R Language: A Powerful Tool for Taming Big Data

Norman Matloff[1], Clark Fitzgerald[2], and Robin Yancey[3]
[1]Department of Computer Science, University of California, Davis, CA, USA
[2]Department of Statistics, University of California, Davis, CA, USA
[3]Department of Electrical and Computer Engineering, University of California, Davis, CA, USA

Definitions

The R language (R Core Team 2017; Chambers 2008; Matloff 2011) is currently the most popular tool in the general data science field. It features outstanding graphics capabilities and a rich set of more than 10,000 library packages to draw upon. (Other notable languages in data science are Python and Julia. Python is popular among those trained in computer science. Julia, a new language, has as top priority producing fast code.) Its interfaces to SQL databases and the C/C++ language are first rate. All of this, along with recent developments regarding memory issues, makes R well poised as a highly effective tool in Big Data applications. In this chapter, the use of R in Big Data settings will be presented.

It should be noted that Big Data can be "big" in one of two ways, phrased in terms of the classical $n \times p$ matrix representing a dataset:

- **Big-n:** Large number of data points.
- **Big-p:** Large number of variables/features.

Both senses will come into play later. For now, though, back to R. Some general information about the language will be presented first, as foundation for the Big Data aspects.

Overview

In terms of syntax, R, along with Python, C/C++, and many others, is ultimately a descendant of ALGOL, and thus a programmer in one of those languages can quickly pick up at least a rough "reading knowledge" of R.

As with Python, R is an *interpreted* language, meaning it is not translated to machine code, unlike the C/C++ language. The interpreted nature of R brings up possible performance issues, a topic to be discussed in section "Uniprocessor Performance Issues".

From a programming style point of view, R (to various degrees) follows the *object-oriented* and *functional programming* philosophies. Some computer scientists believe that they lead to clearer, safer code, and this has influenced the design of R. This in turn will have implications for all R users, as will be explained.

S. Sakr, A. Y. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-77525-8

Vectors and *matrices* are similar to one- and two-dimensional arrays in C. A *list* is like a vector, but with possibly different modes. A *data frame* looks like a matrix, but its columns can be of different modes, e.g., numeric in some, character in others, and logical in still others. These structures are intrinsic parts of the R language, as opposed to add-ons in the case of Python.

Subsetting is a major operation in R. For instance,

```
m[c(1,4,5),]
```

is the submatrix of the matrix **m** consisting of rows 1, 4, and 5 of that matrix.

A seldom cited but widely used and very handy feature of R is the ability to set names of various data elements. For an example of how this can be useful, suppose one has a large data frame, and then split it into chunks of rows according to some criterion. The original row names will be retained:

```
> d <- data.frame(x=c(5,12,13), y=c(8,88,888))
> da <- d[c(1,3),]
> d
   x   y
1  5   8
2 12  88
3 13 888
> da
   x   y
1  5   8
3 13 888
> row.names(da)
[1] "1" "3"
```

Such information could be quite useful and is attained without adding an extra column in the data frame that may need to be excluded in subsequent statistical computations.

One of R's most lauded features is its ability to produce beautiful, highly expressive graphics, in a manner accessible to even nonspecialists. Base R graphics is used for simpler plots or for advanced applications operating at a more finely detailed level (Murrell 2011; Chang 2013). For higher-level applications, the **ggplot2** (Wickham 2016) and **lattice** (Sarkar 2008) packages are quite powerful and are widely used. Another notable graphics package is Plotly (Plotly Technologies Inc. 2015), which produces especially esthetically appealing figures. A quite usable R interface is available.

Uniprocessor Performance Issues

As mentioned, R is an interpreted language, which raises performance concerns. The chief remedy is *vectorization*, referring to the fact that vector operations should be used instead of loops.

Vectorization and R as a Functional Language

First, the functional language nature of R is reflected in the fact that, for instance, the **+** operator is actually a function. The expression **x+y**, for vectors **x** and **y**, is actually the function call **'+'(x,y)**. The key point is that function is written in C, not in R. (This should not be confused with the fact that R itself – meaning the R interpreter – is written in C.) Thus C-level speed is attained, which could be an issue for very long vectors.

By the way, <− is a function too. So, z <− x + y is actually

```
'<−'(z,'+'(x,y))
```

It turns out that many R expressions can be vectorized. For instance, the **ifelse()** function vectorizes the classical if–then–else construct:

```
> x <− 1:3
> y <− c(5,12,13)
> ifelse(y > 8,x,x+1)
[1] 2 2 3
```

Interfacing R to C/C++

But in some R applications, vectorization is not enough; the general speed of C must be obtained directly, i.e., entire R functions must be written in C. For instance, **dplyr**, a popular R package for manipulation of data frames, is written partly in C, and **data.table**, an extremely fast (but compatible) alternative to data frames (section "Data Input/Output, etc. with data.table"), is based largely on C++.

This is a standard approach in the use of scripting languages such as R and Python. Typically only a portion of one's code requires high performance. It thus makes sense to write that portion in C/C++ while retaining the convenience and expressiveness of R/Python for most of one's code.

R has two main functions, **.C()** and **.Call()**, for calling C/C++ functions from R code. They are fairly easy to use, and many R programmers use another popular R package, **Rcpp**, that aims to further simplify the R/C interface process (Eddelbuettel 2013).

Interfaces that allow R code and Python to call each other are also available, such as the **rpy2** package.

R and Big Data

The sheer size of Big Data calls for parallel computation, either on multicore machines or clusters. (Define a *physical* cluster to be a set of independent machines connected via a network. Later, *virtual* clusters will be introduced, which will consist of a number of R invocations running independently. They may be running on a physical cluster, a multicore machine, or a combination of the two.)

Note carefully that the issue is not only one of computation time but also of memory capacity. The latter point is just as important as the former; an application may be too large for a single machine, but if the data is broken into chunks and distributed to the nodes in a cluster, the application might be accommodated.

A detailed treatment of parallel computation in R is given in Matloff (2015). An overview is presented here.

Memory Issues

Earlier versions of R, 2.x.x, set a limit of $2^{31} - 1$ bytes for object size. For modern 64-bit machines, this limited vectors, for instance, to about 250 million numbers, which was not sufficient in certain Big Data contexts. R 3.x.x changed this limit to about 2^{52}, more than enough for even the biggest of Big.

Thus the constraint faced by most people in the R community is not in R limitation on object size, but in the memory sizes of their machines. R stores all objects in memory, so memory size may be an issue in R applications in Big Data.

The bigmemory Package

One solution is the **bigmemory** package (Kane et al. 2013), which offers programmers a choice between storing objects in memory or on disk, with the latter option being attractive on machines with only moderate amounts of RAM. The key point is that from the programmer's point of view, the data look like they are in memory.

Objects of class **bigmemory** must be of matrix type. A vector is represented as a one-row or one-column matrix and a scalar as a 1×1 matrix.

Recall that operators like **+** are actually implemented as functions. The array indexing operator [is another example of this. What **big.memory** does is replace R's **[** function by special-purpose C++ code that relays the requested data reference to either memory or disk, according to what the user's code originally requested.

Another advantage of **bigmemory** among many of its users is that it provides a workaround to R's "no side effects" policy, which comes from R's status as a functional language. What this means is that a function call should not change any of its arguments.

For example, the statement

```
sort(x)
```

will *not* change **x**. It returns the sorted version of **x**, but **x** itself doesn't change. If one wants that change, one needs to write

```
x <- sort(x)
```

Now consider the innocuous-looking statement

```
z[5] <- 8
```

As noted, assignment is a function call, in this to the function '[<–'. Under a no-side-effects policy, this would be implemented internally as

```
z <- '[<-'(z,5,8)
```

with the right-hand side returning a *new* copy of **z**. In a Big Data context, the creation of this new vector could be quite costly in execution time.

Recent versions of R try to avoid this to some extent, but by placing **z** in a **bigmemory** object, the problem is definitely avoided.

Other Ways to Circumvent Memory Size Problems

As noted, one solution to this problem is to exploit the fact that **bigmemory** can store an object on disk but have it appear to the programmer as if it were in memory. Two other methods are common:

- If one is running on a cluster, one can use the data in distributed form, with one chunk of the data at each cluster node. The chunks may fit into memory even if the full data does not.
- If the data is stored on disk in an SQL database, various R packages for interfacing to SQL are available, such as **RMySQL**.

In such settings, it may also be useful to then use a Software Alchemy approach; see sections "Distributed Computation" and "Software Alchemy". Or in the cases in which the application algorithm needs only sums, such as linear regression models, one can read in data chunk-by-chunk, updating the sums at each stage.

Threaded Code

The standard mechanism used for parallel computation by multicore programs in C/C++ is *threaded* code (Matloff 2015; Breshears 2009). Here several copies of one's code run simultaneously, sharing global data. That latter trait is key. For many algorithms, efficient parallelization requires shared-memory computation. If one has a multicore machine (or a manycore coprocessor, such as the Intel Xeon Phi) and the problem can fit into available memory, threaded programming of some kind is needed.

Roughly, here is how threads work. Say, for instance, the threads are executing some iterative algorithm. At the end of each iteration, each thread will need to know the results of the executions of the various threads, as this will determine the course of action in the next iteration. Accessing shared variables for this kind of action is the essence of threaded programming.

Threads systems must have some form of "lock" variables to avoid *race conditions*. That term refers to a situation in which two threads attempt to change a certain variable at approximately the same time, possibly resulting in incorrect results. A lock variable assures that only one thread at a time can access the sensitive variable.

Threads in C/C++

Say one wishes to find the sum of a very long array **x** of length **n**, with four threads. Each thread would have an ID, 0, 1, 2, or 3, stored in the variable **myID**. Each thread would have its own running sum, stored in **mySum**. The code would look something like this:

```
parfor (i = 0; i < n; i++) {
   if (i % 4 == ID) mySum +
   = x[i];
}
// lock the lock (not shown)
tot += mySum;
// unlock the lock (not shown)
```

Here "parfor" means that all the threads would execute the loop simultaneously. (The variables **myID** and **mySum** would be local to each thread, while **tot** would be global to all.)

Typically one does not write threaded code directly, opting instead to use a higher-level interface to threads, the most widely used being OpenMP (Breshears 2009). Intel's Threads Building Blocks (Reinders 2007) may produce faster code, but requires more programming skill (and patience), as it is more complex and uses C++ templates.

Threads in R

Scripting languages typically do not offer true threads programming. Python threads, for instance, must use something called the global interpreter lock, which prevents parallel execution of threads. Julia threads are, as of this writing, experimental.

R does not offer threading at all. However, one can still write threaded applications, in one of two ways:

- If one wants to limit one's code to R, a quasi-threading environment is provided by the **Rdsm** package (Matloff 2015). It runs on top of the **bigmemory** system discussed earlier, as well as atop the **parallel** package that is included with base R (section "partools").

  To see how this works, let's again consider our earlier example. **Rdsm**, in creating a shared variable **z** – in memory, not on disk – will save on disk information as to where in memory **z** resides. If one then has multiple invocations of R running simultaneously, they can all access that same memory location, and thus share **z**, thus attaining the essence of threaded code.

  In the array-summing example above, the programmer would call a **parallel** function, **splitIndices()**, to explicitly assign values of **i** to the various threads. Instead of an explicit loop, the code would be vectorized, looking something like this, using the built-in R function **sum()**:

```
myIndices <- splitIndices
(etc.)[[myID]]
mySum <- sum(x[1,myIndices])
# lock the lock variable
 (not shown)
totx[1,1] <- totx[1,1] + mySum
# unlock the lock variable
(not shown)
```

  Here **x** and **totx** are variables in shared memory.

- One can have one's R code access C/C++ code that does threading, say using OpenMP. In the above array-summing example, the C code could be used without change. (The R system file **R.h** must be included.)

  The **data.table** package is written largely in C++ and makes use of OpenMP.

  Another important example is the Basic Linear Algebra Subroutines (BLAS) library. Lots of BLAS libraries exist, and when building R one has the option of using one other than what is included in the R source code. One that is very fast is OpenBLAS, which again uses OpenMP to run threads.

Message-Passing Parallel Code in R

In addition to the shared-memory world view seen above, another paradigm of parallel computation is *message-passing*. Here, instead of having different processes communicate via shared variables, they explicitly send information to each other.

This approach is generally used on clusters rather than multicore machines, though many users do use it on the latter as well. In either case, it must be kept in mind that a major source of speed-sapping overhead is the time spent sending the messages, especially in the cluster case. For this reason, these approaches are generally efficient only in settings in which the time between network accesses is long. This in turn means the application is such that a large amount of computation is done between network accesses, a situation known as *coarse-grained parallelism*.

It should be repeated, though, that for many large problems, memory size is the overriding constraint. In such cases, a message-passing approach on a cluster may be the only feasible solution.

The "parallel" Package

A workhorse of parallel computation in R is the built-in **parallel** package. Its main virtue is its conceptual simplicity; users can readily write their own **parallel** code after seeing a few examples.

The package was adapted from the previous user-contributed packages **snow** and **multicore**. The focus here will be on the former case, which implements *scatter/gather* operations.

As with **Rdsm** (and **ddr** below), there will be many independent invocations of R. When the user starts R, let's refer to that as the "manager" invocation. It launches new invocations of R, termed "workers." One writes code to run on the manager that distributes data to the workers – the *scatter* phase – and sends the workers commands to execute on the data given them. They return results, which are *gathered* at the manager. The latter pieces them together to obtain the desired outcome.

Here is a simple example, again involving an array-summing operation, in parallel over the workers:

```
library(parallel)
cls <- makeCluster(2)
x <- c(5,12,13,8,88)
clusterExport(cls,'x')
clusterApply(cls, 1:2, function(i) myID <<- i)
clusterEvalQ(cls,library(parallel))
clusterEvalQ(cls,myIndices <-
   splitIndices(5,2)[[myID]])
Reduce(sum(clusterEvalQ(cls,sum(x[myIndices]))
```

All this runs on the manager. It first makes a virtual cluster of two workers, meaning two new invocations of R connected to the manager. It then creates an example vector **x** and "exports" it to the workers. (This is not very efficient, as each worker needs only part of **x**.) Next, it sets up an ID at each worker, by applying the given function to (1,2) at the workers. Then it loads the **parallel** package at each worker; the function **clusterEvalQ()** simply instructs each worker to execute the given code locally. Next it runs

```
clusterEvalQ(cls,myIndices
  <- splitIndices(5,2)[[myID]])
```

This instructs each worker to assign to **myIndices** as indicated. The call to **splitIndices()** returns

```
[[1]]
[1] 1 2

[[2]]
[1] 3 4 5
```

i.e., an R list whose first component is (1,2) and the other (3,4,5). The purpose of this is to have Worker 1 handle elements 1 and 2 of **x**, while Worker 2 will handle the rest. After execution of the above statement, the variable **myIndices** will have the value (1,2) at Worker 1 and (3,4,5) at Worker 2.

Finally, there is the **Reduce()** call. Inside is a call to **clusterEvalQ()**, which has each worker sum its portion of **x**. That call returns to the manager an R list consisting of the two sums, 17 and 109. **Reduce()** then repeatedly applies **sum()** to each element of that list, resulting in the full sum 126.

As can be seen, compared to the shared-memory paradigm, the programmer must do quite a bit of work just for this simple operation. This is typical of message-passing systems. Many programmers use the **foreach** package (Weston 2017) as a simpler, more convenient wrapper for **parallel** in situations in which the main goal is to parallelize a loop.

Rmpi

A major limitation of **parallel** is that communication is possible only between a worker and the manager. The **Rmpi** package (Yu 2014), based on the famous C/C++/FORTRAN message-passing library MPI (Nielsen 2016), is much more general, allowing direct communication from any worker to any other. For some applications, this can increase performance tremendously.

Rmpi is more complex to program (and to configure) and thus is too involved to present in further detail here.

Distributed Computation

One sees much in the press about the Hadoop and Spark frameworks. These have been proven quite effective on very large systems for simple tabulatory computations such as sums, counts, and data grouping. And they do have R interfaces for then, e.g., **sparklyr** (Luraschi et al. 2017).

However, a major drawback to these frameworks is that one essentially must do a global sort after each operation, whether needed or not, potentially quite a drain on speed. Though Spark is a major improvement over Hadoop, neither is well-suited to the more complex statistical and data-wrangling operations typical in data science. Here two alternatives are presented.

