CHAPTER 2
Object Oriented Database: A Survey

2.1 Preamble.

Recent developments in Database Management System and related work with present study are reviewed in this chapter. Starting with a brief discussion on some of the important DBMS project, efforts had been made to focus on the application modeling, execution modeling and concurrency control issues in DBMS.

Several prototype DBMS were being or had already been developed. The underlying data models used in these DBMS were mostly relational or object oriented. To capture the execution semantics, most of these systems had adopted rule based approach. Through the following paragraphs the relational DBMS projects are discussed at the beginning and the projects on Object Database systems are reviewed afterwards.

2.2 Relational Model.

Relational model being record oriented and simple, offers good storage management, improved reliability and security. However, a normalized relation often fails to capture the semantics of the conceptual world, especially in cases involving management of complex information and inter relationships. Several researchers elaborately discussed of semantic capabilities and limitations of relational model through their studies\(^8, 27, 31, 61, 63, 74, 78.\)

Codd\(^{31}\) proposed a structure for a relational model indenting to form the basis for a high level data language that would have yield maximum independence between the organization and representation of data, and the programs that use and manipulate this data. Codd had provided a basis for checking consistency, and for eliminating redundancy within the data. He had also proposed an abstract characterization of the class of queries which were computable.
He proposed the relational model with several components e.g., a set of *relations* (commonly referred to as *tables*), a set of operations that act upon and manipulate relations, a set of rules for maintaining *integrity* within the database, a *relation* (which is the relational model equivalent of an *entity* in Entity-Relationship diagrams) is a table with rows and columns, which had two components, a *relation schema*, and a *relation instance*. The *schema* defined the structure of the relation; it described the relation's name, the names and *domains* of each column (*attribute*, or *field*) in the table. He described that an instance of a relation would must be with properties e.g. atomic attributes would not represent a group of values or an array and column values would be of the same kind, unique row, unique name for each column, every row in the table would must contained a primary key (it could not be NULL), foreign keys for cross reference values across the relation to provide a way of checking and enforcing integrity with the database.

However, in spite of its elegance, the relational model is not free from limitations. Some of which are summarized as follows.

- **Single domain restriction**: Relational model requires that one field of a record must always assume value from only one domain, which is not always desirable. For example, the month field of a relation that involves date of purchase of books. This field can logically assume any of the two values ‘7’ or ‘July’, meaning essentially the same thing. However in the relational model one cannot update a date stored in one of these formats (integer) with a date presented according to the other format (string).

- **Homogeneity of tuples**: The relational model requires that all the tuples of a relation be homogeneous, horizontally as well as vertically. Horizontal homogeneity implies that the field structure and formats of the fields of all the tuples must be identical. One cannot have two different tuples containing two different types of values in the same relation.
• **Fixed set of operations:** Only a fixed set of operations are allowed, e.g. simple arithmetic operations, relational operations on data values and relational algebraic operations like projection, selection, join etc, general purpose computation involving data stored in the database requires a host language interface. Such an interface, if not adequately designed, constantly presents a non-uniform view to the user, who must then have to have two different models present in his/her mind all the time. This complicates the task of semantic analysis and debugging of the program.

• **Extensive use of join** operations becomes necessary for representing complex relationships among the entities.

• **Dynamic objects** such as sets have to be implemented by several records and join operations. Unless properly implemented, representation of dynamic objects may turn out to be inadequate and inefficient.

• **Low-level concurrency control:** Many tuples may have to be locked for locking an object. For example, a supplier relation SA (S,A) containing information regarding the name and addresses of suppliers. In a multi-user environment, a transaction that access information about a particular supplier would require a lock on all the tuples with the supplier-name as given, or, in the worst case, a lock on the entire relation. The tuple level locking increases the size of the lock table as well as complexity of the concurrency control. The alternative strategy which locks the entire relation instead, inevitably reduces concurrency.

• **Redundancy:** In a relational model the only way to maintain referential integrity is through common values, which can be used to participate in a join. This generally leads to data redundancy due to duplication of the joinable fields in many relations.

• **Expressive power of Query Language:** The Expressive power of query languages based on relational algebra and calculus is quite limited. Although the relational query languages are elegant because they allow easy composition of queries.
Besides having a well-defined declarative semantics, they cannot express many useful queries, e.g. those involving a transitive closure of relations. Expressive power and tractability of relational query languages had been studied by Chandra and Harel\textsuperscript{27}.

In a relational database, conceptual world semantics are captured with the help of data dependencies (functional, multi-valued, hierarchical dependencies) and various normal forms\textsuperscript{3}. However, many significant problems related to normalization (e.g. key determination, BCNF decomposition) turn out to be NP (Normal Point) hard. Views and integrity constraints enhance the modeling capability of a relational schema. Update over relational views, however, as problems in the relational world.

In order to overcome major difficulties associated with the relational model, several approaches had been suggested to extend the model while maintaining its basic spirit of mathematical elegance and simplicity. For instance, in Alpha\textsuperscript{8} relational algebra has been extended with recursive queries by Agarwal.

