Database Indexes

Quiz!

How costly is this operation (naive solution)?

course	*per*	*weekday*	*hour*	*room*
TDA356	2	VR	Monday	13:15
TDA356	2	VR	Thursday	08:00
TDA356	4	HB1	Tuesday	08:00
TDA356	4	HB1	Friday	13:15
TIN090	1	HC1	Wednesday	08:00
TIN090	1	HA3	Thursday	13:15

n

```
SELECT *
FROM   Lectures
WHERE  course = 'TDA356'
       AND period = 2;
```

Go through all *n* rows, compare with the values for course and period = 2*n* comparisons

Quiz!

Can you think of a way to make it faster?

```
SELECT *
FROM   Lectures
WHERE  course = 'TDA356'
       AND period = 2;
```

If rows were stored sorted according to the values course and period, we could get all rows with the given values faster (O(log n) for tree structure).

Storing rows sorted is expensive, but we can use an *index* that given values of these attributes points out all sought rows (an index could be a hash map, giving O(1) complexity to lookups).

Index

- When relations are large, scanning all rows to find matching tuples becomes very expensive.
- An *index* on an attribute A of a relation is a data structure that makes it efficient to find those tuples that have a fixed value for attribute A.
  - Example: a hash table gives amortized O(1) lookups.

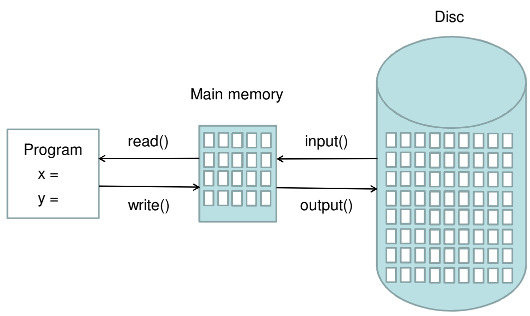
Quiz!

Asymptotic complexity (O(x) notation) is misleading here. Why?

The asymptotic complexity works for data structures in main memory. But when working with stored persistent data, the running time of the data structure, once in main memory, is negligible compared to the time it takes to read data from disk. What really matters to get fast lookups in a database is to minimize the number of disk blocks accessed (could use asymptotic complexity over disk block accessing though).

Indexes help here too though. If a relation is stored over a number of disk blocks, knowing in which of these to look is helpful.

Disc and main memory



Typical costs

- Some typical costs of disk accessing for database operations on a relation stored over n blocks:
  - Query the full relation: n (disk operations)
  - Query with the help of index: k, where k is the number of blocks pointed to (1 for key).
  - Access index: 1
  - Insert new value: 2 (one read, one write)
  - Update index: 2 (one read, one write)

Example:

```
SELECT *
FROM   Lectures
WHERE  course = 'TDA356'
       AND period = 2;
```

Assume Lectures is stored in *n* disk blocks. With no index to help the lookup, we must look at all rows, which means looking in all *n* disk blocks for a total cost of *n*.

With an index, we find that there are 2 rows with the correct values for the course and period attributes. These are stored in two different blocks, so the total cost is 3 (2 blocks + reading index).

Quiz!

How costly is this operation?

```
SELECT *
FROM   Lectures, Courses
WHERE  course = code;
```

Lectures: n disk blocks

Courses: m disk blocks

No index:
Go through all *n* blocks in Lectures, compare the value for course from each row with the values for code in all rows of Courses, stored in all *m* blocks. The total cost is thus ***n * m*** accessed disk blocks.

Index on code in Courses:
Go through all *n* blocks in Lectures, compare the value for course from each row with the index. Since course is a key, each value will exist at most once, so the cost is ***2 * n + 1*** accessed disk blocks (1 for fetching the index once).

CREATE INDEX

- Most DBMS support the statement

```
CREATE INDEX index name
  ON table (attributes);
```

  - Example:

```
CREATE INDEX courseIndex
  ON Courses (code);
```

  - Statement not in the SQL standard, but most DBMS support it anyway.
  - Primary keys are given indexes implicitly (by the SQL standard).

