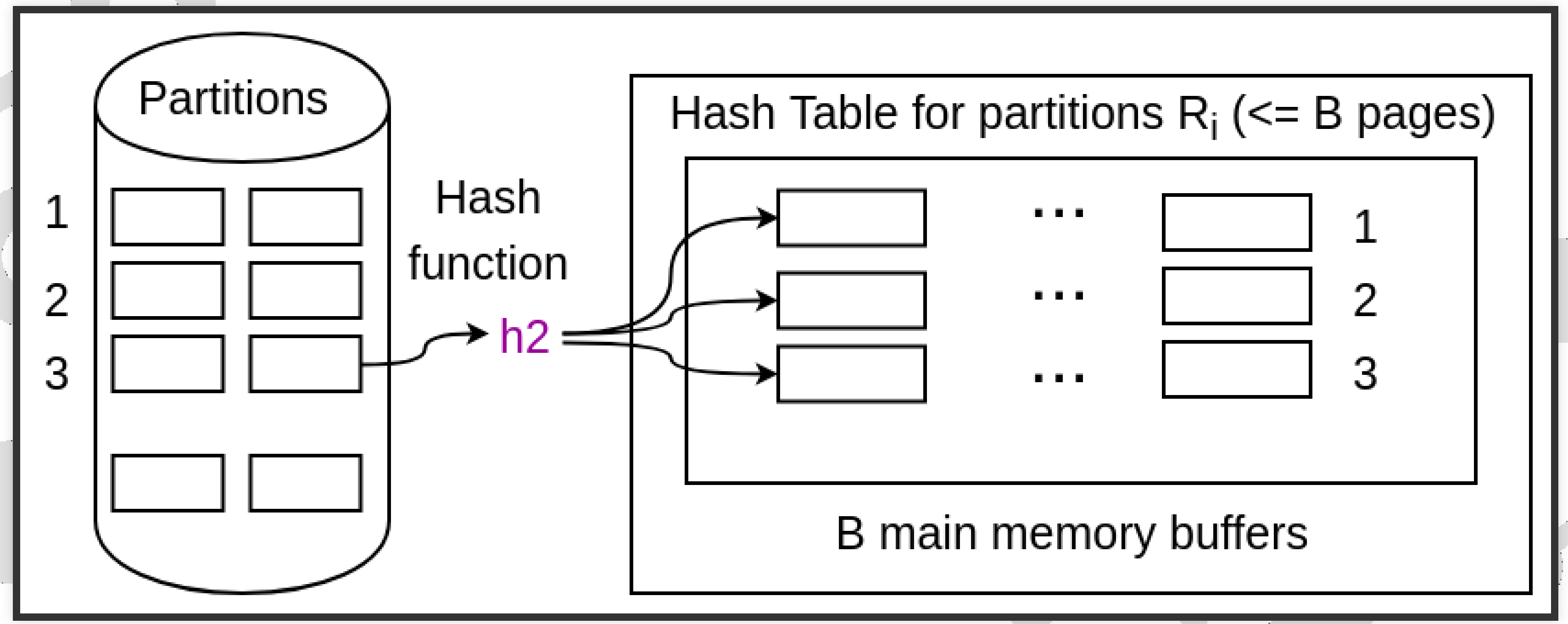
DUPLICATE ELIMINATION USING HASHING

Partition

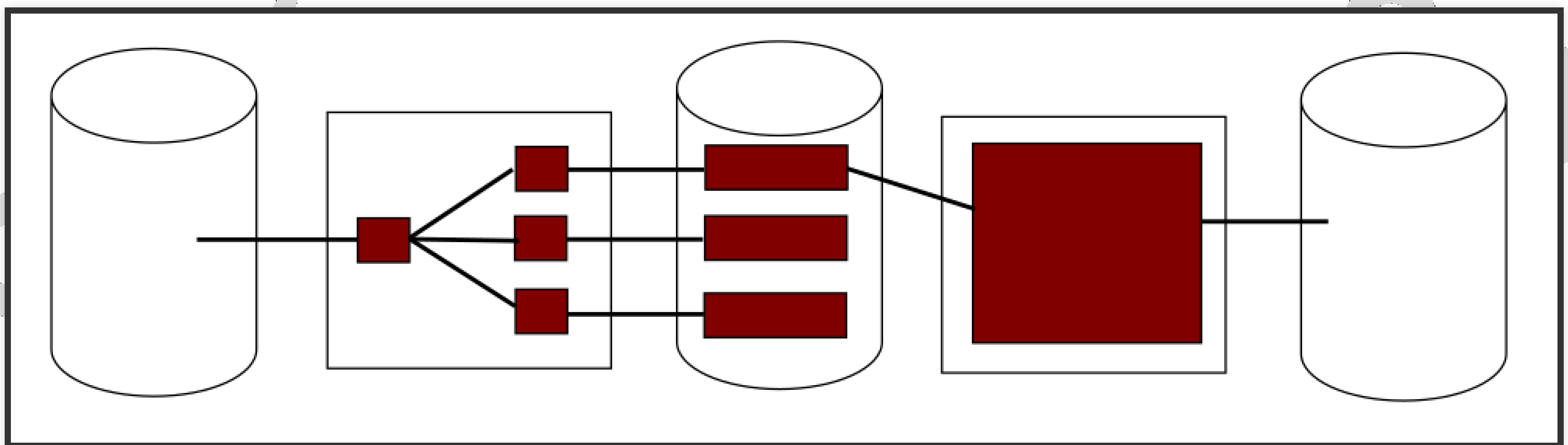


DUPLICATE ELIMINATION USING HASHING

Rehash

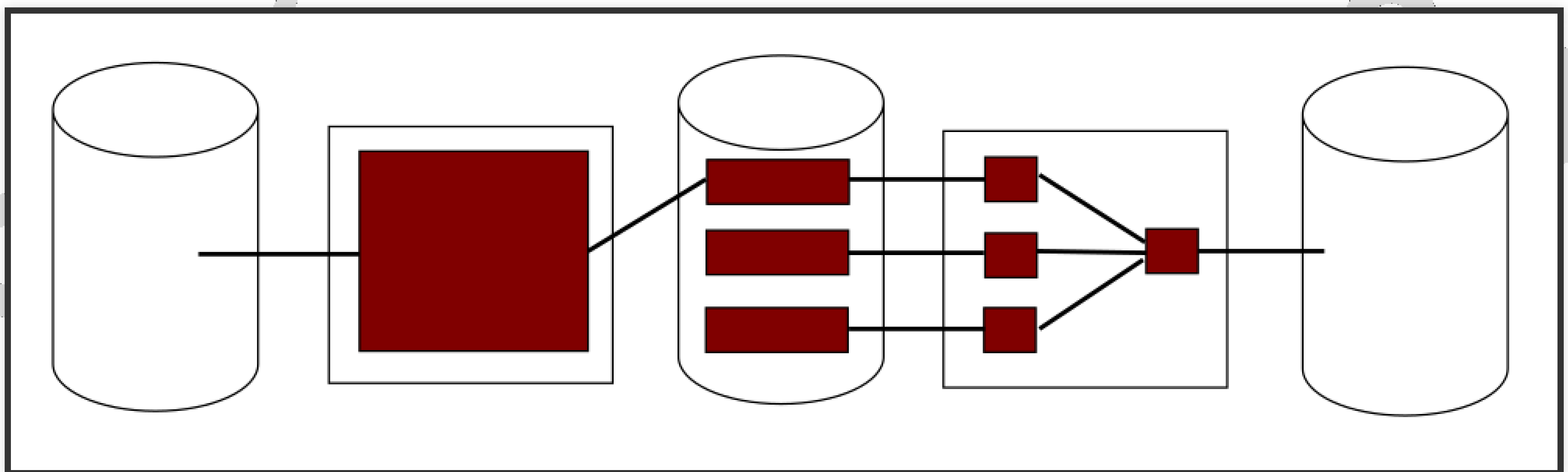


COST OF EXTERNAL HASHING



cost = 3*|R| IO's

HOW DOES THIS COMPARE WITH EXTERNAL SORTING?



cost = 3*|R| IO's

HOW ABOUT THE MEMORY REQUIREMENTS?

MEMORY REQUIREMENT FOR EXTERNAL HASHING

How big of a table can we **hash** in two passes?

- B-1 "partitions" result from Phase 0
- Each should be no more than B pages in size
- Answer: B(B-1).

Said differently:

- We can hash a table of size N pages in about $\sqrt{N}$ space
  - Note: assumes hash function distributes records evenly!

MEMORY REQUIREMENT FOR EXTERNAL SORTING

How big of a table can we **sort** in two passes?

- Each “sorted run” after Phase 0 is of size B
- Can merge up to B-1 sorted runs in Phase 1
- Answer: B(B-1).

Said differently:

- We can sort a table of size N pages in about $\sqrt{N}$ space

SORTING VS HASHING FOR PROJECTIONS

So which is better ??

Based on our simple analysis:

- Same memory requirement for 2 passes ($\sqrt{N}$)
- Same IO cost

Digging deeper …

SORTING VS HASHING FOR PROJECTIONS

Sorting pros:

- Great if input already sorted (or almost sorted)
