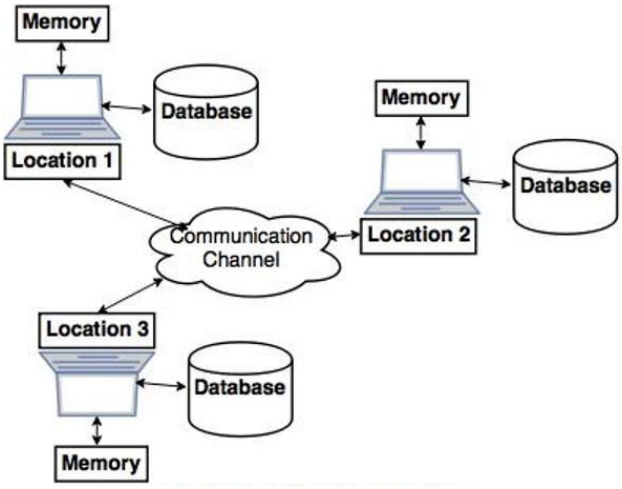
UNIT V ADVANCED TOPICS

Distributed Databases: Architecture, Data Storage, Transaction Processing – Object-based Databases: Object Database Concepts, Object-Relational features, ODMG Object Model, ODL, OQL – XML Databases: XML Hierarchical Model, DTD, XML Schema, XQuery – Information Retrieval: IR Concepts, Retrieval Models, Queries in IR systems.

DISTRIBUTED DATABASES

A distributed database is a database in which not all storage devices are attached to a common processor. It may be stored in multiple computers, located in the same physical location; or may be dispersed over a network of interconnected computers.

- Distributed database is a system in which storage devices are not connected to a common processing unit.
- Database is controlled by Distributed Database Management System and data may be stored at the same location or spread over the interconnected network. It is a loosely coupled system.
- Shared nothing architecture is used in distributed databases.



Distributed Database System

- Communication channel is used to communicate with the different locations and every system has its own memory and database.

Reliability: In distributed database system, if one system fails down or stops working for some time another system can complete the task.

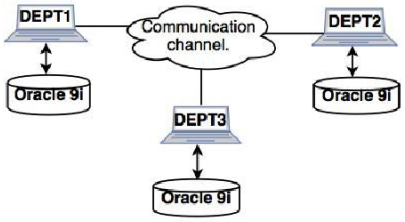
Availability: In distributed database system reliability can be achieved even if sever fails down. Another system is available to serve the client request.

Performance: Performance can be achieved by distributing database over different locations. So the databases are available to every location which is easy to maintain.

1. Homogeneous distributed databases system:

- Homogeneous distributed database system is a network of two or more databases (With same type of DBMS software) which can be stored on one or more machines.
- So, in this system data can be accessed and modified simultaneously on several databases in the network. Homogeneous distributed system are easy to handle.

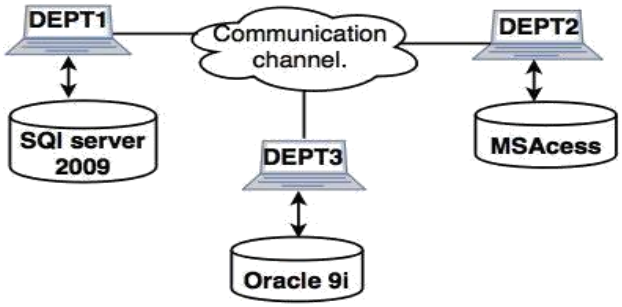
  Example: Consider that we have three departments using Oracle-9i for DBMS. If some changes are made in one department then, it would update the other department also.



Homogeneous distributed system

2. Heterogeneous distributed database system.

- Heterogeneous distributed database system is a network of two or more databases with different types of DBMS software, which can be stored on one or more machines.
- In this system data can be accessible to several databases in the network with the help of generic connectivity (ODBC and JDBC).
- **Example:** In the following diagram, different DBMS software are accessible to each other using ODBC and JDBC.



Heterogeneous distributed system

Distributed DBMS Architectures

DDBMS architectures are generally developed depending on three parameters −

Distribution − It states the physical distribution of data across the different sites.

Autonomy − It indicates the distribution of control of the database system and the degree to which each constituent DBMS can operate independently.

Heterogeneity − It refers to the uniformity or dissimilarity of the data models, system components and databases.

Architectural Models

Some of the common architectural models are –

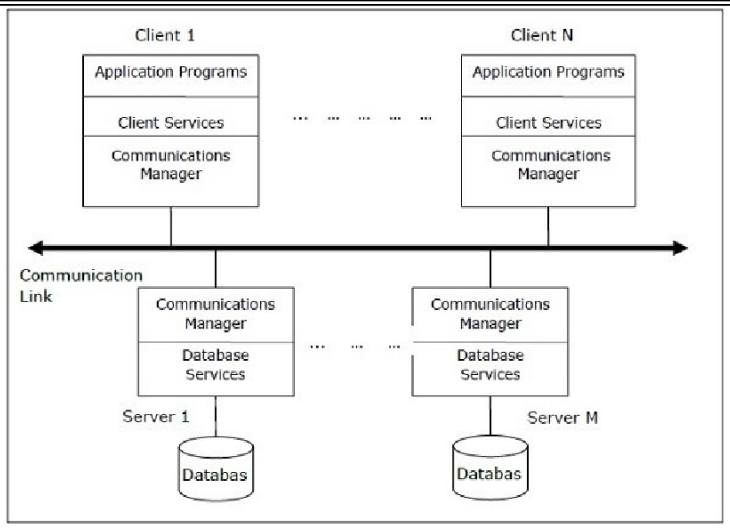
- Client - Server Architecture for DDBMS
- Peer - to - Peer Architecture for DDBMS
- Multi - DBMS Architecture

Client - Server Architecture for DDBMS

This is a two-level architecture where the functionality is divided into servers and clients. The server functions primarily encompass data management, query processing, optimization and transaction management. Client functions include mainly user interface. However, they have some functions like consistency checking and transaction management.

The two different client - server architecture are −

- Single Server Multiple Client
- Multiple Server Multiple Client (shown in the following diagram)



Peer- to-Peer Architecture for DDBMS

In these systems, each peer acts both as a client and a server for imparting database services. The peers share their resource with other peers and co-ordinate their activities.

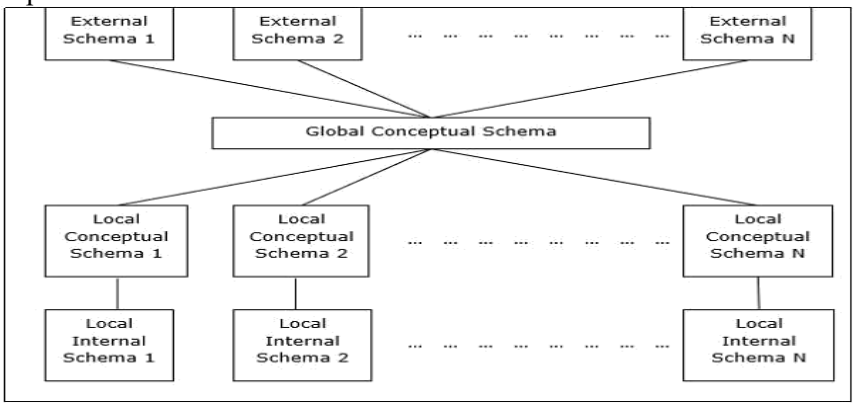
This architecture generally has four levels of schemas −

Global Conceptual Schema − Depicts the global logical view of data.

