



Evaluating Relational Operations: Part I



Relational Operators

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



Select Operator

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

Select Operator



- ❖ Three cases
- ❖ 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



- ❖ Assume that select operator is applied over a relation with N tuples stored in P data pages
- ❖ What is the cost of select operation in this case (in terms of # I/Os)?

Select Operator



- ❖ Three cases
- ❖ 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

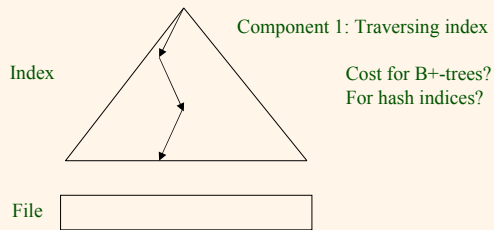
Case 2: Example



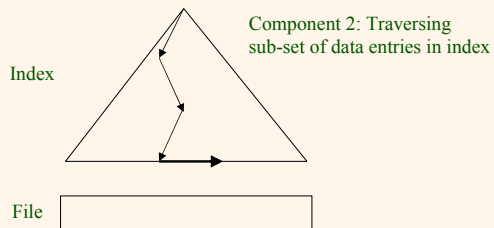
```
SELECT *  
FROM   Sailor S  
WHERE  S.Age = 25 AND S.Salary > 100K
```

❖ Have B+-tree index on (Age, Salary)

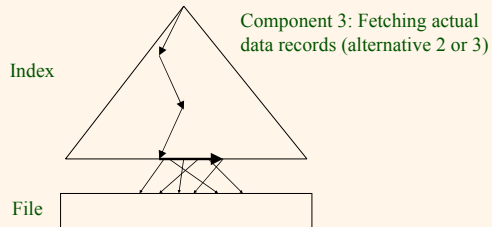
Case 2: Cost Components



Case 2: Cost Components



Case 2: Cost Components



Cost of Component 1

- ❖ D is cost of reading/writing one page to disk (using random disk I/O)
- ❖ Hash index
 - Cost = D
- ❖ B+-tree
 - Cost = D * (height of tree)

Cost of Component 2

- ❖ N data entries (= # data tuples if alternative 2)
- ❖ Hash index
 - Linear hashing
 - B hash buckets
 - Average cost = $D * (N/B - 1)$
- ❖ B+ tree index
 - L = average number of entries per leaf page
 - S = **Selectivity** (fraction of tuples satisfying selection)
 - Average cost = $D * ((S * N/L) - 1)$

Cost of Component 3



- ❖ $S \cdot N$ data entries satisfy selection condition
 - S is selectivity, N is total number of data entries
- ❖ T is number of data tuples per page
- ❖ Hash index
 - Worst-case cost = $D \cdot S \cdot N$ (if unclustered index)
 $D \cdot S \cdot N / T$ (if clustered index)
- ❖ B+ tree index
 - Worst-case cost = Same as hash index

Putting it all together



- ❖ Total cost of select operations using unclustered B+ tree index
 - $D \cdot (\text{Height} + (S \cdot N / L - 1) + S \cdot N)$
- ❖ Should we always use index in this case?
 - Depends on selectivity of selection condition!
 - $D \cdot (\text{Height} + (S \cdot N / L - 1) + S \cdot N) < D \cdot P$
 - $S < (P - \text{Height} + 1) \cdot L / (N(L + 1))$
 - Simple optimization!
- ❖ What about a clustered index?

Component 3: Optimization



- ❖ Alternative 2 or 3, unclustered index
- ❖ Find qualifying data entries from index
- ❖ **Sort** the rids of the data entries to be retrieved
 - Remember rid = (page ID, slot #)
- ❖ Fetch rids in order
 - Ensures each data page is read from disk just once!
 - Although number of data pages retrieved still likely to be more than with clustering

Select Operator



- ❖ Three cases
- ❖ 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 3: Example



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

- ❖ Have Hash index on Age

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 better depending on selectivity

Case 3: Example



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

- ❖ Have Hash index on Age
- ❖ Have B+ tree index on Salary

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 of data entries
 - Apply remaining predicates on retrieved tuples
 - Return those tuples that satisfy all predicates

Evaluation Alternatives



- ❖ Alternative 2
 - Get rid of data records using each index
 - Use index on Age and index on Salary
 - **Intersect the rids**
 - We'll discuss intersection soon
 - Retrieve the tuples corresponding to the rids
 - Apply remaining predicates on retrieved tuples
 - Return those tuples that satisfy all predicates
- ❖ Alternative 3
 - Sequential scan!

