XPath • CSS • DOM • Selenium Rosetta Stone and Cookbook

Sprinkled with Selenium usage tips, this is both a general-purpose set of recipes for each technology as well as a cross-reference to map from one to another.
The validation suite for this reference chart (http://bit.ly/gTd5oc) provides example usage for each recipe supported by Selenium (the majority of them).

Category	Recipe	XPath (1.0 – 2.0)	CSS (CSS1 – 3)	DOM	Selenium
General	Whole web page	xpath=/html	css=html	document.documentElement	NA
	Whole web page body	xpath=/html/body	css=body	document.body	NA
	All text nodes of web page	//text() ⌧	NA	NA	NA
	Element <E> by absolute reference	xpath=/html/body/.../.../.../E	css=body>...>...>...>E	document.body.childNodes[i]...childNodes[j]	NA
Tag	Element <E> by relative reference	//E	css=E	document.gEBTN('E')[0]	NA
	Second <E> element anywhere on page	xpath=(//E)[2]	NA	document.gEBTN('E')[1]	NA
	Image element	//img	css=img	document.images[0]	NA
	Element <E> with attribute A	//E[@A]	css=E[A]	dom=for each (e in document.gEBTN('E')) if (e.A) e❶	NA
	Element <E> with attribute A containing text 't' exactly	//E[@A='t']	css=E[A='t'] ❷	NA	NA
	Element <E> with attribute A containing text 't'	//E[contains(@A,'t')]	css=E[A*='t'] ❷	NA	NA
	Element <E> whose attribute A begins with 't'	//E[starts-with(@A, 't')]	css=E[A^='t'] ❷	NA	NA
	Element <E> whose attribute A ends with 't'	//E[ends-with(@A, 't')] ⌧ ◀OR▶ //E[substring(@A, string-length(@A) - string-length('t')+1)='t']	css=E[A$='t'] ❷	NA	NA
	Element <E> with attribute A containing word 'w'	//E[contains(concat('◉', @A, '◉'), '◉w◉')]	css=E[A~='w'] ❷	NA	NA
	Element <E> with attribute A matching regex 'r'	//E[matches(@A, 'r')] ⌧	NA	NA	NA
	Element <E1> with id I1 or element <E2> with id I2	//E1[@id=I1] | //E2[@id=I2]	css=E1#I1,E2#I2	NA	NA
	Element <E1> with id I1 or id I2	//E1[@id=I1 or @id=I2]	css=E1#I1,E1#I2	NA	NA
Attribute❸	Attribute A of element <E>	//E/@A ⌧ {Se: //E@A }	NA {Se: css=E@A }	document.gEBTN('E')[0].getAttribute('A') ⌧ {Se: document.gEBTN('E')[0]@A }	NA
	Attribute A of any element	//*/@A ⌧ {Se: //*@A }	NA {Se: css=*@A }	NA	NA
	Attribute A1 of element <E> where attribute A2 is 't' exactly	//E[@A2='t']/@A1 ⌧ {Se: //E[@A2='t']@A1 }	NA {Se: css=E[A2='t']@A1 }	NA	NA
	Attribute A of element <E> where A contains 't'	//E[contains(@A,'t')]/@A ⌧ {Se: //E[contains(@A,'t')]@A }	NA {Se: css=E[A*='t']@A }	NA	NA
Id & Name	Element <E> with id I	//E[@id='I']	css=E#I	NA	NA
	Element with id I	//*[@id='I']	css=#I	document.gEBI('I')	id=I
	Element <E> with name N	//E[@name='N']	css=E[name=N]	NA	NA
	Element with name N	//*[@name='N']	css=[name=N]	document.getElementsByName('N')[0]	name=N
	Element with id X or, failing that, a name X	//*[@id='X' or @name='X']	NA	NA	X ◀OR▶ identifier=X
	Element with name N & specified 0-based index 'v'	//*[@name='N'][v+1]	css=[name=N]:nth-child(v+1)	NA	name=N index=v
	Element with name N & specified value 'v'	//*[@name='N'][@value='v']	css=[name=N][value='v']	NA	name=N value=v
Lang & Class	Element <E> is explicitly in language L or subcode	//E[@lang='L' or starts-with(@lang, concat('L', '-'))]	css=E[lang|=L]	NA	NA
	Element <E> is in language L or subcode (possibly inherited)	NA	css=E:lang(L)	NA	NA
	Element with a class C	//*[contains(concat('◉', @class, '◉'), '◉C◉')]	css=.C	document.getElementsByClassName('C')[0]	NA
	Element <E> with a class C	//E[contains(concat('◉', @class, '◉'), '◉C◉')]	css=E.C	NA	NA
Text & Link	Element containing text 't' exactly	//*[.='t']	NA	NA	NA
	Element <E> containing text 't'	//E[contains(text(),'t')]	css=E:contains('t') ❹	NA	NA
	Link element	//a	css=a	document.links[0]	NA
	<a> containing text 't' exactly	//a[.='t']	NA	NA	link=t
	<a> containing text 't'	//a[contains(text(),'t')]	css=a:contains('t') ❹	NA	NA
	<a> with target link 'url'	//a[@href='url']	css=a[href='url']	NA	NA
	Link URL labeled with text 't' exactly	//a[.='t']/@href	NA	NA	NA
Parent & Child	First child of element <E>	//E/*[1]	css=E > *:first-child { Se: css=E > * }	document.gEBTN('E')[0].firstChild❺	NA
	First <E> child	//E[1]	css=E:first-of-type ⌧ { Se: css=E }	document.getEBTN('E')[0]	NA
	Last child of element E	//E/*[last()]	css=E *:last-child	document.gEBTN('E')[0].lastChild❺	NA
	Last <E> child	//E[last()]	css=E:last-of-type ⌧	document.gEBTN(E)[document.gEBTN(E).length-1]	NA
	Second <E> child	//E[2] ◀OR▶ //E/following-sibling::E	css=E:nth-of-type(2) ⌧	document.getEBTN('E')[1]	NA
	Second child that is an <E> element	//*[2][name()='E']	css=E:nth-child(2)	NA	NA
	Second-to-last <E> child	//E[last()-1]	css=E:nth-last-of-type(2) ⌧	document.gEBTN(E)[document.gEBTN(E).length-2]	NA
	Second-to-last child that is an <E> element	//*[last()-1][name()='E']	css=E:nth-last-child(2) ⌧	NA	NA
	Element <E1> with only <E2> children	//E1/[E2 and not(*[not(self::E2)])]	NA	NA	NA
	Parent of element <E>	//E/..	NA	document.gEBTN('E')[0].parentNode	NA
	Descendant <E> of element with id I using specific path	//*[@id='I']/ . . ./. . ./. . ./E	css=#I > . . . > . . . > . . . > E	document.gEBI('I')…gEBTN('E')[0]	NA
	Descendant <E> of element with id I using unspecified path	//*[@id='I']//E	css=#I E	document.gEBI('I').gEBTN('E')[0]	NA
	Element <E> with no children	//E[count(*)=0]	css=E:empty	NA	NA
	Element <E> with an only child	//E[count(*)=1]	NA	NA	NA
	Element <E> that *is* an only child	//E[count(preceding-sibling::*)+count(following-sibling::*)=0]	css=E:only-child	NA	NA
	Element <E> with no <E> siblings	//E[count(../E) = 1]	css=E:only-of-type ⌧	NA	NA
	Every Nth element starting with the (M+1)th	//E[position() mod N = M + 1]	css=E:nth-child(Nn + M)	NA	NA
Sibling	Element <E1> following some sibling <E2>	//E2/following-sibling::E1	css=E2 ~ E1	NA	NA
	Element <E1> immediately following sibling <E2>	//E2/following-sibling::*[1][name()='E1']	css=E2 + E1	NA	NA
	Element <E1> following sibling <E2> with one intermediary	//E2/following-sibling::*[2][name()='E1']	css=E2 + * + E1	NA	NA
	Sibling element immediately following <E>	//E/following-sibling::*	css=E + *	document.gEBTN('E')[0].nextSibling❺	NA
	Element <E1> preceding some sibling <E2>	//E2/preceding-sibling::E1	NA	NA	NA
	Element <E1> immediately preceding sibling <E2>	//E2/preceding-sibling::*[1][name()='E1']	NA	NA	NA
	Element <E1> preceding sibling <E2> with one intermediary	//E2/preceding-sibling::*[2][name()='E1']	NA	NA	NA
	Sibling element immediately preceding <E>	//E/preceding-sibling::*[1]	NA	document.gEBTN('E2')[0].previousSibling❺	NA
Table Cell	Cell by row and column (e.g. 3rd row, 2nd column)	//*[@id='TestTable']//tr[3]//td[2] {Se: //*[@id='TestTable'].2.1 }	css=#TestTable tr:nth-child(3) td:nth-child(2) {Se: css=#TestTable.2.1 }	document.gEBI('TestTable').gEBTN('tr')[2].gEBTN('td')[1] {Se: document.gEBI('TestTable').2.1 }	NA
	Cell immediately following cell containing 't' exactly	//td[preceding-sibling::td='t']	NA	NA	NA
	Cell immediately following cell containing 't'	//td[preceding-sibling::td[contains(.,'t')]]	css=td:contains('t') ~ td ❹	NA	NA
Dynamic	User interface element <E> that is disabled	//E[@disabled]	css=E:disabled	NA	NA
	User interface element that is enabled	//*[not(@disabled)]	css=*:enabled	NA	NA
	Checkbox (or radio button) that is checked	//*[@checked]	css=*:checked	NA	NA
	Element being designated by a pointing device	NA	css=E:hover ⌧	NA	NA
	Element has keyboard input focus	NA	css=E:focus ⌧	NA	NA
	Unvisited link	NA	css=E:link ⌧	NA	NA
	Visited link	NA	css=E:visited ⌧	NA	NA
	Active element	NA	css=E:active ⌧	NA	NA

LEGEND

- XPath
- CSS
- DOM
- Selenium

{Se: . . . }	Selenium-only variation
⌧	Not supported by Selenium
◉	Space character
expression	CSS3 or XPath 2.0

DOM abbreviations:

gEBI getElementById
gEBTN getElementsByTagName

2011.04.05 • Version 1.0.2

Download the latest version from Simple-Talk http://bit.ly/gTd5oc.