Both of the packages in this section operate with distributed data. For instance, say **z** is a data frame with 10 million rows, and one has 4 workers in a cluster in the above sense. One could store 2.5 million rows at each worker, and to the degree possible and as long as possible, the data is *kept* distributed in that manner, over the course of many data and statistical operations. In the above scatter/gather terms, one scatters but then avoids gathering for as long as possible. The goal, of course, is to avoid costly network communication delays. Informally let's call this policy "Leave It There" (LIT).

LIT is a very simple idea, yet a very powerful one.

partools

Both Hadoop and Spark are typically run on distributed file systems. The **partools** package (Matloff et al. 2017a), which runs on top of **parallel**, is predicated on the view that this is the best approach, but that the operation structures of Hadoop and Spark, with frequent network communication and disk read/writes, are unsuited for statistical/data science applications.

Again, consider a simple example of four workers. One might store the data in four files, **x.1**, **x.2**, and so on. The manager code would execute something like

```
fileread(cls,'x','x',1)
```

Here **cls** is the name of the virtual cluster; the basename of the distributed file is "x"; and one wishes the distributed data frame at the virtual cluster nodes to also be named "x."

Execution of the distributed program proceeds mainly with manager calls to **clusterEvalQ()**, instructing the workers to perform certain tasks. For instance, in a linear regression problem, the code at the manager might look like this:

```
clusterEvalQ(cls,convert factors to dummy variables)
clusterEvalQ(cls,replace NA values by means)
clusterEvalQ(cls,remove outliers, say by a "3 sigma" rule)
calm(cls,regression formula)
```

That last function call implements Software Alchemy, to be explained in section "Software Alchemy". It runs the R **lm()** linear model function at each cluster node and combines the results to obtain the overall estimated regression coefficients.

Note that the data, even after various operations have been performed, is *still* distributed, available for further distributed operations, again following the LIT philosophy. And at the end of the session, the user can save the modified distributed data frame to a distributed file.

Numerous utility functions are available, including ones to convert data between distributed and monolithic forms: **distribsplit()**, to convert a monolithic data frame to a distributed one, and **distribcat()**, to go in the opposite direction.

The package also includes a number of statistical and tabulatory functions to do non-LIT operations, such as **distribagg()**, which performs the R **aggregate()** function at each node and then combines appropriately.

In addition, **partools** offers direct point-to-point communication between cluster nodes, which greatly extends the ability to write fast parallel code. One of the applications of that facility is an implementation of the Hyperquicksort distributed sorting algorithm.

ddR

In designing a package for distributed computation, one desirable trait would be (at least near) compatibility with existing serial code, enabling software reuse. The **ddR** package (distributed data in R) satisfies this criterion, defining distributed versions of some R data structures, including arrays, lists, and data frames. Several algorithms have been implemented in this framework, such as generalized linear models and random forests. The package runs on top of **parallel** or another HP product, **distributedR**.

Here is an example of **ddR** code to initialize and subtract two distributed matrices.

```
library(ddR)
chunk <- 2
nc <- 5
n <- chunk * nc
x <- as.darray(matrix(rnorm(n)), psize = c(nc, 1))
y <- as.darray(matrix(rnorm(n)), psize = c(nc, 1))
setMethod("-", signature(e1 = "ParallelObj",
                         e2 = "ParallelObj"),
function(e1, e2) {
    dmapply(‘-‘, e1, e2, output.type = "darray",
            combine = "cbind")
})
delta <- x - y
as.vector(collect(delta))
```

The package uses R’s S4 object-oriented programming model to define operations on distributed objects, in this case subtraction. The S4 function **setMethod()** is shown above to define subtraction between two objects of class `ParallelObj`, creating a third object of that class.

Software Alchemy

Real-world Big Data applications on parallel platforms tend not to be “embarrassingly parallel,” due to inter-process communication costs. This can severely limit potential speedup.

Fortunately, for statistical/machine learning applications, this frequently can be resolved by a method also known as “Software Alchemy” (SA) in Matloff (2016). Here the data is divided up into chunks which are distributed to each available cluster node, and one averages all the results of the same computation from each process. In this way we “alchemically” transform the original problem into an embarrassingly parallel problem that produces a statistically equivalent result: The asymptotic covariance matrix of the SA estimator can be shown to be the same as that of the estimator based on the full set, i.e., it achieves the same accuracy. (It should be noted that the speed of convergence of the asymptotics may depend on p, the number of variables/features in the full dataset (Bühlmann et al. 2016).)

Theoretically, the speedup can be derived in time complexity terms. Suppose the complexity of the full algorithm is known to be $O(n^d)$, where n is the number of data points. Let r denote the number of worker threads. Then the SA complexity will be approximately $O[(n/r)^d] = O(n^d/r^d)$, a speedup of about r^d. If $d > 1$, this would be a superlinear speedup, a rarely seen event in the parallel computation world, but that has appeared experimentally in plots of the output using SA (Matloff 2016). (The same analysis shows, interestingly, that one obtains a speedup even in the serial case, for $d > 1$.)

Our experiments so far have shown SA to be useful for several widely used machine learning algorithms. For example, consider the famous Forest Cover dataset in the UCI Machine Learning Data Repository (Lichman 2017), consisting of half a million samples for classification into one of seven classes. Using the random forests algorithm, we can produce a speedup of 4.2 without loss in accuracy. This reduces computation time by over 10 min by just taking advantage of a standard four-core machine with six processes. (The machine used here has *hyperthreading*, which makes it perform in some settings as if it has eight cores.)

In the classification case, one must use a variation of SA. In predicting the class of a new case, each cluster node produces its own prediction. Then "voting" is used: The final predicted class is the one predicted the most often among the r cluster nodes.

The major advantages of SA are its simplicity and generality. One can use it to attain a good speedup on almost any statistical/machine learning algorithm.

Data Input/Output, etc. with data.table

As mentioned, **data.table** is an extension of the data frame construct (Dowle 2017). In Big Data contexts, the use of the **data.table** package is indispensable. For most operations – reading from/writing to disk, data grouping, and so on – it is much faster than the standard R counterparts.

Graphics

Though one might at first think, "The more data, the better," Big Data can present real problems in terms of graphics. With so many points to plot, there is a real risk of encountering the "black screen problem," with the points solidly filling major areas of the computer screen, thus rendering the graph meaningless. A number of techniques have been developed to deal with this (Unwin et al. 2007).

In addition, with even moderate values of p, the number of variables/features in our dataset, plotting the data in a visually interpretable way is challenging. Data can readily be visualized in the case $p = 2$, and possibly $p = 3$, but it is difficult for $p > 2$. One method to address this problem is *parallel coordinates* (Inselberg 2009). Here each data point is displayed as a segmented line connecting dots at heights given by the values of the variables in that data point.

Direct use of parallel coordinates with Big Data will typically lead to the "black screen problem." The **cdparcoord** package (Matloff et al. 2017b; Yang et al. 2017) aims to solve this problem by plotting only the most frequently appearing data tuples.

Conclusions

R is a very powerful tool for data science, developed *by* statisticians, *for* statisticians. It has truly excellent graphics packages, built-in matrix and linear algebra operations, several types of object-oriented programming models to choose from, and so on. The tremendous repository of contributed packages, CRAN, is a huge advantage by itself.

Care must be taken with speed and memory issues. These are handled by the use of vectorization and taking advantage of several good R facilities for parallel programming, as well as the availability of the **bigmemory** package. Properly used, R is a very strong tool for taming Big Data.

Cross-References

- ▶ Julia
- ▶ Python
- ▶ Parallel Graph Processing
- ▶ Parallel Processing with Big Data

References

Breshears C (2009) The art of concurrency: a thread monkey's guide to writing parallel applications. O'Reilly Media, Sebastopol
Bühlmann P, Drineas P, Kane M, van der Laan M (2016) Handbook of big data. Chapman & Hall/CRC handbooks of modern statistical methods. CRC Press, Boca Raton
Chambers J (2008) Software for data analysis: programming with R. Statistics and computing. Springer, New York
Chang W (2013) R graphics cookbook. Oreilly and associate series. O'Reilly Media, Sebastopol, CA
Dowle M (2017) Data analysis using data.table. https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html
Eddelbuettel D (2013) Seamless R and C++ integration with Rcpp. Use R! Springer, New York
Inselberg A (2009) Parallel coordinates: visual multidimensional geometry and its applications. Springer, New York
Kane MJ, Emerson J, Weston S (2013) Scalable strategies for computing with massive data. J Stat Softw 55(14):1–19
Lichman M (2017) UCI machine learning repository. http://archive.ics.uci.edu/ml
Luraschi J, Ushey K, Allaire J (2017) Sparklyr: R interface to Apache Spark. https://CRAN.R-project.org/package=sparklyr
Matloff N (2011) The art of R programming: a tour of statistical software design. No starch press series. No Starch Press, San Francisco
Matloff N (2015) Parallel computing for data science: with examples in R, C++ and CUDA. Chapman & Hall/CRC the R series. CRC Press, Boca Raton
Matloff N (2016) Software Alchemy: turning complex statistical computations into embarassingly–parallel ones. J Stat Softw 71(4):1–15
Matloff N, Fitzgerald C, Davis R, Yancey R, Huang S (2017a) partools: tools for the 'Parallel' package. https://github.com/matloff/partools
Matloff N, Yang V, Nguyen H (2017b) cdparcoord: top frequency-based parallel coordinates. https://CRAN.R-project.org/package=cdparcoodr
Murrell P (2011) R graphics, 2nd edn. Chapman & Hall/CRC the R series. Taylor & Francis, Boca Raton, FL
Nielsen F (2016) Introduction to HPC with MPI for data science. Undergraduate topics in computer science. Springer International Publishing, Cham
Plotly Technologies Inc (2015) Collaborative data science. https://plot.ly
R Core Team (2017) R: a language and environment for statistical computing. In: R foundation for statistical computing, Vienna. https://www.R-project.org/
Reinders J (2007) Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly series. O'Reilly Media, Sebastopol
Sarkar D (2008) Lattice: multivariate data visualization with R. Use R! Springer, New York
Unwin A, Theus M, Hofmann H (2007) Graphics of large datasets: visualizing a million. Statistics and computing. Springer, New York
Weston S (2017) foreach: provides foreach looping construct for R. https://CRAN.R-project.org/package=foreach
Wickham H (2016) Ggplot2: elegant graphics for data analysis. Use R! Springer International Publishing, New York
Yang V, Nguyen H, Matloff N, Xie Y (2017) Top-frequency parallel coordinates plots (arxiv). arXiv:1709.00665
Yu H (2014) [Rmpi] news. http://www.stats.uwo.ca/faculty/yu/Rmpi/

Ramp Secret Sharing Scheme (RSSS)

▶ Security and Privacy in Big Data Environment

RDF Compression

Miguel A. Martínez-Prieto[1], Javier D. Fernández[2], Antonio Hernández-Illera[1], and Claudio Gutiérrez[3]
[1]Department of Computer Science, Universidad de Valladolid, Valladolid, Spain
[2]Complexity Science Hub Vienna, Vienna University of Economics and Business, Vienna, Austria
[3]Department of Computer Science and Millenium Institute on Foundations of Data, Universidad de Chile, Santiago, Chile

Synonyms

Linked data compression; Semantic data Compression

Definitions

RDF compression can be defined as the problem of encoding an RDF dataset using less bits than that required by text-based traditional serialization formats like `RDF/XML`, `NTriples`, or `Turtle`, among others. These savings immediately lead to more efficient storage (i.e., archival) and less transmission costs (i.e., less bits over the wire). Although this problem can be easily solved through universal compression (e.g., `gzip` or `bzip2`), optimized *RDF-specific compressors* take advantage of the particular features of RDF datasets (such as semantic redundancies) in order to save more bits or to provide retrieval operations on the compressed information. RDF self-indexes are focused on this latter task.

RDF self-indexes are RDF compressors that provide indexing features in a space close to that of the compressed dataset and can be accessed with no prior (or partial) decompression. These properties enhance scalability (i.e., less resources are required to serve semantic data) and speed up access as more information can be managed in higher levels of the memory hierarchy (typically, main memory or cache). In addition, efficient search algorithms have been proposed to resolve basic queries on top of self-indexed datasets. As a result, RDF self-indexes have been adopted as a core component of semantic search engines and lightweight Linked Data servers.

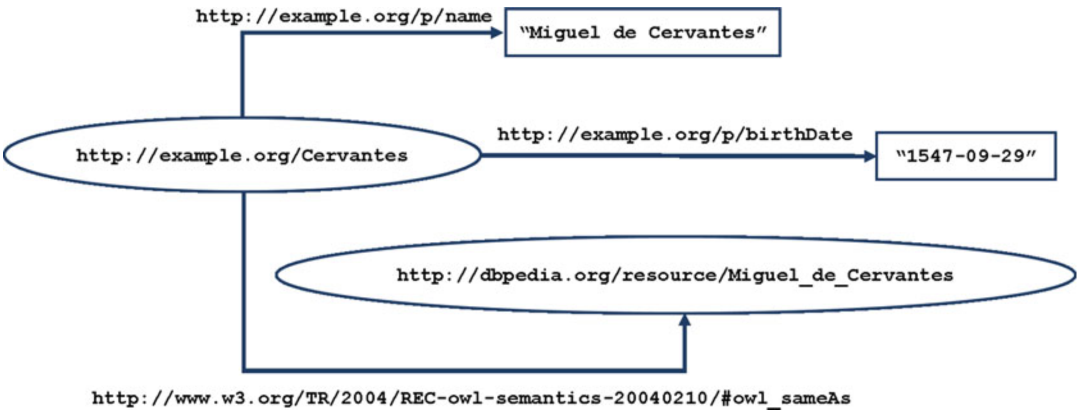
Finally, *RDF stream compressors* specifically focus on compressing a (continuous) stream of RDF data in order to improve exchange processes, typically in real time. This constitutes a more recent trend that exploits different trade-offs between the space savings achieved by the compressor and the latency introduced in the compression/decompression processes.

This entry introduces basic notions of RDF compression, RDF self-indexing, and RDF stream compression and discusses how existing approaches deal with (and remove) redundant information in semantic datasets.

Overview

RDF (Schreiber and Raimond 2014) stands for *Resource Description Framework*, which allows information about resources (documents, people, physical/logical objects, etc.) to be easily expressed in the form of triples. Each triple comprises the resource being described (referred to as subject), a property of that resource (predicate), and the corresponding value (object). Assuming infinite, mutually disjoint sets U (RDF Uniform Resource Identifiers, *URIs*), B (blank nodes), and L (RDF literals), a triple $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ is called an RDF triple (Gutiérrez et al. 2004). A triple (s, p, o) is usually represented as a labeled directed graph: $s \xrightarrow{p} o$.

Figure 1 shows an RDF graph that comprises three triples about the Spanish writer *Miguel de Cervantes*. The node named as <http://example.org/Cervantes> declares the corresponding RDF resource, while the labeled edges provide his


http://example.org/p/name
"Miguel de Cervantes"
http://example.org/Cervantes
http://example.org/p/birthDate
"1547-09-29"
http://dbpedia.org/resource/Miguel_de_Cervantes
http://www.w3.org/TR/2004/REC-owl-semantics-20040210/#owl_sameAs

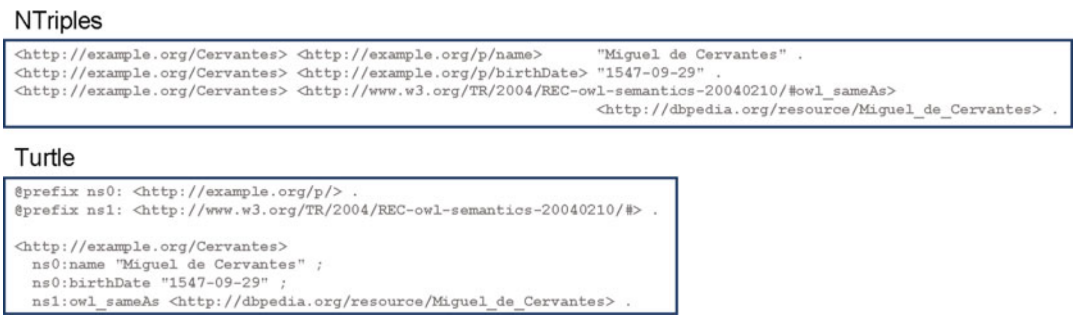
RDF Compression, Fig. 1 RDF graph which comprises three triples about *Miguel de Cervantes* *name* and *birth date* as literals and a *same as* link pointing to a similar resource <http://dbpedia.org/resource/Miguel_de_Cervan-tes>, described in DBpedia, a partial conversion of Wikipedia to RDF.

