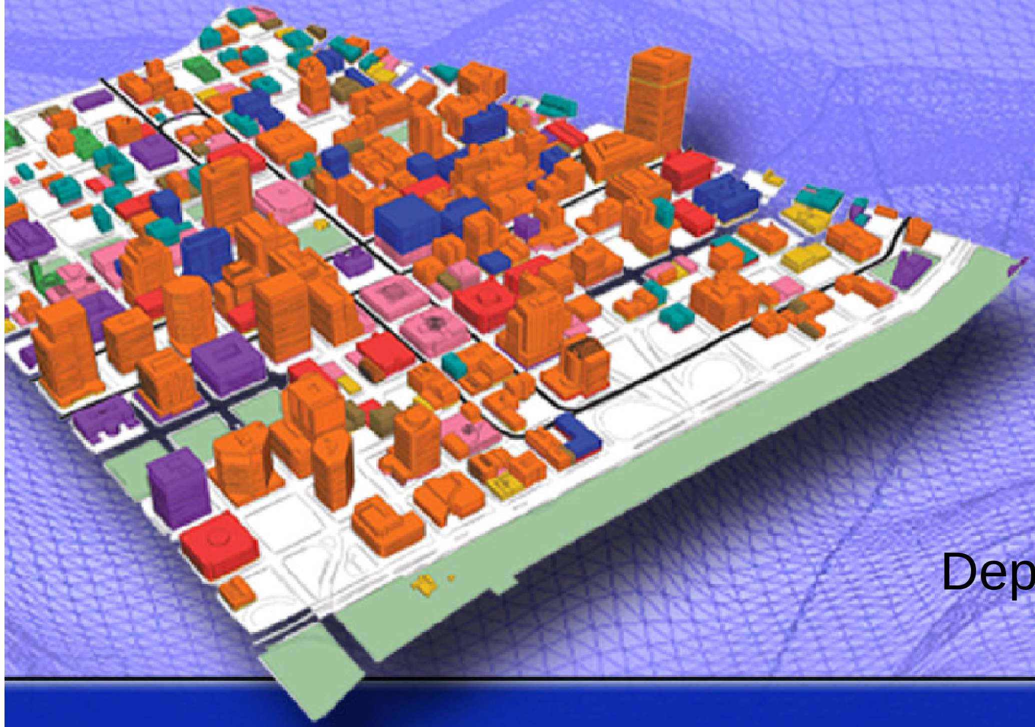
Complex Object Management in Databases: About the Preparedness of Database Technology for New Emerging Applications



Markus Schneider

University of Florida

Department of Computer & Information Science &Engineering

Outline

Research described in this talk is funded by

Outline

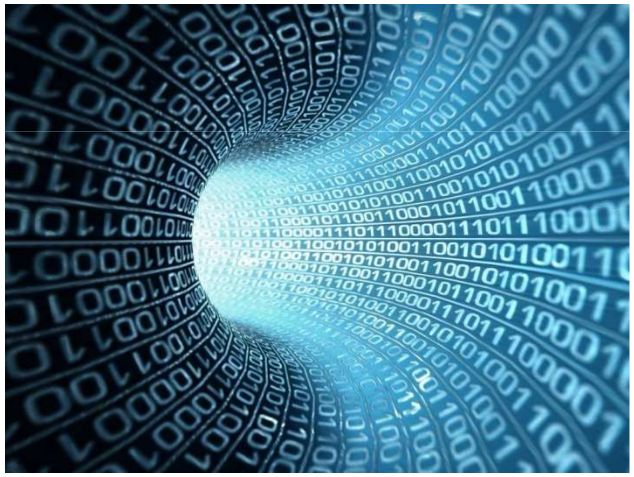
Big Data (I)

- Quantity and nature of data has changed over the years
- Nature of the data at the beginning of database technology
  - Alphanumerical data
  - Simple structure of data
  - Manageable volumes of data
- Nature of the data nowadays: “big data”
  - Very large volumes of simple alphanumerical data
  - Diverse data
  - Distributed data
  - Single, large complex application objects
- “Big Data” is a loosely defined term and refers to large, diverse, complex, and/or distributed data sets that are difficult to capture, store, manage, query, and analyze with conventional database management tools.

Big Data (II)

- Very large volumes of simple alphanumerical data are generated as raw data from sensing devices like instruments, sensors, satellites, mobile devices, cameras, microphones, radio-frequency identification readers, etc. and usually have a simple internal structure.

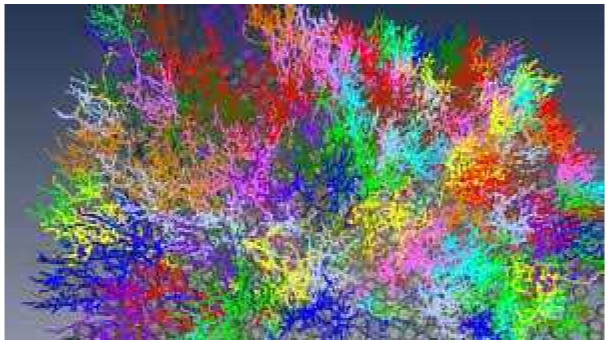
- Problem: Many of these data stay stored as raw data although they describe complex objects (like hurricanes, maps, DNA structures), that is, the composition step of raw data to complex data is often missing.

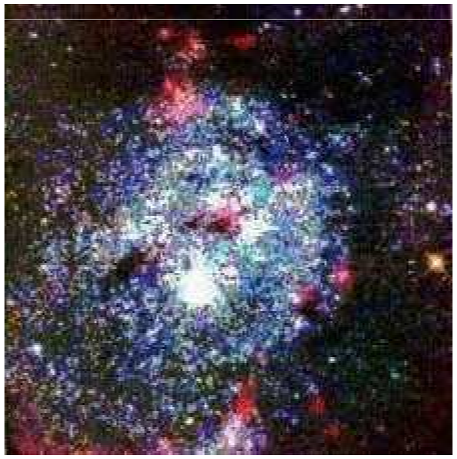


- Diverse data such as text, geometry, images, video, or sound have rather different properties and operations and lead to increased difficulties of big data processing.

Big Data (III)

- Single large, complex application objects are characteristic for new emerging (that is, non-traditional) applications including biological, genomic, meteorological, multimedia, web, digital library, imaging, scientific, location-based, geospatial, and spatiotemporal applications.
- Examples

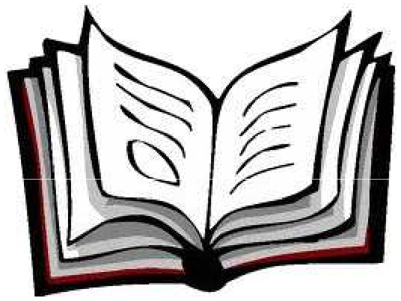

The brain's complex network of 70 billion neurons and thousands of kilometers of circuits

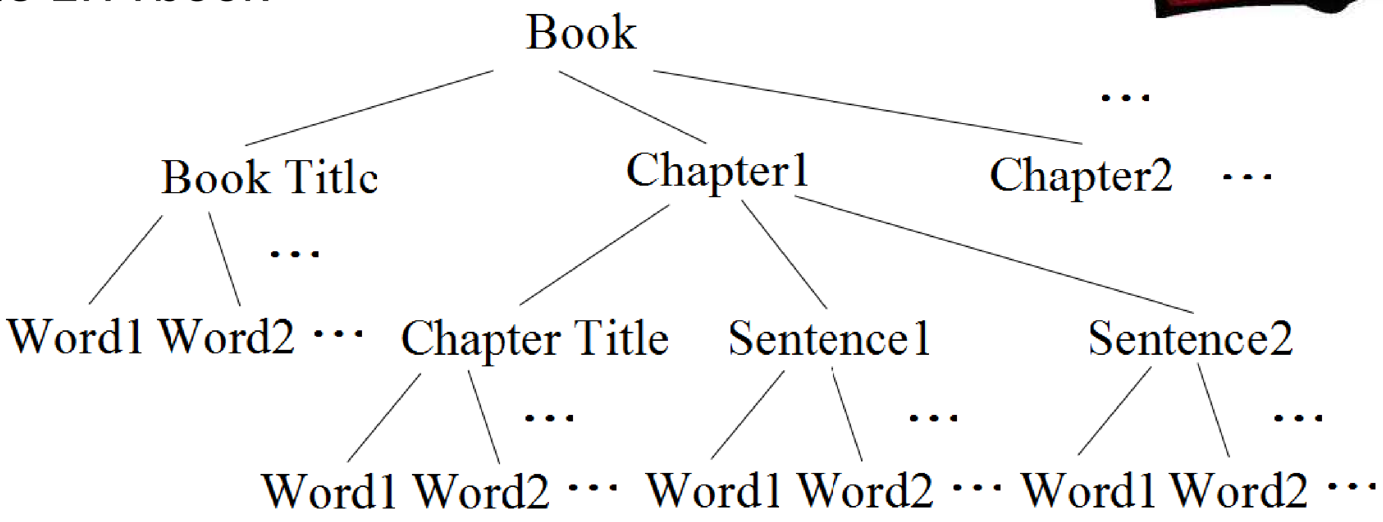

The Bubble complex (the Hodge object), a star cluster, in NGC 6946

A street map

What are Complex (Application) Objects? (I)

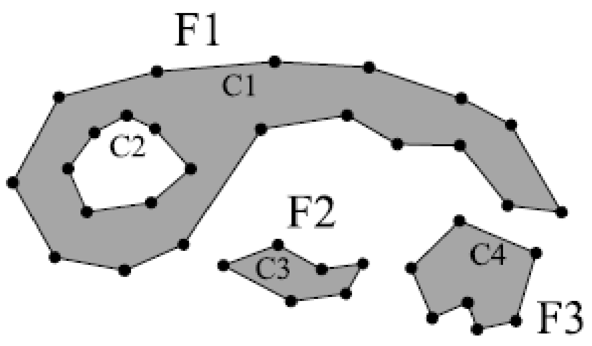
- Properties
  - highly structured
  - large in size (but can also be small)
  - of variable representation length
- Domain-specific application knowledge required
- Example 1: A book

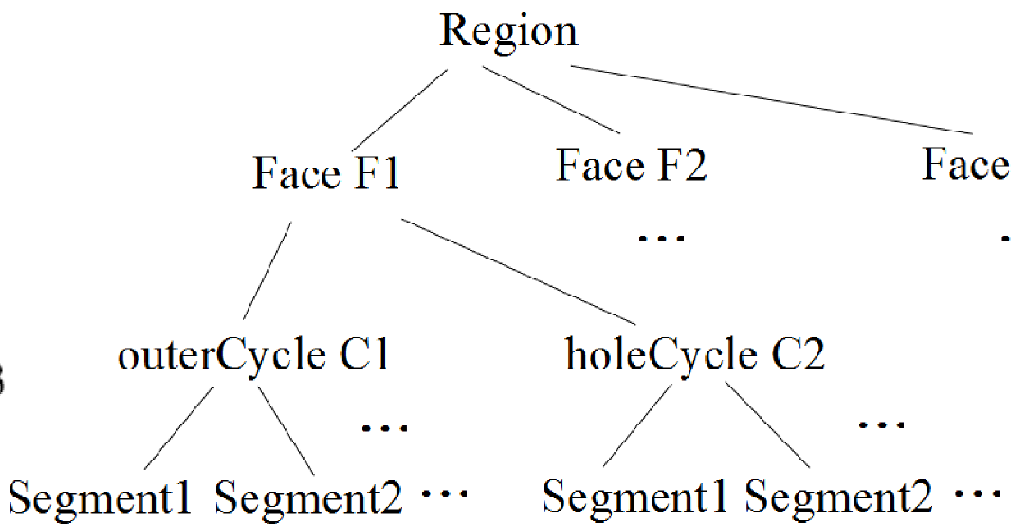




What are Complex (Application) Objects? (II)

❖ Example 2: A complex spatial region object





Focus of this Talk

- ❖ Main topic: Database management for complex application objects
- ❖ Subtopics of this talk
  - ➢ What are the current solutions for complex object management?
  - ➢ Which strengths and weaknesses do they reveal?
  - ➢ What are the requirements of complex object management in databases?
  - ➢ How can we store, retrieve, and manage complex, highly-structured, variable-length, and large-sized application objects in a database?
  - ➢ How can we support efficient insertions and updates of their components?
  - ➢ Novel two-step approach: Intelligent Binary Large Objects (iBLOBs) and Type Structure Specification (TSS)