- Great if need output to be sorted anyway
- Not sensitive to “data skew” or “bad” hash functions

Hashing pros:

- Highly parallelizable
- Can exploit extra memory to reduce # IOs (stay tuned…)

JOIN

Joins are very common.

```
SELECT *
FROM Reserves R1, Sailors S1
WHERE R1.sid = S1.sid
```

$R \times S$ is large; so, $R \times S$ followed by a selection is inefficient.

Many approaches to reduce join cost - lets have a look.

SIMPLE NESTED LOOPS JOIN

Tuple at a time nested loops evaluation: $R \bowtie S$:

```
foreach tuple r in R do
    foreach tuple s in S do
        if r.sid == s.sid then
            add <r, s> to result
```

Cost = $(p_R\,|R|)|S| + |R|$ = 100 * 1000 * 500 + 1000 IOs

At 10ms/IO ⇒ Total time ~6 days

- What if smaller relation (S) was “outer”?
- What is the cost if one relation can fit entirely in memory?

PAGE-ORIENTED NESTED LOOPS JOIN

Lets do a page at a time!

```
foreach page bR in R do
    foreach page bS in S do
        foreach tuple r in bR do
            foreach tuple s in bS do
                if r == s then
                    add <r, s> to result
```

PAGE-ORIENTED NESTED LOOPS JOIN

Cost = |R|*|S| + |R| = 1000 * 500 + 1000

If smaller relation (S) is outer, cost = 500 * 1000 + 500

Much better than naive per-tuple approach!

