ADB - Module 3

Query Evaluation and External Sorting

Introduction

- In this module, we present an overview of how queries are evaluated in an RDBMS.
- We begin with a discussion of how a DBMS describes the data that it manages, including tables and indexes.
- This **descriptive data**, or **metadata**, stored in **special tables** called the ***system catalogs***, is used to find the best way to evaluate a query.
- **SQL queries** are **translated** into an **extended form of relational algebra**, and **query evaluation plans** are **represented** as **trees of relational operators**, along with **labels** that **identify the algorithm to use** at each node.
- Thus, **relational operators** serve as **building blocks** for **evaluating queries**, and the implementation of these operators is carefully optimized for good performance.
- In general, queries are composed of several operators, and the algorithms for individual operators can be combined in many ways to evaluate a query.
- The **process** of **finding a good evaluation plan** is called ***query optimization***.
- We consider a number of example queries using the following schema:

  Sailors(*sid:* `integer`, *sname:* `string`, *rating:* `integer`, *age:* `real`)
  Reserves(*sid:* `integer`, *bid:* `integer`, *day:* `dates`, *rname:* `string`)

- We assume that **each tuple** of **Reserves** is **40 bytes** long, that a **page can hold 100** Reserves **tuples**, and that we have **1000 pages** of such tuples.
- Similarly, we assume that **each tuple** of **Sailors** is **50 bytes** long, that a **page can hold 80** Sailors **tuples**, and that we have **500 pages** of such tuples.

---

1. The System Catalog

- An RDBMS maintains **information about** every **table** and **index** that it contains.
- The **descriptive information** is itself **stored** in a **collection** of **special tables** called the ***catalog tables***. An example of a catalog table is shown in the next page. The **catalog tables** are **also called** the
  - ***data dictionary***
  - the ***system catalog***, or
  - **simply** the ***catalog***.

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	Sailors	integer	1
sname	Sailors	string	2
rating	Sailors	integer	3
age	Sailors	real	4
sid	Reserves	integer	1
bid	Reserves	integer	2
day	Reserves	dates	3
rname	Reserves	string	4

- The above figure shows the tuples in the Attribute_Cat table that describe the attributes of these 2 tables.
- Note that in addition to the tuples describing Sailors and Reserves, other tuples (the first 4 listed) describe the 4 attributes of the Attribute_Cat table itself!
- The **catalog tables describe all the tables** in the database, **including** the catalog tables **themselves**.
- When information about a table is needed, it is obtained from the system catalog.
- The fact that the system catalog is also a collection of tables is very useful. For example, catalog tables can be queried just like any other table, using the query language of the DBMS!
- Further, all the techniques available for implementing and managing tables apply directly to catalog tables.
- The choice of catalog tables and their schemas is not unique and is made by the implementor of the DBMS.

---

2. Introduction to Operator Evaluation

- Several alternative algorithms are available for implementing each relational operator, and for most operators no algorithm is universally superior.
- We now describe some common techniques used in developing evaluation algorithms for relational operators, and introduce the concept of **access paths**, which **are the different ways in which rows** of a **table** can be **retrieved**.

2.1 Three Common Techniques

- The algorithms for various relational operators actually have a lot in common. A few simple techniques are used to develop algorithms for each operator:
  - **Indexing**: If a selection or join condition is specified, **use an index to examine** just the tuples that satisfy the condition.
  - **Iteration**:
    - **Examine all tuples** in an input table, one after the other.
    - If we need only a few fields from each tuple and there is an index whose key contains all these fields, instead of examining data tuples, we can scan all index data entries. (**index-only evaluation strategies**)
  - **Partitioning**: By **partitioning tuples on a sort key**, we can often decompose an operation into a less expensive collection of operations on partitions. ***Sorting*** and ***hashing*** are two commonly used partitioning techniques.

2.2 Access Paths

- **An access path is a way of retrieving tuples from a table** and **consists of either**
  - (1) **a file scan** or
  - (2) **an index plus a matching selection condition**
- Every relational operator accepts 1 or more tables as input, and the access methods used to retrieve tuples contribute significantly to the cost of the operator.
- Consider a **simple selection** that is a **conjunction of conditions of the form** attr op value, where op is one of the comparison operators. ($<, >, \leq, \geq, =, \neq$)
- **Such selections** are said to be in ***conjunctive normal form*** (**CNF**), and **each condition** (**attr op value**) is called a ***conjunct***.
- Intuitively, an ***index*** **matches** a ***selection condition*** if the index 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 $attribute = value$ 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 attribute op value for each attribute is a ***prefix*** of the index's search key. (**<a>** and **<a, b>** are **prefixes of key <a, b, c>** , but <a, c> and <b, c> are not.)

- An index can **match some subset of** the **conjuncts** in a selection condition (in CNF), even though it does not match the entire condition. We refer to the **conjuncts that the index matches** as the ***primary conjuncts*** in the selection.

The following examples illustrate access paths.

- If we have a **hash index** H on the search key $< rname, bid, sid >$, we can use the index to retrieve just the Sailors tuples that satisfy the condition $rname = 'Joe' \wedge bid = 5 \wedge sid = 3$, because the index matches the entire condition.
  On the other hand, if the selection condition is $rname = 'Joe' \wedge bid = 5$, or some condition on date, this index does not match. That is, a hash index cannot be used to retrieve just the tuples that satisfy these conditions.
  In contrast, if the index were a **B+ tree**, it would match both $rname = 'Joe' \wedge bid = 5 \wedge sid = 3$ and $rname = 'Joe' \wedge bid = 5$. However, it would not match $bid = 5 \wedge sid = 3$ (since tuples are sorted primarily by $rname$).
- If we have an index (hash or tree) on the search key $< bid, sid >$ and the selection condition $rname = 'Joe' \wedge bid = 5 \wedge sid = 3$, we can use the index to retrieve tuples that satisfy $\boldsymbol{bid = 5 \wedge sid = 3}$; these are the **primary conjuncts**. The fraction of tuples that satisfy these conjuncts (and whether the index is clustered) determines the number of pages that are retrieved. The **additional condition** on $\boldsymbol{rname}$ must **then** be **applied to each retrieved tuple** and will eliminate some of the retrieved tuples from the result.
- If we have **an index on** the search key $< \boldsymbol{bid}, \boldsymbol{sid} >$ **and** we also have a **B+ tree index on** $\boldsymbol{day}$, the **selection condition** $day < 8/9/2002 \wedge bid = 5 \wedge sid = 3$ **offers us a choice**. **Both indexes match** (a part of) **the selection condition**, and we **can use either to retrieve tuples**. **Whichever** index **we use**, the **conjuncts** in the selection condition **that are not matched** by the index (Eg: $bid = 5 \wedge sid = 3$ if we use the B+ tree index on day) **must be checked for each retrieved tuple**.

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 two access paths**: the **index** and a **scan of the data file**.
- **Sometimes**, of course, we can **scan the index itself**, giving us a **third access path**.

- The **most selective** access path is the one that **retrieves the fewest pages**; using the most selective access path minimizes the cost of data retrieval.
- The selectivity of an access path depends on the primary conjuncts in the selection condition (with respect to the index involved).
- **Each conjunct acts as a filter on the table**. The **fraction of tuples** in the table **that satisfy a given conjunct** is called the **reduction factor**.
- **When** there are **several primary conjuncts**, the **fraction of tuples that satisfy all** of them can be approximated by the **product of their reduction factors**.

---

3. Algorithms for Relational Operations

- We now briefly discuss evaluation algorithms for the main relational operators.
- Like in previous modules, we consider only I/O costs and measure I/O costs in terms of the number of page I/Os.

Relation **S**

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

Notations

- σ ⟶ ***select* rows** from a relation
- π ⟶ ***project* columns** from a relation
- Ex: $\sigma_{rating > 8}(S)$

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
58	Rusty	10	35.0

$\pi_{sname,rating}(S)$

sname	*rating*
yuppy	9
Lubber	8
guppy	5
Rusty	10

- The selection condition is a Boolean combination (i.e., an expression using the logical connectives ∧ and ∨) of terms that have the form attribute op constant or attribute1 op attribute2, where op is one of the comparison operators $\leq, =, \neq, \geq$ or $>$.
- Example: $\pi_{sname,rating}(\sigma_{rating>8}(S))$

sname	*rating*
yuppy	9
Rusty	10

3.1 Selection

- The selection operation is a simple retrieval of tuples from a table, and its implementation is essentially covered in our discussion of access paths.

- To summarize, **given a selection** of the form **σ** $_{\text{R.attr } op \text{ value}}$(R), **if there is no index on R.attr**, **we have to scan R**.
- If one or more indexes on R match the selection, we can use the index to retrieve matching tuples, and apply any remaining selection conditions to further restrict the result set.
- As an example, consider a selection of the form $rname < 'C\%'$ on the Reserves table. Assuming that names are uniformly distributed with respect to the initial letter, for simplicity, we estimate that roughly 10% of Reserves tuples are in the result. This is a total of 10,000 tuples, or 100 pages.
- If we have a **clustered B+ tree index** on the $rname$ field of Reserves, we can retrieve the qualifying tuples with **100 I/Os** (plus a few I/Os to traverse from the root to the appropriate leaf page to start the scan).
- However, **if the index is unclustered**, we could have **up to 10,000 I/Os** in the **worst case**, since each tuple could cause us to read a page.
- As a 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**.

3.2 Projection

- The projection operation requires us to drop certain fields of the input, which is easy to do.
- The expensive aspect is to ensure that no duplicates appear in the result.
- For example, if we only want the sid and bid fields from Reserves, we could have duplicates if a sailor has reserved a given boat on several days.
- **If duplicates need not be eliminated** (Eg: the **DISTINCT** keyword is **not included** in the **SELECT clause**), **projection consists of simply retrieving a subset of fields** from each tuple of the input table.
- This can be accomplished by **simple iteration on** either the **table or** an **index whose key contains all necessary fields**. (**Doesn't matter** if index is **clustered /unclustered**, **since** the **values we want are in** the **data entries** of the index **itself**!)
- If we have to eliminate duplicates, we typically have to use partitioning. Suppose we want to obtain sid, bid by projecting from Reserves. We can partition by
    - (1) **scanning** Reserves to obtain sid, bid pairs **and**
    - (2) **sorting** these pairs using sid, bid as the sort key.