Outline

General Problems of Current Approaches

- Abstraction problem: no “real” ADTs in databases, restricted high-level modeling of complex application objects
- Data management problem: specialized file format solutions lack any well established DBMS features
- Generality problem: non-uniform concepts and mechanisms for supporting complex objects, lack of portability
- Update problem: no support for random updates
- Acceptance problem: useful concepts of DBMS research prototypes cannot be easily transferred to commercial DBMS

Abstraction Problem (I)

Relational features in the SQL standard

❖ No complex object support

❖ Flat tables only, atomic data types only, no type constructors available

❖ Internals of a single complex object are spread over several tables

Point	<u>PId</u>	X	Y
	1	5	5
	2	6	7
	3	4	6

Segment	<u>SId</u>	SPId	EPId
	1	1	2
	2	2	3
	3	1	3

Region	<u>RId</u>	<u>SId</u>
	1	1
	1	2
	1	3

❖ Expensive joins required to bring object information together

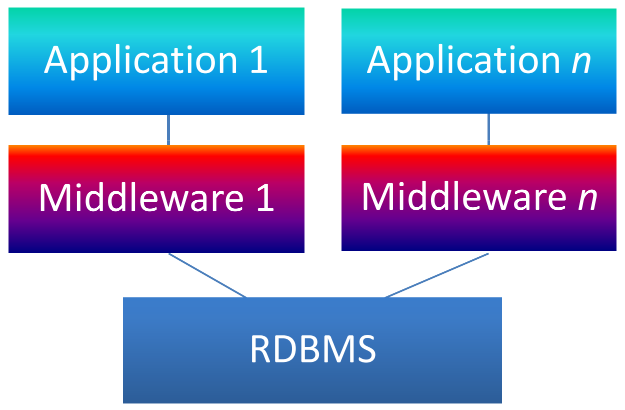
❖ But still no object in our hands

❖ No domain-specific operations can be defined since they cannot be expressed by the Relational Algebra and by SQL

Abstraction Problem (II)

Relational features in the SQL standard (*continued*)

❖ Domain-specific data structures and operations have to be implemented in a middleware layer on top of the DBMS



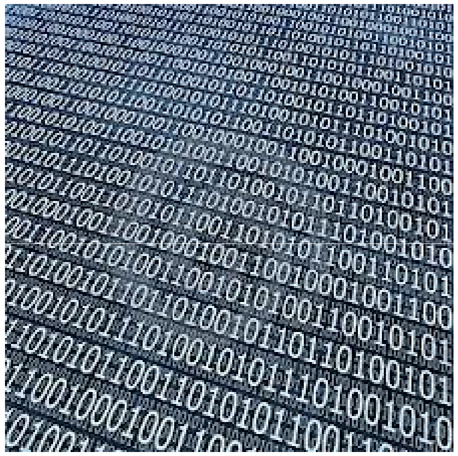
❖ Database as a repository for primitive data

❖ Consequences: Loss of important DBMS services like query processing, concurrency control, transactions, recovery, backup, …

Abstraction Problem (III)

Relational features in the SQL standard (*continued*)

❖ Advantage: Large values can be stored in BLOBs (Binary Large Objects)



❖ Shortcomings of BLOBs
- ➢ Provide and process byte sequences, no structure preservation of objects
- ➢ Offer low-level interface for simple read/write access to byte ranges
- ➢ Provide no high-level view of complex objects and their components
- ➢ No methods to access internal components of complex objects

Abstraction Problem (IV)

Specialized file formats

- ❖ Reaction to the lack of support for complex objects in relational databases by scientists
- ❖ Examples: HDF, NetCDF, XML
- ❖ File formats allow one to represent hierarchical and multidimensional data
- ❖ Problems:
  - ➢ HDF and NetCDF are binary and compressed data formats
  - ➢ XML is a textual format, produces large data volumes, and does not support data encapsulation (information hiding)
  - ➢ File formats do not support efficient insertions and updates
  - ➢ No support of standard DBMS functions like transaction management, concurrency control, and recovery
  - ➢ HDF and NetCDF have no support for querying (no query language)

Abstraction Problem (V)

Object-relational features in the SQL standard

- Object-oriented features allow one to specify user-defined types (UDT) and user-defined functions (UDF) (*create type* …)
- But object-oriented concepts are insufficient for defining complex objects
  - They reveal the structure and representation of data, all internal attributes of a UDT are public, no encapsulation of the internal implementation of a type, no ADTs but pseudo ADTs
  - Not usable for proprietary data (no information hiding)
  - Fixed-length types can only be defined, by using the built-in types and type constructors of the object-oriented features of SQL
  - SQL is not a programming language: complex operations cannot be expressed in SQL
- Vendor solutions: extension packages like cartridges, data blades, and extenders with predefined functionality, difficult to create by a user

Abstraction Problem (VI)

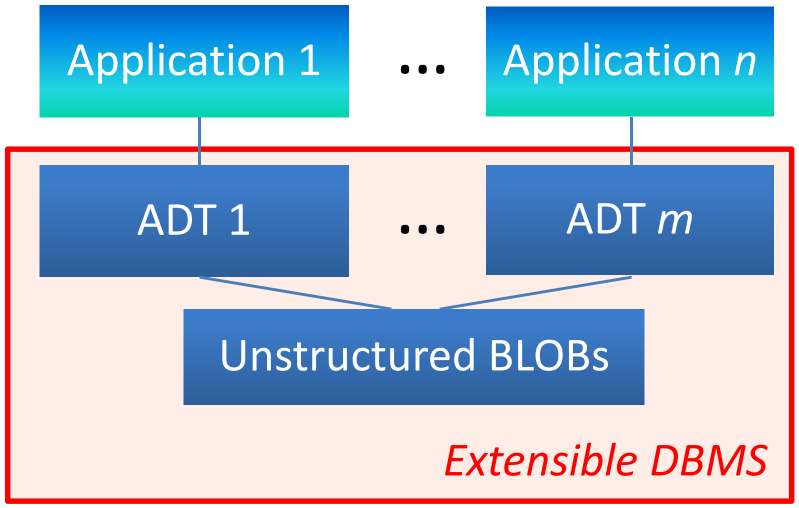
Object-relational features in the SQL standard (*continued*)

❖ High-level ADTs are registered in the DBMS, and their objects are stored in low-level BLOBs

❖ No structure preservation of application objects in BLOBs

❖ Byte level operations in BLOBs complicate the implementation of high-level component retrieval and update operations in ADTs

❖ For component access, whole BLOB must be loaded, or incremental load



Data Management Problem

- Special file formats like NetCDF and HDF5 store objects in files
- Advantage: Hierarchical and multi-dimensional data can be stored
- Problems
  - No standard DBMS functionality available like
    - Transaction management
    - Recovery, backup
    - Concurrency control
    - Data security
  - No query language available
  - Each “query” has to be formulated as an application program

Generality Problem

- BLOB implementations have different interfaces with different functionalities in different DBMS
- Consequences
  - Software portability across different DBMS limited
  - Non-uniformity of the BLOB interfaces makes it difficult for third-party developers to create database applications that are DBMS independent
  - Partial reimplementation of type systems for different DBMS needed
- Software extension packages like cartridges, data blades, and extenders are DBMS specific
  - Lack of portability

Update Problem

- ❖ No literature about updates on complex application problems (high-level problem)
- ❖ BLOBs allow data to be
  - ➢ appended
  - ➢ truncated
  - ➢ modified through the overwriting of bytes
- ❖ BLOBs do not support general data
  - ➢ insertions
  - ➢ deletions

  User has to explicitly shift data

Acceptance Problem

- ❖ Useful concepts of DBMS research prototypes
  - ➢ are tailor-made for a certain problem area
  - ➢ assume and utilize a specialized infrastructure
  - ➢ cannot be easily transferred to commercial DBMS
- ❖ Consequences
  - ➢ Lack of adequate appreciation
  - ➢ User's question of whether the achievements can be used in their own (commercial or public domain) DBMS has to be negated
- ❖ On the other hand, scientists want
  - ➢ open-source solutions
  - ➢ the ability to recompile the entire software stack
  - ➢ to avoid high license fees for commercial DBMS software

Outline

1. Introduction and Motivation
2. Problems of Complex Object Management Approaches
3. Requirements of Complex Object Management in Databases
4. iBLOB: Complex Object Management in Databases through Intelligent Binary Large Objects
5. TSS: Representing the Structure of Complex Application Objects through Type Structure Specifications
6. Conclusions and Future Work

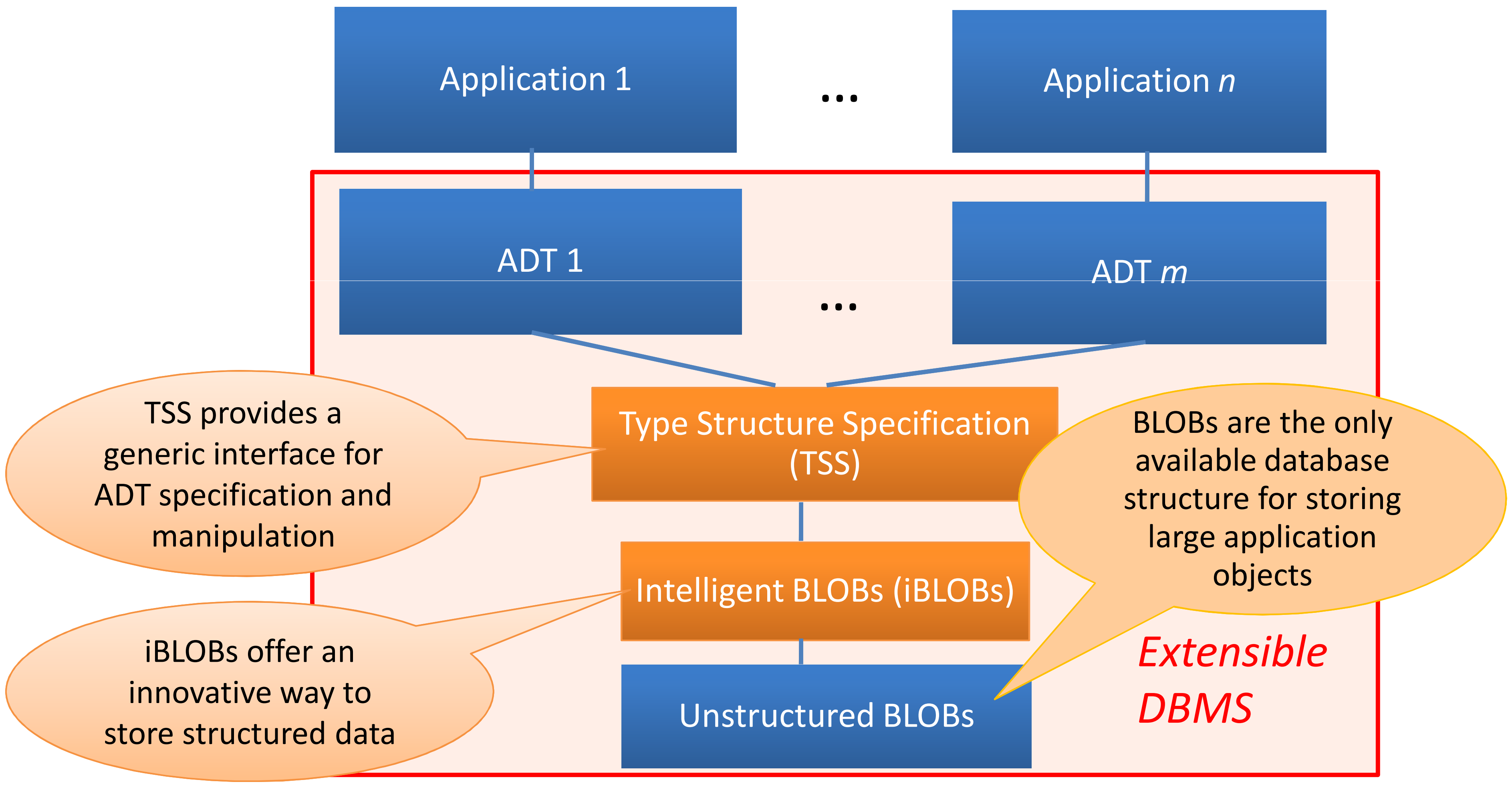
Requirements

1. User must be able to design, implement, and integrate own complex data types into databases
2. Design and implementation must be independent of the DBMS
3. Design of complex data types must be performed at a high abstraction level and not at a low byte level
4. Enable multi-structured data representations of the same complex object
5. Complex data types should be designed as proper abstract data types that hide implementation details for both data structures and algorithms
6. Complex objects should be arranged in compact storage structures (no main memory pointer structures) in order to avoid expensive serialization and deserialization cost
7. Stored binary data should correspond to the representation that algorithms can process to avoid data conversion
8. Efficient access to components of complex objects must be possible to avoid their complete loading into main memory
9. Updates (random insertions and deletions) should be possible