Local Conceptual Schema − Depicts logical data organization at each site.

Local Internal Schema − Depicts physical data organization at each site.

External Schema − Depicts user view of data.



Multi - DBMS Architectures

This is an integrated database system formed by a collection of two or more autonomous database systems. Multi-DBMS can be expressed through six levels of schemas −

Multi-database View Level − Depicts multiple user views comprising of subsets of the integrated distributed database.

Multi-database Conceptual Level − Depicts integrated multi-database that comprises of global logical multi-database structure definitions.

Multi-database Internal Level − Depicts the data distribution across different sites and multi-database to local data mapping.

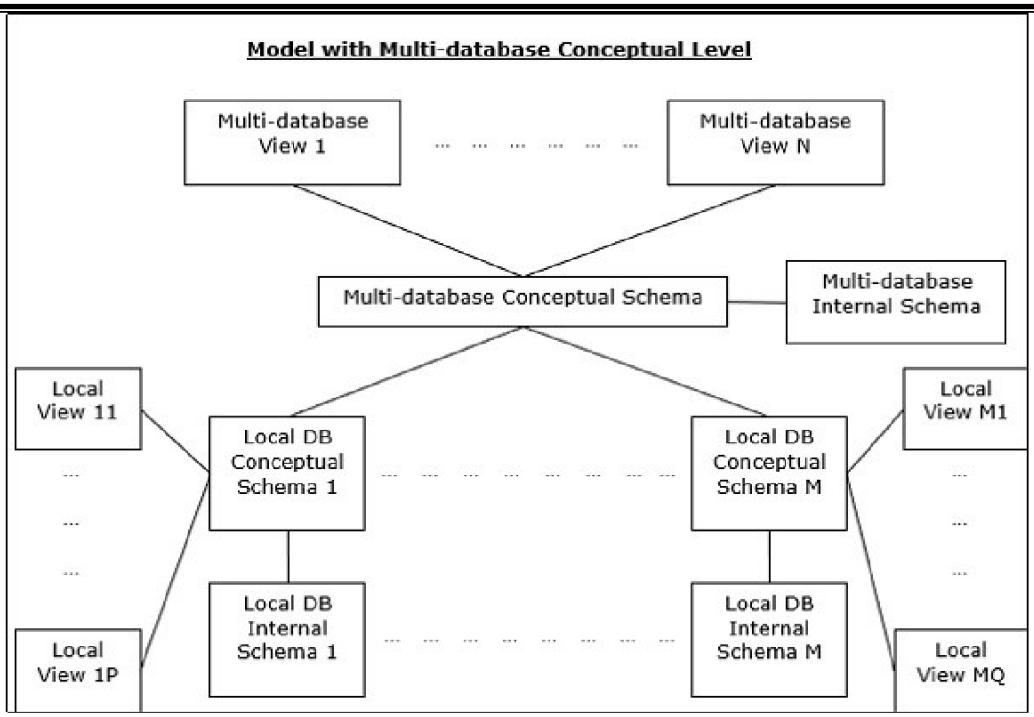
Local database View Level − Depicts public view of local data.

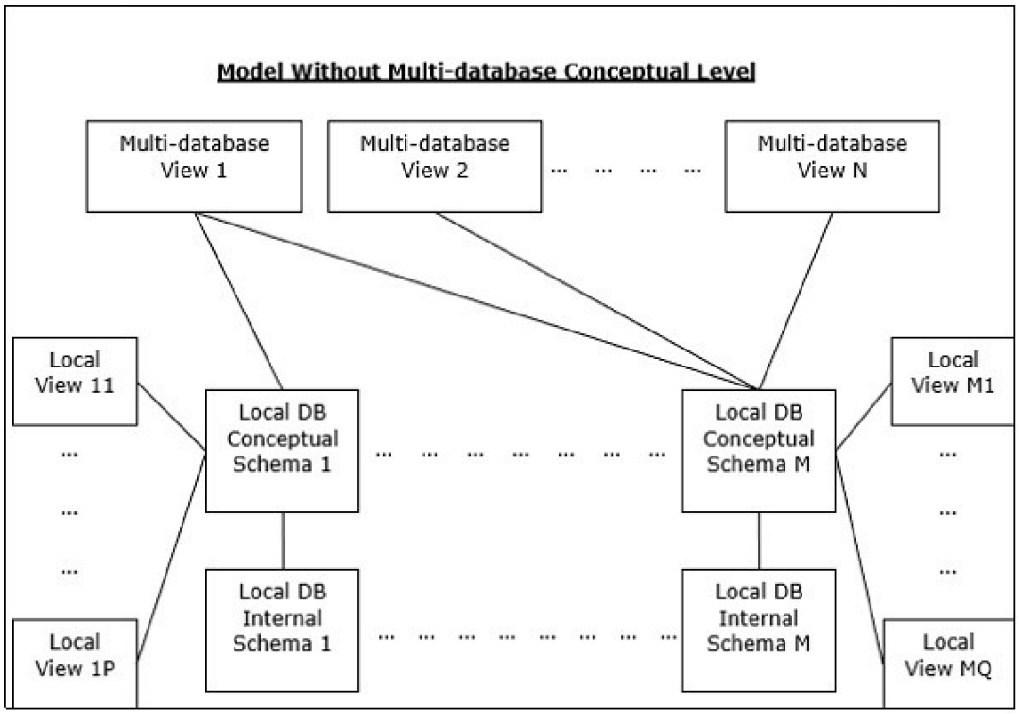
Local database Conceptual Level − Depicts local data organization at each site.

Local database Internal Level − Depicts physical data organization at each site.

There are two design alternatives for multi-DBMS −

- Model with multi-database conceptual level.
- Model without multi-database conceptual level.





DISTRIBUTED DATA STORAGE

Consider a relation r that is to bestored in the database. There are two approaches to storing this relation in the distributed database:

Replication. The system maintains several identical replicas of the relation, and stores each replica at a different site. The alternative to replication is to store only one copy of relation r.

Fragmentation. The system partitions the relation into several fragments, and stores each fragment at a different site.

Data Replication

If relation r is replicated, a copy of relation r is stored in two or more sites. In the most extreme case, we have **full replication**, in which a copy is stored in every site in the system.

There are a number of advantages and disadvantages to replication.

Availability If one of the sites containing relation r fails, then the relation r can be found in another site. Thus, the system can continue to process queries involving r, despite the failure of one site.

