Unit 4:Object Oriented Database Systems

Syllabus: -

Unit 4: Object Oriented Database Systems **LH 3**

- Performance Issues in OODBMS
- Application Selection for OODBMS
- The Object Oriented Database Paradigm Manifesto
- The Mandatory Features ➢ The Optional Features

Performance Issues In OODBMS

Performance evaluation of OODBMS products is a very new area and no standard performance metrics have been evolved. Even in the judgment on the type of performance, there has been substantial disagreement. One point of view emphasises that query processing is the key to performance and there should be declarative query statements to be optimised by a query optimiser. On the other hand, others feel that pointer processing or pointer chasing is the prime concern of performance. Accordingly the performance metrics for OODBMSs is also debatable. However, some metrics proposed (Joseph John et al., 1989) include the following:

1. Query processing rates
2. Bulk operation of single operation on set of objects
3. Saving and retrieving of complex object states by the object identifier
4 Speed of complete operations at the application level.

There have been no universally acceptable OODBMS benchmarks, On the Oher hand, there have been many benchmarks in the market for relational DBMSs, but benchmarks comparing relational systems with object-oriented System are even harder to define and measure, for several reason. For example, sharing of responsibilities between DBMS and application is done differently in these disparate environment.

In extended relational system an extended type code execution speed must also be considered. Finally, It is also widely believed that the two environments do not address the of same application domains, in which case it can be easily shown that, for a class of applications, one of them is faster than the other. In other words, while a benchmark would 'prove' that one system is N times faster than another, It would illustrate only a certain class of applications for which a system is good. We shall, in the following sections, examine specific performance issues from a technical angle.

Object-oriented abstraction support

The nature of object-oriented abstractions itself may pose potential performance problems. This is based on certain implementation techniques namely dynamic binding of messages to method code and automatic storage management in the form of garbage collection, in which are essential for supporting object-oriented abstraction.

Dynamic memory dispatcher

As in Smalltalk, the classic object-oriented language, methods are usually not called by name, but indirectly at run time through a dispatcher table associated with object class. If the message is not found there, then the entire super class hierarchy of the object is searched, and so on, until the root class of the object is reached. If the message is not found even there, then

an error is reported. Even though this allows substantial flexibility, generally associated with object-oriented programming, it adds an additional overhead at run time for method look up and context switching vis-à-vis a simple subroutine call and, therefore, adversely affects performance

In certain other systems supporting a stronger type mechanism than in Smalltalk, it is possible to use the type information itself to bind many, if not all, of the messages to methods at compile time. This is followed tor C++ in Vbase and other C++ based OODBMSs. Research (Chambers et al., 1989; Connors et al., 1989; and Dixon et al., 1989) indicates that the application of advanced compiler and other implementation technology to object-oriented languages and systems provides substantial potential for major performance improvements, even in systems with object models more flexible than Smalltalk.

Garbage collection and referential integrity

Many object-oriented environments eliminate unnecessary objects auto objects automatically using garbage collection instead of requiring users to explicitly delete when they are not needed anymore. Garbage collection is performed when no reference to objects exists in the system-maintained objects. They obvious advantages: the programmer need not worry as to when to delete an object.; it is also guaranteed that references to an object that might exists in other objects are always valid (since the fact that a reference exists object exists). This is even more an advantage when objects have complex interrelationships with other objects are required to be maintained. However, this leads to consequent performance overheads of garbage collection.

Therefore, when performance is critical, explicit deletion is required to be provided. Thus, there is a trade-off. Garbage collection is also related to semantics. In Smalltalk, again, a class describes the intended use of its instances by describing their structure and collection of all its instances.

Garbage collection precisely matches the intentional semantics, since, once the last reference to an object disappears, there is no possible way to access it within the system and, therefore, its space can be safely reclaimed. On the other hand, extensional semantics requires all instances (such as find all employees) even when more explicit inter-object references to employee object do not exist. In such cases garbage collection would not make sense at all, since a reference to each object instance (employee object instance) from the class would always exist.

High performance oriented systems, however, avoid a performance overhead for garbage collection by placing storage management entirely under the programmer's control.

Referential integrity is another consideration. Referential integrity of complex objects with deeply structured and inter-related objects necessitates Supporting complex forms of structural integrity that may be required in the new applications aimed at by object management support from relational extensions. Basic referential integrity defined for a single relation system is an extremely simple case. Maintaining referential integrity by relational extension is in no way simpler than what would be required in non-relational object-oriented systems.