General Notes

Indexing (all): XPath and CSS use 1-based indexing; DOM and Selenium's table syntax use 0-based indexing.

Prefixes (all): xpath= required unless expression starts with // • dom= required unless expression starts with "document." • css= ***always*** required • identifier= ***never*** required.

Cardinality (Selenium): XPath and CSS may specify a node set **or** a single node; DOM **must** specify a single node. When a node set is specified, Selenium returns just the first node.

Content (XPath): Generally should use normalize-space() when operating on display text.

Footnotes

❶ DOM has limited capability with a simple 'document…' expression; however, arbitrary JavaScript code may be used as shown in this example.

❷ CSS does not support qualifying elements with the **style** attribute, as in div[style*='border-width'].

❸ Selenium uses a special syntax for returning attributes; normal XPath, CSS, and DOM syntax will fail.

❹ CSS: The CSS2 **contains** function is ***not in CSS3***; however, Selenium supports the superset of CSS1, 2, and 3.

❺ DOM: **firstChild**, **lastChild**, **nextSibling**, and **previousSibling** are problematic with mixed content; they will point to empty text nodes rather than desired elements depending on whitespace in web page

XPath

nodename	Selects all nodes with the name "*nodename*"
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

Wildcards and Multiple Paths

*	Matches any element node
@*	Matches any attribute node
node()	Matches any node of any kind
|	Place between paths to select several paths
/bookstore/*	Selects all the child element nodes of the bookstore element
//*	Selects all elements in the document
//title[@*]	Selects all title elements which have at least one attribute of any kind
//book/title | //book/price	Selects all the title AND price elements of all book elements
//title | //price	Selects all the title AND price elements in the document

Wildcards and Multiple Paths (cont)

/bookstore/book/title | //price	Selects all the title elements of the book element of the bookstore element AND all the price elements in the document

Location Step Examples

The syntax for a location step is:

`axisna me: :no det est [pr edi - cate]`

child::book	Selects all book nodes that are children of the current node
attribute::lang	Selects the lang attribute of the current node
child::*	Selects all element children of the current node
attribute::*	Selects all attributes of the current node
child::text()	Selects all text node children of the current node
child::node()	Selects all children of the current node
descendant::book	Selects all book descendants of the current node
ancestor::book	Selects all book ancestors of the current node
ancestor-or-self::book	Selects all book ancestors of the current node - and the current node as well if it is a book node
child::*/child::price	Selects all price grandchildren of the current node

XPath Operators

|	Computes two node-sets	//book | //cd
+	Addition	6 + 4
-	Subtraction	6 - 4
*	Multiplication	6 * 4
div	Division	8 div 4
=	Equal	price=9.80
!=	Not equal	price!=9.80
<	Less than	price<9.80
<=	Less than or equal to	price<=9.80
>	Greater than	price>9.80
>=	Greater than or equal to	price>=9.80
or	or	price=9.80 or price=9.70
and	and	price>9.00 and price<9.90
mod	Modulus	5 mod 2

Predicates

/bookstore/book[1]	Selects the first book element that is the child of the bookstore element.
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element

Predicates (cont)

/bookstore/book[position()<3]	Selects the first two book elements that are children of the bookstore element
//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='en']	Selects all the title elements that have a "lang" attribute with a value of "en"
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

Note: In IE 5,6,7,8,9 first node is[0], but according to W3C, it is [1]. To solve this problem in IE, set the SelectionLanguage to XPath.

In JavaScript: xml.setProperty("SelectionLanguage","XPath");

XPath Examples

bookstore	Selects all nodes with the name "bookstore"
/bookstore	Selects the root element bookstore
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document
bookstore//book	Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
//@lang	Selects all attributes that are named lang
/bookstore/book/title	Selects all title nodes that are descendants of book that are descendants of bookstore
/bookstore/book[1]/title	Selects the title of the first book node under the bookstore element
/bookstore/book/price[text()]	Selects the text from all bookstore/book/price nodes
/bookstore/book[price>35]/price	Selects all the bookstore/book/price nodes with a price greater than 35

Note: If the path started with a slash (/) it always represents an absolute path to an element!

XPath Axes

ancestor	Selects all ancestors of the current node
ancestor-or-self	Selects all ancestors of the current node and the current node itself
attribute	Selects all attributes of the current node
child	Selects all children of the current node
descendant	Selects all descendants of the current node
descendant-or-self	Selects all descendants of the current node and the current nod itself
following	Selects everything in the document after the closing tag of the current node
following-sibling	Selects all siblings after the current node
namespace	Selects all namespace nodes of the current node
parent	Selects the parent of the current node
preceding	Selects all nodes that appear before the current node in the document, except ancestors, attribute nodes, and namespace nodes
preceding-sibling	Selects all siblings before the current node
self	Selects the current node

XQuery Cheat Sheet

by BenHuf via cheatography.com/163984/cs/34363/

Overview

XQuery is ***the*** language for querying XML data

XQuery for XML is like SQL for databases

XQuery is built on XPath expressions

XQuery is supported by all major databases

XQuery is a W3C Recommendation

XQuery is a language for finding and extracting elements and attributes from XML documents.

Here is an example of what XQuery could solve:

"Select all CD records with a price less than $10 from the CD collection stored in cd_catalog.xml"

FLWOR Expressions

For	Selects a sequence of nodes
Let	Binds a sequence to a variable
Where	Filters the nodes
Order by	Sorts the nodes
Return	What to return (gets evaluated once for every node)

Example:

```
for $x in doc("bo oks.xm l")/ boo kst ore /book
where $x/pri ce>30
order by $x/title
return $x/title
```

The For Clause

The For Clause (cont)

```
return <te st> x={$x} and y={$y} </t est>
```

Returns

```
<te st>x=10 and y=100< /te st>
<te st>x=10 and y=200< /te st>
<te st>x=20 and y=100< /te st>
<te st>x=20 and y=200< /te st>
```

The Let Clause

This

```
let $x := (1 to 5)
return <te st> {$x }</ tes t>
```

Returns

```
<te st>1 2 3 4 5</ tes t>
```

The let clause allows variable assignments and it avoids repeating the same expression many times. The let clause does not result in iteration.

The Where Clause

This

where $x/price>30 and $x/price<100

Returns Nodes only where the price is between 30 and 100

The where clause is used to specify one or more criteria for the result.

The order by Clause

This

```
for $x in doc("bo oks.xm l")/ boo kst ore /book
order by $x/@ca tegory, $x/title
return $x/title
```

Returns

```
<title lang="e n">Harry Potter </t itl e>
<title lang="e n">E veryday Italia n</ title
<title lang="e n">L earning XML</t itl e>
<title lang="e n">X Query Kick Start< /ti tle>
```

The order by clause is used to specify the sort order of the result.

To loop a specific number of times in a for clause, you may use the **to** keyword.

This

```
for $x in (1 to 5)
return <te st> {$x }</ tes t>
```

Returns

```
<te st> 1</ tes t>
<te st> 2</ tes t>
<te st> 3</ tes t>
<te st> 4</ tes t>
<te st> 5</ tes t>
```

To count the iteration use the **at** keyword

This

```
for $x at $i in doc("bo oks.xm l")/ boo kst -
ore /bo ok/ title
return <bo ok> {$i}. {data( $x) }</ boo k>
```

Returns

```
<bo ok>1. Everyday Italia n</ boo k>
<bo ok>2. Harry Potter </b ook>
<bo ok>3. XQuery Kick Start< /bo ok>
<bo ok>4. Learning XML</b ook>
```

it is also allowed with more than one expression in the for clause. Use comma to separate each in expression.

This

```
for $x in (10,20), $y in (100,200)
```

The Return Clause

This

```
for $x in doc("bo oks.xm l")/ boo kst ore /book
return $x/title
```

Returns

```
<title lang="e n">E veryday Italia n</ tit le>
<title lang="e n">Harry Potter </t itl e>
<title lang="e n">X Query Kick Start< /ti tle>
<title lang="e n">L earning XML</t itl e>
```

The return clause specifies what is to be returned.

XQuery Basics

doc()	doc("books.xml")	Used to open a file
Path Expressions	doc("books.xml")**/bookstore/book/title**	Used to navigate through elements in an XML document
Predicates	doc("books.xml")/bookstore/book**[price<30]**	Used to limit the extracted data

XQuery Basic Syntax Rules

XQuery is case-sensitive

XQuery elements, attributes, and variables must be valid XML names

An XQuery string value can be in single or double quotes

An XQuery variable is defined with a $ followed by a name, e.g. $bookstore

XQuery comments are delimited by (: and :)

```
(: XQuery Comment :)
```

Path Expressions vs. FLWOR Expressions

Path Expression	FLWOR Expression
`doc("bo oks.xm - l")/ boo kst - ore /bo ok[ pri - ce> 30] /title`	`for $x in doc("books.xml")/bookstore/book where $x/price>30 return $x/title`

These expressions yield the same result.

XQuery Comparisons

General Comparisons: =, !=, <, <=, >, >=

```
$books tor e// book/@q > 10
```

Returns true if any q attributes have a value greater than 10

Value Comparisons: eq, ne, lt, le, gt, ge

```
bookst ore //b ook/@q gt 10
```

Returns true if there is only one q attribute returned by the expression and its value is greater than 10. If more than one 1 is returned, and error occurs

Examples of Function Calls

Example 1: In an element

```
<na me> {up per -ca se( $bo okt itl e)} </n ame>
```

Example 2: In the predicate of a path expression

```
doc("bo oks.xm l")/ boo kst ore /bo ok[ sub str -
ing (ti tle ,1, 5)= 'Ha rry']
```

Example 3: In a let clause

```
let $name := (subst rin g($ boo kti tle ,1,4))
```

A call to a function can appear where an expression may appear.

