CHAPTER ONE

Concepts for object oriented database

Introduction

Database systems that were based on the object data model were known originally as object-oriented databases (OODBs). But, are now referred to as **object databases** (**ODBs**).Traditional data models and systems, such as network, hierarchical, and relational have been quite successful in developing the database technologies required for many traditional business database applications. However, they have certain shortcomings when more complex database applications must be designed and implemented like databases for engineering design and manufacturing, biological and other sciences, telecommunications, geographic information systems, and multimedia. These ODBs were developed for applications that have requirements requiring more complex structures for stored objects. A key feature of object databases is the power they give the designer to specify both the ***structure*** of complex objects and the ***operations*** that can be applied to these objects.

Another reason for the creation of object-oriented databases is the vast increase in the use of object-oriented programming languages for developing software applications. Databases are fundamental components in many software systems, and traditional databases are sometimes difficult to use with software applications that are developed in an object-oriented programming language such as C++ or Java. Object databases are designed so they can be directly or seamlessly integrated with software that is developed using object-oriented programming languages.

Overview of object oriented concepts

The term object-oriented abbreviated OO or O-O has its origins in OO programming languages (OOPLs). Today OO concepts are applied in the areas of databases, software engineering, knowledge bases, artificial intelligence, and computer systems in general.
An object in relational database concept is a data structure used to either store or reference data.
Example: table (most commonly used), indexes, stored procedures, sequences, views.
An **object** typically has two components: these are:

1. state (value) and
2. Behavior (operations).

It can have a complex data structure as well as specific operations defined by the programmer. Objects in an OOPL exist only during program execution. Therefore, they are called transient objects.
An OO database can extend the existence of objects. So that they are stored permanently in a database, and hence the objects become persistent objects that exist beyond program termination and can be retrieved later and shared by other programs. In other words, OO databases store persistent objects permanently in secondary storage and allow the sharing of these objects among multiple programs and applications. This requires the incorporation of other well-known features of database management systems, such as:

- Indexing mechanisms: to efficiently locate the objects.
- Concurrency control: to allow object sharing among concurrent programs, and
- Recovery from failures.

An OO database system will typically interface with one or more OO programming languages to provide persistent and shared object capabilities.

Object identity, object structure, and type constructors

One goal of an ODB is to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be identified and operated upon. Hence, a **unique identity** is assigned to each independent object stored in the database. This unique identity is typically implemented via a unique, system-generated **object identifier (OID)**. The value of an OID may not be visible to the external user but is used internally by the system to identify each object uniquely and to create and manage inter object references. The OID can be assigned to program variables of the appropriate type when needed.
The main property required of an OID is that

A. It be **immutable:** that is the OID value of a particular object should not change. This preserves the identity of the real-world object being represented. Hence, an ODMS must have some mechanism for generating OIDs and preserving the immutability property.

B. It is also desirable that **each OID be used only once;** that is, even if an object is removed from the database, its OID should not be assigned to another object.

These two properties imply that the OID should not depend on any attribute values of the object, since the value of an attribute may be changed or corrected. We can compare this with the relational model, where each relation must have a primary key attribute whose value identifies each tuple uniquely. If the value of the primary key is changed, the tuple will have a new identity, even though it may still represent the same real world object. Alternatively, a real-world object may have different names for key attributes in different relations, making it difficult to ascertain that the keys represent the same real-world object.

Example: using the Emp_id of an EMPLOYEE in one relation and the Ssn in another.

It is also inappropriate to base the OID on the physical address of the object in storage since the physical address can change after a physical reorganization of the database. However, some early ODMSs have used the physical address as the OID to increase the efficiency of object retrieval. If the physical address of the object changes, an indirect pointer can be placed at the former address, which gives the new physical location of the object. It is more common to use long integers as OIDs and then to use some form of hash table to map the OID value to the current physical address of the object in storage.

Another feature of ODBs is that objects may have a type structure of arbitrary complexity in order to contain all of the necessary information that describes the object. In contrast, in traditional database systems, information about a complex object is often scattered over many relations or records, leading to loss of direct correspondence between a real-world object and its database representation.

In ODBs, a complex type may be constructed from other types by nesting of **type constructors**. The three most basic constructors are:

1. **One type (atom) constructor:** This includes the basic built-in data types of the object model, which are similar to the basic types in many programming languages: integers, strings, floating-point numbers, enumerated types, Booleans, and so on. These basic data types are called **single valued** or **atomic** types, since each value of the type is considered an atomic (indivisible) single value.

2. **Struct (tuple) constructor:** This can create standard structured types, such as the tuples (record types) in the basic relational model. A structured type is made up of several components and is also sometimes referred to as a compound or composite type. More accurately, the struct constructor is not considered to be a type, but rather a **type generator**, because many different structured types can be created.

For example: two different structured types that can be created are: struct Name<FirstName: string, MiddleInitial: char, LastName: string>, and
struct CollegeDegree<Major: string, Degree: string, Year: date>.

Notice that the type constructors' atom and struct are the only ones available in the original (basic) relational model.

3. **Collection (or multivalued) type constructors:** include the **set (T)**, **list (T)**, **bag (T)**, **array (T)**, and **dictionary (K, T)** type constructors. These allow part of an object or literal value to include a collection of other objects or values when needed. These constructors are also considered to be **type generators** because many different types can be created.

For example: set (string), set (integer), and set (Employee) are three different types that can be created from the set type constructor. All the elements in a particular collection value must be of the same type. For example, all values in a collection of type set (string) must be string values.

Encapsulation of operations, methods, and persistence

Encapsulation of Operations: The concept of encapsulation is one of the main characteristics of OO languages and systems. It is also related to the concepts of abstract data types and information hiding in programming languages. In traditional database models and systems this concept was not applied, since it is customary to make the structure of database objects visible to users and external programs. In these traditional models, a number of generic database operations are applicable to objects of all types.