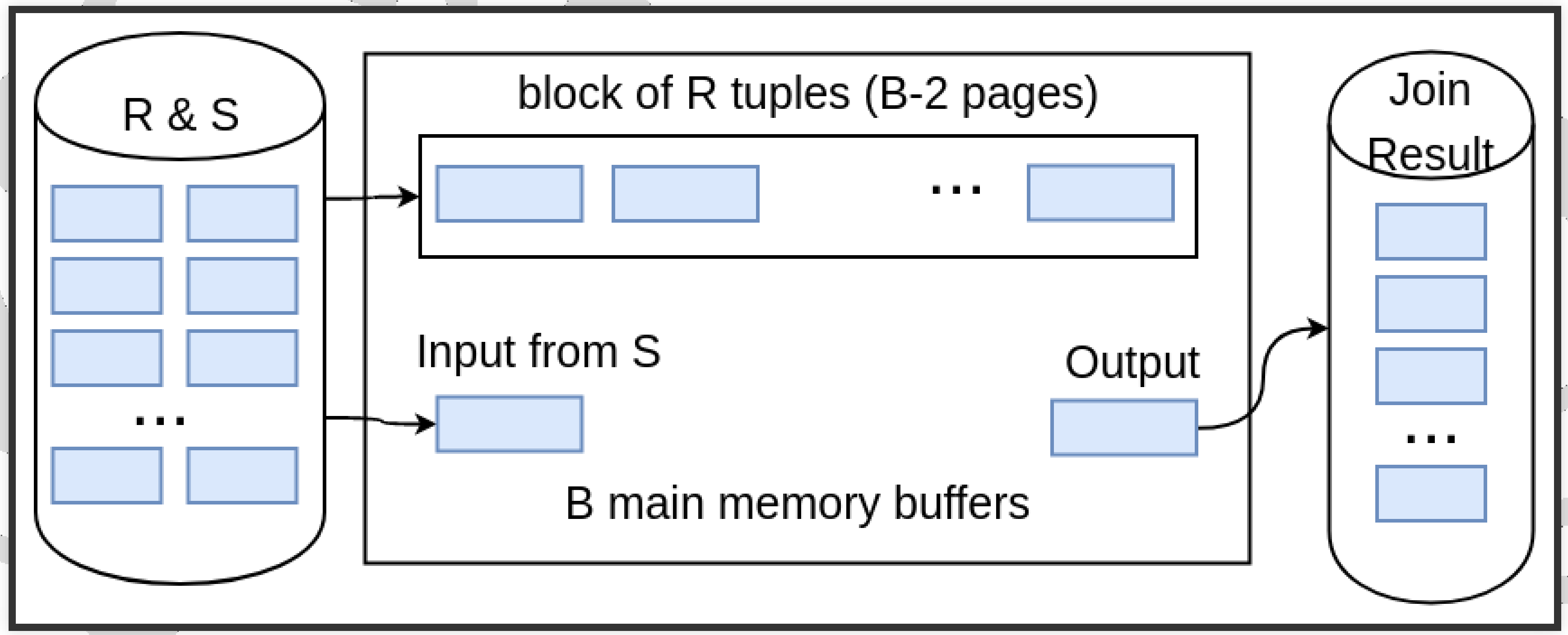
- At 10ms/IO ⇒ total time ~ 1.4 hour

The trick is to reduce the # complete reads of the inner table

BLOCK NESTED LOOPS JOIN

Page-oriented NL doesn't exploit extra buffers :(

Idea to use memory efficiently:



Cost: Scan outer + (#outer blocks * scan inner)

#outer blocks = ⌈ # pages of outer / blocksize ⌉

BLOCK NESTED LOOPS JOIN EXAMPLE

Say we have B = 100+2 memory buffers

Join cost = |outer| + (#outer blocks * |inner|)

#outer blocks = |outer| / 100

BLOCK NESTED LOOPS JOIN EXAMPLE

With R as outer (|R| = 1000):

- Scanning R costs 1000 IO's (done in 10 blocks)
- Per block of R, we scan S; costs 10 * 500 I/Os
- Total = 1000 + 10 * 500.
- At 10ms/IO, total time: ~ 1 minute

BLOCK NESTED LOOPS JOIN EXAMPLE

With S as outer (|S| = 500):

- Scanning S costs 500 IO's (done in 5 blocks)
- Per block of S, we scan R; costs 5*1000 IO's
- Total = 500 + 5*1000.
- At 10ms/IO, total time: ~ 55 seconds

BLOCKED ACCESS AND DOUBLE-BUFFERING

Taking blocked acces into account, best approach is to split buffer pool evenly between outer and inner.

- Will result in more page fetching
- Signifcant reduce seeking for pages

Could also use double buffering (like in external sort)

INDEX NESTED LOOPS JOIN

$R \bowtie S$:

```
foreach tuple r in R do
    foreach tuple s in S where r.att1 == s.att2 do
        add <r, s> to result
```

- Outer loop use file scan
- Inner loop uses index to find matching tuples

Cost = $|R| + (|R| * p_R) *$ cost to find matching S tuples

COST OF INDEX NESTED LOOPS JOIN

- If index uses Alt. 1, cost = cost to traverse tree from root to leaf.
- For Alt. 2 or 3:
  1. Cost to lookup RID(s); typically 2-4 IO's for B+Tree.
  2. Cost to retrieve records from RID(s); depends on clustering.
     - Clustered index: 1 I/O per page of matching S tuples.
     - Unclustered: up to 1 I/O per matching S tuple.

SORT-MERGE JOIN

1. Sort R on join attr(s)
2. Sort S on join attr(s)
3. Scan sorted-R and sorted-S in tandem, to find matches

Cost: Sort R + Sort S + (|R|+|S|)

In the worst case, last term could be |R|*|S| (very unlikely!)

Question: what is worst case?

COST OF SORT-MERGE JOIN

Suppose B = 35 buffer pages:

- Both R and S can be sorted in 2 passes
- Total join cost = 4*1000 + 4*500 + (1000 + 500) = 7500

Suppose B = 300 buffer pages:

- Again, both R and S sorted in 2 passes
- Total join cost = 7500

SORT-MERGE JOIN REFINEMENT

Do the join during the final merging pass of sort !

If have enough memory, can do:

1. Read R and write out sorted runs
2. Read S and write out sorted runs
3. Merge R-runs and S-runs, and find $R \bowtie S$ matches