Concurrency control techniques in OODBMSs

Conventionally concurrency control is a mechanism of serialization of contending parallel transactions on the same data in the database. concurrency control algorithms have, generally, been based upon two-phase locking and time stamping so as to synchronise accesses to shared data, both in centralised and distributed database situations.

However, in the OODBMS environment the conventional so due to the fact control mechanisms are inadequate or restrictive. This is so due to fact that many object oriented application deals with long and complex object involved in long-lived activities, such as long field transactions.

A long transaction which takes considerable amount of time for execution may lock out a large part of the database for a lone be time in the pessimistic approach and may defer the settings of lock but may be aborted when it attempts to commit in the optimistic approach. The volume of work involved in a rollback on such an abort may be too large and may acceptable. Further, in databases based on conventional models, concurrent transactions are assumed to necessarily conflict and cannot be prevent from concurrently accessing shared data. On the other hand, in the object oriented environment the object may span a layered domain from simple values of variables (in traditional programming languages) to large complex object.

Contention is not very likely on objects corresponding to program variable.. Imposing conventional pessimistic concurrency control on such single will lead to unnecessarily overhead. In some cases such as , knowledge based application, the processes (i.e. programs) may require partial results provided by other processes.

On the other hand, for large transactions and situations of cooperation of concurrent processes, various concurrency control mechanisms have been proposed: multilevel atomicity, check-in/check-out of objects, "blackboard" mechanisms for processing messages and partial results, and support for various types of locks and notification rules. In a multilevel atomicity approach, a set of transactions is recursively partitioned into equivalent classes and transactions within the same equivalence classes are synchronised less strictly than transactions from different equivalence classes. In a check-in/ check-out approach, objects are checked out in a few specified modes, permitting or disabling a concurrent check-out by any other program.

Other sophisticated techniques include pre-analysis for determining how best to schedule potentially conflicting activities, and optimistic concurrency control mechanisms which permit the creation of multiple concurrent images of an object, followed by execution of arbitration procedures to resolve inconsistencies among the object versions.

Seamless application interface

Conventionally, any DBMS always provides persistent storage for its type system. On the other hand storage for object-oriented programming language is not possible beyond the lifetime of a single program execution, unless the programmer give special instruction to save the object in a file or disk or uses DBMS as an intermediate agency. Such languages have been conventionally known as host languages. Conversely, for recalling an object created

earlier execution of some other program, the programmer must necessary give an explicit instructions to fetch the object from back store or from DBMS.

This implies the responsibility of the programmer for managing the storage of the objects and also to write program code to translate between the internal representation of the object in the program space (main or virtual memory) and their external representation on the secondary storage on the file or relations, which is quite different from main memory forms as arrays, etc. relations, Such mapping may be extremely complex in object-oriented environment. Complex pointer manipulation and type checking have to be taken up by the programmer while modern programming languages support elaborate type strong type checking.

No type checking may be done when the object is written on to disk , if a conventional file system is used or a different type system may exist if a DBMS is used. Such data written on to disk can be modified by a different program, thus leading to unpredictable types of the same data when it is read back.

This problem has been partially addressed by some modern programming languages by integrating a language and a DBMS. An example of such languages is ADAPLEX (Smith, 1983). These languages also contain powerful query facilities for retrieval and specific facilities for transactions. Nevertheless, the programmer sees two different type systems-one for the programming language itself and the other for the database.

Therefore, a seamless or uniform type system in object-oriented language programming (which is supported in an OODBMS) or the ODBMS itself, makes life much easier for the programmer and offers many advantages in programming development and debugging as follows:

1. No conversion software is required to be developed by the programmer (for converting the programming language representation to the database representation) on the same type used in both the language and DBMS.
2. Software already developed in a non-persistent object-oriented environment (like C++) can be directly imported and used in the database environment without modification (while providing necessary extensions for the DBMS).

3. Programming activity in a language can be continued in a DBMS environment also, without any additional learning requirements.

However, difficulties in providing seamlessness can be encountered in certain situations like when the DBMS supports multiple languages which divergent type systems or when a given language has absolutely no feature that will enable implementation of a database transaction.