For example: in the relational model, the operations for selecting, inserting, deleting, and modifying tuples are generic and may be applied to any relation in the database. The relation and its attributes are visible to users and to external programs that access the relation by using these operations. The concept of encapsulation is applied to database objects in ODBs by defining the **behavior** of a type of object based on the **operations** that can be externally applied to objects of that type. Some operations may be used to create (insert) or destroy (delete) objects; other operations may update the object state; and others may be used to retrieve parts of the object state or to apply some calculations. Still other operations may perform a combination of retrieval, calculation, and update. In general, the **implementation** of an operation can be specified in a general-purpose programming language that provides flexibility and power in defining the operations.

Type hierarchies and inheritance

Inheritance allows the definition of new types based on other predefined types, leading to a type (class) hierarchy.

A type is defined by assigning it a type name and then defining a number of attributes (instance variables) and operations (methods) for the class. In the simplified model we use in this section, the attributes and operations are together called functions, since attributes resemble functions with zero arguments. A function name can be used to refer to the value of an attribute or to refer to the resulting value of an operation (method). We use the term function to refer to both attributes and operations, since they are treated similarly in a basic introduction to inheritance.

A class in its simplest form has a class name and a list of visible (public) functions. When specifying a class in this section, we use the following format, which does not specify arguments of functions, to simplify the discussion: CLASS_NAME: function, function, … , function **Example:** a class that describes characteristics of a PERSON may be defined as follows: PERSON: Name, Address, Birth_date, Age, Ssn In the PERSON type, the Name, Address, Ssn, and Birth_date functions can be implemented as stored attributes, whereas the Age function can be implemented as an operation that calculates the Age from the value of the Birth_date attribute and the current date. The concept of subtype is useful when the designer or user must create a new type that is similar but not identical to an already defined type. The subtype then inherits all the functions of the predefined type, which is referred to as the super type.

Example: suppose that we want to define two new types EMPLOYEE and STUDENT as follows: EMPLOYEE: Name, Address, Birth_date, Age, Ssn, Salary, Hire_date, Seniority STUDENT: Name, Address, Birth_date, Age, Ssn, Major, Gpa Since both STUDENT and EMPLOYEE include all the functions defined for PERSON plus some additional functions of their own, we can declare them to be subtypes of PERSON. Each will inherit the previously defined functions of PERSON—namely, Name, Address, Birth_date, Age, and Ssn. For STUDENT, it is only necessary to define the new (local) functions Major and Gpa, which are not inherited. Presumably, Major can be defined as a stored attribute, whereas Gpa may be implemented as an operation that calculates the student's grade point average by accessing the Grade values that are internally stored (hidden) within each STUDENT object as hidden attributes. For EMPLOYEE, the Salary and Hire_date functions may be stored attributes, whereas Seniority may be an operation that calculates Seniority from the value of Hire_date. Therefore, we can declare EMPLOYEE and STUDENT as follows: EMPLOYEE subtype-of PERSON: Salary, Hire_date, Seniority STUDENT subtype-of PERSON: Major, Gpa In general, a subtype includes all of the functions that are defined for its super type plus some additional functions that are specific only to the subtype. Hence, it is possible to generate a type hierarchy to show the supertype/subtype relationships among all the types declared in the system.

CHAPTER TWO

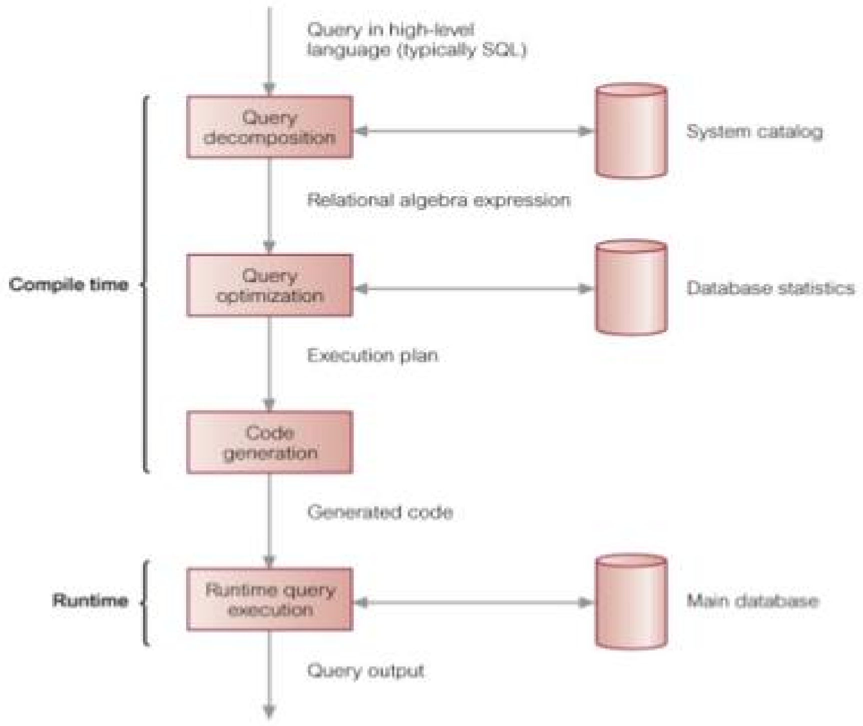
Query Processing and Optimization

Introduction

Query optimization: The process of choosing a suitable execution strategy for processing a query.

- A query typically has many possible execution strategies, and the process of choosing a suitable one for the processing a query is known as query optimization
- Query optimization is an activity conducted by a query optimizer in a DBMS to select the best available strategy for executing the query
- High-level query languages like SQL (for RDBMS) are more declarative in nature because they specifies what the intended results of the query are, rather than identifying the details of how the result should be obtained.
- SQL queries are translated into corresponding relational algebra for the optimization.

