Distributed Database

Munawar, PhD

DISTRIBUTED OBJECT

Why Object DBMS

Some applications require

- storage and management of abstract data types (e.g., images, design documents) ➔ rich type system supporting user-defined abstract types;
- need to explicitly represent composite and complex objects without mapping to flat relational model;
- need more powerful languages without the impedance mismatch.

Fundamental Concepts (1)

- Object
  - An entity in the system that is being modeled.
  - <OID, state, interface>
- OID: object identifier
  - Immutable
- State
  - Atomic or constructed value
  - Atomic values are instance variables (or attributes)
  - Constructed values can be set or tuple
- Interface
  - State and behaviour
  - Behavior captured by methods
- Object states may change, but OID remains identical

Fundamental Concepts (2)

- Composition (aggregation)
  - Composite type (Car) and composite object
  - Allows referential sharing – objects refer to each other by their OIDs as values of object-based variables
  - Composition relationships can be represented by composition (aggregation) graph
- Subclassing and inheritance
  - Subclassing is based on specialization: class A is a specialization of class B if A's interface is a superset of B's interface.
  - Inheritance: result of subclassing – class A's properties consist of what is defined for it as well as the properties of class B that it inherits

Object Distribution

- New problems due to encapsulation of methods together with object state.
- Fragmentation can be based on
  - State
  - Method definitions
  - Method implementation
- Class extent can be fragmented

Fragmentation Alternatives

- Horizontal
  - Primary
  - Derived
  - Associated
- Vertical
- Hybrid
- Path partitioning

Horizontal Fragmentation

- Primary
  - Defined similar to the relational case
- Derived
  - Due to the fragmentation of a subclass
  - Due to fragmentation of a complex attribute
  - Due to method invocation

Vertical Fragmentation

- For a class *C*, fragmenting it vertically into C_1, ..., C_m produces a number of classes, each of which contains some of the attributes and some of the methods.
  - Each fragment is less defined than the original class
- Issues
  - Subtyping relationship between *C*'s superclasses and subclasses and the fragment classes
  - Relationship of the fragment classes among themselves
  - Location of the methods when they are not simple methods

Path Partitioning

- Clustering all of the objects forming a composite object into a partition
- Can be represented as a hierarchy of nodes forming a structural index
  - Each node of the index points to objects of the domain class of the component object

Cache Consistency

- Avoidance-based
  - Prevents access to stale cache data by ensuring that clients cannot update an object if it is being read by other clients
    - Object in cache is stale if it has already been updated and committed to the database by a different client
  - Stale data cannot exist in the cache
- Detection-based
  - Detect stale object access at a validation step at commit time
  - Stale data is allowed to exist in the cache
- Each can further classified based on when the client informs the server about writes
  - Synchronous
  - Asynchronous
  - Deferred

Alternative Cache Consistency Algorithms

- Avoidance-based synchronous
- Avoidance-based asynchronous
- Avoidance-based deferred
- Detection-based synchronous
- Detection-based asynchronous
- Detection-based deferred

Object Identifier Management

- Physical object identifier (POID)
  - OID is equated with the physical address of the corresponding object
  - Address can be disk page address and an offset from the base address
  - \+ Object can be obtained directly from the OID
  - \- Parent object and all indexes need to be updated when object moves
- Logical identifier (LOID)
  - System-wide unique
  - A mapping has to occur to map it to the physical address
  - \+ Object can be easily moved
  - \- Indirection overhead

Thank You !

Munawar, PhD