RDF is an extremely simple, but powerful logical model, which can be used to describe and integrate a variety of data from different domains. This flexibility has motivated the adoption of RDF as one of the mainstream semi-structured data model in life sciences, geography, or open-government projects, among other fields of knowledge. Cross-domain datasets, like the aforementioned DBpedia, demonstrate how heterogeneous data can be effectively integrated regardless of its (lack of) structure.

RDF excels at logical level, but important scalability issues arise at physical level. The case of DBpedia is exemplary. The latest edition of this dataset *(DBpedia 2016-10)* consists of roughly 13 billion triples which describe people, places, organizations, films, species, or diseases, among other information available at *Wikipedia, Wikipedia Commons*, and *Wikidata*. These descriptions also exploit that the source information is commonly expressed in different languages to consolidate a multilingual RDF-based encyclopedia. The resulting knowledge base is extensively used for multiple purposes and services (e.g., semantic search, entity disambiguation, translation, etc.), and it constitutes a valuable resource for academics and semantic web practitioners. However, although DBpedia can be queried online via different APIs, some applications and services require a complete view of the dataset; hence, consumers must deal with the high-cost requirements of this huge dataset in terms of space (its current dump needs several hundreds of gigabytes) and the required processing power to operate on this data volume.

These numbers endorse RDF compression as a scalable technique to alleviate the costs of RDF management, which are particularly relevant in a resource-constrained scenario (e.g., light clients or low-performance networks) and Big Semantic Data applications. But, *why are RDF compressors so effective?* Attending to the foundations of data compression, *compression is possible because data is normally represented in a form that is longer than absolutely necessary* (Salomon 2007). This situation is particularly significant in semantic data. An example is depicted in Fig. 2, which shows the previous RDF graph serialized in NTriples (Beckett 2014) and Turtle (Beckett et al. 2014). RDF files are plenty of redundancy, and three different classes are considered (Pan et al. 2014):

Symbolic redundancy is due to symbol repetitions. The main contributors of this significant source of redundancy are the large and highly repetitive URIs used for naming purposes. In practice, an RDF dataset comprises many different URIs, but they are usually defined from a small group of domains and tend to have common long prefixes. Note, for instance, that the prefix http://example.org/ is shared by three URIs in our example. Some RDF serialization formats, like Turtle, introduce the `@prefix` constructor to partly address this issue. This allows the original URI to be rewritten as a short reference to the shared prefix (e.g., `ns0` or `ns1` in Fig. 2)


NTriples

```
<http://example.org/Cervantes> <http://example.org/p/name>      "Miguel de Cervantes" .
<http://example.org/Cervantes> <http://example.org/p/birthDate> "1547-09-29" .
<http://example.org/Cervantes> <http://www.w3.org/TR/2004/REC-owl-semantics-20040210/#owl_sameAs>
                                                                 <http://dbpedia.org/resource/Miguel_de_Cervantes> .
```

Turtle

```
@prefix ns0: <http://example.org/p/> .
@prefix ns1: <http://www.w3.org/TR/2004/REC-owl-semantics-20040210/#> .

<http://example.org/Cervantes>
  ns0:name "Miguel de Cervantes" ;
  ns0:birthDate "1547-09-29" ;
  ns1:owl_sameAs <http://dbpedia.org/resource/Miguel_de_Cervantes> .
```

RDF Compression, Fig. 2 Example of RDF serializations (NTriples & Turtle) and the remaining suffix. Symbol repetitions are also present in infixes/suffixes of URIs, but their impact is much lower. On the other hand, symbolic redundancy from literals depends on the domain being described, and it can be more difficult to predict. Finally, note that no restrictions are made on blank node serialization, but they also tend to share substrings (as in URIs) (Martínez-Prieto et al. 2012b).

Universal compressors, like `gzip` or `bzip2`, remove this class of redundancy and generate files which are suitable for storage and exchange. However, this class of compression prevents other purposes, like random access to the compressed data. *String dictionaries* (Martínez-Prieto et al. 2016) arise as an alternative that provides (slightly) less compression than universal techniques but efficient RDF retrieval operations. Dictionary compression identifies all different strings (*vocabulary*) in an RDF dataset and assigns an integer identifier (ID) to each one (typically and ID $i \in [1, n]$ being n the vocabulary size). This simple decision allows for replacing each term occurrence in the RDF dataset by its corresponding ID, thus shifting from managing potential large strings to small-/medium-size integers. In other words, the original RDF graph is re-encoded as a string dictionary, organizing the vocabulary, and an ID-graph. It is worth noting that RDF vocabularies reach non-negligible sizes and must be also compressed (Martínez-Prieto et al. 2012b).

Syntactic redundancy underlies to the graph-shaped structure of RDF datasets and how this structure is serialized. The simplest serialization formats, like NTriples, write one triple per line and serialize explicitly all its components (e.g., the subject URI is written three times, once per triple, in Fig. 2). In contrast, syntaxes like Turtle introduce subject-based encoding to alleviate this issue, i.e., it allows triples to be encoded as adjacency lists of (predicate, object) pairs related to each subject. Similar observations can be done for predicates and objects. Predicates tend to be repeated across different subjects, but they are often restricted to a domain and range of application, whether explicitly described in a vocabulary (ontology) or implicitly in the data. For example, the *salary* predicate is not typically related to a *book* but a *person*, while the converse applies to the *pages* property. Grouping and splitting the description of books and persons may save repetitions. In turn, some objects are paired with some particular predicates (e.g., 25 °C is tight to a *temperature* predicate), but these pairs could be reused for different subjects.

Particular *graph compressors* should be used to minimize this class of redundancy. These techniques rearrange graphs in terms of their adjacency information and encode it using list or matrix compression approaches. The most well-known techniques come from web or social graph compression scenarios (Boldi and Vigna 2004), but some of them (Brisaboa et al. 2014) have been tuned to address the particular needs of RDF graphs. It is worth noting that graph compressors often operate on integer representations, so an initial dictionary compression stage is first performed to obtain the corresponding ID-graphs.

Semantic redundancy, in contrast to the previous ones, appears at the logical level. It arises when less triples can be used to provide the same knowledge. Thus, the redundancy does not depend on how triples are encoded, but on the knowledge they provide.

This redundancy cannot be removed using traditional compression approaches. In this case, specific RDF compressors must be designed from scratch in order to obtain the minimal subset of "canonical triples" that allows the original knowledge to be effectively reconstructed. Thus, the compression is achieved by reducing the number of unnecessary encoded triples. It is worth noting that symbolic and syntactic redundancies may be still present on the canonical graph; hence, other forms of compression can be applied on top of the semantic-compressed dataset in order to achieve better compression ratios.

Key Research Findings

RDF compression has emerged as an active research and development field over the past years, and some *lossless compressors* (i.e., techniques that can exactly recreate the original dataset from its compressed representation) have been proposed. RDF compressors can be classified into *physical* and *logical*: the former exploits symbolic/syntactic redundancy, while the latter focuses on semantic-based redundancy. Finally, *hybrid* compressors perform at physical and logical levels.

The simplest form of **physical compression** is *universal compression*, i.e., using any general-purpose technique (e.g., `gzip` or `bzip2`) to directly compress an RDF file. This choice is simple and efficient in terms of compression ratio and delays (fast to process), and it can be easily integrated in other workflows as it makes use of standard and widespread techniques. In turn, RDF-specific physical compressors can be designed from scratch in order to deal with specific RDF characteristics and achieve better performance than general-purpose techniques.

RDF-specific physical compressors usually perform *dictionary compression*. That is, they translate the original RDF graph into a new representation which includes a string dictionary and an ID-graph encoding:

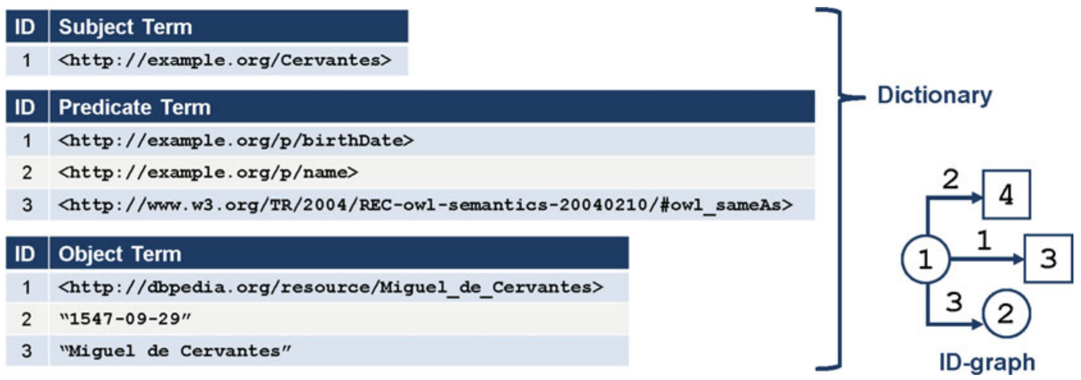
- The **dictionary** organizes the RDF vocabulary, which comprises all different terms used in the dataset.
- The **ID-graph** replaces the original terms by their corresponding IDs in the dictionary.

Figure 3 shows a dictionary-compressed representation of the graph example. In this case, the dictionary comprises three independent mappings by subjects, predicates, and objects, but other configurations are possible. Triples are then encoded as `(1,1,2) (1,2,3) (1,3,1)`, where the original terms can be easily retrieved by using the corresponding mapping; e.g., `(1,2,3)` is translated into:

- `S.get(1) = `<http://example.org/Cervantes>.
- `P.get(2) = `<http://example.org/p/name>.
- `O.get(3) = "Miguel de Cervantes".`

Physical compressors propose different approaches to organize and compress RDF dictionaries and to encode the corresponding ID-graph representations.

Dictionary compression has not received much particular attention, in spite that representing the RDF vocabulary usually takes more space than the ID-graph encoding (Martínez-Prieto et al. 2012b). As in our example, RDF vocabularies can be intuitively partitioned by term roles (subjects, predicates, and objects). However, this approach is not totally effective because terms that perform subject and object roles are encoded twice, and these terms can sum up to 60% of the vocabulary terms (Martínez-Prieto et al. 2012b). Thus, a four-vocabulary configuration (Atre et al. 2010) is massively used, where the dictionary holds independent mappings for (i) shared *subjects-objects*, (ii) single *subjects* (not occurring as objects), and (iii) single *objects* (not occurring as subjects) and *predicates*.



ID	Subject Term
1	`<http://example.org/Cervantes>`

ID	Predicate Term
1	`<http://example.org/p/birthDate>`
2	`<http://example.org/p/name>`
3	`<http://www.w3.org/TR/2004/REC-owl-semantics-20040210/#owl_sameAs>`

ID	Object Term
1	`<http://dbpedia.org/resource/Miguel_de_Cervantes>`
2	`"1547-09-29"`
3	`"Miguel de Cervantes"`

RDF Compression, Fig. 3 Dictionary-compressed representation

Vocabularies are further splitted by URIs, blank nodes, and literals for choosing the best compression technique for each collection of RDF terms. Martínez-Prieto et al. (2016) propose different approaches to compress these RDF vocabularies. URI (and blank node) vocabularies can be effectively compressed using Front-Coding, a differential encoding mechanism that exploits shared long prefixes and reduces URI space requirements up to 20 times. On the other hand, literal vocabularies are, in general, less predictable, hampering their compression. Thus, dictionaries that combine grammar-based compression with Front-Coding or hashing can report huge space savings of up to six to ten times the original space. Besides space reductions, these dictionary compression techniques must be able to efficiently translate IDs into the corresponding RDF terms, ensuring efficient *decoding* performance, most of them working at microsecond level. These techniques often support the inverse *search* functionality to translate an RDF term into its corresponding ID, which is essential for RDF self-indexing. In addition, they can provide *prefix*- and *substring*-based retrieval of RDF terms.

Once removed symbol repetitions, *ID-graph compression* looks for syntactic redundancy on the resulting ID-graph. These techniques model the graph in terms of adjacency lists or matrices, and look for regularities or patterns, which are succinctly encoded. HDT (Fernández et al. 2013) proposes *BitmapTriples*, one of the pioneer approaches for (RDF) ID-graph compression. In essence, it transforms the graph into a forest of three-level trees: each tree is rooted by a subject ID, having its adjacency list of predicates in the second level and, for each of them, the adjacency list of related objects in the third (leaf) level. The whole forest is then compressed using two ID sequences (for predicates and objects) and two bit sequences which encode the number of branches and leaves of each tree. This simple encoding reports interesting compression ratios (10–25% of the original space), while supporting efficient triple decoding. Furthermore, BitmapTriples allows subject-based queries to be resolved by traversing subject trees from the root. HDT consolidates a binary serialization format by joining Front-Coding and BitmapTriples to compress dictionaries and ID-graphs, respectively.

OFR (Swacha and Grabowski 2015) proposes another compression scheme for ID-graphs. It first performs dictionary compression (terms are organized into a multi-dictionary using differential encoding), and the resulting ID-graph is sorted by objects and subjects. In this case, *run-length* and *delta* compression (Salomon 2007) are applied to exploit multiple object occurrences, and the non-decreasing order of the consecutive subjects, respectively. OFR compressed files are then recompressed using universal techniques like `gzip` or `7zip`. The resulting OFR effectiveness improves HDT, but its inner data organization discourages any chance of efficient retrieval.

Efficient RDF retrieval is addressed by RDF self-indexes. These approaches do not just compress the ID-graph, but also provide indexing capabilities over it. HDT-FoQ (Martínez-Prieto et al. 2012a) enhances HDT to also support predicate- and object-based queries, adding inverted indexes for predicate and object adjacency lists that, all together, provide excellent performance for resolving SPARQL triple patterns. k^2-triples (Álvarez-García et al. 2014) provides an alternative organization of the ID-graph, encoding a (binary) adjacency matrix of (subject, object) pairs per predicate. These matrices are very sparse and can be easily compressed using k^2-trees (Brisaboa et al. 2014). The k^2-triples approach improves HDT-FoQ compression ratios, and reports competitive numbers for all triple patterns binding the predicate, but results in a poor performance in those queries with unbounded predicates. This is mitigated by adding two additional indexes to store the predicates related to each subject and object, but the pattern that only binds the predicate remains slow. RDFCSA (Brisaboa et al. 2015) is the most recent RDF self-index, which encodes the ID-graph as a compressed suffix array (CSA). RDFCSA also ensures efficient lookup performance, competing with k^2-triples at the cost of using more space.

Logical compressors look for (redundant) triples that can be inferred from others. These triples are removed from the original graph, and only the resulting *canonical subgraph* is finally serialized. Different approaches have been followed to obtain these canonical subgraphs. The initial approaches (Iannone et al. 2005; Meier 2008) are based on the notion of *lean subgraph*. The lean subgraph is a subset of the original graph that has the property of being the smallest subgraph that is instance of the original graph. The number of removed triples by a lean subgraph strongly depends on the graph features, but a reasonable lower limit is two removed triples per blank node (Iannone et al. 2005). Nevertheless, some triples of a lean graph can still be derived from others; hence, some semantic redundancy can still be present (Meier 2008).

The rule-based (RB) compressor (Joshi et al. 2013) uses mining techniques to detect objects that are commonly related to a particular predicate (intra-property patterns) and to group frequent predicate-object pairs (inter-property patterns). These patterns are then used as generative rules to remove triples that can be inferred from such patterns. RB is not so effective by itself, and only inter-property patterns enable significant amount of triples to be removed. Venkataraman and Sreenivasa Kumar (2015) state that frequent patterns are not so expressive to capture semantic redundancy and suggest that effectiveness can be improved using more expressive rules. In this case, Horn rules are mined from the dataset, and all triples matching their head part are removed. The resulting canonical subgraph is then compressed using RB. This Horn-rule-based compressor outperforms RB effectiveness, but it introduces latencies in compression and decompression processes.

