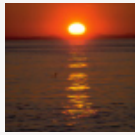
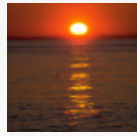
Chapter 13: Query Processing

Database System Concepts, 5th Ed.



Chapter 13: Query Processing

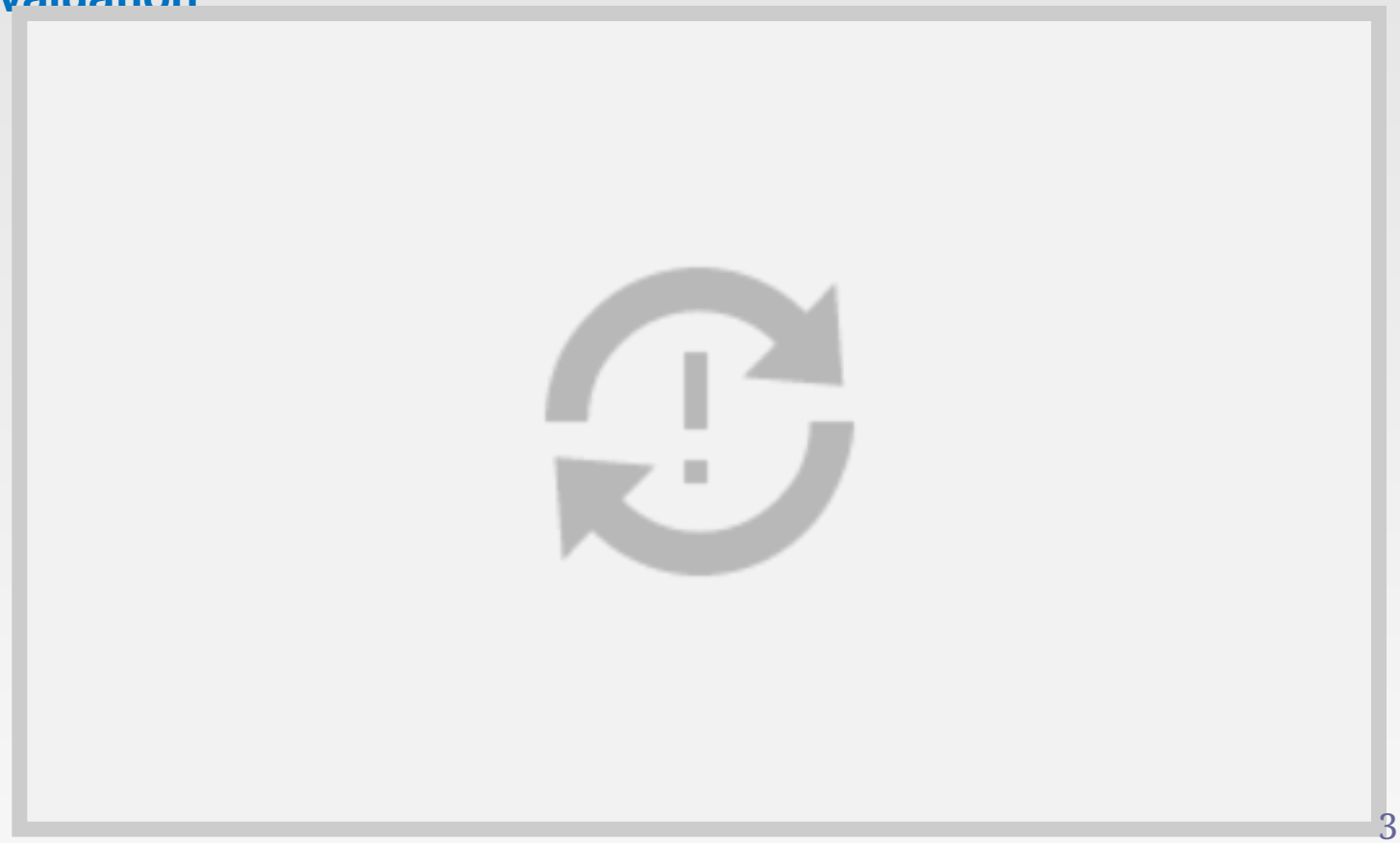
- Overview
- Measures of Query Cost
- Selection Operation
- Join Operation
- Evaluation of Expressions