Cost = $3*|R| + 3*|S|$

SORT-MERGE JOIN

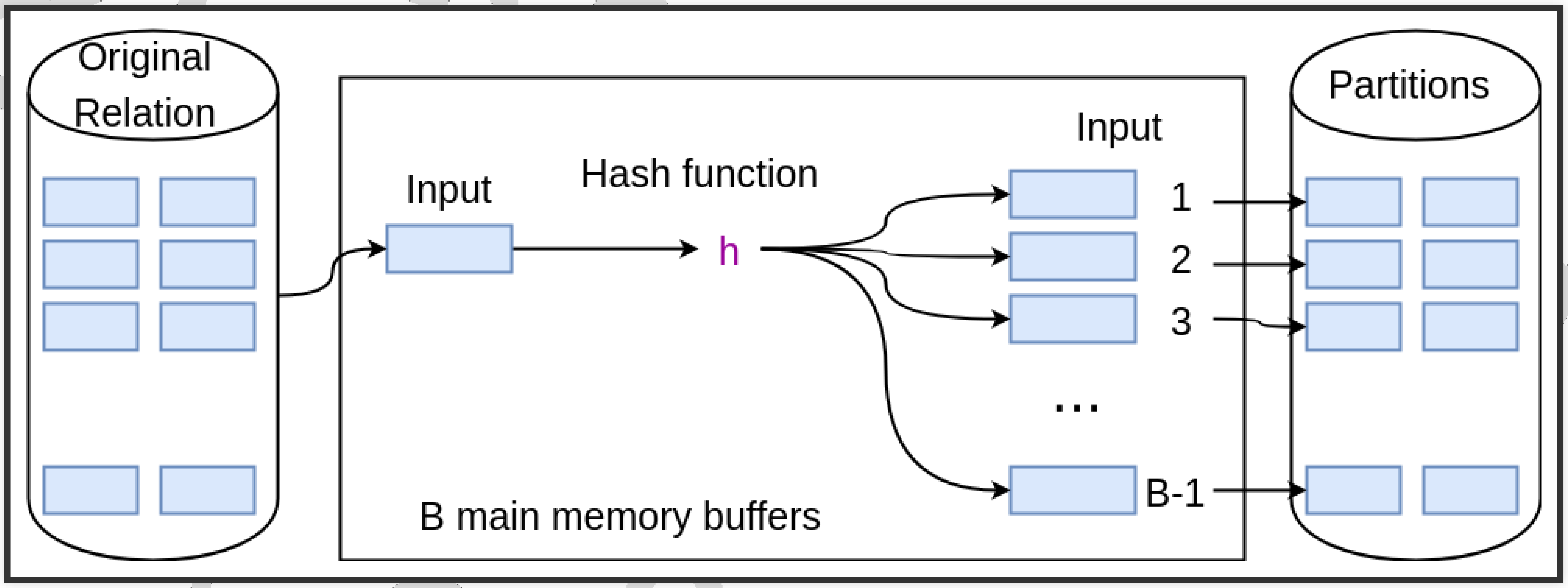
Sort-merge join an especially good choice if:

- one or both inputs are already sorted on join attribute(s)
- output is required to be sorted on join attribute(s)

HASH JOIN

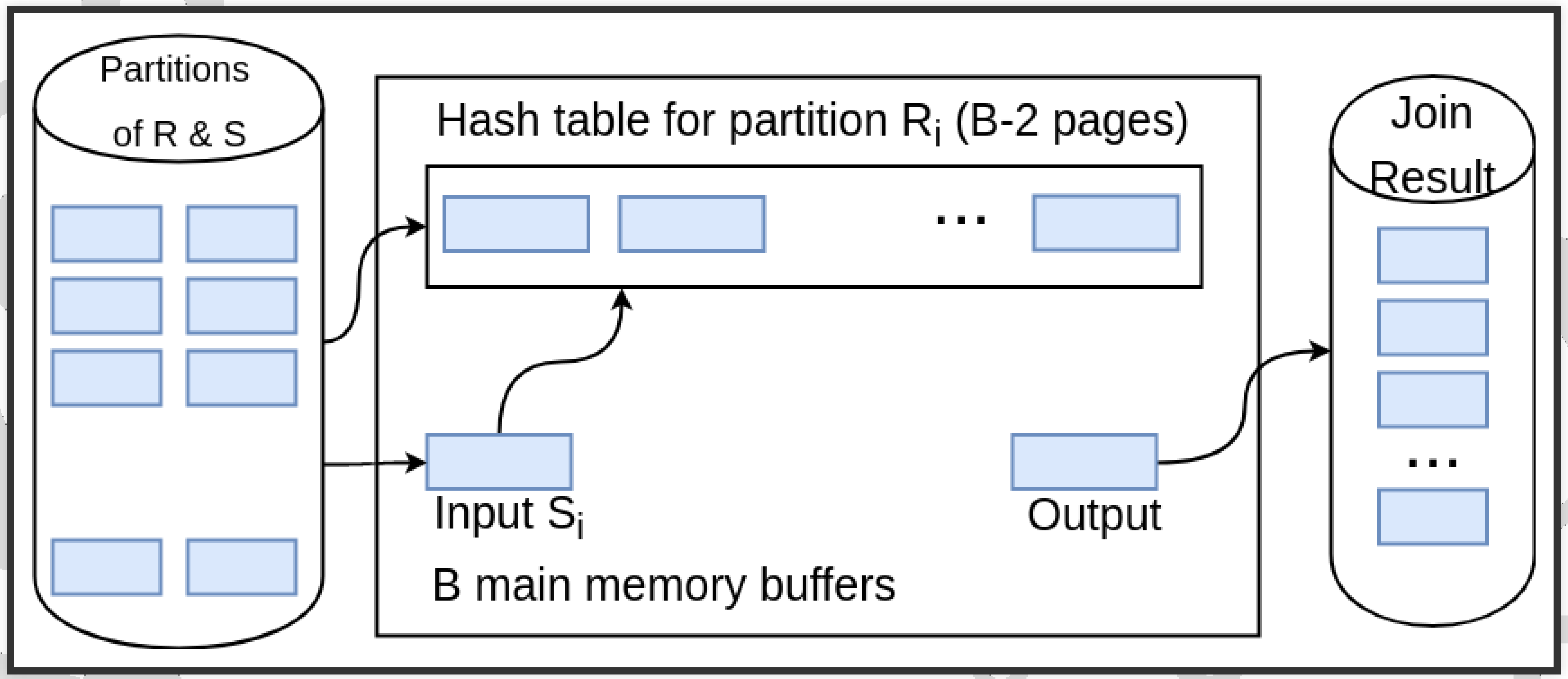
Use idea for large dataset: **Partitioning** and **Matching**

Partitioning phase: read + write both relations ⇒ 2(|R|+|S|) I/Os



HASH JOIN

Matching phase: read both relations ⇒ |R|+|S| I/Os



Total cost of 2-pass hash join = 3(|R|+|S|)

MEMORY REQUIREMENTS AND OVERFLOW HANDLING

- Q: how much memory needed for 2-pass hash join?
- what is cost of 2-pass sort merge join?
- how much memory needed for 2-pass sort merge join?

