Object-Oriented Databases
Storage and Indexing

• Type Hierarchy Indexing
• Aggregation Path Indexing
• Collection Operations
Motivation

- Management of large data sets on persistent storage
  - physical storage management
  - content-based access
- Requirements of object-oriented model surpasses those of relational model
  - storage, clustering and management of complex objects on physical persistent media
  - access through object references and query predicates
  - type inheritance hierarchies
  - relationships
  - multi-valued properties and collections
- Additional storage and indexing technologies necessary
Object-Oriented Storage

- Object-oriented storage layouts are often not very different from relational systems
- Difference stems from the algorithms used to manage physical storage
  - data structures to represent complex objects
  - grouping or clustering records of complex objects
  - grouping or clustering of referenced objects
  - management of free space
  - management of buffers
Storage Model

- Physical storage is partitioned into value and access structures.
- Data managed in data sets:
  - records of identical type
  - attribute set $A = \{A_1, \ldots, A_n\}$
  - domains $D_i = \{min_i, max_i\}$
- Pages or clusters correspond to a disk block or a sequence of disk blocks of fixed size $b$
- Records are stored in pages:
  - list of $n$ values $t = (v_1, \ldots, v_n)$
- Addresses reference pages
- Functions to map records to pages and vice-versa
Terminology

- **Query**
  - **point and range queries**: match over exact values or intervals
  - **single class and class hierarchy**: result contains instances of a single class or instances of root class and all of its subclasses
  - **one- and multi-dimensional**: one or more matching conditions

- **Index**
  - **unique and non-unique**: index over keys or non-key fields (primary or secondary index)
  - **sequential and non-sequential key**: index over ordered or unordered values
  - **one- and multi-dimensional**: index over one or more fields
  - **compound**: one-dimensional index over more than one value by concatenating fields
  - **placing (clustering) and non-placing (non-clustering)**: search data structure that does (or does not) allocate record physically
Index Data Structure Overview

- **Sequential organisation**
  - maintains list of storage pages that belong to data set
  - new records placed in most recently allocated storage chunk
  - during query all pages have to be retrieved

- **Subspace mapping**
  - decomposition of data space into subspace
  - overlapping and non-overlapping subspaces
  - B-trees, K-d trees, grid files

- **Point mapping**
  - direct mapping of data space elements to storage chunks
  - function determines record signature used to find address
  - hashing, extensible hashing
Sequential Mapping

Index for Data Set

Page Directory

- 3927
- 6345
- 8172
- ...

Data Set

- 3927
  - Record
- 6345
  - Record
  - Record
- 8172
  - Record
  - Record
  - Record
Subspace Mapping

Index for Data Set

Subspace Directory

Data Set

Page

Record

Page

Record

Page

Record

Record

Record

boundary value
Point Mapping

Point Mappings (Signatures)

<table>
<thead>
<tr>
<th>6</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>7</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

every value combination yields a record signature

Index for Data Set

Point Directory

- 0
- 1
- 2
- 3
- ...
- n

Data Set

Page
- Record
- Record
- Page
- Record
- Page
- Record
- Page
- Record
- Page
- Record
- Page
Existing Index Data Structures

- One-dimensional index data structures
  - B-tree, $B^+$-tree
  - extensible and linear hashing
  - bounded disorder files
  - signature files
  - partial indices

- Multi-dimensional index data structures
  - K-d tree, R-tree, H-tree, hB-tree, Quadtree, TV-tree, cell tree
  - grid files

- Information Systems
  - lecture covers many of these data structures
  - [https://globis.ethz.ch/#!/course/information-systems-2016/](https://globis.ethz.ch/#!/course/information-systems-2016/)
Type Hierarchy Indexing

- Object-oriented queries
  - directly reference one type
  - implicitly refer to a sub-hierarchy, i.e. a set of types

- Two design approaches for type hierarchy index data structures
  - type grouping: first-level order criterion is object type
  - key grouping: top-level data structure organises key values

- Index design has influence on resulting I/O performance of point and range queries
Single Class Index (SC-Index)

- Originally introduced by the ORION system in 1989
- Index construction for an attribute of a type $t$
  - construct a search structure for all types in sub-hierarchy of $t$
  - search data structures called SC-Index components
  - evaluator needs to traverse all components referenced by query
- Usually implemented using B+-trees, other data structures could be used
Class Hierarchy Index (CH-Index)