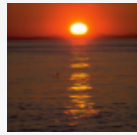


Basic Steps in Query Processing

The basic steps involved in processing a query are as follows:

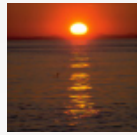
1. Parsing and Translation
2. Optimization
3. Evaluation





Basic Steps in Query Processing

- **Parsing and Translation**
  - The first action the system must take in query processing is to translate a given query into its internal form.
  - In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on.
  - The system constructs a parse-tree representation of the query, which is then translated into a relational-algebra expression.
- **Optimization**
  - Given a query, there are generally a variety of methods for computing the answer to the query.
  - For example, in SQL, a query could be expressed in several different ways. Each SQL query can then be translated into its corresponding relational-algebra expression.



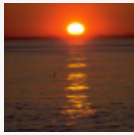
Basic Steps in Query Processing

- **Optimization (Contd…)**
  - As an illustration, consider the following query:

    select *balance*

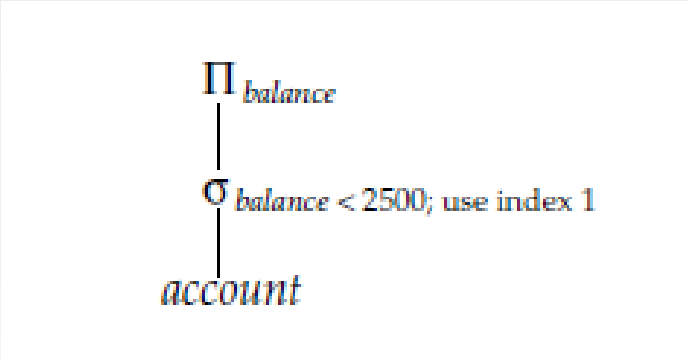
    from *account*

    where *balance < 2500*
  - This query can be translated into either of the following relational-algebra expressions:
    - $\sigma_{balance<2500}(\Pi_{balance}(account))$
    - $\Pi_{balance}(\sigma_{balance<2500}(account))$
  - To fully specify how to evaluate a query, we need to provide the relational algebra expression along with instructions specifying how to evaluate each operation.
  - A relational-algebra operation along with instructions on how to evaluate it is called an **evaluation primitive (annotated expression).**



Basic Steps in Query Processing

- **Optimization (Contd…)**
  - Annotated expression specifying detailed evaluation strategy is called a **query evaluation-plan**.
    - E.g., An index can be used on *balance* to find accounts with balance < 2500 quickly and efficiently. Indexes are a powerful tool used in the background of a database to speed up querying.
    - Or can perform complete scan of the database and discard accounts with balance ≥ 2500.



A query-evaluation plan

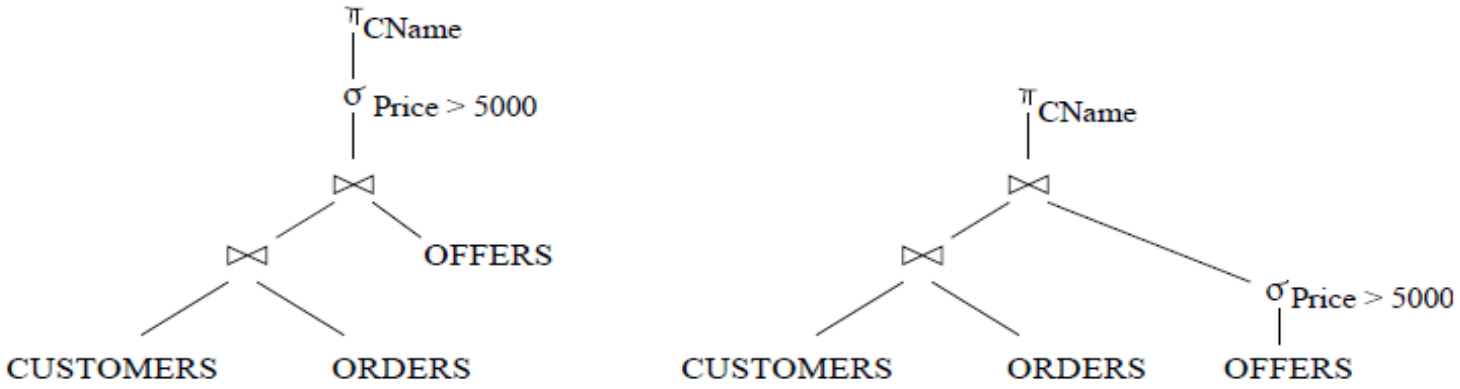
Another Example of Query Evaluation-plan