Increased parallelism. In the case where the majority of accesses to the relation r result in only the reading of the relation, then several sites can process queries involving r in parallel. The more replicas of r there are, the greater the chance that the needed data will be found in the site where the transaction is executing. Hence, data replication minimizes movement of data between sites. **Increased overhead on update.** The system must ensure that all replicas of a relation r are consistent; otherwise, erroneous computations may result. Thus, whenever is updated, the update must be propagated to all sites containing replicas. The result is increased overhead. For example, in a banking system, where account information is replicated in various sites, it is necessary to ensure that the balance in a particular account agrees in all sites.

Data Fragmentation

If relation r is fragmented, r is divided into a number of fragments r1, r2,...,rn. These fragments contain sufficient information to allow reconstruction of the original relation r.

There are two different schemes for fragmenting a relation: horizontal fragmentation and vertical fragmentation.

- Horizontal fragmentation splits the relation by assigning each tuple of r to one or more fragments.
- Vertical fragmentation splits the relation by decomposing the scheme R of relation r.

In **horizontal fragmentation**, a relation r is partitioned into a number of subsets, r1, r2,...,rn. Each tuple of relation r must belong to at least one of the fragments, so that the original relation can be reconstructed, if needed.

account1 = branch name = “Hillside” (account)
account2 = branch name = “Valleyview” (account)

Horizontal fragmentation is usually used to keep tuples at the sites where they are used the most, to minimize data transfer.

In general, a horizontal fragment can be defined as a selection on the global relation r. That is, we use a predicate Pi to construct fragment ri:

$$r_i = \sigma_{P_i}(r)$$

We reconstruct the relation r by taking the union of all fragments; that is:

$r = r1 \cup r2 \cup \cdots \cup rn$

Transparency

The user of a distributed database system should not be required to know where the data are physically located nor how the data can be accessed at the specific local site. This characteristic, called **data transparency**, can take several forms:

Fragmentation transparency. Users are not required to know how a relation has been fragmented.

Replication transparency. Users view each data object as logically unique. The distributed system may replicate an object to increase either system performance or data availability. Users do not have to be concerned with what data objects have been replicated, or where replicas have been placed.

Location transparency. Users are not required to know the physical location of the data. The distributed database system should be able to find any data as long as the data identifier is supplied by the user transaction.

DISTRIBUTED TRANSACTIONS

There are two types of transaction that we need to consider.

- **Local transactions** are those that access and update data in only one local database;
- **Global transactions** are those that access and update data in several local databases.

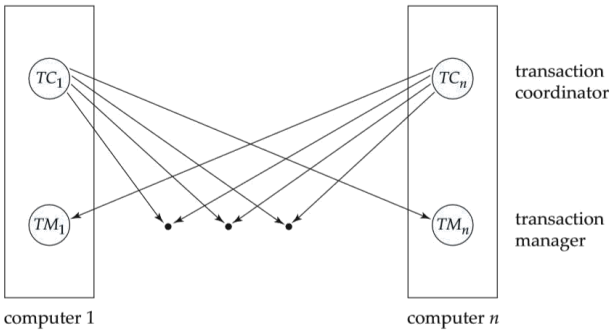
System Structure

Each site has its own local transaction manager, whose function is to ensure the ACID properties of those transactions that execute at that site. The various transaction managers cooperate to execute global transactions. To

understand how such a manager can be implemented, consider an abstract model of a transaction system, in which each site contains two subsystems:

- The **transaction manager** manages the execution of those transactions (or sub transactions) that access data stored in a local site.
- The **transaction coordinator** coordinates the execution of the various transactions (both local and global) initiated at that site.

- Maintaining a log for recovery purposes.
- Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site.



The transaction coordinator subsystem is not needed in the centralized environment, since a transaction accesses data at only a single site. A transaction coordinator, as its name implies, is responsible for coordinating the execution of all the transactions initiated at that site. For each such transaction, the coordinator is responsible for:

• Starting the execution of the transaction.

• Breaking the transaction into a number of sub transactions and distributing these sub transactions to the appropriate sites for execution.

• Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

System Failure Modes

- Failure of a site.
- Loss of messages.
- Failure of a communication link.
- Network partition.

OBJECT-BASED DATABASES

An object-oriented database system is a database system that natively supports an object-oriented type system, and allows direct access to data from an object-oriented programming language using the native type system of the language.

Complex Data Types

Traditional database applications have conceptually simple datatypes. The basic data items are records that are fairly small and whose fields are atomic.

In recent years, demand has grown for ways to deal with more complex data types. Consider, for example, addresses. While an entire address could be viewed as an atomic data item of type string, this view would hide details such as the street address, city, state, and postal code, which could be of interest to queries. On the other hand, if an address were represented by breaking it into the components (street address, city, state, and postal code), writing queries would be more complicated since they would have to mention each field. A better alternative is to allow structured datatypes that allow a type address with subparts street address, city, state, and postal code.

Structured Types

Structured types allow composite attributes of E-R designs to be represented directly. For instance, we can define the following structured type to represent a composite attribute name with component attribute *firstname* and *lastname:*

```
create type Name as
(firstname varchar(20),
lastname varchar(20))
final;
```

Such types are called **user-defined** types in SQL. The **final** and **not final** specifications are related to subtyping.

The components of a composite attribute can be accessed using a **"dot"** notation; for instance, name. firstname returns the firstname component of the name attribute. An access to attribute name would return a value of the structured type Name.

We can also create a table whose rows are of a user-defined type. For example, we could define a type Person Type and create the table person as follows

```
create type PersonType as (
name Name,
address Address,
dateOfBirth date)
not final
create table person of PersonType;
```

Type Inheritance

Suppose that we have the following type definition for people:

```
create type Person
(name varchar(20),
address varchar(20));
```

We may want to store extra information in the database about people who are students, and about people who are teachers. Since students and teachers are also people, we can use inheritance to define the student and teacher types in SQL:

```
create type Student
under Person
(degree varchar(20),
 department varchar(20));
create type Teacher
under Person
(salary integer,
department varchar(20));
```

Both Student and Teacher inherit the attributes of Person—namely, name and address. Student and Teacher are said to be subtypes of Person, and Person is a supertype of Student, as well as of Teacher.

Table Inheritance

Sub tables in SQL correspond to the E-R notion of specialization/generalization. For instance, suppose we define the people table as follows:

```
create table people of Person;
```

We can then define tables students and teachers as sub tables of people, as follows:

```
create table students of Student
under people;
create table teachers of Teacher
under people;
```

The types of the sub tables (Student and Teacher in the above example) are subtypes of the type of the parent table (Person in the above example). As a result, every attribute present in the table people is also present in the sub tables students and teachers.

Array and Multiset Types in SQL

SQL supports two collection types: arrays and multisets

A multiset is an unordered collection, where an element may occur multiple times.

Multisets are like sets, except that a set allows each element to occur at most once.