- Maintains only one search structure for all objects of all types of indexed hierarchy
- Evaluator scans through B⁺-tree once (single-scan)
  - selects OIDs of types referenced by query
  - discards other OIDs
- Point queries always perform good, range queries depend on number of referenced types
  - good when queries aim at indexed type and all subtypes
  - bad if only few types of indexed hierarchy hit by the query
H-Tree

- A H-tree consists of a set of nested B+-trees
- Nesting reflects structure of indexed type hierarchy
  - each H-tree component of indexed type is nested with H-trees of immediate subtypes of indexed type
  - H-tree index for an attribute of inheritance sub-graph is H-tree hierarchy nested according to supertype-subtype relation
- Aims to avoid full scans of each B-tree component when several types are queried
H-Tree ...
Class Division Index (CD-Index)

- Find a compromise between
  - indexing and storing instance set for each type
  - indexing and storing extent for each type
- Maintains a specific family of type sets for indexed hierarchy
- Each type set is managed using a search data structure
- Parameters $q$ and $r$ give upper bounds for decompositions
  - $q$: number of search structures required to build a type extent
  - $r$: number of times a type set is managed redundantly or replicated

CD-index with space pruning

Result of rake-contract heuristic

$q = 3$
$r = 2$

$q = 1$
$r = 3$
Multi-Key Type Index (MT-Index)

- Compromise between type and key grouping approaches
- Interprets type membership as an additional object attribute
  - symmetrical indexing of object types and attribute values
  - able to support indexing of more than one attribute with single search data structure
- Can be built using any multi-dimensional data structure
  - BV-tree, hB-tree or hBΠ-tree
- Performance of MT-index depends on linearisation of indexed type hierarchy
- Query evaluation
  - traversal to collect set of relevant disk page addresses
  - check all records and discard those not qualifying for request
Aggregation Path Indexing

- Backward queries without full object scans
  - find all persons working for an organisation located in Zurich
- Forward without retrieving intermediate objects
  - find the city where person o12 works
- Path decomposition schemes
Nested Index and Path Index

- Given a path $C(1).A(1).A(2)\ldots A(n)$ where $C(1)$ is a class, $A(1)$ is an attribute of class $C(1)$ and $A(i)$ is an attribute of a class $C(i)$ such that $C(i)$ is the domain of attribute $A(i-1)$ of class $C(i-1)$.

- A \textit{nested index} provides a direct association between an endig object and corresponding starting objects along a path.

- A \textit{path index} records all subpaths leading to an ending object. As opposed to nested indexes, path indexes can be used to evaluate predicates on all classes along the path.

- Both of these indexes may be implemented using trees or hash tables.
Nested and Path Index Example

Nested Index \( NI(Person\.works\-for\.situated\-at\.city) \)

| \( NI(Person\.city) \) |  
|---|---|
| Basel | o11 o12 |
| Zurich | o13 |

Path Index \( PI(Person\.works\-for\.situated\-at\.city) \)

| NI |  
|---|---|
| Basel | o11.o27.o34, o12.026.033 |
| Zurich | o13.o27.o34 |
Multi-Index (MX)

- Introduced by GemStone in 1989
- Divide path of arbitrary length into sub-paths
  - sub-paths all have length one
  - index maintained over sub-paths
- Multi-Index for path $P$, $MX(P)$, consists of a set of index components $IX(P_i)$
  - $MX(Person\.works-for\.situated-at\.city) = \{IX(Person\.works-for), IX(Organisation\.situated-at), IX(Location\.city)\}$
- Query evaluation
  - concatenating $n$ index edges requires $n$ index scans
  - supports backwards but not forward traversals and queries
Multi-Index Example

Multi-Index MX(Person.works-for.situated-at.city)

<table>
<thead>
<tr>
<th>IX(Person.works-for)</th>
<th>IX(Organisation.situated-at)</th>
<th>IX(Location.city)</th>
</tr>
</thead>
<tbody>
<tr>
<td>o21</td>
<td>o31</td>
<td>Basel</td>
</tr>
<tr>
<td>o11 o12</td>
<td>o21 o22</td>
<td>o33</td>
</tr>
<tr>
<td>o22</td>
<td>o32</td>
<td>Zurich</td>
</tr>
<tr>
<td>o13</td>
<td>o23</td>
<td>o31 o32</td>
</tr>
<tr>
<td>o23</td>
<td>o34</td>
<td></td>
</tr>
<tr>
<td>o13 o14</td>
<td>o23 o24</td>
<td></td>
</tr>
</tbody>
</table>