- A relational algebra expression may have many equivalent expressions, e.g.,

$$\pi_{\texttt{CName}}(\sigma_{\texttt{Price}>5000}((\texttt{CUSTOMERS} \bowtie \texttt{ORDERS}) \bowtie \texttt{OFFERS}))$$

$$\pi_{\texttt{CName}}((\texttt{CUSTOMERS} \bowtie \texttt{ORDERS}) \bowtie (\sigma_{\texttt{Price}>5000}(\texttt{OFFERS})))$$

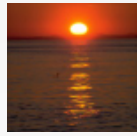
Representation as *logical query plan* (a tree):



Non-leaf nodes ≡ operations of relational algebra (with parameters); Leaf nodes ≡ relations

Basic Steps in Query Processing

- **Query Evaluation**
  - The different evaluation plans for a given query can have different costs.
  - In order to optimize a query, a query optimizer must know the cost of each operation.
  - Among all semantically equivalent expressions, the one with the least costly evaluation plan is chosen for a query.
  - Once the query plan is chosen, **query-execution engine takes a** query-evaluation plan, executes that plan, and returns the answer to the query.
- Cost of query evaluation also depends on the **database buffer** size.
- A database buffer is **a temporary storage area in the main memory.**
- Its purpose is to minimize the number of times the database needs to access secondary memory. This is done by keeping frequently accessed data in the buffer pool.



Catalog Information for Cost Estimation

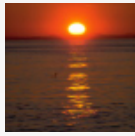
Cost estimate of a query evaluation plan is based on **statistical information** obtained from the **system catalog**.

Information about relations and attributes:

- N_R: number of tuples in the relation R.
- B_R: number of blocks that contain tuples of the relation R.
- S_R: size of a tuple of R.
- F_R: blocking factor; number of tuples from R that fit into one block ($F_R = \lceil N_R/B_R \rceil$)
- $V(A, R)$: number of distinct values for attribute A in R.
- $SC(A, R)$: selectivity of attribute A
  $\equiv$ average number of tuples of R that satisfy an equality condition on A.

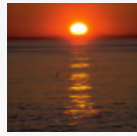
  $SC(A, R) = N_R/V(A, R)$.

NOTE: The size of a tuple means the amount of memory (in bytes) taken by a tuple.



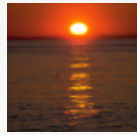
Measures of Query Cost

- Cost of query evaluation is generally measured as total elapsed time for answering a query
- Many factors contribute to the measure of query cost:
  - *Disk accesses, CPU time* to execute a query, and cost of network *communication* in case of a distributed or parallel system
- The disk access cost is only used to compute the cost of a query evaluation plan as it is the predominant cost (*disk access rate is very slow compared to CPU speed and memory accesses*).
- We ignore CPU costs for simplicity.



Measures of Query Cost (Cont.)

- Disk access cost is measured by taking into account
  - Number of seeks, Number of blocks read, Number of blocks written
    4. Cost to write a block is greater than cost to read a block as the data is read back after being written to ensure that the write was successful
- So, the **number of block transfers** to/from disk and the **number of seeks** are used as the query cost measures
- Disk seek operation causes the physical positioning of the read/write head of the disk at a particular location on disk.
  - t_T – time to transfer one block
  - t_S – time for one disk seek
  - **Cost for *b* block transfers plus *S* seeks = $b * t_T + S * t_S$**

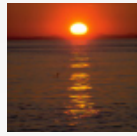


Selection Operation

- **Selection Operation:** The select operation must search through the data files to find records that satisfy the selection condition (**file scan**).