Query processing issues and indexing

Due to the larger overheads involved in OODBMS environment (in comparison with a simple relational model), query processing and query optimization technique assume greater importance . Efficient query processing depends on use of pre computed results, in the form of Index. Indexing can be complex for an OODBMS.

For example, When inheritance of properties or attributes takes place, indexes are required to be crated to meet the particular situation therein. While an index require a definite procedure of an object instance , inheritance may not assume that. Therefore indexing in such cases requires type declaration that ensures regularity which is assumed or an index to be created. For example. colour red an index may be created. But in an inherited attribute may never be inherited for certain subtypes while some others may inherit a colour.

An index is required to be always up-to-date and this becomes difficult in an OODBS (or Relational extension) because it is difficult to keep track an output from a method or function. For example, it income depends interest rates which may change depending on the method, to keep an index on income necessitates an update of the income every time the method is executed. There may be many such derived data cases in a database, which need to be tracked constantly to maintain an index up-to- date.

Further, indexing on a structure may be required to keep it up-to-date but that may adversely affect the privacy of the object as it bypasses the protocol of the object. One solution suggested (Graef and Maier, 1987) for this problem is to reveal facts of the internal object structure in the object protocol and allow an index only on revealed structures.

Query optimisation techniques also improve performance in query processing. The issues involved in query optimisation for an OODBMS environment are quite different and complicated as compared to those tor a relational DBMS, since the relational model contains a fixed set of operations (such as selection, projection and join of relational algebra) and simple normalised relations with fixed access methods of the DBMS. The OODBMSS environment, on the other hand, contains no such simplistic operators and functions, and the different O0DBMS products have adopted different query processing constructors and functions.

However, in OODBMSs, extensible or arbitrary data tapes and associated operations can be defined and added to the system along with user defined access methods to support them. These queries in OODBMS contain arbitrary combinations of such user-defined operations. Each new type associated new operation creates a new algebra, unknown to the query optimiser. Without prior knowledge of the algebra, the optimiser discover the equivalence presenting transformations for optimiser process. Further associated access method of objects are also generally encapsulated within the interface objects . Thus, the query optimiser cannot determine the cost of transformation when the access method is hidden in this manner.

The optimiser, therefore, must also be extendable. When the object is defined, the methods for implementing operations of the class must be defined. These descriptions can be used by the optimiser in determining for processing a given query.

Research on these query optimisation issues has been reported for relational extensions (Stonebraker, 1986a and 1986b) and for OODBMS (Graef and Maier, 1987). Query optimisation is essential for improved performance of OODBMSs, if they have to survive in the market with stiff competition from the large RDBMS and ORDBMS product range.

APPLICATION SELECTION FOR OODBMS

The applications areas of OODBMS are multimedia, CAD/CAM, GIS, CASE tools, etc., which require data management support for complex and varying object datatypes. These datatypes cannot be adequately supported by relational mode-based DBMSs.

However, OODBMs are also increasingly receiving attention in traditional business applications such as financial services, computer-aided publishing, desktop publishing etc. In fact, some OODBMSs (VISION, for example) are utilised in the financial service sector as a primary application, since some core financial modelling can be resolved using large complex datatypes or objects. Similar desktop publicising application requires numerous fonts with associated graphics and pictures. An OODBMS will enable representation both of the characteristics of individual layout elements and the role and location of each element in the overall layout. Other features which such applications need are common to CAD/CAM, requiring highly interactive user interfaces.

When there are complex entities (or objects) involved in the application, these complex objects are interrelated in a complex manner so there is no choice but to go for an OODBMS. If the same application is implemented in the relational environment, will lead to a significant loss of information and knowledge content of the application.

So, in order to determine whether or not to adopt OODBMS, the following knowledge will help to take better decision making :

1. Functional requirements of the application
2. Structural complexity of the database .
3. Adequate performance for the application
4. Ease to use features
5. Price-performance ratio.

However, applications can be broadly categorised as:

1. Those which necessarily require object features, and whose performance requirements also demand DBMSs with a highly Specialised object-oriented implementation.
2. Those which necessarily require object features but do not necessarily require high performance ratings.
3. Those which are fairly traditional applications but need OODBMS only for consistency with object-oriented analysis and approaches of software engineering in a system's life cycle.