Phases of Query Processing



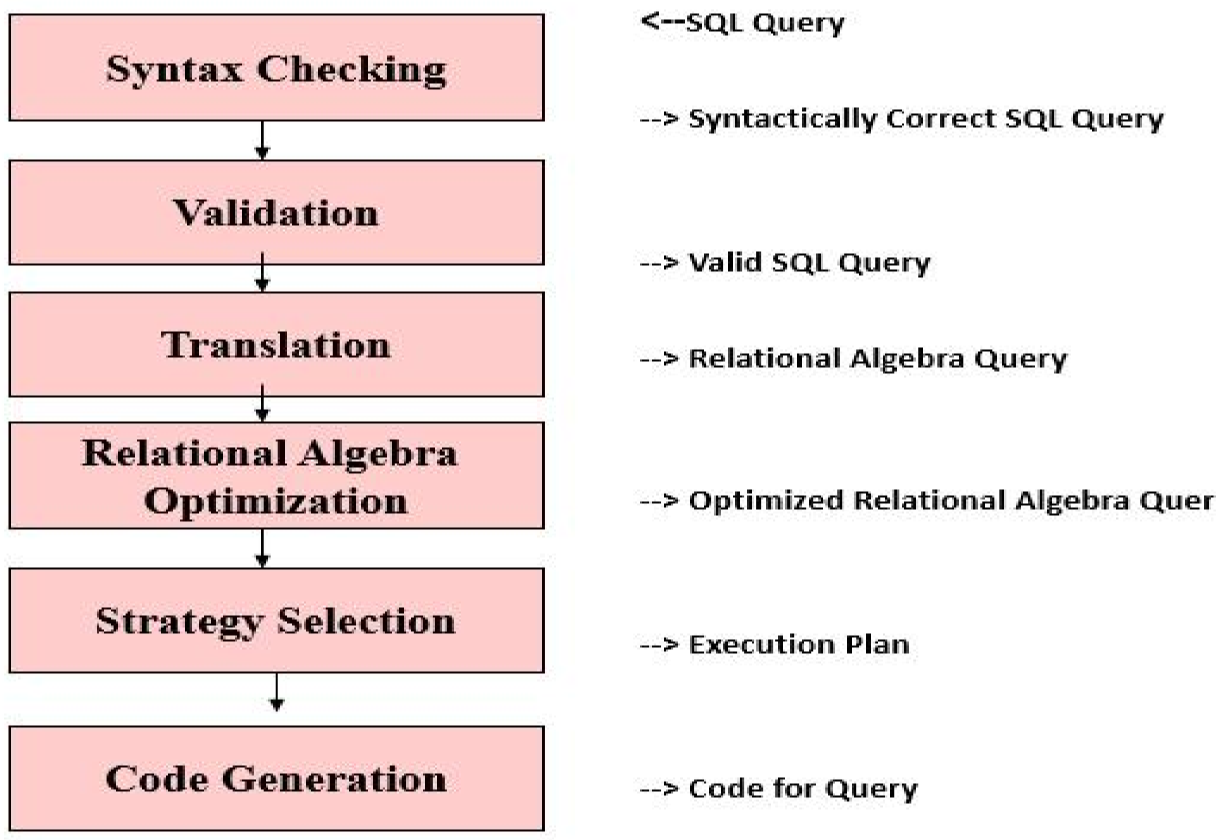
Two internal representations of a query:

- ✓ Query Tree
- ✓ Query Graph

Code can be:

- ✓ Executed directly (interpreted mode)
- ✓ Stored and executed later whenever needed (compiled mode).

Query Optimization...



Translating SQL queries into Relational Algebra

An SQL query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses if these are part of the block.

Nested queries within a query are identified as separate query blocks.

Inner blocks could be:

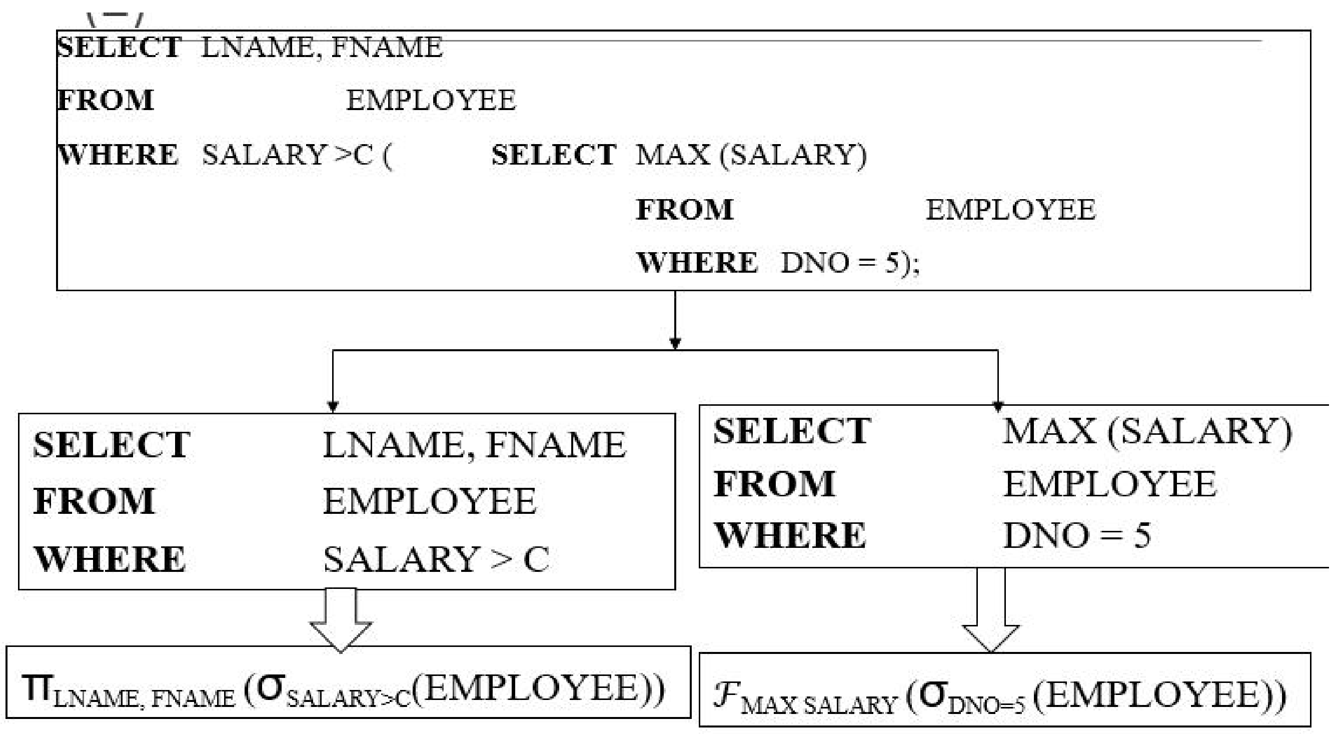
- uncorrelated nested query- the inner block needs to be evaluated only once
- correlated nested query-a tuple variable from the outer block appears in the WHERE-clause of the inner block

SQL queries are decomposed into query blocks, which form the basic units that can be translated into the algebraic operators and optimized.