  $\sigma_{A=a}(R)$

  where, *a* is a constant value, *A* is an attribute of relation *R.*
- **File scans** are search algorithms that locate and retrieve records that fulfill a selection condition.
- The cost of a selection operation depends on a number of factors, including whether a database index exists on the selection condition or not and if the record exists in the database or not.
- **Basic Algorithms**
- We consider a selection operation on a relation whose tuples are stored together in one file.
- Two scan algorithms to implement the selection operation are:
  - **S1 (Linear Search)**
  - **S2 (Binary Search)**



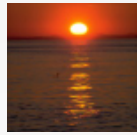
Selection Operation

- **S1 (Linear Search)**
  - Linear Search sequentially checks each element in the list until it finds a match or the list is exhausted.
  - For selection operation using S1, the system scans each file block and tests all records to see whether they satisfy the selection condition.
  - For a selection on a key attribute, the system can terminate the scan if the required record is found, without looking at the other records of the relation.
  - The cost of linear search, in terms of number of I/O operations, is given by the following expression:

    Cost(S1) = $B_R/2$ (On an average, if the record exists)

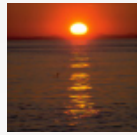
    Cost(S1) = B_R (If the record does not exist)

    where, B_R: **Number of disk blocks containing tuples of R.**



Selection Operation

- **S2 (Binary Search)**
  - Linear search works on both sorted and unsorted list, while binary search requires sorted data.
  - If a file is sorted on an attribute, and the selection condition is an equality comparison on the attribute, we can use a binary search to locate records that satisfy the selection.
  - For selection operation using S2, the binary search algorithm continuously divides the sorted list and compares the middle element with the target value.
  - The system performs the binary search on the blocks of the file.
  - The number of blocks that need to be examined to find a block containing the required records is $\lceil \log_2(B_R) \rceil$ where, B_R *denotes the number of blocks in* the file as the time complexity of binary search is O(log *n*), where *n* is the number of elements in the list.



Selection Operation

- **S2 (Binary Search) (Contd…)**
  - If the selection is on a non-key attribute, more than one block may contain required records, and the cost of reading the extra blocks has to be added to the cost estimate of binary search.
  - The cost of binary search for the file ordered based on attribute A (primary index) is as follows:

$$\text{cost(S2)} = \lceil \log_2(B_R) \rceil + \left\lceil \frac{SC(A, R)}{F_R} \right\rceil - 1$$

  - $\lceil \log_2(B_R) \rceil \equiv$ cost to locate the first tuple using binary search
  - Second term $\equiv$ blocks that contain records satisfying the selection.
  - If A is primary key, then $SC(A, R) = 1$, hence $\text{cost(S2)} = \lceil \log_2(B_R) \rceil$.

Example to Illustrate the cost of Selection Operation by S2 Algorithm

Example (for Employee DB)

- $F_{Employee} = 10$;
  $V(Deptno, Employee) = 50$ (different departments)

- $N_{Employee} = 10,000$ (Relation Employee has 10,000 tuples)

- Assume selection $\sigma_{Deptno=20}(Employee)$ and Employee is sorted on search key Deptno :

  $\Longrightarrow$ 10,000/50 = 200 tuples in Employee belong to Deptno 20;

  (assuming an equal distribution)

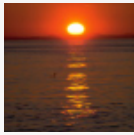
  200/10 = 20 blocks for these tuples

  $\Longrightarrow$ A binary search finding the first block would require $\lceil \log_2(1,000) \rceil = 10$ block accesses

  Total cost of binary search is 10+20 block accesses (versus 1,000 for linear search and Employee not sorted by Deptno).

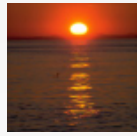
Selection Operation Using Indices

- **Index Scan:** Index scan refers to search algorithms that use an index.
- Index is a type of data structure that is used to locate and access the data in a database table quickly.
- Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.
- Index scan involves searching an index page to find records that match the condition part of a query.
- Thus, by creating a separate index structure based on specific columns, DBMSs can access data more quickly and efficiently, thus, reducing the time and resources required for data retrieval operations.



Indexing Methods

- **Primary Index**
  - If the index is created on the basis of the primary key of the table, then it is known as primary indexing.
  - In this indexing, the data file is sorted or clustered based on the primary key
  - The primary key values are unique for each record and does not contain duplicates.
  - It is also called **clustered index**.
- **Secondary Index**
  - A secondary index is an index defined on columns other than primary key attribute(s)
  - It is not based on primary key and thus may contain duplicates.
  - It is also called **non-primary/non-clustered index**.