Suppose we wish to record information about books, including a set of keywords for each book. Suppose also that we wished to store the names of authors of a book as an array; unlike elements in a multiset, the elements of an array are ordered, so we can distinguish the first author from the second author, and so on. The following example illustrates how these array and multiset-valued attributes can be defined in SQL:

```
create type Publisher as
(name varchar(20),
branch varchar(20));
create type Book as
(title varchar(20),
Autho_arrray varchar(20) array [10],
Pub_date date, publisher Publisher, keyword_set varchar(20) multiset);
create table books of Book;
```

The first statement defines a type called Publisher with two components: a name and a branch. The second statement defines a structured type Book that contains a title, an author array, which is an array of up to 10 author names, a publication date, a publisher (of type Publisher), and a multiset of keywords. Finally, a table books containing tuples of type Book is created.

Object-Identity and Reference Types in SQL

Object-oriented languages provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specified type. For example, in SQL we can define a type *Department* with a field *name* and a field *head* that is a reference to the type *Person,* and a table *departments* of type *Department*, as follows:

```
create type Department (
 name varchar(20),
head ref(Person) scope people);
create table departments of Department;
```

Here, the reference is restricted to tuples of the table people. The restriction of the scope of a reference to tuples of a table is mandatory in SQL, and it makes references behave like foreign keys.

Object-relational Features

Object-relational database systems are basically extensions of existing relational database systems. Changes are clearly required at many levels of the database system. However, to minimize changes to the storage-system code (relation storage, indices, etc.), the complex datatypes supported by object-relational systems can be translated to the simpler type system of relational databases.

Sub tables can be stored in an efficient manner, without replication of all inherited fields, in one of two ways:

- Each table stores the primary key (which may be inherited from a parent table) and the attributes that are defined locally. Inherited attributes (other than the primary key) do not need to be stored, and can be derived by means of a join with the super table, based on the primary key.
- Each table stores all inherited and locally defined attributes. When a tuple is inserted, it is stored only in the table in which it is inserted, and its presence is inferred in each of the super tables. Access to all attributes of a tuple is faster, since a join is not required.

The ODMG· Object Model

The **ODMG object model** is the data model upon which the object definition language (ODL) and object query language (OQL) are based. It is meant to provide a standard data model for object databases, just as SQL describes a standard data model for relational databases. It also provides a standard terminology in a field where the same terms were sometimes used to describe different concepts.

Objects and Literals

Objects and literals are the basic building blocks of the object model. The main difference between the two is that an object has both an **object identifier** and a **state** (or current value), whereas a literal has a value (state) but **no object identifier**. In either case, the value can have a complex **structure. The object** state can change over time by modifying the object value. A literal is basically a constant value, possibly having a complex structure, but it does not change.

An object has five aspects: identifier, name, lifetime, structure, and creation.

1. The **object identifier** is a unique system-wide identifier (or Object_id). Every object must have an object identifier.
2. Some objects may optionally be given a unique **name** within a particular ODMS—this name can be used to locate the object, and the system should return the object given that name. Obviously, not all individual objects will have unique names. Typically, a few objects, mainly those that hold collections of objects of a particular object type—such as ***extents***—will have a name. These names are used as ***entry points*** to the database; that is, by locating these objects by their unique name, the user can then locate other objects that are referenced from these objects. Other important objects in the application may also have unique names, and it is possible to give *more than one name* to an object. All names within a particular ODMS must be unique.
3. The ***lifetime*** of an object specifies whether it is a *persistent object* (that is, a database object) or *transient object* (that is, an object in an executing pro-gram that disappears after the program terminates). Lifetimes are independent of types—that is, some objects of a particular type may be transient whereas others may be persistent.
4. The ***structure*** of an object specifies how the object is constructed by using the type constructors. The structure specifies whether an object is ***atomic*** or not. An ***atomic object*** refers to a single object that follows a user-defined type, such as *Employee* or *Department*. If an object is not atomic, then it will be composed of other objects. For example, a *collection object* is not an atomic object, since its state will be a collection of other objects. In the ODMG model, an atomic object is any *individual user-defined object*. All values of the basic built-in data types are considered to be *literals*.
5. ***Object creation*** refers to the manner in which an object can be created. This is typically accomplished via an operation new for a special ***Object_Factory*** interface.

In the object model, a ***literal*** is a value that *does not have* an object identifier. However, the value may have a simple or complex structure.

There are three types of literals: atomic, structured, and collection.

1. **Atomic literals** correspond to the values of basic data types and are predefined. The basic data types of the object model include long, short, and unsigned integer numbers (these are specified by the keywords *long, short, unsigned long, and unsigned short* in ODL), regular and double precision floating point numbers *(float, double),* Boolean values *(boolean*), single characters *(char),* character strings *(string),* and enumeration types *(enum),* among others.
2. ***Structured literals*** correspond roughly to values that are constructed using the tuple constructor. The built-in structured lit-erals include *Date, Interval,Time, and Timestamp.*
3. ***Collection literals*** specify a literal value that is a collection of objects or values but the collection itself does not have an Object_id. The collections in the object model can be defined by the type generators set<T>, bag<T>, list<T>, and array<T>, where T is the type of objects or values in the collection.28 Another collection type is dictionary<K, V>, which is a collection of associations <K, V>, where each K is a key (a unique search value) associated with a value V; this can be used to create an index on a collection of values V.

The notation of ODMG uses three concepts: ***interface, literal, and class.*** Following the ODMG terminology, we use the word **behavior** to refer to *operations* and **state** to refer to *properties* (attributes and relationships).

An **interface** specifies only behavior of an object type and is typically ***noninstantiable*** (that is, no objects are created corresponding to an interface). Although an interface may have state properties (attributes and relationships) as part of its specifications, these cannot be inherited from the interface. Hence, an interface serves to define operations that can be inherited by other interfaces, as well as by classes that define the user-defined objects for a particular application.

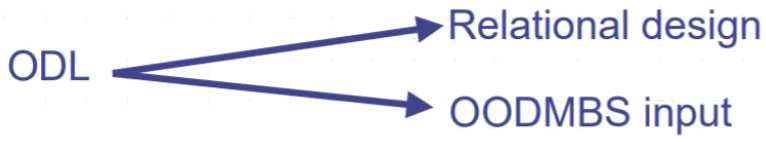
A **class** specifies both state (attributes) and behavior (operations) of an object type, and is instantiable. Hence, database and application objects are typically created based on the user-specified class declarations that form a database schema.

Finally, a **literal** declaration specifies state but no behavior. Thus, a literal instance holds a simple or

complex structured value but has neither an object identifier nor encapsulated operations.

ODL: OBJECT DEFINITION LANGUAGE

Object Definition Language (ODL) is the specification language defining the interface to object types conforming to the ODMG Object Model. Often abbreviated by the acronym ODL.This language's purpose is to define the structure of an Entity-relationship diagram.



Class Declarations

- interface < name > {elements = attributes, relationships, methods }

Element Declarations

- attribute < type > < name > ;
- **relationship < rangetype > < name > ;**

- float gpa(in: Student) raises(noGrades) float = return type.
- in: indicates Student argument is read-only.
  Other options: out, inout.

