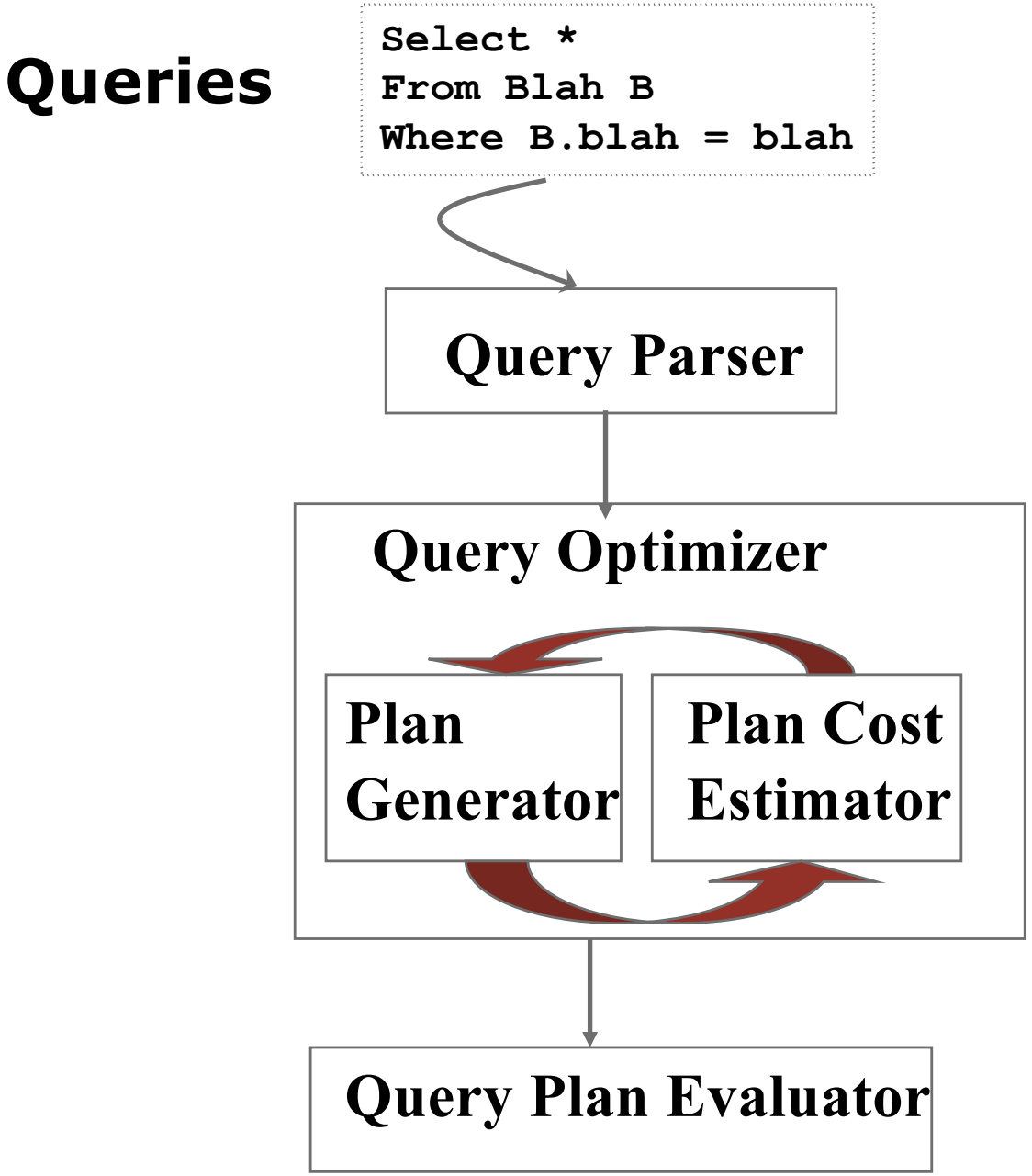
UNIVERSITY OF COPENHAGEN

Data Processing: Implementation of Relational Operators

ACS, Yongluan Zhou & Marcos Vaz Salles

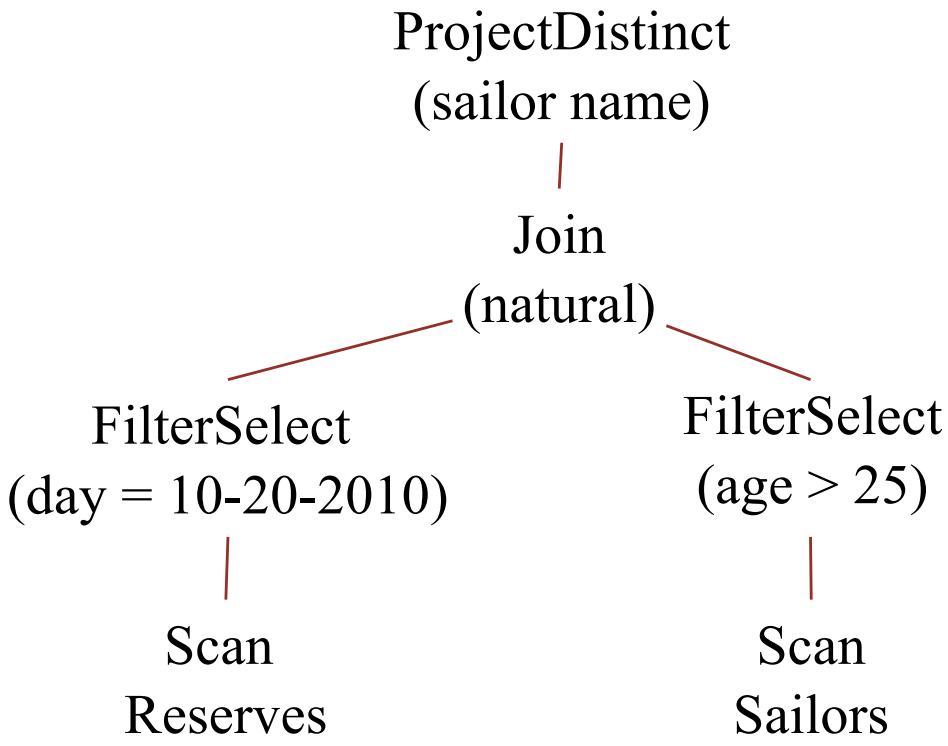
with slides revised by Michael Kirkedal Carøe, originally from *Database Management Systems, Ramakrishnan and Gehrke*

Query Sub-System



Translating SQL Query to Relational Operators

```
SELECT DISTINCT S.name
FROM R JOIN S
WHERE R.day = 10-20-2010 AND S.age>25
```



What should we learn today?

- Implementation of relational operators,
  - including selections, projections, joins, set operations, and aggregation
- Loop-based implementations
  - and techniques such as use of blocks and indices to improve their performance
- Hashing- and sorting-based implementation
  - durabili
- Operator Interface

Relational Operators

We now study implementation alternatives

- Select
- Project
- Join
- Set operations (union, intersect, except)
- Aggregation

Select Operator

```
SELECT  *
FROM    Sailor S
WHERE   S.Age = 25 AND S.Salary > 100K
```

- How best to perform? Depends on:
  - what indexes are available
  - expected size of result
- Case 1: No index on any selection attribute
- Case 2: Have "matching" index on all selection attributes
- Case 3: Have "matching" index on some (but not all) selection attributes

Case 1: No index on any selection attribute

```
SELECT *
FROM   Sailor S
WHERE  S.Age = 25 AND S.Salary > 100K
```

- Single loop: Just scan and filter!
- If relation has N pages, cost = N
  - Assume |S| = 1000 pages, cost = 1000 pages

Case 2: "Matching" index on all selection attributes

```
SELECT  *
FROM    Sailor S
WHERE   S.Age = 25 AND S.Salary > 100K
```

- Assume index on (Age, Salary)

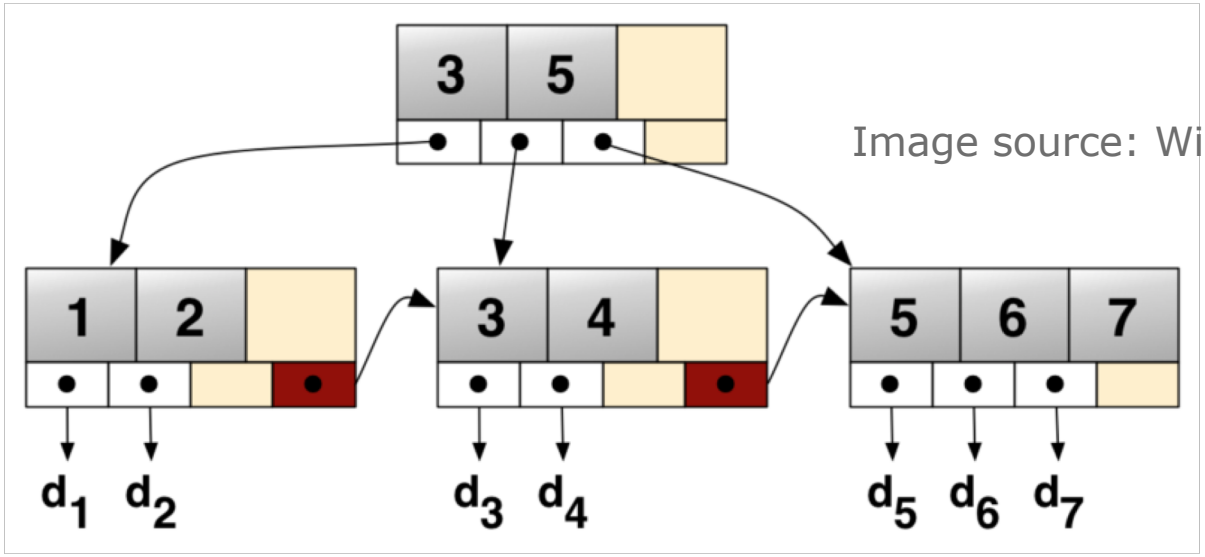


Image source: Wikipedia

Basic concepts: B+ trees

- A **B+ tree** represents sorted data which is identified by a key
- Allows for efficient insertion, retrieval and removal of records
- Used (amongst other) for database table indices