Important properties

- Indexes are separate data stored by itself.
  - Can be created
    - ✓ on newly created relations
    - ✓ on existing relations
      - will take a long time on large relations.
  - Can be dropped without deleting any table data.
- SQL statements do not have to be changed
  - a DBMS automatically uses any indexes.

Quiz!

Why don't we have indexes on all attributes for faster lookups?

- Indexes require disk space.
- Modifications of tables are more expensive.
  - Need to update both table and index.
- Not always useful
  - The table is very small.
  - We don't perform lookups over it (Note: lookups ≠ queries).
- Using an index costs extra disk block accesses.

Rule of thumb

- Mostly queries on tables – use indexes for key attributes.
- Mostly updates – be careful with indexes!

Quiz!

Assume we have an index on Lectures for (course, period, weekday) which is the key. How costly are these queries?

Lectures: n disk blocks

```
SELECT *
FROM   Lectures
WHERE  course = 'TDA356'
       AND period = 2;
```

```
SELECT *
FROM   Lectures
WHERE  weekday = 'Monday'
       AND room = 'VR';
```

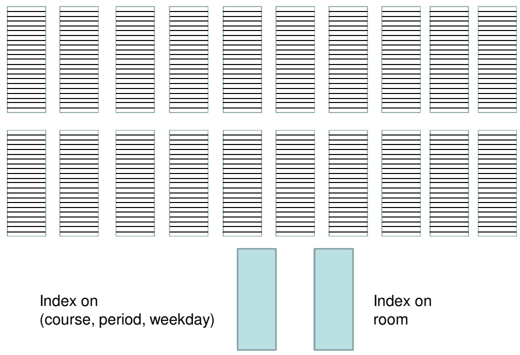
A multi-attribute index is typically organized hierarchically. First the rows are indexed according to the first attribute, then according to the second within each group, and so on.
Thus the ***left*** query costs at most ***k + 1*** where k is the number of rows matching the values. The ***right*** query can't use the index, and thus costs ***n***, where n is the size of the relation in disk blocks.

Example: Suppose that the Lectures relation is stored in 20 disk blocks, and that we typically perform three operations on this table:
- insert new lectures (Ins)
- list all lectures of a particular course (Q1)
- list all lectures in a given room (Q2)

Let's assume that in an average week there are:
- 2 lectures for each course, and
- 10 lectures in each room.

Let's also assume that
- each course has lectures stored in 2 blocks, and
- each room has lectures stored in 7 (some lectures are stored in the same block).

Lectures example: blocks



Costs

Insert new lectures (Ins)
List all lectures of a particular course (Q1)
List all lectures in a given room (Q2)

	Case A	Case B	Case C	Case D
	No index	Index on (course, period, weekday)	Index on room	Both indexes
Ins	2	4	4	6
Q1	20	3	20	3
Q2	20	20	8	8

The amortized cost depends on the proportion of operations of each kind.

Ins	Q1	Q2	Case A	Case B	Case C	Case D
0.2	0.4	0.4	16.4	10	12	5.6
0.8	0.1	0.1	5.6	5.5	6	5.9
0.1	0.6	0.3	18.2	8.2	14.8	4.8

Dense index on sequential file

KBB056	
KMB017	
TDA357	
TMS145	
UMF012	
UMF018	

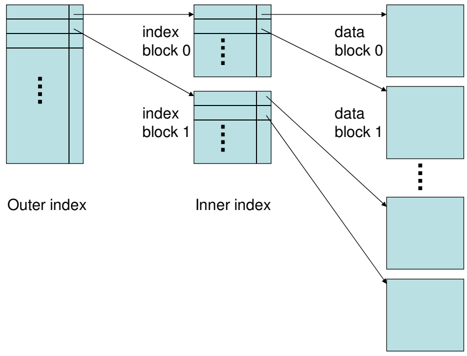
KBB056	KC	Monday	08	
KMB017	MVH12	Tuesday	08	
KMB017	MVH12	Wednesday	15	
TDA357	HA4	Monday	10	
TDA357	HB1	Thursday	10	
TMS145	KC	Friday	08	
UMF012	MVF23	Friday	13	
UMF012	MVF23	Monday	13	
UMF018	MVF23	Tuesday	10	