Relationships

- use inverse to specify inverse relationships
- at most one' semantics remain
- multiplicity
  - if many-many between C and D, then use Set<D> and Set<C>, respectively
  - if many-one from C to D, then use D in C and Set<C> in D
  - if many-one from D to C, then use C in D and Set<D> in C
  - if one-one between C and D, then use D and C, respectively

Datatypes

- basis
  - atomic: integer, float, character, character string, boolean, and enumeration
  - classes
- type constructors (can be composed to create complex types)
  - set: Set<T>
  - bag: Bag<T>
  - list: List<T> (sequential access)
  - array: Array<T,i> (random access)
  - dictionary: Dictionary<T,S>
  - structures
- difference between sets, bags, and lists
- rules for types and relationships
  - type of a relationship is either a class type or a (single use of a) collection type constructor applied to a class type' [FCDB]
  - type of an attribute is built starting with atomic type(s)' [FCDB]
- relationship types cannot involve
  - atomic types (e.g., Set<integer>),
  - structures (e.g., Struct N {Movie field1, Star field2}, or
  - two applications of collection types (e.g., Set<Array<Star, 10>>)

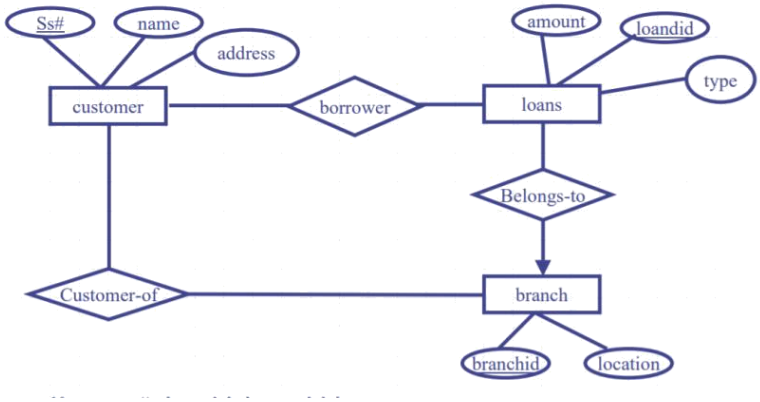
Similarities between E/R and ODL

- both support all multiplicities of relationships
- both support inheritance

Differences between E/R and ODL

- attributes are single-valued in E/R; can be multivalued in ODL
- keys required in E/R, optional in ODL
- can only model one key in E/R; can model all (or any subset) in ODL
- multiway relationships supported in E/R; only binary relationships supported in ODL
- no multiple inheritance in E/R; ODL supports multiple inheritance

Banking Example 1



- **Keys:** ss#, loanid, branchid
- **Cardinality constraint**: each loan belongs to a single branch.

OQL - OBJECT QUERY LANGUAGE

OQL is the way to access data in an O2 database. OQL is a powerful and easy-to-use SQL-like query language with special features dealing with complex objects, values and methods.

SELECT, FROM, WHERE SELECT
<list of values>
FROM <list of collections and variable assignments>
WHERE <condition>

The SELECT clause extracts those elements of a collection meeting a specific condition. By using the keyword DISTINCT duplicated elements in the resulting collection get eliminated. Collections in FROM can be either extents (persistent names - sets) or expressions that evaluate to a collection (a set). Strings are enclosed in double-quotes in OQL. We can rename a field by if we prefix the path with the desired name and a colon.

Example Query 1

Give the names of people who are older than 26 years
old: SELECT SName: p.name
FROM p in People
WHERE p.age >
26 (hit Ctrl-D)

Dot Notation & Path Expressions

We use the dot notation and path expressions to access components of complex values.

Let variables t and ta range over objects in extents (persistent names) of Tutors and TAs (i.e., range over objects in sets Tutors and TAs).

ta.salary -> real
t.students -> set of tuples of type tuple(name: string, fee: real) representing students
t.salary -> real

Cascade of dots can be used if all names represent objects and not a collection.

Example Query 2

Find the names of the students of all tutors:
SELECT s.name

```
FROM Tutors t, t.students s
```

Here we notice that the variable t that binds to the first collection of FROM is used to help us define the second collection s. Because students is a collection, we use it in the FROM list, like t.students above, if we want to access attributes of students.

Subqueries in FROM Clause

Example Query 3

Give the names of the Tutors which have a salary greater than $300 and have a student paying more than $30:

```
SELECT t.name
FROM ( SELECT t FROM Tutors t WHERE t.salary > 300 ) r, r.students s
WHERE s.fee > 30
```

Subqueries in WHERE Clause

Example Query 4

Give the names of people who aren't TAs:

```
SELECT p.name
FROM p in People
WHERE not ( p.name in SELECT t.name FROM t in TAs )
```

Set Operations and Aggregation

The standard O2C operators for sets are + (union), * (intersection), and - (difference). In OQL, the operators are written as UNION, INTERSECT and EXCEPT , respectively.

Example Query 5

Give the names of TAs with the highest salary:

```
SELECT t.name
FROM t in TAs
WHERE t.salary = max ( select ta.salary from ta in TAs )
```

GROUP BY

The GROUP BY operator creates a set of tuples with two fields. The first has the type of the specified GROUP BY attribute. The second field is the set of tuples that match that attribute. By default, the second field is called PARTITION.

Example Query 6

Give the names of the students and the average fee they pay their Tutors:

```
SELECT sname, avgFee: AVG(SELECT p.s.fee FROM partition p)
FROM t in Tutors, t.students s
GROUP BY sname: s.name
```

Embedded OQL

Instead of using query mode, you can incorporate these queries in your O2 programs using the "o2query" command:

```
run body {
 o2 real total_salaries;
o2query( total_salaries, "sum ( SELECT ta->get_salary \
FROM ta in TAs )" );
printf("TAs combined salary: %.2f\n", total_salaries);
};
```

The first argument for o2query is the variable in which you want to store the query results. The second argument is a string that contains the query to be performed. If your query string takes up several lines, be sure to backslash (\) the carriage returns.

XML DATABASES

XML Hierarchical (Tree) Data Model

The basic object in XML is the XML document. Two main structuring concepts are used to construct an XML document: elements and attributes.

An example of an XML element called . As in HTML, elements are identified in a document by their start tag and end tag. The tag names are enclosed between angled brackets < ... >, and end tags are further identified by a

slash.</…..>.

Complex elements are constructed from other elements hierarchically, whereas **simple elements** contain data values. A major difference between XML and HTML is that XML tag names are defined to describe the meaning of the data elements in the document, rather than to describe how the text is to be displayed. This makes it possible to process the data elements in the XML document automatically by computer programs. Also, the XML tag (element) names can be defined in another document, known as the schema document, to give a semantic meaning to the tag names that can be exchanged among multiple users. In HTML, all tag names are predefined and fixed; that is why they are not extendible.