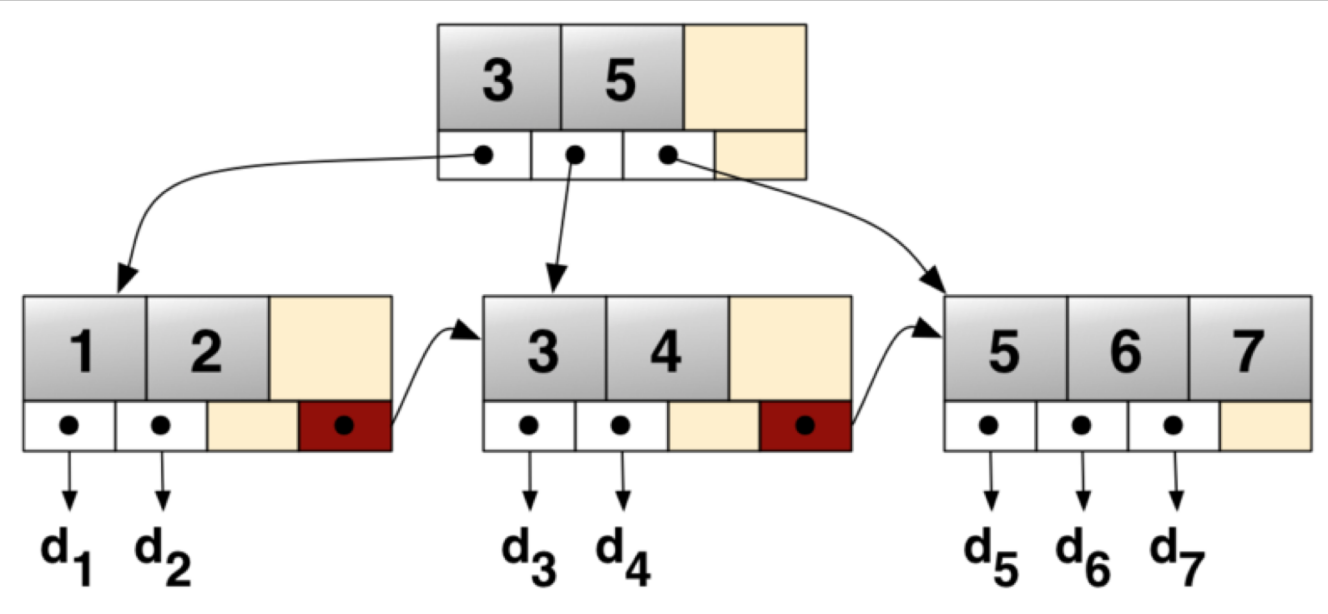


Image source: Wikipedia

Basic concepts: B+ trees

- Each node in the tree occupies a page
- Entries in non-leave nodes → called index entries: <key value, page_id>
- Entries in leaf nodes → called data entries: either containing actual data or pointer to them

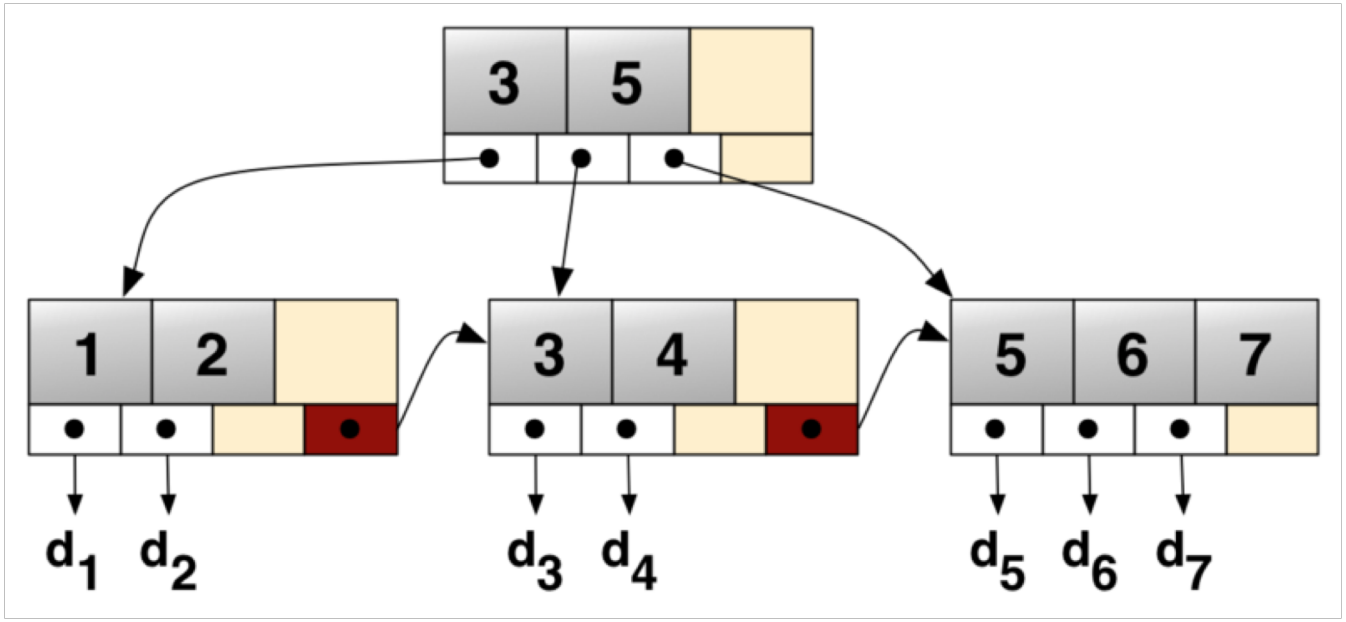
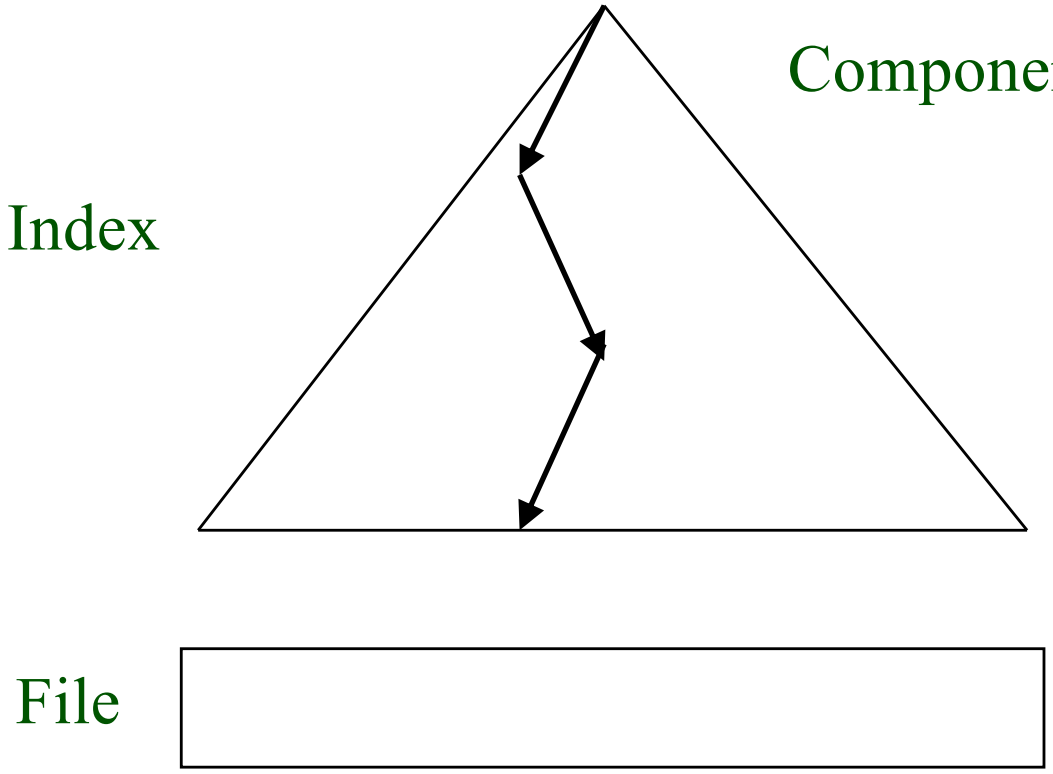
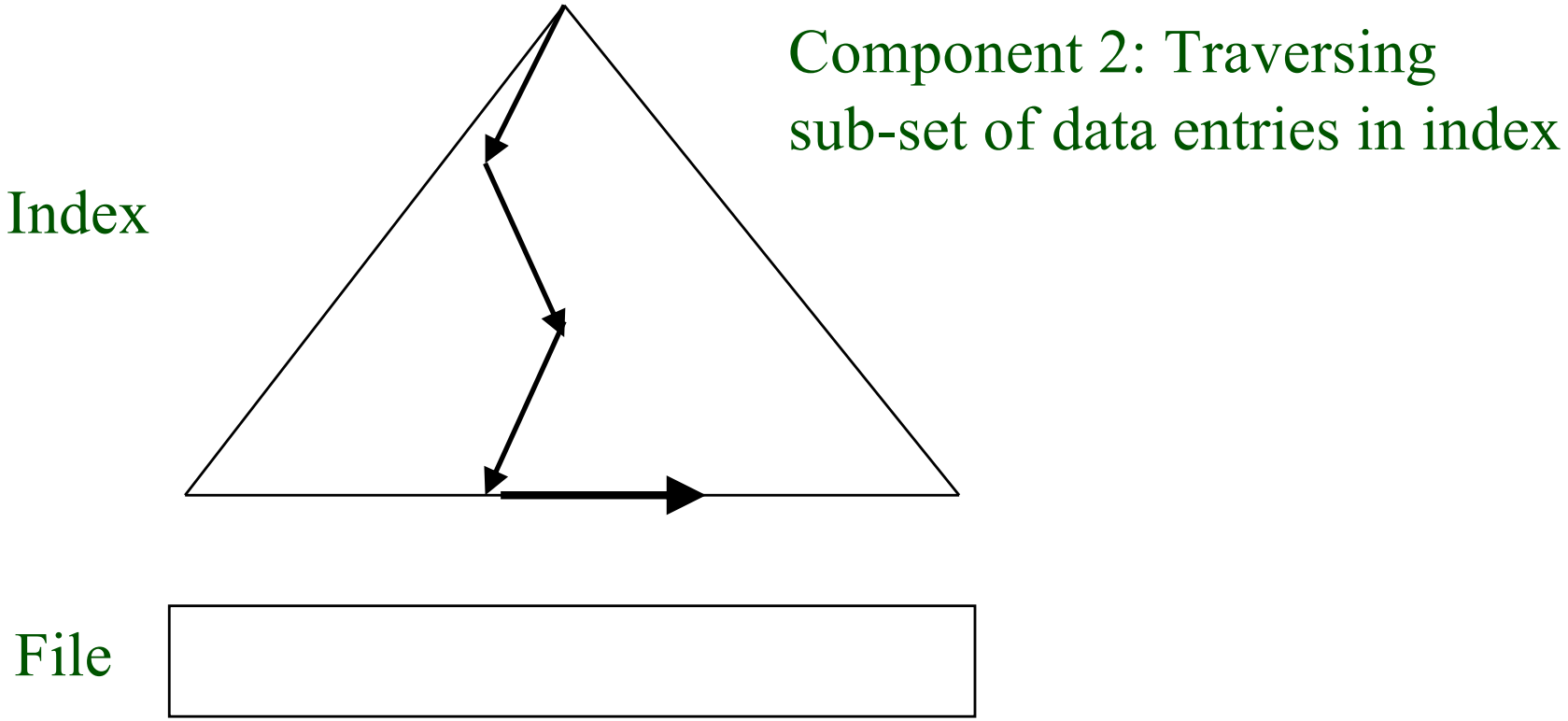