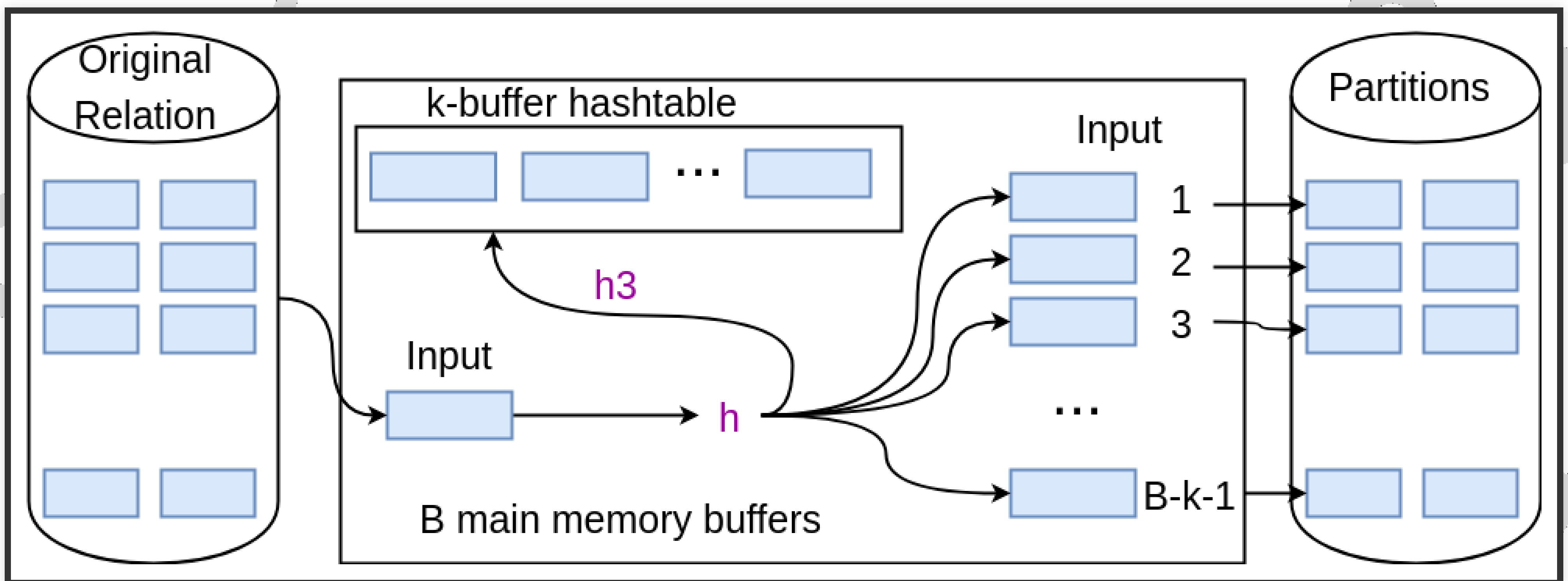
UTILIZING EXTRA MEMORY: HYBRID HASH JOIN

Have B memory buffers

Want to hash relation of size N

If $B < N < B^2$, we will have unused memory …

HYBRID HASH JOIN



k has to be greater than the size of a partition: $k \geq \frac{N}{B-k-1}$

FURTHER DETAILS OF HYBRID HASHING

If we have even more memory, then we can put more than one bucket in the memory

In general, if we can find ($k \leq B$), such that $k \geq \frac{mN}{B-k-1}$, then we can put m buckets in the memory, while the remaining $B - k - 1 - m$ buckets are stored on the disk.

HASH JOIN VS SORT-MERGE JOIN

Sorting pros:

- Good if input already sorted, or need output sorted
- Not sensitive to data skew or bad hash functions

Hashing pros:

- Often cheaper due to hybrid hashing
- For join: # passes depends on size of smaller relation
- Highly parallelizable

OTHER OPERATIONS

GROUPING AND AGGREGATION

Can you modify the projection algorithm to perform Grouping and Aggregation?

```
SELECT R.sid, Count(*)
FROM Reserves R
GROUP BY R.sid
```

GROUPING AND AGGREGATION

Here is the algorithm - what should be modified?

1. Scan R, extract only the needed attributes
2. Sort the resulting set
3. Remove adjacent duplicates

Modify the external sort algorithm:

- Modify Pass 0 to eliminate unwanted fields.
- Modify Passes 1+ to eliminate duplicates.

GROUPING AND AGGREGATION

1. Generate sorted (on the grouping attributes) runs (pass 0 to n-1)
2. Get the tuple with the least key v value and start a new group
    1. Prepare to compute the aggregates for the group
    2. Get all the tuples with key v and update the aggregates of the current group
        - min, max
        - count
        - sum
        - avg
    3. If a buffer becomes empty, read from the disk
3. If there are more tuples goto 2.

SET OPERATIONS

Question: How can we perform union, difference and intersection?