It is possible to characterize three main types of XML documents:

- **Data-centric XML documents.** These documents have many small data items that follow a specific structure and hence may be extracted from a structured database. They are formatted as XML documents in order to exchange them over or display them on the Web. These usually follow a predefined schema that defines the tag names.
- **Document-centric XML documents**. These are documents with large amounts of text, such as news articles or books. There are few or no structured data elements in these documents.
- **Hybrid XML documents.** These documents may have parts that contain structured data and other parts that are predominantly textual or unstructured. They may or may not have a predefined schema.

```
<Projects>
    <Project>
        <Name>ProductX</Name>
        <Number>1</Number>
        <Location>Bellaire</Location>
        <Dept_no>5</Dept_no>
        <Worker>
            <Ssn>123456789</Ssn>
            <Last_name>Smith</Last_name>
            <Hours>32.5</Hours>
        </Worker>
        <Worker>
            <Ssn>453453453</Ssn>
            <First_name>Joyce</First_name>
            <Hours>20.0</Hours>
        </Worker>
    </Project>
    <Project>
        <Name>ProductY</Name>
        <Number>2</Number>
        <Location>Sugarland</Location>
        <Dept_no>5</Dept_no>
        <Worker>
            <Ssn>123456789</Ssn>
            <Hours>7.5</Hours>
        </Worker>
        <Worker>
            <Ssn>453453453</Ssn>
            <Hours>20.0</Hours>
        </Worker>
        <Worker>
            <Ssn>333445555</Ssn>
            <Hours>10.0</Hours>
        </Worker>
    </Project>
...
</Projects>
```

XML documents that do not follow a predefined schema of element names and corresponding tree structure are known as schemaless XML documents. It is important to note that data-centric XML documents can be considered either as semistructured data or as structured data

DOCUMENT TYPE DEFINITION (DTD)

The document type definition (DTD) is an optional part of an XML document. The main purpose of a DTD is much like that of a schema: to constrain and type the information present in the document.

```
<!DOCTYPE university [
    <!ELEMENT university ( (department|course|instructor|teaches)+)>
    <!ELEMENT department ( dept_name, building, budget)>
    <!ELEMENT course ( course_id, title, dept_name, credits)>
    <!ELEMENT instructor (IID, name, dept_name, salary)>
    <!ELEMENT teaches (IID, course_id)>
    <!ELEMENT dept_name( #PCDATA )>
    <!ELEMENT building( #PCDATA )>
    <!ELEMENT budget( #PCDATA )>
    <!ELEMENT course_id ( #PCDATA )>
    <!ELEMENT title ( #PCDATA )>
    <!ELEMENT credits( #PCDATA )>
    <!ELEMENT IID( #PCDATA )>
    <!ELEMENT name( #PCDATA )>
    <!ELEMENT salary( #PCDATA )>
] >
```

However, the DTD does not in fact constrain types in the sense of basic types like integer or string. Instead, it constrains only the appearance of sub elements and attributes within an element. The DTD is primarily a list of rules for what pattern of subelements may appear within an element.

Example of a DTD

Thus, in the DTD, a university element consists of one or more course, department, or instructor elements; the operator specifies "or" while the + operator specifies "one or more." Although not shown here, the ∗ operator is used to specify "zero or more," while the? operator is used to specify an optional element (that is, "zero or one"). The course element contains sub elements course id, title, dept name, and credits (in that order).

Similarly, department and instructor have the attributes of their relational schema defined as sub elements in the DTD. Finally, the elements course id, title, dept name, credits, building, budget, IID, name, and salary are all declared to be of type #PCDATA. The keyword #PCDATA indicates text data; it derives its name, historically, from "parsed character data." Two other special type declarations are empty, which says that the element has no contents, and any, which says that there is no constraint on the sub elements of the element; that is, any elements, even those not mentioned in the DTD, can occur as sub elements of the element. The absence of a declaration foran element is equivalent to explicitly declaring the type as any.

XML SCHEMA

XML Schema defines a number of built-in types such as string, integer, decimal date, and boolean. In addition, it allows user-defined types; these may be simple types with added restrictions, or complex types constructed using constructors such as complex Type and sequence.

Note that any namespace prefix could be used in place of xs; thus we could replace all occurrences of "xs:" in the schema definition with "xsd:" without changing the meaning of the schema definition. All types defined by XML Schema must be prefixed by this namespace prefix. The first element is the root element university, whose type is specified to be University Type, which is declared later. The example then defines the types of elements department, course, instructor, and teaches. Note that each of these is specified by an element with tag xs:element, whose body contains the type definition.

The type of department is defined to be a complex type, which is further specified to consist of a sequence of elements dept name, building, and budget. Any type that has either attributes or nested sub elements must be specified to be a complex type. Alternatively, the type of an element can be specified to be a predefined type by the attribute type; observe how the XML Schema types xs: string and xs: decimal are used to constrain the types of data elements such as dept name and credits. Finally, the example defines the type University Type as containing zero or more occurrences of each of department, course, instructor, and teaches. Note the use of ref to specify the occurrence

of an element defined earlier.

XML Schema can define the minimum and maximum number of occurrences of sub elements by using minOccurs and max Occurs. The default for both minimum and maximum occurrences is 1, so these have to be specified explicitly to allow zero or more department, course, instructor, and teaches elements

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="building" type="xs:string"/>
            <xs:element name="budget" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="course">
            <xs:element name="course_id" type="xs:string"/>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="credits" type="xs:decimal"/>
</xs:element>
<xs:element name="instructor">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="salary" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

.

```
<xs:element name="teaches">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="course_id" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="UniversityType">
    <xs:sequence>
        <xs:element ref="department" minOccurs="0"
                maxOccurs="unbounded"/>
        <xs:element ref="course" minOccurs="0"
                maxOccurs="unbounded"/>
        <xs:element ref="instructor" minOccurs="0"
                maxOccurs="unbounded"/>
        <xs:element ref="teaches" minOccurs="0"
                maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

Attributes are specified using the xs:attribute tag. For example, we could have defined dept name as an attribute by adding:
<xs:attribute name = "dept name"/>
within the declaration of the department element. Adding the attribute use = "required" to the above attribute specification declares that the attribute must be specified, whereas the default value of use is optional. Attribute specifications would appear directly under the enclosing complex Type specification, even if elements are nested within a sequence specification.

In addition to defining types, a relational schema also allows the specification of constraints. XML Schema allows the specification of keys and key references, corresponding to the primary-key and foreign-key definition in SQL. In SQL, a primary-key constraint or unique constraint ensures that the attribute values do not recur within the relation. In the context of XML, we need to specify a scope within which values are unique and form a key. The selector is a path expression that defines the scope for the constraint, and field declarations specify the elements or attributes that form the key. To specify that dept name forms a key for department elements under the root university element, we add the following constraint specification to the schema definition:

```
<xs:key name = "deptKey">
    <xs:selector xpath = "/university/department"/>
    <xs:field xpath = "dept_name"/>
</xs:key>
```

Correspondingly a foreign-key constraint from course to department may be defined as follows:

```
<xs: name = "courseDeptFKey" refer="deptKey">
    <xs:selector xpath = "/university/course"/>
    <xs:field xpath = "dept_name"/>
</xs:keyref>
```

XML Schema offers several benefits over DTDs, and is widely used today. Among the benefits that we have seen in the examples above are these:

- It allows the text that appears in elements to be constrained to specific types, such as numeric types in specific formats or complex types such as sequences of elements of other types.
- It allows user-defined types to be created.
- It allows uniqueness and foreign-key constraints.
- It is integrated with namespaces to allow different parts of a document to conform to different schemas.

In addition to the features we have seen, XML Schema supports several other features that DTDs do not, such as these:

- It allows types to be restricted to create specialized types, for instance by specifying minimum and maximum values.
- It allows complex types to be extended by using a form of inheritance.

XQUERY

XPath allows us to write expressions that select items from a tree-structured XML document. XQuery permits the specification of more general queries on one or more XML documents. The typical form of a query in XQuery is known as a FLWR expression, which stands for the four main clauses of XQuery and has the following form:

FOR<variable bindings to individual nodes (elements)>
LET <variable bindings to collections of nodes (elements)>
WHERE <qualifier conditions>
RETURN<query result specification>

There can be zero or more instances of the FOR clause, as well as of the LET clause in a single XQuery. The WHERE clause is optional, but can appear at most once, and the RETURN clause must appear exactly once. Let us

illustrate these clauses with the following simple example of a XQuery.