- We can then **scan the sorted pairs** and easily **discard duplicates, which are now adjacent**.
- The **projection operation can be optimized by combining** the **initial scan** of Reserves **with the scan in the first pass of sorting**.
- **Similarly**, the **scanning of sorted pairs** can be **combined with** the **last pass of sorting**. With such an optimized implementation, **projection with duplicate elimination** requires
    - a **first pass** in which the **entire table is scanned**, and only **pairs** sid, bid are **written out**, and
    - a **final pass** in which **all pairs are scanned**, but **only one copy** of each pair is **written out**.
- In addition, there **might be** an **intermediate pass** in which **all pairs are read** from **and written to disk**.
- The **availability of appropriate indexes** can **lead to less expensive plans than sorting** for duplicate elimination. If we have an index whose **search key contains all the fields retained by** the **projection**, **we can sort** the **data entries** in the index, **rather than** the **data records** themselves.
- **If all** the **retained attributes appear in** a **prefix of** the **search key for** a ***clustered index***, we can do even better; we can **simply retrieve data entries using** the **index**, and **duplicates** are **easily detected since** they are **adjacent**. These plans are further examples of ***index-only evaluation strategies***, which we discussed in Module 1.

3.3 Join

- Joins are expensive operations and very common. Therefore, they have been widely studied, and systems typically support several algorithms to carry out joins.
- Consider the join of Reserves and Sailors, with the join condition $Reserves.sid = Sailors.sid$. Suppose that one of the tables, say Sailors, has an index on the sid column. We can scan Reserves and for each tuple, use the index to probe Sailors for matching tuples. This approach is called ***index nested loops join***.
- Suppose that we have a hash-based index using Alternative (2) on the sid attribute of Sailors and that it takes about **1.2 I/Os** on average **to retrieve** the **appropriate page** of the **index**.
- Since sid is a key (i.e., unique) for Sailors, we have at most one matching tuple.

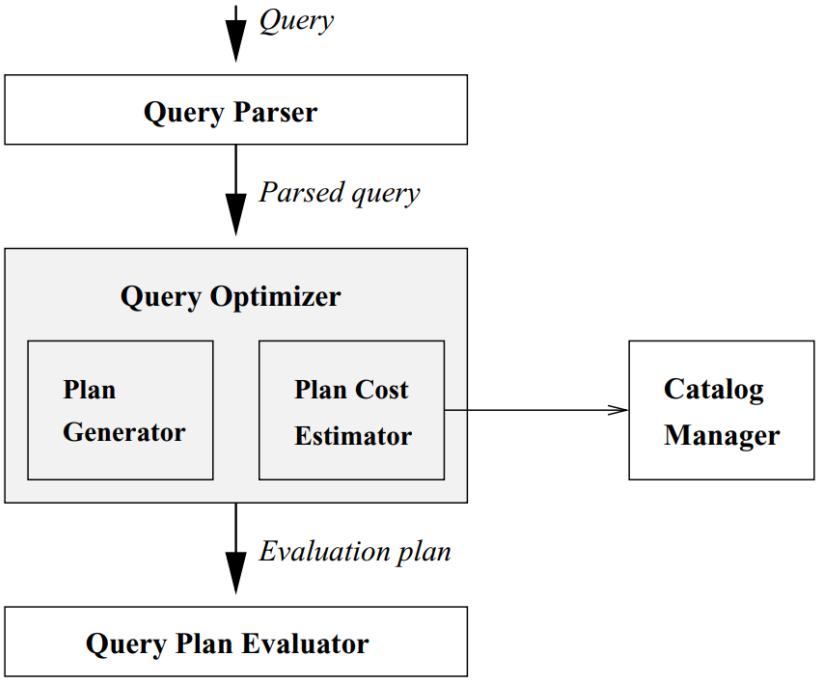
- Indeed, sid in Reserves is a *foreign key* referring to Sailors, and therefore we have **exactly 1 matching Sailors tuple for each Reserves tuple**.
- Let us consider the cost of scanning Reserves and using the index to retrieve the matching Sailors tuple for each Reserves tuple.
- The cost of scanning Reserves is 1000. There are $100 * 1000$ tuples in Reserves.
- For each of these tuples, retrieving the index page containing the rid of the matching Sailors tuple costs 1.2 I/Os; in addition, we have to retrieve the Sailors page containing the qualifying tuple.
- Therefore, we have $100{,}000 * (1 + 1.2)$ I/Os to retrieve matching Sailors tuples. The total cost is 221,000 I/Os.
- **If we do not have an index that matches** the **join condition on either table**, we **cannot use index nested loops**.
- In this case, we can **sort both tables on** the **join column**, and **then scan** them **to find matches**. This is called **sort-merge join**.
- Assuming that **we can sort Reserves in 2 passes**, and **Sailors in 2 passes as well**, let us consider the **cost of sort-merge join**.
- Consider the join of the tables Reserves and Sailors. Because we **read and write** (1+1=**2**) Reserves in each pass, the **sorting cost** is $\mathbf{2 * 2 * 1000 = 4000}$ **I/Os**. Similarly, **sorting cost of Sailors** is $\mathbf{2 * 2 * 500 = 2000}$ **I/Os**.
- In addition, the second phase of the sort-merge join algorithm requires an **additional scan of both tables**. Total cost is 4000 + 2000 + 1000 + 500 = **7500 I/Os**.
- Observe that the **cost of *sort-merge join***, **which does not require** a **pre-existing index**, is **lower than** the **cost of *index nested loops join***. In addition, the result of the sort-merge join is sorted on the join column.
- Index nested loops has the nice property that it is **incremental**. The cost of our example join is incremental in the number of Reserves tuples that we process.
- Therefore, **if** some **additional selection** in the query **allows** us to **consider only** a **small subset** of Reserves tuples, we **can avoid computing** the **join of Reserves and Sailors in** its **entirety**.
- For instance, suppose that we only want the **result of the join for $\boldsymbol{bid = 101}$**, and there are **very few such reservations**.
- Using *index nested loops join,* for each such Reserves tuple having $bid = 101$, **we probe Sailors using $\boldsymbol{sid}$**, and we are done.

- If we use **sort-merge join**, **on the other hand**, we **have to scan** the **entire Sailors table at least once**, and the cost of this step alone is likely to be much higher than the entire cost of *index nested loops join*.
- Observe that the choice of index nested loops join is based on considering the query as a whole, including the extra selection on Reserves, rather than just the join operation by itself.
- This leads us to our next topic, ***query optimization***, which is the process of finding a good plan for an entire query.

---

4. Introduction to Query Optimization

- Query optimization is one of the most important tasks of a relational DBMS.
- One of the strengths of relational query languages is the wide variety of ways in which a user can express and thus the system can evaluate a query.
- Although this flexibility makes it easy to write queries, **good performance relies greatly on** the **quality of** the **query optimizer**.
- A more detailed view of the query optimization and execution layer in the DBMS architecture from what we learnt in DBMS is shown below:

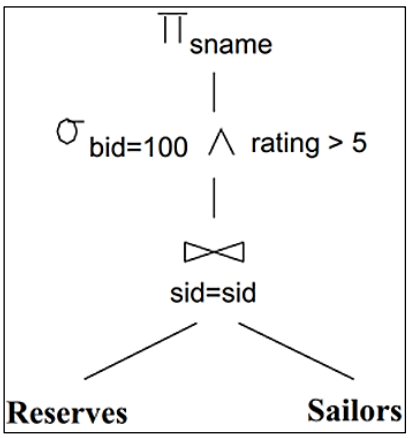


- Queries are parsed and then presented to a **query optimizer**, which is responsible for **identifying** an **efficient execution plan**.
- The **optimizer generates alternative plans** and **chooses** the **plan with** the **least estimated cost**.

- The **space of plans considered** by a typical relational query optimizer can be **understood by recognizing** that a **query** is essentially **treated as** a **σ – π– ⋈** ***algebra expression*** (where ⋈ represents joins)
- **Optimizing** such a **relational algebra expression involves 2 basic steps**:
  - Enumerating (list out) alternative plans for evaluating the expression. Typically, an **optimizer considers a subset of all possible plans** because the number of possible plans is very large.
  - **Estimating the cost** of each enumerated plan and **choosing** the **plan with** the **lowest estimated cost**.

4.1 Query Evaluation Plans

- A **query evaluation plan consists** of an **extended relational algebra tree**, with **additional annotations/labels at each node indicating** the **access methods** to use **for each table** and the **implementation method** to use **for each relational operator**.



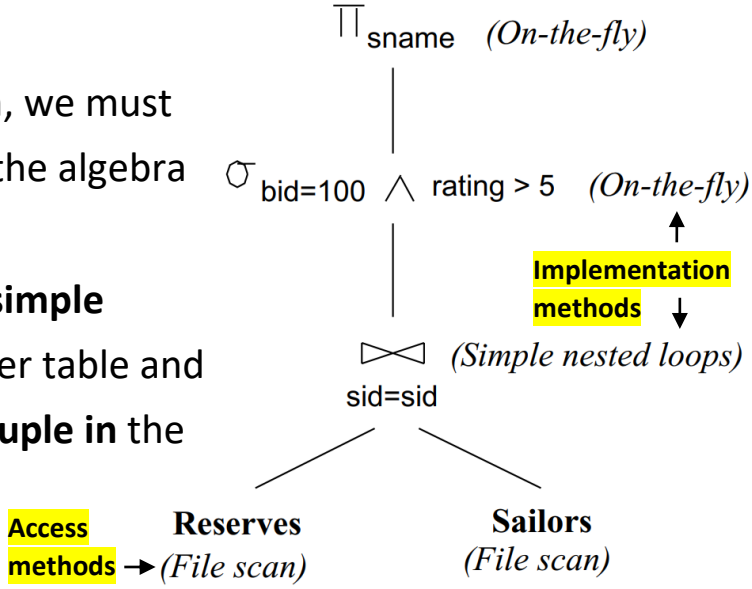
- Consider the following SQL query:

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

- This query can be **expressed in relational algebra** as follows:

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

- This expression is shown in the form of a tree above:
- The algebra expression partially specifies how to evaluate the query—we first compute the natural join of Reserves and Sailors, then perform the selections, and finally project the *sname* field.
- **To obtain** a **fully specified evaluation plan**, we must **decide** on an **implementation** for each of the algebra operations involved.
- For example, we can use a page-oriented **simple *nested loops join*** with Reserves as the outer table and **apply selections** and **projections to each tuple in** the **result of the join as it is produced**.
- This **query evaluation plan** is as shown.