Hybrid compressors compact the RDF graph by first using a logical approach to remove redundant triples and then performing physical compression at serialization level. Although these techniques could combine the best of logical and physical compression, their application has received relatively little attention until now.

HDT++ (Hernández-Illera et al. 2015) revisits HDT to introduce some methods to detect syntactic and semantic redundancy. HDT++ brings out the inherent structure of RDF by detecting and grouping the different set of predicates (*predicate families*) used to describe subjects. The original RDF graph is encoded as a set of subgraphs, one per predicate family. The `rdf:type` values are attached to each predicate family, hence removing these triples from the subgraphs. Finally, HDT++ uses local IDs for the terms in each subgraph, thus reducing the number of required bits. As a result, HDT++ reduces the original HDT ID-graph space requirements up to two times for more structured datasets and reports significant improvements even for highly semi-structured datasets.

The *graph pattern-based* (GPB) compressor (Pan et al. 2015) shares some common features with HDT++, also grouping subjects by predicate families, called *entity description patterns* (EDPs). Each EDP is encoded as a pair which includes the corresponding pattern and all instances matching it. This policy consolidates the simplest GPB encoding scheme (LV0), but patterns are then merged to obtain better patterns (LV1), and the merging process can be recursively performed (LV2). GPB results are not compared with other physical compressors, but they excel at logical level, where GPB-LV2 is able to remove more triples than RB.

Finally, *RDF2NormRDF* (Ticona-Herrera et al. 2015) is not a compressor, but an approach to normalize RDF data. It applies different transformation rules to remove duplicated edges and nodes and also deals with particular blank node features. At physical level, RDF2NormRDF focuses on namespace issues and normalizing types and language tags encodings. As expected, RDF2NormRDF does not generally remove more redundancy than other compressors, such as HDT, but it reports better compression ratios for small datasets.

Real-Time Compression

All previous compressors share a common feature: they are designed to perform in batch processing scenarios where time requirements are not so strict. That is, the target system can wait a reasonable time to receive the compressed data, and the compression process is performed *offline*.

However, more and more systems perform *online* data processing (in real time), where delays must be minimized. In these cases, the volume of RDF is traditionally smaller, but triples are continuously generated, and these semantic streams must be efficiently distributed over a network.

Thus, a key challenge for RDF stream processing systems is the ability to consume increasingly large volumes of data with varying and potentially high input rates. This scenario also claims for RDF compression that, in this case, put the focus on minimizing the elapsed time since the original piece of data is available at the producer, until it is ready to be used at the consumer. This process encompasses three tasks: (i) *data compression* (at the producer), (ii) *data transmission* over a network, and (iii) *data decompression* (at the consumer). The real challenge for RDF stream compressors is finding the right space/time trade-off which optimizes the overall workflow performance. RDF-specific compression over streaming data is usually performed at *physical level* and mostly as adaptations of existing RDF compressors.

Streaming HDT (Hasemann et al. 2012) adapts HDT to simplify the associated metadata and restrict the number of terms managed by the dictionary; hence, shorter IDs are used. The approach is then tailored to constrained devices, where the vocabulary of terms is very limited. In turn, RDSZ (Fernández et al. 2014a) uses differential encoding to take advantage of the similarities between consecutive triples sent over the wire. In addition, the resultant scheme is compressed with Zlib to exploit additional redundancies. Overall, RDSZ gains in compression (17% on average) are at the cost of increasing the processing time.

ERI (Fernández et al. 2014b) is an RDF stream compressor that adapts the W3C Efficient XML Interchange (EXI) format (Schneider et al. 2014) for RDF data. Similarly to HDT++ and GPB, ERI tries to detect and encode the predicate families but in a dynamic fashion. In addition, highly repeated object values (besides the values for *rdf:type*) are also encoded in the families. Then, the non-repeated values for each particular predicate in the family are encoded in a channel, where specific compression can be applied (the concept is then similar to leveraging the locality in column-based databases). As a result, ERI achieves a slightly better compression than general-purpose stream-enabled compressors (such as Zlib) with limited latency overhead. Note that the XML-based compression of EXI can also be adapted to RDF streams by (i) forcing the serialization format of RDF to RDF/XML and, optionally, (ii) generating an application-specific grammar that encodes the repeated values.

The pattern-based approach of ERI has been also followed by two recent techniques, PatBin (Lhez et al. 2017) and FSSD (Karim et al. 2017). PatBin approach performs dictionary-based compression and then splits the graph in ID-patterns (essentially predicate families) and value/variable bindings. The Factorizing Semantic Sensor Data (FSSD) technique uses a deductive database system to encode the repeated predicate families and object values as datalog rules, which are then applied on the input data to achieve important size reductions. However, although FSSD has been tested on sensor data, the factorization is still performed in an offline fashion. In general, finding patterns in RDF Streams efficiently can be seen as an orthogonal approach.

Examples of Application

RDF compression has been widely adopted by the Semantic Web community as a standard technique to reduce storage and transmission costs when downloading RDF datasets. Most of the publishers in the Linked Open Data (LOD) cloud make use of universal compression, given its simplicity, usability, and widespread adoption. This is particularly true for projects publishing massive amounts of RDF data, such as DBpedia or Bio2RDF.

Nonetheless, RDF-specific compressors, and in particular RDF self-indexes, are receiving increased attention. Projects like LOD Laundromat (Beek et al. 2014) or Triple Pattern Fragments (TPF) (Verborgh et al. 2016) describe two interesting use cases exploiting compressed RDF. LOD Laundromat is an initiative to crawl and clean (removing syntax errors) RDF data from the LOD cloud. As a result, it exposes more than 650 K cleansed datasets which are delivered in HDT format and can be queried using TPF interfaces. TPF focuses on alleviating the burden of endpoints by serving simple SPARQL triple patterns, paginating the results. This simplification allows servers to scale, while clients can always execute more complex SPARQL queries on top of TPFs by taking care of integrating and filtering the results. Given the simplicity of the required infrastructure at the server, TPF interfaces can make use of RDF self-indexes to serve low-cost operations, being HDT the most used backend in practice. The recently published *LOD-a-lot* dataset (Fernández et al. 2017) combines the benefits from both projects to provide a practical example of efficient management of compressed Big Semantic Data. *LOD-a-lot* integrates all data from LOD Laundromat into a cross-domain mashup of more than 28 billion triples and several terabytes of space (in NTriples). This dataset is then exposed as HDT and the corresponding TPF interface. The queryable self-indexed HDT of such large portion of the LOD cloud takes 524 GB and can serve fast triple pattern resolution with an affordable memory footprint (in practice, 15.7 GB). These numbers are a strong evidence of how RDF compression contributes to make Big Semantic Data management feasible in most Linked Data servers (for online consumption) and clients (for downloading and offline consumption).

RDF compression and self-indexes have also been actively used in other Semantic Web areas such as (i) SPARQL querying and recommender systems (Martínez-Prieto et al. 2012a; Heitmann and Haye 2014), leveraging the retrieval operations supported by self-indexes to support more complex queries; (ii) reasoning (Cure et al. 2015), optimizing the RDF dictionary and triples encoding to serve inference capabilities; (iii) versioned RDF or RDF archives (Fernández et al. 2016), where RDF compression is used to preserve (and query) the history of an RDF dataset; and (iv) constrained and mobile devices (Käbisch et al. 2015) in order to maximize the exploitation of their storage/processing capabilities.

Finally, RDF compression has also been highlighted by RDF stream processing systems: CQELS Cloud (Le-Phuoc et al. 2013) uses a basic dictionary-based approach to process and move IDs (integers) between nodes, and Ztreamy (Fisteus et al. 2014) exploits the Zlib compressor with similar purposes. A last trend regards querying on compressed RDF streams (without prior decompression). A recent approach (Déme et al. 2017) extends RDFSZ to resolve basic SPARQL queries, as well as filters and aggregations.

Future Directions for Research

The state of the art of RDF compression shows many and varied techniques which exploit symbolic, syntactic, and semantic redundancy from different perspectives. They (i) save much storage space, (ii) reduce network latencies, or (iii) improve RDF retrieval performance, among other achievements. Nonetheless, RDF compression still remains an open challenge. We can categorize future directions in three categories, *low-level* optimization, *scalable management*, and *applications*.

First, physical and logical compression should be independently explored to determine more effective approaches which, desirably, should be plugged into powerful hybrid compressors. These achievements should be aligned with recent advances in succinct data structures and self-indexes, which try to squeeze the space requirements while providing additional retrieval operations. Querying compressed RDF is a main direction for research to consolidate scalable and efficient semantic search engines. These additional *low-level* optimizations can boost the adoption of RDF compression and self-indexing, which is already at the edge of conforming a core component of scalable Semantic Web applications.

In general, RDF compression has already proved its ability for storing, exchanging, processing, and querying Big Semantic Data.

However the *scalable management* of compressed RDF datasets has room for optimization. On the one hand, the compression of Big Semantic Data, as well as the creation of RDF self-indexes, suffers from scalability problems, as they can require high amounts of memory and processing power. Although this process is a one-off task in many scenarios, it can introduce unaffordable costs for publishers. Current efforts focus on exploring computation models like MapReduce or Spark to alleviate this burden, yet additional research must be conducted to consolidate a scalable approach for Big Semantic Data compression. On the other hand, most of RDF self-indexes are mainly static or costly to update; hence, dynamic techniques for succinct data structures and self-indexes are an active area of research.

Finally, RDF compression and its *applications* have many unexplored directions. Adoption of RDF compressed techniques in combination with existing triple stores and reasoners, integration within semantic data warehouses, resolution of temporal queries on compressed RDF archives, or integration within natural language processing tasks (e.g., entity discovering and linking) are only some prominent applications which claim for innovative RDF compression techniques to scale in a Big Semantic Data scenario.

Cross-References

▶ RDF Serialization and Archival
▶ Semantic Search

References

Álvarez-García S, Brisaboa N, Fernández JD, Martínez-Prieto MA, Navarro G (2014) Compressed vertical partitioning for efficient RDF management. Knowl Inf Syst 44(2):439–474

Atre M, Chaoji V, Zaki M, Hendler J (2010) Matrix “Bit” loaded: a scalable lightweight join query processor for RDF data. In: 19th international conference on World Wide Web (WWW), pp 41–50

Beckett D (2014) RDF 1.1 N-Triples. W3C recommendation. https://www.w3.org/TR/2014/REC-n-triples-20140225/

Beckett D, Berners-Lee T, Prud’hommeaux E, Carothers G (2014) RDF 1.1 turtle. W3C recommendation. https://www.w3.org/TR/2014/REC-turtle-20140225/

Beek W, Rietveld L, Bazoobandi HR, Wielemaker J, Schlobach S (2014) LOD Laundromat: a uniform way of publishing other people’s dirty data. In: 13th international semantic web conference (ISWC), pp 213–228

Boldi P, Vigna S (2004) Webgraph framework I: compression techniques. In: 13th international conference on World Wide Web (WWW), pp 595–602

Brisaboa N, Ladra S, Navarro G (2014) Compact representation of web graphs with extended functionality. Inf Syst 39(1):152–174

Brisaboa N, Cerdeira-Pena A, Fariña, Navarro G (2015) A compact RDF store using suffix arrays. In: 22nd international symposium on string processing and information retrieval (SPIRE), pp 103–115

Cure O, Naacke H, Randriamalala T, Amann B (2015) LiteMat: a scalable, cost-efficient inference encoding scheme for large RDF graphs. In: 2015 IEEE international conference on big data (Big Data), pp 1823–1830

Déme NB, Dia AF, Boly A, Kazi-Aoul Z, Chiky R (2017) An efficient approach for real-time processing of RDSZ-based compressed RDF streams. In: 15th international conference on software engineering research, management and applications (SERA), pp 147–166

Fernández JD, Martínez-Prieto MA, Gutiérrez C, Polleres A, Arias M (2013) Binary RDF representation for publication and exchange. J Web Semant 19:22–41

Fernández N, Arias J, Sánchez L, Fuentes-Lorenzo D, Corcho Ó (2014a) RDSZ: an approach for lossless RDF stream compression. In: 11th European conference on the semantic web (ESWC), pp 52–67

Fernández JD, Llaves A, Corcho O (2014b) Efficient RDF interchange (ERI) format for RDF data streams. In: 13th international semantic web conference (ISWC), pp 244–259

Fernández JD, Umbrich J, Polleres A, Knuth M (2016) Evaluating query and storage strategies for RDF archives. In: 12th international conference on semantic system (SEMANTiCS), pp 41–48

Fernández JD, Beek W, Martínez-Prieto MA, Arias M (2017) LOD-a-lot – a queryable dump of the LOD cloud. In: 16th international semantic web conference (ISWC), vol 2, pp 75–83

Fisteus JA, Fernández García N, Sánchez Fernández L, Fuentes-Lorenzo D (2014) Ztreamy: a middleware for publishing semantic streams on the web. J Web Semant 25:16–23

Gutiérrez C, Hurtado C, Mendelzon AO (2004) Foundations of semantic web databases. In: 23rd ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems (PODS), pp 95–106

Hasemann H, Kroller A, Pagel M (2012) RDF provisioning for the Internet of things. In: 3rd international conference on the Internet of things (IOT), pp 143–150

Heitmann B, Haye C (2014) SemStim at the LOD-RecSys 2014 challenge. In: Semantic web evaluation challenge (SemWebEval), pp 170–175

Hernández-Illera A, Martínez-Prieto MA, Fernández JD (2015) Serializing RDF in compressed space. In: 21st data compression conference (DCC), pp 363–372

Iannone L, Palmisano I, Redavid D (2005) Optimizing RDF storage removing redundancies: an algorithm. In: 18th international conference on industrial and engineering applications of artificial intelligence and expert systems (IEA/AIE), pp 732–742

Joshi A, Hitzler P, Dong G (2013) Logical linked data compression. In: 10th extended semantic web conference (ESWC), pp 170–184

Käbisch S, Peintner D, Anicic D (2015) Standardized and efficient RDF encoding for constrained embedded networks. In: 12th European conference on the semantic web (ESWC), pp 437–452

Karim F, Vidal ME, Auer S (2017) Efficient processing of semantically represented sensor data. In: 13th international conference on web information systems and technologies (WEBIST), pp 252–259

Le-Phuoc D, Quoc HNM, Le Van C, Hauswirth M (2013) Elastic and scalable processing of linked stream data in the cloud. In: 12th international semantic web conference (ISWC), pp 280–297

Lhez J, Ren X, Belabbess B, Curé O (2017) A compressed, inference-enabled encoding scheme for RDF stream processing. In: 14th European conference on the semantic web (ESWC), pp 79–93

Martínez-Prieto MA, Arias M, Fernández JD (2012a) Exchange and consumption of huge RDF data. In: 9th extended semantic web conference (ESWC), pp 437–452

Martínez-Prieto MA, Fernández JD, Cánovas R (2012b) Compression of RDF dictionaries. In: 27th ACM international symposium on applied computing (SAC), pp 1841–1848

Martínez-Prieto M, Brisaboa N, Cánovas R, Claude F, Navarro G (2016) Practical compressed string dictionaries. Inf Syst 56:73–108

Meier M (2008) Towards rule-based minimization of RDF graphs under constraints. In: 2nd international conference on web reasoning and rule systems (RR), pp 89–103

Pan J, Gómez-Pérez J, Ren Y, Wu H, Zhu M (2014) SSP: compressing RDF data by summarisation, serialisation and predictive encoding. Technical report. http://www.kdrive-project.eu/wp-content/uploads/2014/06/WP3-TR2-2014_SSP.pdf

Pan J, Gómez-Pérez J, Ren Y, Wu H, Haofen W, Zhu M (2015) Graph pattern based RDF data compression. In: 4th joint international conference on semantic technology (JIST), pp 239–256

Salomon D (2007) Data compression: the complete reference. Springer, New York

Schneider J, Kamiya T, Peintner D, Kyusakov R (2014) Efficient XML interchange (EXI) format 1.0. W3C recommendation

Schreiber G, Raimond Y (2014) RDF 1.1 primer. W3C working group note. https://www.w3.org/TR/rdf11-primer/

Swacha J, Grabowski S (2015) OFR: an efficient representation of RDF datasets. In: 4th symposium on languages, applications and technologies (SLATE), pp 224–235

Ticona-Herrera R, Tekli R, Chbeir J, Laborie S, Dongo I, Guzman R (2015) Toward RDF normalization. In: 34th international conference on conceptual modeling (ER), pp 261–275

Venkataraman G, Sreenivasa Kumar P (2015) Horn-rule based compression technique for RDF data. In: 30th annual ACM symposium on applied computing (SAC), pp 396–401

Verborgh R, Vander Sande M, Hartig O, Van Herwegen J, De Vocht L, De Meester B, Haesendonck G, Colpaert P (2016) Triple pattern fragments: a low-cost knowledge graph interface for the web. J Web Semant 37–38: 184–206

RDF Dataset Profiling

Stefan Dietze[1], Elena Demidova[1], and Konstantin Todorov[2]
[1]L3S Research Center, Leibniz Universität Hannover, Hanover, Germany
[2]LIRMM, University of Montpellier, Montpellier, France

Definitions

In the context of this chapter, *an RDF dataset* is defined in accordance with the dataset definition in the Vocabulary of Interlinked Datasets (VoID), (http://vocab.deri.ie/void), namely, “*A Dataset* is a set of RDF triples that are published, maintained or aggregated by a single provider.” According to VoID, a dataset represents a meaningful collection of triples as envisioned by its provider. *An RDF dataset profile* is a formal representation of a set of dataset characteristics (features). It describes the dataset and aids dataset discovery, recommendation, and comparison with regard to the represented features. *A dataset profile feature* is a characteristic describing a certain attribute of the dataset. For instance, “dataset conciseness” is a dataset profile feature providing information on the degree of redundancy of the information contained in the dataset. A dataset profile is extensible with respect to the features it contains. Usually, the relevant feature set is application-oriented and depends on the envisaged application scenarios.