```
LET $d := doc(www.company.com/info.xml)
FOR $x IN $d/company/project[projectNumber = 5]/projectWorker,
    $y IN $d/company/employee
WHERE $x/hours gt 20.0 AND $y.ssn = $x.ssn
RETURN <res> $y/employeeName/firstName, $y/employeeName/lastName,
                $x/hours </res>
```

1. Variables are prefixed with the $ sign. In the above example, $d, $x, and $y are variables.
2. The LET clause assigns a variable to a particular expression for the rest of the query. In this example, $d is assigned to the document file name. It is possible to have a query that refers to multiple documents by assigning multiple variables in this way.
3. The FOR clause assigns a variable to range over each of the individual items in a sequence. In our example, the sequences are specified by path expressions. The $x variable ranges over elements that satisfy the path expression $d/company/project[projectNumber = 5]/projectWorker. The $y variable ranges over elements that satisfy the path expression $d/company/employee. Hence, $x ranges over projectWorker elements, whereas $y ranges over employee elements.
4. The WHERE clause specifies additional conditions on the selection of items. In this example, the first condition selects only those projectWorker elements that satisfy the condition (hours gt 20.0). The second condition specifies a join condition that combines an employee with a projectWorker only if they have the same ssn value.
5. Finally, the RETURN clause specifies which elements or attributes should be retrieved from the items that satisfy the query conditions. In this example, it will return a sequence of elements each containing for employees who work more that 20 hours per week on project number 5.

XQuery has very powerful constructs to specify complex queries. In particular, it can specify universal and existential quantifiers in the conditions of a query, aggregate functions, ordering of query results, selection based on position in a sequence, and even conditional branching. Hence, in some ways, it qualifies as a full-fledged programming language.

INFORMATION RETRIEVAL

IR CONCEPTS

Information retrieval is the process of retrieving documents from a collection in response to a query (or a search request) by a user. Information retrieval is "the discipline that deals with the structure, analysis, organization, storage, searching, and retrieval of information" as defined by Gerald Salton, an IR pioneer.

Information in the context of IR does not require machine-understandable structures, such as in relational database systems. Examples of such information include written texts, abstracts, documents, books, Web pages, e-mails, instant messages, and collections from digital libraries. Therefore, all loosely represented (unstructured) or semi structured information is also part of the IR discipline.

IR systems go beyond database systems in that they do not limit the user to a specific query language, nor do they expect the user to know the structure (schema) or content of a particular database. IR systems use a user's information need expressed as a **free-form search request** (sometimes called a **keyword search query**, or just query) for interpretation by the system.

An IR system can be characterized at different levels: by *types of users*, *types of data*, *and the types of the information need*, along with the size and scale of the information repository it addresses. Different IR systems are designed to address specific problems that require a combination of different characteristics. These characteristics can be briefly described as follows:

Types of Users

The user may be an *expert user* (for example, a curator or a librarian), who is searching for specific information that is clear in his/her mind and forms relevant queries for the task, or a *layperson user* with a generic information need.

Types of Data

Search systems can be tailored to specific types of data. For example, the problem of retrieving information

about a specific topic may be handled more efficiently by customized search systems that are built to collect and retrieve only information related to that specific topic. The information repository could be hierarchically organized based on a concept or topic hierarchy. These topical domain-specific or vertical IR systems are not as large as or as diverse as the generic World Wide Web, which contains information on all kinds of topics.

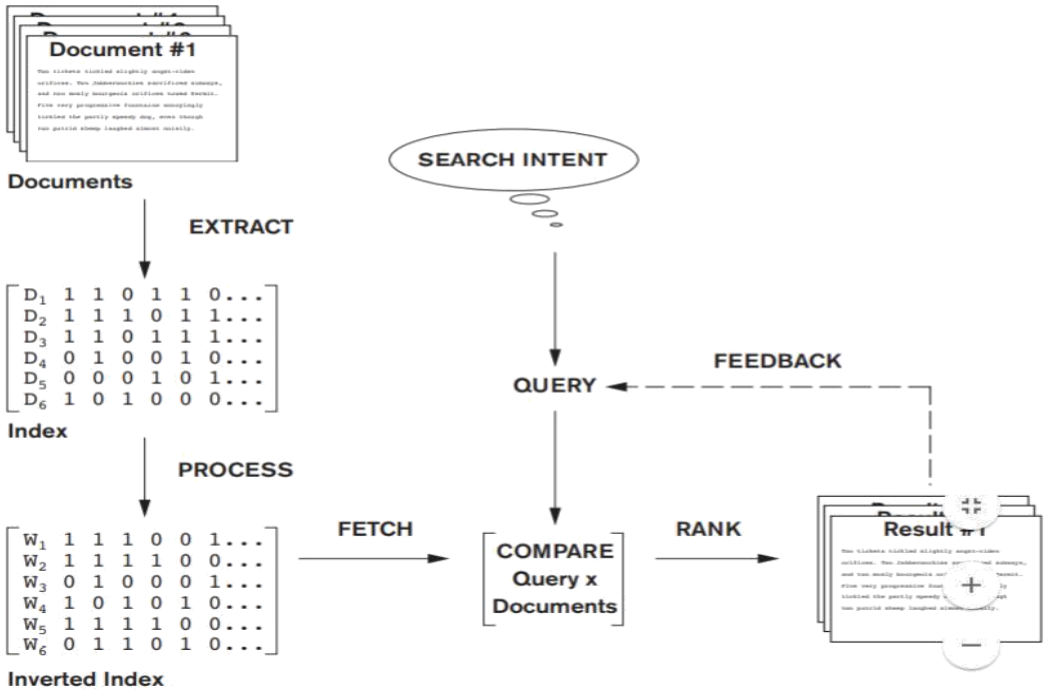
Types of Information Need

In the context of Web search, users' information needs may be defined as navigational, informational, or transactional.

Navigational search refers to finding a particular piece of information (such as the Georgia Tech University Website) that a user needs quickly. The purpose of **informational search** is to find current information about a topic (such as research activities in the college of computing at Georgia Tech—this is the classic IR system task). The goal of **transactional search** is to reach a site where further interaction happens (such as joining a social network, product shopping, online reservations, accessing databases, and so on).

RETRIEVAL MODELS

There are the three main statistical models—Boolean, vector space, and probabilistic—and the semantic model.



Simplified IR process Pipeline

Boolean Model

In this model, documents are represented as a set of terms. Queries are formulated as a combination of terms using the standard Boolean logic set-theoretic operators such as AND, OR and NOT. Retrieval and relevance are considered as binary concepts in this model, so the retrieved elements are an "exact match" retrieval of relevant documents.

Boolean retrieval models lack sophisticated ranking algorithms and are among the earliest and simplest information retrieval models. These models make it easy to associate metadata information and write queries that match the contents of the documents as well as other properties of documents, such as date of creation, author, and type of document.

Vector Space Model

The vector space model provides a framework in which term weighting, ranking of retrieved documents, and relevance feedback are possible. Documents are represented as features and weights of term features in an n-dimensional vector space of terms. Features are a subset of the terms in a set of documents that are deemed most

relevant to an IR search for this particular set of documents. The process of selecting these important terms (features) and their properties as a sparse (limited) list out of the very large number of available terms (the vocabulary can contain hundreds of thousands of terms) is independent of the model specification. The query is also specified as a terms vector (vector of features), and this is compared to the document vectors for similarity/relevance assessment.

In the vector model, the document term weight wij (for term i in document j) is represented based on some variation of the TF (term frequency) or TF-IDF (term frequency-inverse document frequency) scheme (as we will describe below). TF-IDF is a statistical weight measure that is used to evaluate the importance of a document word in a collection of documents. The following formula is typically used:

$$\text{cosine}(d_j, q) = \frac{\langle d_j \times q \rangle}{\|d_j\| \times \|q\|} = \frac{\sum_{i=1}^{|V|} w_{ij} \times w_{iq}}{\sqrt{\sum_{i=1}^{|V|} w_{ij}^2} \times \sqrt{\sum_{i=1}^{|V|} w_{iq}^2}}$$

In the formula given above, we use the following symbols:

- dj is the document vector.
- q is the query vector.
- wij is the weight of term i in document j.
- wiq is the weight of term i in query vector q.
- |V| is the number of dimensions in the vector that is the total number of important keywords (or features).

Probabilistic Model

In the probabilistic framework, the IR system has to decide whether the documents belong to the **relevant set** or the **nonrelevant** set for a query. To make this decision, it is assumed that a predefined relevant set and nonrelevant set exist for the query, and the task is to calculate the probability that the document belongs to the relevant set and compare that with the probability that the document belongs to the nonrelevant set.

Given the document representation D of a document, estimating the relevance R and nonrelevance NR of that document involves computation of conditional probability P(R|D) and P(NR|D). These conditional probabilities can be calculated using Bayes' Rule

$$P(R|D) = P(D|R) \times P(R)/P(D)$$
$$P(NR|D) = P(D|NR) \times P(NR)/P(D)$$

A document D is classified as relevant if P(R|D) > P(NR|D). Discarding the constant P(D), this is equivalent to saying that a document is relevant if:

$$P(D|R) \times P(R) > P(D|NR) \times P(NR)$$

The likelihood ratio P(D|R)/P(D|NR) is used as a score to determine the likelihood of thedocument with representation D belonging to the relevant set.

Semantic Model

Semantic approaches include different levels of analysis, such as morphological, syntactic, and semantic analysis, to retrieve documents more effectively. In morphological analysis, roots and affixes are analyzed to determine the parts of speech (nouns, verbs, adjectives, and so on) of the words.

The development of a sophisticated semantic system requires complex knowledge bases of semantic information as well as retrieval heuristics. These systems often require techniques from artificial intelligence and expert systems. Knowledge bases like Cyc15 and WordNet16 have been developed for use in knowledge-based IR systems based on semantic models.

QUERIES IN IR SYSTEMS

The queries formulated by users are compared to the set of index keywords. Most IR systems also allow the use of Boolean and other operators to build a complex query. The query language with these operators enriches the expressiveness of a user's information need.

Keyword Queries

Keyword-based queries are the simplest and most commonly used forms of IR queries: the user just enters keyword combinations to retrieve documents. The query keyword terms are implicitly connected by a logical AND operator. A query such as 'database concepts' retrieves documents that contain both the words 'database' and 'concepts' at the top of the retrieved results. In addition, most systems also retrieve documents that contain only 'database' or only 'concepts' in their text. Some systems remove most commonly occurring words (such as a, the, of, and so on, called stop words) as a preprocessing step before sending the filtered query keywords to the IR engine.

Boolean Queries

Some IR systems allow using the AND, OR, NOT, ( ), + , and – Boolean operators in combinations of keyword formulations. AND requires that both terms be found. OR lets either term be found. NOT means any record containing the second term will be excluded. '( )' means the Boolean operators can be nested using parentheses. '+' is equivalent to AND, requiring the term; the '+' should be placed directly in front of the search term.'–' is equivalent to AND NOT and means to exclude the term; the '–' should be placed directly in front of the search term not wanted. Complex Boolean queries can be built out of these operators and their combinations, and they are evaluated according to the classical rules of Boolean algebra.

Phrase Queries

When documents are represented using an inverted keyword index for searching, the relative order of the terms in the document is lost. In order to perform exact phrase retrieval, these phrases should be encoded in the inverted index or implemented differently (with relative positions of word occurrences in documents). A phrase query consists of a sequence of words that makes up a phrase. The phrase is generally enclosed within double quotes. Each retrieved document must contain at least one instance of the exact phrase. Phrase searching is a more restricted and specific version of proximity searching.

Proximity Queries

Proximity search refers to a search that accounts for how close within a record multiple terms should be to each other. The most commonly used proximity search option is a phrase search that requires terms to be in the exact order.

Wildcard Queries

Wildcard searching is generally meant to support regular expressions and pattern matching-based searching in text. In IR systems, certain kinds of wildcard search support may be implemented—usually words with any trailing characters (for example, 'data*' would retrieve data, database, datapoint, dataset, and so on).

Natural Language Queries

There are a few natural language search engines that aim to understand the structure and meaning of queries written in natural language text, generally as a question or narrative. This is an active area of research that employs techniques like shallow semantic parsing of text, or query reformulations based on natural language understanding. The system tries to formulate answers for such queries from retrieved results. Some search systems are starting to provide natural language interfaces to provide answers to specific types of questions, such as definition and factoid questions, which ask for definitions of technical terms or common facts that can be retrieved from specialized databases.