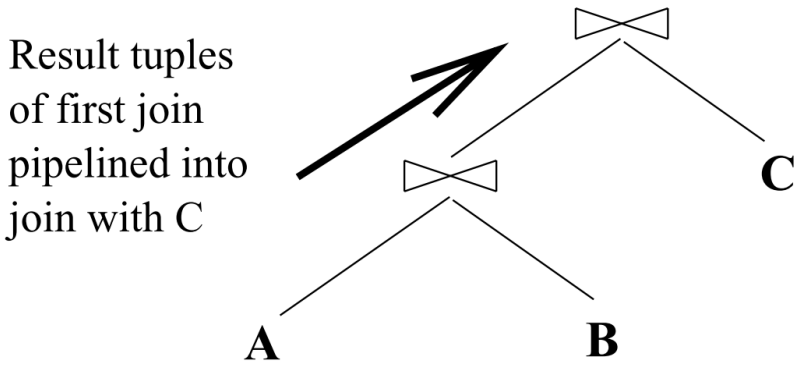


- In drawing the query evaluation plan, we have used the convention that the ***outer table*** is the *left child* of the *join* operator. We adopt this convention henceforth.

4.2 Multi-operator Queries: Pipelined Evaluation

- When a **query** is **composed of several operators**, the **result of one operator** is sometimes **pipelined to another operator without creating a temporary table** to hold the intermediate result.
- The query evaluation plan shown previously, pipelines the output of the join of Sailors and Reserves into the selections and projections that follow.
- **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.
- **If** the **output of** an **operator** is **saved in a temporary table 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.
- There are many opportunities for pipelining in typical query plans, even simple plans that involve only selections.
- Consider a **selection query** in which **only part of** the **selection condition matches** an **index**. We can think of such a query as containing **2 instances of the selection operator**:
    - the **first** contains the **primary**, or **matching, part** of the original selection condition, and
    - the second contains the **rest of** the **selection condition**.
- We can evaluate such a query by applying the primary selection and writing the result to a temporary table (TT) and then applying the second selection to the TT.
- In contrast, a **pipelined evaluation** consists of **applying** the **second selection to each tuple in** the **result of** the **primary selection *as it is produced*** and **adding tuples that qualify to** the **final result**.
- **When** the **input table to** a **unary operator** (e.g., **selection** or **projection**) is **pipelined into it**, **we** sometimes **say** that the **operator is applied *on-the-fly***.

- As a second and more general example, consider a join of the form (A ⋈ B) ⋈ C, as shown below as a tree of join operations.



Simple Nested Join: It is a tuple-at-a-time nested loops evaluation. We scan the outer relation R, and **for each tuple r ∈ R, we scan the entire inner relation S**.

Index Nested Join: If there is an index on one of the relations on the join attribute(s), we can take advantage of the index by making the indexed relation be the inner relation. **For each tuple r ∈ R, we use the index to retrieve matching tuples of S**.

- **Conceptually**, the **evaluation** is **initiated from** the **root**, and node joining A and B (**A ⋈ B**) **produces tuples as and when** they are **requested by** its **parent node**.
- **When** the **root node gets** a ***page of tuples*** **from** its **left child** (the outer table), **all matching inner tuples** are **retrieved** (**using** either an **index or** a **scan**) **and joined with matching outer tuples**; the **current page** of outer tuples is **then discarded**. A **request** is **then made to** the **left child for** the **next page of tuples**, and the **process** is **repeated**.
- **Pipelined evaluation** is thus a **control strategy governing** the **rate at which different joins in** the **plan proceed**.
- It has the *great virtue* of *not writing* the *result of intermediate joins* to a *temporary file* because the *results* are *produced*, *consumed*, and *discarded 1 page at a time*.

4.3 The Iterator Interface

- A query evaluation plan is a tree of relational operators and is executed by calling the operators in some (possibly interleaved) order.
- Each operator has one or more inputs and an output, which are also nodes in the plan, and tuples must be passed between operators according to the plan's tree structure.
- To **simplify the code responsible for coordinating the execution of a plan**, the **relational operators that form the nodes** of a plan tree (which is to be evaluated using pipelining) typically **support** a ***uniform iterator interface***, **hiding** the **internal implementation details of** each **operator**.
- The iterator interface for an operator includes the functions $\boldsymbol{open}$, $\boldsymbol{get\_next}$, and $\boldsymbol{close}$.

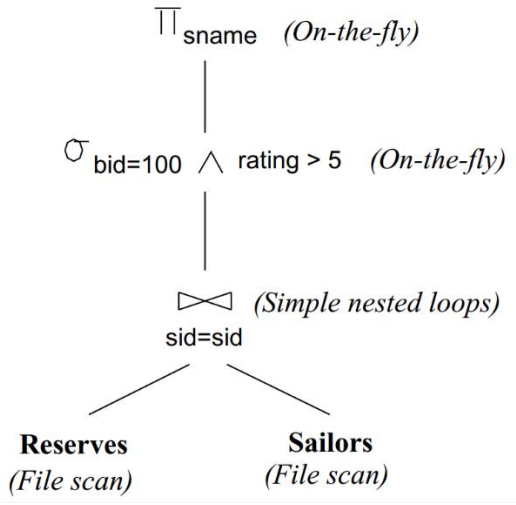
- The ***open*** function **initializes** the **state of** the **iterator by allocating buffers for** its **inputs** and **output**, and is also used to **pass** in **arguments** such as **selection conditions that modify** the **behavior of** the **operator**.
- The code for the ***get_next*** function **calls** the ***get_next*** function **on each input node and** calls **operator-specific code** to **process** the **input tuples**.
- The **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.
- **When all output tuples** have been **produced through repeated calls to *get_next***, the ***close*** function is called (by the code that initiated execution of this operator) to **deallocate state information**.
- The iterator interface supports pipelining of results naturally.
- The **decision to pipeline or materialize** input tuples **is encapsulated in** the **operator-specific code** that processes input tuples.
- **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.

---

5. Alternative Plans: A Motivating Example

- Consider the example query and the corresponding query evaluation plan as shown.

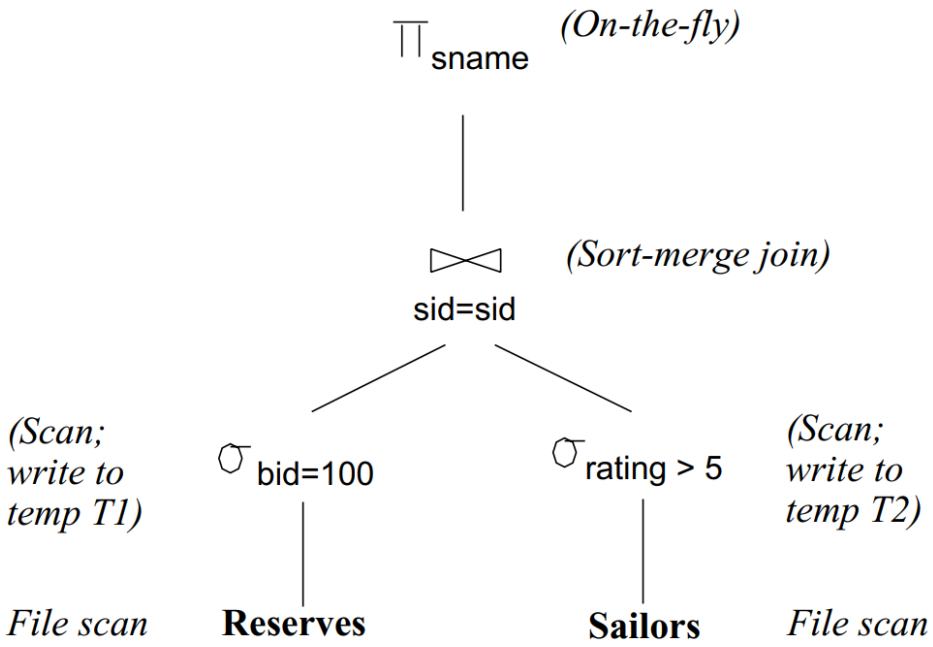
```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
        AND R.bid = 100 AND S.rating > 5
```



- Let us consider the cost of evaluating this plan.
- The cost of the join is 1000 + 1000 ∗ 500 = 501,000 page I/Os. (***simple nested join***)
- The selections and projection are done on-the-fly and do not incur additional I/Os.
- Total cost of this plan is therefore 501,000 page I/Os. **This plan is admittedly naïve**.
- We now consider several alternative plans for evaluating this query. Each alternative improves on the original plan in a different way and introduces some optimization ideas that are examined in more detail

5.1 Pushing Selections

- A join is a relatively expensive operation, and a **good heuristic** is to **reduce** the **sizes of the tables to be joined as much as possible**.
- One approach is to **apply selections early**; if a selection operator appears after a join operator, it is worth examining whether the selection can be 'pushed' ahead of the join.
- As an example, the selection $bid = 100$ involves only the attributes of Reserves and can be applied to Reserves before the join.
- Similarly, the selection $rating > 5$ involves only attributes of Sailors and can be applied to Sailors before the join.
- Let us **suppose** that the **selections** are **performed using** a simple **file scan**, that the **result of each selection** is **written to a temporary table** on disk, and that the **temporary tables** are then **joined using** a ***sort-merge join***.
- The resulting query evaluation plan is as shown:



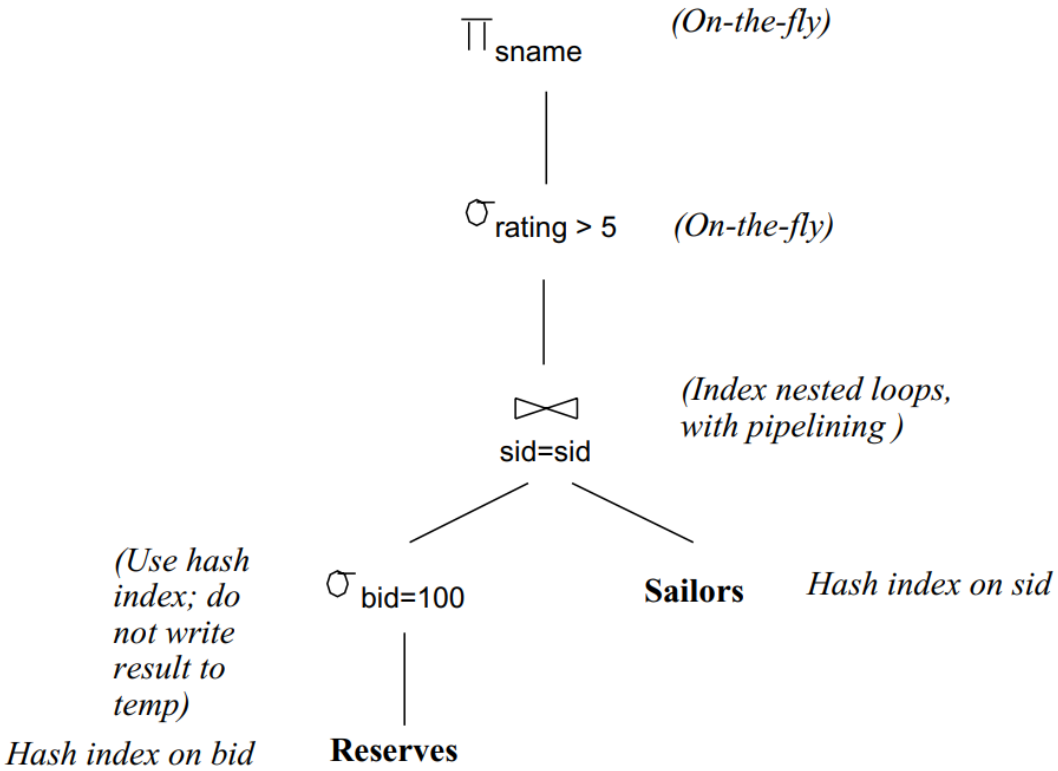
- Let us assume that 5 buffer pages are available and estimate the cost of this query evaluation plan.
- The cost of applying $bid = 100$ to Reserves is the cost of scanning Reserves (1000 pages) plus the cost of writing the result to a temporary table, say T1.
- Let's assume, that there are 100 boats and reservations are spread out uniformly across all boats and number of pages in T1 is therefore 10 (1000 pages ÷ 100 boats)
- The cost of applying rating > 5 to Sailors is the cost of scanning Sailors (500 pages) plus the cost of writing out the result to a temporary table, say, T2.

- If we assume that ratings are uniformly distributed over the range 1 to 10, we can approximately estimate the size of T2 as 250 pages.
- To do a sort-merge join of T1 and T2, let us assume that a straightforward implementation is used in which the two tables are first completely sorted and then merged. (**Number of passes is calculated using External Merge Sort formula**)
- Since 5 buffer pages (B) are available, we can sort T1 (which has 10 pages (N)) in 2 passes. 2 runs of 5 pages each are produced in the 1st pass and these are merged in the 2nd pass. ($N1 = 10/5 = 2,\ \#ofPasses = (\log_{B-1} N1) + 1 = \log_4 2 + 1 \approx 2$)
- In each pass, we read and write 10 pages; thus, the cost of sorting T1 is 2 (read+write) $*$ 2 (passes) $*$ 10 = 40 page I/Os.
- We need 4 passes to sort T2, which has 250 pages. The cost is $2 * 4 * 250 = 2000$ page I/Os. ($N1 = 250/5 = 50,\ \#ofPasses = \log_{B-1} N1 + 1 = \log_4 50 + 1 \approx 4$)
- **To merge** the **sorted versions of T1 and T2**, we need to **scan these tables**, and the **cost** of this step is 10 + 250 = 260.
- The final projection is done on-the-fly, and by convention we ignore the cost of writing the final result.
- The total cost of the plan shown previously, is the sum of the **cost of the selection** ($1000_{(scan)}+10_{(write\ to\ T1)}+500_{(scan)}+250_{(write\ to\ T2)} = 1760$) and the **cost of the join** ($40_{(sorting\ T1)}+2000_{(sorting\ T2)}+260_{(scan)} = 2300$), i.e., 4060 page I/Os.
- **Sort-merge join** is **one of several join methods**. We may be able to reduce the cost of this plan by choosing a different join method.
- As an alternative, suppose that we used **block nested loops** join instead of sort-merge join. Using T1 as the outer table, **for every 3-page block of T1**, we **scan all of T2**; thus, we scan T2 4 times. (T1's 10 pages ÷ 3 ⟶ 4 blocks)
- The cost of the join is hence the cost of scanning T1 (10) + the cost of scanning T2 ($4 * 250 = 1000$). The cost of the plan is now $1760_{(scan)} + 1010_{(join)} = 2770$ page I/Os.
- A **further refinement** is to **push the projection**, just like we pushed the selections past the join.
- Observe that **only** the $\boldsymbol{sid}$ attribute **of T1** and the $\boldsymbol{sid}$ and $\boldsymbol{sname}$ attributes **of T2 are really required**. As we scan Reserves and Sailors to do the selections, we could also **eliminate unwanted columns**.
- This **on-the-fly projection reduces** the **sizes of** the **temporary tables** T1 and T2. The reduction in size of T1 is substantial because only an integer field is retained.

- In fact, **T1** now **fits within 3 buffer pages** and we can perform a **block nested loops join** with a single scan of T2. The cost of the join step drops to under 250 page I/Os, and the total cost of the plan drops to about 2000 I/Os.

5.2 Using Indexes

- If indexes are available on the Reserves and Sailors tables, even better query evaluation plans may be available.
- For example, suppose that we have a **clustered static hash index on** the $\boldsymbol{bid}$ field of Reserves and another **hash index on** the $\boldsymbol{sid}$ field of Sailors.
- We can then use the query evaluation plan shown below:



- The **selection** $\boldsymbol{bid = 100}$ is **performed** on $Reserves$ by using the hash index on bid **to retrieve only matching tuples**.
- As before, assuming 100 boats are available and that reservations are spread out uniformly across all boats, we can **estimate** the number of **selected tuples to be** 100,000/100 = **1000**. **Since** the **index** on bid is **clustered**, **these 1000 tuples appear consecutively within** the **same bucket**; therefore, the **cost** is **10 page I/Os**. (given, 1 page can hold 100 tuples)
- For each selected tuple, we retrieve matching $Sailors$ tuples using the hash index on the sid field. The join is pipelined.
- For each tuple in the result of the join, we perform the selection $rating > 5$ and the projection of $sname$ on-the-fly.

- There are several important points to note here:
  - **Since** the **result of the selection** on Reserves is **not materialized** (and instead pipelined), the **optimization of projecting out fields that are not needed** subsequently is **not required**.
  - The join field sid is a key for Sailors. Therefore, at most one Sailors tuple matches a given Reserves tuple. For a **hash index**, **1.2 page I/Os is a good estimate of the cost for retrieving a data entry**. Assuming that the sid hash index on Sailors uses **Alternative (1)** for data entries, **1.2 I/Os is the cost to retrieve a matching Sailors tuple** (and if one of the **other two alternatives** is used, the cost would be **2.2 I/Os**).
    The cost does not depend on whether this index is clustered because there is at most one matching Sailors tuple.
  - We have **chosen not to push** the selection $\boldsymbol{rating > 5}$ **ahead of** the **join because** if we performed the selection before the join, the **selection would involve scanning Sailors** (assuming that no index is available on the rating field of Sailors). Further, whether or not such an index is available, **once we apply such** a **selection**, we have **no index on** the $\boldsymbol{sid}$ **field of** the **result of** the **selection** (hence, we'll have to scan the result of the selection, to identify matching tuples for the join). Thus, **pushing selections** ahead of joins is a **good heuristic**, but **not always** the **best strategy**. Typically, as in this example, the **existence of useful indexes** is the **reason a selection is not pushed**. (Otherwise, selections are pushed.)
- Let us estimate the cost of the plan shown previously.
- The selection of Reserves tuples costs 10 I/Os, as we saw earlier.
- There are 1000 such tuples, and for each, the cost of finding the matching Sailors tuple is **1.2 I/Os** (**cost of retrieving a data entry**), on average.
- The cost of this step (the join) is therefore 1200 I/Os.
- All remaining selections and projections are performed on-the-fly. The total cost of the plan (selection + join) is 1210 I/Os.

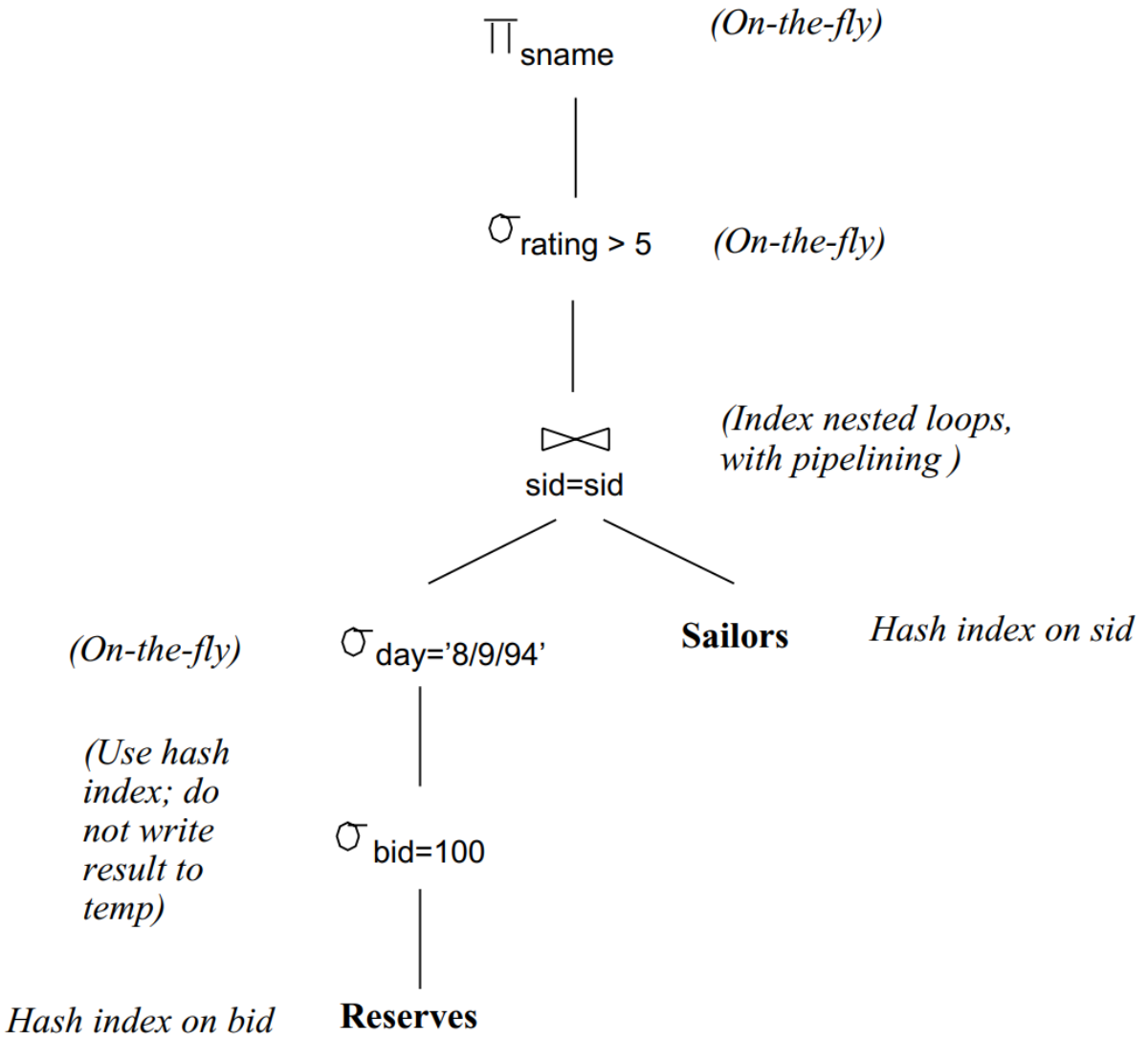
- As noted earlier, **this plan does not utilize clustering of** the **Sailors index**.
- The plan can be further refined if the index on the sid field of Sailors is clustered.
- Suppose we materialize the result of performing the selection $bid = 100$ on Reserves and sort this temporary table. This table contains 10 pages.