Outline

1. Introduction and Motivation
2. Problems of Complex Object Management Approaches
3. Requirements of Complex Object Management in Databases
4. iBLOB: Complex Object Management in Databases through Intelligent Binary Large Objects
5. TSS: Representing the Structure of Complex Application Objects through Type Structure Specifications
6. Conclusions and Future Work

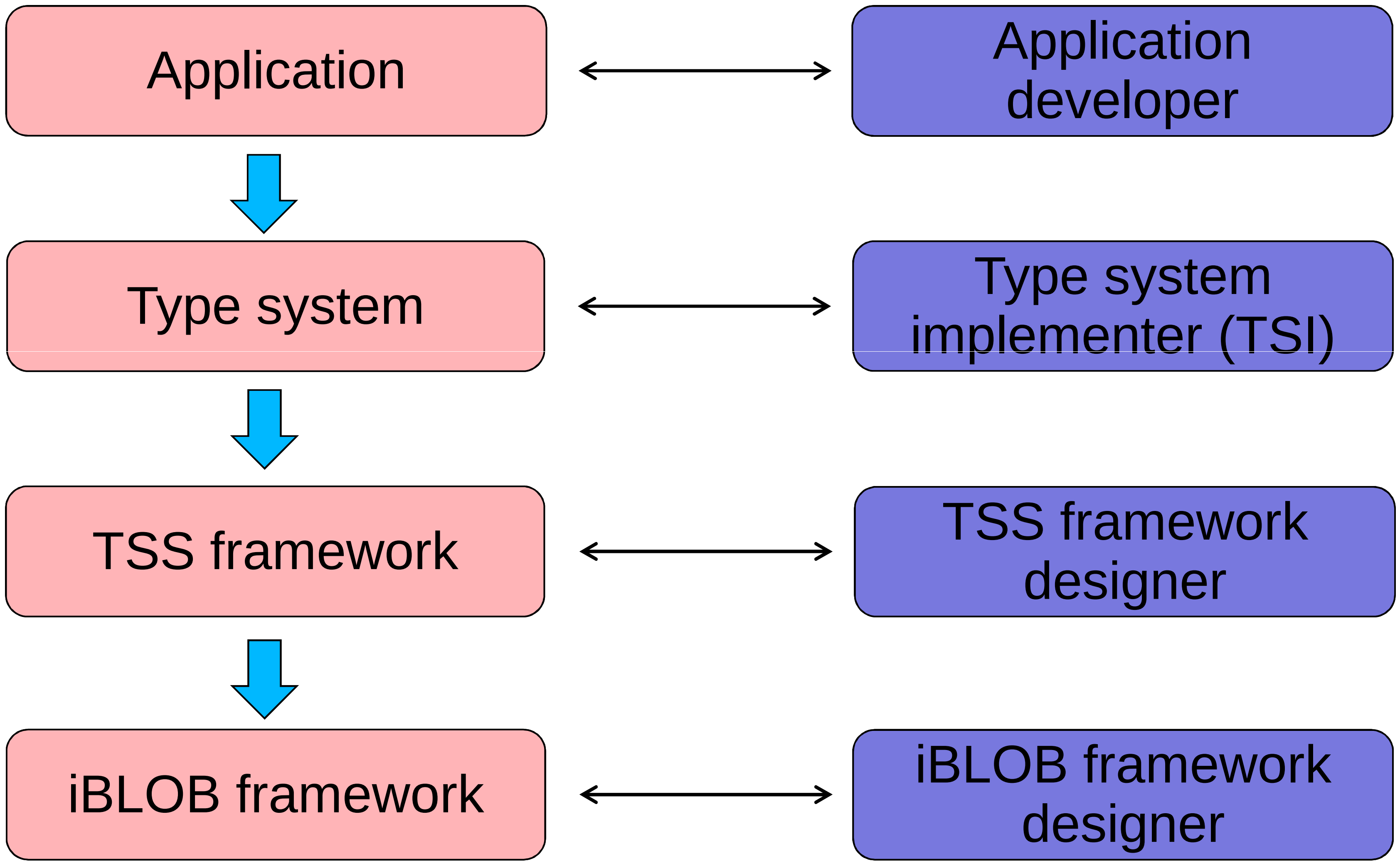
System Architecture


Application 1
…
Application *n*
ADT 1
…
ADT *m*
Type Structure Specification (TSS)
Intelligent BLOBs (iBLOBs)
Unstructured BLOBs
TSS provides a generic interface for ADT specification and manipulation
iBLOBs offer an innovative way to store structured data
BLOBs are the only available database structure for storing large application objects
Extensible DBMS

iBLOB Framework

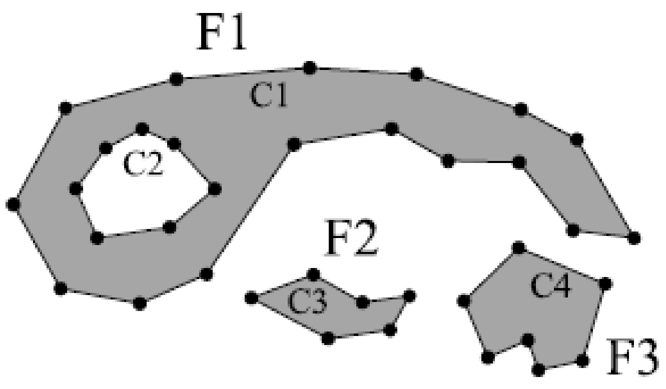
- ❖ Idea: smartly extend compact storage by preserving the complex application object structure internally and by providing application-friendly random access and update interfaces to object components
- ❖ Compact storage can, for example, be
  - ➢ traditional database BLOBs
  - ➢ file structures
  - ➢ array structures (persistent, main memory)
- ❖ iBLOBs (intelligent Binary Large Objects) are application neutral and do not understand the semantics of complex application objects
- ❖ iBLOB components to preserve structure in unstructured storage
  - ➢ Structure index: Preserving the physical structure of application objects in unstructured storage space
  - ➢ Sequence index: Preserving the logical structure of application objects, especially for updates

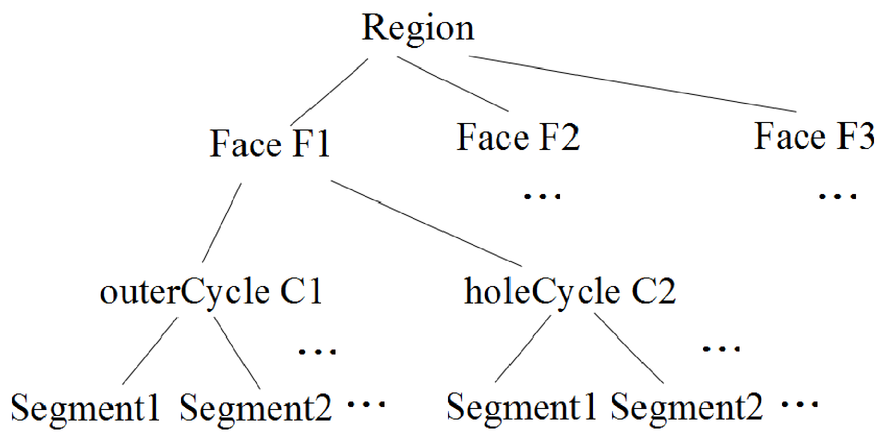
Components and their Designers



Structure Index (I)

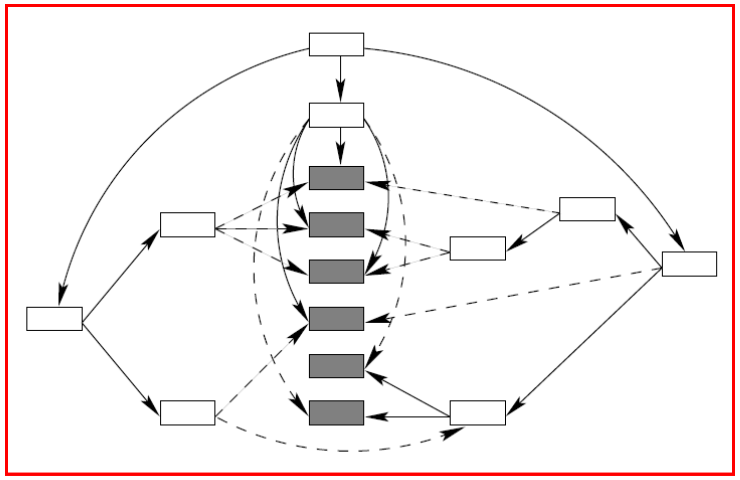
- An iBLOB is defined as a hierarchy (even as a graph) with three kinds of *anonymous* component sub-objects
  - Base objects: *leaf nodes* representing indivisible base components
  - Structure objects: *internal nodes* showing levels with additional structure objects and/or base objects
  - Reference objects: *pointer nodes* linking to other base or structure objects and supporting the construction of secondary structures for an application object

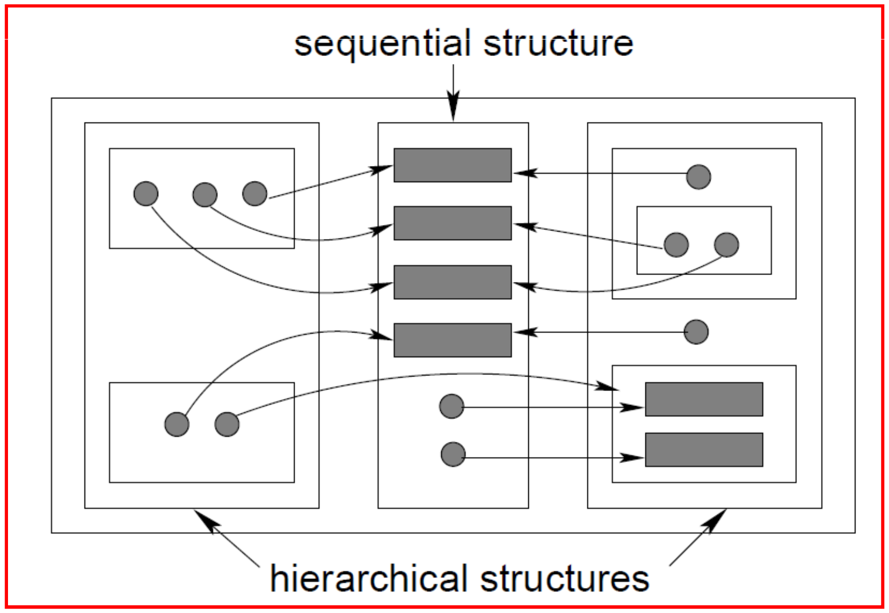




Structure Index (II)

- Schematic representation of a multi-structured iBLOB
  - Primary and secondary structures providing different views of a complex application object
  - Conceptual user view of a complex object does often not coincide with the representation (data structure) view of a complex object

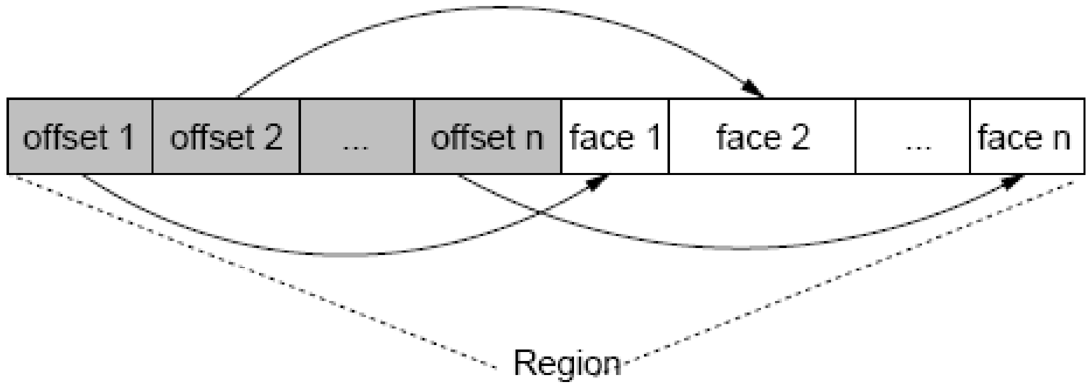




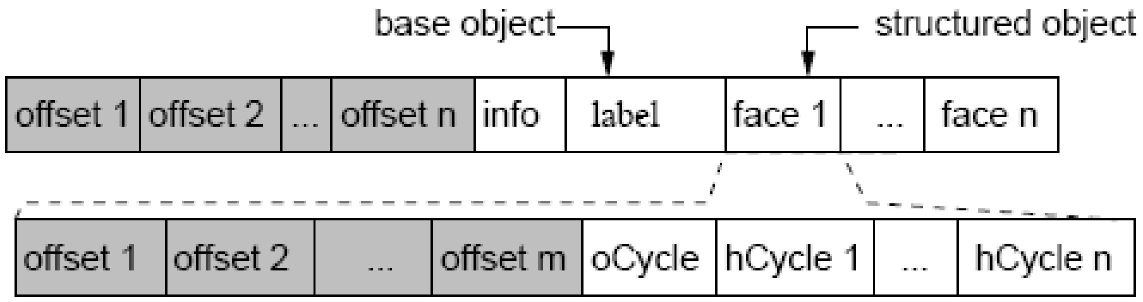
Shaded rectangles	:=	base objects
Un-shaded rectangles	:=	structured objects
Circles with arrows	:=	reference objects