The query optimizer would then choose an execution plan for each block.

SQL query is translated to an equivalent extended relational algebra expression, represented as a query tree data structure that is then optimized.

Example



Represent Relational Algebra by Query Tree

A **query tree** is a tree data structure that corresponds to an extended relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and it represents the relational algebra operations as internal nodes.

An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The order of execution of operations *starts at the leaf nodes*, which represents the input database relations for the query, and *ends at the root node*, which represents the final operation of the query. The execution terminates when the root node operation is executed and produces the result relation for the query.

Using Heuristic in Query Optimization

Process for heuristics optimization:

1. The parser of a high-level query generates an initial internal representation;

2. Apply heuristics rules to optimize the internal representation.
3. A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.

The main heuristic is to apply first the operations that reduce the size of intermediate results.

Example: Apply SELECT and PROJECT operations before applying the JOIN or other binary operations.

Query tree: A tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as **leaf nodes** of the **tree**, and represents the relational algebra operations as internal nodes.

An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation.

Query graph: A graph data structure that corresponds to a relational calculus expression. It does *not* indicate an order on which operations to perform first. There is only a *single* graph corresponding to each query.

Example: Query "Find the last names of employees born after 1957 who work on a project named 'Aquarius'."

SQL

SELECT LNAME

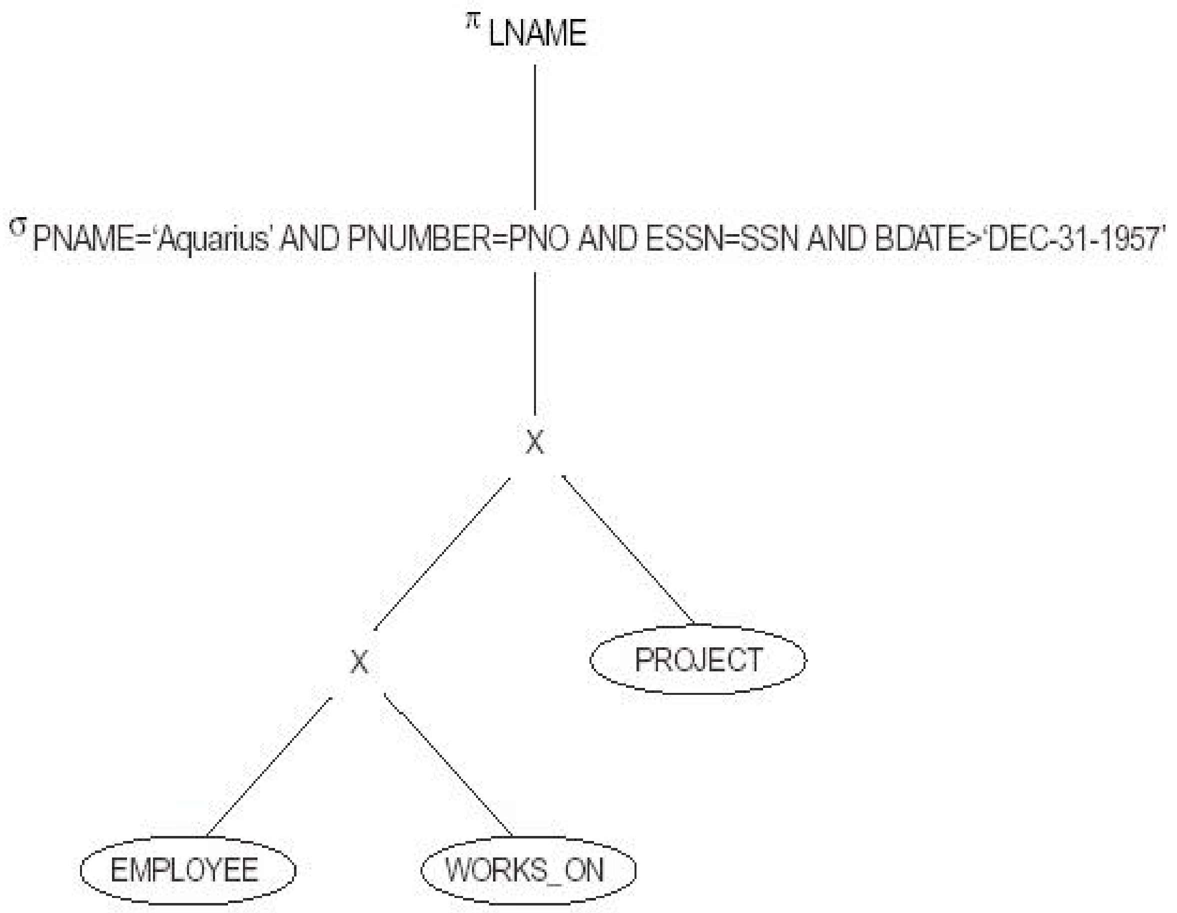
FROM EMPLOYEE, WORKS_ON, PROJECT

WHERE PNAME='Aquarius' **AND** PNUMBER=PNO **AND** ESSN=SSN **AND** BDATE.'1957-12-31';