2.3 Nested Relations Model.

Similarly, nested relational models\textsuperscript{2,44} had been proposed to extend the relational theory beyond 1NF relations. In nested relations a field of a relation may be a relation itself. It serves some of the requirements for complex fields and their meaningful associations. Portioned normal forms\textsuperscript{2,3} had been shown to represent a class of complex objects quite efficiently.

The other line of extension of the relational model had been the enhancement of the expressive power of the query languages without changing the basic model of normalized relations. POSTGRES\textsuperscript{79} used a data type QUEL to store a QUEL query as part of a relation. Such a stored program approach enhanced the expressive power of the language quite appreciably. In this approach, accessing a part of a relation requires
evaluation of the stored QUEL program and using the result of the evaluation as the desired result.

Inspired by logic programming, deductive models had been introduced on top of basic relational structure to enhance the overall modeling capability and expressive power of the system. In the deductive database model, the relations are treated as models of basic predicates, called the Extensional Predicates. New predicates can be defined using recursive Horn Clause rules (called the Intentional predicates). Database schema is defined using these predicates and rules. The semantics of the database schema is defined as a model of these rules. This provides an excellent declarative semantics of a database. The evolution of data models like Datalog are based on this approach. Query processing and optimization strategies had been proposed for these data models. The deductive approach has drawn the two 'bases', the knowledge base and the database, closer so that efficiency of access mechanisms in databases can be integrated with the expressive power of knowledge bases. However, such an approach faces some of the problems inherent in logic programming, namely artificial introduction of control into the language (cut, ordering of rules and sub goals, like which is not desirable in a declarative language) and lack of efficiency of inference, database update often creates problems. In fact, an appropriate model of update of a logic database was not available. The main problem arises during updation of the intentional databases and propagation of updates to the extensional database accurately and consistently. Then some logic based approaches to database updation had been proposed in the literature.

2.4 E-R Model.

Cehn proposed the entity-relationship (E-R) model in 1976 with a view to model the conceptual world more accurately than the record based models. He proposed a modeling schema based on two fundamental concepts- entities of the world and the relationships among the entities. These relationships could be constrained on the basis of cardinality restrictions. Thus one may have either 1:1, 1:N or M:N relationships. The
E-R model supports derived data types, modeled as weak entities. An entity consists of several simple attributes. Later a variety of extensions of the E-R model were proposed in order to capture more complex data semantics.

Introduction of E-R model is generally marked as the beginning of the journey to a new class of data models, called the Semantic Data Models. Semantic data models are aimed at capturing semantics of an enterprise with the help of complex structural constructs. A number of semantic models had been proposed in the last decade, all of which attempt to capture the real world semantics with the aid of increasingly complex structural constructs and abstraction.

2.5 Semantic Model.

Semantic data types in a semantic data model are not restricted to a fixed set of primitive types e.g. integer, float, char, and string etc. Higher order types can be constructed in a manner close to the natural semantics of the data to be represented. Thus one can model the address of a person as an entity of type ADDRESS, which in turn may consist of different components like city, street etc. This is in contrast with the 1NF paradigm, where one cannot have an address of a person as an entity of a distinct type rather, it becomes necessary to introduce different attributes like street, city etc. This type of construct that composes types by aggregating tributes of other types is called aggregation. This construction facility of abstract types reduces the semantic overloading on types.

The other important aspect of semantic data models consists of the abstraction mechanism and encapsulation. The objects can be viewed at different levels of abstraction so as to reveal the object details in a controlled and systematic way. The common construct used for abstraction is ISA relationship among object types. In an ISA hierarchy, subtypes inherit properties from the super types. Due to this inheritance of properties, the ISA hierarchy is also referred to as the inheritance hierarchy. An ISA
hierarchy is called a multiple inheritance hierarchy or multiple hierarchy where a
type may have multiple supertype. Otherwise it is called a simple hierarchy.

The notion of ISA relationships and property inheritance for enriching a model for
representation of semantic relationships is borrowed from earlier research on
knowledge representation schemes had on semantic networks and frame. Derived
schema components provide another dimension of abstraction supported by a
few semantic models. These Derived types allow users to define new schema
components which are dependent on other portions of the schema.

The functional data model is another semantic data model which
does not construct objects with structural constructs unlike other semantic data
models. The functional data model was proposed with the aim of providing a
model and definition/manipulation of language capable of representing applications with
naturalness and simplicity. In functional data model functions are used to define the
aggregation of attributes to form an entity. Thus employee may be defined as follows.

DECLARE EMPLOYEE () = > Entity
DECLARE name (EMPLOYEE) = > string
DECLARE address (EMPLOYEE) = > ADDRESS
DECLARE salary (EMPLOYEE) = > real
DECLARE works-under (EMPLOYEE) = > EMPLOYEE

Attributes of entities as well as the relationships are modeled in a similar manner and
may be viewed as functions, which may be single valued or multi valued. This provides
a clear, concise and uniform representation of relationships among entities.