QUERY EVALUATION PLANS

Intro to Query Optimization

QUERY EVALUATION PLANS

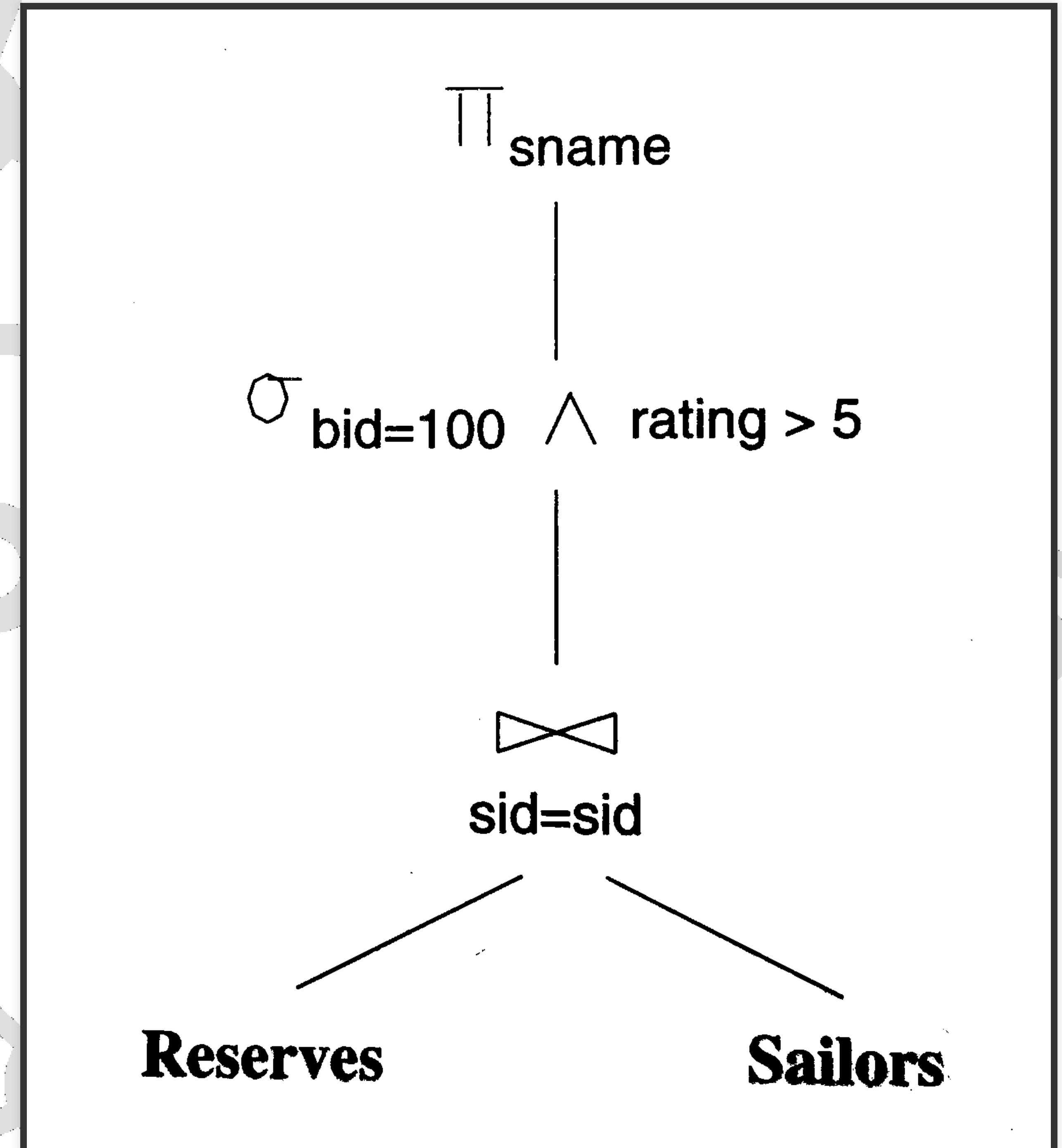
```sql
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
    AND R.bid = 100 AND S.rating > 5
```