Structure Index (III)

❖ The structure index consists of links to components (represented by offsets) and the actual components

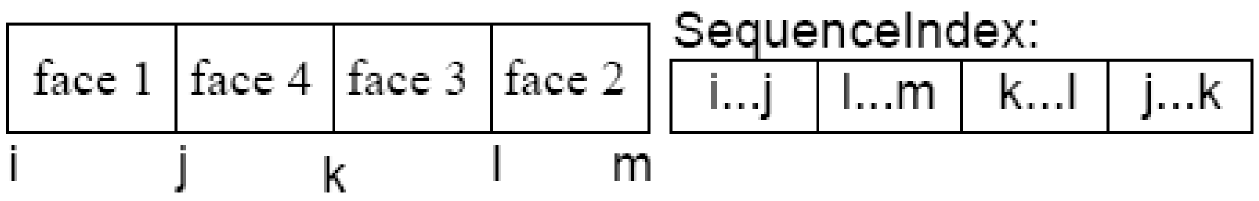


- Three types of components: base objects, structured objects, and reference objects



Sequence Index (I)

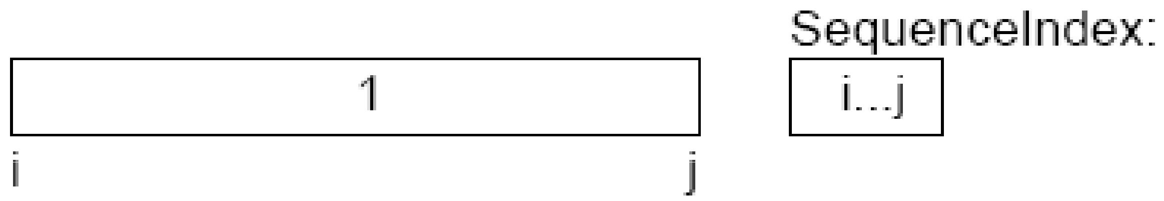
- Our solution to the problem of updating sub-objects of complex application objects
  - Physically store new components or updated parts of existing components *at the end* of the structure index (append operation is fast on BLOBs)
  - Use the sequence index to preserve the logically correct order
  - All operations on the iBLOB get routed through the sequence index for retrieving the exact physical byte address to operate on



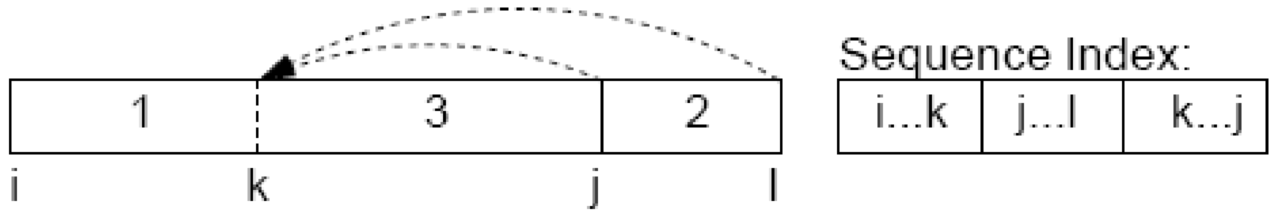
Sequence Index (II)

❖ Operation *insert*

Initial in-order and defragmented data in the structure index, and the sequence index



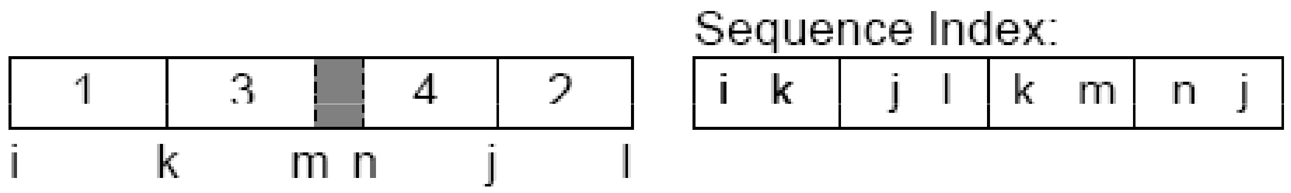
Insertion of block [j…l] at position k



Sequence Index (III)

❖ Operation *delete*

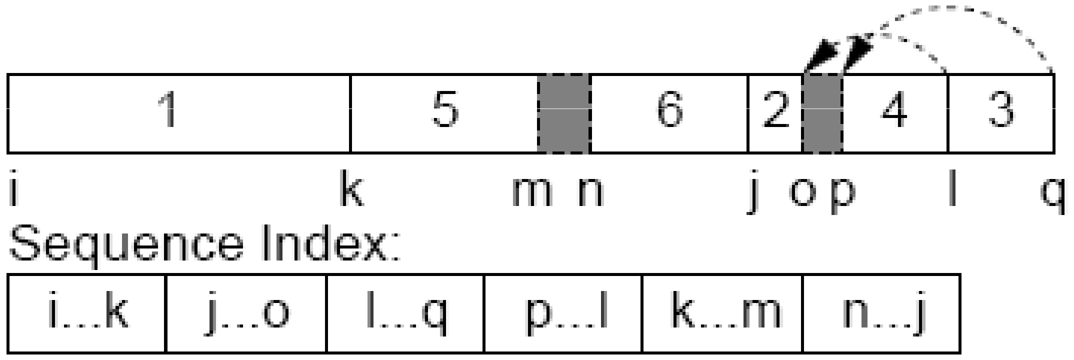
Deletion of block [m…n]



Sequence Index (IV)

❖ Operation *update*

Update of block [o…p] by block [l…q]



iBLOB Interface (I)

- Generic interface for constructing, retrieving, and manipulating iBLOBs

- Assumed data types
  - *Int* for representing integers
  - *Storage* as a storage structure handle type (e.g. BLOB handle, file descriptor, array address)
  - *Locator* as a reference type for referencing an iBLOB or any of its sub-objects by their physical address
  - *Stream* as an output channel for reading byte blocks of arbitrary size from an iBLOB object or any of its sub-objects
  - *data* as a representation of a base object

iBLOB Interface (II)

$$create : \rightarrow iBLOB \quad (1)$$

$$create : Storage \rightarrow iBLOB \quad (2)$$

$$create : iBLOB \rightarrow iBLOB \quad (3)$$

$$copy : iBLOB \times iBLOB \rightarrow iBLOB \quad (4)$$

$$locateiBLOB : iBLOB \rightarrow Locator \quad (5)$$

$$locate : iBLOB \times Locator \times Int \rightarrow Locator \quad (6)$$

$$getStream : iBLOB \times Locator \rightarrow Stream \quad (7)$$

$$insert : iBLOB \times data \times Int \times Locator \times Int \rightarrow iBLOB \quad (8)$$

$$insert : iBLOB \times iBLOB \times Locator \times Int \rightarrow iBLOB \quad (9)$$

$$remove : iBLOB \times Locator \times Int \rightarrow iBLOB \quad (10)$$

$$append : iBLOB \times data \times Int \times Locator \rightarrow iBLOB \quad (11)$$

$$append : iBLOB \times iBLOB \times Locator \rightarrow iBLOB \quad (12)$$

$$length : iBLOB \times Locator \rightarrow Int \quad (13)$$

$$count : iBLOB \times Locator \rightarrow Int \quad (14)$$

$$resequence : iBLOB \rightarrow iBLOB \quad (15)$$

iBLOB Interface (III)

- Construction and duplication
  - (1) *create*() creates an empty iBLOB
  - (2) *create*(*sh*), constructs an iBLOB from a storage structure handle *sh*
  - (3) *create*(*s*) is a copy constructor
  - (4) *copy*(s_1, s_2) copies an iBLOB object s_2 into another iBLOB object s_1

- Internal referencing
  - (5) *locateiBLOB*(*s*) yields a locator of the topmost hierarchical level of the iBLOB *s*
  - (6) *locate*(*s*, *l*, *i*) references a next level sub-object from locator *l* by its slot *i*

iBLOB Interface (IV)

❖ Read and write

- (7) *getStream*(*s*, *l*) consecutively reads arbitrary size data from the object referenced by locator *l* of iBLOB *s*
- (8) *insert*(*s*, *d*, *z*, *l*, *i*) inserts a base object *d* of size *z* into the object referenced by locator *l* at slot *i* of iBLOB *s*
- (9) *insert*(*s*, s_1, *l*, *i*) inserts an entire iBLOB s_1 into the object referenced by locator *l* at slot *i* of iBLOB *s*
- (10) *remove*(*s*, *l*, *i*) deletes the sub-object at slot *i* from the parent component with locator *l* of iBLOB *s*
- (11) *append*(*s*, *d*, *z*, *l*) appends a base object *d* of size *z* to the object referenced by locator *l* of iBLOB *s*
- (12) *append*(*s*, s_1, *l*) appends an iBLOB s_1 to the object referenced by locator *l* of iBLOB *s*

iBLOB Interface (V)

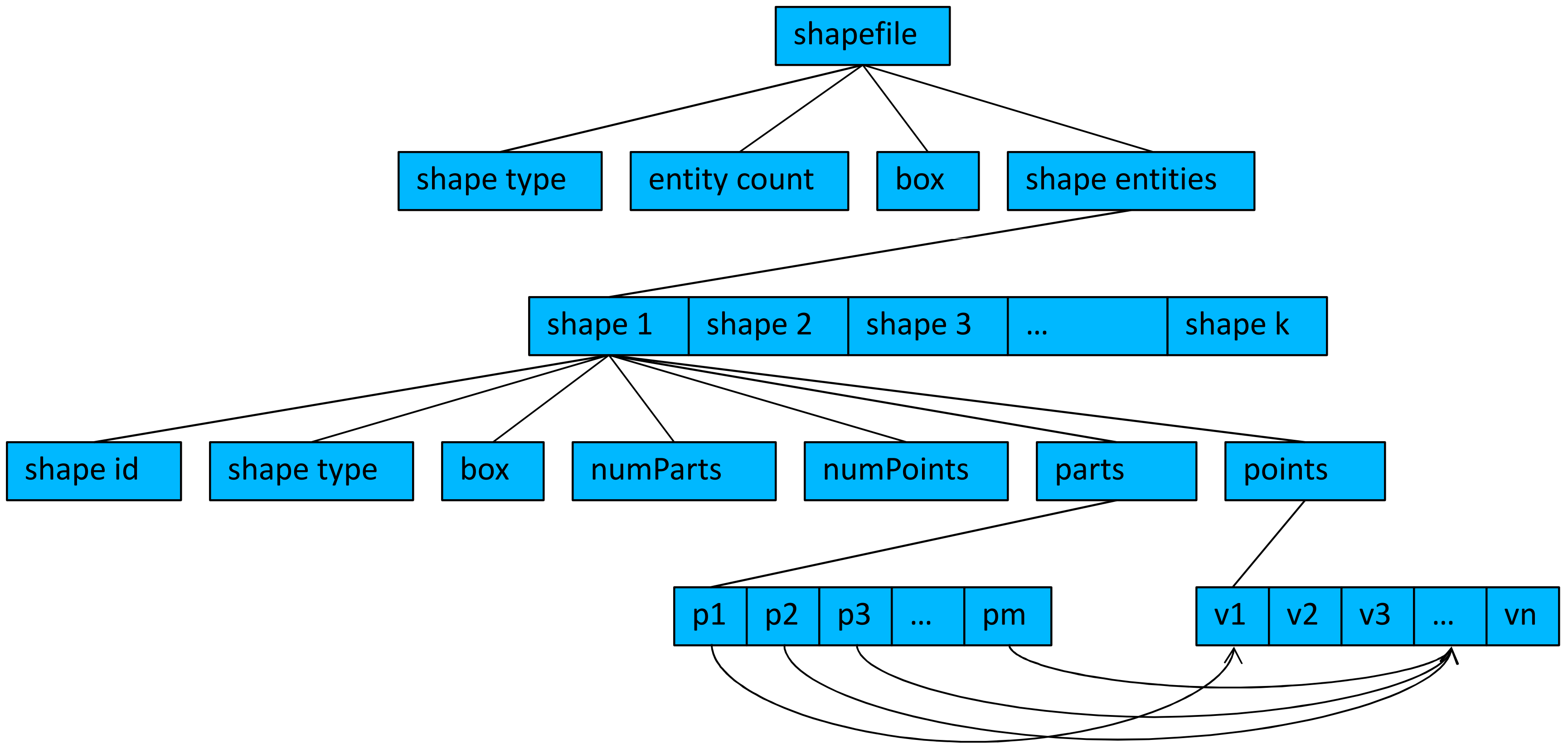
- Properties and Maintenance
  - (13) *length*(*s*, *l*) returns the actual size of the object referenced by locator *l* of iBLOB *s*
  - (14) *count*(*s*, *l*) returns the number of sub-objects of the object referenced by locator *l* of iBLOB *s*
  - (15) *resequence*(*s*) reorganizes and defragments the iBLOB *s*

iBLOB Evaluation – First Results (I)