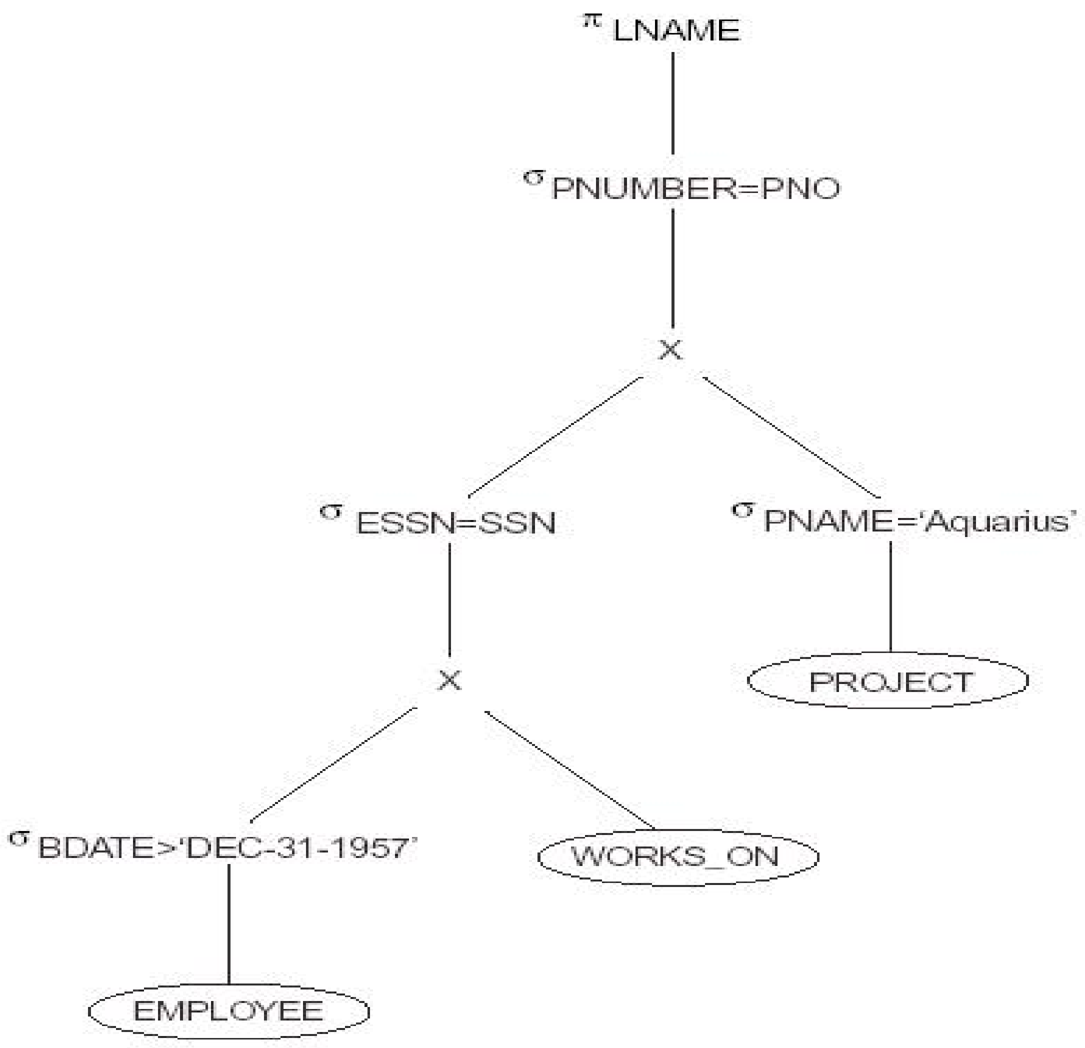
RA: try you?

There are five different alternative query trees for this SQL in the next page.

(a)