Multimedia or CAD/CAM fall in category 1 mentioned above. In such cases the mismatch between them and the set oriented capabilities of the relational DBMS is considered to be so great that only an OODBMS can handle them. Since performance is also a critical criterion in such cases, a close integration with programming languages (like C++) features and its efficiency considerations for pointer handling are important.

Sometimes the applications require continuity and interoperability with the industry standard languages, e.g. SQL. In such cases, systems like Iris can be considered, as they provide SQL compatibility. Relational extensions such as POSTGRES also may contend for such application areas.

Another subclass of this class of applications consists of those that require a front end integration tool to combine conventional application requiring relational DBMS with other systems that require object capabilities. Some such complex situations are faced in engineering and aerospace corporations which, conventionally, have a large array of CAD data and also a large amount of conventional (MIS) data (in the relational model).

These two types of data could not be integrated since CAD data could not be fitted in conventional data in a relational model as the necessary facilities required for CAD do not exist in the relational model. In all such classes of applications, the appropriate approach may be to continue with the relational model for the conventional applications and build front ends with OODBMS for CAD data as integration tools to provide access to other data.

Most OODBMS vendors are planning to support interfaces to back-end relational systems. Similarly, most relational vendors are providing, or planning to provide, object-oriented front-end interfaces. Relational extensions like POSTGRES may serve the purpose in a much better manner due to the inherent integration of both the relational and object-oriented approaches. The whole range of ORDBMS products fall in this class with Oracle 8 and DB2/UDB as some of the competing products.

Applications which require OODBMS but less specialised implementation can directly use any standard commercial OODBMS. Applications which may use an OODBMS since they adopted an Object oriented application development approach (category 3 above) have, nevertheless, a choice to continue with relational DBMS at the implementation level.

Thus, they can use better relational or extended relational object oriented DBMS for the implementation phase. In this ace, OODBMS may provide a better fit with development approach and less Overhead in data conversion. The OODBMS based implementation of a system based on OOAD will also provide a seamless interface and a full system based on semantic capture while an RDBMS, in such cases, will not only result in “impedance mismatches” but also in a substantial loss of semantics.

The primary motivation for the object-oriented development approach, both in analysis and design stages, is to support not only existing approach applications but also for future applications involving distributed co-operative processing, and an event driven interactive user interface.

THE OBJECT-ORIENTED DATABASE PARADIGM MANIFESTO

The Object-Oriented Database manifesto suggests the characteristics an OODBMS should pose, these characteristics can be classified as either (a) mandatory or (b) optional and (c) open. While the mandatory and optional features identify the essentials and non-essentials, respectively, the open features offer a choice for the designer of OODBMS to choose one of the many equally acceptable solutions.

The Mandatory Features

I. Complex objects.

Complex objects are either aggregations of simple objects or are obtained by applying functional constructors over them. The simple objects are integers, characters, byte strings, Boolean and floating point numbers. Further, atomic objects can be added. On the other hand, complex objects are tuples, sets, lists, arrays etc. Any OODBMS should support the minimum set of constructors, viz. sets, tuples, lists. While sets and tuples are essential for collections of objects (sets) or properties of an entity (tuples), lists and arrays are fundamental requirements for scientific applications. Any object constructor should be applicable to any object, e.g union and intersection.

II. Object identity

In an OODBMS an object existence independent of its value. Two objects may be identical (i.e. same object) or they can be equal (of the same value) without being identical. Thus, identical objects can be distinguished from equal objects in object sharing and object updates.

Examples of object sharing can be: a child is common to both parents (Father and Mother), i.e. a child object can be shared by both the parents since both of them are parent of same child.

An example object updating with object identity can be: when a child's attributes are updated; the updates so effected will apply to the child (shared) of both parents

In an environment where object identity does not exist separately from object value, the two subjects, i.e. the child of one parent (Father) and the child of the other parent (Mother) have to be separately updated. Thus, object identity is an essential and powerful data manipulation primitive basis for set, tuple and complex object manipulation. Supporting object identity automatically means supporting object operations such as object assignment, object copy and tests tor object identity and equality. In relational systems, relations are value based and object identity, therefore, becomes central to object orientation.

III. Object encapsulation.

Encapsulation originates from abstract datatypes in programming languages. In this environment an object has an interface part and an implementation part. The interface part specifies the set of operations that can be performed on the object and is the only part visible on the object. The implementation part has a data part and a procedural part. The data part is a representation of the state of the object and the procedural part describes (in a given programming language) the implementation of each operation. In a database environment this principle of encapsulation is translated in terms of the object, encapsulating both the programs and the data.