- Goal: to compare different options and show that the iBLOB approach is efficient in storage, data access, and data manipulation
- Example application: *shapefile* (*.shp*)
  - popular geospatial vector data format for GIS software in a binary file
  - It contains geometric object descriptions like *points*, *polylines*, and *polygons* as wells as thematic data like name or temperature for geographic objects like wells, rivers, or lakes.
- Evaluation setup
  - Test cases: the 56 TIGER 2010 county maps for USA
  - File sizes: from 9KB to 9M, number of vertices: from 551 to 604747
  - Database environment: Oracle11g
  - Approaches to compare: BLOB, iBLOB, and XMLType
  - Operations to compare: *create*, *single read*, *single insert*, *single delete*

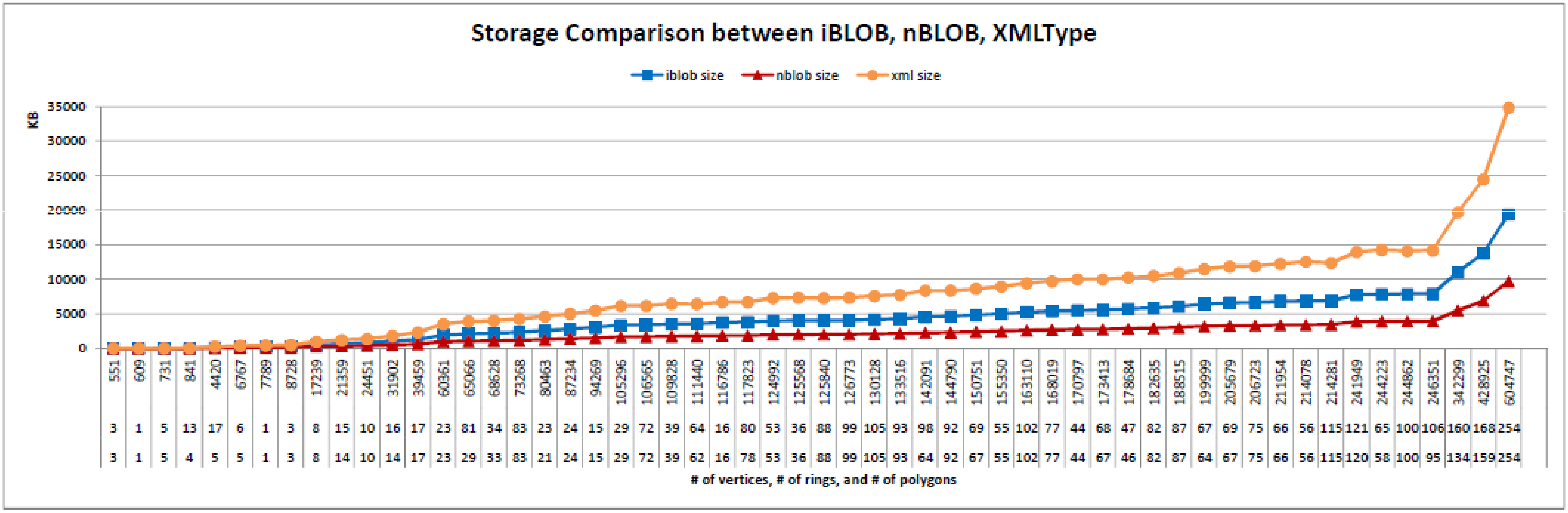
iBLOB Evaluation – First Results (II)

❖ Structure of a polygon shapefile hierarchy



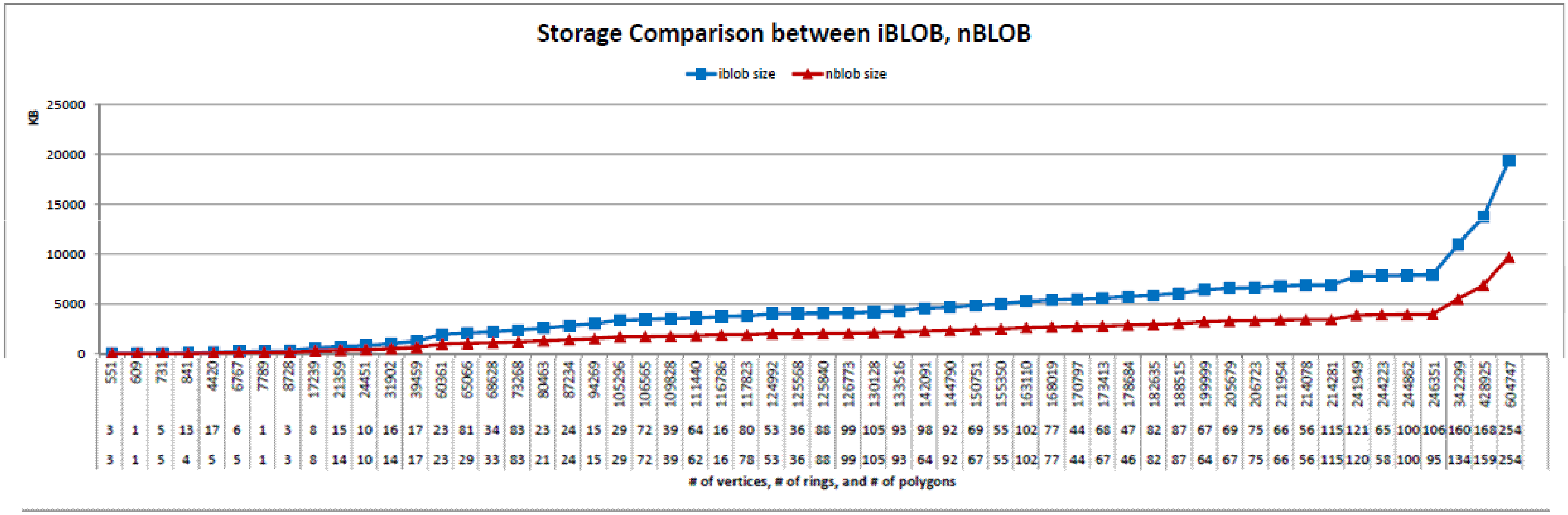
iBLOB Evaluation – First Results (III)

❖ Storage comparison between iBLOB, nBLOB, and XMLType



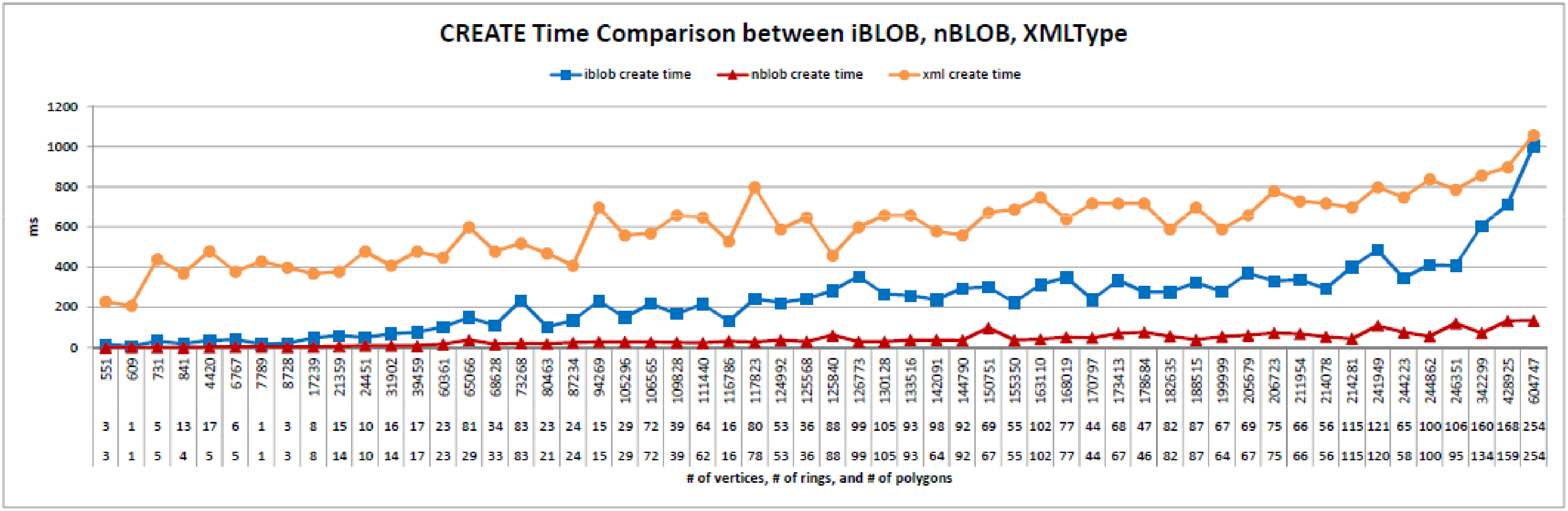
iBLOB Evaluation – First Results (IV)

❖ Storage comparison between iBLOB and nBLOB



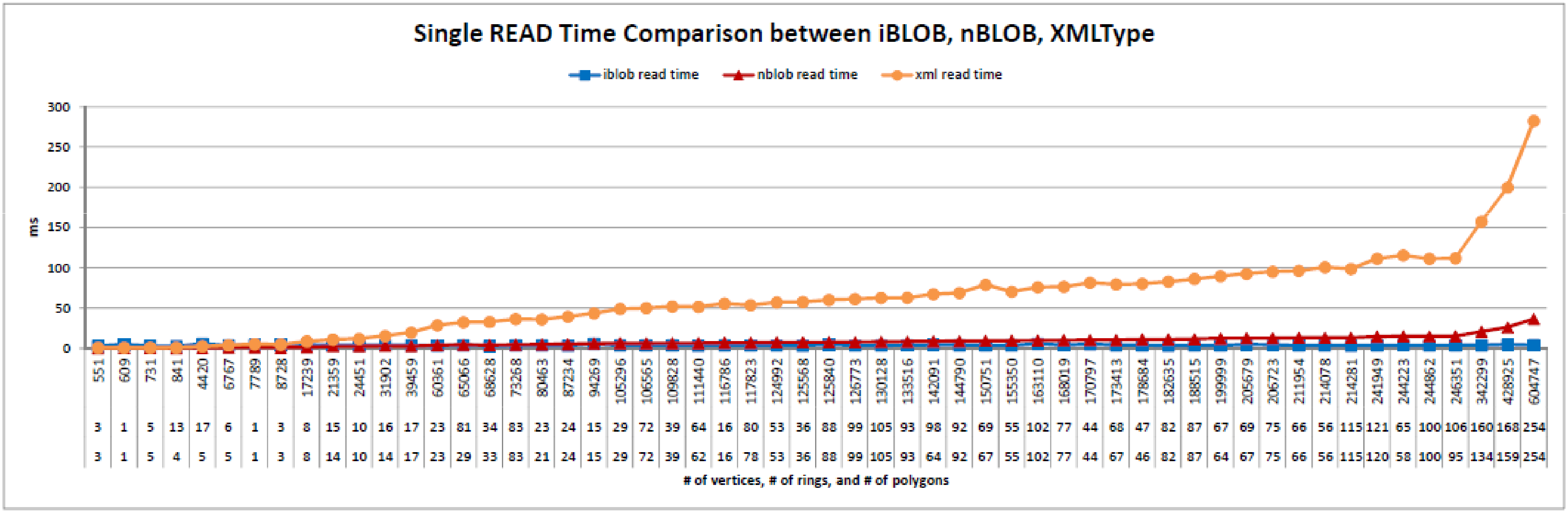
iBLOB Evaluation – First Results (V)