Sparse index on sequential file

KBB056	
TDA357	
UMF012	

KBB056	KC	Monday	08	
KMB017	MVH12	Tuesday	08	
KMB017	MVH12	Wednesday	15	
TDA357	HA4	Monday	10	
TDA357	HB1	Thursday	10	
TMS145	KC	Friday	08	
UMF012	MVF23	Friday	13	
UMF012	MVF23	Monday	13	
UMF018	MVF23	Tuesday	10	

Multi-level indexes

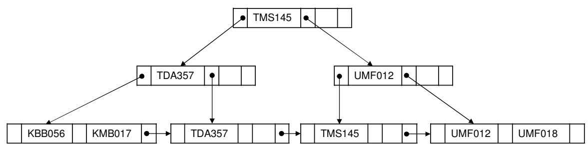


Secondary index on *room name*

HA4	
HB1	
KC	
MVF23	
MVH12	

KBB056	KC	Monday	08	
KMB017	MVH12	Tuesday	08	
KMB017	MVH12	Wednesday	15	
TDA357	HA4	Monday	10	
TDA357	HB1	Thursday	10	
TMS145	KC	Friday	08	
UMF012	MVF23	Friday	13	
UMF012	MVF23	Monday	13	
UMF018	MVF23	Tuesday	10	

B$^+$-tree



- widely used
- non-leaf nodes form a hierarchy of sparse indexes
- insertions and deletions require only small, local changes

Quiz!

- Indexes are incredibly useful (although they are not part of the SQL standard).
- Doing it wrong is costly.
- Requires knowledge about the internals of a DBMS.
  - How is data stored? How large is a block?
- A DBMS should be able to decide better than the user what indexes are needed, from usage analysis.

So why don't they??

Summary – indexes

- Indexes make certain lookups and joins more efficient.
  - Disk block access matters.
  - Multi-attribute indexes
- **`CREATE INDEX`**
- Dense, sparse, multi-level and secondary
- Usage analysis
  - What are the expected operations?
  - How much do they cost?
    - Σ(cost of operation)x(proportion of operations of that kind)

XML

Semistructured data
XML, DTD, (XMLSchema)
XPath, XQuery

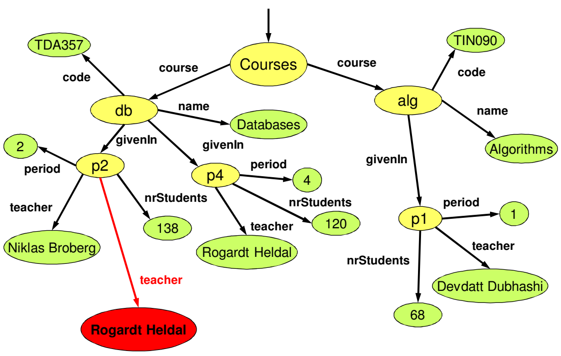
Quiz!

Assume we have a single course (Databases) that is the exception to the rule in that it has two responsible teachers (Niklas Broberg, Rogardt Heldal) when given in the 2nd period. How can we model this?

1. Allow all courses to have two teachers. We extend the GivenCourses table with another attribute teacher2, and put NULL there for all other courses.
2. Allow courses to have any number of teachers. We create a separate table Teaches with attributes course, period and teacher, and make all three be the key.