Image source: Wikipedia

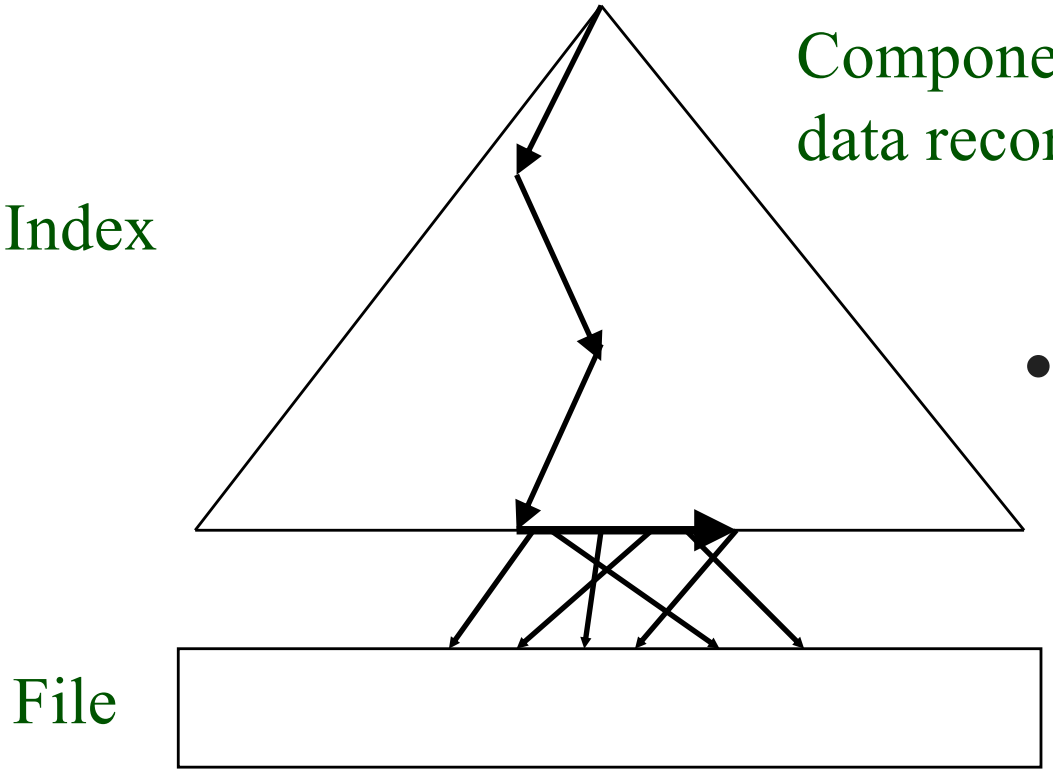
Case 2: Cost Components



Case 2: Cost Components



Case 2: Cost Components

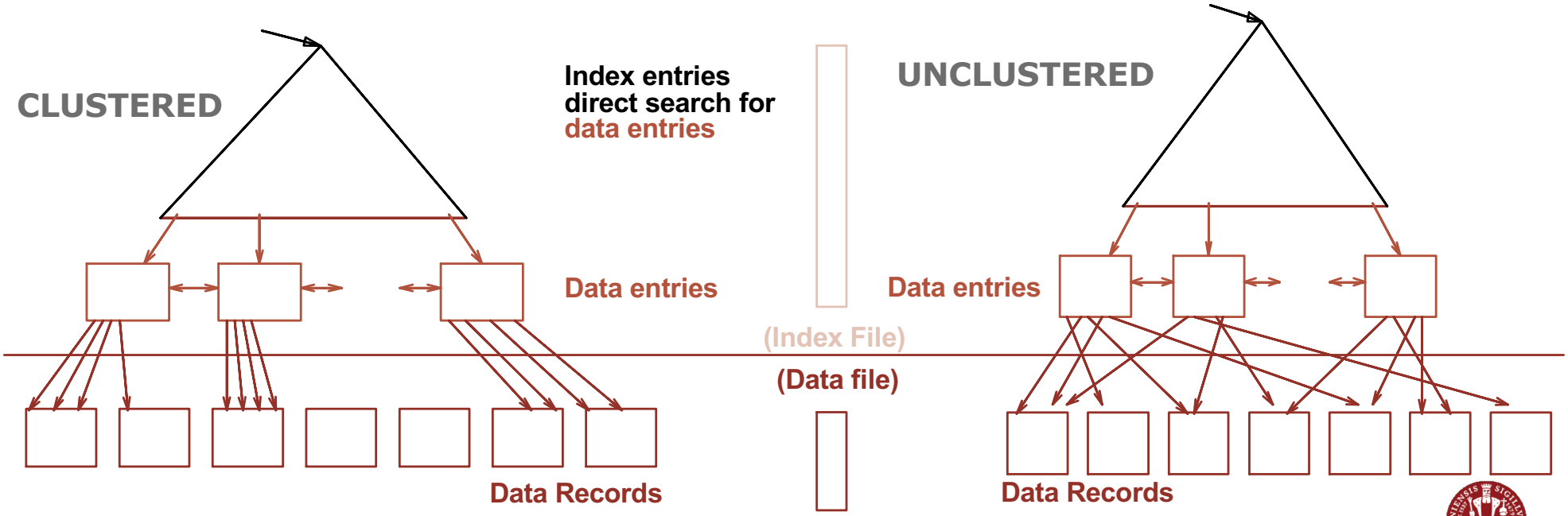


Component 3: Fetching actual data records (indirect indexes)

- Depending on how many records fetched (selectivity), random I/O cost **may exceed sequential scan cost**!

Case 2: Cost Component 3

- Assume selectivity = 10% (100 pages, 10000 tuples):
  - If clustered index, cost is 100 I/Os;
  - If unclustered, could be up to 10000 I/Os!



Case 3: "Matching" index on some attributes

```
SELECT  *
FROM    Sailor S
WHERE  S.Age = 25 AND S.Salary > 100K
```

- Assume index on Age only

Case 3: Evaluation Alternatives

- Alternative 1
  - Use available index (on Age) to get superset of relevant data entries
  - Retrieve the tuples corresponding to the set of data entries
  - Apply remaining predicates on retrieved tuples
  - Return those tuples that satisfy all predicates
- Alternative 2
  - Sequential scan! (always available)
  - May be again better depending on selectivity

Case 3: "Matching" index on some attributes

```
SELECT  *
FROM    Sailor S
WHERE   S.Age = 25 AND S.Salary > 100K
```

- Assume separate indices on Age and on Salary

Case 3: Evaluation Alternatives

- Alternative 1
  - Choose most selective access path (index)
    - Could be index on Age or Salary, depending on selectivity of the corresponding predicates
  - Use this index to get superset of relevant data entries
  - Retrieve the tuples corresponding to the set
  - Apply remaining predicates on retrieved tuples
  - Return those tuples that satisfy all predicates
- Alternative 2
  - Get rids of data records using each index
    - Use index on Age and index on Salary
  - Intersect the rids
  - Retrieve the tuples corresponding to the rids
  - Apply remaining predicates on retrieved tuples
  - Return those tuples that satisfy all predicates
- Alternative 3
  - Sequential scan!

Questions?

Relational Operators

- We now study implementation alternatives

- Select
- Project
- Join
- Set operations (union, intersect, except)
- Aggregation

Projection

```
SELECT  DISTINCT S.Name, S.Age
FROM    Sailor S
```

Main issue is duplicate elimination.