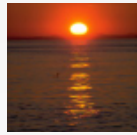


Important Notations

Information about indexes:

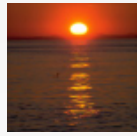
- $\mathtt{HT}_\mathtt{I}$: number of levels in index I (B^+-tree).
- $\mathtt{LB}_\mathtt{I}$: number of blocks occupied by leaf nodes in index I (first-level blocks).
- $\mathtt{Val}_\mathtt{I}$: number of distinct values for the search key.

Note: The **B+ tree** is a balanced binary search tree. It follows a multi-level **index** format. In the **B+ tree**, leaf nodes denote the actual data pointers.



Selection Operation Using Index Scan

- **S3 – Primary index I for A, A primary key, with equality comparison A = a (as the selection criteria)**
- Search key value will be the primary key and index will be created on it.
- In the where clause, we will have equality comparison with the primary key. For E.g., *select * from student where std_id = 105*;
- These type of indexes are normally stored in B+ tree with height 'h'.
- **Cost(S3) = HT_I + 1** (only 1 tuple satisfies the condition)
  - Where, HT_I: No. of levels in the B+ tree that is used to store the index I.
- The cost is equal to the cost of locating the search key (HT_I) in the index file and one is added for retrieving the actual data record from the data file.

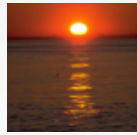


Selection Operation Using Index Scan

- **S4 – Primary index I on non-key attribute A with equality comparison A = a (as the selection criteria)**
- This is same as S3 but multiple records may meet the condition as the attribute is a non-key attribute.
- Hence, we add the number of blocks containing the required records to the Cost.

$$\text{cost}(S4) = HT_I + \left\lceil \frac{SC(A, R)}{F_R} \right\rceil$$

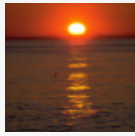
- Where, HT_I: No. of levels in the B+ tree that is used to store the index I.
- First term HT_I - Cost of locating the search key
- Second term - Specifies the number of blocks containing the desired records that satisfy the selection condition.



Selection Operation Using Index Scan

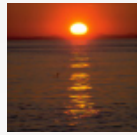
- S5 – Non-Primary/Secondary index on non-key attribute A with equality comparison A = a (as the selection criteria)
- **Cost(S5) = HT_I + SC(A, R)**

  Where, SC(A, R) is the average number of tuples that satisfy an equality condition on A.
- This is the worst-case cost where every record resides in a different block on the disk.
- The cost is equal to the cost of locating the search key (HT_I) in the index file and one block access for each data record retrieval.



Selections Involving Comparisons

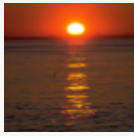
- When the comparison operators (<, ≤, >, ≥) are used to retrieve records from a file sorted by the search attribute, the first record satisfying the selection condition is first located.
- The total blocks before (<, ≤) or after (>, ≥) is added to the cost of locating the first record.
- Selections of the form $\sigma_{A \leq v}(R)$ or $\sigma_{A \geq v}(R)$ are implemented by using a file scan or binary search, or by using the following algorithms:
  - **S6 – Selection Involving a primary index on A,** or
  - **S7 – Selection Involving a secondary index on A**



Selections Involving Comparisons

S6 – Selection Involving a primary index on A

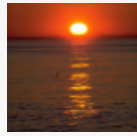
- When the selection condition given by the user is a comparison, then we use a primary ordered index, such as the primary B^+ tree index.
- For example: **select * from student where std_id ≥ 105;**
- As the index is on primary key, records are also sorted on the key.
- The parser has to seek the first record satisfying the condition std_id = 105. Thereafter, it has to add to cost the number of blocks satisfying the condition std_id > 105.
- **Cost(S6) = Seek time + 'B' number of blocks before (<, ≤) or after (>, ≥) is added**
  - Where, seek time is the time taken by a disk drive's read/write head to locate the area on the disk from where the stored data is to be read



Selections Involving Comparisons

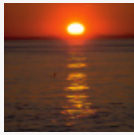
S7 – Selection Involving a secondary index on A with comparison operator