1 means lots of NULLs, 2 means we must introduce a new table. Seems overkill for such an easy task…

Example: A different way of thinking about data…



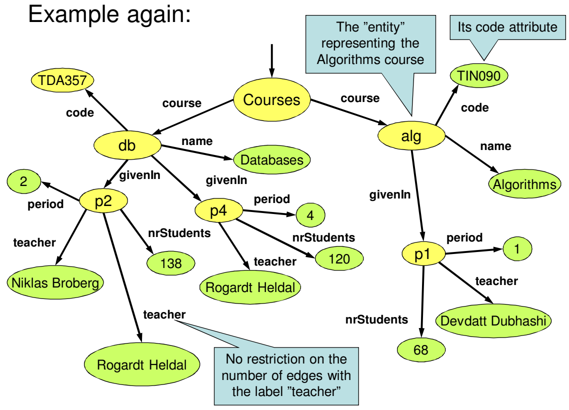
Semi-structured data (SSD)

- More flexible data model than the relational model.
  - Think of an object structure, but with the type of each object its own business.
  - Labels to indicate meanings of substructures.
- Semi-structured: it is structured, but not everything is structured the same way!

SSD Graphs

- Nodes = ”objects”, ”entities”
- Edges with labels represent attributes or relationships.
- Leaf nodes hold atomic values.
- Flexibility: no restriction on
  - Number of edges out from a node.
  - Number of edges with the same label
  - Label names

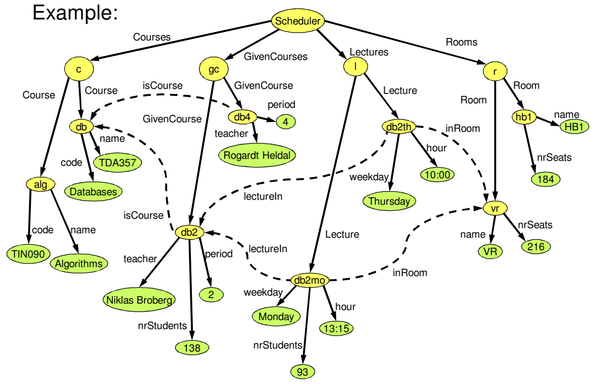
Example again:



Relationships in SSD graphs

- Relationships are marked by edges to some node, that doesn't have to be a child node.
  - This means a SSD graph is not a tree, but a true graph.
  - Cyclic relationships possible.
- Using relationships, it is possible to directly mimic the behavior of the relational model.
  - Graph is three levels deep – one for a relation, the second for its contents, the third for the attributes.
  - References are inserted as relationship edges.
- SSD is a generalization of the relational model!

Example:



Schemas for SSD

- Inherently, semi-structured data does not have schemas.
  - The type of an object is its own business.
  - The schema is given by the data.
- We can of course restrict graphs in any way we like, to form a kind of "schema".
  - Example: All "course" nodes must have a "code" attribute.

XML

- XML = eXtensible Markup Language
- Derives from document markup languages.
  - Compare with HTML: HTML uses "tags" for formatting a document, XML uses "tags" to describe semantics.
- Key idea: create tag sets for a domain, and translate data into properly tagged XML documents.

XML vs SSD

- XML is a language that describes data and its structure.
  - Cf. relational data: SQL DDL + data in tables.
- The data model behind XML is semi-structured data.
  - Using XML, we can describe an SSD graph as a tagged document.

Example XML document:

Child nodes are represented by child elements inside the parent element.

A node is represented by an element marked by a start and an end tag.

```
<Scheduler>
  <Courses>
    <Course code="TDA357" name="Databases">
      <GivenIn
        nrStudents="138"
        teacher="Niklas Broberg">2</GivenIn>
      <GivenIn
        nrStudents="93"
        teacher="Rogardt Heldal">4</GivenIn>
    </Course>
  </Courses>
</Scheduler>
```

Leaf nodes with values can be represented as either attributes…

… or as element data

Note that XML is case sensitive!

XML explained

- An XML element is denoted by surrounding tags:
```
<Course>...</Course>
```
- Child elements are written as elements between the tags of its parent, as is simple string content:
```
<Course><GivenIn>2</GivenIn></Course>
```
- Attributes are given as name-value pairs inside the starting tag:
```
<Course code="TDA357">…</Course>
```
- Elements with no children can be written using a short-hand:
```
<Course code="TDA357" />
```

Example again:

Starting tags of elements

Attributes

```
<Scheduler>
  <Courses>
    <Course code="TDA357" name="Databases>
      <GivenIn
        nrStudents="138"
        teacher="Niklas Broberg">2</GivenIn>
      <GivenIn
        nrStudents="93"
        teacher="Rogardt Heldal">4</GivenIn>
    </Course>
  </Courses>
</Scheduler>
```

Child elements inside the parents

String content (CDATA)

Note that XML is case sensitive!