- Selecting the tuples costs 10 page I/Os (as before), writing out the result to a temporary table costs another 10 I/Os, and with 5 buffer pages, sorting this temporary table costs $2_{(r/w)} * 2_{(passes)} * 10 = 40$ I/Os.
- The selected Reserves tuples can now be retrieved in order by sid.
- If a sailor has reserved the same boat many times, all corresponding Reserves tuples are now retrieved consecutively; the matching Sailors tuple will be found in the buffer pool on all but the first request for it.
- This improved plan also demonstrates that **pipelining** is **not always** the **best strategy**.

- Now, consider the following query:

```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
        AND R.bid = 100 AND S.rating > 5
        AND R.day = '8/9/2002'
```

- The query evaluation plan is shown:



- Suppose that $\boldsymbol{bid}$ **and** $\boldsymbol{day}$ **form a key** for Reserves.
- Let us estimate the cost of the plan shown previously.
- The selection $bid = 100$ costs 10 page I/Os, as before, and the additional selection $day =$ '$8/9/2002$' is applied on-the-fly, eliminating all but (at most) 1 Reserves tuple (because, **<$\boldsymbol{bid}$, $\boldsymbol{day}$>** is a **key** so **each pair** is **unique**).
- There is at most one matching Sailors tuple, and this is retrieved in 1.2 I/Os.
- The selection on $rating$ and the projection on $sname$ are then applied on-thefly at no additional cost.
- The total cost of the plan is thus about 11 I/Os.
- In contrast, if we modify the naive plan in the naïve plan presented earlier to perform the additional selection on day together with the selection bid=100, the cost remains at 501,000 I/Os.

---

6. What a Typical Optimizer Does

- A relational query optimizer uses relational algebra equivalences to identify many equivalent expressions for a given query.
- For each such equivalent version of the query, all available implementation techniques are considered for the relational operators involved, thereby generating several alternative query evaluation plans.
- The optimizer estimates the cost of each such plan and chooses the one with the lowest estimated cost.

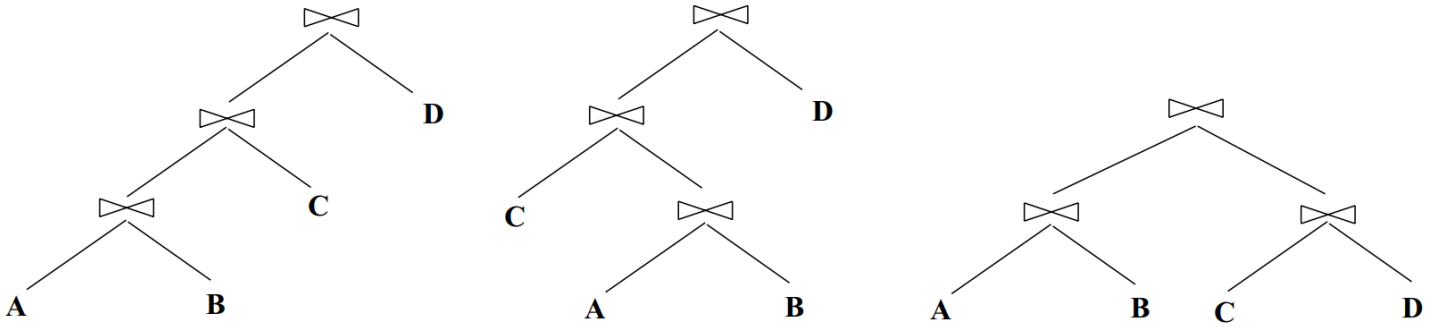
6.1 Alternative Plans Considered

- **Two relational algebra expressions over** the **same** set of **input tables** are said to be ***equivalent*** **if they produce** the **same result** on all instances of the input tables.
- Relational algebra equivalences play a central role in identifying alternative plans.
- Consider a basic SQL query consisting of a SELECT clause, a FROM clause, and a WHERE clause.
- This is easily represented as an algebra expression; the fields mentioned in the SELECT are projected from the cross-product of tables in the FROM clause, after applying the selections in the WHERE clause.
- The use of equivalences enables us to convert this initial representation into equivalent expressions.

- In particular:
  - Selections and cross-products can be combined into joins.
  - Joins can be extensively reordered
  - Selections and projections, which reduce the size of the input, can be "pushed" ahead of joins.
- Pushing the selection in a query ahead of the join may yield a dramatically better evaluation plan as we have seen previously.

Left-Deep Plans

- Consider a query of the form A ⋈ B ⋈ C ⋈ D; i.e., the natural join of 4 tables.
- 3 relational algebra operator trees that are equivalent to this query (based on algebra equivalences) are shown:



- By convention, the left child of a join node is the outer table and the right child is the inner table as discussed previously.
- By adding details such as the join method for each join node, it is straightforward to obtain several query evaluation plans from these trees.
- The **first two trees** in are examples of **linear trees**. In a linear tree, **at least 1 child of a join node is** a **base table**.
- The **first tree** is an example of a **left-deep tree** as the **right child of each join node** is a **base table**.
- The **third tree** is an example of a **non-linear** or **bushy tree**.
- **Optimizers typically use** a **dynamic-programming approach** to **efficiently search** the **class of** all **left-deep plans**.
- The **second** and **third kinds of trees** are therefore **never considered**.
- There are **23^5 other left-deep plans** that **differ only in** the **order** that **tables** are **joined**. If any of these plans has selection and projection conditions other than the joins themselves, these conditions are applied as early as possible given the choice of a join order for the tables.

- There are 2 main reasons for this decision to concentrate on left-deep plans, or plans based on left-deep trees:
    - **As** the **number of joins increases**, the **number of alternative plans increases rapidly** and it becomes **necessary to prune** the **space of alternative plans**.
    - **Left-deep trees allow** us to **generate** all **fully pipelined plans**; i.e., plans in which all joins are evaluated using pipelining.
      (Inner tables must always be materialized because we must examine the entire inner table for each tuple of the outer table. So, a plan in which an inner table is the result of a join, like the second and third trees, forces us to materialize the result of that join.)

6.2 Estimating the Cost of a Plan

- The cost of a plan is the sum of costs for the operators it contains.
  The cost of individual relational operators in the plan is estimated using information, obtained from the system catalog, about properties (e.g., size, sort order) of their input tables.
- The **cost of a plan**, **in terms of I/O costs**, can be **broken down into 3 parts**:
    - (1) reading the input tables (possibly multiple times in the case of some join and sorting algorithms),
    - (2) writing intermediate tables, and (possibly)
    - (3) sorting the final result (if the query specifies duplicate elimination or an output order).
- The **third part is common to all plans** (unless one of the plans happens to produce output in the required order), and, **in** the **common case** that a **fully-pipelined plan** is chosen, **no intermediate tables** are **written**.
- Thus, the **cost for a fully-pipelined plan** is **dominated by** part (1) ⟶ **reading input tables**. This cost depends greatly on the access paths used to read input tables.
- For plans that are not fully pipelined, the cost of materializing temporary tables can be significant. The cost of materializing an intermediate result depends on its size, and the size also influences the cost of the operator for which the temporary table is an input table.
- The result size for a join can be estimated by multiplying the maximum result size, which is the product of the input table sizes, by the reduction factor of the join condition.

Q1) Describe three techniques commonly used when developing algorithms for relational operators. Explain how these techniques can be used to design algorithms for the selection, projection, and join operators.

- The three techniques commonly used are indexing, iteration, and partitioning:
  - **Indexing**: If a selection or join condition is specified, use an index to examine just the tuples that satisfy the condition.
  - **Iteration**: Examine all tuples in an input table, one after the other. If we need only a few fields from each tuple and there is an index whose key contains all these fields, instead of examining data tuples, we can scan all index data entries.
  - **Partitioning**: By partitioning tuples on a sort key, we can often decompose an operation into a less expensive collection of operations on partitions. Sorting and hashing are two commonly used partitioning techniques.
- They can be used to design algorithms for selection, projection, and join operators as follows:
  - **Selection**:
    - For a selection with more than one tuple matching the query (in general, at least 5%), indexing like B+ Trees are very useful.
    - This comes into play often with range queries.
    - It allows us to not only find the first qualifying tuple quickly, but also the other qualifying tuples soon after (especially if the index is clustered).
    - For a selection condition with an equality query where there are only a few (usually 1) matching tuples, partitioning using hash indexing is often appropriate.
    - It allows us to find the exact tuple we are searching for with a cost of only a few (typically 1 or 2) I/Os.
  - **Projection**:
    - The projection operation requires us to drop certain fields of the input, which can result in duplicates appearing in the result set.
    - If we do not need to remove these duplicates, then the iteration technique can efficiently handle this problem.
    - On the other hand, if we do need to eliminate duplicates, partitioning the data and applying a sort key is typically performed.

- **Join**:
  - It is the most expensive database operation and can be a combination of all three techniques.
  - A join operation typically has **multiple selection** and **projection** elements built into it, so the importance of having appropriate indexes or of partitioning the data is just as above, if not more so.
  - When possible, the **individual selections** and **projections** are **applied to** 2 **relations before** they are **joined**, **so as to decrease** the **size of** the **intermediate table**.
  - As an example consider joining two relations with 100,000 tuples each and only 5 % of qualifying tuples in each table. Joining before applying the selection conditions, would result in a huge intermediate table size that would then have to be searched for matching selections. Alternatively, consider applying parts of the selection first. We can then perform a join of the 5,000 qualifying tuples found after applying the selection to each table, that can then be searched and handled significantly faster.