User-Defined Functions

If you cannot find the XQuery function you need, you can write your own. User-defined functions can be defined in the query or in a separate library.

Syntax

```
declare function prefix :fu nct ion _na me( $pa -
rameter as datatype)
as return Dat atype
{
...fun ction code here...
};
```

Example: Declared in the Query

```
declare function local: min Pri ce($p as xs:dec -
ima l?,$d as xs:dec imal?)
as xs:dec imal?
{
let $disc := ($p * $d) div 100
return ($p - $disc)
};
```

How to call the function

User-Defined Functions (cont)

```
<mi nPr ice >{l oca l:m inP ric e($ boo k/p ric -
e,$ boo k/d isc oun t)} </m inP ric e>
```

Note:
Use the declare function keyword.
The name of the function must be prefixed.
The data type of the parameters are mostly the same as the data types defined in XML Schema.
The body of the function must be surrounded by curly braces.

XQuery Terminology

Nodes	In XQuery there are 7 kinds: element, attribute, text, namespace, processing-instruction, comment, and document (root)
Atomic Values	Atomic values are nodes with no children or parent
Items	Items refer to nodes and atomic values
Parent	Each element and attribute has one parent
Children	Element nodes may have 0, 1, or more children.
Siblings	Nodes that have the same parent
Ancestors	A node's parent, parent's parent, etc.
Descen-dants	A node's children, children's children, etc.

XQuery Conditional Expressions

If-Then-Else expressions are allowed in XQuery

```
for $x in doc("bo oks.xm l")/ boo kst ore /book
return if ($x/@c ate gor y="c hil dre n")
then <ch ild >{d ata ($x /ti tle )}< /ch ild>
else <ad ult >{d ata ($x /ti tle )}< /ad ult>
```

Note: "if-then-else" syntax: parentheses around the if expression are required. else is required, but it can be just else ().

Returning HTML

This

```
for $x in doc("bo oks.xm l")/ boo kst ore /book
where $x/pri ce>30
order by $x/title
return $x/title
```

Yields

```
<ul>
<li ><title lang="e n">E veryday Italia n</ tit -
le> </l i>
<li ><title lang="e n">Harry Potter </t itl e>
< /li>
<li ><title lang="e n">L earning XML</t itl e>
< /li>
<li ><title lang="e n">X Query Kick Start< /ti -
tle ></ li>
</u l>
====== === === === === === === === === === === -
=== === === ======
```

This

```
<ul>
{
for $x in doc("bo oks.xm l")/ boo kst ore /bo -
ok/ title
order by $x
return <li >{d ata ($x )}< /li>
}
</u l>
```

Yeilds

```
<ul>
<li >Ev eryday Italia n</ li>
<li >Harry Potter </l i>
<li >Le arning XML</l i>
<li >XQuery Kick Start< /li>
</u l>
```

Note: To eliminate the title element and show only data inside the title element use data($x)

Adding Elements and Attributes to the Result

This

```
<ht ml>
<bo dy>
<h1 >Bo oks tor e</ h1>
<ul>
{
for $x in doc("bo oks.xm l")/ boo kst ore /book
order by $x/title
return <li >{d ata ($x /ti tle)}. Category:
{data( $x/ @ca teg ory )}< /li>
}
</u l>
</b ody>
</h tml>
```

Yields

```
<ht ml>
<bo dy>
<h1 >Bo oks tor e</ h1>
<ul>
<li >Ev eryday Italian. Category: COOKIN G</ li>
<li >Harry Potter. Category: CHILDR EN< /li>
<li >Le arning XML. Category: WEB</l i>
<li >XQuery Kick Start. Category: WEB</l i>
</u l>
</b ody>
</h tml>
```

====== === === === === === === === === === ======

This

```
<ht ml>
<bo dy>
<h1 >Bo oks tor e</ h1>
<ul>
{
```

Adding Elements and Attributes to the Result (cont)

```
for $x in doc("bo oks.xm l")/ boo kst ore /book
order by $x/title
return <li class= " {da ta( $x/ @ca teg ory )}"> -
{da ta( $x/ tit le) }</ li>
}
</u l>
</b ody>
</h tml>
```

Yields

```
<ht ml>
<bo dy>
<h1 >Bo oks tor e</ h1>
<ul>
<li class= " COO KIN G">E veryday Italia n</ li>
<li class= " CHI LDR EN"> Harry Potter </l i>
<li class= " WEB " >Le arning XML</l i>
<li class= " WEB " >XQuery Kick Start< /li>
</u l>
</b ody>
</h tml>
```

Regular Expressions cheat sheet

Basic matching

Each symbol matches a single character:

`.`	anything[1]
`\d`	digit in `0123456789`
`\D`	non-digit
`\w`	"word" (letters and digits and _)
`\W`	non-word
`␣`	space
`\t`	tab
`\r` `\n`	return new line[2]
`\s`	whitespace (`␣`, `\t`, `\r`, `\n`)
`\S`	non-whitespace

Character classes

Character classes `[...]` match any of the characters in the class. **Ex:** `[aeiou]` matches vowels. Use `^` to specify the complement set: `[^aeiou]` matches non-vowels (including non-letters!). Use `-` to specify a range of letters: `[a-e]` matches `abcde` and `[0-9a-f]` matches `0123456789abcdef`.

Boundaries

Boundary characters are helpful in "anchoring" your pattern to some edge, but do not select any characters themselves.

`\b`	word boundaries (as defined as any edge between a `\w` and `\W`)
`\B`	non-word-boundaries
`^`	the beginning of the line
`$`	the end of the line

Ex: `\bcat\b` finds a match in "the cat in the hat" but not in "locate".

Disjunction

`(X|Y)`	`X` or `Y`

Ex: `\b(cat|dog)s\b` matches cats and dogs.

"Quantifiers"

`X*`	0 or more repetitions of `X`
`X+`	1 or more repetitions of `X`
`X?`	0 or 1 instances of `X`
`X{`m`}`	exactly m instances of `X`
`X{`m`,}`	at least m instances of `X`
`X{`m`,`n`}`	between m and n (inclusive) instances of `X`

By default, quantifiers just apply to the one character. Use `(...)` to specify explicit quantifier "scope."

Ex: `ab+` matches `ab`, `abb`, `abbb`, `abbbb`...
`(ab)+` matches `ab`, `abab`, `ababab`...

Quantifiers are by default *greedy* in regex. Good regex engines support adding `?` to a quantifier to make it *lazy*.

Ex: *greedy:* `^.*b` aabaaba
lazy: `^.*?b` aabaaba

Special characters

The characters `{}[]()^$.|*+?\` (and `-` inside `[...]`) have special meaning in regex, so they must be "escaped" with `\` to match them.

Ex: `\.` matches the period `.` and `\\` matches the backslash `\`.

Backreferences

Count your open parentheses `(` from the left, starting with 1. Whatever is matched by parenthesis number n can be referenced later by `\`n.

Ex: `\b(\w+)␣\1\b` matches two identical words with a space in between

Backreferences are useful for *find/replaces*:

Ex: Finding `\b(\w+)er\b` and replacing with `more \1` will map "the taller man" ↦ "the more tall man" and "I am shorter" ↦ "I am more short".

Advanced

Read about "non-capturing parentheses" and "look-ahead" and "look-behind" online. Also, visualize your regexes as finite-state machines at `http://www.regexper.com/`.

---

[1] ...except line breaks, depending on your engine.

[2] Depending on where you got your file, line breaks may be `\r`, `\n`, or `\r\n`. Also, in some regex engines (e.g. TextWrangler), `\r` and `\n` match the same things.

mitcho@mitcho.com

SQL Basics Cheat Sheet

LearnSQL • com

SQL, or *Structured Query Language*, is a language to talk to databases. It allows you to select specific data and to build complex reports. Today, SQL is a universal language of data. It is used in practically all technologies that process data.

SAMPLE DATA

COUNTRY

id	name	population	area
1	France	66600000	640680
2	Germany	80700000	357000
...	...	...	...

CITY

id	name	country_id	population	rating
1	Paris	1	2243000	5
2	Berlin	2	3460000	3
...	...	...	...	...

QUERYING SINGLE TABLE

Fetch all columns from the country table:

```
SELECT *
FROM country;
```

Fetch id and name columns from the city table:

```
SELECT id, name
FROM city;
```

Fetch city names sorted by the rating column in the default ASCending order:

```
SELECT name
FROM city
ORDER BY rating [ASC];
```

Fetch city names sorted by the rating column in the DESCending order:

```
SELECT name
FROM city
ORDER BY rating DESC;
```

ALIASES

COLUMNS

```
SELECT name AS city_name
FROM city;
```

TABLES

```
SELECT co.name, ci.name
FROM city AS ci
JOIN country AS co
  ON ci.country_id = co.id;
```

FILTERING THE OUTPUT

COMPARISON OPERATORS

Fetch names of cities that have a rating above 3:

```
SELECT name
FROM city
WHERE rating > 3;
```

Fetch names of cities that are neither Berlin nor Madrid:

```
SELECT name
FROM city
WHERE name != 'Berlin'
  AND name != 'Madrid';
```

TEXT OPERATORS

Fetch names of cities that start with a 'P' or end with an 's':

```
SELECT name
FROM city
WHERE name LIKE 'P%'
  OR name LIKE '%s';
```

Fetch names of cities that start with any letter followed by 'ublin' (like Dublin in Ireland or Lublin in Poland):

```
SELECT name
FROM city
WHERE name LIKE '_ublin';
```

OTHER OPERATORS

Fetch names of cities that have a population between 500K and 5M:

```
SELECT name
FROM city
WHERE population BETWEEN 500000 AND 5000000;
```

Fetch names of cities that don't miss a rating value:

```
SELECT name
FROM city
WHERE rating IS NOT NULL;
```

Fetch names of cities that are in countries with IDs 1, 4, 7, or 8:

```
SELECT name
FROM city
WHERE country_id IN (1, 4, 7, 8);
```

QUERYING MULTIPLE TABLES

INNER JOIN

JOIN (or explicitly **INNER JOIN**) returns rows that have matching values in both tables.

```
SELECT city.name, country.name
FROM city
[INNER] JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	3	Iceland

LEFT JOIN

LEFT JOIN returns all rows from the left table with corresponding rows from the right table. If there's no matching row, **NULL**s are returned as values from the second table.