The basic concept of semantic data models had been formalized into the theoretical
model IFO by Abiteboul and Hull. The IFO model captures the semantic constructs of
aggregation and specialization (ISA), where objects are represented structurally using
the concept of grouping (SET of objects). Objects are further represented structurally using the concept of a structure-rep (representations).

A structure-rep is identified as a directed tree. Nodes of a structure-rep represent objects, which may be either primitive (printable) or non-primitive (abstract). Edges joining nodes represent the structural relationships, e.g. specialization, generalization, aggregation and grouping among the objects represented by the nodes. Functional relationships among objects are represented by the concept of fragment-rep (representations). Structure representations are normalized and their properties like dominance and equivalence of representations studied. Abiteboul and Hull have used this formal model to define an updated semantics of objects in a semantic database. Based on the IFO model, a calculus based query language and a functional query language have also been proposed. A comprehensive survey of the work on theoretical research on constructed types may be found in Hull.

The query languages for complex objects and nested relations, and the expressive power of such languages had been analyzed in 2, 4, 12, 43, 50. Both calculus based and algebraic languages for non-INF relations had been developed and shown to had equivalent expressive power. 12, 42, 43, 74 The format model 28 provides an alternative approach for dealing with constructed types. Formats were built hierarchically from three constructs: aggregation, grouping, and marked or disjoint union of two or more types. This work focuses on comparing the data capacity of formats by constructing normalized format structures and studying containment and equivalence properties of format structures.

The interactions of ISA relationships with static integrity constraints like covering constraints, disjointness constraint, and inheritance constraints had been studied in 10. Atzeni and Parker had examined the interaction of ISA relationships and disjointness constraints and had formulated a sound and complete set of inference rules for these constraints. It had also been shown that some of the decision problems related to such constraints were solvable in polynomial time. For example, the satisfiability problem,
concerned with finding whether a node in a schema is non-empty in at least one instance of the schema, can be solved in polynomial time. Lenzerini\textsuperscript{65} had studied the covering constraints in addition to disjointness constraints. In this case the satisfiability problem had been shown to be NP-complete.

To summarize, it may be stated that during the evolution process, the data models have so far been enriched to capture more semantics by improving the type structure (as in the semantic data models) and by enriching the language associated with it. The improvement of query languages has followed three distinct paths. In one approach, declarative languages like logic programming had been adopted as the carrier. Functional data models adopted functional languages in coherence with the style of functional programming\textsuperscript{11, 24} whereas Semantic models like T\-AXIS used procedural languages like PASCAL. Thus one could find a clear interrelationship between the development of programming languages and that of database models, especially keeping in mind recent developments of relational databases. An evolutionary nature of data models and languages is presented in table 2.1

<table>
<thead>
<tr>
<th>Language</th>
<th>Underlying concept</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBOL like</td>
<td>Relational Algebra</td>
<td>file</td>
</tr>
<tr>
<td>C</td>
<td>Pointer</td>
<td>RDBMS</td>
</tr>
<tr>
<td>C++</td>
<td>Pointer</td>
<td>OODBMS</td>
</tr>
</tbody>
</table>

Table 2.1: Language and Model.

2.6 Object Oriented Data Model.

Improvements of data models, from flat record-oriented models to semantic models, have enabled many complex structural aspects of entities in the world to be adequately represented. Kim\textsuperscript{50} through ORION implemented complex objects using linked and variable data structures. The complex objects are stored in the files with object identifiers and variable fields. Kim\textsuperscript{49, 50} also compared two alternative organizations of objects in a file. In WISS (Wisconsin Storage System)\textsuperscript{35} of uninterrupted bit string, an
object must be access fully even if a part of it is necessary. It occurs because storage of objects stores the complete object as a long record in a file approach whereas in a complex object approach the complete object is being stored as a collection of linked records, so that every element (tuple) of the object may be accessed partially/independently. When accessed, objects (fully or partially) are brought to the application workspace and there the objects are organized as a network of object structures. An object structure is a data structure that identifies an object and maintains access information, which is used for improvement of efficiency of access and update.

Considering this aspect of object storage many researchers attracted to modular technique of database design 22, 23, 30, 34. These modular techniques are models and aids that help the designer in solving the problem they are facing in regards to database management. A module is characterized by an interface for communication with external objects. The internal schema of a module implements the structure and physical concepts necessary for handling requests such as status of the title, identification of borrower to whom the book is issued or translating the book details to barcode system for generation barcode labels etc, received through the hidden internal schema of a module may change independently in any other mode. A program is nothing but an interaction among modules. This philosophy underlying the object oriented programming entails certain advantages, which are important for large scale DBMS design 47. These include

Modularity: Independence of modules ensures modular development of software. Isolation of internal schema of a module from others alleviates modification of the implementation of a module independent of others. This reduces both the cost and time of development of a system, and makes debugging easier. Any modification in the circulation subsystem may achieve with out disturbing the serial control module.