Overview

A number of popular dataset registries have emerged, which tackle the problem of dataset discovery through the curation of lightweight dataset descriptions, often also exposing structured metadata according to the state-of-the-art vocabularies such as DCAT (http://www.w3.org/TR/vocab-dcat/) or VoID. Popular examples include DataHub (http://www.datahub.io) or DataCite (https://www.datacite.org/), while the LinkedUp Catalog (http://data.linkededucation.org/linkedup/catalog/) (for education) represents a domain-specific example. However, while such metadata is usually edited and curated manually, it is often sparse, not in sync with the constant evolution of the actual datasets, and prone to errors.

On the one hand, as the Web of Data as a whole is evolving along with the constant evolution of individual datasets, manual assessment and representation of a large variety of dataset features is neither feasible nor sustainable. On the other hand, a wide variety of competing as well as complementary approaches exist, aimed at automatic assessment and description of arbitrary datasets. This body of work is spanning several research communities and includes works in fields such as *dataset characterization*, *data summarization*, *dataset assessment*, or *dataset profiling*. While the problem of dataset profiling is of particular importance in the context of the Web of Data, it has been identified and approached already in other related fields, such as general database and data management research.

Emerging from the aforementioned works, a wealth of tools, methods, vocabularies, and applications for assessing, describing, and profiling datasets has become available throughout the past few years, where Ben Ellefi et al. (2017) provides an initial overview and classification.

The aim of this chapter is to provide researchers, dataset providers, and application developers with an overview of *dataset profiling* and closely related approaches, including *dataset profile features*, *feature extraction methods and tools*, *vocabularies*, and example *applications* to encourage experimentation and facilitate the broader use of RDF datasets.

Key Findings

In order to provide an overview of RDF dataset profiling, we structure key findings into types of *features*, related *methods and tools*, *vocabularies* for their representation, and actual *applications*.

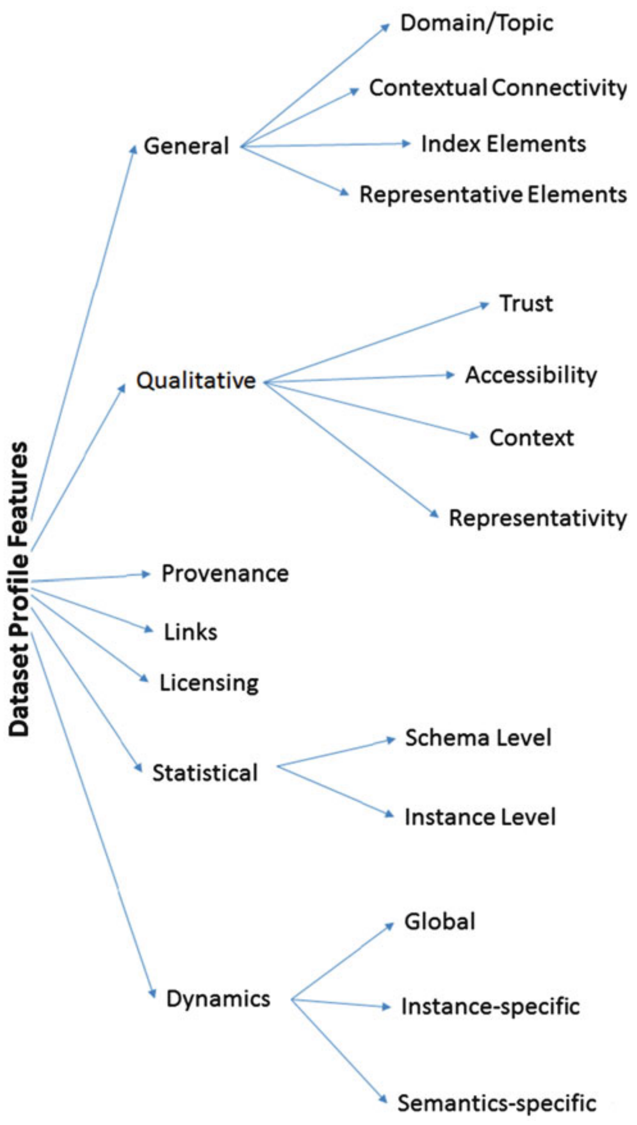
Features, Methods, and Tools

This section provides an inventory of dataset features of relevance for dataset profiling, organized in an extensible feature taxonomy, depicted in Fig. 1. This taxonomy reflects the authors' consensus and provides one of several possible categorization systems. In particular, the features are organized into the following top-level categories: *General*, *Qualitative*, *Provenance*, *Links*, *Licensing*, *Statistical*, and *Dynamics*. This taxonomy guides the categorization of the extraction methods and tools, of which we provide examples for each category. The taxonomy also serves as a backbone for the classification of profiling vocabularies and applications, introduced in the following section. For a complete overview, the reader is referred to the survey (Ben Ellefi et al. 2017).

General features. Features in this category carry high-level semantic information, including the *domain* (field of knowledge) or *topic* of the dataset; its *contextual connectivity*, understood as the extensional and/or topical overlap with other datasets; the *index elements* that are pointing to the dataset in a given index or a set of indices; as well as the dataset *representative elements*, i.e., the sets of most descriptive types, properties, or instance samples to characterize the entire dataset.

Extraction Methods and Tools. General features typically require domain knowledge with respect to the content of the dataset. As a best

RDF Dataset Profiling, Fig. 1 A taxonomy of dataset profile features



practice, these features should be provided by the data domain experts (e.g., data providers or maintainers) to ensure a high-quality profile. The *topic* and the *contextual connectivity* of the dataset can be described through named entities (NE) extracted from the literal values, as well as categories and clusters of these entities. Furthermore, RDF dataset profiles can include selected *index elements* from the instance-level and schema-level indices or data summaries such as a QTree (Harth et al. 2010). *Representative elements* of an RDF dataset can be the entities with thorough descriptions or predicates obtained using key discovery approaches. Tools in this category include Linked Data Observatory (Fetahu et al. 2014) and voiDge (Böhm et al. 2011).

Qualitative features. The *Qualitative* features category groups together a number of features known from the long-going study of data quality in computer science in general and the Web of Data in particular (Zaveri et al. 2016; Bizer and Cyganiak 2009). These features are organized into the following subcategories: (1) Trust, (2) Accessibility, (3) Representativity, and (4) Context/Task Specificity. *Trust* features relate to the verifiability of information, its correctness or believability, and the reputation of the publisher and that of the dataset. *Accessibility* regards the facility of access to information and the versatility of access methods, but also the degree of security of information transactions, as well as the access performance.

The features in the *Representativity* group provides information in terms of completeness, understandability, semantic accuracy, conciseness (or redundancy), and consistency (i.e., presence of contradictory information).

Finally, the *Context/Task Specificity* group includes features that measure data quality with respect to a specific task.

Extraction Methods and Tools. For the features in the *Accessibility* category, assessment methods include: (1) exploiting dataset metadata and user annotations to assess *trust*; (2) querying SPARQL endpoints and analysing URIs to measure *availability*; (3) analyzing use of digital signatures and provenance information to assess *security*; (4) assessing latency and scalability in response to the user requests to measure *performance*; and (5) analyzing available serialization formats and languages to assess *versatility*.

Representativity features can be assessed through: (1) exploiting statistical distributions to measure *completeness*; (2) detection of human-readable labeling, metadata availability, as well as communication channels to assess *understandability*; (3) statistical methods and crowdsourcing to measure *semantic accuracy/-correctness*; (4) measuring the proportion of unique elements at the schema and instance level to assess *conciseness*; and (5) verification of the schema definitions, correctness of the schema usage, and identification of data inconsistencies (e.g., through semantic constraints) to measure *consistency*.

Context features can be assessed through relevance, sufficiency, and timeliness (e.g., the age of the data and its temporal validity as it is delivered to the user) for a specific task or query.

Example tools to generate qualitative features in several categories include WIQA (Bizer and Cyganiak 2009) and ODPW (Umbrich et al. 2015).

Provenance features. These features are seen as contextual metadata that provide indicators about the origin, timeliness, currency, and update cycles of datasets. These are important characteristics that allow to understand the origins of data, to trace errors, and, ultimately, to establish trust.

Extraction Methods and Tools. Provenance features can be extracted from the annotations specified by the data provider via provenance vocabularies using tools like WIQA and ODPW.

Links features. *Links features* are understood as either the number of datasets, with which a dataset is interlinked, or the number of triples in a dataset, in which the subject and the object refer to different datasets. We distinguish between (i) *explicit links* (when datasets have linked instances via `owl:sameAs` statements) and (ii) *implicit links* (when datasets share topics or contexts, expressed, e.g., by an `rdfs:seeAlso` statement).

Extraction Methods and Tools. Features related to schema-level and instance-level links can be accessed through the number of interlinked external instances or datasets. These features can be extracted by tools like voiDge (Böhm et al. 2011).

Licensing features. *Licensing features* comprise the type of license under which a dataset is published, indicating whether reproduction, distribution, modification, or redistribution is permitted.

Extraction Methods and Tools. Features in the licensing group are meant to be augmented manually by the data provider and can be collected from the dataset metadata using tools like ODPW.

Statistical features. This type of features may refer to the size of a dataset, its coverage, average number of triples, property co-occurrence, and others. These characteristics can be measured at the *schema level* (class/properties usage count (in general, per subject and per object) or class/properties hierarchy depth) and at the *instance level* (URI usage per subject (/object), triples having a resource (/blanks) as subject (/object), triples with literals, min(/max/avg.) per data type, etc.).

Extraction Methods and Tools. Features from this group can be extracted automatically by applications like LODStats using statement-streaming approaches (Auer et al. 2012). Example statistics include frequencies of vocabulary usage, average length of literals, and number of namespaces used. ProLOD++ (Abedjan et al. 2014) supports different granularities of statistics using clustering approaches prior to the statistics generation.

Dynamics features. This category reflects the dynamic aspects of a dataset and includes features in the global, instance-specific, and semantics-specific categories. In principle, every dataset feature can be dynamic, i.e., changing over time. However, the dynamic aspects of a dataset are considered as separate features describing data, while nonetheless acknowledging their transversal nature (the fact that they span over several of the feature categories described above). These features are organized into three subcategories. *Global* features concern dynamicity aspects related to life span, stability (as an aggregation of the stability of multiple dataset characteristics), and history of update (frequency, degree, and patterns of change). *Instance-specific features* refer to the level of growth of the entities in a dataset, the stability of the identifiers (URIs), and the links between entities. Finally, *semantic-specific* features reflect changes related to structure, domain, and vocabulary.

Extraction Methods and Tools. Global dynamic features can be extracted automatically from the SPARQL endpoints via tracking changes using broadcasting or query techniques as well as by regular crawling of LOD data sources and statistical analysis of the crawls, e.g., in Dyldo (Käfer et al. 2013). Instance-specific dynamic features include observation of resource referencing through notification services, monitoring of links stability, and time-travel tools to access archived representations of the URIs. Note that the extraction of dynamic features, as well as statistical, qualitative, and links features, would in general require less domain expertize and can be extracted automatically by applications in many cases. To cope with the large scale of data during automatic feature extraction, various sampling techniques can be applied (Fetahu et al. 2014).

Vocabularies

Vocabularies for representing dataset profiles range from general dataset metadata vocabularies to specific vocabularies aimed at representing particular features or feature categories.

The Vocabulary of Interlinked Datasets (VoID) (Alexander et al. 2009) provides a core vocabulary for describing datasets and their links, following a similar rationale as the Data Catalog (DCAT) vocabulary (http://www.w3.org/TR/vocab-dcat/), which is partly derived from Dublin Core. Both are widely used to generate basic profiles (Ben Ellefi et al. 2017) and are commonly extended with more feature-specific vocabularies.

With respect to *quality*, the dataset quality (daQ) vocabulary (http://purl.org/eis/vocab/daq) (Debattista et al. 2014) and the Data Quality Vocabulary (DQV) (https://www.w3.org/TR/vocab-dqv/) provide complementary terms for annotating DCAT dataset descriptions with quality aspects and metrics. Fürber and Hepp (2011) describes the DQM Ontology (http://semwebquality.org/dqm-vocabulary/v1/dqm), a general vocabulary for representing data quality features.

Regarding *provenance*, *voidp* (Omitola et al. 2011) builds on and extends VoID to describe the provenance relationships of data across linked datasets, while the *provenance vocabulary* (http://trdf.sourceforge.net/provenance/ns.html) was developed to describe provenance of Linked Data.

From the perspective of archiving and long-term preservation of data, the *Data Dictionary for Preservation Metadata (PREMIS)* (http://bit.ly/premisOntology) set of terms can be used to describe the provenance of archived, digital objects (e.g., files, bitstreams, aggregations, and datasets).

Most notably, the *PROV Ontology (PROV-O)* (http://www.w3.org/TR/prov-o/) was published as a W3C Recommendation to be a new quasi-standard for representing provenance and is part of a larger *PROV Family of Documents* (Missier et al. 2013) created to support "the widespread publication and use of provenance information of Web documents, data, and resources."

Links as important features of Linked Data datasets are represented through a variety of means, covering both schema-level and entity-level links, e.g., instantiating VoID linksets or using *SKOS* (https://www.w3.org/TR/2009/REC-skos-reference-20090818/) as a formal vocabulary for defining taxonomic and mapping relations among both concepts and entities. In addition, the Expressive and Declarative Ontology Alignment Language (EDOAL) (http://alignapi.gforge.inria.fr/edoal.html) enables the representation of correspondences between entities and concepts in different datasets beyond mere mapping relationships (equivalence, subsumption) using complex formalisms.

General resource metadata vocabularies provide basic features to indicate licensing information, including the *DCMI Metadata Terms* (http://dublincore.org/documents/dcmi-terms), featuring dedicated *license* and *rights* properties.

These are complemented through dedicated *licensing* vocabularies, such as ccREL (REL, or rights expression language) vocabulary (https://wiki.creativecommons.org/wiki/CC_REL), which facilitates the representation of Creative Commons licenses in RDF. Similarly, the *Open Digital Rights Language (ODRL)* vocabulary (http://www.w3.org/community/odrl/two/model/) enables the fine-grained specification of licensing terms (rights, policies, etc.) in a machine-readable format.