Q2) What is an access path? When does an index match an access path? What is a primary conjunct, and why is it important?

- An access path is a way of retrieving tuples from a table and consists of either (a file scan *or* an index) plus a matching selection condition.
- An index *matches* a selection condition if the index can be used to retrieve just the tuples that satisfy the condition.
- An index can match some subset of conjuncts in a selection condition even though it does not match the entire condition and we refer to the conjunct that the index matches as the ***primary conjuncts*** in the selection.
- Primary conjuncts are important because they **allow us to quickly discard information we do not need** and only focus in on searching/sorting the data that more closely matches the selection conditions.

Q3) What information is stored in the system catalogs?

- Information about relations, indexes, and views is stored in the system catalogs. This includes file names, file sizes, and file structure, the attribute names and data types, lists of keys, and constraints.

Q4) What are the benefits of making the system catalogs be relations?

- There are several advantages to storing the system catalogs as relations.
- Relational system catalogs take advantage of all of the implementation and management benefits of relational tables: **effective information storage and rich querying capabilities**.

Q5) What is the goal of query optimization? Why is optimization important?

- The goal of query optimization is to avoid the worst plans and find a good plan.
- The goal is to find a good plan and not an optimal plan (as that would be expensive).
- The difference in cost between a good plan and a bad plan can be several orders of magnitude: a good query plan can evaluate the query in seconds, whereas a bad query plan might even take days!

Q6) Describe pipelining and its advantages.

- Pipelining allows us to avoid creating and reading temporary relations; the I/O savings can be substantial.

Q7) Give an example query and plan in which pipelining cannot be used.

- **Bushy query plans** often cannot take advantage of pipelining because of limited buffer or CPU resources.
- Consider a bushy plan in which we are doing a selection on two relations, followed by a join.
- We cannot always use pipelining in this strategy because the result of the selection on the first relation may not fit in memory, and we must wait for the second relation's selection to complete before we can begin the join.

Q8) Describe the iterator interface and explain its advantages.

- The iterator interface for an operator includes the functions $\boldsymbol{open}$, $\boldsymbol{get\_next}$, and $\boldsymbol{close}$.
- It hides the details of how the operator is implemented, and allows us to view all operator nodes in a query plan uniformly.

Q9) Why do query optimizers consider only left-deep join trees? Give an example of a query and a plan that would not be considered because of this restriction.

- There are two main reasons for the decision to concentrate on left-deep plans:
  - As the number of joins increases, the **number of alternative plans increases** rapidly and it becomes necessary to prune the space of the alternative plans.
  - Left-deep trees **allow us to generate all fully pipelined plans**; i.e., plans in which all joins are evaluated using pipelining.
- Consider the join A ⋈ B ⋈ C ⋈ D. The query plan (A ⋈ B) ⋈ (C ⋈ D) would never be considered because it is a **bushy tree**.25

Q10) Consider a relation $R(a, b, c, d, e)$ containing 5,000,000 records, where each data page of the relation holds 10 records. R is organized as a sorted file with secondary indexes. Assume that $R.a$ is a candidate key for R, with values lying in the range 0 to 4,999,999, and that R is stored in $R.a$ order. For each of the following relational algebra queries, state which of the following three approaches is most likely to be the cheapest:

- **Access the sorted file for R directly.**
- **Use a (clustered) B+ tree index on attribute $R.a$.**
- **Use a linear hashed index on attribute $R.a$.**

1. $\sigma_{a<50{,}000}(R)$
2. $\sigma_{a=50{,}000}(R)$
3. $\sigma_{a>50{,}000 \wedge a<50{,}010}(R)$
4. $\sigma_{a\neq50{,}000}(R)$

1. For this selection, the choice of accessing the sorted file is slightly superior in cost to using the clustered B+ tree index simply because of the lookup cost required on the B+ tree.
2. A linear hashed index should be cheapest here.
3. A B+ tree should be the cheapest of the three.
4. Since the selection will require a scan of the available entries, and we're starting at the beginning of the sorted index, accessing the sorted file should be slightly more cost-effective, again because of the lookup time.

Q11) Consider the following schema with the Sailors relation:

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

For each of the following indexes, list whether the index matches the given selection conditions. If there is a match, list the primary conjuncts.

1. A B+-tree index on the search key ⟨ Sailors.sid ⟩.
   (a) $\sigma_{Sailors.sid<50,000}(Sailors)$
   (b) $\sigma_{Sailors.sid=50,000}(Sailors)$
2. A hash index on the search key ⟨ Sailors.sid ⟩.
   (a) $\sigma_{Sailors.sid<50,000}(Sailors)$
   (b) $\sigma_{Sailors.sid=50,000}(Sailors)$
3. A B+-tree index on the search key ⟨ Sailors.sid, Sailors.age ⟩.
   (a) $\sigma_{Sailors.sid<50,000 \wedge Sailors.age=21}(Sailors)$
   (b) $\sigma_{Sailors.sid=50,000 \wedge Sailors.age>21}(Sailors)$
   (c) $\sigma_{Sailors.sid=50,000}(Sailors)$
   (d) $\sigma_{Sailors.age=21}(Sailors)$
4. A hash-tree index on the search key ⟨ Sailors.sid, Sailors.age ⟩.
   (a) $\sigma_{Sailors.sid=50,000 \wedge Sailors.age=21}(Sailors)$
   (b) $\sigma_{Sailors.sid=50,000 \wedge Sailors.age>21}(Sailors)$
   (c) $\sigma_{Sailors.sid=50,000}(Sailors)$
   (d) $\sigma_{Sailors.age=21}(Sailors)$

1. (a) Match. Primary conjuncts are: $Sailors.sid < 50{,}000$
   (b) Match. Primary conjuncts are: $Sailors.sid = 50{,}000$
2. (a) No Match. Range queries cannot be applied to hash indexes.
   (b) Match. Primary conjunct are: $Sailors.sid = 50{,}000$
3. (a) Match. Primary conjunct are: $Sailors.sid < 50{,}000$ and $Sailors.sid < 50{,}000 \wedge Saliors.age = 21$
   (b) Match. Primary conjunct are: $Sailors.sid = 50{,}000$ and $Sailors.sid = 50{,}000 \wedge Saliors.age > 21$
   (c) Match. Primary conjunct are: $Sailors.sid = 50{,}000$
   (d) No Match. The index on ⟨ Sailors.sid, Sailors.age ⟩ is primarily sorted on Sailors.sid, therefore the entire relation would need to be searched to find those with a particular Sailors.age value.

4. (a) Match. Primary conjunct are: $Sailors.sid = 50{,}000$ and $Sailors.sid = 50{,}000 \wedge Saliors.age = 21$
   (b) Match. Primary conjunct are: $Sailors.sid = 50{,}000$
   (c) Match. Primary conjunct are: $Sailors.sid = 50{,}000$
   (d) No Match. The index on ⟨ Sailors.sid, Sailors.age ⟩ does not allow us to retrieve sets of sailors with age equal to 21.

Q12) Consider again the schema with the Sailors relation:

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

Assume that each tuple of Sailors is 50 bytes long, that a page can hold 80 Sailors tuples, and that we have 500 pages of such tuples. For each of the following selection conditions, estimate the number of pages retrieved, given the catalog information in the question.

1. Assume that we have a B+-tree index T on the search key ⟨ Sailors.sid ⟩, and assume that $IHeight(T) = 4$, $INPages(T) = 50$, $Low(T) = 1$, and $High(T) = 100{,}000$.
   (a) $\sigma_{Sailors.sid<50{,}000}(Sailors)$
   (b) $\sigma_{Sailors.sid=50{,}000}(Sailors)$

   (a) Assuming uniform distribution, around half of the 80 * 500 = 40,000 tuples will match the selection condition. The total cost is then the cost of finding the first leaf node (4 I/O's) plus the cost of retrieving the matching tuples.
   - If the index is clustered, then the total cost is 4 + 250 I/O's = 254 I/O's.
   - If the index is unclustered, then the cost of retrieving each matching tuple could be as high as 20,000 I/O's (one I/O for each matching tuple), leading to a total cost of 20,004 I/O's.
   - If the index is unclustered, then doing a file scan is most likely the preferred method with a cost of 500 I/O's.

   (b) Since *sid* is a PK for the relation we expect only one matching tuple for the hash index, therefore the cost is just the height of the tree (4 I/O's) plus the cost of reading the qualifying tuple's page (1 I/O) which adds up to be 5 I/O's.

2. Assume that we have a hash index T on the search key ⟨ Sailors.sid ⟩, and assume that $IHeight(T) = 2$, $INPages(T) = 50$, $Low(T) = 1$, and $High(T) = 100{,}000$.
   (a) $\sigma_{Sailors.sid<50{,}000}(Sailors)$
   (b) $\sigma_{Sailors.sid=50{,}000}(Sailors)$

(a) Since a hash index cannot help us for range queries, the index is useless, and therefore we must do a file scan at a cost of 500 pages I/O's

(b) Since sid is a PK for the relation we expect only one matching tuple for the hash index, therefore the cost is just the height of the tree (2 I/O's) plus the cost of reading the qualifying tuple's page (1 I/O) which adds up to be 3 I/O's

---

External Sorting

- In this part, we consider a widely used and relatively **expensive operation**; **sorting records according to a search key**.
- We begin by considering the many uses of sorting in a database system.
- We introduce the idea of external sorting by considering a very simple algorithm.
- By using repeated passes over the data, even very large datasets can be sorted with a small amount of memory.
- This algorithm is generalized to develop a realistic external sorting algorithm later.

---

1. When does a DBMS Sort Data?

- Sorting a collection of records on some (search) key is a very useful operation.
- The key can be a single attribute or an ordered list of attributes. Sorting is required in a variety of situations, including the following important ones:
  - Users may want **answers in some order**; for example, by increasing age
  - Sorting records is the **first step** in **bulk loading a tree index**
  - Sorting is useful for **eliminating duplicate copies** in a collection of records
  - A widely used algorithm for performing a very important relational algebra operation, called ***join***, requires a sorting step.
- Although main memory sizes are growing rapidly, the ubiquity of database systems has led to increasingly larger datasets as well.
- When the data to be sorted is too large to fit into available main memory, we need an ***external sorting*** algorithm.
- Such algorithms seek to minimize the cost of disk accesses.

---

2. A Simple Two-Way Merge Sort

- We begin by presenting a simple algorithm to illustrate the idea behind external sorting. This algorithm **utilizes only 3 pages of main memory**, and it is presented only for pedagogical purposes.