Security: The reduced and controlled interaction among modules offers greater scope for developing effective authorization models to enforce security in such a system, especially since each object is given the power of judgment. Security models can further
be distributed. Also different levels of security on objects can be enforced at different levels of abstraction of objects [25, 41, 43, 56, 59, 84]. The data about a book created by the Acquisition section may not be required by the Technical Processing section as a whole. Only the required portion of the data tuple can be displayed for modification through the Technical Processing section.

This situation may be described as Distributed System model. It is an inherent feature of the object oriented style of programming. Several object oriented programming languages had been proposed in the literature e.g. Smalltalk, [19, 22, 10] C++, [46, 49, 50, 81], Common loops [18, 30] etc. A repository of such object oriented programming languages for object oriented data model may be found in 1, 2, 7, 17, 18, 19, 20, 21, 24, 30, 41, 46, 47, 48, 51, 58, 68, 71, 75.

Object oriented data models evolved following the same philosophy as object oriented programming language. In fact this had added another approach to the evolutionary history of programming languages and data models. An object oriented model retains the basic type structure of semantic data models. Although there is not much consensus among the different object oriented data models proposed so far, it is still possible to identify certain common features that all the object oriented models seem to agree upon to apply the technique to different aspect of library management such as periodical binding, budget allocation of books and periodicals.

Albano [9] proposed database language in the style of functional programming [11, 24, 75], where functions have class values, with static type checking. The query language of proposed is interactive. The type consists of the type [21] constructors like tuple construction, sequencing, discriminated union, array, function and abstract data type. The abstraction mechanisms include classification, aggregation and grouping.

Studies using these features are object identity, explicit semantic constructs like aggregation, grouping, abstraction or specialization, and behavioral description based on messages and methods had been made world over in the last decades for almost every aspect of library operations such as selection of titles, determining multiple copy requirements, selection of journals and establishment of loan facilities, etc.
Comprehensive database systems based on Object oriented model have also been reported in, ORION 50.

With the introduction of object oriented paradigm as a new style of programming, data model research found a new direction. Object oriented programming (or OOP in short) is based on the notion of abstract data types, which enrich the type of structure by abstracting operations on objects along with the structural information. Object types define not only their structural associations but also operational descriptions2, 4, 7, 9, 13, 47, 63 of the objects. Hence object oriented data models are powerful enough to support natural integration of structural as well as behavioral features of objects.

An object in OOP16, 83 is not merely a structured data of a particular type. It is an individual, which possesses the capability to express its behavior. The behavior of an object is an integrated part of the object itself. Other objects can send messages to an object requesting it to exhibit certain behavior or to provide some service. In this sense an object is a module.

- Object: Objects can be of two kinds primitive and non primitive. A primitive or standard object is a built-in object identified by its value. Literals like integer, floating point numbers, character strings etc., belong to this category of objects. Non-primitive objects are constructed from primitive objects using abstraction and other semantic constructs. Object identifiers identify these objects.
- Class: Objects sharing some common properties are grouped in a class of objects. The common features shared by all objects in a class are used to specify the class. Class specification forms the basic component of an object oriented database (OOD) schema. Properties of a class include structural and behavioral inter-relationships among objects.
- Hierarchy: Classes in an OOD schema are organized in a hierarchy induced by ISA relationships in the same way as the type hierarchy of
a semantic data model. In fact, classes in an OOD schema are quite often treated as somewhat similar to abstract data types.

- Aggregation: Aggregation is an explicit semantic construct used for construction of composite object classes. Objects of component classes hold is-part-of relationship with the composite object. An object class may thus be defined by aggregation of attributes and parts. The distinction between attributes refers to an intentional property of an object, e.g. statement or responsibility or second author of the book. Most of the systems use one concept or the other.

- Grouping: Grouping is another common semantic construct used for defining an object as a collection or set of other objects. For example, a LIBRARY may be regarded a collection of objects consisting of Books.

- Messages and Methods: Messages and methods are used as the building blocks for behavioral modeling of objects. Messages establish the interface with external objects. External objects can send a request to an object by sending an appropriate message to it. The receiver of the message then executes one or more methods, which are nothing but codes for execution, for serving the message. The methods are the implementation codes, which are invisible to the external objects. Methods associated with a message may be modified without any side effect on the external entities. External objects may send the messages as long as the message interfaces do not change.

Classes in a schema are named. The ISA relation defines the inheritance hierarchy. A Class in a schema in ORION, IRIS schema, or in O₂ may have multiple super-classes and whose properties may be inherited from the super-classes selectively. Attributes and methods of a super-class may also be overridden in a subclass. While inheriting properties, conflicts may arise because of the same name being present in different super-classes. Such conflicts are resolved either by the designer of the
schema or based on certain priorities assigned to the super-classes. A formal methodology for resolving name collisions in a multiple hierarchy is presented in\textsuperscript{8,59}.