To support the representation of dataset *statistics*, vocabularies such as the RDF Data Cube vocabulary (http://www.w3.org/TR/vocab-data-cube), SDMX (http://sdmx.org), or SCOVO (http://vocab.deri.ie/scovo) are used. The VoID guidelines, for instance, recommend the use of SCOVO to share statistical dataset features (Alexander et al. 2009). Auer et al. present LODStats (Auer et al. 2012), a framework for dataset analytics, which introduces a set of 32 statistical features and uses the most recommended combination of VoID and the Data Cube vocabulary.

While there does exist a wealth of methods for assessing characteristics related to the *dynamic* and evolution of datasets, including the *Talis Changeset vocabulary* (http://vocab.org/changeset/schema.html), the Delta vocabulary (http://www.w3.org/2004/delta), RMO introduced by Graube et al. (2014), or the *Triplify Update vocabulary* (http://triplify.org/vocabulary/update), most vocabularies in this area are dedicated to representing the actual evolution of a dataset, rather than higher-level observations about dynamics. A more abstract approach is offered by the *Dataset Dynamics (DaDy) vocabulary* (http://vocab.deri.ie/dady), which allows the representation of more abstract dynamics-related observations for a specific dataset, to be used in conjunction with VoID.

Applications

This section illustrates the use of RDF dataset profiles in several cross-domain applications without being exhaustive.

Data linking applications aim to annotate, disambiguate, and interlink entities and events by often using natural language processing (NLP) techniques and external sources including Linked Data. In this context, popular services include DBpedia Spotlight (Daiber et al. 2013) and Babelfy (Moro et al. 2014). Data linking applications typically use features from the *General* category, such as *topics*, *domains*, or *representative elements*.

Applications for data curation, cleansing, and maintenance rely on or generate profile features in order to improve the overall data quality. This includes the application of statistical methods for outliers detection (correcting errors in numerical values) (e.g., in Paulheim and Bizer 2014), link correction, as well as error detection and correction by using existing links. Features from the *Statistical*, *Provenance* and *Dynamics* categories are largely used in this group of applications. Note that new features can also be generated as an outcome of these applications.

Schema inference applications have been developed recently, aiming at filling the gap generated by the lack of explicit vocabularies or by incomplete specifications (e.g., Paulheim and Bizer 2014; Konrath et al. 2012). *Statistical* characteristics of datasets, together with *Provenance* features, play an important role for this type of applications.

Applications for query answering over distributed data comprise the generation of ordered query plans against the mediated schema on a number of data sources. In order to guide distributed query processing, existing applications rely on indices of varying granularity including schema-level indices and data summaries (e.g., Harth et al. 2010). The majority of existing query applications rely on the *General* and *Statistical* characteristics at the schema and data levels. Finally, quality-aware query applications also take into account features from the *Qualitative* category (e.g., *completeness* and *accuracy*).

Reuse of datasets can become apparent when datasets are linked to from other datasets. In this context, *Links* features can provide an indication of dataset reuse (Endris et al. 2017).

Future Challenges

Overall, applications of the whole spectrum of dataset profile feature categories can be found, including general, qualitative, statistical, and dynamic features discussed in this entry. However, individual applications as well as tools commonly use or support only a very limited number of features. Typical stakeholders of the discussed techniques are both data providers, such as archival organizations, libraries, or individual data hosters, and data consumers, for instance, application developers or (domain-specific) data retrieval and search engines. In addition, with respect to Web-scale dataset discovery, the heterogeneous and fragmented landscape of feature definitions and applied vocabularies hinders the automated interpretation of dataset profiles, requiring further effort for consumers of dataset profiles for interpreting, mapping, and disambiguating existing profiles.

The growing adoption of schema.org markup, particularly of scientific, datasets, driven also by existing community efforts such as the W3C community group on *schema.org for Datasets* (https://www.w3.org/community/schemaorg4datasets/) has led to a widespread availability of semi-structured dataset annotations. However, while this constitutes an unprecedented source of dataset-centric information, the diverse and often poorly structured nature of embedded markup (Yu et al. 2017) poses the need for further processing of such dataset profiles, for instance, through disambiguating, resolving, and augmenting dataset descriptions.

While reconciliation of dataset profiles is a general issue of relevance beyond just embedded Web markup, future work has to provide a stronger emphasis on obtaining, resolving, consolidating, and cleaning datasets profiles from a wide variety of signals, including structured RDF dataset annotations, embedded Web page markup, but also less explicit indicators observable on the Web, for instance, in dataset registries, unstructured Web pages, or scientific publications.

Cross-References

- ▶ Data Profiling
- ▶ Data Quality and Data Cleansing of Semantic Data
- ▶ Federated RDF Query Processing
- ▶ Semantic Interlinking
- ▶ Semantic Search

References

Abedjan Z, Grütze T, Jentzsch A, Naumann F (2014) Profiling and mining RDF data with prolod++. In: Proceedings of the 30th international conference on data engineering, ICDE 2014, Chicago, 31 Mar–4 Apr 2014, pp 1198–1201

Alexander K, Cyganiak R, Hausenblas M, Zhao J (2009) Describing linked datasets – on the design and usage of void, the 'vocabulary of interlinked datasets'. In: WWW 2009 workshop: linked data on the web (LDOW2009), Madrid

Auer S, Demter J, Martin M, Lehmann J (2012) Lodstats – an extensible framework for high-performance dataset analytics. In: Proceedings of the 18th international conference on knowledge engineering and knowledge management, EKAW 2012, Galway City, 8–12 Oct 2012, pp 353–362
Ben Ellefi M, Bellahsene Z, John B, Demidova E, Dietze S, Szymanski J, Todorov K (2017) RDF dataset profiling – a survey of features, methods, vocabularies and applications. Semant Web J
Bizer C, Cyganiak R (2009) Quality-driven information filtering using the WIQA policy framework. J Web Sem 7(1):1–10
Böhm C, Lorey J, Naumann F (2011) Creating void descriptions for web-scale data. J Web Sem 9(3): 339–345
Daiber J, Jakob M, Hokamp C, Mendes PN (2013) Improving efficiency and accuracy in multilingual entity extraction. In: Proceedings of the 9th international conference on semantic systems, I-SEMANTICS 2013, Graz, 4–6 Sept 2013, pp 121–124
Debattista J, Lange C, Auer S (2014) daQ, an ontology for dataset Quality information. In: Proceedings of the workshop on linked data on the web co-located with the 23rd international world wide web conference (WWW 2014), Seoul, 8 Apr 2014
Endris KM, Giménez-García JM, Thakkar H, Demidova E, Zimmermann A, Lange C, Simperl E (2017) Dataset reuse: an analysis of references in community discussions, publications and data. In: Proceedings of the ninth international conference on knowledge capture (K-CAP 2017)
Fetahu B, Dietze S, Nunes BP, Casanova MA, Taibi D, Nejdl W (2014) A scalable approach for efficiently generating structured dataset topic profiles. In: Proceedings of the 11th ESWC conference 2014, Anissaras, 25–29 May 2014, pp 519–534
Fürber C, Hepp M (2011) Towards a vocabulary for data quality management in semantic web architectures. In: Proceedings of the 1st international workshop on linked web data management, LWDM'11. ACM, New York, pp 1–8
Graube M, Hensel S, Urbas L (2014) R43ples: revisions for triples – an approach for version control in the semantic web. In: Proceedings of the 1st workshop on linked data quality co-located with 10th international conference on semantic systems, LDQ@SEMANTiCS 2014, Leipzig, 2 Sept 2014
Harth A, Hose K, Karnstedt M, Polleres A, Sattler KU, Umbrich J (2010) Data summaries for on-demand queries over linked data. In: Proceedings of the 19th international conference on world wide web, WWW'10. ACM, New York, pp 411–420
Käfer T, Abdelrahman A, Umbrich J, O'Byrne P, Hogan A (2013) Observing linked data dynamics. In: Proceedings of the 10th ESWC conference, Montpellier, 26–30 May 2013, pp 213–227
Konrath M, Gottron T, Staab S, Scherp A (2012) Schemex – efficient construction of a data catalogue by stream-based indexing of linked data. J Web Sem 16: 52–58
Missier P, Belhajjame K, Cheney J (2013) The W3C PROV family of specifications for modelling provenance metadata. In: Joint 2013 EDBT/ICDT conferences, EDBT'13. Proceedings, Genoa, 18–22 Mar 2013, pp 773–776
Moro A, Raganato A, Navigli R (2014) Entity linking meets word sense disambiguation: a unified approach. TACL 2:231–244
Omitola T, Zuo L, Gutteridge C, Millard IC, Glaser H, Gibbins N, Shadbolt N (2011) Tracing the provenance of linked data using void. In: Proceedings of the international conference on web intelligence, mining and semantics, WIMS'11. ACM, New York, pp 17:1–17:7
Paulheim H, Bizer C (2014) Improving the quality of linked data using statistical distributions. Int J Semant Web Inf Syst 10(2):63–86
Umbrich J, Neumaier S, Polleres A (2015) Quality assessment and evolution of open data portals. In: Proceedings of the 3rd international conference on future internet of things and cloud, FiCloud 2015, Rome, 24–26 Aug 2015, pp 404–411
Yu R, Gadiraju U, Fetahu B, Dietze S (2017) Fusem: query-centric data fusion on structured web markup. In: Proceedings of the 2017 IEEE 33nd international conference on data engineering (ICDE). IEEE
Zaveri A, Rula A, Maurino A, Pietrobon R, Lehmann J, Auer S (2016) Quality assessment for linked data: a survey. Semant Web 7(1):63–93

RDF Formats

▶ RDF Serialization and Archival

RDF Graph Embeddings

▶ Knowledge Graph Embeddings

RDF Serialization and Archival

Javier D. Fernández[1] and Miguel A. Martínez-Prieto[2]
[1]Complexity Science Hub Vienna, Vienna University of Economics and Business, Vienna, Austria
[2]Department of Computer Science, Universidad de Valladolid, Valladolid, Spain

Synonyms

RDF formats; RDF syntaxes

Definitions

RDF serialization is the process of writing down RDF graphs into a machine-readable format. RDF formats mainly differ in the concrete syntax to serialize RDF statements (called "triples") and how to group or nest a set of statements, influencing the amount of storage space and bandwidth required for preserving and exchanging such data. These differences can be rather marginal for small RDF graphs, where the selection of a particular format is mostly driven by user preferences, the set of tools managing the RDF format, and the interoperability with other applications. In contrast, choosing an adequate serialization format can affect the overall performance and present important scalability issues when managing Big Semantic Data collections.

Additional challenges arise in scenarios where triples must be annotated with information about their context, such as provenance, trust, or quality information, to name but a few. The most standard solution in RDF is to consider *named graphs*, i.e., different RDF graphs are managed under a single RDF dataset. Diverse RDF formats have been proposed to cover this scenario and serialize annotated statements (called "quads"), at the cost of paying additional costs to represent triples that can be repeated across graphs.

This situation is particularly challenging when different versions of an RDF graph must be preserved, given that graphs can be near-copies of others. This problem is commonly referred to as *RDF archival*, where specific archival policies have been proposed in recent literature.

This entry provides a historical review of RDF serialization formats to understand their evolution over the years. Basic features are covered for each format, paying special attention to its capabilities for quad serialization. XML-based and text-oriented formats are first introduced to illustrate how RDF was originally used for metadata description. Their limitations led to JSON-based syntaxes, which overcome some processing challenges, but do not scale to the high demanding needs of Big Semantic Data management. This fact motivates the proposal of binary formats, able to deal with the storing and exchanging needs of large RDF collections. Finally, RDF archival proposals are surveyed and we conclude presenting open research trends.

Overview

RDF *(Resource Description Framework)* (Schreiber and Raimond 2014) proposes a logical model for expressing information about physical (e.g., people, buildings, vehicles, or pictures, among others) and abstract (e.g., films, songs, cities, etc.) resources. The information is represented as ternary relations, called RDF triples (or statements), which are organized into hyperlinked clouds, referred to as RDF graphs. The resulting knowledge is typically oriented for machine consumption. In the following, the main RDF concepts are briefly introduced.

RDF Triple. RDF statements are built on a simple ⟨subject, predicate, object⟩ structure called *RDF triple* (aka statement), which sets a particular value (the "object") for a given feature ("predicate") of the resource ("subject") being described. For instance, an informal representation of a triple which sets the birth date of the singer Bruce Springsteen can be ⟨Bruce Springsteen, is born on, 1949-09-23⟩.

The RDF data model (Cyganiak et al. 2014) establishes some restrictions about the universe of possible values for each component of a triple. Thus, a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where I stands for IRIs, B for blank nodes, and L for literals. These mutual disjoint sets of RDF terms are described as follows:

IRIs. International Resource Identifiers (IRIs) are used to identify resources in RDF. For instance, <http://example.org/Springsteen> is used to name the aforementioned resource about Bruce Springsteen. Note that IRIs can also be used to identify predicate and object values of a triple; e.g., <http://example.org/p/birthDate> is a valid IRI to identify the corresponding "birth date" property. IRIs are global identifiers, so they can be reused to provide additional information about a resource or to convey the same meaning, e.g., a birth date feature in a different context.

Blank Nodes. RDF uses blank nodes (also called anonymous nodes) to declare the existence of a resource without using a particular IRI. Blank nodes can play both subject and object roles, while they never mean that the IRI is unknown for the corresponding resource. In turn, the scope of the blank node is limited to the RDF graph where it is used.

Literals. Final values (numbers, names, dates, etc.) are expressed as RDF literals, always used as objects. Literals are declared by default as strings (a language tag can optionally be associated in this case), but other datatypes can be used. The value "1949-09-23" is an example of an RDF literal.

Figure 1 illustrates the above RDF concepts. It comprises three triples that describe a new resource about Bruce Springsteen, identified by the aforementioned IRI <http://example.org/Springsteen>. Two of these triples set his birth name and birth date (using IRIs <http://example.org/p/name> and <http://example.org/p/birthDate>, respectively), i.e., `"Bruce Frederick Joseph Springsteen"` and "1949-09-23." The third triple connects the new resource with an existing description of Bruce Springsteen in DBpedia (<http://dbpedia.org/page/Bruce_Sprinsgteen>), a conversion of Wikipedia to RDF. Note that the predicate reuses the sameAs property, described in the OWL ontology.

RDF is traditionally modeled as a labeled directed graph (as seen in the previous figure) because it provides an easy-to-understand (visual) explanation of the RDF data. This fact motivates the adoption of this concept as part of the RDF model.

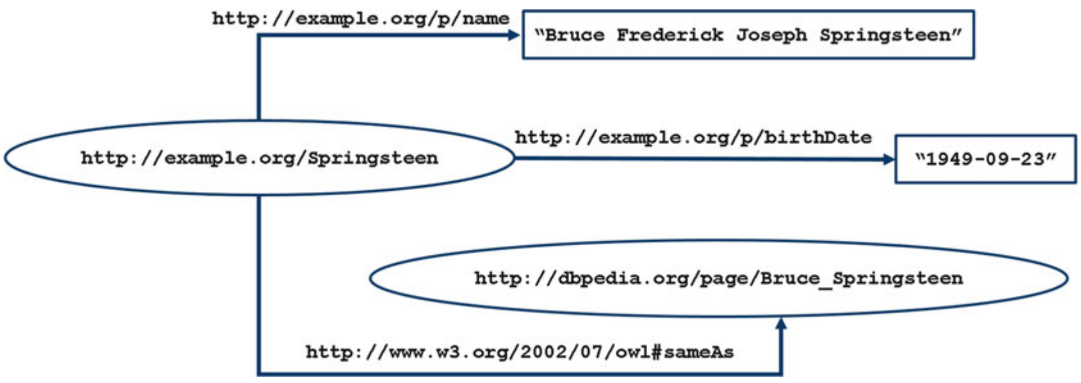
RDF Graph. An RDF graph *G* is a set of triples declared under the same scope. Thus, a triple belongs to an RDF graph, and a graph contains a well-determined set of triples.

It is worth noting that an RDF graph is only a "mental model," and its triples must be serialized for preservation or exchanging purposes. Each serialization format has its particular features, but all ensure RDF graphs to be effectively written down.