- *Creation* time



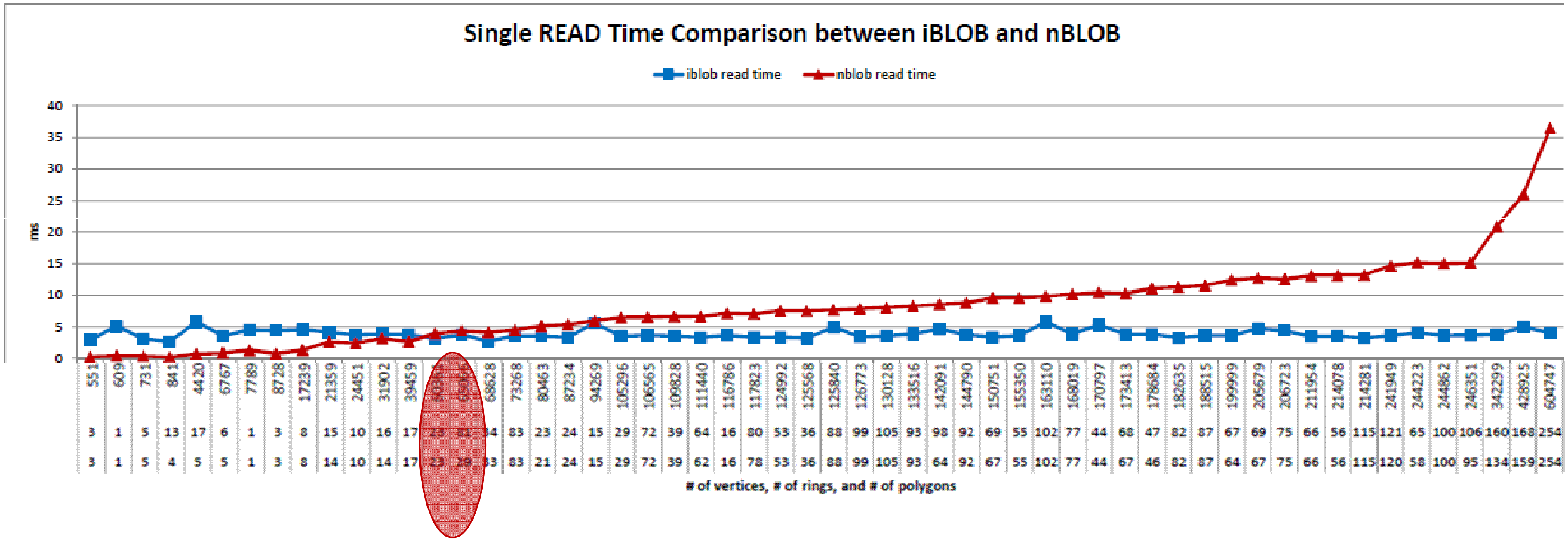
iBLOB Evaluation – First Results (VI)

- Single *read* time between iBLOB, nBLOB, and XMLType



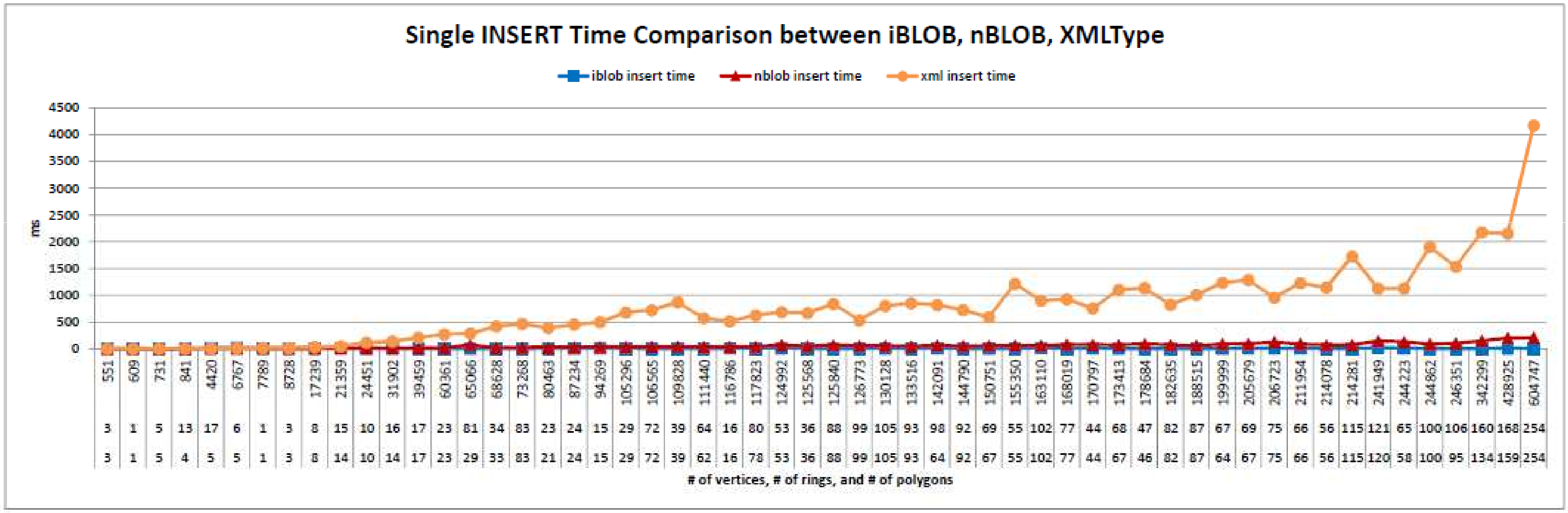
iBLOB Evaluation – First Results (VII)

- Single *read* time between iBLOB and nBLOB



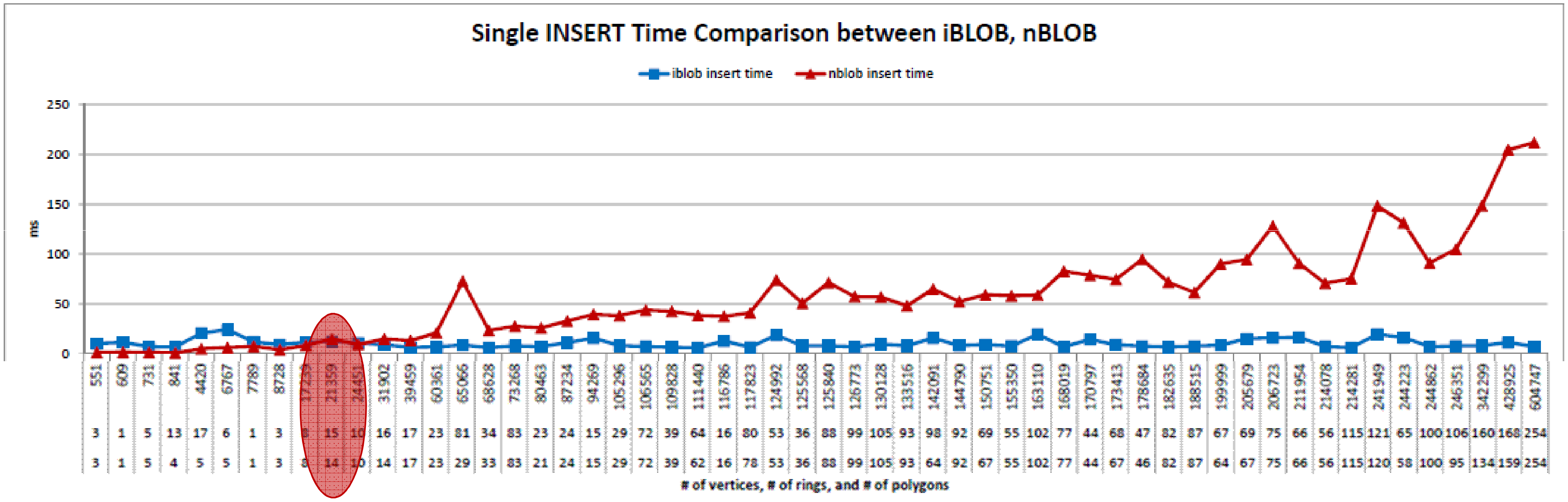
iBLOB Evaluation – First Results (VIII)

- Single *insert* time between iBLOB, nBLOB, and XMLType



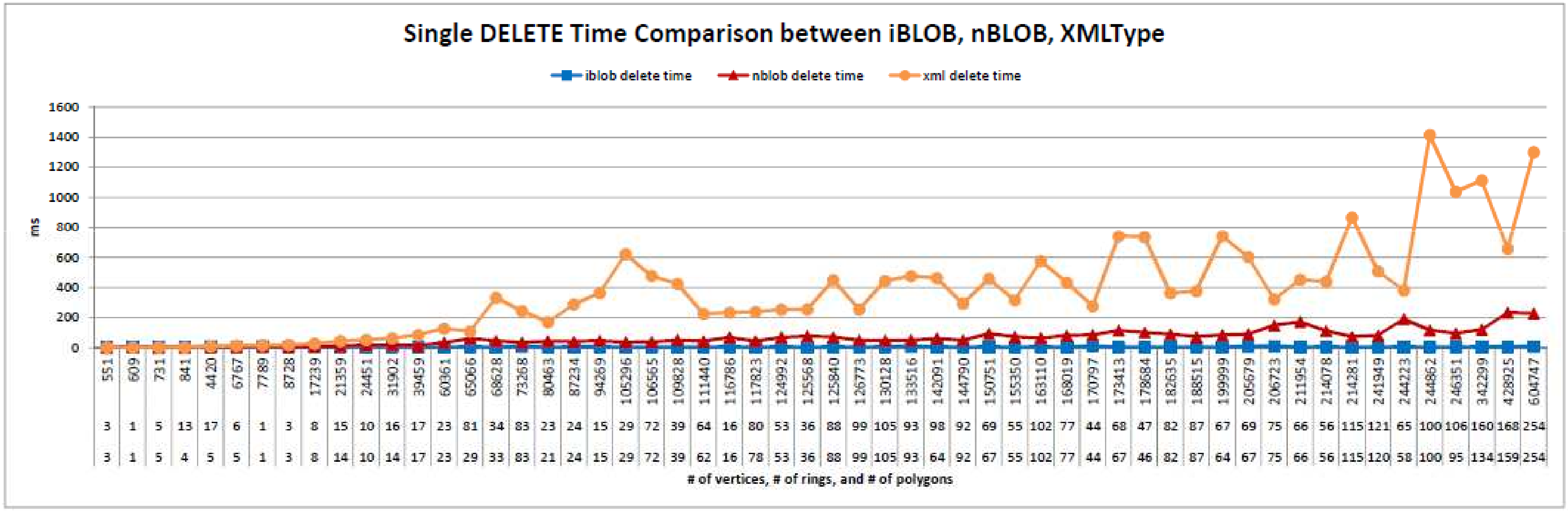
iBLOB Evaluation – First Results (IX)

- Single *insert* time between iBLOB and nBLOB



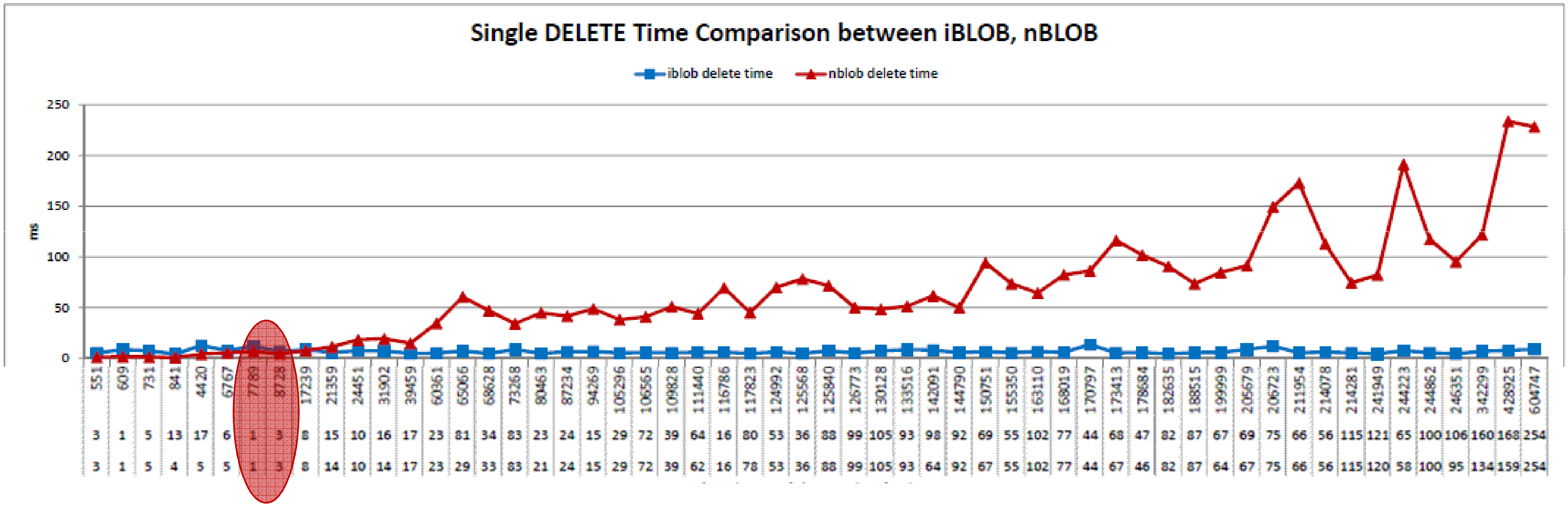
iBLOB Evaluation – First Results (X)