An O\textsubscript{2} class is associated with a concrete type constructed using tuple and set constructor\textsuperscript{34}. The concrete type defines the complex object structure of a class of objects in O\textsubscript{2}. An O\textsubscript{2} system uses structural subtyping for checking type compatibility of objects. A Type T\textsubscript{1} is said to be a structural subtype of another type T\textsubscript{2} if all the attributes present in T\textsubscript{2} are present in T\textsubscript{1}.

In the event of inheritance hierarchy allowing overriding and selective inclusion of properties from super-classes, inheritance and subtyping cannot be treated equivalently. The difference between inheritance and subtyping may be characterized after Bertino et al\textsuperscript{15} as follows. Inheritance is a reusability mechanism allowing a class to be defined from another class, possibly by extending and /or modifying the super-class definition. A type T\textsubscript{1}, on the other hand, is a subtype of a type T\textsubscript{2} if an instance of T\textsubscript{1} can be used in place of an instance of T\textsubscript{2}. Subtyping is characterized by a set of rules ensuring that no type violations occur when the instance of a subtype T\textsubscript{1} replaces an instance of a supertype T\textsubscript{2}.

If there is a possibility of overriding of properties, there also lies the scope of violating such typing consistencies. A method inherited from a super-class C to a subclass D may involve an attribute A, which is possibly overridden in the subclass D. In such a case, the guarantee of typing consistency is not immediate from the inheritance hierarchy. However, when the inheritance is total such problems do not arise. In such a case, one may treat the inheritance equivalently as subtyping, treating a class as a representation of a type.

Object instances are generally unnamed. Each non-primitive object is given a unique internal identifier, by which it is referred to by other objects. Objects are accessed and manipulated with associative queries. However, a system like O\textsubscript{2} allows naming of objects so that an object can also be accessed directly by name.
In case of unnamed objects, referential integrity is maintained by allowing the objects/sub objects to be deleted only if they are not being referred to. IRIS models methods as functions, which are implemented by operations. An IRIS\textsuperscript{89} operation is a computation that may or may not return result. An operation is defined on types and is applicable to instances of the corresponding types. The model underlying the language is the relational model. IRIS functions may be of different types including

- **Stored function:** A function in IRIS may be stored as an explicit table and access using standard relational database operations.
- **Derived function:** Derived functions are specified as a method to derive these functions in terms of other known functions. These definitions are compiled into internal relational algebra representations that are interpreted when the function is called.
- **Foreign function:** Functions written in some external general purpose programming language like C may be plugged into IRIS by linking the executable code through foreign functions. The code is executed when a foreign function is invoked.
- **Update function:** An update operation changes the future behavior of a stored or derived IRIS function.
- **Rule:** Rules are modeled as functions, which can be multi-valued or single-valued. Rules may be conjunctive, disjunctive and recursive.

ORION extends Common LISP\textsuperscript{88} with object-orientation. The language is syntactically similar to other such extensions of LISP for object oriented programming. ORION provides system messages (i.e. message that may be sent to the OOD system) to create a new class, define its relationships with other classes in the inheritance hierarchy, and define its attributes and methods to be inherited from different superclasses. Class messages (i.e. messages that may be sent to a class rather than to an object of the class) create instance of a class and assign values to the attributes. Associative queries allow selection of all or any of a set of objects qualifying a predicate. Messages for deletion and change of an instance are also provided. ORION further
supports querying complex objects recursively up to different depths \(2, 4, 12, 43, 50, 59\). Static provides a LISP based language with associative access.

\(O_2^{34}\) supports a uniform interface with the database management system to an arbitrary host language like C, Basic etc. When \(O_2\) is interfaced with the language \(L\), it is named \(L O_2\) (e.g. \(C O_2\), \(B asic O_2\)) and the programming style of language \(L\) is followed in \(L O_2\). The essential database language features provided include naming and updating an object and associative access.

An SQL like language, RELOOP, based on relational algebra had been proposed by Cluet et al for querying \(O_2\) objects. Its translation to \(C O_2\) had been proposed by DEUX\(^{34}\).

Abiteboul\(^1\) has proposed a deductive object oriented database language. This project has the following objectives.

- should provide an easy to use language for query and update.
- should be extensible for new data types.
- should support rich data types while maintaining the traditional separation between schema and instance.

The model of the object oriented database language is based on the co-existence between two concepts-class and relations. The language enforces a strong type checking with well-typed expressions. Types can be either base types e.g. \(i nt\), \(f loat\), \(n um\) etc. or constructed types such as tuple types, finite set types, finite list types, union and intersection type\(^1\).

Though the language is somewhat declarative, defined by rules similar to stratified Datalog rules, it is not purely declarative in the sense that the rules are associated with some control constructs.
Methods can be either stored functions or derived functions. A method can again be either single-valued or multi-valued.

Rules of the form \textit{body} <- \textit{head} are constructed, where \textit{head} is a positive literal and \textit{body} is a list of positive and negative literals. Rules are stratified with respect to negation as in stratified datalog. Terms and literals are constructed using constants, tuples, sets, lists, dereferencing of object ids, projection, relation and class names, and method applications.