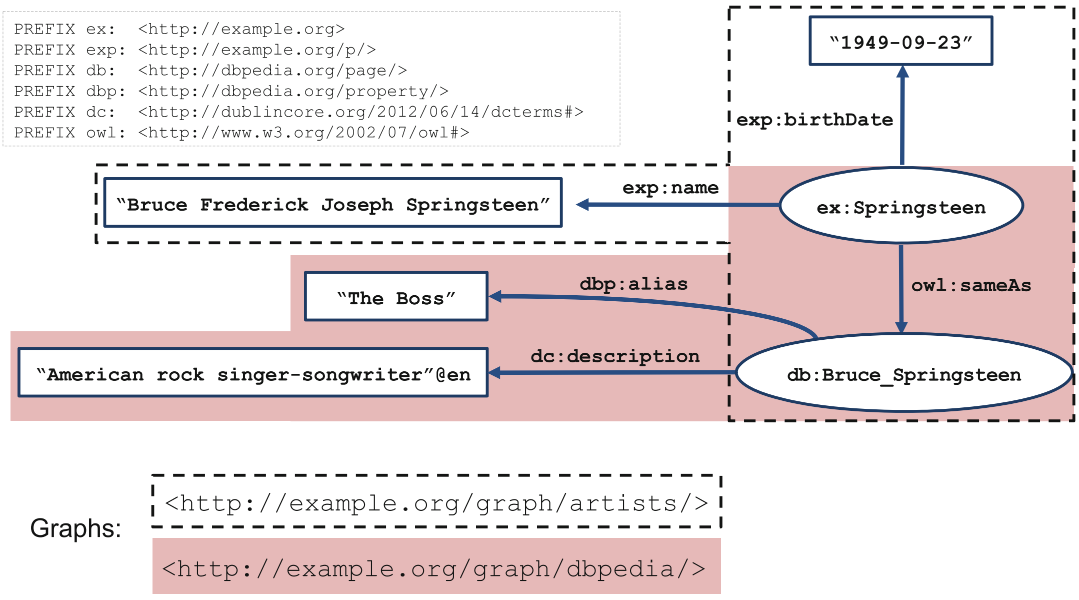
RDF 1.1 extends the original model to support grouping RDF graphs within a single RDF dataset, enabling triples from different contexts to be managed together.

RDF Dataset. An RDF dataset *D* is a collection of RDF graphs, where one of them is considered the "default graph." The dataset contains zero or more "named graphs." Each named graph is a pair consisting of an IRI or a blank node (the graph name) and an RDF graph. Graph names are unique within the RDF dataset, and blank nodes can be shared between graphs.

Figure 2 shows an RDF dataset which comprises two RDF graphs (note that a prefix notation is used to compact their IRIs). The first graph


http://example.org/p/name
"Bruce Frederick Joseph Springsteen"
http://example.org/Springsteen
http://example.org/p/birthDate
"1949-09-23"
http://dbpedia.org/page/Bruce_Springsteen
http://www.w3.org/2002/07/owl#sameAs

RDF Serialization and Archival, Fig. 1 RDF graph which comprises three triples about *Bruce Springsteen*


PREFIX ex: <http://example.org>
PREFIX exp: <http://example.org/p/>
PREFIX db: <http://dbpedia.org/page/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dc: <http://dublincore.org/2012/06/14/dcterms#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
"1949-09-23"
exp:birthDate
"Bruce Frederick Joseph Springsteen"
exp:name
ex:Springsteen
"The Boss"
dbp:alias
owl:sameAs
"American rock singer-songwriter"@en
dc:description
db:Bruce_Springsteen
Graphs:
<http://example.org/graph/artists/>
<http://example.org/graph/dbpedia/>

RDF Serialization and Archival, Fig. 2 RDF dataset which comprises two RDF graphs

(dashed) is identified by the IRI <http://example.org/graph/artists> and includes the set of three triples showed in Fig. 1. The new named graph (solid background) declares three more triples about Bruce Springsteen, one of them shared with the original graph: ⟨ex:Springsteen, owl:sameAs, db:Bruce_Springsteen⟩.

The notion of RDF dataset also applies to a logical level. However, serializing triples from a dataset brings an additional requirement, as their context must be preserved.

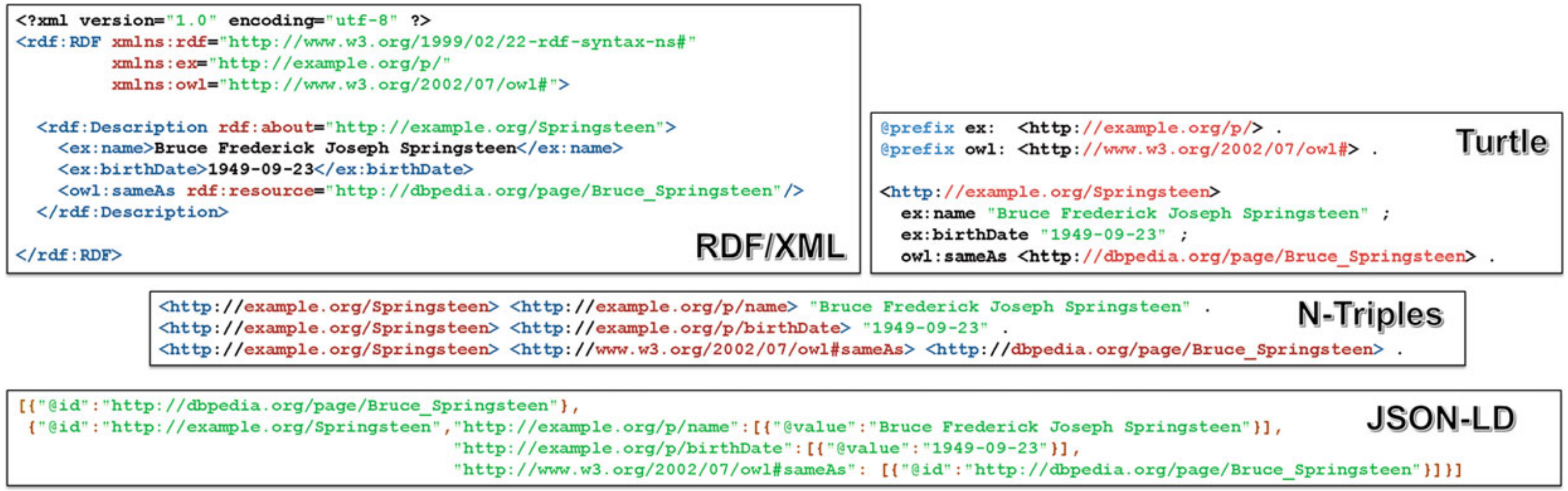
RDF Quad. An RDF quad is an extended statement that includes the corresponding triple and the name of the graph that declares it (aka context). More formally, an RDF quad q is a quadruple ⟨subject, predicate, object, graph⟩, where graph refers to the name of a graph which exists in the dataset. Thus, a quad $(s, p, o, g) \in (I \cup B) \times I \times (U \cup B \cup L) \times (I \cup B)$.

Serialization Formats

The RDF model describes the previous concepts using an abstract syntax, but it does not restrict how they are effectively serialized. Thus, RDF data can be written down in different ways, while several serialization formats are standards and widely accepted by the Semantic Web community. These formats allow RDF graphs to be effectively serialized, but only some of them are able to cover particular RDF dataset needs.

`RDF/XML` (Gandon and Schreiber 2014) was released hand in hand with the initial W3C RDF Recommendation. In early dates, RDF/XML was meant to be an ideal first serialization for RDF graphs as it could leverage all XML-based solutions. However, RDF/XML overloads the representation with verbose human-focused information, which can serve the intended exchanging purposes, but only on a small scale. Nonetheless, RDF/XML includes some naive compacting features, such as the possibility to (i) implicitly create blank nodes without giving a concrete identifier, (ii) omit nodes and place values as property attributes in XML, (iii) abbreviate IRI references via base IRIs (namespaces) and relative references, and (iv) create collections to define a set of terms related to a subject.

Figure 3 shows an example of an RDF/XML serialization that encodes triples from Fig. 1. In practice, the result is an XML document, which can be parsed, processed, and queried using well-established technologies (DOM, XPath, XSLT,


```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ex="http://example.org/p/"
         xmlns:owl="http://www.w3.org/2002/07/owl#">

  <rdf:Description rdf:about="http://example.org/Springsteen">
    <ex:name>Bruce Frederick Joseph Springsteen</ex:name>
    <ex:birthDate>1949-09-23</ex:birthDate>
    <owl:sameAs rdf:resource="http://dbpedia.org/page/Bruce_Springsteen"/>
  </rdf:Description>

</rdf:RDF>
```
RDF/XML

```
@prefix ex:  <http://example.org/p/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

<http://example.org/Springsteen>
  ex:name "Bruce Frederick Joseph Springsteen" ;
  ex:birthDate "1949-09-23" ;
  owl:sameAs <http://dbpedia.org/page/Bruce_Springsteen> .
```
Turtle

```
<http://example.org/Springsteen> <http://example.org/p/name> "Bruce Frederick Joseph Springsteen" .
<http://example.org/Springsteen> <http://example.org/p/birthDate> "1949-09-23" .
<http://example.org/Springsteen> <http://www.w3.org/2002/07/owl#sameAs> <http://dbpedia.org/page/Bruce_Springsteen> .
```
N-Triples

```
[{"@id":"http://dbpedia.org/page/Bruce_Springsteen"},
 {"@id":"http://example.org/Springsteen","http://example.org/p/name":[{"@value":"Bruce Frederick Joseph Springsteen"}],
                                          "http://example.org/p/birthDate":[{"@value":"1949-09-23"}],
                                          "http://www.w3.org/2002/07/owl#sameAs": [{"@id":"http://dbpedia.org/page/Bruce_Springsteen"}]}]
```
JSON-LD

RDF Serialization and Archival, Fig. 3 Serializations of triples from Fig. 1

etc.). However, its document orientation is an important weakness to deal with large amounts of RDF triples. Besides, it does not support named graphs.

Trix (Carroll and Stickler 2004) proposes another XML syntax for RDF which organizes triples by graphs, allowing multiple graphs to be serialized into the same document. It is a first approximation to an RDF dataset serialization, but the resulting format shows the same drawbacks that RDF/XML.

XML-based formats have lost relevance, and their usage is limited to small RDF graph serializations (e.g., descriptive metadata about a Web page).

`N3 (Notation3)` (Berners-Lee and Connolly 2011) is a format designed with human readability in mind. Although it may makes sense in the first times of RDF, managing and processing Big Semantic Data are far from any human capability. However, this format breaks with the XML predominance and introduces some interesting constructors which tackle particular RDF features.

sN3 proposes the use of namespaces, as in XML. It is an effective compaction mechanism which allows relative IRIs to be declared to their corresponding namespace. On the other hand, N3 also introduces constructors for triples encoding in the form of adjacency lists: predicate lists allow subjects to be written only once for all triples containing it, while object lists concatenate all object values related with a pair ⟨`subject, predicate`⟩, which is written once.

This format proposes some other constructors which goes beyond the needs of RDF serialization, making the format relatively complex for such purpose. N3 does not support quads.

`N-Triples` (Becket 2014) is an extremely simple line-based syntax, easy to parse and generate. In essence, the subject, predicate, and object terms are separated by a white space, and the triple is terminated with a "." followed by a new line. IRIs are enclosed in "<" and ">" and literals in " ″ ," and blank nodes start with "_:." Figure 3 shows an N-Triples serialization that basically lists the corresponding triples. Note that N-Triples writes down each full term as many times as it is used in a triple, resulting in a simple but extremely verbose serialization due to long-term repetitions. As a result, N-Triples files need much more space than others, which can result in scalability issues for Big Semantic Data management.

On the other hand, N-Triples can be easily extended to support quad serialization. It only needs the graph name to be appended to the triple. `N-Quads` (Carothers 2014) formalizes this approach, featuring the same characteristics and limitations as N-Triples.

`Turtle` (Beckett et al. 2014) is a widely used format that exploits the previous experience of N3 and N-Triples. On the one hand, it delimits the expressive power of N3 to only serialize valid RDF graphs. On the other hand, it addresses N-Triples drawbacks to consolidate a more practical format.

Figure 3 also shows a Turtle excerpt that illustrates some of its more relevant features. For instance, it shows the use of namespaces. Note that each one is declared by the `@prefix` constructor, while IRIs in the terms are rewritten in relative form to their corresponding namespaces. The figure also illustrates the predicate list encoding proposed in N3; e.g., http://example.org/Springsteen is written once, but it plays the role of a subject for three different triples. Turtle supports object lists too, and it introduces more constructors and different kinds of syntactic sugar to alleviate RDF verbosity.

Although Turtle is a popular format, it does not support quads. As in the previous case, a new format, called `TriG` (Bizer and Cyganiak 2014), extends Turtle to allow RDF dataset serialization. It basically encloses triples that belong to each named graph in the dataset.

`JSON-LD` (Sporny et al. 2014) exploits JSON features to serialize RDF. It comes with the advantage of using a well-established scheme that is easy to parse and widely accepted by Web APIs. The main focus, then, is to be easy for humans to read and write and easy for machines to parse and generate automatically. JSON-LD is designed to be usable directly as JSON, with no knowledge of RDF. Note that JSON-LD supports named graphs natively, and it is gaining increasingly attention by the community.

Key Research Findings

The above serialization formats have been successfully used for managing small- and medium-sized RDF graphs. However, the steady adoption of RDF, in particular in the context of linked data (Bizer et al. 2009), brings larger graphs including hundreds of millions and even billion triples. For instance, the latest version of DBpedia (2016–10), an RDF conversion of Wikipedia, consists of roughly 13 billion triples, and LOD Laundromat (Beek et al. 2014), a service crawling RDF datasets, reports that around 4000 datasets contain more than 1 million triples.

In addition, named graphs are increasingly incorporated to consolidate complex RDF datasets. However, formats for quads are less mature and also suffer from the lack of scalability. This problem is particularly challenging when the corresponding RDF dataset is a historical archive of a graph, containing its different states over the time.

This section delves into detail of the most innovative binary serialization formats, designed with volume issues in mind in order to solve the aforementioned scalability issues. Some of them also cover quad management, although managing context information is a challenge by itself, which is also reviewed below. Finally, RDF archival foundations are introduced, summarizing the most recent approaches.

Binary Serialization and Compression

Traditional RDF formats were not designed for a scenario of large-scale and machine-understandable Web of data. Their syntaxes have constructors which organizes RDF statements in a human-readable way that adds unnecessary overheads for storing, exchanging, and consuming RDF graphs. Although this scalability issue can be partially solved through universal compression (e.g., gzip or bzip2) over such formats, specific RDF binary serializations and compressors have been also proposed. These tailored solutions mostly focus on taking advantage of particular features of RDF data in order to reduce the verbosity and produce important space savings at large scale. In the following, the three most prominent binary serializations are briefly reviewed: *HDT*, *RDF Binary*, and *RDF4J*. We then list solutions focused on streaming and provide a summary of RDF compression techniques to provide a big picture of the current state of the art (the interested reader can find a chapter specifically devoted to RDF compression).

The `HDT` (Fernández et al. 2011, 2013) format proposes a binary syntax for RDF data focused on producing very compact serializations to speed up data exchange, but also efficient data parsing and access. HDT minimizes the repetition of terms (IRIs, blank nodes, and literals) using the so-called HDT Dictionary, which assigns a numerical ID to each different term. Then, the graph structure of the dataset is managed as a graph of (term) IDs, in the HDT Triples component. Both dictionary and triples components are then compacted (e.g., looking for common string prefixes in the terms) and partially indexed. HDT is one of the most widespread RDF binary formats, mainly due to the HDT adoption as a compact data store for LOD Laundromat (Beek et al. 2014), and the data back end of lightweight APIs such as Triple Pattern Fragments (Verborgh et al. 2016).

HDT traditionally focuses on representing single RDF graphs. A recent approach, named HDTQ (Fernández et al. 2018), extends HDT to represent named graphs, keeping compact and retrieval features.

The `RDF binary` format (RDF Binary 2017) is an alternative solution proposed by the well-known Jena semantic framework. It consists of very simple mappings to encode triples in Apache Thrift (Apache Thrift 2017), which provides a scalable cross-language platform. In this case, rather than compactness, RDF binary mostly focuses on avoiding to parse the textual RDF triples; hence, the overall processing is sped up. RDF binary supports both RDF graphs and RDF datasets (named graphs) encoded as a stream of quads.