- Assume we do *not* have any indices

How would you implement duplicate elimination?
Can you use sorting or hashing?

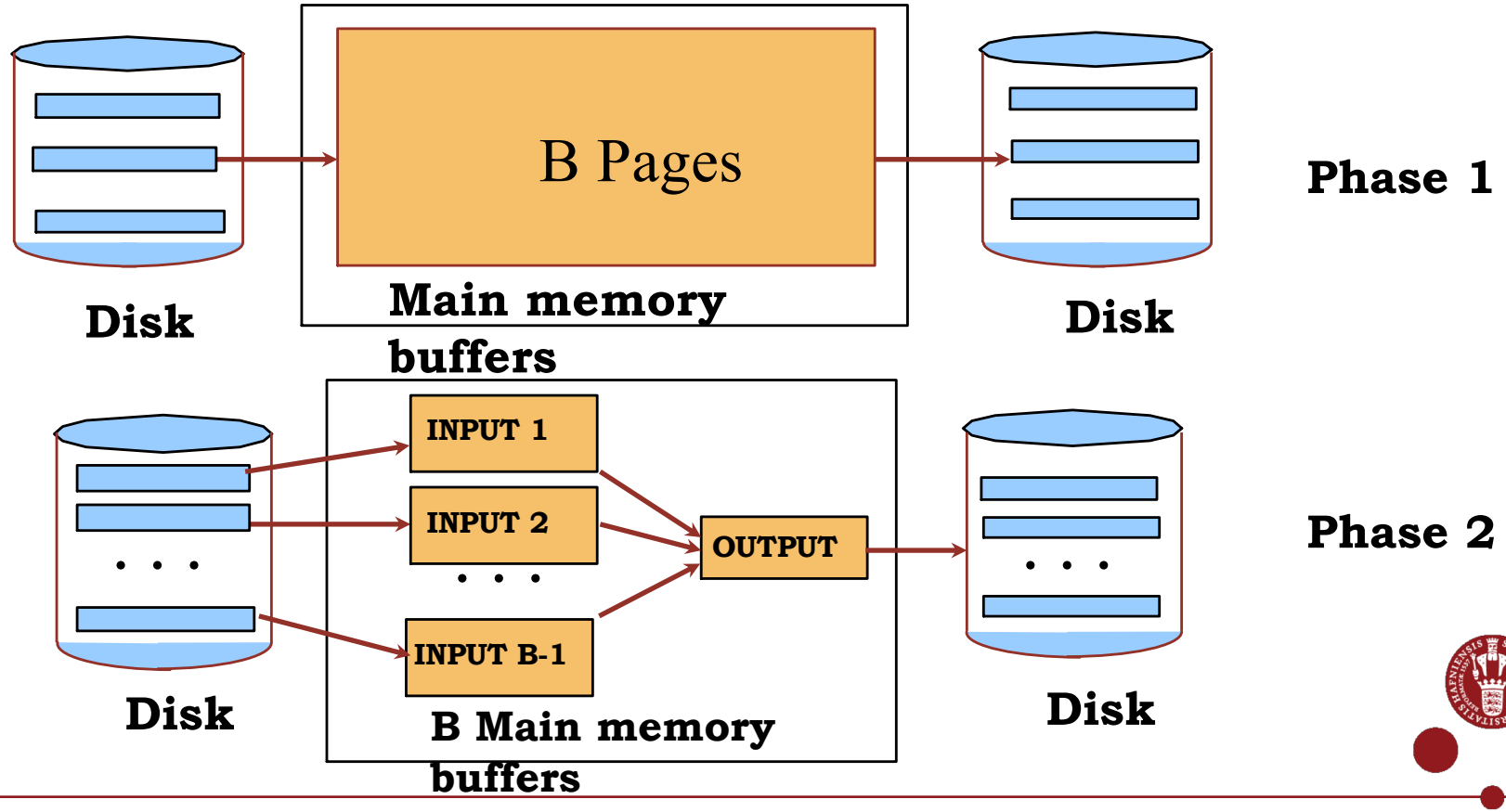
Projection without Indices

```
SELECT  DISTINCT S.Name, S.Age
FROM    Sailor S
```

- We have *no* indices
- What strategies can we use?
  - **Sorting:** Duplicates adjacent after sorting
  - **Hashing:** Duplicates hash to same buckets (in disk and memory)

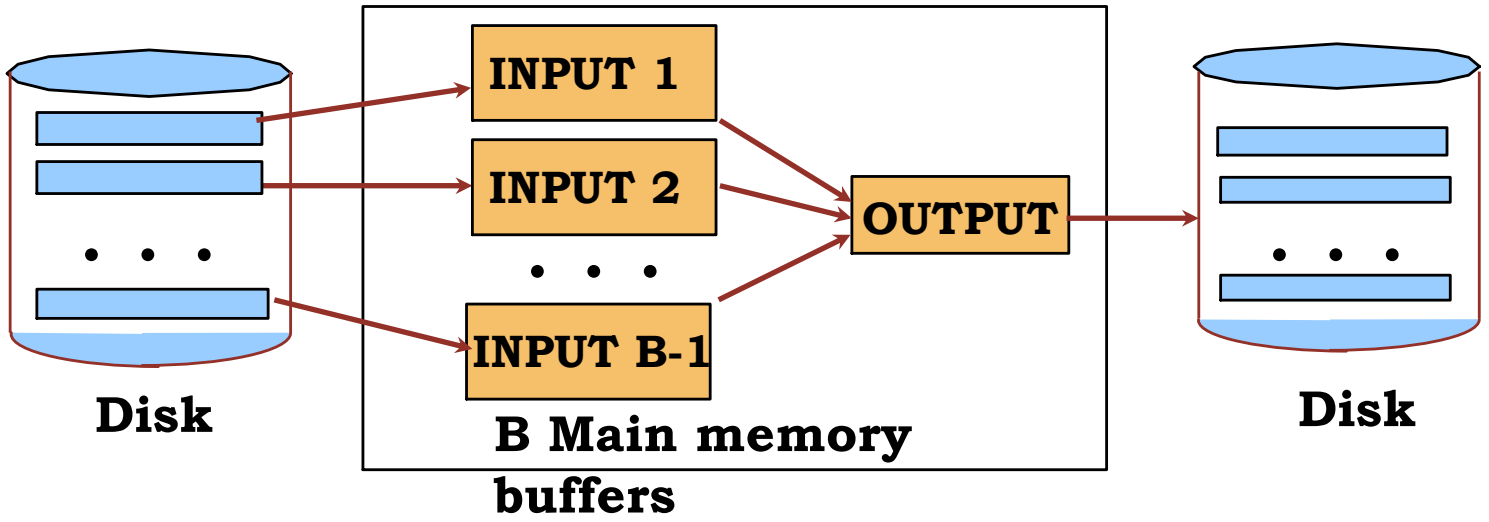
Do-it-yourself Recap: External Sorting

- What were the two phases of multi-way external sorting and how did they work?
- What optimizations could you apply to external sorting?



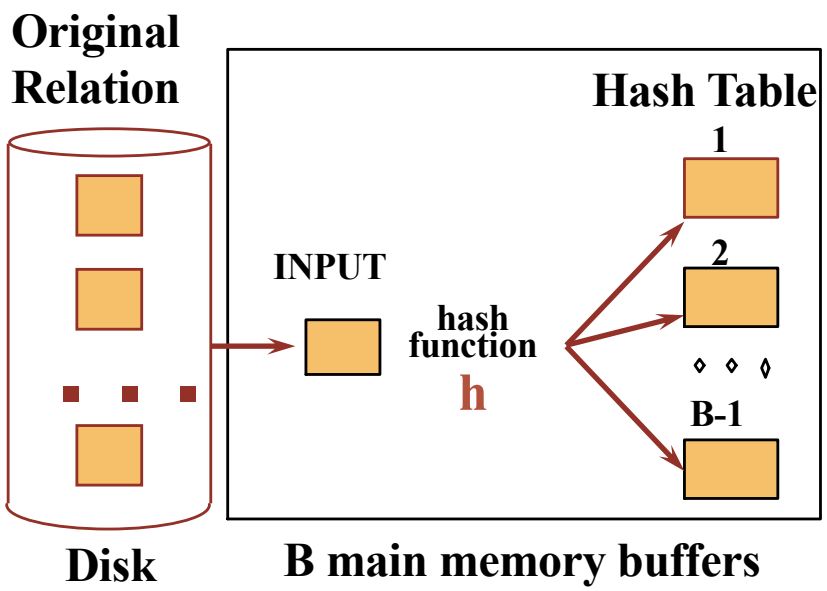
Projection with External Sorting

- Phase 1
  - Project out unwanted columns
  - Still produce runs of length B pages
  - But tuples in runs are smaller than input tuples (so smaller runs)
- Phase 2
  - Eliminate duplicates during merge
  - Even smaller runs
  - **Pipeline results to next operator, up to N I/Os saved**