External functions written in a host language can be interfaced to the rules maintaining the appropriate type structure. Stratified fixpoint semantics of the rules had been provided by Abiteboul. A few explicit controls like execute once, sequencing etc had been suggested in order to easily express some applications which would otherwise involve complex declarative structures.

Data update is again affected by rules. Creation of objects is controlled by variables in the head of a rule that do not appear in the body. These variables are existentially quantified so that an object is created by an instance of the rule that satisfies the body provided there is none in the database satisfying the head. This also terminates the possibility of an infinite number.

Galileo provides an object oriented database language in the style of functional programming where functions are first class values, with static type checking. The query language of Galileo is interactive, with type system consisting of the type constructors like tuple construction, sequencing, discriminated union, array, function and abstract data type. The abstraction mechanisms include classification, aggregation and grouping.

The language GODAL supported a rule-oriented paradigm with polymorphic typing. Here types are treated as first class values, which allow dynamic typing.
2.6.1 Storage Organization.

ORION implements complex objects\textsuperscript{50,51} using linked and variable data structures. The complex objects are stored in the files with object identifiers and variable fields. A root record in a database file stores primitive object fields (integer, float, string etc.) fields of variable length (e.g. string) are stored as pointers to RAM (random access memory) for common storage of such variable fields. Variable size complex components are sets, which are implemented as linked list of elements. A complex object structure is organized into a configuration DAG (directed acyclic graph). In order to efficiently evaluate a recursive query, ORION clusters the nodes of a configuration DAG so that disk access is minimized. ORION identifies the following properties to be ensured by a clustering mechanism to minimize disk access.

1. **ABD (Ancestor before Descendant) Property**: Any given node must precede all its descendants.

2. **CD (clustered Descendants)**: The descendants of a given node are clustered.

ORION maintains various indexes of objects. Secondary indexes of complex objects may be created based on path-expressions. A path-expression is specified as a composition of attribute names, which is nothing but a form of implicit join among objects. For example, the path expression \texttt{w.Location.stack}, for an object variable \texttt{w} type \texttt{books} refers to the stack of Location of the book instance \texttt{w}. With path-expression indexes, objects can be easily found to relate a particular object through the path expression. Several variations of such path-expression indexes are available for ORION objects\textsuperscript{50}. Indexes of objects in an ORION class may be organized in any one of the following different ways. A **single-hierarchy indexing** on some attribute in class \texttt{C} constructs indexes of objects that are instances of and none of its sub-classes. Alternatively a **class-hierarchy indexing** may be used to construct indexes on all the object instances of the class \texttt{C} and of it subclasses. A single-hierarchy indexing is useful for an associative search over instances, which belong to class \texttt{C}, and to none of
its sub-classes, whereas a class-hierarchy indexing is useful when the search involves all the instances of the class and its sub-classes.

Storage organization of static recognizes three levels of abstraction. The lowest level, file level concerns access and management of pages in file. At the next higher level, storage level data is viewed as a collection of records. Management of indexes is done at this level, where large records are organized as tree of blocks. Sets are implemented as B-sets, which are extensions of B-tree. The topmost level of storage abstraction, the function level, deals with processing of queries. At this level objects and their inter-relationships are interpreted at the schema level of the schema.

2.6.2 Integrity Constraints.

The rich semantic constructs provided for semantic and object oriented modeling capture most of the constraints that normally arise when modeled with classical data models. For example, the ISA relationship captures the inclusion dependency of relational databases. Attribute relationships are functions; hence such relationships could be used to represent functional dependencies. Similarly, the grouping construct could be used for modeling multi-valued functions.

Static integrity constraints like disjointness constraints, covering constraints which are similar in the case of ODDs with semantic data models, and their interactions with ISA relationships had been discussed in the works of Atzeni and Parker\textsuperscript{10} Lenzerini\textsuperscript{57,58}. Theoretical results concerning the computational complexity and tractability of these constraints had also been reported in these works. Such constraints are specified in the form of equations and containments. For example, a constraint equation \texttt{DEPT.EMPLOYEES.WORKS_UNDER = DEPT .MANAGERS}, introduces a constraint over departments (DEPT) that the set of managers under whom the employees of a department works is the same as the set of managers for that department. In this equation, it is assumed that the class \texttt{DEPT} has the set valued attributes \texttt{EMPLOYEES}
and MANAGERS, and the class EMPLOYEE has an attribute WORKS_UNDER, corresponding to the manager under which a particular employee works.

Buchmann, Carrera and Galindo\textsuperscript{23} had identified value based constraints such as range constraints (restriction of the domain of values that an object may assume), enumeration constraints (which enumerate certain elements that must belong to a domain of objects) and relationship constraints (which require some arithmetic-logical relationships among certain objects). Buchmann et al\textsuperscript{23} had also identified some structural constraints involving the structural aspects of objects. Structural constraints are again classified as component classes, number of component objects and structural relationships between component objects.