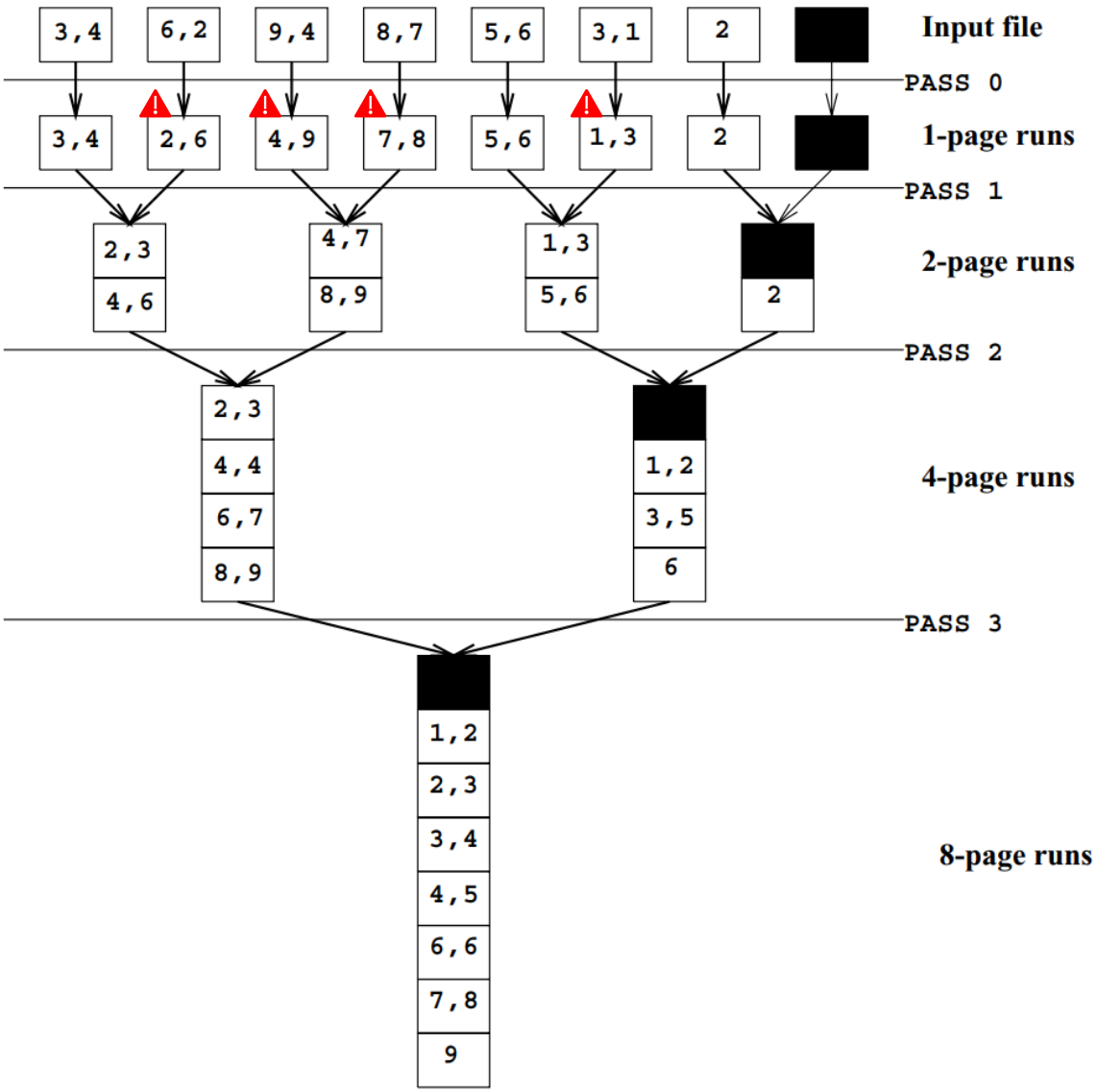
- In practice, many more pages of memory are available, and we want our sorting algorithm to use the additional memory effectively; such an algorithm is presented in the next section.
- When sorting a file, several **sorted subfiles** are typically **generated** in **intermediate steps**. In this chapter, we refer to **each sorted subfile** as a **run**.
- Even if the entire file does not fit into the available main memory, we can sort it by breaking it into smaller subfiles, sorting these subfiles, and then merging them using a minimal amount of main memory at any given time.
- In the **first pass**, the **pages in the file are read** in **one at a time**.
- **After a page is read in**, the **records in it are sorted** and the **sorted page** (a **sorted run** (subfile) that is **1 page long**) is **written out**.
- Quicksort or any other in-memory sorting technique can be used to sort the records on a page.
- In subsequent passes, **pairs of runs from** the the **previous pass output** are **read in and merged to produce runs** (subfiles) that are **twice as long**.
- This algorithm is shown below:

```
// Given a file on disk, sorts it using 3 buffer pages
proc 2_way_extsort (file)
    // Produce runs (subfiles) that are 1 page long: Pass 0
    Read each page into memory, sort it, write it out.
    // Merge pairs of runs to produce longer runs until only
    // 1 run (containing all records of input file) is left
    While the number of runs at end of previous pass is > 1:
        // Pass i = 1, 2, ...
        While there are runs to be merged from previous pass:
            Choose next 2 runs (from previous pass).
            Read each run into an input buffer; page at a time.
            Merge the runs and write to the output buffer;
            force output buffer to disk one page at a time.
endproc
```

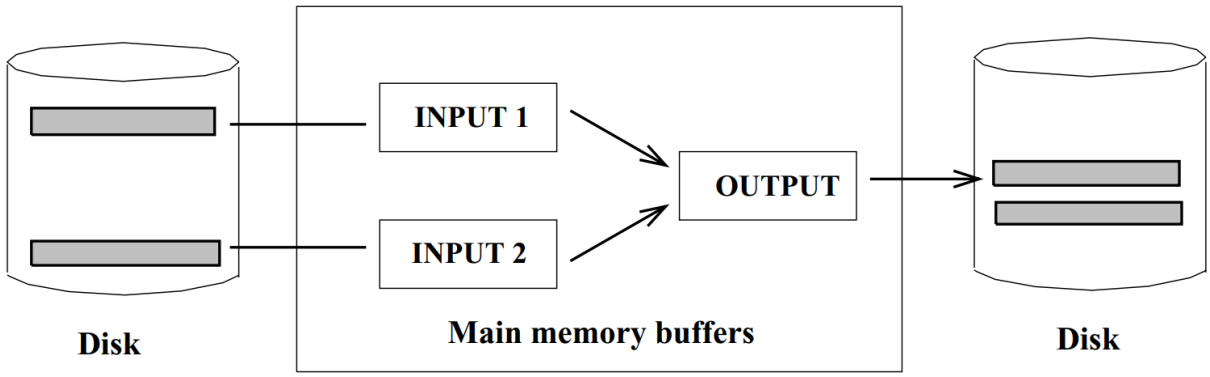
- **If** the **number of pages** in the input file **is 2^k**, for some k, then:
  - Pass 0 produces 2^k sorted runs of 1 page each,
  - Pass 1 produces 2^{k-1} sorted runs of 2 pages each,

- Pass 2 produces 2^{k-2} sorted runs of 4 pages each,
  and so on, until
- Pass k produces 1 sorted run of 2^k pages.

- In each pass, we read every page in the file, process it, and write it out.
- Therefore, we have **2 disk I/Os per page**, **per pass**.
- The **number of passes** is $[\log_2 N] + 1$, where **N** is the **number of pages in the file**.
- The overall cost is = $2 * \#pages * \#passes \longrightarrow 2N([\log_2 N] + 1)$ I/Os.
- The algorithm is illustrated on an example input file containing 7 pages as shown:



- The sort takes 4 passes ($[\log_2 7] + 1 \approx 4$), and in each pass, we read and write 7 pages, for a total of 56 I/Os.
- The dark pages in the figure illustrate what would happen on a file of eight pages; the number of passes remains at 4, but we read and write an additional page in each pass for a total of 64 I/Os.
- This algorithm requires just 3 buffer pages in main memory as the figure below illustrates:

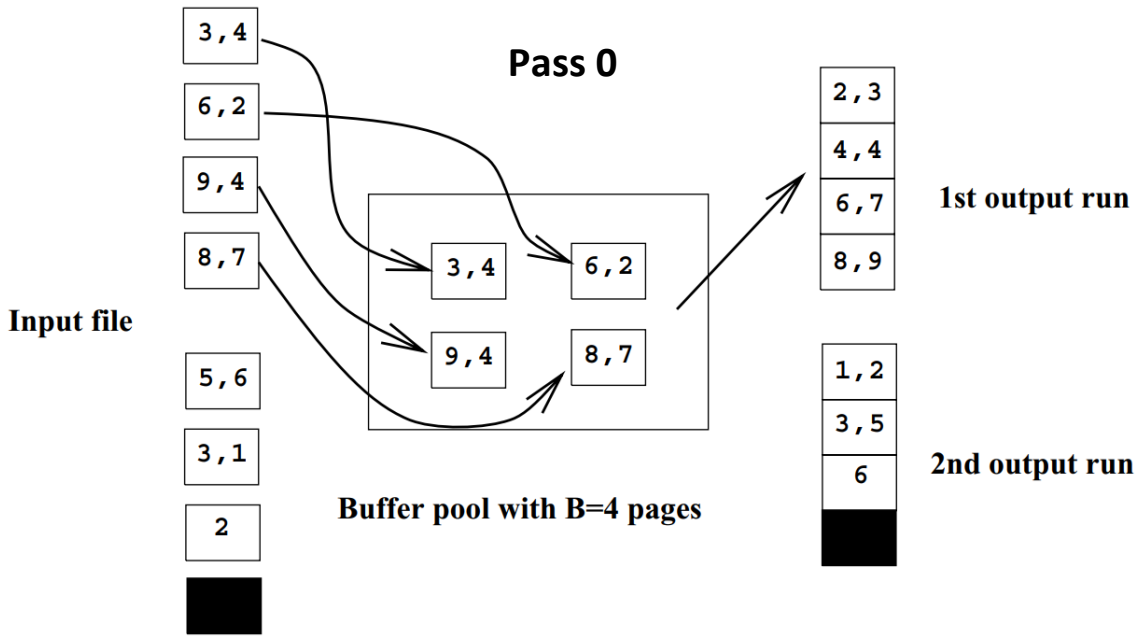


- This observation raises an important point: **Even if we have more buffer space** available, this **simple algorithm does not utilize it effectively**.
- The **external merge sort** algorithm that we discuss next **addresses this problem**.