XML namespaces

- XML is used to describe a multitude of different domains. Many of these will work together, but have name clashes.
- XML defines *namespaces* that can disambiguate these circumstances.
  - Example:

Use xmlns to bind namespaces to variables in this document.

```
<sc:Scheduler
    xmlns:sc="http://www.cs.chalmers.se/~dbas/xml"
    xmlns:www="http://www.w3.org/xhtml">
  <sc:Course code="TDA357" sc:name="Databases"
      www:name="dbas" />
</sc:Scheduler>
```

Quiz!

What's wrong with this XML document?

```
<Course code="TDA357">
 <GivenIn period="2" >
 <GivenIn period="4" >
</Course>
```

No end tags provided for the **GivenIn** elements!
We probably meant e.g. **<GivenIn … />**

What about the name of the course? Teachers?

Well-formed and valid XML

- *Well-formed XML* directly matches semi-structured data:
  - Full flexibility – no restrictions on what tags can be used where, how many, what attributes etc.
  - Well-formed means syntactically correct.
    - E.g. all start tags are matched by an end tag.
- *Valid XML* involves a schema that limits what labels can be used and how.

Well-formed XML

- A document must start with a *declaration*, surrounded by **<? … ?>**
  - Normal declaration is:
```
<?xml version="1.0" standalone="yes" ?>
```
    … where standalone means basically "no schema provided".
- Structure of a document is a *root element* surrounding well-formed sub-documents.

DTDs

- DTD = Document Type Definition
- A DTD is a schema that specifies what elements may occur in a document, where they may occur, what attributes they may have, etc.
- Essentially a context-free grammar for describing XML tags and their nesting.

Basic building blocks

- ELEMENT: Define an element and what children it may have.
  - Children use standard regexp syntax: `*` for 0 or more, `+` for 1 or more, `?` for 0 or 1, `|` for choice, commas for sequencing.
  - Example: `<!ELEMENT Courses (Course*)>`
- ATTLIST: Define the attributes of an element.
  - Example:
    ```
    <!ATTLIST Course
      code CDATA #REQUIRED>
    ```
  - Course elements are required to have an attribute `code` of type `CDATA` (string).

Example: Part of a DTD for the Scheduler domain

```
<!DOCTYPE Scheduler [
  <!ELEMENT Scheduler (Course*)>
  <!ELEMENT Course (GivenIn*)>
  <!ELEMENT GivenIn (#PCDATA)>

  <!ATTLIST Course
    code CDATA #REQUIRED
    name CDATA #REQUIRED
  >
  <!ATTLIST GivenIn
    teacher CDATA #IMPLIED
    nrStudents CDATA "0"
  >
]>
```

A Scheduler element can have 0 or more Course elements as children.

PCDATA means Character Data, i.e. a string. DTDs have (almost) no other base types.

These attributes must be set… (Cf. NOT NULL)

…but not this one.

Default value is 0

Quiz: If we want courses to be able to have more than one teacher, what could we do?

One suggestion is to make a "Teacher" element with PCDATA content, and allow GivenIn elements to have 1 or more of those as children. Period could be an attribute instead.