- This is same as the selection algorithm involving secondary index with equality operator (S5) but in this case the operator is a comparison operator. For example: **select * from student where age ≥ 18;**
- Here, age is the search key column and std_name is the index column.
- One seek is required to locate record where age = 18.
- Rest of the matching records are searched sequentially.
- **Cost(S7) = HT_I + SC(A, R) + Number of blocks searched sequentially where age > 18**
- Where, HT_I - Cost of locating the search key, SC(A, R): No. of tuples satisfying the equality condition



Join Operation

- We will study several algorithms for computing the join of relations, and also analyse their respective costs.
- The term **equi-join** is used to refer to a join of the form: $\boldsymbol{R} \bowtie_{R.A = S.B} \boldsymbol{S}$ where, *A* and *B* are attributes or sets of attributes of relations *R* and *S* respectively.
- The different algorithms used to implement the join operation are as follows:
  - **Nested-loop join**
  - **Block nested-loop join**
  - **Indexed nested-loop join**
  - **Merge-join/Sort-Merge join**
- Choice of a particular algorithm is based on its cost estimate.

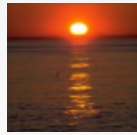


Join Operation Example

Example:

Assume the query CUSTOMERS $\bowtie$ ORDERS (with join attribute only being CName)

- $N_{CUSTOMERS} = 5{,}000$ tuples
- $F_{CUSTOMERS} = 20$, i.e., $B_{CUSTOMERS} = 5{,}000/20 = 250$ blocks
- $N_{ORDERS} = 10{,}000$ tuples
- $F_{ORDERS} = 25$, i.e., $B_{ORDERS} = 400$ blocks
- $V(\mathtt{CName}, \mathtt{ORDERS}) = 2{,}500$, meaning that in this relation, on average, each customer has four orders
- Also assume that CName in ORDERS is a foreign key on CUSTOMERS



Estimating the Size of Join

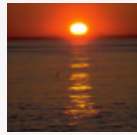
- The Cartesian product $R \times S$ results in $N_R * N_S$ tuples; each tuple requires $S_R + S_S$ bytes.

1) Size of Join using Information about Foreign Key

It is assumed that Cname is a foreign key in ORDERS referencing CUSTOMERS.

If $\text{schema}(R) \cap \text{schema}(S) =$ foreign key in S referencing R, then the number of tuples in R⋈S is exactly N_S.

- In the example query CUSTOMERS ⋈ ORDERS, CName in ORDERS is a foreign key of CUSTOMERS; the result thus has exactly $N_{ORDERS} = 10{,}000$ tuples



Estimating the Size of Join

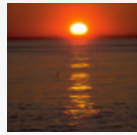
2) Size of Join without using Information about Foreign Key

- If schema(R) ∩ schema(S) = {A}, which is not a key for R or S
- Then, the number of tuples in ***R*** ⋈ ***S*** is given by the following expression:

$$= Min\left(\frac{N_R \times N_S}{V(A,S)}, \frac{N_R \times N_S}{V(A,R)}\right)$$

- For this example, size estimates for CUSTOMERS ⋈ ORDERS without using information about foreign keys:
  - V(CName; CUSTOMERS) = 5,000,
  - V(CName; ORDERS) = 2,500
- The two estimates are: 5,000*10,000/2,500=20,000 and 5,000*10,000/5,000=10,000.

No. of tuples in CUSTOMERS ⋈ ORDERS = Min(20000, 10000) = 10, 000

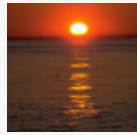


Nested-loop Join

- A nested loop join is a join that contains a pair of nested for loops.
- To perform the nested-loop join on two relations **R** and **S**, we use an algorithm known as the **Nested-loop join algorithm**.
- The computation takes place as follows:

  $\mathbf{R} \bowtie_{\mathbf{c}} \mathbf{S}$

  Where, R is known as the **outer relation** and S is the **inner relation** of the join.
- It is because the for loop of R encloses the for loop of S.

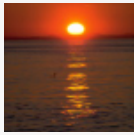


Nested-Loop Join

- Evaluate the condition join R $\bowtie_C$ S

- **for each** tuple t_R **in** R **do begin**
  for each tuple t_S **in** S **do begin**
  check whether pair (t_R, t_S) satisfies join condition
  if they do, add $t_R \circ t_S$ to the result
  end
  end

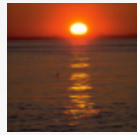
- R is called the *outer* and S the *inner* relation of the join.