- Single *delete* time between iBLOB, nBLOB, and XMLType



iBLOB Evaluation – First Results (XI)

- Single *delete* time between iBLOB and nBLOB



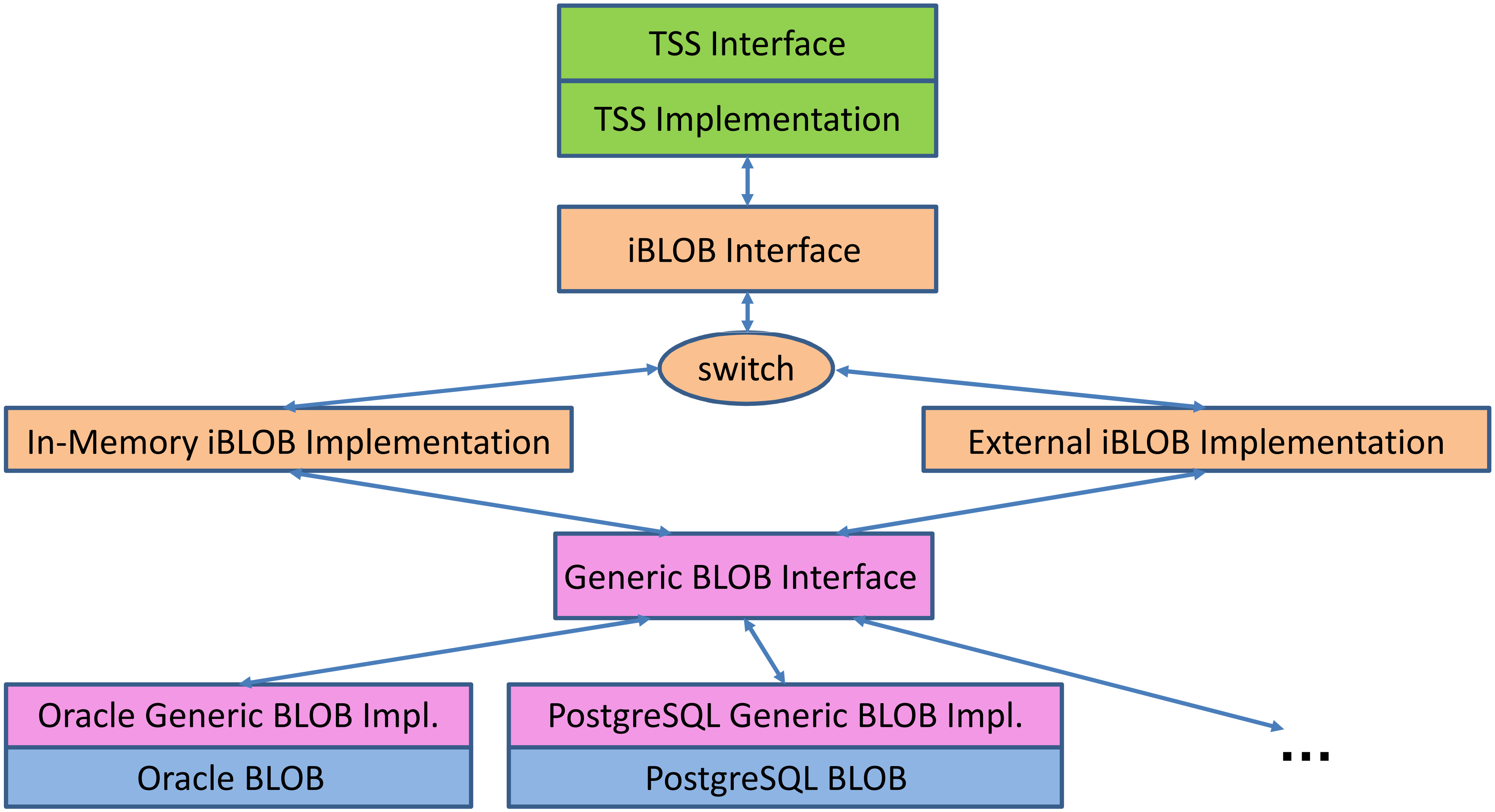
551	609	731	841	4420	6767	7789	8728	17239	21359	24451	31902	39459	60361
3	1	5	13	17	6	1	3	8	15	10	16	17	23
3	1	5	4	5	5	1	3	8	14	10	14	17	23

65066	68628	73268	80463	87234	94269	105296	106565	109828	111440	116786	117823	124992	125568
81	34	83	23	24	15	29	72	39	64	16	80	53	36
29	33	83	21	24	15	29	72	39	62	16	78	53	36

125840	126773	130128	133516	142091	144790	150751	155350	163110	168019	170797	173413	178684	182635
88	99	105	93	98	92	69	55	102	77	44	68	47	82
88	99	105	93	64	92	67	55	102	77	44	67	46	82

188515	199999	205679	206723	211954	214078	214281	241949	244223	244862	246351	342299	428925	604747
87	67	69	75	66	56	115	121	65	100	106	160	168	254
87	64	67	75	66	56	115	120	58	100	95	134	159	254

iBLOB Architecture


TSS Interface
TSS Implementation
iBLOB Interface
switch
In-Memory iBLOB Implementation
External iBLOB Implementation
Generic BLOB Interface
Oracle Generic BLOB Impl.
Oracle BLOB
PostgreSQL Generic BLOB Impl.
PostgreSQL BLOB
...

Requirements Checked (I)

- ? User must be able to design, implement, and integrate own complex data types into databases
- ✓ Design and implementation must be independent of the DBMS
- ? Design of complex data types must be performed at a high abstraction level and not at a low byte level
- ✓ Enable multi-structured data representations of the same complex object
- ? Complex data types should be designed as proper abstract data types that hide implementation details for both data structures and algorithms
- ✓ Complex objects should be arranged in compact storage structures (no main memory pointer structures) in order to avoid expensive serialization and deserialization cost
- ✓ Stored binary data should correspond to the representation that algorithms can process to avoid data conversion
- ✓ Efficient access to components of complex objects must be possible to avoid their complete loading into main memory
- ✓ Updates (random insertions and deletions) should be possible

Outline

Abstract Data Types in Databases

- ADTs used as attribute types in relational (or other) schemas
- Example:

  states(sname : string; spop : integer; sarea : region)
  hurricanes(hname : string; hzone : hmregion)

- Operations and predicates on ADTs embedded into an SQL extension
- intersects: region $\times$ region $\rightarrow$ bool
  intersection: region $\times$ region $\rightarrow$ region
  traversed: hmregion $\rightarrow$ region
- Example query: Find all pairs of states and hurricanes where the hurricane crossed the state, and determine the impacted area

```
SELECT sname, hname, intersection(sarea, traversed(hzone)) AS dzone
FROM states, hurricanes
WHERE sarea intersects traversed(hzone)
```

TSS Framework (I)

- Problems of the direct use of the iBLOB interface
  - Component-wise access possible but still at a low level
  - Semantics of complex object and sub-components not existent
- The TSS framework
  - is an abstract, type structure oriented component on top of the iBLOB component
  - represents the type structure of a complex data type in the form of a TSS grammar
  - internally processes the grammar and automatically creates data type objects which follow the specified grammar
  - provides a clean, user-friendly, and generic TSS interface for the TSI to access and manipulate complex application objects at a high semantic level on the basis of iBLOBs

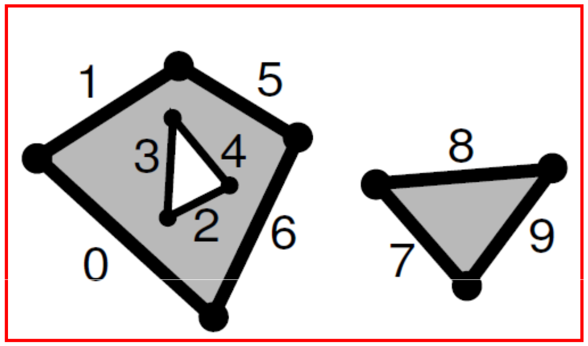
TSS Framework (II)

- A type structure is defined as a hierarchy with three kinds of nodes
  - Base object types: *leaf nodes* representing indivisible base components
  - Structure object types: *internal nodes* showing levels with additional structure objects and/or base objects
  - Reference object types: *pointer nodes* linking to other base or structure objects. These represent secondary structures in the application object.

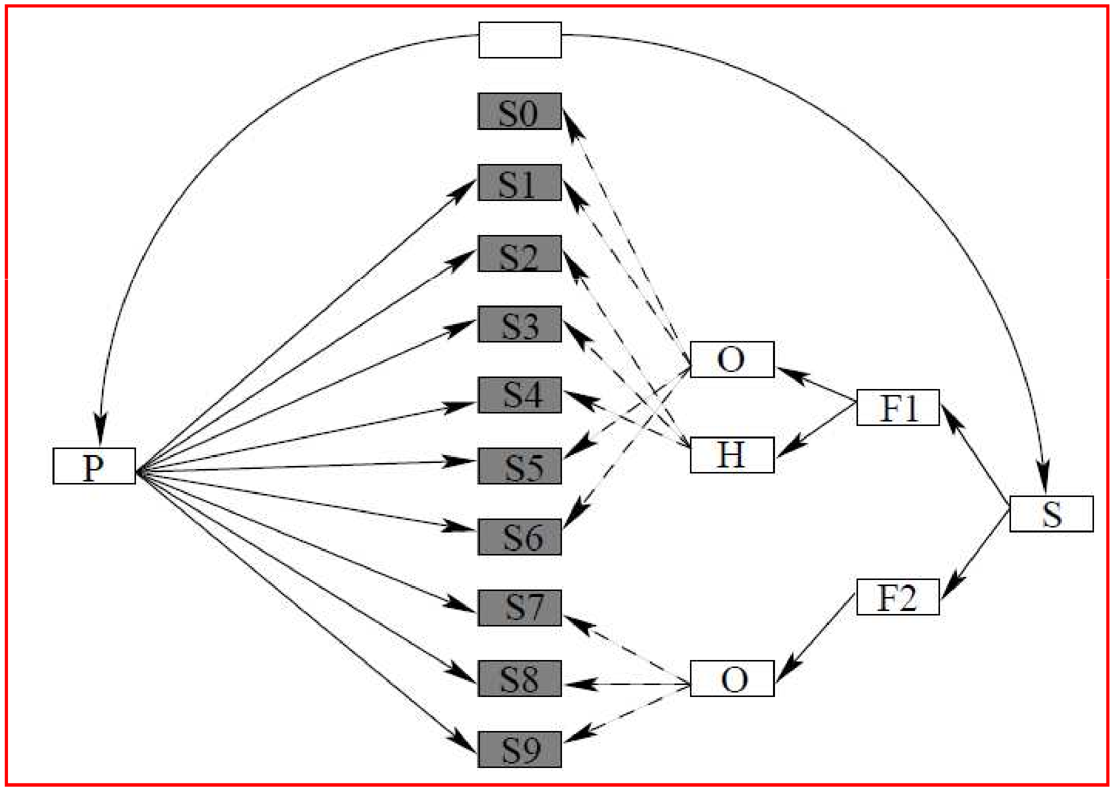
- Four main TSS concepts
  - TSS grammar
  - TSS parser
  - Path navigator
  - TSS engine