Duplicate Elimination with Hashing

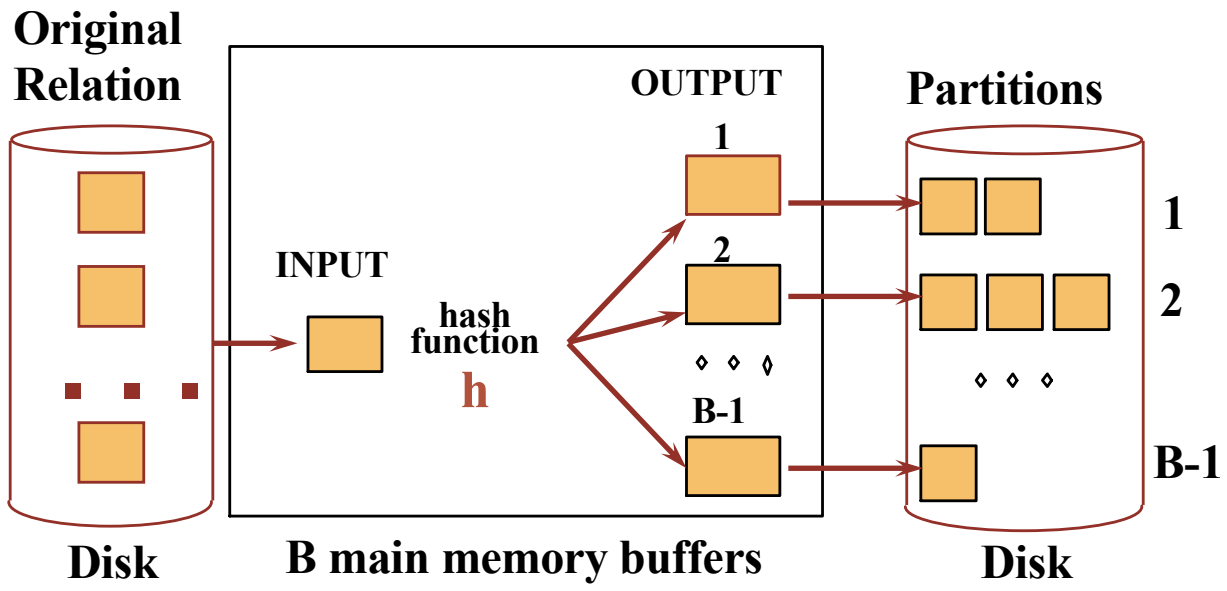
1. Apply hash function
2. Look for duplicates in the corresponding bucket
3. If the input buffer is empty, then read in a page and goto 1.



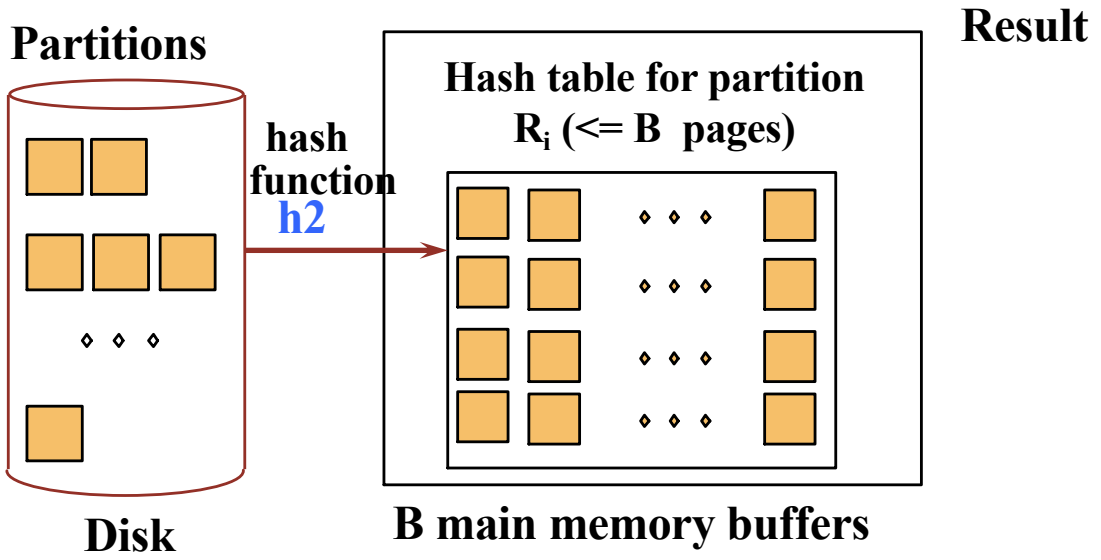
How to build hash table for data larger than the memory?

Duplicate Elimination using Hashing

- Partition:



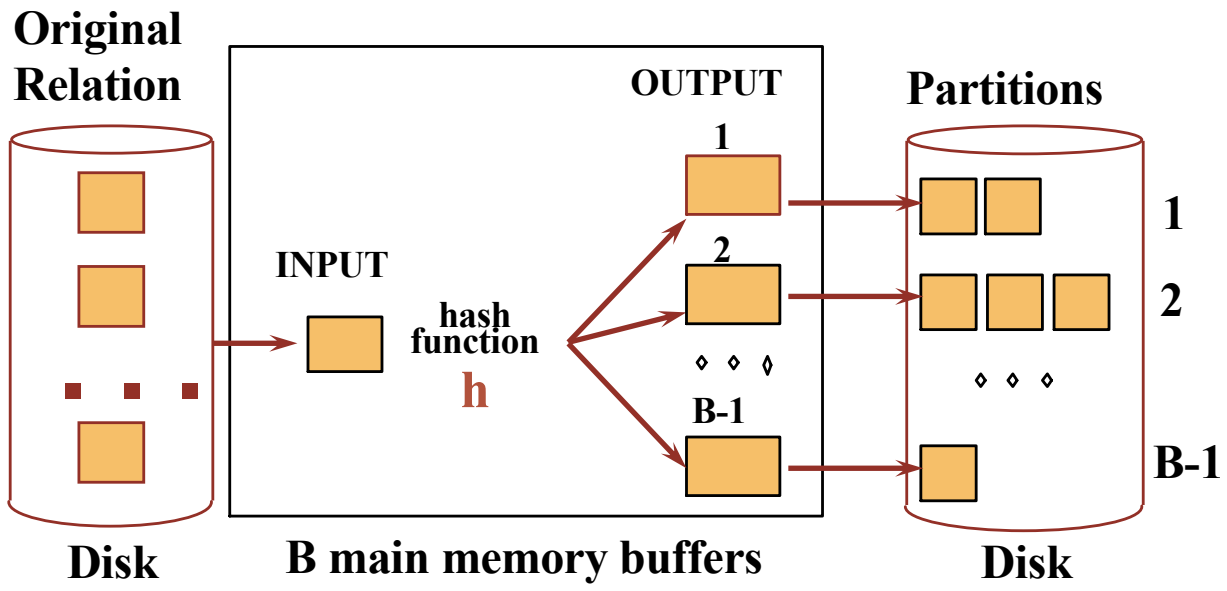
- Rehash:

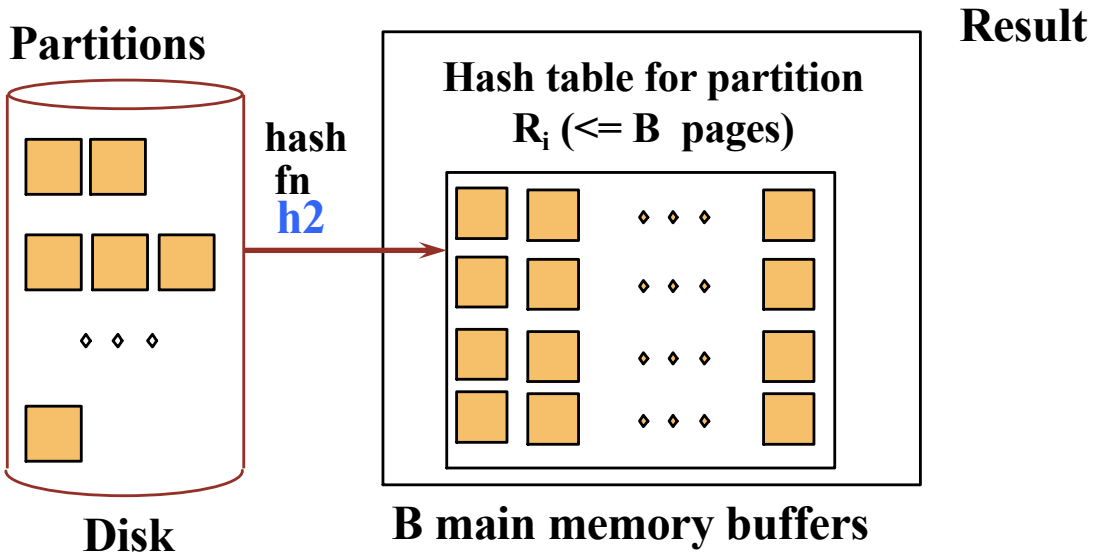


General Idea

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

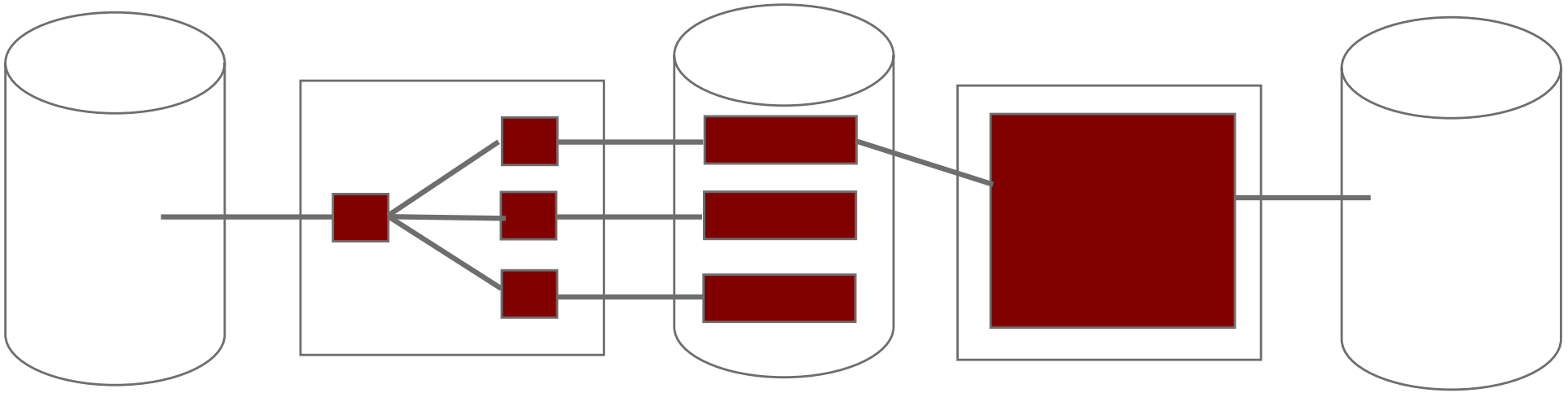
Duplicate Elimination using Hashing





What if a partition still cannot fit into memory?