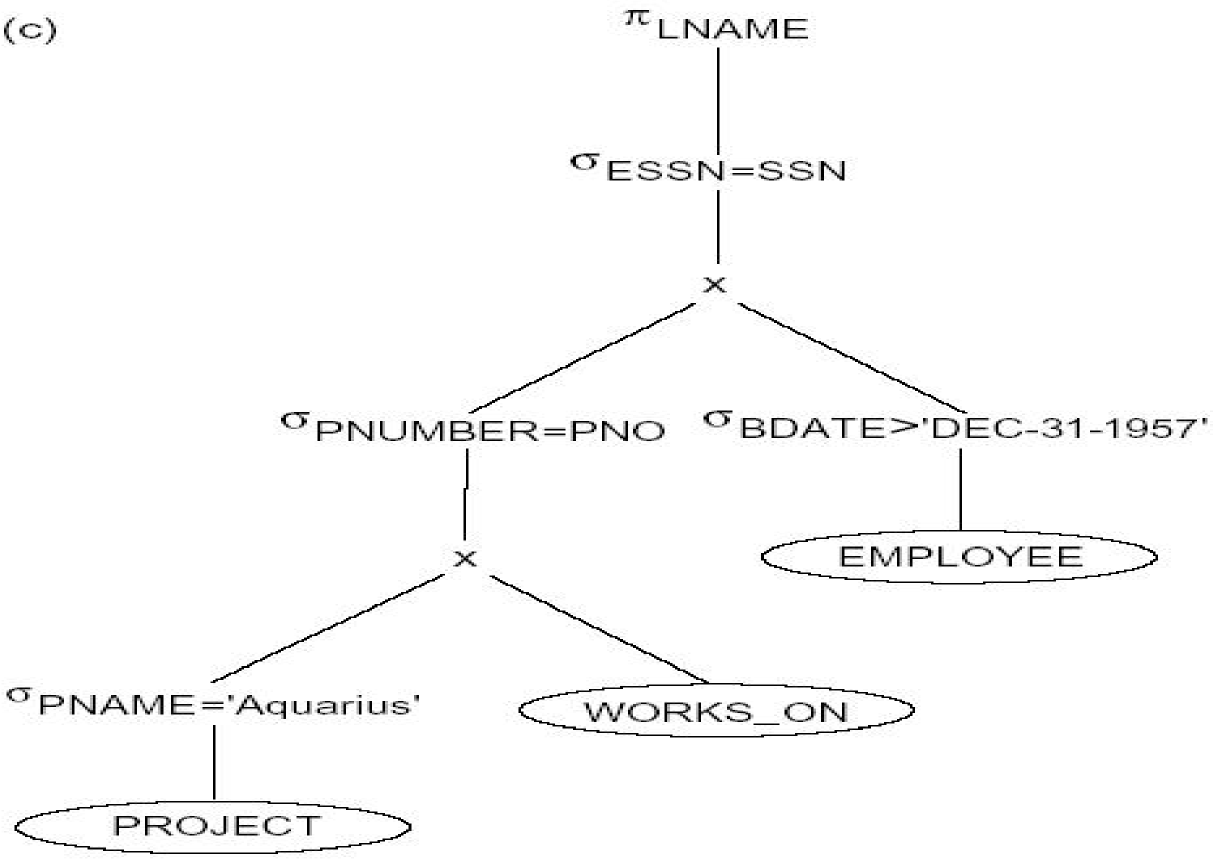

π LNAME
σ PNAME='Aquarius' AND PNUMBER=PNO AND ESSN=SSN AND BDATE>'DEC-31-1957'
X
X
PROJECT
EMPLOYEE
WORKS_ON

(b)