In a relational environment, an entity (e.g. employee) is value based in a tuple and the transaction (procedure in DML) on the entity is meant to be On the other hand, in an object-oriented system an object separate. (employee) has a data part and an operation part. When storing an object in the OODBMS, both data and operations are stored in the database. Thus, there is only a single model for data and operations and the information can be hidden. No operations other than those specified in the interface can be restriction holds good for both update and retrieval performed. This operations. Encapsulation provides a form of logical data independence, i.e. the implementation of an object type can be changed without changing any programs using that object type. Thus, the application programs are protected from implementation changes in lower layers of the system. Only when the operations are visible and data and the implementation of operations are hidden in the object, can we say a proper encapsulation is implemented. Evidently encapsulation leads to greater data independence and greater user convenience.

IV. Object types.

A type in an object-oriented system summarises the Common features of a set of objects, which is similar to the notion of an abstract data type in an object-oriented programming language environment. Independently, types of entities or objects have also been defined and used extensively in semantic database systems, i.e. the type notion has prevailed parallelly in semantic database systems as well in object-oriented as programming language environments and perhaps, it prevailed in semantic systems earlier than object-oriented programming languages (E-R model and RM/T were proposed in the late 1970s; see Prabhu, 1992). A type has Wo parts: the interface and its implementation. Only the interface part consisting of a list of operations along with the type of input and output parameter and visible to users while the implementation part consisting of a data part an operations part is seen only by the designer. The data part describe of the internal structures of an object's data. The operation part consists the procedures which, as already discussed, implement the operations of t king interface part.

A type or an abstract data type becomes an effective tool for checking program correctness in a programming environment by checking, at run into the type against those declared at compile time by the programmer. Thus types are used to check the correctness of a program at compile time – a program cannot be modified at run time.

V. Object classes.

Object classes are similar to object types. Though type and class are often used synonymously in the literature, there are however, subtle differences. Class defines a group of objects with, at least, some attributes in common. Thus, the specification of class is the same as that of type, but it is more of a run-time notion. If we consider the objects belonging to an

object warehouse, to start with, a hypothetical object factory can produce or create new objects by performing new operations on the class or by cloning some prototype object representations of the class. The object warehouse may permit the class to be attached to its extension, i.e. the set of objects that are instances of the class. Operations can be applied on all elements of the class to be manipulated. Classes have no role in checking program correctness. Their role is limited to creation and manipulation of an object. The differences between type and Class are very subtle and, for many practical purposes, they can be considered on the same lines.

VI. Class Type hierarchies and inheritance.

Inheritance of a property or an attribute is a powerful and efficient mechanism for manipulating objects in application situations. In the relational approach, in the absence of a provision for property inheritance from higher classes to lower classes, it is necessary to execute procedures redundantly for all transactions on attributes that may be common between parent and child objects. For example, in a University database there will be Employees and there will be Students. Each employee has a name, age, date of birth, etc., and each student also has the same attributes. Any transaction on their attributes has to be performed separately and, therefore, duplicated on both of them separately in d relational environment. On the other hand, in an object-oriented environment both employees and students will be special cases, i.e. sub-classes or Suo types of a super class person, each instance of which has the attributes name, age, date of birth, etc. With only one transaction on person we ca obtain the desired updates on all the lower level classes, viz. Employees ad Students. These are called type inheritance, property inheritance or attribute inheritance.

Inheritance can be of various types. It can be (a) substitution inheritance, (b) inclusion inheritance, (c) constraint inheritance or (d) specialization inheritance

Substitution inheritance is based on behaviour and not on values. A type t inherits from a type t' if we can perform more operations on objects of type t than on objects of type t'. Thus, t' can be substituted anywhere for t.

Inclusion inheritance denotes classification. In this case a type t is a subtype of type t' if every object of type t is also an object of type t'. Thus, it is based on structure and not operations.

Constraint inheritance is a sub-case of inclusion inheritance. A type t is subtype of t' if every object of t is not only an object of t' but also satisfies a given constraint (e.g. a retired person is a subtype of person with a constraint that the age of the person be more than 60).