Cost Analysis of Nested-Loop Join

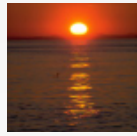
- The nested-loop join algorithm is expensive in nature.
- It is because it computes and examines each pair of tuples in the two given relations.

- Worst case: db buffer can only hold one block of each relation
  $\Longrightarrow B_R + N_R * B_S$ disk accesses
- Best case: both relations fit into db buffer
  $\Longrightarrow B_R + B_S$ disk accesses.



Block Nested-Loop Join

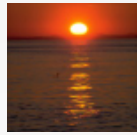
- A major reduction in the number of disk accesses can be done if we process the relations on a per block basis rather than per tuple basis.
- Block Nested-Loop Join is a variant of nested-loop join in which each block of the inner relation is paired with each block of the outer relation.
- Within each pair of blocks, the block nested-loop join pairs each tuple of one block with each tuple in the other block to produce all pairs of matching tuples.
- All pairs of tuples that satisfy the given join condition are added to the result of the join.



Block Nested-Loop Join

- Evaluate the condition join R $\bowtie_C$ S

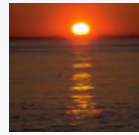
- **for each** block B_R **of** R **do begin**
  for each block B_S **of** S **do begin**
  for each tuple t_R **in** B_R **do**
  for each tuple t_S **in** B_S **do**
  check whether pair (t_R, t_S)
  satisfies join condition
  if they do, add $t_R \circ t_S$ to the result
  end end end end



Cost Analysis of Block Nested-Loop Join

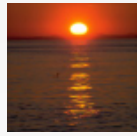
- There is a major difference between the cost of block nested-loop join and nested loop-join algorithm.
- **Worst case:** In the worst case of block nested-loop join, each block in the inner relation S is read only once for each block in the outer relation R. (Instead of once for each tuple in the outer relation as in the Nested-loop join)
- **Best case:** In the best case, the inner relation fits entirely into memory.

- Worst case: db buffer can only hold one block of each relation
  $\Longrightarrow$ $B_R + B_R * B_S$ disk accesses.
- Best case: both relations fit into db buffer
  $\Longrightarrow$ $B_R + B_S$ disk accesses.



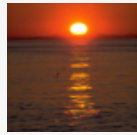
Index Nested-Loop Join

- If an index is present on the join attribute of the inner loop, we can replace the file scans with highly efficient index lookups. Such type of join method is known as **indexed nested-loop join**.
- Thus, an index nested loop join is a database join algorithm that uses an index to speed up the process of joining two tables
- Indexes provide an optimized technique for improving the performance of the nested-loop join.
- For each tuple t_R in the outer relation R, use the index to lookup tuples in S that satisfy join condition with t_R
- If indexes are available on both R and S, use the one with the fewer tuples as the outer relation.



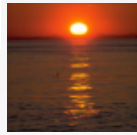
Cost Analysis of Index Nested-Loop Join

- Worst case: db buffer has space for only one page of R and one page of the index associated with S:
  - B_R disk accesses to read R, and for each tuple in R, perform index lookup on S.
  - Cost of the join: $B_R + N_R * c$, where c is the cost of a single selection on S using the join condition.



Example of Cost Estimate of Index Nested Loop Join

- **Example:**
  - Compute CUSTOMERS ⋈ ORDERS, with CUSTOMERS as the outer relation.
  - Let ORDERS have a primary B^+-tree index on the join-attribute CName, which contains 20 entries per index node
  - Since ORDERS has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data records (based on tuple identifier).
  - Since $N_{CUSTOMERS}$ is 5,000, the total cost is 250 + 5000 ∗ 5 = 25,250 disk accesses.
  - This cost is lower than the 100,250 accesses needed for a block nested-loop join.



Sort-Merge Join

- The sort-merge join is a common join algorithm in database systems using sorting.
- The join predicate needs to be an equality join predicate.
- This algorithm consists of two phases:
  - **Sorting** both the relations on the join attribute
  - **Merging** the sorted relations by scanning them sequentially and looking for qualifying tuples that satisfy the join condition
- The algorithm sorts both relations on the join attribute and then merges the sorted relations by scanning them sequentially and looking for qualifying tuples.