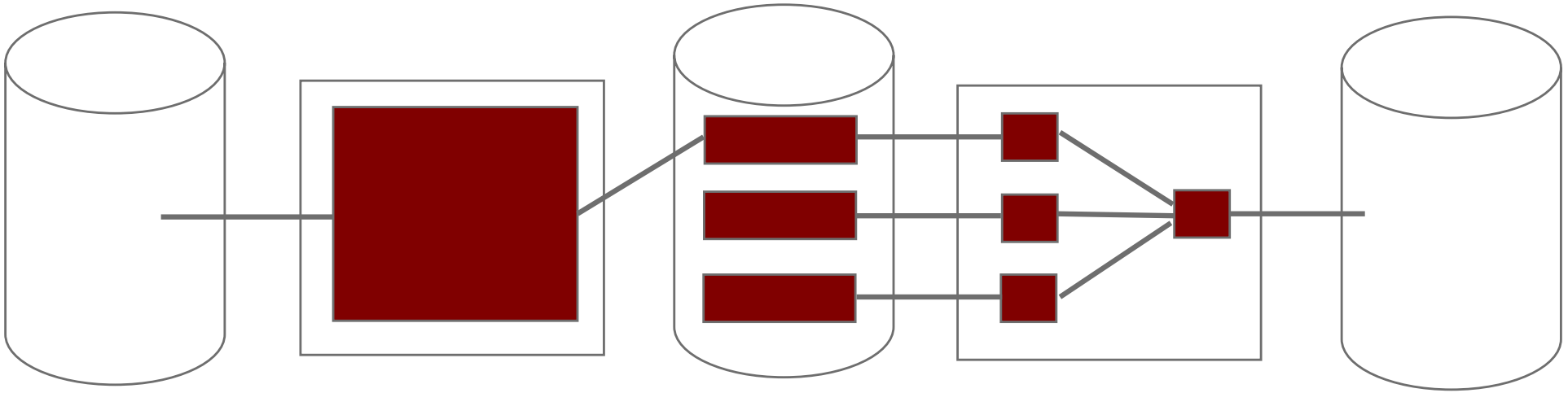
Cost of External Hashing



cost = 3*|R| IO's

How does this compare with external sorting?

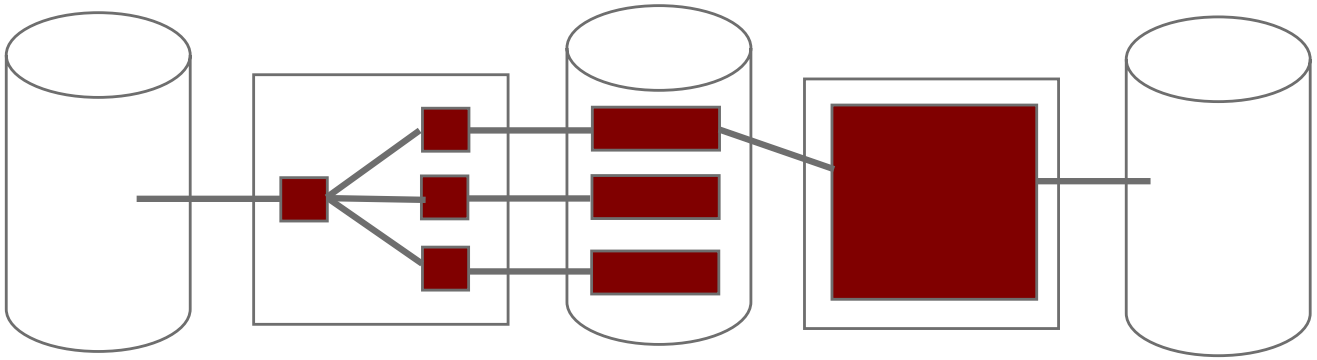
Cost of External Sorting



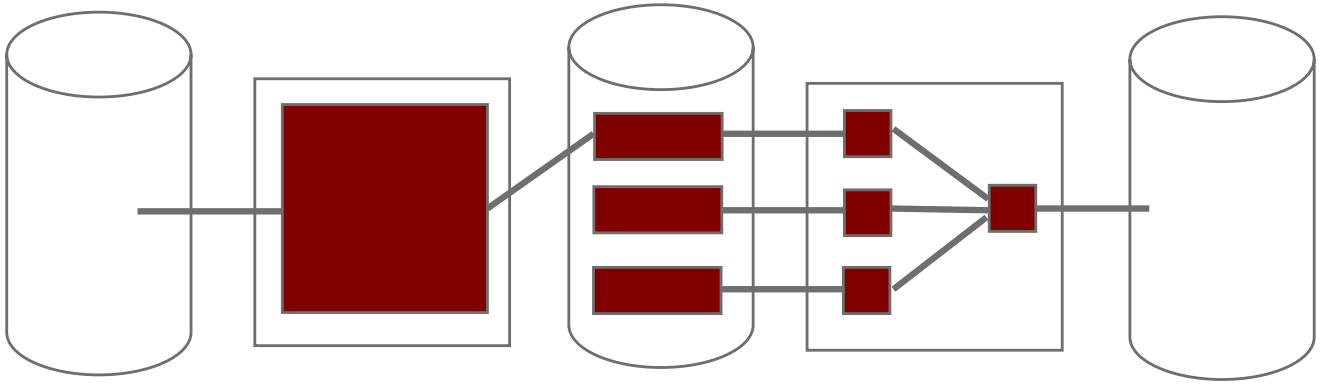
cost = 3*|R| IO's

Duality of External Hashing and External Sorting

External Hashing
cost = 3*|R| IO's



External Sorting
cost = 3*|R| IO's



Duality of Sorting and Hashing

- **Sorting**
  - Physical division, logical combination
    - Split followed by merge
    - Recurse on merging
  - Sequential write (phase 1), random read (phase 2)
  - Fan-in
  - If pipelining and sqrt(N) < B < N, Total Cost = 3N
- **Hashing**
  - Logical division, physical combination
    - Partition followed by concatenate
    - Recurse on partitioning
  - Random write (phase 1), sequential read (phase 2)
  - Fan-out
  - If pipelining and sqrt(N) < B < N, Total Cost = 3N

Source: G. Graefe, Query Evaluation Techniques for Large Databases. ACM Computing Surveys, Vol 25, No. 2, June 1993 (partial)

So which is better??

- 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 with hybrid hashing *(will not covered in this course)*

Relational Operators

- We now study implementation alternatives

- Select
- Project
- Join (equi-joins only)
- Set operations (union, intersect, except)
- Aggregation

Joins

- Joins are very common.

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

- Join techniques we will cover today:
  1. Nested-loops join
  2. Index-nested loops join
  3. Sort-merged join
  4. Hash join

Simple Nested Loops Join

R⋈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 = (pR*|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

R ⋈ S:

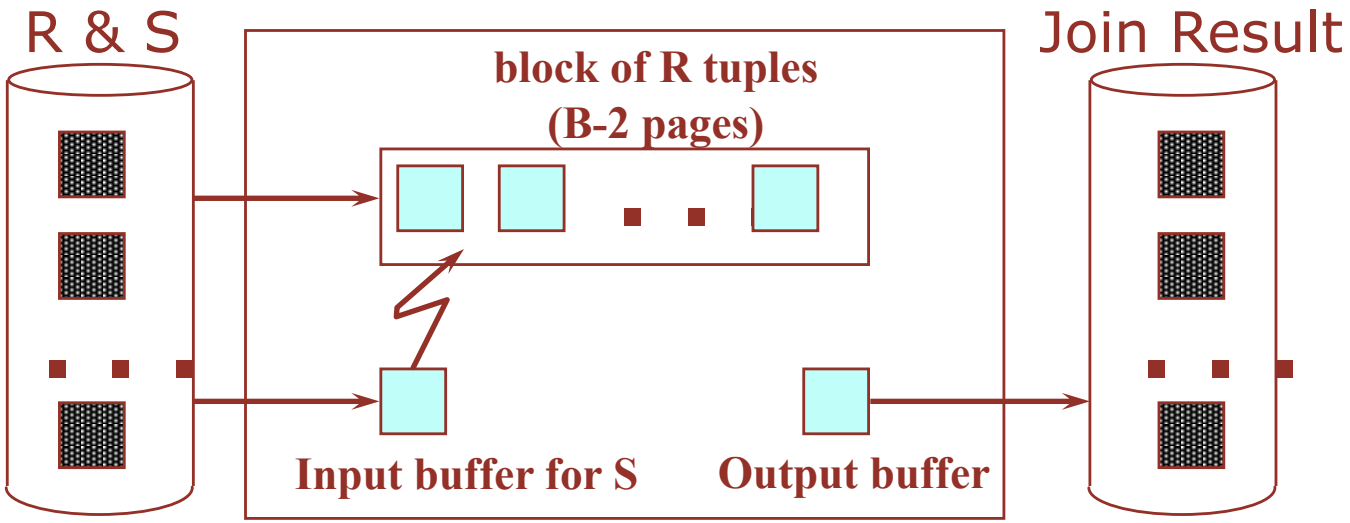
foreach page b_R in R do
  foreach page b_S in S do
    foreach tuple r in b_R do
    foreach tuple s in $b_S$do
      if r_i == s_j then add <r, s> to result

- Cost = |R|*|S| + |R| = 1000*500 + 1000
- If smaller relation (S) is outer, cost = 500*1000 + 500
- Much better than naïve per-tuple approach!
  - At 10ms/IO, total time ~ 1.4 hour
- The trick is to reduce the # complete reads of the inner table

Can we reduce it even further?