Event-trigger mechanisms\textsuperscript{53} are used for modeling dynamic constraints in many systems. A constraint is specified as a pair of event and trigger, which specifies that the said trigger will be raised on occurrence of a particular event. When these events occur, there is a potential of consistency violation and the trigger is raised. The action of the trigger is to invoke particular methods that enforce the consistency. An approach based on preconditions-postconditions had been suggested\textsuperscript{62} for ensuring consistency on execution of a method. A pre-condition of a method may be specified which is checked before execution of the method. The method is executed only if the pre-condition is satisfied by the current state of the database. At the end of execution of the method the post-condition is checked. The effect of the method is disregarded if the post-condition is not satisfied by the database, including the desired effect of the method. Applications that are expected to use OODBMS, e.g. Bibliographic information, CAD, Image processing, Cartography etc. require some functionalities of a database management system, which are not provided by traditional database systems. These functionalities include version mechanisms and long transactions for concurrency control.

2.6.3 Transactions and Concurrency Control.

Transaction in many ODS applications\textsuperscript{23} such as CAD, unlike the traditional database applications which could be very long, possibly continuing for days together. In some
cases, it may not be possible to lock objects throughout the duration of the transaction. Keeping locks for a long duration would drastically reduce the degree of concurrency. Moreover, when multiple versions of an object are supported, there may be several valid states of a given object simultaneously existing in the database. Such situations did not arise in traditional database systems. Concurrency control and recovery schemes in object oriented databases must therefore take care of these special requirements.

A check-in/check-out protocol is frequently used for transactions in ODS environments. When a transaction requires an object, it locks the object in the database, checks-out a copy of the object and places it in its private database. All updations in this copy used to be made in the private database independent of other transactions. When the transaction commits, this updated object used to check into the database (also called the public database). Locks are released on a successful check-in or on an abort. These locks had to be maintained as long duration locks, and were placed in secondary storage so that they could survive system crashes.

An object, once checked out by a user, is not available to other users unless it is checked-in. In order to make distributed of incomplete objects among users cooperating as a group of a design, Kim et al\textsuperscript{50} suggested maintaining a semi-public database. The incomplete objects but in a state to be used by other cooperating designers may be checked into the semipublic database. Other users may check the object out of the semi-public database for their use. However, if a transaction $T_i$ checks out an object, say $A$ from the semi-public database already checked-in by $T_j$, $T_i$ is said to be dependent on $T_j$.

Most object oriented database management systems use locking schemes for concurrency control. ORION provides both object level and page level locks. An object level lock locks the complex object of a class accessed in ORION, where the class is also locked so that the class cannot be modified by other transactions. So schema evolution is required in this regards.
2.7 Schema Evolution.

Schema evolution refers to dynamic change of a database schema. Over the period of development of a design, it is often necessary to modify different aspects of database schema. Such changes may involve construction of a new class of objects, deletion of a class and modification of specification (structural as well as behavioral) of an existing class of objects. Kim and Nguyen had presented taxonomy of basic schema changes. Possible schema change operations include the following.

- Changing class definitions, e.g. adding, renaming and dropping an attribute or a method, and changing the inheritance of an attribute or a method.
- Modification of the inheritance hierarchy is changing the relationships among classes.
- Addition, deletion or renaming of an existing class.

Schema change operations must keep the database schema consistent. This involves consistency of method definition and other constraints. For example, as an attribute is deleted from a class, all the methods referring to the attribute either directly or indirectly must be appropriately modified. One important issue related to support for schema changes is how to propagate such changes to the instances of the affected classes. However, this issue concerns only some types of schema changes, e.g. the deletion of an instance attribute from a class. Other changes, such as deletion of a method, do not affect instances. Different approaches had been suggested for dealing with schema changes.

These approaches based on deferred effect delays the propagation of changes; even indefinitely. The change is affected on an instance only when it is required. In this approach, so as an attribute is deleted, it may never be physically deleted from the corresponding object records. The argument behind the deferring approach is that immediate propagation requires a reorganization of all the instances, which can be very
expensive. In the deferred approach, objects accessed are post-processed (e.g. screening out the deleted attributes) for presentation to the user. The post-processing is necessary at every access, which may increase the cost of access.

In a prototype of object oriented knowledge base system called Sherpa Nguyen\textsuperscript{69} suggested dynamic propagation of effects of schema changes to instances. This approach is based on an extended notion of object class, called relevant classes, which systematically take into account the partial completeness of the objects. This propagation is achieved by characterizing the modifications in terms of the relevant classes and then grouping the changes to simultaneously update the instances belonging to the same relevant class. This approach thus provides for both immediate and deferred update. The effects of schema change on existing methods may require recompilation of methods. For example, a method can invoke another method which is modified. Similarly, deletion of an attribute of a class may invalidate a method that accesses it. This issue arises in systems where type checking and binding are executed at compile time.