```
SELECT city.name, country.name
FROM city
LEFT JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	NULL	NULL

RIGHT JOIN

RIGHT JOIN returns all rows from the right table with corresponding rows from the left table. If there's no matching row, **NULL**s are returned as values from the left table.

```
SELECT city.name, country.name
FROM city
RIGHT JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
NULL	NULL	NULL	3	Iceland

FULL JOIN

FULL JOIN (or explicitly **FULL OUTER JOIN**) returns all rows from both tables – if there's no matching row in the second table, **NULL**s are returned.

```
SELECT city.name, country.name
FROM city
FULL [OUTER] JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	NULL	NULL
NULL	NULL	NULL	3	Iceland

CROSS JOIN

CROSS JOIN returns all possible combinations of rows from both tables. There are two syntaxes available.

```
SELECT city.name, country.name
FROM city
CROSS JOIN country;
```

```
SELECT city.name, country.name
FROM city, country;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
1	Paris	1	2	Germany
2	Berlin	2	1	France
2	Berlin	2	2	Germany

NATURAL JOIN

NATURAL JOIN will join tables by all columns with the same name.

```
SELECT city.name, country.name
FROM city
NATURAL JOIN country;
```

CITY			COUNTRY	
country_id	id	name	name	id
6	6	San Marino	San Marino	6
7	7	Vatican City	Vatican City	7
5	9	Greece	Greece	9
10	11	Monaco	Monaco	10

NATURAL JOIN used these columns to match rows: **city.id**, **city.name**, **country.id**, **country.name**. **NATURAL JOIN** is very rarely used in practice.

SQL Basics Cheat Sheet

LearnSQL
• com

AGGREGATION AND GROUPING

GROUP BY **groups** together rows that have the same values in specified columns. It computes summaries (aggregates) for each unique combination of values.

CITY		
id	name	country_id
1	Paris	1
101	Marseille	1
102	Lyon	1
2	Berlin	2
103	Hamburg	2
104	Munich	2
3	Warsaw	4
105	Cracow	4

→

CITY	
country_id	count
1	3
2	3
4	2

AGGREGATE FUNCTIONS

- `avg(expr)` − average value for rows within the group
- `count(expr)` − count of values for rows within the group
- `max(expr)` − maximum value within the group
- `min(expr)` − minimum value within the group
- `sum(expr)` − sum of values within the group

EXAMPLE QUERIES

Find out the number of cities:

```
SELECT COUNT(*)
FROM city;
```

Find out the number of cities with non-null ratings:

```
SELECT COUNT(rating)
FROM city;
```

Find out the number of distinctive country values:

```
SELECT COUNT(DISTINCT country_id)
FROM city;
```

Find out the smallest and the greatest country populations:

```
SELECT MIN(population), MAX(population)
FROM country;
```

Find out the total population of cities in respective countries:

```
SELECT country_id, SUM(population)
FROM city
GROUP BY country_id;
```

Find out the average rating for cities in respective countries if the average is above 3.0:

```
SELECT country_id, AVG(rating)
FROM city
GROUP BY country_id
HAVING AVG(rating) > 3.0;
```

SUBQUERIES

A subquery is a query that is nested inside another query, or inside another subquery. There are different types of subqueries.

SINGLE VALUE

The simplest subquery returns exactly one column and exactly one row. It can be used with comparison operators =, <, <=, >, or >=.

This query finds cities with the same rating as Paris:

```
SELECT name
FROM city
WHERE rating = (
  SELECT rating
  FROM city
  WHERE name = 'Paris'
);
```

MULTIPLE VALUES

A subquery can also return multiple columns or multiple rows. Such subqueries can be used with operators IN, EXISTS, ALL, or ANY.

This query finds cities in countries that have a population above 20M:

```
SELECT name
FROM city
WHERE country_id IN (
  SELECT country_id
  FROM country
  WHERE population > 20000000
);
```

CORRELATED

A correlated subquery refers to the tables introduced in the outer query. A correlated subquery depends on the outer query. It cannot be run independently from the outer query.

This query finds cities with a population greater than the average population in the country:

```
SELECT *
FROM city main_city
WHERE population > (
  SELECT AVG(population)
  FROM city average_city
  WHERE average_city.country_id = main_city.country_id
);
```

This query finds countries that have at least one city:

```
SELECT name
FROM country
WHERE EXISTS (
  SELECT *
  FROM city
  WHERE country_id = country.id
);
```

SET OPERATIONS

Set operations are used to combine the results of two or more queries into a single result. The combined queries must return the same number of columns and compatible data types. The names of the corresponding columns can be different.

CYCLING		
id	name	country
1	YK	DE
2	ZG	DE
3	WT	PL
...	...	...

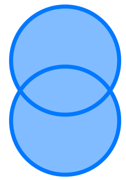
SKATING		
id	name	country
1	YK	DE
2	DF	DE
3	AK	PL
...	...	...

UNION

`UNION` combines the results of two result sets and removes duplicates. **`UNION ALL`** doesn't remove duplicate rows.

This query displays German cyclists together with German skaters:

```
SELECT name
FROM cycling
WHERE country = 'DE'
UNION / UNION ALL
SELECT name
FROM skating
WHERE country = 'DE';
```

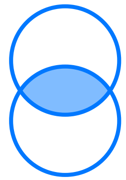


INTERSECT

`INTERSECT` returns only rows that appear in both result sets.

This query displays German cyclists who are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
INTERSECT
SELECT name
FROM skating
WHERE country = 'DE';
```

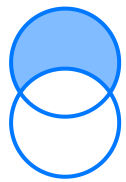


EXCEPT

`EXCEPT` returns only the rows that appear in the first result set but do not appear in the second result set.

This query displays German cyclists unless they are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
EXCEPT / MINUS
SELECT name
FROM skating
WHERE country = 'DE';
```



Try out the interactive **SQL Basics** course at **LearnSQL.com**, and check out our other SQL courses.

Data Wrangling

with pandas Cheat Sheet
http://pandas.pydata.org

Pandas API Reference   Pandas User Guide

Creating DataFrames

	a	b	c
1	4	7	10
2	5	8	11
3	6	9	12

```
df = pd.DataFrame(
        {"a" : [4, 5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
      index = [1, 2, 3])
```
Specify values for each column.

```
df = pd.DataFrame(
     [[4, 7, 10],
      [5, 8, 11],
      [6, 9, 12]],
     index=[1, 2, 3],
     columns=['a', 'b', 'c'])
```
Specify values for each row.

		a	b	c
N	v			
D	1	4	7	10
	2	5	8	11
e	2	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
index = pd.MultiIndex.from_tuples(
      [('d', 1), ('d', 2),
       ('e', 2)], names=['n', 'v']))
```
Create DataFrame with a MultiIndex

Method Chaining

Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

```
df = (pd.melt(df)
        .rename(columns={
                'variable':'var',
                'value':'val'})
        .query('val >= 200')
     )
```

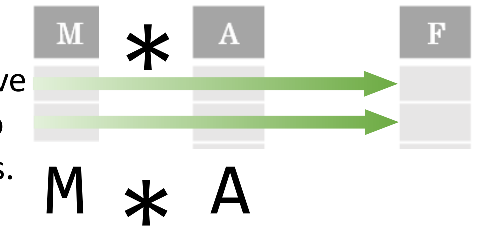
Tidy Data – A foundation for wrangling in pandas

In a tidy data set:

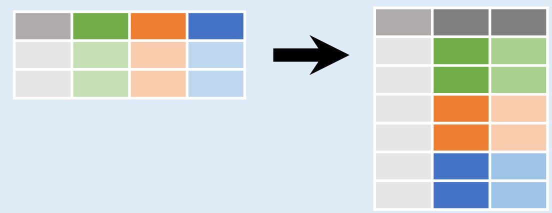
Each **variable** is saved in its own **column**

Each **observation** is saved in its own **row**

Tidy data complements pandas's **vectorized operations**. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.



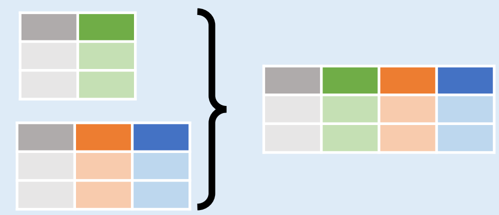
Reshaping Data – Change layout, sorting, reindexing, renaming



`pd.melt(df)`
Gather columns into rows.

`df.pivot(columns='var', values='val')`
Spread rows into columns.

`pd.concat([df1,df2])`
Append rows of DataFrames



`pd.concat([df1,df2], axis=1)`
Append columns of DataFrames

`df.sort_values('mpg')`
Order rows by values of a column (low to high).

`df.sort_values('mpg', ascending=False)`
Order rows by values of a column (high to low).

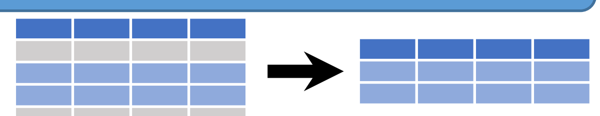
`df.rename(columns = {'y':'year'})`
Rename the columns of a DataFrame

`df.sort_index()`
Sort the index of a DataFrame

`df.reset_index()`
Reset index of DataFrame to row numbers, moving index to columns.

`df.drop(columns=['Length', 'Height'])`
Drop columns from DataFrame

Subset Observations - rows



`df[df.Length > 7]`
Extract rows that meet logical criteria.