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 = $\lceil \#\ of\ pages\ of\ outer\ /\ blocksize \rceil$

Examples of Block Nested Loops Join

- Say we have B = 100+2 memory buffers
- Join cost = |outer| + (#outer blocks * |inner|)
  - #outer blocks = |outer| / 100

- 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

- 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

Index Nested Loops Join

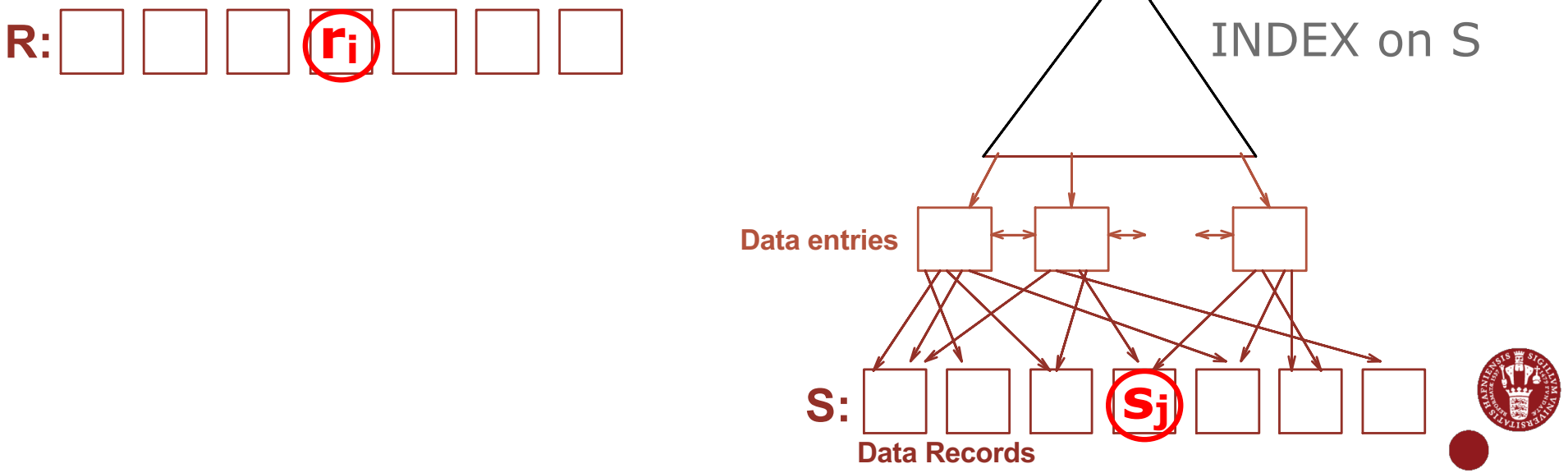
R ⋈ S:

foreach tuple r in R do

~~foreach tuple s in S where r_i == s_j do~~

add <r, s> to result

lookup(r_i)



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

Example:

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

Q: What if all the sid in the two tables are identical, and memory can only hold 2 records from each table at the same time

sid	sname	rating	age
22	dustin	7	45.0
22	yuppy	9	35.0
22	lubber	8	55.5
22	guppy	5	35.0
22	rusty	10	35.0

sid	bid	day	rname
22	103	12/4/96	guppy
22	103	11/3/96	yuppy
22	101	10/10/96	dustin
22	102	10/12/96	lubber
22	101	10/11/96	lubber
22	103	11/12/96	dustin

Cost of Sort-Merge Join

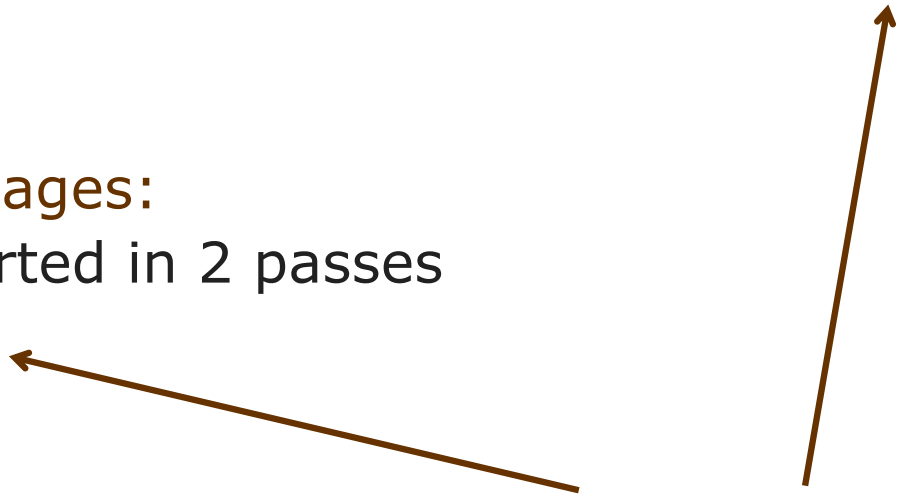
- Cost: Sort R + Sort S + (|R|+|S|)
  - But in the worst case, last term could be |R|*|S| *(very unlikely!)*
  - Q: what is worst case?

Suppose B = 35 buffer pages:

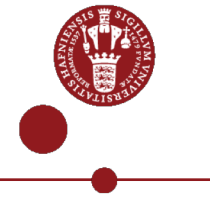
- Both R and S can be sorted in 2 passes
- Total join cost = 4*1000 + 4*500 + (1000 + 500) = 7500

Suppose B = 300 buffer pages:

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



Block-Nested-Loop cost = 2500 … 15,000



Other Considerations …

- An important 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 ⋈ S matches

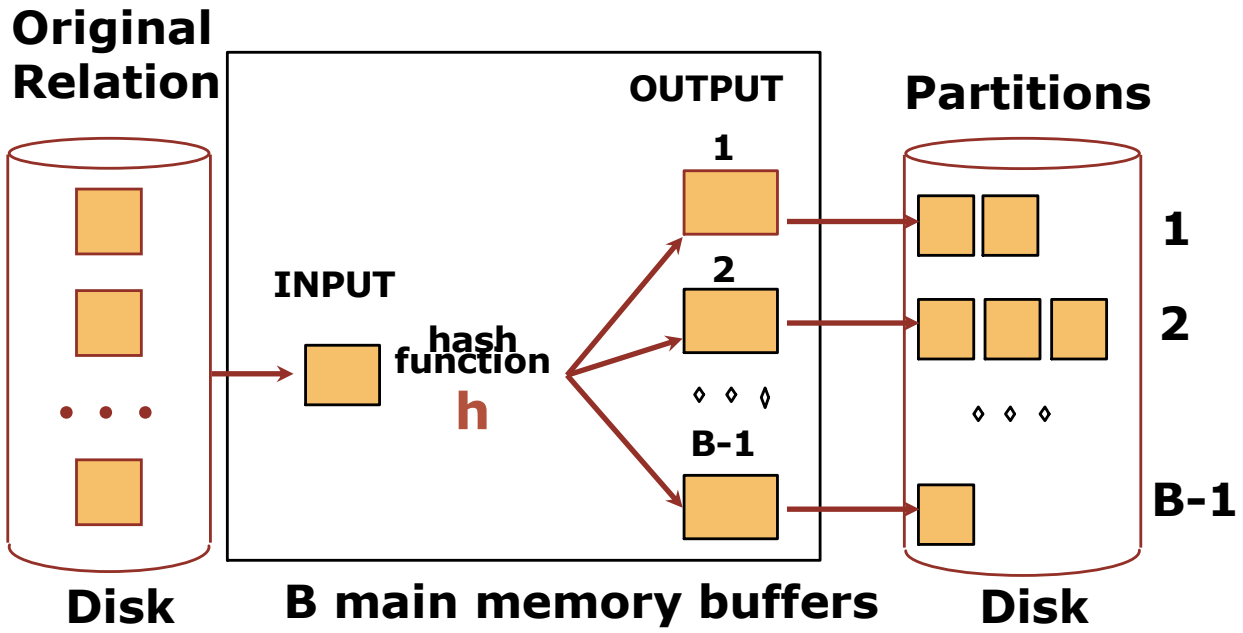
  Cost = 3*|R| + 3*|S|

  Q: how much memory is "enough"

- 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)

GRACE Hash Join

- Partition both relations using hash fn h: R tuples in partition i will only match S tuples in partition i.
- R join S = R_1 join S_1 U … U R_{B-1} join S_{B-1}