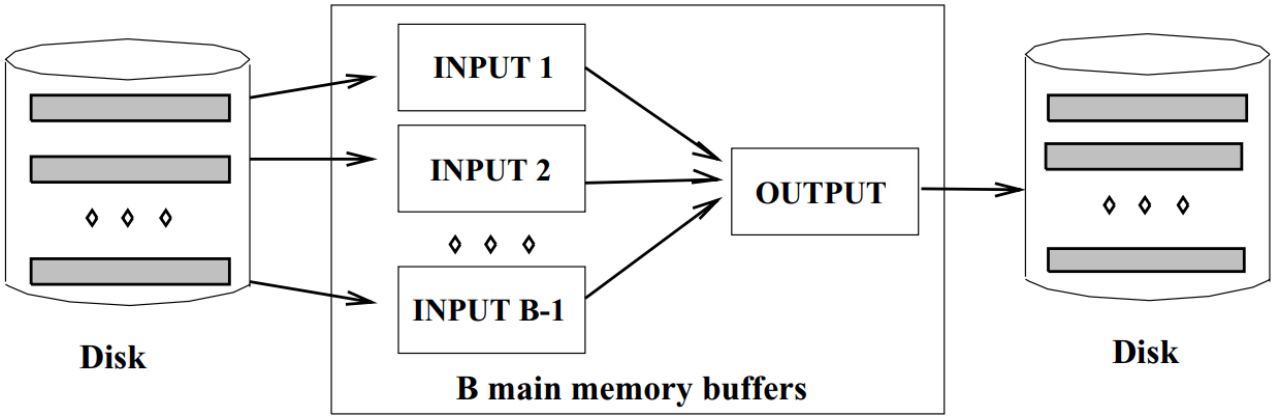
---

3. External Merge Sort

- Suppose that **B buffer pages** are **available in memory** and that we need to sort a large file with N pages. The **intuition behind** the generalized algorithm that we now present is to **retain** the **basic structure** of **making multiple passes while** trying to **minimize** the **number of passes**.
- There are two important modifications to the two-way merge sort algorithm:
  - **In Pass 0**, **read in B pages at a time** and **sort internally** to **produce N/B runs of B pages each** (except for the last run, which may contain fewer pages). This modification is illustrated below using the input file from the previous section's diagram and a buffer pool with 4 pages.



  - In passes i = 1, 2, . . . use **B−1 buffer pages for input** and use the remaining **1 page for output**. Hence, you do a **(B − 1)-way merge in each pass**. The utilization of buffer pages in the merging passes is illustrated below. (below is Passes i > 0)



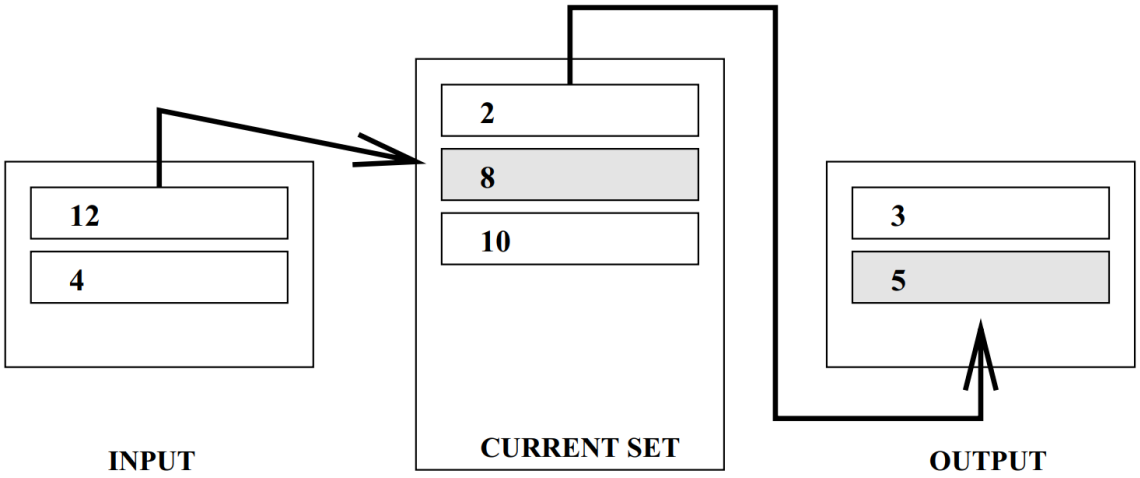
- The **first refinement reduces** the **number of runs produced by Pass 0** to $N1 = N/B$, versus **N for the 2-way merge**.
- The **second refinement** is even more important. By doing a **(B – 1)-way merge**, the number of passes is reduced dramatically—including the initial pass, it becomes $(\log_{B-1} N1 + 1)$ versus $(\log_2 N + 1)$ for the 2-way merge algorithm.
- Because B is typically quite large, the savings can be substantial.
- The external merge sort algorithm is given below:

```
// Given a file on disk, sorts it using B buffer pages
proc extsort (file)
    // Produce runs (subfiles) that are B pages long: Pass 0
    Read B pages into memory, sort it, write out a run.
    // Merge B − 1 runs at a time to produce longer runs until only
    // 1 run (containing all records of input file) is left
    While the number of runs at end of previous pass is > 1:
        // Pass i = 1, 2, ...
        While there are runs to be merged from previous pass:
            Choose next B − 1 runs (from previous pass).
            Read each run into an input buffer; page at a time.
            Merge the runs and write to the output buffer;
            force output buffer to disk one page at a time.
endproc
```

3.1 Minimizing the Number of Runs

- **In Pass 0** we **read in B pages at a time** and **sort them internally** to **produce N/B runs of B pages each** (except for the last run, which may contain fewer pages).
- With a more aggressive implementation, called **replacement sort**, we can write out runs of approximately 2B internally sorted pages on average.

- This improvement is achieved as follows. This process is illustrated below:



- We begin by reading in pages of the file of tuples to be sorted, say R, until the buffer is full, reserving 1 page for use as an input buffer and 1 page for use as an output buffer.
- We refer to the **B – 2 pages** of R tuples that are not in the input or output buffer as the ***current set***.
- Suppose that the file is to be sorted in ascending order on some **search key *k***. **Tuples** are **appended to** the **output buffer in ascending order by *k* value**.
- The idea is to repeatedly **pick** the **tuple in** the **current set with** the **smallest *k*** value that is still **greater than** the **largest *k*** value **in** the **output buffer** and **append it to** the **output buffer**. (Initially o/p buffer is empty, so pick tuple with smallest *k*)
- For the output buffer to remain sorted, the **chosen tuple** must satisfy the condition that its ***k*** value be $\geq$ to the **largest *k*** value **currently in** the **output buffer**.
- The tuple in the current set that is going to be appended to the output next is highlighted, as is the most recently appended output tuple.
- **Of all tuples** (here, 8, 10) in the current set **that satisfy** this condition, we **pick** the one with the **smallest k value** (i.e., 8) and **append it to** the **output buffer**.
- **Moving this tuple to the output** buffer **creates some space in** the **current set**, which we use to **add the next input tuple to** the **current set**.
- When all tuples in the input buffer have been consumed in this manner, the next page of the file is read in.
- Of course, the **output buffer is written out when it is full**, thereby extending the current run.
- The important question is this: **When do we have to terminate the current run and start a new run?**

- **As long as some tuple *t*** in the **current set** has a **bigger *k*** value **than** the most **recently appended output tuple**, **we can append *t* to** the **output buffer** and the **current run** can be **extended**.
- **When every tuple in** the **current set** is **smaller than** the **largest tuple in** the **output buffer**, the **output buffer** is **written out** and **becomes** the **last page in** the **current run** (subfile).
- We then start a new run and continue the cycle of writing tuples from the input buffer to the current set to the output buffer.
- This algorithm **produces runs** that are about **2B pages long**, on average.

Q1) Suppose you have a file with 10,000 pages and you have three buffer pages. Answer the following questions for each of these scenarios, assuming external sorting algorithm is used:

(a) A file with 10,000 pages and 3 available buffer pages.

(b) A file with 20,000 pages and 5 available buffer pages.

(c) A file with 2,000,000 pages and 17 available buffer pages.

1. How many runs will you produce in the first pass?
2. How many passes will it take to sort the file completely?
3. What is the total I/O cost of sorting the file?
4. How many buffer pages do you need to sort the file completely in just two passes?

- The answer to each question is given below.
  1. In the first pass (Pass 0), N/B runs of B pages each are produced, where N is the number of file pages and B is the number of available buffer pages:
     a. ⌈10000/3⌉ = 3334 sorted runs
     b. ⌈20000/5⌉ = 4000 sorted runs
     c. ⌈2000000/17⌉ = 117648 sorted runs
  2. The number of passes required to sort the file completely, including the initial sorting pass, is $\log_{B-1} N1 + 1$, where N1 = N/B is the number of runs produced by Pass 0:
     a. $\lceil \log_2 3334 \rceil + 1$ = 13 passes
     b. $\lceil \log_4 4000 \rceil + 1$ = 7 passes
     c. $\lceil \log_{16} 117648 \rceil + 1$ = 6 passes

3. Since each page is read and written once per pass, the total number of page I/Os for sorting the file is $2 * N * (\#passes)$:
    a. $2 * 10000 * 13 = 260000$
    b. $2 * 20000 * 7 = 280000$
    c. $2 * 2000000 * 6 = 24000000$
4. Solving for x:
    a. $\log_{x-1}\left(\frac{10000}{x}\right) + 1 = 2 \rightarrow x = \lceil 100.5 \rceil = 101$ pages
    b. $\log_{x-1}\left(\frac{20000}{x}\right) + 1 = 2 \rightarrow x = \lceil 141.9 \rceil = 142$ pages
    c. $\log_{x-1}\left(\frac{2000000}{x}\right) + 1 = 2 \rightarrow x = \lceil 1414.7 \rceil = 1415$ pages

---

Q2) Consider the refinement to the external sort algorithm that produces runs of length 2B on average, where B is the number of buffer pages. This **refinement** was described **under** the **assumption** that **all records are** the **same size**. **Explain why this assumption is required**.

- The assumption that all records are of the same size is used when the algorithm moves the smallest entry with a key value large than ***k*** to the output buffer and replaces it with a value from the input buffer.
- This ”replacement” will only work if the records of the same size.
- If the entries are of variable size, then we must also keep track of the size of each entry, and replace the moved entry with a new entry that fits in the available memory location.
- Dynamic programming algorithms have been adapted to decide an optimal replacement strategy in these cases.