Relational Operators



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

Example



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

Evaluation Alternatives



- ❖ Alternative 1
 - Using Indices
- ❖ Alternative 2
 - Based on sorting
- ❖ Alternative 3
 - Based on hashing

Example



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

- ❖ Have B+ tree index on (Name, Age)

Evaluation Using “Covering” Index



- ❖ Simply scan leaf levels of index structure
 - No need to retrieve actual data records
 - **Index-only scan**
- ❖ Works so long as the index search key includes all the projection attributes
 - Extra attributes in search key are okay
 - Best if projection attributes are prefix of search key
 - Can eliminate duplicates in single pass of index-only scan
- ❖ Other examples
 - Hash index on (SSN, Name, Age)
 - B+ tree index on (Age, # Dependents, Name)

Example



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

- ❖ Have Hash index on Name
- ❖ Have B+ tree index on Age
- ❖ Sailor relation has 100 other attributes!

Evaluation Using RID Joins



- ❖ Retrieve (SearchKey1, RID) pairs from first index
- ❖ Retrieve (SearchKey2, RID) pairs from second index
- ❖ **Join** these based on RID to get (SearchKey1, SearchKey2, RID) triples
 - We will discuss joins soon!
- ❖ Project out the third column to get the desired result

Evaluation Alternatives



- ❖ Alternative 1
 - Using Indices
- ❖ Alternative 2
 - Based on sorting
- ❖ Alternative 3
 - Based on hashing

Example



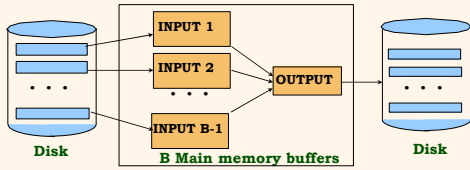
```
SELECT DISTINCT S.Name, S. Age
FROM   Sailor S
```

- ❖ No indices

General External Merge Sort

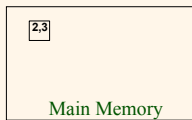
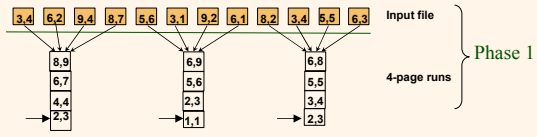
❖ Phase 2: Make multiple passes to merge runs

- Pass 1: Produce runs of length $B(B-1)$ pages
- Pass 2: Produce runs of length $B(B-1)^2$ pages
- ...
- Pass P: Produce runs of length $B(B-1)^P$ pages



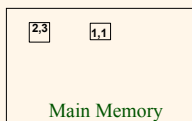
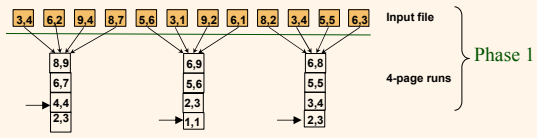
General External Merge Sort: Phase 2