QUERY EVALUATION PLANS

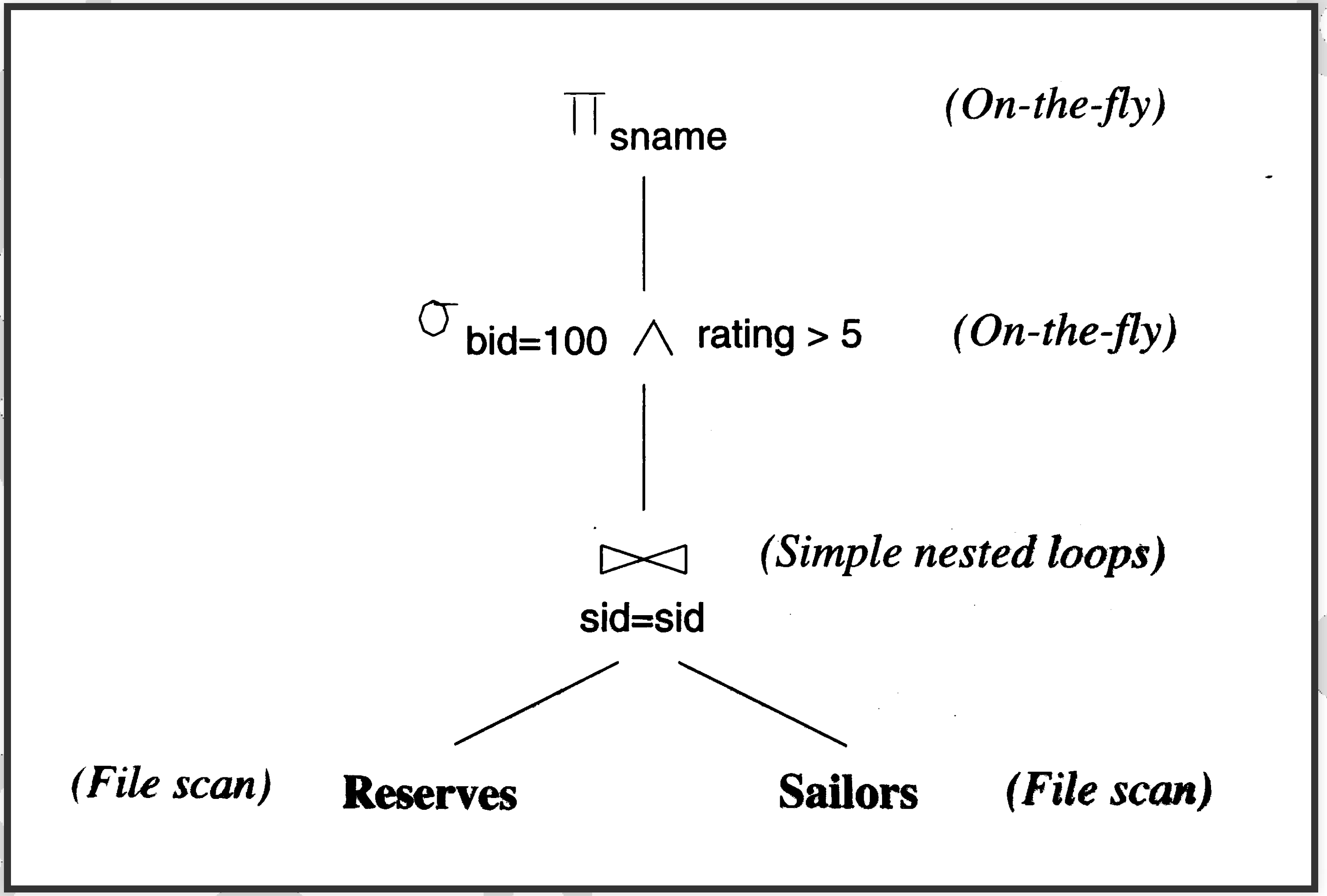
As relational algebra

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

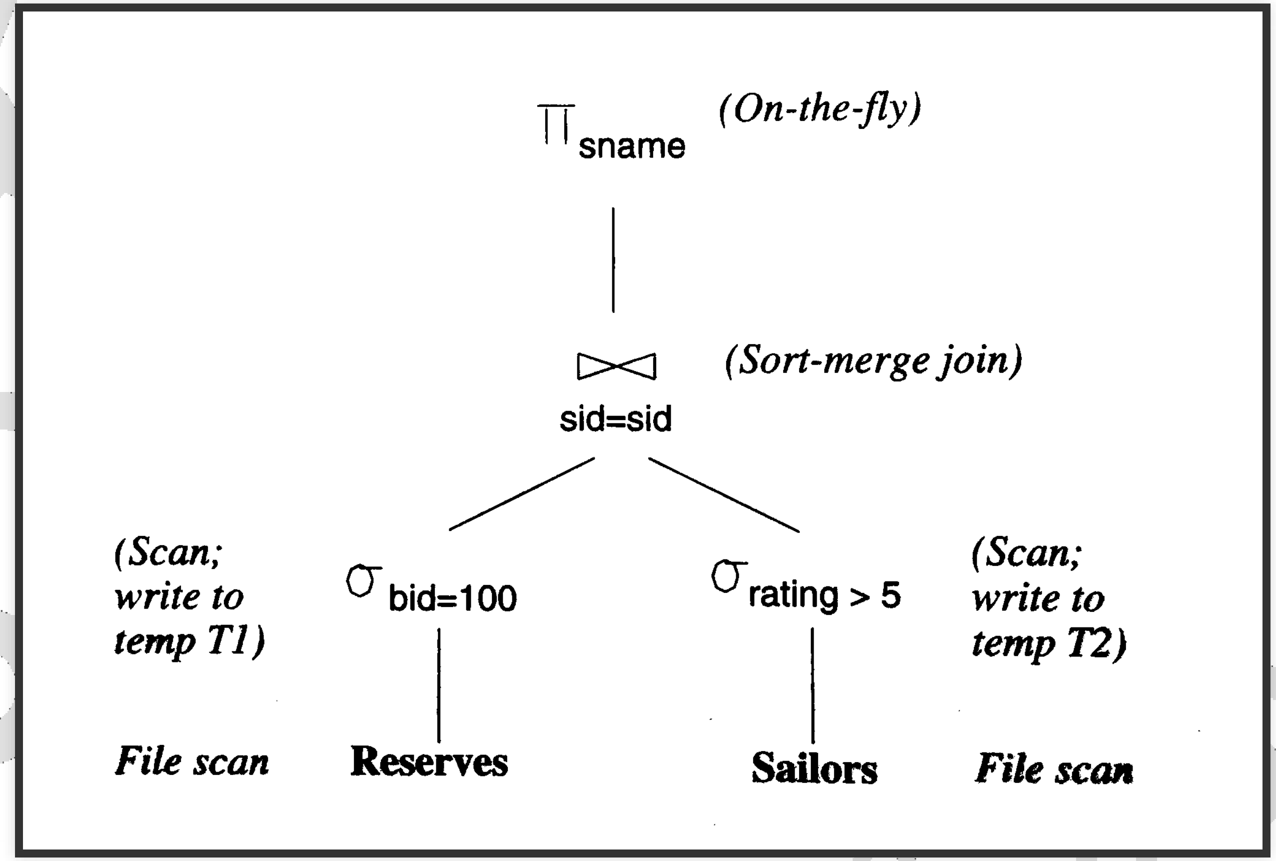
QUERY EVALUATION PLANS



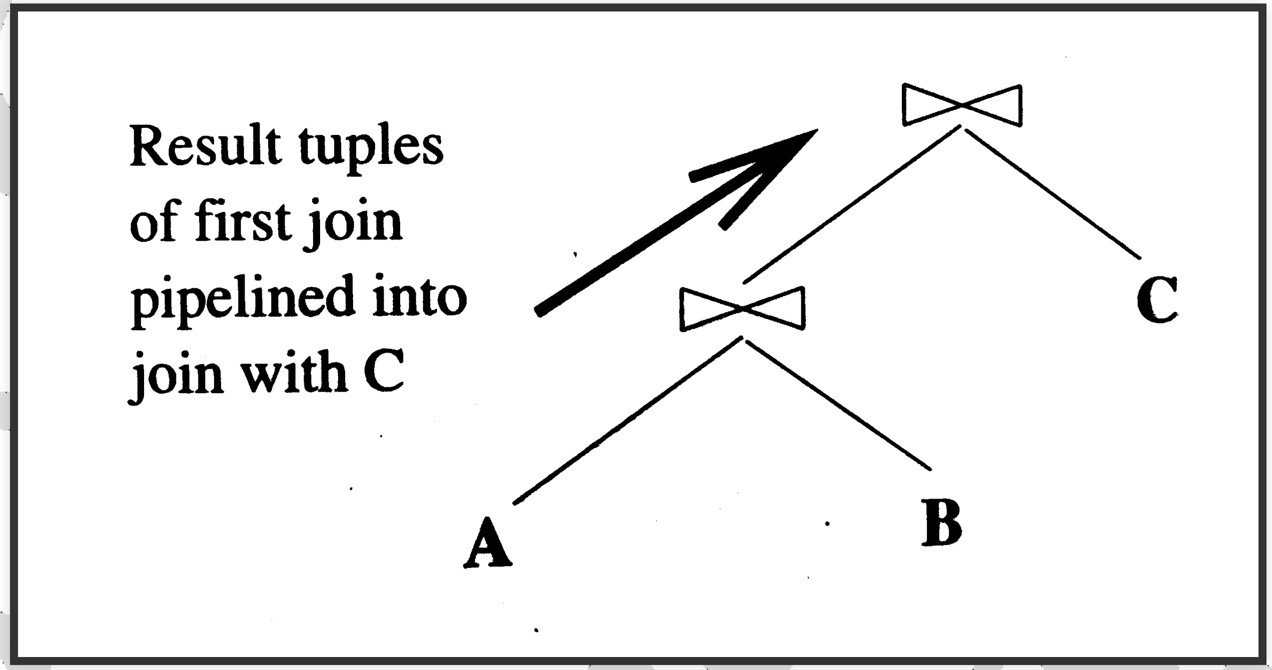
QUERY EVALUATION PLANS



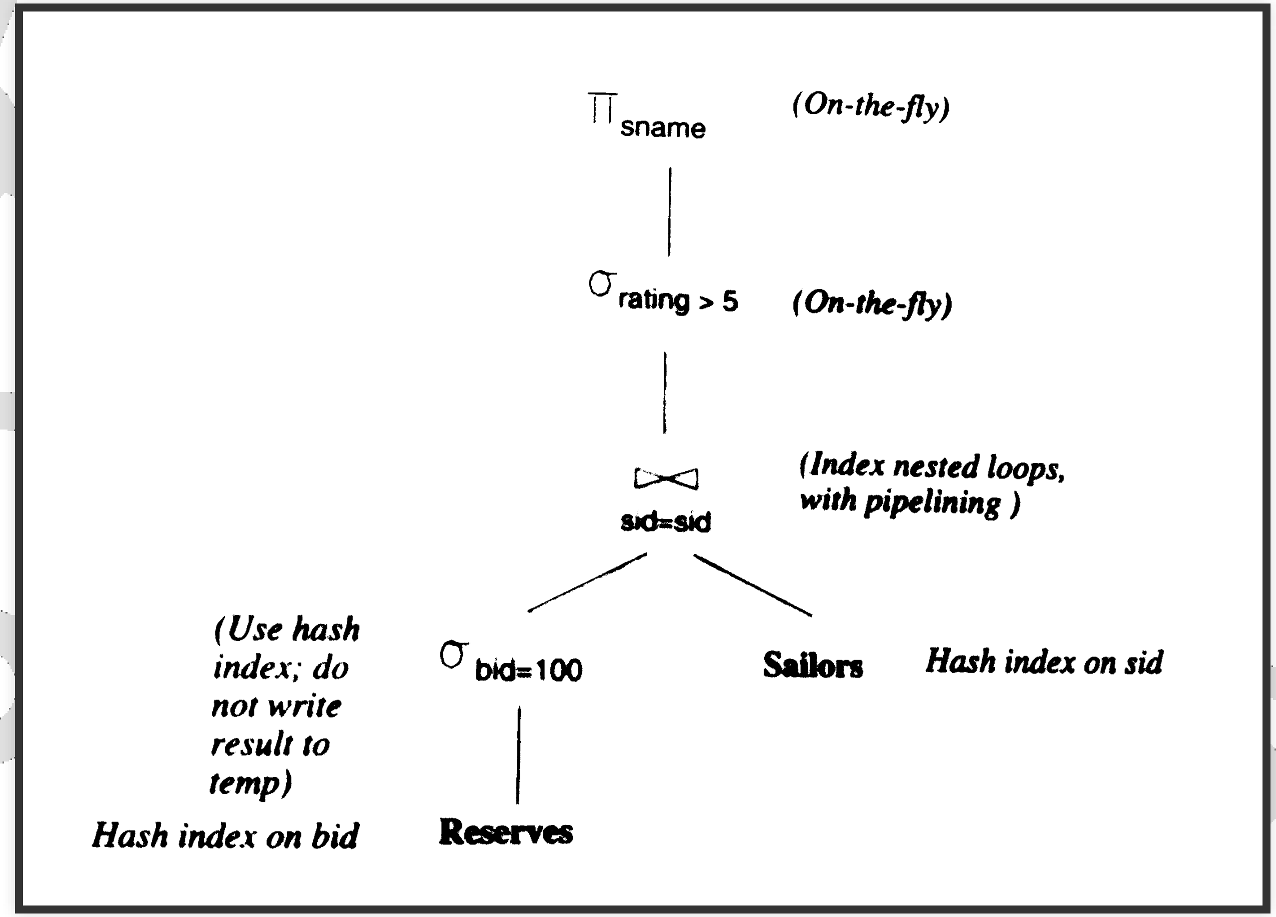
PUSHING SELECTIONS



PIPELINING



USING INDEXES



OPTIMIZATION

Much more next time!

SUMMARY

A virtue of relational DBMSs: queries are composed of a few basic operators

- The implementation of these operators can be carefully tuned
  - Sorting is an important technique for evaluating many operators
  - Index could be exploited in many cases too (be careful with unclustered index)
- Many alternative implementation techniques for each operator
  - No universally superior technique for most operators.
- Must consider available alternatives
  - Called **Query optimization** — we will study this topic next!

QUESTIONS?