`df.drop_duplicates()`
Remove duplicate rows (only considers columns).

`df.sample(frac=0.5)`
Randomly select fraction of rows.

`df.sample(n=10)` Randomly select n rows.

`df.nlargest(n, 'value')`
Select and order top n entries.

`df.nsmallest(n, 'value')`
Select and order bottom n entries.

`df.head(n)`
Select first n rows.

`df.tail(n)`
Select last n rows.

Subset Variables - columns

`df[['width', 'length', 'species']]`
Select multiple columns with specific names.

`df['width']` *or* `df.width`
Select single column with specific name.

`df.filter(regex='regex')`
Select columns whose name matches regular expression *regex*.

Using query

query() allows Boolean expressions for filtering rows.

```
df.query('Length > 7')
df.query('Length > 7 and Width < 8')
df.query('Name.str.startswith("abc")',
            engine="python")
```

Subsets - rows and columns

Use **df.loc[]** and **df.iloc[]** to select only rows, only columns or both.
Use **df.at[]** and **df.iat[]** to access a single value by row and column.
First index selects rows, second index columns.

`df.iloc[10:20]`
Select rows 10-20.

`df.iloc[:, [1, 2, 5]]`
Select columns in positions 1, 2 and 5 (first column is 0).

`df.loc[:, 'x2':'x4']`
Select all columns between x2 and x4 (inclusive).

`df.loc[df['a'] > 10, ['a', 'c']]`
Select rows meeting logical condition, and only the specific columns .

`df.iat[1, 2]` Access single value by index

`df.at[4, 'A']` Access single value by label

Logic in Python (and pandas)

<	Less than	!=	Not equal to
>	Greater than	`df.column.isin(values)`	Group membership
==	Equals	`pd.isnull(obj)`	Is NaN
<=	Less than or equals	`pd.notnull(obj)`	Is not NaN
>=	Greater than or equals	`&,|,~,^,df.any(),df.all()`	Logical and, or, not, xor, any, all

regex (Regular Expressions) Examples

`'\.'`	Matches strings containing a period '.'
`'Length$'`	Matches strings ending with word 'Length'
`'^Sepal'`	Matches strings beginning with the word 'Sepal'
`'^x[1-5]$'`	Matches strings beginning with 'x' and ending with 1,2,3,4,5
`'^(?!Species$).*'`	Matches strings except the string 'Species'

Cheatsheet for pandas (http://pandas.pydata.org/ originally written by Irv Lustig, Princeton Consultants, inspired by Rstudio Data Wrangling Cheatsheet

Summarize Data

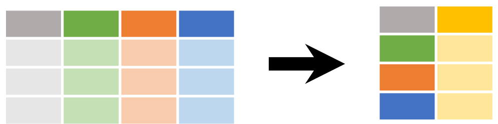
`df['w'].value_counts()`
Count number of rows with each unique value of variable

`len(df)`
# of rows in DataFrame.

`df.shape`
Tuple of # of rows, # of columns in DataFrame.

`df['w'].nunique()`
# of distinct values in a column.

`df.describe()`
Basic descriptive and statistics for each column (or GroupBy).



pandas provides a large set of summary functions that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups. When applied to a DataFrame, the result is returned as a pandas Series for each column. Examples:

`sum()`
Sum values of each object.

`count()`
Count non-NA/null values of each object.

`median()`
Median value of each object.

`quantile([0.25,0.75])`
Quantiles of each object.

`apply(function)`
Apply function to each object.

`min()`
Minimum value in each object.

`max()`
Maximum value in each object.

`mean()`
Mean value of each object.

`var()`
Variance of each object.

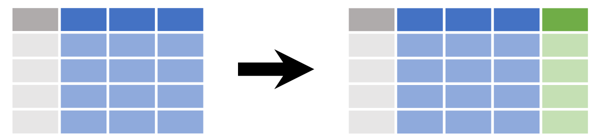
`std()`
Standard deviation of each object.

Handling Missing Data

`df.dropna()`
Drop rows with any column having NA/null data.

`df.fillna(value)`
Replace all NA/null data with value.

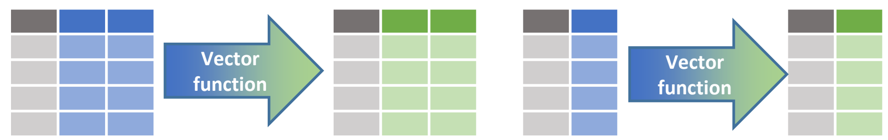
Make New Columns



`df.assign(Area=lambda df: df.Length*df.Height)`
Compute and append one or more new columns.

`df['Volume'] = df.Length*df.Height*df.Depth`
Add single column.

`pd.qcut(df.col, n, labels=False)`
Bin column into n buckets.



pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (a pandas Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

`max(axis=1)`
Element-wise max.

`clip(lower=-10,upper=10)`
Trim values at input thresholds

`min(axis=1)`
Element-wise min.

`abs()`
Absolute value.

Group Data

`df.groupby(by="col")`
Return a GroupBy object, grouped by values in column named "col".

`df.groupby(level="ind")`
Return a GroupBy object, grouped by values in index level named "ind".

All of the summary functions listed above can be applied to a group. Additional GroupBy functions:

`size()`
Size of each group.

`agg(function)`
Aggregate group using function.

The examples below can also be applied to groups. In this case, the function is applied on a per-group basis, and the returned vectors are of the length of the original DataFrame.

`shift(1)`
Copy with values shifted by 1.

`rank(method='dense')`
Ranks with no gaps.

`rank(method='min')`
Ranks. Ties get min rank.

`rank(pct=True)`
Ranks rescaled to interval [0, 1].

`rank(method='first')`
Ranks. Ties go to first value.

`shift(-1)`
Copy with values lagged by 1.

`cumsum()`
Cumulative sum.

`cummax()`
Cumulative max.

`cummin()`
Cumulative min.

`cumprod()`
Cumulative product.

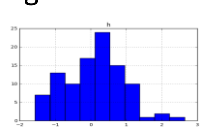
Windows

`df.expanding()`
Return an Expanding object allowing summary functions to be applied cumulatively.

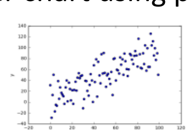
`df.rolling(n)`
Return a Rolling object allowing summary functions to be applied to windows of length n.

Plotting

`df.plot.hist()`
Histogram for each column



`df.plot.scatter(x='w',y='h')`
Scatter chart using pairs of points



Combine Data Sets

adf

x1	x2
A	1
B	2
C	3

+

bdf

x1	x3
A	T
B	F
D	T

=

Standard Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NaN

`pd.merge(adf, bdf, how='left', on='x1')`
Join matching rows from bdf to adf.

x1	x2	x3
A	1.0	T
B	2.0	F
D	NaN	T

`pd.merge(adf, bdf, how='right', on='x1')`
Join matching rows from adf to bdf.

x1	x2	x3
A	1	T
B	2	F

`pd.merge(adf, bdf, how='inner', on='x1')`
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NaN
D	NaN	T

`pd.merge(adf, bdf, how='outer', on='x1')`
Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

`adf[adf.x1.isin(bdf.x1)]`
All rows in adf that have a match in bdf.

x1	x2
C	3

`adf[~adf.x1.isin(bdf.x1)]`
All rows in adf that do not have a match in bdf.

ydf

x1	x2
A	1
B	2
C	3

+

zdf

x1	x2
B	2
C	3
D	4

=

Set-like Operations

x1	x2
B	2
C	3

`pd.merge(ydf, zdf)`
Rows that appear in both ydf and zdf (Intersection).

x1	x2
A	1
B	2
C	3
D	4

`pd.merge(ydf, zdf, how='outer')`
Rows that appear in either or both ydf and zdf (Union).

x1	x2
A	1

```
pd.merge(ydf, zdf, how='outer',
         indicator=True)
.query('_merge == "left_only"')
.drop(columns=['_merge'])
```
Rows that appear in ydf but not zdf (Setdiff).

Cheatsheet for pandas (http://pandas.pydata.org/) originally written by Irv Lustig, Princeton Consultants, inspired by Rstudio Data Wrangling Cheatsheet

HADOOP AND MAPREDUCE CHEAT SHEET

Hadoop & MapReduce Basics

Hadoop

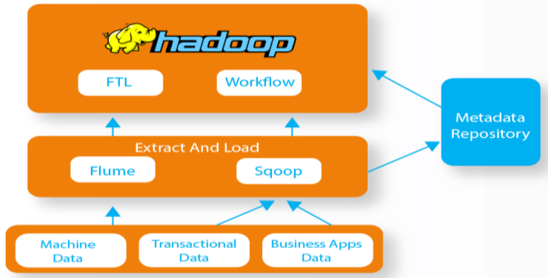
Hadoop is a framework basically designed to handle a large volume of data both structured and unstructured

HDFS

Hadoop Distributed File System is a framework designed to manage huge volumes of data in a simple and pragmatic way. It contains a vast amount of servers and each stores a part of file system

In order to secure Hadoop, configure Hadoop with the following aspects

- Authentication:
    - Define users
    - Enable Kerberos in Hadoop
    - Set-up Knox gateway to control access and authentication to the HDFS cluster
- Authorization:
    - Define groups
    - Define HDFS permissions
    - Define HDFS ACL's
- Audit:
  Enable process execution audit trail
- Data protection:
  Enable wire encryption with Hadoop



Hadoop HDFS List File Commands	Tasks
hdfs dfs –ls /	Lists all the files and directories given for the hdfs destination path
hdfs dfs –ls –d /hadoop	This command lists all the details of the Hadoop files
hdfs dfs –ls –R /hadoop	Recursively lists all the files in the Hadoop directory and al sub directories in Hadoop directory
hdfs dfs –ls hadoop/ dat*	This command lists all the files in the Hadoop directory starting with 'dat'