System O\textsubscript{2} provides a technique that supports two application-running modes: the development mode and the execution mode. In the development mode the application designer can interactively modify the schema without having to perform extensive recompilation, since this mode supports late binding. The end-user works in the execution mode. In the execution mode, on the other hand, the schema is compiled to generate an optimized code for methods. The schema in this mode is assumed to be static and schema change operations are disallowed on the compiled portion of the schema. This strategy integrates the flexibility of application prototyping with performance and safety considerations. Let now the theoretical issue of OODB be reviewed.
Though object oriented database systems are gaining importance and are being applied in several problem domains, many theoretical issues related to this model still remain unresolved. Some of the theoretical issues related to OOD are mentioned below.

**Formal Model:** An appropriate formal model of object oriented database systems is lacking. The great variety of concepts used in different object oriented systems developed so far makes the development of a single formal model difficult, if not impossible. Beeri\textsuperscript{14} suggested a logic-oriented approach for development of a formal framework that contains most of the features found in current object oriented database systems. This framework contains two parts. The first is a structural object model, that includes the concepts of structured objects, object identity, value and some form of inheritance. Values and object identities are distinguished and two different forms of equalities (identity and value equality) are examined. An object system is described as a directed graph where nodes of the graph represent objects and labeled edges represent relationships such as attribute association, membership etc. Calculus based declarative languages had been examined as a possible model of query languages. The second part deals with higher-order concepts, such as classes and functions as data, methods and their associations with objects, and inheritance.

Logic based query languages and updates in ODS environments had been analyzed by Abiteboul\textsuperscript{1,2,3,4,5,6,7}. Abiteboul explored the possibilities of using declarative languages based on object identity\textsuperscript{7}. Both stratified and inflationary semantics of such programs had been discussed along with complexities of such query languages.

Hull\textsuperscript{42, 43, 44} analyzed the expressive power of object oriented schema and languages. An ODS schema is said to be regular\textsuperscript{44} if its methods are implemented using basic assignment operators, message call, and creation/deletion of objects, insertion/deletion/assignment operations on sets, and looping construct \textit{for each}. A regular implementation also allows conditional \textit{if-then}, with relational operations on variables.
and membership tests for different types, nesting of conditional statements are not allowed. Hull\textsuperscript{44} had demonstrates that object oriented queries with assignment, relational algebra operations and point-wise method applications over a schema with regular implementation of methods which allow creation of objects by a method are at least as powerful as the class of computable queries\textsuperscript{32}. It has also been shown in the same work by Hull that queries over object oriented databases, which allow set constructor as an explicit typing construct are intractable, even when creation of objects is not allowed in methods. On the other hand, prohibiting set as an explicit type constructor and allowing modeling of sets with multi-valued attributes and multi-valued methods renders the queries tractable. Such a class of databases with regular implementation of methods is termed algebraic and contains the class of transitive closure queries.

The present work attempts to design a bibliographical database management system with all semantics of the fundamental features of an object oriented system on a uniform basis for library.
2.9 Reference:


21. BRUCE, K. B., and WEGNER, P. An algebraic model or subtype and inheritance. *In Advances in Database Programming Languages*, F. Bancilhon and P.


34. DEUX, O. et al. The story of O₂. IEEE Transactions on knowledge and Data Engineering 2, 1 (March 1990), pp. 91-108.

35. CHOU, H. T. Et al. Design and implementation of the Wisconsin Storage System. Software Practice and Experience 5, 10 (October 1985).


41. HESELINK, M. A mathematical model of non-determinism in abstract data types. ACM Transactions on Programming Languages and Systems 10, 1 (January 1988), pp. 87-117.


67. MIYAZAKI, N. Selection propagation in deductive databases. Data and 

68. MOSS, J. E. B. Semantics for transactions in shared object worlds. In Advances 
in Database Programming Languages, F. Bancilhon and P. Buneman, Eds., 

69. NGUYEN, G. AND RIEU, D. Schema evolution in object oriented database 


71. OHORI, A. Orderings and types in databases. In Advances in Database 
Programming Languages, F. Bancilhon and P. Buneman, Eds., ACM Press, 


Co., 1952.

74. ROTH, M., KORTH, H., and SILBERSCHATZ, A. Extended algebra and 
calculus for not 1NF Relational Databases. Tech. Rep. TR-85-19, University of 
Texas at Austin, 1985.

75. SHIPMAN, D. The functional data model and the data language DAPLEX. 
ACM Transactions on Database Systems 6, 1 (March 1981), 140-173.

76. SOWA, J. Conceptual Structures: Information processing in mind and Ma­ 

77. STEFIK, M., AND BOBROW, D.. Object oriented programming: Themes and 

78. STONEBRAKER, M., Ed. The INGRES papers: Anatomy of a Relational 

79. STONEBRAKER, M., and ROWE, L. The design of POSTGRES. In Proceedings 

80. STONEBRAKER, M., WONG, E., KREPS, P., and HELD, G. Design and 
implementation of INGRES. ACM Transaction on Database System I, 3 
(September 1976), pp.189-222.


89. WILKINSON, K., LYNGBOEK, P., and HASAN, W. The Iris architecture and implementation. *IEEE Transactions on knowledge and Data Engineering* 2, 1 (March 1990), pp.63-75.