Non-tree structures

- DTDs allow references between elements.
  - The type of one attribute of an element can be set to ID, which makes it unique.
  - Another element can have attributes of type IDREF, meaning that the value must be an ID in some other element.

```
<!ATTLIST Room
  name ID #REQUIRED>
<!ATTLIST Lecture
  room IDREF #IMPLIED>
```

```
<Scheduler>
 … <Room name="VR" … />
 … <Lecture room="VR" … />
</Scheduler>
```

Beginning of document with DTD

```
<?xml version="1.0"
      encoding="utf-8"
      standalone="no" ?>
<!DOCTYPE Scheduler [
 <!ELEMENT Scheduler
           (Courses,Rooms)>
 <!ELEMENT Courses (Course*)>
 <!ELEMENT Rooms (Room*)>
 <!ELEMENT Course (GivenIn*)>
 <!ELEMENT GivenIn (Lecture*)>
 <!ELEMENT Lecture EMPTY>
 <!ELEMENT Room EMPTY>

 <!ATTLIST Course
   code ID #REQUIRED
   name CDATA #REQUIRED >
 <!ATTLIST GivenIn
   period CDATA #REQUIRED
   teacher CDATA #IMPLIED
   nrStudents CDATA "0" >
 <!ATTLIST Lecture
   weekday CDATA #REQUIRED
   hour CDATA #REQUIRED
   room IDREF #IMPLIED >
 <!ATTLIST Room
   name ID #REQUIRED
   nrSeats CDATA #IMPLIED >
]>
```

Document body

```
<Scheduler>
 <Courses>
  <Course code="TDA357"
     name="Databases">
   <GivenIn period="2"
      teacher="Niklas Broberg"
      nrStudents="138">
    <Lecture weekday="Monday"
       hour="13:15" room="VR" />
    <Lecture weekday="Thursday"
       hour="10:00" room="HB1" />
   </GivenIn>
   <GivenIn period="4"
      teacher="Rogardt Heldal">
   </GivenIn>
  </Course>
 </Courses>
 <Rooms>
  <Room name="VR" nrSeats="216"/>
  <Room name="HB1" nrSeats="184"/>
 </Rooms>
</Scheduler>
```

courses.xml (a smaller example)

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Courses [
 <!ELEMENT Courses (Course*)>
 <!ELEMENT Course (GivenIn*)>
 <!ELEMENT GivenIn EMPTY>
 <!ATTLIST Course
   code ID #REQUIRED
   name CDATA #REQUIRED >
 <!ATTLIST GivenIn
   period CDATA #REQUIRED
   teacher CDATA #IMPLIED >
]>

<Courses>
 <Course name="Databases" code="TDA357">
  <GivenIn period="2" teacher="Niklas Broberg" />
  <GivenIn period="4" teacher="Rogardt Heldal" />
 </Course>
 <Course name="Algorithms" code="TIN090">
  <GivenIn period="1" teacher="Devdatt Dubhashi" />
 </Course>
</Courses>
```

Quiz!

What's wrong with DTDs?

- Only one base type – CDATA.
- No way to specify constraints on data other than keys and references.
- No way to specify what elements references may point to – if something is a reference then it may point to any key anywhere.
- …

XML Schema

- Basic idea: why not use XML to define schemas of XML documents?
- XML Schema instances are XML documents specifying schemas of other XML documents.
- XML Schema is much more flexible than DTDs, and solves all the problems listed and more!
- DTDs are still the standard – but XML Schema is the recommendation (by W3)!

Example: fragment of an XML Schema:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema”>

 <element name=”Course”>
  <complexType>
   <attribute name=”code” use=”required” type=”string”>
   <attribute name=”name” use=”required” type=”string”>
   <sequence>
    <element name=”GivenIn” maxOccurs=”4”>
     <complexType>
      <attribute name=”period” use=”required”>
       <simpleType>
        <restriction base=”integer”>
         <minInclusive value=”1” />
         <maxInclusive value=”4” />
        </restriction>
       </simpleType>
      </attribute>
      <attribute name=”teacher” use=”optional” type=”string” />
      <attribute name=”nrStudents” use=”optional” type=”integer” />
      <sequence>...</sequence>
     </complexType>
    </element>
   </sequence>
  </complexType>
 </element>
</schema>
```

Multiplicity constraint: A course can only be given at most four times a year.

Value constraint: Period must be an integer, restricted to values between 1 and 4 inclusive.

We can have keys and references as well, and any general assertions (though they can be tricky to write correctly).