Hdfs basic commands	Tasks
hdfs dfs -put logs.csv /data/	This command is used to upload the files from local file system to HDFS
hdfs dfs -cat /data/logs.csv	This command is used to read the content from the file
hdfs dfs -chmod 744 /data/logs.csv	This command is used to change the permission of the files
hdfs dfs -chmod –R 744 /data/logs.csv	This command is used to change the permission of the files recursively
hdfs dfs -setrep -w 5 /data/logs.csv	This command is used to set the replication factor to 5
hdfs dfs -du -h /data/logs.csv	This command is used to check the size of the file
hdfs dfs -mv logs.csv logs/	This command is used to move the files to a newly created subdirectory
hdfs dfs -rm -r logs	This command is used to remove the directories from Hdfs
stop-all.sh	This command is used to stop the cluster
start-all.sh	This command is used to start the cluster
Hadoop version	This command is used to check the version of Hadoop
hdfs fsck/	This command is used to check the health of the files
Hdfs dfsadmin –safemode leave	This command is used to turn off the safemode of namenode
Hdfs namenode -format	This command is used to format the NameNode
hadoop [--config confdir]archive -archiveName NAME -p	This command is used to create a Hadoop archieve
hadoop fs [generic options] -touchz <path> ...	This is used to create an empty files in a hdfs directory
hdfs dfs [generic options] -getmerge [-nl] <src> <localdst>	This is used to concatenate all files in a directory into one file
hdfs dfs -chown -R admin:hadoop /new-dir	This is used to change the owner of the group

Commands	Tasks
yarn	This command shows the yarn help
yarn [--config confdir]	This command is used to define configuration file
yarn [--loglevel loglevel]	This can be used to define the log level, which can be fatal, error, warn, info, debug or trace
yarn classpath	This is used to show the Hadoop classpath
yarn application	This is used to show and kill the Hadoop applications
yarn applicationattempt	This shows the application attempt
yarn container	This command shows the container information
yarn node	This shows the node information
yarn queue	This shows the queue information

MapReduce

MapReduce is a framework for processing parallelizable problems across huge datasets using a large number of systems referred as clusters. Basically, it is a processing technique and program model for distributed computing based on Java

Mahout

Apache **Mahout** is an open source algebraic framework used for data mining which works along with the distributed environments with simple programming languages

Components of MapReduce

PayLoad: The applications implement Map and Reduce functions and form the core of the job

MRUnit: Unit test framework for **MapReduce**

Mapper: Mapper maps the input key/value pairs to the set of intermediate key/value pairs

NameNode: Node that manages the **HDFS** is known as **namednode**

DataNode: Node where the data is presented before processing takes place

MasterNode: Node where the **jobtrackers** runs and accept the job request from the clients

SlaveNode: Node where the Map and Reduce program runs

JobTracker: Schedules jobs and tracks the assigned jobs to the task tracker

TaskTracker: Tracks the task and updates the status to the **job tracker**

Job: A program which is an execution of a Mapper and Reducer across a dataset

Task: An execution of Mapper and Reducer on a piece of data

Task Attempt: A particular instance of an attempt to execute a task on a **SlaveNode**

Commands used to interact with MapReduce	
Commands	**Tasks**
hadoop job -submit <job-file>	used to submit the Jobs created
hadoop job -status <job-id>	shows map & reduce completion status and all job counters
hadoop job -counter <job-id> <group-name><countername>	prints the counter value
hadoop job -kill <job-id>	This command kills the job
hadoop job -events <job-id> <fromevent-#> <#-of-events>	shows the event details received by the job tracker for given range
hadoop job -history [all] <jobOutputDir>	Prints the job details, and killed and failed tip details
hadoop job -list[all]	This command is used to display all the jobs
hadoop job -kill-task <task-id>	This command is used to kill the tasks
hadoop job -fail-task <task-id>	This command is used to fail the task
hadoop job -set-priority <job-id> <priority>	Changes and sets the priority of the job
HADOOP_HOME/bin/hadoop job -kill <JOB-ID>	This command kills the job created
HADOOP_HOME/bin/hadoop job -history <DIR-NAME>	This is used to show the history of the jobs

Important commands used in MapReduce	
Usage: mapred [Generic commands] <parameters>	
Parameters	**Tasks**
-input directory/file-name	Shows Inputs the location for mapper
-output directory-name	Shows output location for the mapper
-mapper executable or script or JavaClassName	Used for Mapper executable
-reducer executable or script or JavaClassName	Used for reducer executable
-file file-name	Makes the mapper, reducer, combiner executable available locally on the computing nodes
-numReduceTasks	This is used to specify number of reducers
-mapdebug	Script to call when the map task fails
-reducedebug	Script to call when the reduce task fails

intelliPaat

FURTHERMORE:
Big Data Hadoop Certification Training Course

Ontology Syntax and Semantics Summary

Ontologies

- Formal conceptualization of a domain of interest.
- Logical theories making knowledge machine-processable.
- Defines terminology and semantic relationships between terms.

Syntax: Description Logics

- **Basic building blocks**:
  - Atomic concepts (unary predicates) like `Mother`, `Sister`.
  - Atomic roles (binary predicates) such as `hasChild`, `isMarriedTo`.
  - Individuals are constants like `alice`, `bob`.
- **Complex concepts**:
  - Constructed using operators like ¬, ⊓ (and), ⊔ (or), ∃ (exists).
  - E.g., `Mother ⊔ Father` (mothers or fathers), `Mother ⊓ ¬∃hasChild.Male` (mothers who don't have any male child).
- **TBox (Terminological Box)**:
  - Specifies general knowledge about the domain.
  - Defines concepts and roles and relationships between them.
- **ABox (Assertional Box)**:
  - Contains specific facts about individuals.

Semantics: Interpretation of DL

- **Declarative model-theoretic semantics**:
  - An interpretation maps terms to entities.
  - Satisfaction of TBox and ABox assertions within a model.
  - E.g., `Mother ⊑ Parent` (all mothers are parents), `hasParent ⊑ hasChild-` (if x has parent y then y has child x).
- **Semantics in practice**:
  - A model satisfies the ontology if it satisfies all axioms and assertions.
  - Entailment (`T |= α`) indicates that every model of T satisfies axiom α.

Summary

- Ontologies are formal representations of knowledge.
- They are composed of a TBox and an ABox.
- Description Logics provide a formal syntax and semantics for ontologies.
- DL allows for automated reasoning about ontologies.

University Domain Ontology Example

- **Classes**: Student, Course, Instructor
- **Subclasses**:
  - UndergraduateStudent ⊑ Student
  - GraduateStudent ⊑ Student
- **Relations**:
  - takesCourse: Student → Course
  - teachesCourse: Instructor → Course
- **Properties**:
  - hasName: Student → String
  - hasID: Student → Integer

Example

- TBox: GraduateStudent ⊑ Student

- ABox: hasID(Alice, 12345), takesCourse(Alice, AI101)

Relational Schema Overview

Definition

- A **Relational Schema** is a blueprint of a database that outlines the way data is organized into tables.

Components

- **Tables/Relations**: Collection of related data entries which consists of columns and rows.
- **Attributes**: Columns in a table, each representing a data point.
- **Tuples**: Rows in a table, representing records.

Constraints

- **Primary Key**: A field or a combination of fields that uniquely identifies a tuple within a table.
- **Foreign Key**: An attribute in one table that links to the primary key in another table.
- **Integrity Constraints**: Rules that ensure the reliability of the data (e.g., NOT NULL, UNIQUE, CHECK).

Operations

- **SQL (Structured Query Language)**: Used to perform operations on the data stored in a database.
- **ACID Properties**: Ensures reliable processing of database transactions (Atomicity, Consistency, Isolation, Durability).

SQL Commands

- **DDL (Data Definition Language)**: Defines schema components (e.g., `CREATE`, `DROP`).
- **DML (Data Manipulation Language)**: Manages data within schema objects (e.g., `SELECT`, `INSERT`, `UPDATE`, `DELETE`).

Remember to ensure that your data model and SQL commands align with the relational schema to maintain data integrity and efficiency.

Bookstore Relational Schema Example

Tables

- Books (ISBN, Title, Author, Price)
- Customers (CustomerID, Name, Email)
- Orders (OrderID, CustomerID, ISBN, Quantity, OrderDate)

Primary Keys

- Books: ISBN
- Customers: CustomerID
- Orders: OrderID

Foreign Keys

- Orders: CustomerID references Customers(CustomerID)
- Orders: ISBN references Books(ISBN)

Example Data

- Books: ('978-3-16-148410-0', 'Database Systems', 'C.J. Date', 85.00)

- Customers: (1001, 'Jane Doe', 'jane.doe@example.com')

- Orders: (5001, 1001, '978-3-16-148410-0', 1, '2023-01-01')

Knowledge Graphs and Data Access

Knowledge Graphs

- A knowledge graph represents interlinked descriptions of entities, real-world objects, events, situations, or abstract concepts.
- Knowledge Graphs are less formally semantic than ontologies but contain large volumes of factual information.

RDF (Resource Description Framework)

- RDF graph: a set of triples in the form `(subject, predicate, object)`.
- Example: `Ulm AlbertEinstein "1879-03-14"^^xsd:date`

SPARQL

- A standard query language for RDF data.
- Example Query:

```
SELECT ?title ?author
WHERE {
    { ?book dc:title ?title .
      ?book dc:creator ?author }
}
```

Inverted Index Construction Summary

Key Concepts

- **Inverted Index**: A data structure used to map content to its location within a database, document, or a set of documents.
- **Efficiency**: It's crucial for quickly locating data without having to search every row in a database.

Process Overview

1. **Text Preprocessing**: Normalizing text data by removing punctuation, converting to lowercase, etc.
2. **Tokenization**: Splitting text into tokens (usually words).
3. **Stemming/Lemmatization**: Reducing words to their base or root form.
4. **Stop Word Removal**: Eliminating common words that add little value to the index.