Indexing graph for the Multi-Index
Access Support Relations (ASR)

- ASR for aggregation path of length \( n \) is \((n+1)\)-ary relation
- Defined using binary ASRs for sub-paths of length one
  - compositions: \( ASR_{\text{can}} \), \( ASR_{\text{full}} \), \( ASR_{\text{left}} \) and \( ASR_{\text{right}} \)
  - decompositions: Nested Index, Path-Index and Join-Index

<table>
<thead>
<tr>
<th>ASR(Person.works-for)</th>
</tr>
</thead>
<tbody>
<tr>
<td>o11</td>
</tr>
<tr>
<td>o12</td>
</tr>
<tr>
<td>o13</td>
</tr>
<tr>
<td>o13</td>
</tr>
<tr>
<td>o14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASR(Organisation.situated-at)</th>
</tr>
</thead>
<tbody>
<tr>
<td>o21</td>
</tr>
<tr>
<td>o22</td>
</tr>
<tr>
<td>o23</td>
</tr>
<tr>
<td>o23</td>
</tr>
<tr>
<td>o24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASR(Location.city)</th>
</tr>
</thead>
<tbody>
<tr>
<td>o31</td>
</tr>
<tr>
<td>o32</td>
</tr>
<tr>
<td>o33</td>
</tr>
</tbody>
</table>
Access Support Relations (ASR) …

The *canonical* extension, denoted $[t_0.A_1.\cdots.A_n]_{can}$ contains only information about complete paths, i.e., paths originating in $t_0$ and leading (all the way) to $t_n$. Therefore, it can only be used to evaluate queries that originate in an object of type $t_0$ and “go all the way” to $t_n$.

The *left-complete* extension $[t_0.A_1.\cdots.A_n]_{left}$ contains all paths originating in $t_0$ but not necessarily leading to $t_n$, but possibly ending in a *NULL*.

The *right-complete* extension $[t_0.A_1.\cdots.A_n]_{right}$, analogously, contains paths leading to $t_n$, but possibly originating in some object $o_j$ of type $t_j$ which is not referenced by any object of type $t_{j-1}$ via the $A_j$ attribute.

Finally, the full extension $[t_0.A_1.\cdots.A_n]_{full}$ contains all partial paths, even if they do not originate in $t_0$ or do end in a *NULL*. 
ASR Compositions

\[
\text{ASR}(\text{Person.works-for}) \Join \text{ASR}(\text{Organisation.situated-at}) \Join \text{ASR}(\text{Location.city})
\]

<table>
<thead>
<tr>
<th>\text{ASR}_{\text{can}}(\text{Person.works-for.situated-at.city})</th>
<th>\text{ASR}_{\text{full}}(\text{Person.works-for.situated-at.city})</th>
</tr>
</thead>
<tbody>
<tr>
<td>o11 o21 o31 Zurich</td>
<td>– – o33 Basel</td>
</tr>
<tr>
<td>o12 o21 o31 Zurich</td>
<td>– o24 o34 –</td>
</tr>
<tr>
<td>o13 o22 o31 Zurich</td>
<td>o11 o21 o31 Zurich</td>
</tr>
<tr>
<td>o13 o23 o32 Zurich</td>
<td>o12 o21 o31 Zurich</td>
</tr>
<tr>
<td>o14 o23 o32 Zurich</td>
<td>o13 o22 o31 Zurich</td>
</tr>
<tr>
<td></td>
<td>o13 o23 o32 Zurich</td>
</tr>
<tr>
<td></td>
<td>o14 o23 o32 Zurich</td>
</tr>
<tr>
<td></td>
<td>o14 o23 o34 –</td>
</tr>
</tbody>
</table>

\[
\text{ASR}(\text{Person.works-for}) \Join \text{ASR}(\text{Organisation.situated-at}) \Join \text{ASR}(\text{Location.city})
\]
ASR Compositions

(ASR(Person.works-for) \bowtie ASR(Organisation.situated-at)) \bowtie ASR(Location.city)

| ASR_{\text{left}}(\text{Person.works-for.situated-at.city}) |
|-----------------|-----------------|-----------------|
| o11             | o21             | o31             | Zurich          |
| o12             | o21             | o31             | Zurich          |
| o13             | o22             | o31             | Zurich          |
| o13             | o23             | o32             | Zurich          |
| o14             | o23             | o32             | Zurich          |
| o14             | o23             | o34             | –               |

| ASR_{\text{right}}(\text{Person.works-for.situated-at.city}) |
|-----------------|-----------------|-----------------|
| –               | –               | o33             | Basel           |
| o11             | o21             | o31             | Zurich          |
| o12             | o21             | o31             | Zurich          |
| o13             | o22             | o31             | Zurich          |
| o13             | o23             | o32             | Zurich          |
| o14             | o23             | o32             | Zurich          |