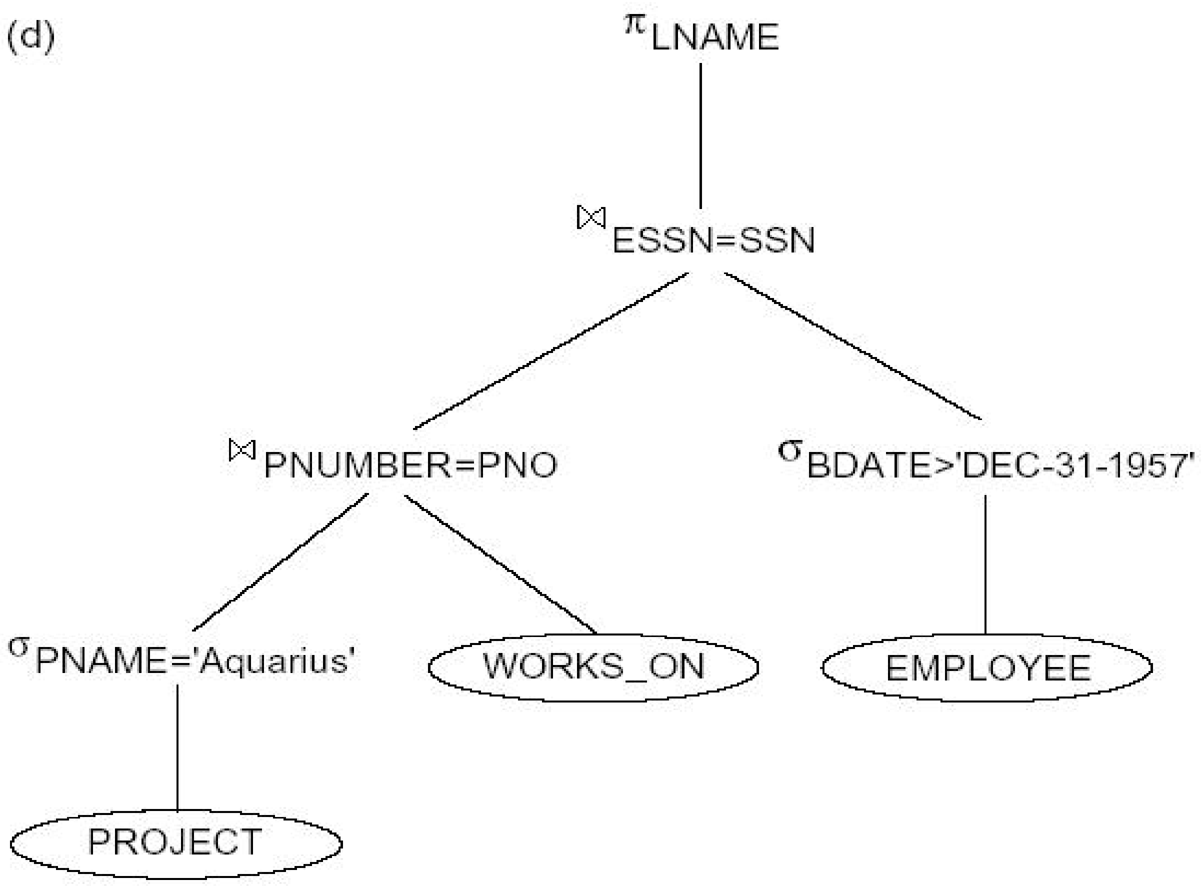


π LNAME
σ PNUMBER=PNO
X
σ ESSN=SSN
σ PNAME='Aquarius'
X
PROJECT
σ BDATE>'DEC-31-1957'
WORKS_ON
EMPLOYEE

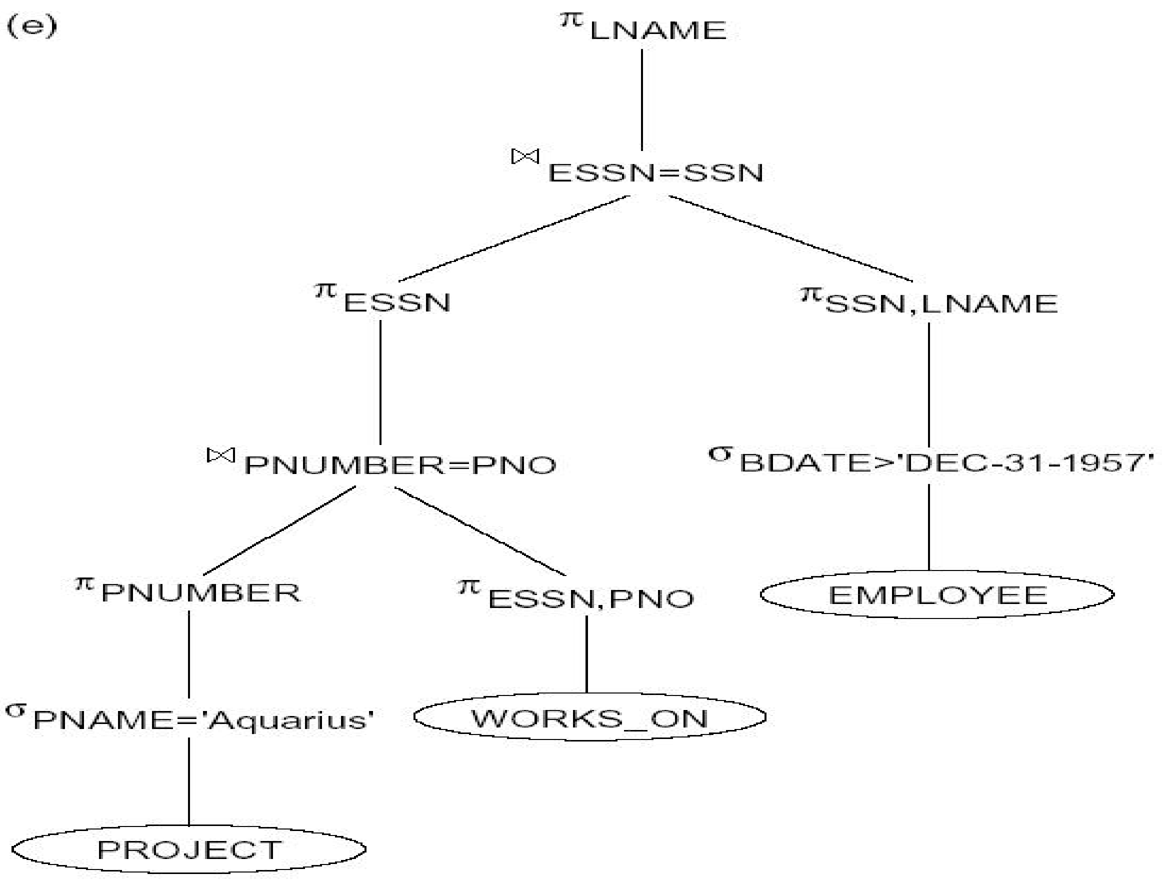
(c)


π_{LNAME}
$\sigma_{ESSN=SSN}$
×
$\sigma_{PNUMBER=PNO}$
$\sigma_{BDATE>'DEC-31-1957'}$
×
EMPLOYEE
$\sigma_{PNAME='Aquarius'}$
WORKS_ON
PROJECT

(d)


π_{LNAME}
$\bowtie_{ESSN=SSN}$
$\bowtie_{PNUMBER=PNO}$
$\sigma_{BDATE>'DEC-31-1957'}$
$\sigma_{PNAME='Aquarius'}$
WORKS_ON
EMPLOYEE
PROJECT

(e)



General Transformation Rules for Relational Algebra Operations:

1. Cascade of s: A conjunctive selection condition can be broken up into a cascade (sequence) of individual s operations:

   $s_{\text{c1 AND c2 AND ... AND cn}}(R) = s_{c1}\,(s_{c2}\,(\ldots(s_{cn}(R))\ldots)\,)$

2. Commutativity of s: The s operation is commutative:

   $s_{c1}\,(s_{c2}(R)) = s_{c2}\,(s_{c1}(R))$

3. Cascade of p: In a cascade (sequence) of p operations, all but the last one can be ignored:

   $p_{List1}\,(p_{List2}\,(\ldots\,(p_{Listn}(R))\ldots)) = p_{List1}(R)$

4. Commuting s with p: If the selection condition c involves only the attributes A1, ..., An in the projection list, the two operations can be commuted:

   $p_{A1,\,A2\ldots\,An}\,(s_c\,(R)) = s_c\,(p_{A1,\,A2\ldots\,An}\,(R))$

(Other transformation rules are reading assignment for you)

Basic Algorithms for Executing Query Operations

It is all about algorithms that implements the relational algebra operations. These are:

- ✓ External sorting
- ✓ Select
- ✓ Join
- ✓ Project
- ✓ Set operation

External sorting: refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files.

The typical external sorting algorithm uses a sort-merge strategy, which starts by sorting small subfiles called runs of the main file and then merges the stored runs, creating larger sorted subfiles that are merged in turn.

The basic sort-merge algorithm consists of two phases. These are:

- ✓ Sorting phase and
- ✓ Merging phase

In the **sorting phase,** runs (portions or pieces) of the file that can fit in the available buffer space are read into main memory, sorted using an internal sorting algorithm, and written back to disk as temporary sorted subfiles (or runs). The size of a run and **number of initial runs** is dictated by the **number of file blocks (*b*)** and the **available buffer space.**

For example: if = 5 blocks and the size of the file b = 1024 blocks, then =, or 205 initial runs each of size 5 blocks (except the last run which will have 4 blocks). Hence, after the sort phase, 205 sorted runs are stored as temporary subfiles on disk.

In the **merging phase,** the sorted runs are merged during one or more **passes.** The **degree of merging** is the number of runs that can be merged together in each pass. In each pass, one buffer block is needed to hold one block from each of the runs being merged, and one block is needed for containing one block of the merge result.

Implementing Select Operation

Search methods for selection

- ✓ Linear search
- ✓ Binary search
- ✓ Using a primary index

- ✓ Using a primary index to retrieve multiple records
- ✓ Using a clustering index to retrieve multiple records
- ✓ Using a secondary index on an equality comparison
- ✓ Conjunctive selection using an individual index
- ✓ Conjunctive selection using a composite index
- ✓ Conjunctive selection by intersection of record pointers

When the optimizer is choosing between multiple simple conditions in a conjunctive select condition, it typically considers the selectivity of each condition.

Selectivity (S): the ratio of the number of records that satisfy the condition to the total number of records in the relation.