Example

- Documents: `D1: "The quick brown fox"`, `D2: "A quick brown dog"`.
- Preprocessed: `D1: ["quick", "brown", "fox"]`, `D2: ["quick", "brown", "dog"]`.

- Inverted Index:
  - `quick`: {D1, D2}
  - `brown`: {D1, D2}
  - `fox`: {D1}
  - `dog`: {D2}

Considerations

- **Storage**: Consider the cost of storing the index and the frequency of updates.
- **Scalability**: For large datasets, distributed systems and batch processing are essential.

Applications

- **Search Engines**: Allow quick keyword-based queries over large datasets.
- **Database Management**: Enhance the speed of query processing.

Advanced Databases: Exam reference solution

Pierre Senellart

14 December 2022

A first version of this reference solution has been produced by GPT-4, which has then been manually verified, corrected, modified. Because of this, answers tend to be long and verbose – this is suitable for a reference solution, but it is not expected that students write such detailed answers. Much more concise answers are fine. For many questions, there is not one valid answer: there are many ways to approach the problem.

1. (1 point)

   To estimate the amount of data required to store 50 years of TV programs, we can make the following assumptions:

   - There are 100 TV channels.
   - Each channel schedules 20 programs per day, on average, for 365 days per year (ignoring leap years), for a total of 7 300 programs per year per channel.
   - Each program has meta-information that takes an average of 10 000 bytes to store.
   - Each hour of archived video, including audio and subtitle tracks, takes 10^9 bytes to store.

   Based on these assumptions, we can estimate the amount of data required to store 50 years of TV programs:

   Meta-information: 100 channels $\times$ 7 300 programs/year $\times$ 10 000 bytes/program $\times$ 50 years = 3.65×10^{11} bytes, or 365 gigabytes.

   Video information: 100 channels $\times$ 7 300 programs/year $\times$ 1 hour/program $\times$ 10^9 bytes/hour $\times$ 50 years = 3.65×10^{16} bytes, or 36.5 petabytes.

   To store and manage 365 gigabytes of meta-information, a relatively small storage system could be used, such as a single high-capacity hard drive or a cloud-based storage service. The data could also be backed up using a redundant array of independent disks (RAID) system or a cloud-based backup service.

   However, storing and managing 36.5 petabytes of video information would require a much more complex and expensive storage infrastructure. A high-capacity storage system, such as a network-attached storage (NAS) or a storage area network (SAN), would be needed. The system would likely consist of many high-capacity hard drives connected to a large amount of nodes (of the order of 1000), configured in a RAID system for data redundancy and protection against disk failure. Additionally, a backup system would be necessary, such as an off-site backup or a cloud-based backup service, to ensure data recovery in case of disasters.

   Given the large amount of data involved, the hardware required to store and manage it would be expensive and require careful planning and maintenance to ensure its reliability and longevity. The archival institution would need to consider factors such as power consumption, cooling, and physical space requirements when designing the storage system.

2. (3 points)

   Here is a proposed schema for storing all meta-information about the TV programs:

   a) Programs
      - id (integer, primary key)
      - title (text)
      - type (text)
      - summary (text)
      - duration (integer)

   b) Persons
      - id (integer, primary key)
      - name (text)

   c) ProgramPersons
      - program_id (integer, foreign key to Programs)
      - person_id (integer, foreign key to Persons)
      - role (text)

   d) Series
      - id (integer, primary key)
      - title (text)
      - description (text)
      - type (text)

   e) Episodes
      - id (integer, primary key, foreign key to Programs)
      - series_id (integer, foreign key to Series)
      - season_number (integer)
      - episode_number (integer)

   f) ProgramSchedulings
      - program_id (integer, foreign key to Programs)
      - tv_channel (text)
      - start_datetime (datetime)
      - end_datetime (datetime)

   This schema allows for the storage of information about individual programs, the persons involved in their creation, series, episodes, and schedulings. Each table has a primary key column (id) that uniquely identifies each row, and foreign key constraints are used to link the tables together.

   Storing this amount of meta-information is feasible in a regular database management system such as Oracle or PostgreSQL, as the total amount of data is relatively small (365 gigabytes). The schema is designed to be efficient and scalable, allowing for easy retrieval of information and the addition of new data as needed.

   However, storing the video information is a much more challenging task due to its large size (36.5 petabytes). It would likely require a specialized storage system and data management approach, such as a distributed file system or a cloud-based storage solution, rather than a traditional database management system. The video files could be stored on separate storage devices and linked to the meta-information using a unique identifier or filename.

3. (1 point)

   Assuming that the Fort Boyard game show is part of a series in the Episodes table, and that the ProgramPersons table links programs to persons involved:

```sql
SELECT person_id, name, MIN(start_datetime) AS earliest_airdate
FROM ProgramPersons
JOIN Persons ON person_id = Persons.id
JOIN Programs ON program_id = Programs.id
JOIN Episodes ON Programs.id = Episodes.id
JOIN ProgramSchedulings ON Programs.id = ProgramSchedulings.program_id
WHERE Episodes.title = 'Fort Boyard'
GROUP BY person_id, name
```

   This query retrieves the roles and names of all participants involved in the Fort Boyard game show, as well as the earliest airdate for each episode they participated in. It does this by joining the relevant tables together and filtering the results based on the Fort Boyard episode title. The MIN() function is used to get the earliest airdate for each episode.

4. (1 point)

   The number of updates that would typically be issued on a database containing information about TV programs would depend on the specific use case and the level of activity of the institution. However, in general, it is likely that updates would be infrequent compared to the amount of read operations, as the majority of the data is historical and unlikely to change. The updates could include new program information, changes to scheduling information, and updates to the participants or creators of programs. The order of magnitude of the number of updates would be 2 000 per day, as this is the number of programmings per day – this represents a very low bandwidth of updates.

   A storage solution that implements strong ACID guarantees may not be strictly required for such an application, as the amount of updates is likely to be relatively low. However, ACID guarantees can provide important benefits in terms of data consistency, durability, and isolation, which can be critical for an archival institution. For example, if the institution needs to ensure that all data is accurately recorded and that updates are made in a safe and consistent way, then a storage solution that provides strong ACID guarantees may be necessary.

   Additionally, if the institution needs to perform complex queries or generate reports that involve aggregating or joining data from multiple tables, ACID guarantees can help ensure that the results are accurate and consistent. Finally, if the institution is subject to regulations or requirements that mandate strong data consistency and durability, a storage solution that implements ACID guarantees may be necessary to meet those requirements.

5. (1 point)

   When using BigTable/HBase to store the video information, one way to organize the data into the HBase data model is as follows:

   Key: A unique identifier for each video, typically the program identifier as stored in the meta-information database.

   Column families: One column family for the video file, one for soundtracks and one for subtitles.

   Columns: A column for each video file format and a column for each language or subtitle tracks

For example, the HBase table could be named "VideoData" and have the following column families:

- `"Video"`: contains columns for the original video file and any other video formats, such as compressed versions or versions in different resolutions. `"Audio"`: contains columns for each soundtrack language, with each column storing the audio data in that language. `"Subtitles"`: contains columns for each subtitle language, with each column storing the subtitle data in that language.

To determine a reasonable number of computers/nodes in a BigTable cluster storing this data, we need to consider the amount of data to be stored and the expected read and write throughput. Given that we need to store 36.5 petabytes of data and that video files are typically large, we would need a large number of nodes to achieve reasonable performance and redundancy. A reasonable estimate could be in the range of 100-10000 nodes, depending on the specific requirements and performance goals of the system. However, the exact number of nodes required would need to be determined through testing and benchmarking with the actual data and workload.

6. (1 point)

When using a key-value store such as a distributed hash table (DHT) to store the video information, one way to organize the data is as follows:

Key: A unique identifier for each video, typically the program identifier as stored in the meta-information database.

Value: A data structure containing information about the video file, including the video data and any soundtrack and subtitle tracks.

Alternatively, the key could be composite, with the program identifier and a part corresponding to the different parts of the video file: video, subtitle or audio track for a specific language.

The specific format of the value would depend on the requirements and capabilities of the DHT system used. For example, if the DHT supports storing large values, the value could contain the entire video file and associated tracks. Alternatively, if the DHT has size limitations on the value, the value could contain references to the actual video file stored in a separate storage system.

To determine a reasonable number of nodes in a DHT cluster storing this data, we need to consider the amount of data to be stored, the expected read and write throughput, and the requirements for redundancy and fault tolerance. A reasonable estimate could be in the range of 100-10000 nodes, depending on the specific requirements and performance goals of the system. However, the exact number of nodes required would need to be determined through testing and benchmarking with the actual data and workload. Additionally, the DHT system would need to be designed with replication and consistency mechanisms to ensure data durability and consistency in the face of node failures and network partitions.

7. (1.5 point)

Here is an example MapReduce program (in pseudo-code) that computes representative images (thumbnails) for every video stored in HBase, and stores the result back into HBase:

```
Mapper:
Input: video key-value pairs from HBase
  (key = unique identifier for the video, value = video file data)
Output: video key-thumbnail value pairs
```

```
  (key = same as input, value = thumbnail image data)

def map(key, value):
    # Load video data
    video_data = load_video_data(value)
    # Compute thumbnail image
    thumbnail = compute_thumbnail(video_data)
    # Output key-thumbnail pair
    yield key, thumbnail

Reducer:
Input: video key-thumbnail value pairs from Mapper
  (key = same as input, value = thumbnail image data)
Output: video key-thumbnail value pairs to HBase
  (key = same as input, value = thumbnail image data)

def reduce(key, values):
    # Store thumbnail image data in HBase
    store_thumbnail_in_hbase(key, values[0])
    # Output key-thumbnail pair
    yield key, values[0]

Main program:
# Set up HBase input and output
hbase_input = create_hbase_input()
hbase_output = create_hbase_output()

# Set up MapReduce job
job = create_mapreduce_job(hbase_input, hbase_output)
job.setMapperClass(Mapper)
job.setReducerClass(Reducer)
job.setOutputKeyClass(Text)
job.setOutputValueClass(BytesWritable)

# Run MapReduce job
job.waitForCompletion(true)
```

The program uses the HBase input format to read video data from HBase and the HBase output format to store thumbnail data back into HBase. The Mapper loads each video file, computes a thumbnail image, and outputs the key-value pair with the video key and thumbnail data. The Reducer stores the thumbnail data back into HBase and outputs the key-value pair. The Main program sets up the MapReduce job and runs it.