ASR(Person.works-for) \bowtie (ASR(Organisation.situated-at) \bowtie ASR(Location.city))
ASR Compositions Indexing Graph

- Queries that do not traverse the path at either endpoint cannot be answered efficiently
- Aggregation path can be split into sub-paths
  - for each an ASR extension (partition) is maintained
  - the set of partitions is called a decomposition of an ASR
Nested Index (NX)

- Only allows backwards traversals of the full path
- Equivalent to backward traversal in ASR_{can}
- A Nested Index $NX(P)$ for a path $P$ with length one is equivalent to a Multi-Index $MX(P)$ for the same path
Path-Index (PX)

- Equivalent to backwards traversal of right-complete ASR
- A Path-Index $PX(P)$ for a path $P$ with length one is equivalent to a Multi-Index $MX(P)$ and a Nested Index $NX(P)$ for the same path.
Join-Index (JX)

- Originally introduced to optimise joins in relational DBMS
- Join-Index consists of a set of binary join indices
  - one binary join index per sub-path of length one of indexed path
  - each binary join index kept redundantly in two data structures
  - binary join index equivalent to corresponding binary ASR
Collection Operations

- Object-oriented databases allow multi-valued attributes
  - sets, bags, lists and arrays of values
  - new modelling features and enhanced expressiveness
  - increased complexity of indexing and query optimisation
- OQL provides constructors and operators for collections

```sql
select p.name from Publication p
where "XCM" in p.topics

select p.name from Publication p
where Set("Indexing", "Storage", "OQL") <= p.topics

select p.name from Publication p
where Set("Indexing", "Storage", "OQL") >= p.topics
```
Signature Files

- Originally proposed for information retrieval
- Index construction for a multi-valued property of a type
  - compute element signature for every possible attribute value
  - compute set signature based on element signatures
  - signature file stores set signatures for all objects
- Query evaluation over multi-valued properties
  - compare query set $S_Q$ to each target $S_T$ by matching the query signature $\text{sig}(S_Q)$ and the target signature $\text{sig}(S_T)$
  - matching of signatures yields drops
    - $S_T$ is a drop for $S_Q \subseteq S_T$ iff $(\text{sig}(S_Q) \land \text{sig}(S_T)) = \text{sig}(S_Q)$
    - $S_T$ is a drop for $S_Q \supseteq S_T$ iff $(\text{sig}(S_Q) \land \text{sig}(S_T)) = \text{sig}(S_T)$
  - drops can be actual drops or false drops
Signature Files Example

Query Set $S_Q$

<table>
<thead>
<tr>
<th>Element</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexing</td>
<td>00100001</td>
</tr>
<tr>
<td>Storage</td>
<td>01000001</td>
</tr>
<tr>
<td>OQL</td>
<td>00100010</td>
</tr>
<tr>
<td>Query</td>
<td>01100011</td>
</tr>
</tbody>
</table>

$S_Q \subseteq S_T$

Target Sets $S_T$

<table>
<thead>
<tr>
<th>Element</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexing</td>
<td>00100001</td>
</tr>
<tr>
<td>Storage</td>
<td>01000001</td>
</tr>
<tr>
<td>OQL</td>
<td>00100010</td>
</tr>
<tr>
<td>Modelling</td>
<td>10010000</td>
</tr>
<tr>
<td>Target</td>
<td>11110011</td>
</tr>
</tbody>
</table>

actual drop

<table>
<thead>
<tr>
<th>Element</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>01000001</td>
</tr>
<tr>
<td>OQL</td>
<td>00100010</td>
</tr>
<tr>
<td>Target</td>
<td>01100011</td>
</tr>
</tbody>
</table>

actual drop

false drop

$S_Q \supseteq S_T$

<table>
<thead>
<tr>
<th>Element</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>01000001</td>
</tr>
<tr>
<td>OQL</td>
<td>00100010</td>
</tr>
<tr>
<td>Target</td>
<td>01100011</td>
</tr>
</tbody>
</table>

false drop
Literature

Literature