In specialisation inheritance a type t is a subtype of t' if objects of t are objects of t' with more specific information, e.g. a student object is a person with some extra attributes as course-no., etc.

An OODBMS can provide for any chosen degree of inheritance features from the aforementioned types of inheritance

VII. Overriding, overloading and late binding.

Overriding is the redefinition of implementation of the operations (as display) for each of the types according to the type. For example, in a conventional system the operation display

has multiple versions, i.e. three different operations that can be used differently on graphs, tuples and pictures as display graph, display tuple, etc. In the object-oriented database environment one single operation display in each case is redefined and this is called overriding. Since this redefinition results in a single name display denoting three different programs it is called overloading. The system is the can pick up the appropriate implementation at run time for the same display operation applied to all cases, viz. graphs, tuples and pictures. This approach has an advantage, i.e. greater independence of the programmer from implementation level details, i.e. though type implementors may write different procedures for different cases, the programmer will not be kept aware of the three different procedures: he sees only an abstraction of an operational type.

VII. Computational completeness.

Though any programming language is Computational completeness a query language (or DML or a data sub-language) not necessarily so. Computational completeness denotes the ability to express any computable function using the DML of the Database management System (DBMS), such completeness of DML is a new feature as far as conventional database systems are concerned. For example, SQL is not computationally complete. In an object-oriented database system it is highly that the languages offered be computationally complete. This can be achieved by providing connections with existing programming languages known to be computationally complete. Resource completeness is an essential requirement for object-oriented database systems, i.e. any resource screen or remote communication is essential to be completely available for accessing the object-oriented DBMS. This becomes even more important from the angle of multimedia object data management support.

(ix) Extensibility

The set of types supported by an OODBMS must be extensible, i.e. there must be a provision to provide or add new types with no distinction maintained between system-defined and user-defined types though the implementation methodology may be different.

(x) Persistence.

As is self-evident, any DBMS, whether object-oriented or not, must support persistent data, i.e. the ability of the data to survive the execution of a process, so that it can be reused in another process. In an OODBMS, each object, independent of its type, should be allowed to become persistent without the necessity of explicit translation. Persistence should be implicit, without making it necessary for the programmer to make or copy data in a persistent state.

(xi) Secondary storage management.

Any DBMS, object-oriented or not, will have to support secondary storage management which includes index management, clustering and buffering data, access path selection, and query optimisation with access path selection. All these will be invisible to the user and the application programmer. All this is well understood and is usually provided in all DBMSs under the feature of data independence.

(xi) Concurrency control and recovery.

Any DBMS will have to support concurrency control of contending transactions and recovery of aborted transactions. These features of physical implementation cannot be disconnected by data model improvements (as in the case of a relational to an object-oriented data model). The system should make it invisible to all users and programmers of the mechanisms of concurrency control and recovery. To support atomicity of sequence of

operations and controlled sharing, various mechanisms are now available, which include the traditional approaches involving serialisation and locking and novel approaches without locking.

The Optional Features

The optional features provide greater functionality and performance than the basic minimum core mandatory requirements listed in the previous sections. Some of these features bring in greater object orientation and some others are regular DBMS design-transaction management features which improve performance functionality and are unrelated to object orientation. These features are targeted at new application domains on CAD/CAM, CASE tools, and office automation.

(i) Multiple inheritance.

Inheritance from multiple parents, i.e. multiple inherit inheritance, is an optional feature of an OODBMS and it leads to problems conflict resolution which have several possible solutions.

(ii). Type checking and type inferencing.

As in programming languages, checking at compile time for the object-oriented databases system is preferrable to minimise run time type errors. Type inferencing is also preferable . Ideally only base types are to be declared and the system should infer the temporary types.

(iii). Distributed database.

Distribution is a physical feature independent of object orientation. Thus, the OODBMS should support databases for both centralised and distributed applications.

(iv) Design transactions.

OODBMS should support design transaction/long as against the conventional transaction model of business applications, which is inadequate for the new range of applications of object orientation.

(v) Versions.

Since design environments involve versioning, it is preferable that OODBMS support for this be provided.

========== End unit 4 ==========

Unit 4: Object Oriented Database Systems **LH 3**

- Performance Issues in OODBMS
- Application Selection for OODBMS
- The Object Oriented Database Paradigm Manifesto
- The Mandatory Features ➢ The Optional Features