Note that the functions `load_video_data`, `compute_thumbnail`, and `store_thumbnail_in_hbase` are not provided here and would need to be implemented based on the specifics of the video data and thumbnail generation requirements.

An important observation is that for such an application, the reducer does very little: this is an *embarassingly parallel* task, that does not require any form of aggregation.

8. (4 points)

RDF and SPARQL To represent meta-information about a specific episode of a TV drama using RDF, we could use the following ontology:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix schema: <http://schema.org/> .
@prefix tvd: <http://example.com/tv-drama#> .

tvd:Episode123 a schema:TVEpisode ;
    schema:name "Episode 123" ;
    schema:partOfSeason tvd:Season2 ;
    schema:partOfSeries tvd:TVSeries1 ;
    schema:datePublished "2002-04-01"^^xsd:date ;
    schema:duration "PT45M"^^xsd:duration ;
    schema:description "Description of Episode 123" ;
    schema:actor tvd:Actor1, tvd:Actor2 ;
    schema:director tvd:Director1 ;
    schema:producer tvd:Producer1 ;
    schema:publication tvd:Broadcast1 ;
    rdf:type tvd:ComedyShow .

tvd:Season2 a schema:TVSeason ;
    schema:name "Season 2" ;
    schema:partOfSeries tvd:TVSeries1 .

tvd:TVSeries1 a schema:TVSeries ;
    schema:name "TV Series 1" ;
    schema:description "Description of TV Series 1" .
```

In this example, we have an Episode resource with properties such as its name, duration, description, actors, director, producer, and the date it was published. It is also part of a Season and a TV Series.

To retrieve all different types of programs scheduled on January 1st, 2000, at 00:00, using SPARQL, we could use the following query:

```
PREFIX schema: <http://schema.org/>
PREFIX tvd: <http://example.com/tv-drama#>

SELECT DISTINCT ?type
WHERE {
  ?program schema:datePublished "2000-01-01T00:00:00"^^xsd:dateTime ;
           rdf:type ?type .
}
```

This query selects all programs that have a publication date of January 1st, 2000, at 00:00, and retrieves their RDF type. The DISTINCT keyword ensures that only unique types are returned.

XML and XQuery To represent meta-information about a specific episode of a TV drama using XML, we could use the following schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<tv-drama xmlns="http://example.com/tv-drama">
```

```xml
  <episode type='ComedyShow'>
    <name>Episode 123</name>
    <season>Season 2</season>
    <series>TV Series 1</series>
    <datePublished>2002-04-01</datePublished>
    <duration>PT45M</duration>
    <description>Description of Episode 123</description>
    <actors>
      <actor>Actor 1</actor>
      <actor>Actor 2</actor>
    </actors>
    <director>Director 1</director>
    <producer>Producer 1</producer>
    <publication>
      <broadcast>Channel 1</broadcast>
      <start>2002-04-01T19:00:00</start>
      <end>2002-04-01T19:45:00</end>
    </publication>
  </episode>
  <season>
    <name>Season 2</name>
    <series>TV Series 1</series>
  </season>
  <series>
    <name>TV Series 1</name>
    <description>Description of TV Series 1</description>
  </series>
</tv-drama>
```

In this example, we have an episode element with sub-elements such as its name, duration, description, actors, director, producer, and the date it was published. It is also part of a season and a TV series. The publication element contains information about the TV channel, start, and end times.

To retrieve all different types of programs scheduled on January 1st, 2000, at 00:00, using XQuery, we could use the following query:

```xquery
declare namespace td = "http://example.com/tv-drama";

distinct-values(
  for $program in
    //td:publication[td:start = xs:dateTime('2000-01-01T00:00:00')]
  return $program/parent::td:episode/@type
)
```

This query selects all publication elements that have a start time of January 1st, 2000, at 00:00, and returns the unique values of their parent episode's type attribute. The distinct-values() function ensures that only unique types are returned.

9. (1.5 point)

   For the specific use case of archival of data and metadata about TV programmings, each of the three technologies has specific advantages and disadvantages.

Advantages of RDF+SPARQL:

- RDF is well-suited for representing metadata, which can be highly diverse and unstructured, allowing different metadata fields to be easily added as needed.
- RDF's flexible schema-less data model makes it well-suited for handling evolving metadata over time, as new fields can be easily added or modified.
- SPARQL's graph querying capabilities are well-suited for querying complex relationships between TV programs, their schedules, and their associated metadata.

Disadvantages of RDF+SPARQL:

- RDF can be less efficient for storing and indexing large amounts of metadata compared to more optimized data structures, such as columnar stores.
- SPARQL queries can be complex and difficult to optimize for large datasets, requiring specialized query planning and indexing techniques.
- The lack of standardization in RDF vocabularies and ontologies can make it difficult to integrate metadata from different sources.

Advantages of XML+XQuery:

- XML's hierarchical structure makes it well-suited for representing metadata about TV programs, which can have complex and nested relationships.
- XQuery provides a powerful querying language for hierarchical data, which can be well-suited for querying metadata about TV programs.
- XML has been widely used in the media industry for storing metadata about TV programs, making it a familiar and established technology in this domain.

Disadvantages of XML+XQuery:

- XML can be verbose and inefficient for storing large amounts of metadata, which may require specialized optimization techniques, such as XML compression.
- XQuery may not be as well-supported by third-party tools and technologies compared to other querying languages, such as SQL or SPARQL.
- XML's hierarchical structure may not be well-suited for representing complex relationships between TV programs, such as social media interactions or viewer engagement.

Advantages of Graph Databases+Cypher:

- Graph databases are well-suited for representing complex relationships between TV programs, their schedules, and their associated metadata.
- Cypher provides a user-friendly querying language that can be well-suited for non-experts, such as archivists or media analysts.
- Graph databases can be highly performant for querying complex relationships between TV programs and their metadata, which can be useful for analytics or search applications.

Disadvantages of Graph Databases+Cypher:

- Graph databases may require specialized indexing and storage mechanisms to optimize querying performance, which may require more expertise compared to traditional relational databases.
- Graph databases may not be well-suited for representing structured data, such as tabular data with many rows and columns, which may be a limitation for certain use cases.

- Graph databases may not be as widely adopted in the media industry compared to other technologies, which may limit the availability of expertise and third-party tools.

10. (1 point)

    Ontology-based querying involves using a formal ontology to describe and query data. In this situation, the ontology would define a taxonomy of TV program types, including the specific characteristics that define each type, such as the topic or theme of the program. An ontology-based query would then use this ontology to identify TV programs that have specific characteristics, such as a particular topic or theme.

    Imagine for instance that there is a type “Comedy” that encompasses subconcepts in the ontology such as “DarkComedy” or “SitCom”. When a user queries for a “Comedy” drama, she should be able to obtain results that are marked as “DarkComedy”, as the ontology specificies “DarkComedy” is a subtype of “Comedy”. This requires ontology-based querying, as disregarding the ontology would not produce the expected result.

11. (2 points)

    To design an architecture for cataloging and archiving TV programs over 50 years, I would propose a combination of technologies for storing and querying both the meta-information and the video information.

    For the meta-information, I would use RDF+SPARQL. RDF+SPARQL would allow for flexible and schema-less data modeling, making it easy to add new metadata fields as needed and integrate metadata from different sources. SPARQL’s graph querying capabilities would also be useful for querying complex relationships between TV programs, their schedules, and their associated metadata.

    For the video information, I would use a distributed storage solution such as Hadoop/HDFS or Amazon S3, as well as a distributed computing framework such as Apache Spark to process and analyze the video data. Hadoop/HDFS or Amazon S3 would allow for scalable and cost-effective storage of large amounts of video data, while Spark would provide a distributed computing framework for processing and analyzing the video data at scale. I would also consider using specialized video processing libraries and tools, such as FFmpeg or OpenCV, for extracting metadata and generating representative images or thumbnails of the videos.

    Overall, this architecture would allow for flexible and scalable storage and querying of both the meta-information and video information, using a combination of established and widely adopted technologies that are well-suited for the specific requirements of cataloging and archiving TV programs over 50 years.

12. (2 points)

    An approach to identifying the original source of the data is to use data provenance. Data provenance is a technique for tracking the complete history of how data is created, modified, and propagated through a system. In this case, data provenance could be used to track the original source of the data and any subsequent modifications that were made to it.

    To implement data provenance in this use case, each time a TV production company or TV channel enters data into the database, a provenance record would be created that captures the source of the data, the timestamp of the entry, and any other relevant metadata about the data entry. Each time the data is subsequently modified or updated, a new provenance record would be created that captures the changes made, along with the identity of the person who made the changes and the timestamp of the update.

When issuing a query on the meta-information, users could access the provenance records associated with each metadata record, allowing them to trace the complete history of how the data was created and modified. Users could also use provenance to identify the original source of the data and any subsequent modifications made to it. For example, a query for all program schedulings could include a filter criterion for the provenance record, allowing users to restrict the results to schedulings that were entered by a specific TV channel or TV production company.

Obtaining provenance for the result of a complex query involves tracing the sources of the data used in the query and the operations performed on that data during the query execution. In other words, we need to track the lineage of the data used in the query, from its original sources through any intermediate steps to the final output.

One approach to obtaining provenance for a complex query is to use a provenance tracking system that automatically captures the lineage of data as it moves through the system. For example, the system could track each data item and its associated metadata, including its source, transformations applied to it, and the outputs produced by those transformations. The system could then use this lineage information to generate a complete provenance record for each query result.