Multi-Structured Complex Application Objects

Conceptual user view



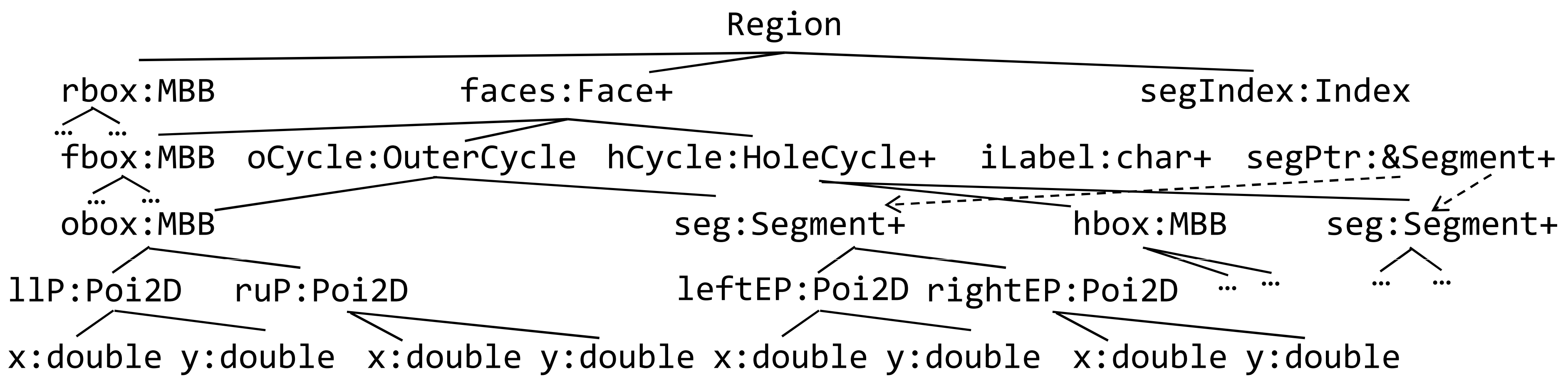
Type structure view / implementation view



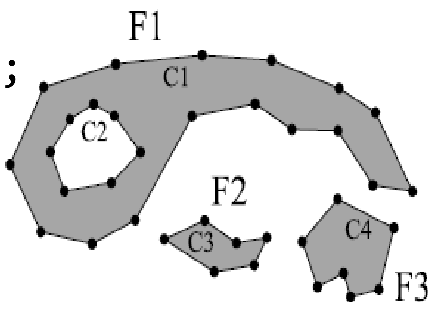
P := primary structure
S := secondary structure
F := face
O := outer cycle
H := hole cycle
S := segment

TSS Grammar

❖ Example: A region ADT and its TSS hierarchy/tree and TSS grammar



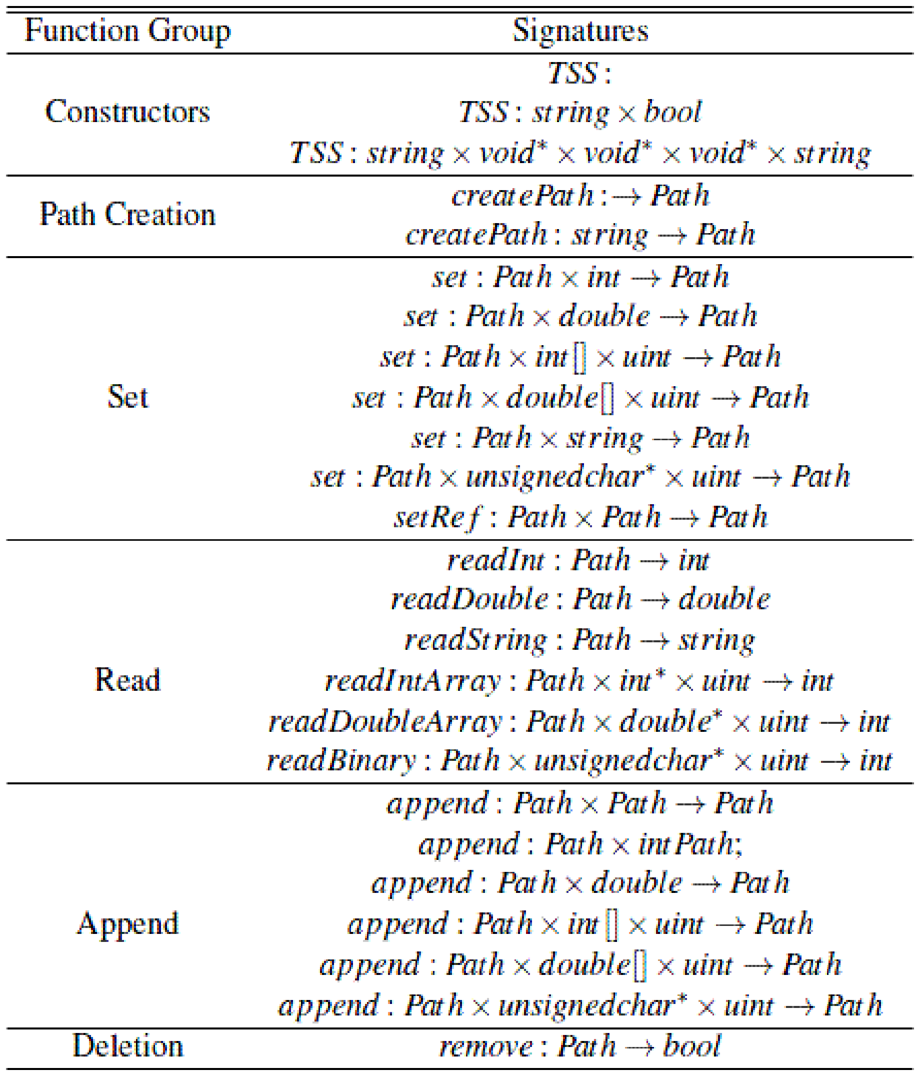
```
Region      ::= rBox:MBB faces:Face+ segIndex:Index;
MBB         ::= llP:Poi2D ruP:Poi2D;
Face        ::= fBox:MBB oCycle:OuterCycle hCycle:HoleCycle+;
OuterCycle  ::= oBox:MBB seg:Segment+;
HoleCycle   ::= hBox:MBB seg:Segment+;
Segment     ::= leftEP:Poi2D rightEP:Poi2D;
Poi2D       ::= x:double y:double;
Index       ::= iLabel:char+ segPtr:&Segment+;
```



TSS Paths

- A path
  - describes a route from the root to a subcomponent of an object
  - is required in order to identify and access parts of a complex object
  - consists of a dot separated string of component names
  - is a human understandable, “semantic pointer” to locations inside a complex object
- Paths are manipulated as strings
- Examples (let *reg* be an object of type *Region*):
  - “reg.rbox”
  - “reg.faces[3].ocycle.seg[4].leftEP.y”
  - “reg.segIndex.segPtr[9]”

TSS Interface (I)



Function Group	Signatures
Constructors	$TSS:$
	$TSS: string \times bool$
	$TSS: string \times void^* \times void^* \times void^* \times string$
Path Creation	$createPath: \rightarrow Path$
	$createPath: string \rightarrow Path$
Set	$set: Path \times int \rightarrow Path$
	$set: Path \times double \rightarrow Path$
	$set: Path \times int[] \times uint \rightarrow Path$
	$set: Path \times double[] \times uint \rightarrow Path$
	$set: Path \times string \rightarrow Path$
	$set: Path \times unsignedchar^* \times uint \rightarrow Path$
	$setRef: Path \times Path \rightarrow Path$
Read	$readInt: Path \rightarrow int$
	$readDouble: Path \rightarrow double$
	$readString: Path \rightarrow string$
	$readIntArray: Path \times int^* \times uint \rightarrow int$
	$readDoubleArray: Path \times double^* \times uint \rightarrow int$
	$readBinary: Path \times unsignedchar^* \times uint \rightarrow int$
Append	$append: Path \times Path \rightarrow Path$
	$append: Path \times int Path;$
	$append: Path \times double \rightarrow Path$
	$append: Path \times int[] \times uint \rightarrow Path$
	$append: Path \times double[] \times uint \rightarrow Path$
	$append: Path \times unsignedchar^* \times uint \rightarrow Path$
Deletion	$remove: Path \rightarrow bool$

TSS Interface (II)

- Constructors
  - *create*(*tssfile*) creates a TSS object from a grammar stored in file *tssfile*
  - *create*(*tssgrammar*) creates a TSS object from a grammar given as an input string in *tssgrammar*
- Path creation
  - *createPath*() returns a default path pointing to the root of the complex data type
  - *createPath*(*strPath*) returns a path based on the input string *strPath*
- Read functions
  - *readInt*(*p*), *readDouble*(*p*), etc. read and return an *int* value, *double* value, etc. from a base object pointed to by path *p*
  - *readIntArray*(*p*, *a*), *readDoubleArray*(*p*, *a*), etc. read an *int* array, *double* array, etc. *a* from a base object pointed to by path *p*. Each function returns the size of the array read.

TSS Interface (III)

- Set functions
  - *set*(*p*, *v*) stores the value *v* of type *int*, *double*, etc. into a new base object at the location pointed to by the path *p* and returns the path to the newly inserted object
  - *set*(*p*, *a*, *n*) stores an array *a* of *n* *int*, *double*, etc. values into a new base object at the location pointed to by the path *p* and returns the path to the newly inserted object
  - *setRef*(p_1, p_2) stores a reference object pointing to a location described by the path p_2 at the location pointed to by the path p_1 and returns the path to the newly inserted object

- Delete function
  - *remove*(*p*) removes the base object, structured object, or reference object pointed to by the path *p*

TSS Interface (IV)

- Append function
  - *append*(*p*, *v*) appends the value *v* of type *int*, *double*, etc. to an *int* array, *double* array, etc. at the location pointed to by the path *p* and returns the path to the newly inserted object
  - *append*(*p*, *a*, *n*) appends an array *a* of *n int*, *double*, etc. values to an *int* array, *double* array, etc. at the location pointed to by the path *p* and returns the path to the newly inserted object
  - *append*(p_1, p_2) appends a reference object pointing to a location described by the path p_2 to an array of reference objects at the location pointed to by the path p_1 and returns the path to the newly inserted object

Requirements Checked (II)

- ✓ User must be able to design, implement, and integrate own complex data types into databases
- ✓ Design and implementation must be independent of the DBMS
- ✓ Design of complex data types must be performed at a high abstraction level and not at a low byte level
- ✓ Enable multi-structured data representations of the same complex object
- ✓ Complex data types should be designed as proper abstract data types that hide implementation details for both data structures and algorithms
- ✓ Complex objects should be arranged in compact storage structures (no main memory pointer structures) in order to avoid expensive serialization and deserialization cost
- ✓ Stored binary data should correspond to the representation that algorithms can process to avoid data conversion
- ✓ Efficient access to components of complex objects must be possible to avoid their complete loading into main memory
- ✓ Updates (random insertions and deletions) should be possible

Requirements Checked (III)

- ✓ User must be able to design, implement, and integrate own complex data types into databases
- ✓ Design and implementation must be independent of the DBMS
- ✓ Design of complex data types must be performed at a high abstraction level and not at a low byte level
- ✓ Enable multi-structured data representations of the same complex object
- ✓ Complex data types should be designed as proper abstract data types that hide implementation details for both data structures and algorithms
- ✓ Complex objects should be arranged in compact storage structures (no main memory pointer structures) in order to avoid expensive serialization and deserialization cost
- ✓ Stored binary data should correspond to the representation that algorithms can process to avoid data conversion
- ✓ Efficient access to components of complex objects must be possible to avoid their complete loading into main memory
- ✓ Updates (random insertions and deletions) should be possible

Outline

1. Introduction and Motivation
2. Problems of Complex Object Management Approaches
3. Requirements of Complex Object Management in Databases
4. iBLOB: Complex Object Management in Databases through Intelligent Binary Large Objects
5. TSS: Representing the Structure of Complex Application Objects through Type Structure Specifications
6. Conclusions and Future Work

Conclusions

- ❖ Available approaches to complex object management in databases reveal shortcomings
- ❖ Application developer and type system implementer depend on but should be independent of DBMS specific implementations of advanced data types
- ❖ Type system implementers should be enabled to design and implement own new type systems (algebras) and integrate them into any DBMS and its query language
- ❖ Talk has shown ongoing research work to accomplish these goals
- ❖ The iBLOB concept and the TSS concept
  - ➢ are intended to free the application developer and the type system implementer from DBMS specific peculiarities and restrictions
  - ➢ enable the type system implementer to be creative and to design new advanced type systems under his/her control

Future Work

- In general, more research needed regarding complex object management in databases in the sense of the “big data” initiative

- Regarding iBLOBs and TSS
  - Intensive functionality and practicability testing needed
  - Derivation of C++ class skeletons
  - Performance testing and comparison, tuning
  - Consequences for query processing

Thank You

for Your Attention!