❖ # buffer pages $B = 4$



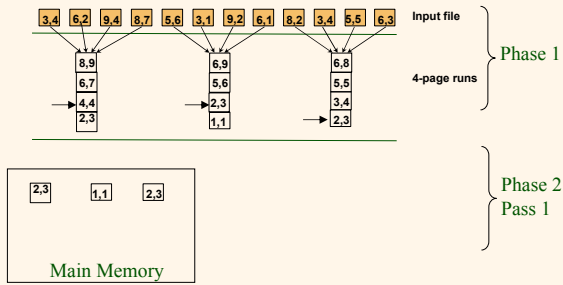
General External Merge Sort: Phase 2

❖ # buffer pages $B = 4$



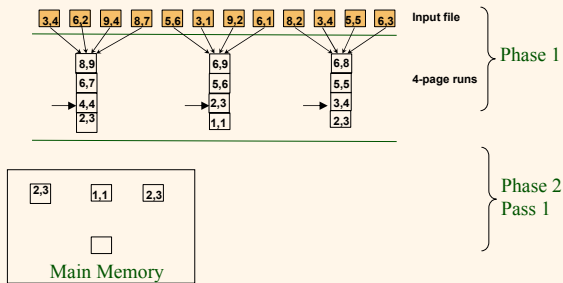
General External Merge Sort: Phase 2

❖ # buffer pages $B = 4$



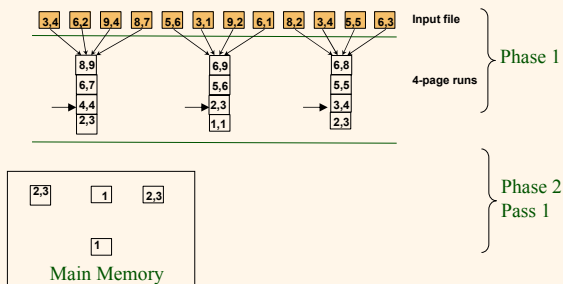
General External Merge Sort: Phase 2

❖ # buffer pages $B = 4$



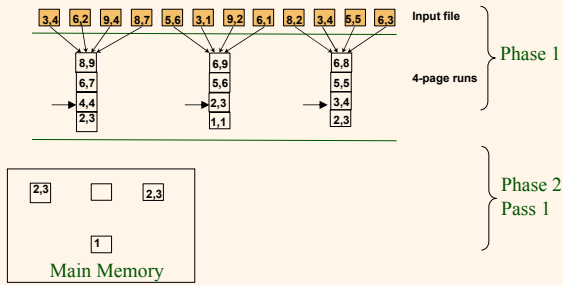
General External Merge Sort: Phase 2

❖ # buffer pages $B = 4$



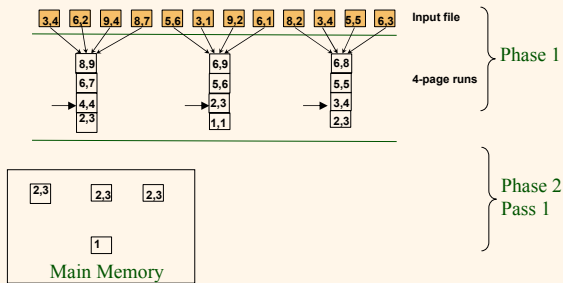
General External Merge Sort: Phase 2

❖ # buffer pages $B = 4$



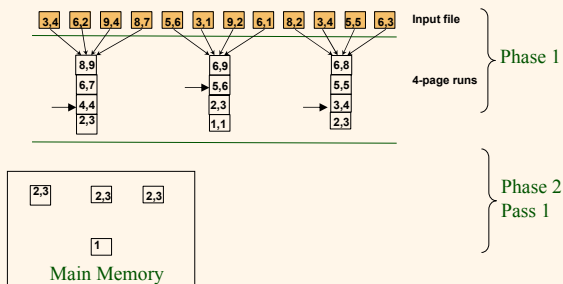
General External Merge Sort: Phase 2

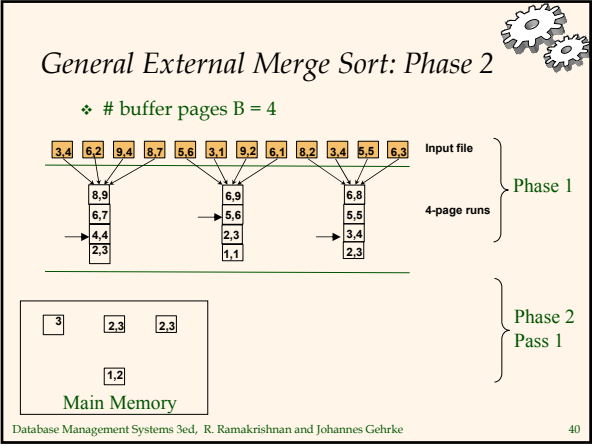
❖ # buffer pages $B = 4$

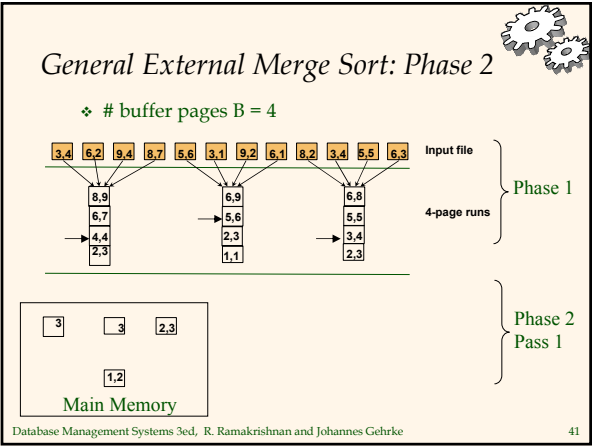


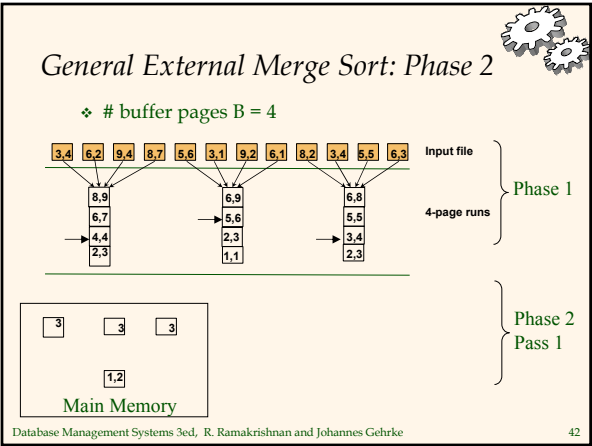
General External Merge Sort: Phase 2

❖ # buffer pages $B = 4$



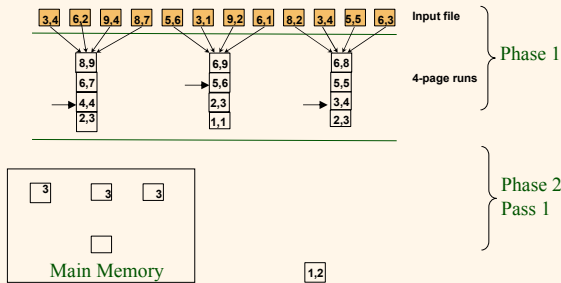






General External Merge Sort: Phase 2

❖ # buffer pages $B = 4$



Modifications to External Sorting

❖ Phase 1

- Project out unwanted columns
- Still produce runs of length B (or $2B$) pages
- But tuples in runs are smaller than input tuples (so smaller runs)

❖ Phase 2

- Eliminate duplicates during merge
- Smaller runs

❖ Exercise: Calculate I/O cost assuming certain size of projection columns and certain distribution of duplicates

Evaluation Alternatives

❖ Alternative 1

- Using Indices