For example, for an equality condition on a key attribute of relation r(R), s= 1/r(R).

For an equality condition on an attribute with I distinct values, s is estimated by (|r(R)|/i) /r(R) or 1/i.

The number of records satisfying a selection condition with selectivity s is estimated to be| r(R)| * s. The smaller this estimate is, the higher the desirability of using that condition first to retrieve records.

Implementing Join Operation

Our focus is in EQUIJOIN and NATURAL JOIN.

We base on the join operation R $_{A=B}$ S for the discussion of the algorithms of join.

Where R and S are the relations to be joined and A and B are domain-compatible attributes of R and S respectively.

Methods for Implementing Joins:

- ✓ Nested-loop join
- ✓ Single-loop join
- ✓ Sort-merge join
- ✓ Hash join

Nested-loop Join: For each record t in R (outer loop), retrieve every records from S (inner loop) and test whether the two records satisfy the join condition t[A] = s[B].

Single-loop join (using an access structure to retrieve the matching records):If an index exists for one of the two join attributes –say, B of S- retrieve each record t in R, one at a time (single loop), and then use the access structure to retrieve directly all matching records s from S that satisfy s[B] = t[A].

Sort-merge join: The records of R and S need to be sorted by value of the join attributes (or external sorting needs to be applied) before applying this algorithm. (Assuming A or B or both are key attributes).Pairs of file blocks are copied into memory buffers in order and the records of each file are scanned only once each for matching with the other file.

Hash join: The records of files R and S are both hashed to the same hash file, using the same hashing function on the join attributes A of R and B of S as hash keys. This algorithm assumes that the smaller of the two files fits entirely into memory buckets after the first phase.

First, a single pass through the file with fewer records (say, R) hashes its records to the hash file buckets; this is called the partitioning phase since the records of R are partitioned into the hash buckets.

In the second phase, called the probing phase, a single pass through the other file (S) then hashes each of its records to probe the appropriate bucket, and that record is combined with all matching records from R in that bucket.

Implement Project Operation

Let us consider the A PROJECT operation $P_{<\text{attribute list}>}$ (R).

If <attribute list> includes a key of relation R, implementation of the above PROJECT is straightforward.

The result of the operation will have the same number of tuples as R, but with only the values for the attributes in <attribute list> in each tuple.

If <attribute list> does not include a key of R, duplicate tuples must be eliminated.

Duplicates can be eliminated either by sorting the result and eliminate duplicate tuples that appear consecutively after sorting or using hashing.

Recall that in SQL queries, the default is not to eliminate duplicates and we are supposed to use the keyword DISTINCT to eliminate duplicate records.

Implementing Set Operation

Set operations include UNION, INTERSECTION, SET DIFFERENCE and CARTESIAN PRODUCT.

These operations are sometimes expensive to implement. The CARTESIAN PRODUCT R X S (R with n records and j attributes; S with m records and k attributes), for example, results in n * m records and j + k attributes. Hence it is important to avoid this operation and to substitute other equivalent operations during query optimization.

The three set operation (UNION, INTERSECTION and SET DIFFERENCE) apply only to union-compatible relations.

- Relations that have the same number of attributes and the same attribute domains.
- Can be implemented by using a variation of sort-merge techniques or by using hashing.

Implementation of UNION: R D S

Using sort-merge technique:

- Sort the two relations on the same attributes.
- Scan and merge both sorted files concurrently.
    - Whenever the same tuple exits in both relations, keep only one of the tuple.

Using hashing:

- First hash (partition) the records of R.
- Then, hash (probe) the records of S, but do not insert duplicate records in the buckers.

Implementation of INTERSECTION: R C S

Using sort-merge technique:

- Sort the two relations on the same attributes.
- Scan both sorted files concurrently.
    - Whenever the same tuple exits in both relations, keep the tuple.

Using hashing:

- First hash (partition) the records of R to the hash table.
- Then, while hashing each record of S, probe to check if an identical record from R is found in the bucket, and if so add the record to the result file.

Implementation of SET DIFFERENCE: R – S

Using sort-merge technique:

- Sort the two relations on the same attributes.
- Scan both sorted files concurrently and.
    - Add the tuples that are available only in R.

Using hashing:

- First hash (partition) the records of R to the hash table.
- Then, while hashing each record of S, probe to check if an identical record from R is found in the bucket, and if so remove that record from the bucket.

Implementing Aggregate Operations

COUNT, AVERAGE, SUM

Dense index (if there is an index entry for every record in the main file).

- Index search can be used.
- The associated computation would be applied to the values in the index.

Non-dense index: The actual number of records associated with each index entry must be used for a correct computation (except for COUNT DISTINCT, where the number of distinct values can be counted from the index itself).

When a GROUP BY clause is used in a query, the aggregate operator must be applied separately to each group of tuples.

The table must first be partitioned into subsets of tuples, where each partition (group) has the same value for the grouping attributes.

The aggregate operators are MIN, MAX, COUNT, AVERAGE and SUM

Either full table scan or index search could be used.

MAX: SELECT MAX (Salary) FROM Employee;

If an (ascending) index on Salary exists for the Employee relation, then the optimizer can decide on using the index to search for the largest value by following the rightmost pointer in each index node from the root to the rightmost leaf.

The MIN aggregate can be handled in a similar manner, except that the leftmost pointer is followed from the root to the leftmost leaf.

Implementing Outer Join

Consider the following SQL query that is based on LEFT OUTER JOIN.

SELECT name, departmentName

FROM (Employee LEFT OUTER JOIN DEPARTMENT ON Dno = Dnumber);

OUTER JOIN can be implemented either by: executing a combination of relational algebra operators. For example, the left outer join operation shown above is equivalent to the following sequence of relational operations:

1. Compute the (inner) JOIN of the Employee and Department tables.
2. Find the Employee tuples that do not appear in the inner JOIN result.
3. Pad each tuple in the #2 with a null Dname field.
4. Apply the UNION operation to #1 and #2 to produce the LEFT OUTER JOIN result.