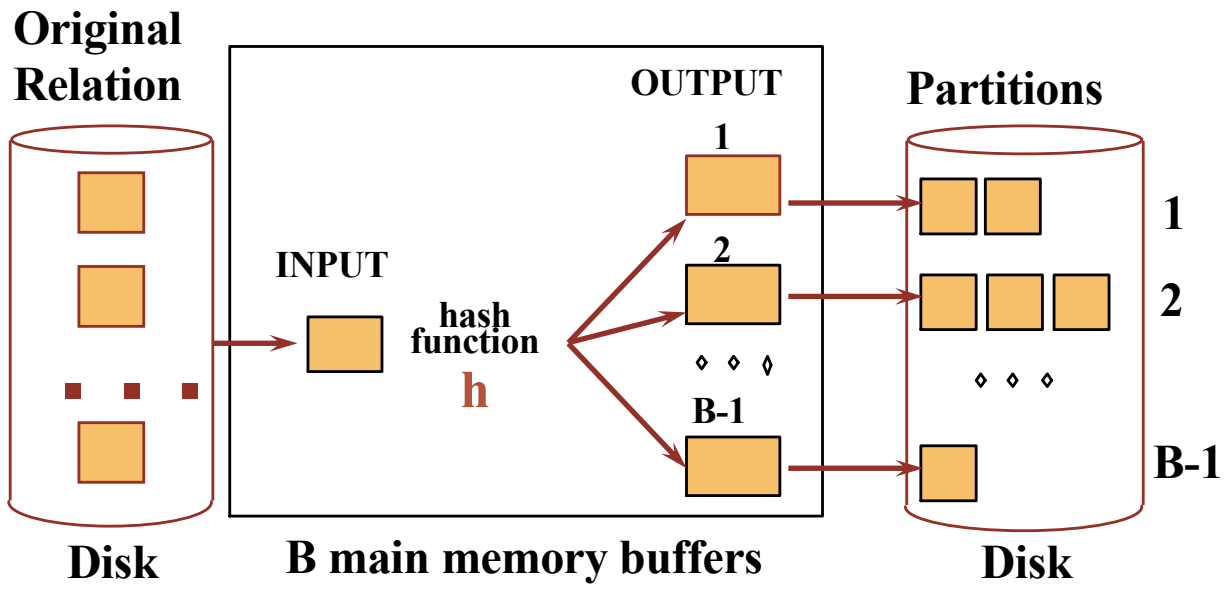


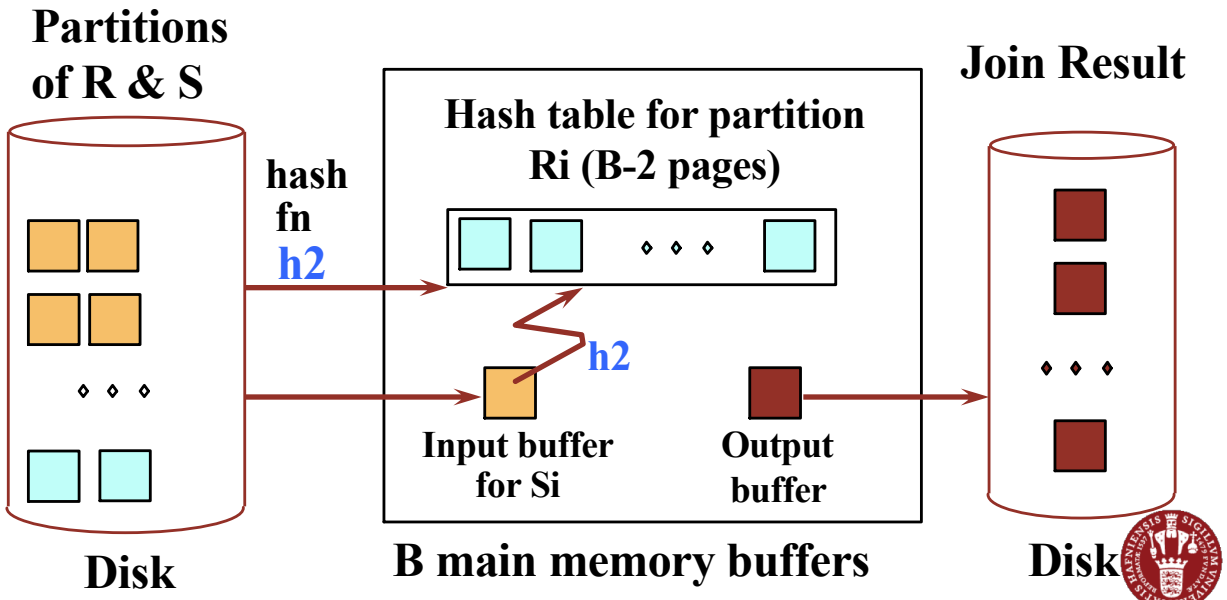
Grace Hash Join

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

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

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





2-Pass Hash Join vs. 2-Pass Sort-Merge Join

- Given a minimum amount of memory, both have cost of 3 (M + N)

- Benefits of hash join
  - Superior if relation sizes differ greatly
  - **Refinement**: hybrid hash join allows for dynamically adjusting to smaller relation fitting in main memory
  - Highly parallelizable

- Benefits of sort-merge join
  - Less sensitive to data skew
  - Result is sorted

Relational Operators

- We now study implementation alternatives

- Select
- Project
- Join
- Set operations (union, intersect, except)
- Aggregation

Set Operations

- Intersection and cross-product special cases of join.
- Union (Distinct) and Except similar; we'll do union.
- **Sorting based approach to union:**
  - Sort both relations (on combination of all attributes).
  - Scan sorted relations and merge them.
  - Alternative: Merge runs from Pass 0 for both relations.
- **Hash based approach to union:**
  - Partition R and S using hash function h.
  - For each S-partition, build in-memory hash table (using h2), scan corresponding R-partition and add tuples to output while discarding duplicates.

Relational Operators

- We now study implementation alternatives

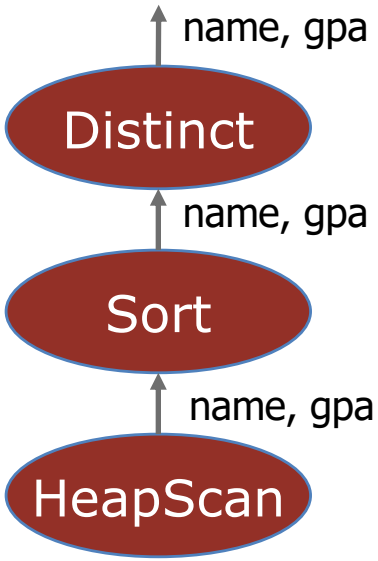
- Select
- Project
- Join
- Set operations (union, intersect, except)
- Aggregation

Homework! ☺

Query Execution Framework

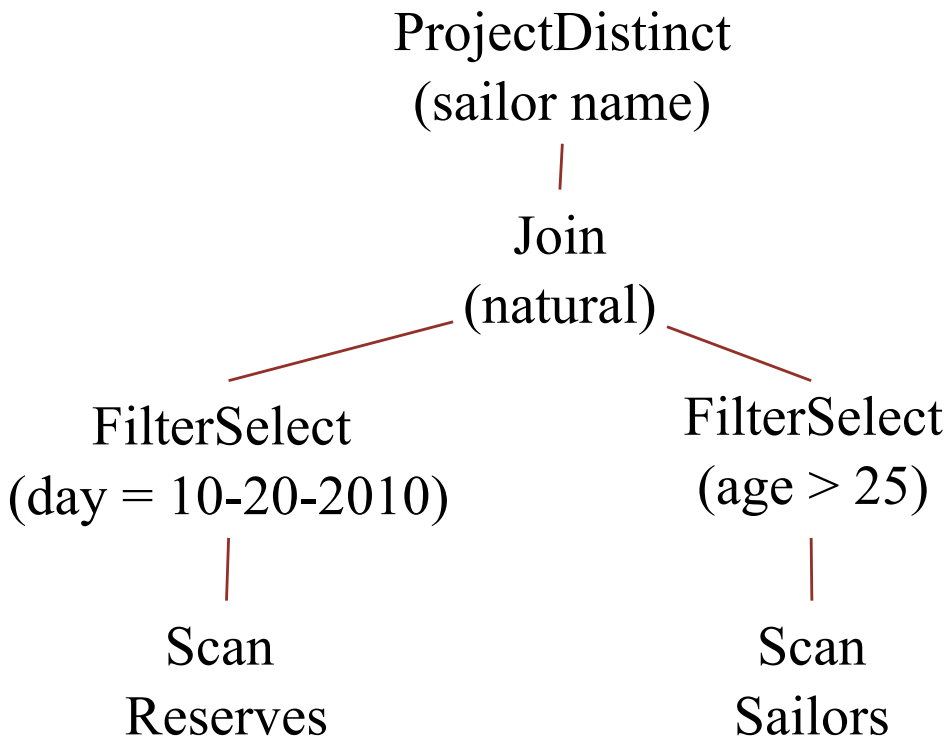
```
SELECT DISTINCT name, gpa
FROM Students
```

One possible query execution plan:



Design Goals for Operator Interface

- Must be able to compose several operators together into operator tree
- Must coordinate how data is passed between operators
- Should only buffer necessary data in main memory



Pull Model

- User requests one tuple at a time from top-level operator
- Operators calculate next tuple by requesting tuples from their input operators

ONC-Interface (iterator)

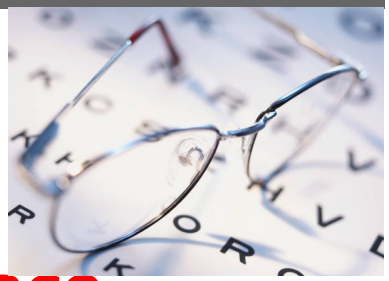
- **Open**
  - Initializes the operator
- **Next**
  - Calculates the next tuple and returns it to caller
- **Close**
  - Cleans up and closes operator

Example: Basic Selection (FilterSelection)

```
FilterSelection {
        Iterator input;
        Predicate P;
        …
        public void open() { input.open(); }

        public Tuple next() {
                Tuple result = null;
                do {
                        result = input.next();
                } while ( !P(result) )
                return result;
        }

        public void close() { input.close(); }
}
```



What should we learn today?

- Discuss the design of a **pull-based interface** for data processing operators, and how such an organization helps with composability and pipelining
- Explain and reason about the **implementation of physical relational operators**, including selections, projections, joins, set operations, and aggregation
- Explain simple **loop-based implementations** to relational operators and techniques such as use of blocks and indices to improve their performance
- Discuss the **duality of hashing and sorting** and how these algorithmic approaches apply to the implementation of relational operators