❖ Alternative 2

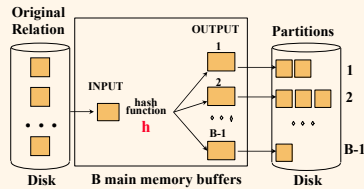
- Based on sorting

❖ Alternative 3

- Based on hashing

Projection Based on Hashing

- ❖ Assume relation does not fit in memory
- ❖ Phase 1
 - Divide relation into partitions
 - No duplicate elimination yet!



Phase 1: Analysis

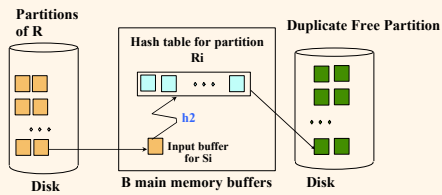
- ❖ Number of data pages = N
 - Assume all attributes are projected out
- ❖ Cost of reading/writing disk page = D
- ❖ Number of Partitions = $B-1$
- ❖ Length of each partition = $N/(B-1)$
- ❖ Cost of Phase 1 = $2 * D * N$

Two Cases for Each Partition

- ❖ Case 1
 - Partitions fits in memory
- ❖ Case 2
 - Partition does not fit in memory

Case 1: Partition Fits in Memory

- ❖ Use $h_2 \leftrightarrow h_1$!



- ❖ R is number of pages in result
 - After eliminating duplicates
- ❖ Cost = $D * (N + R)$

Case 2: Partition Doesn't fit in Memory

- ❖ Recursively apply Phase 1 algorithm on the partition!
 - Use hash function $h_2 \leftrightarrow h_1$!
- ❖ Analysis
 - Size of each partition after P partitioning phases = $N / (B-1)^P$
 - Stop partitioning when : $N / (B-1)^P = B-1$
 - # Partitioning phases = $\log_{B-1}(N) - 1$
 - Total cost of Phase 1 = $2 * D * (\log_{B-1}(N) - 1)$

Comments on Projection

- ❖ Sort-based approach vs. hash-based approach
 - Which one would you choose?
 - Why?
- ❖ Sort-based approach!
 - Better handling of skew
 - Results in sorted order