The `RDF4j` (RDF4j 2017) binary format is proposed and used within the Eclipse RDF4J framework. The RDF4j format partially combines both previous strategies. On the one hand, it mostly tackles parsing and processing efficiency, providing a concrete syntax to delimit the extent of each term and triple. On the other hand, it allows for an in-line declaration of a dictionary, where a term is mapped to an ID which can be referred in another triple. Nonetheless, terms are not compressed themselves (e.g., using prefixes such as in HDT); hence, only partial compression is achieved.

Compression is another way of serializing RDF. As explained, combining universal compression and any serialization format is a common practice, but different compressors have been designed from the scratch to deal with particular RDF requirements. RDF compressors can be classified into *physical* and *logical* compressors. Physical compressors (Fernández et al. 2013; Swacha and Grabowski 2015; Álvarez-García et al. 2014; Brisaboa et al. 2015) exploit symbolic/syntactic redundancy, removing term repetitions and compacting repetitive subgraph structures underlying to the dataset. In contrast, logical compressors (Iannone et al. 2005; Meier 2008; Joshi et al. 2013; Venkataraman and Sreenivasa Kumar 2015) focus on semantic-based redundancy, avoiding to represent triples that can be inferred from others in the RDF graph.

In addition, diverse binary formats and compressors have been proposed for RDF streams, i.e., a continuous flow of RDF data. In this case, the challenge consists of exploiting the trade-offs between the space savings achieved by the format and the latency introduced in the creation and parsing processes. Streaming HDT (Hasemann et al. 2012) adapts HDT to simplify the process by restricting the carried metadata and the maximum length of the dictionary; hence, shorter IDs are used. RDSZ (Fernández et al. 2014b) uses differential encoding to compact the similarities between consecutive triples in the stream. ERI (Fernández et al. 2014a) is an RDF stream compressor that adapts the W3C Efficient XML Interchange (EXI) format (Schneider et al. 2014) for RDF data. Note that EXI encoding can also be directly applied over an RDF/XML or JSON serialization. PatBin (Lhez et al. 2017) and FSSD (Karim et al. 2017) perform dictionary-based compression together with pattern-based encoding.

Context Information

As stated, graph names are increasingly used to capture additional information such as trust, provenance, temporal information and other annotations (Carroll et al. 2005; Zimmermann et al. 2012). Although there exist standard RDF syntaxes (such as N-Quads, Trig or JSON-LD) that represent RDF named graphs, serializing annotated RDF data (quads) efficiently remains an open challenge.

In spite of general approaches, such as AnQL (Zimmermann et al. 2012), most solutions focus on managing provenance information, as this is at the core of the linked data distributed philosophy (Bizer et al. 2009). Besides the aforementioned named graphs and the standard RDF reification (Schreiber and Raimond 2014), i.e., using the RDF vocabulary (*rdf:Statement, rdf:subject, rdf:predicate, and rdf:object*) to refer to statements, the main proposals are singleton properties (Nguyen et al. 2014) and N-ary relations (Noy et al. 2006). The former introduces unique predicates that are then annotated with the metadata of the triple it belongs to. The latest, used in Wikidata, represents a relation between a subject and object with a new resource, which is then connected to the subject, on the one hand, and predicate and object, on the other. Further information can be attached to the new resource in order to annotate the statement.

In addition, two recent solutions have been proposed. (Hartig 2017) extends RDF with a notion of embedded triples (encoded between '$\ll$' and '$\gg$'), which can be directly used as subject or object of other triples. NdFluents (Giménez-García et al. 2017) creates unique versions of the subject and the object for each annotated triple, which are then linked to a context resource and to the original subject and object resources.

RDF Archival

RDF archival is a particular instance of the problem of managing context information. In this case, the context is set by the moment when a new version of an RDF graph is released. In general, RDF data are not static but evolve naturally, without centralized monitoring nor further advise, following the scale-free nature of the Web. Thus, RDF archiving emerges as a novel challenge aimed at assuring quality and traceability of RDF data over time.

On a high level, the World Wide Web Consortium (W3C) provides basic guidelines on how to perform data versioning on datasets published in the Web (Lóscio et al. 2017). The set of recommendations includes (i) providing a version indicator (e.g., via *owl:versionInfo*); (ii) serving different versions via the Memento framework (de Sompel et al. 2010), which can provide access to prior states of RDF resources using datetime negotiation in HTTP; and (iii) providing the changes made in each version. Nonetheless, these recommendations are generic and do not restrict how RDF data versions are stored or queried across time. Initial works on RDF archiving policies and systems are starting to address these issues, proposing different solutions to efficiently archive and query different versions of RDF data.

Main efforts on RDF archiving fall in one of the following four storage strategies: *independent copies (IC)* and *change-based (CB)* and *timestamp-based (TB)* and *hybrid-based (HB)* approaches.

Independent copies (IC) (Klein et al. 2002; Noy and Musen 2004) is the most naive approach where each version (aka snapshot) is managed as a different, complete graph. On the one hand, IC faces scalability problems as static information is duplicated across the versions. In addition, some operations such as knowing the difference between versions require non-negligible processing efforts. On the other hand, version materialization (retrieve certain version) is as efficient as querying a single snapshot.

Change-based approach (CB) (Volkel et al. 2005; Dong-Hyuk et al. 2012; Zeginis et al. 2011) partially addresses the space issues of IC by storing the differences (deltas) between versions. In contrast, CB requires additional computational costs for retrieving a particular version given that deltas need to be propagated.

Timestamp-based approach (TB) (Cerdeira-Pena et al. 2016; Gutierrez et al. 2007; Zimmermann et al. 2012) annotates each triple with its temporal validity, i.e., the version. Compression techniques can be used to minimize the space overheads, e.g., using self-indexes, such as in v-RDFCSA (Cerdeira-Pena et al. 2016), or delta compression in B+Trees (Zaniolo 2016).

Hybrid-based approaches (HB) (Stefanidis et al. 2014; Neumann and Weikum 2010; Zaniolo 2016) combine previous policies to inspect other space/performance trade-offs. In particular, the hybrid IC/CB approach (Dong-Hyuk et al. 2012; Meinhardt et al. 2015; Stefanidis et al. 2014) follows a CB solution where full version materialization is additionally provided in some intermediate steps; hence, delta propagation is mitigated. In contrast, other practical approaches (Graube et al. 2014; Neumann and Weikum 2010; Vander Sander et al. 2013; Zaniolo 2016) follow a TB/CB approach in which triples can be time-annotated only when they are added or deleted. Although this reduces the space needs (as it manages less annotations), version materialization requires to rebuild the delta similarly to CB.

Future Directions for Research

As a result of standardization efforts by the Semantic Web community, there are many diverse standard "plain" RDF serializations available. Despite potential future trends that may result in adaptations for RDF (such as JSON-LD, adapted from JSON), most research efforts focus on efficient representation of annotated triples, in particular to model provenance information (Giménez-García et al. 2017; Hartig 2017).

RDF binary formats and compression have also emerged as active research and development fields over the past years. The main reason is that (i) current plain RDF formats are dominated by a human-centric view and suffer from scalability problems at large scale and (ii) general compressed solutions still miss some types of redundancy underlying to RDF data. In this regard, there is still room for hybrid compressors leveraging syntactic and semantic redundancies. Then, RDF self-indexing (i.e., compressed and indexed RDF data) is still a main direction for research, in particular in the unexplored field of RDF streaming.

Finally, the community is just starting to face serious scalability issues for RDF archival. In the absence of a scalable archival approach at Web scale, RDF data change and vanish without further notice nor trace of previous versions. Future directions in this regard include further research on scalable archival methods (potentially distributed) as well as efficient mechanisms to resolve structured cross-time queries.

Cross-References

▶ RDF Compression

References

Álvarez-García S, Brisaboa N, Fernández JD, Martínez-Prieto MA, Navarro G (2014) Compressed vertical partitioning for efficient RDF management. Knowl Inf Syst 44(2):439–474

Apache Thrift (2017) Apache thrift. https://thrift.apache.org/

Becket D (2014) RDF 1.1 N-Triples: a line-based syntax for an RDF graph. W3C recommendation. https://www.w3.org/TR/n-triples/

Beckett D, Berners-Lee T, Prud'hommeaux E, Carothers G (2014) RDF 1.1 turtle: terse RDF triple language. W3C recommendation. https://www.w3.org/TR/turtle/

Beek W, Rietveld L, Bazoobandi HR, Wielemaker J, Schlobach S (2014) LOD laundromat: a uniform way of publishing other people's dirty data. In: 13th international semantic web conference (ISWC), pp 213–228

Berners-Lee B, Connolly D (2011) Notation3 (N3): a readable RDF syntax. W3C team submission. https://www.w3.org/TeamSubmission/n3/

Bizer C, Cyganiak R (2014) RDF 1.1 TriG: RDF dataset language. W3C recommendation. https://www.w3.org/TR/trig/

Bizer C, Heath T, Berners-Lee T (2009) Linked data-the story so far. Int J Semant Web Inf Syst 5(3):1–22

Brisaboa N, Cerdeira-Pena A, Fariña, Navarro G (2015) A compact RDF store using suffix arrays. In: 22nd international symposium on string processing and information retrieval (SPIRE), pp 103–115

Carothers G (2014) RDF 1.1 N-Quads: A Line-based syntax for an RDF dataset. W3C recommendation. https://www.w3.org/TR/n-quads/

Carroll J, Stickler P (2004) TriX : RDF triples in XML. Technical report, Digital Media Systems Laboratory, HP Laboratories Bristol

Carroll JJ, Bizer C, Hayes P, Stickler P (2005) Named graphs, provenance and trust. In: Proceedings of the 14th international conference on World Wide Web. ACM, pp 613–622

Cerdeira-Pena A, Farina A, Fernández JD, Martınez-Prieto MA (2016) Self-indexing RDF archives. In: Proceeding of DCC

Cyganiak R, Wood D, Lanthaler M (2014) RDF 1.1 concepts and abstract syntax. W3C recommendation. http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/

de Sompel HV, Sanderson R, Nelson ML, Balakireva L, Shankar H, Ainsworth S (2010) An HTTP-based versioning mechanism for linked data. In: Proceeding of LDOW

Dong-Hyuk I, Sang-Won L, Hyoung-Joo K (2012) A version management framework for RDF triple stores. Int J Softw Eng Know 22(1):85–106

Fernández JD, Martínez-Prieto MA, Gutiérrez C, Polleres A (2011) Binary RDF representation for publication and exchange (HDT). W3C member submission. http://www.w3.org/Submission/HDT/

Fernández JD, Martínez-Prieto MA, Gutiérrez C, Polleres A, Arias M (2013) Binary RDF representation for publication and exchange. J Web Semant 19:22–41

Fernández JD, Llaves A, Corcho O (2014a) Efficient RDF interchange (ERI) format for RDF data streams. In: 13th international semantic web conference (ISWC), pp 244–259

Fernández N, Arias J, Sánchez L, Fuentes-Lorenzo D, Corcho Ó (2014b) RDSZ: an approach for lossless RDF stream compression. In: 11th European conference on the semantic web (ESWC), pp 52–67

Fernández JD, Martínez-Prieto MA, Polleres A, Reindorf J (2018) HDTQ: managing RDF datasets in compressed space. In: European semantic web conference

Gandon F, Schreiber G (2014) RDF 1.1 XML syntax. W3C recommendation. https://www.w3.org/TR/rdf-syntax-grammar/

Giménez-García JM, Zimmermann A, Maret P (2017) Ndfluents: an ontology for annotated statements with inference preservation. In: European semantic web conference. Springer, pp 638–654

Graube M, Hensel S, Urbas L (2014) R43ples: revisions for triples. In: Proceeding of LDQ, vol CEUR-WS 1215, paper 3

Gutierrez C, Hurtado C, Vaisman A (2007) Introducing time into RDF. IEEE Trans Knowl Data Eng 19(2):207–218

Hartig O (2017) Foundations of RDF* and SPARQL* – an alternative approach to statement-level metadata in RDF. In: Proceeding of AMW

Hasemann H, Kroller A, Pagel M (2012) RDF provisioning for the internet of things. In: 3rd international conference on the internet of things (IOT), pp 143–150
Iannone L, Palmisano I, Redavid D (2005) Optimizing RDF storage removing redundancies: an algorithm. In: 18th international conference on industrial and engineering applications of artificial intelligence and expert systems (IEA/AIE), pp 732–742
Joshi A, Hitzler P, Dong G (2013) Logical linked data compression. In: 10th extended semantic Web conference (ESWC), pp 170–184
Karim F, Vidal ME, Auer S (2017) Efficient processing of semantically represented sensor data. In: 13th international conference on Web information systems and technologies (WEBIST), pp 252–259
Klein M, Fensel D, Kiryakov A, Ognyanov D (2002) Ontology versioning and change detection on the Web. In: Proceeding of EKAW, pp 197–212
Lhez J, Ren X, Belabbess B, Curé O (2017) A compressed, inference-enabled encoding scheme for RDF stream processing. In: 14th European conference on the semantic Web (ESWC), pp 79–93
Lóscio BF, Burle C, Calegari N (2017) Data on the web best practices. W3C recommendation 31 Jan 2017
Meier M (2008) Towards rule-based minimization of RDF graphs under constraints. In: 2nd international conference on web reasoning and rule systems (RR), pp 89–103
Meinhardt P, Knuth M, Sack H (2015) Tailr: a platform for preserving history on the web of data. In: Proceeding of SEMANTiCS. ACM, pp 57–64
Neumann T, Weikum G (2010) x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. Proc VLDB Endow 3(1–2):256–263
Nguyen V, Bodenreider O, Sheth A (2014) Don't like RDF reification? Making statements about statements using singleton property. In: Proceedings of the 23rd international conference on World Wide Web. ACM, pp 759–770
Noy NF, Musen MA (2004) Ontology versioning in an ontology management framework. IEEE Intell Syst 19(4):6–13. https://doi.org/10.1109/MIS.2004.33
Noy N, Rector A, Hayes P, Welty C (2006) Defining n-ary relations on the semantic web. W3C working group note 12(4)
RDF Binary (2017) RDF binary using apache thrift. https://jena.apache.org/documentation/io/rdf-binary.html
RDF4j (2017) Rdf4j binary RDF format. http://docs.rdf4j.org/rdf4j-binary/
Schneider J, Kamiya T, Peintner D, Kyusakov R (2014) Efficient XML interchange (EXI) Format 1.0. W3C recommendation
Schreiber G, Raimond Y (2014) RDF 1.1 primer. W3C working group note. https://www.w3.org/TR/rdf11-primer/
Sporny M, Longley D, Kellogg G, Lanthaler M, Lindström N (2014) JSON-LD 1.0: a JSON-based serialization for linked data. W3C recommendation. https://www.w3.org/TR/json-ld/
Stefanidis K, Chrysakis I, Flouris G (2014) On designing archiving policies for evolving RDF datasets on the Web. In: Proceeding of ER, pp 43–56
Swacha J, Grabowski S (2015) OFR: an efficient representation of RDF datasets. In: 4th symposium on languages, applications and technologies (SLATE), pp 224–235
Vander Sander M, Colpaert P, Verborgh R, Coppens S, Mannens E, Van de Walle R (2013) R&Wbase: git for triples. In: Proceeding of LDOW
Venkataraman G, Sreenivasa Kumar P (2015) Horn-rule based compression technique for RDF data. In: 30th annual ACM symposium on applied computing (SAC), pp 396–401
Verborgh R, Vander Sande M, Hartig O, Van Herwegen J, De Vocht L, De Meester B, Haesendonck G, Colpaert P (2016) Triple pattern fragments: a low-cost knowledge graph interface for the Web. J Web Semant 37–38: 184–206
Volkel M, Winkler W, Sure Y, Kruk S, Synak M (2005) Semversion: a versioning system for RDF and ontologies. In: Proceeding of ESWC
Zaniolo SGJGC (2016) RDF-TX: a fast, user-friendly system for querying the history of RDF knowledge bases. In: Proceeding of EDBT
Zeginis D, Tzitzikas Y, Christophides V (2011) On computing deltas of RDF/S knowledge bases. ACM Trans Web (TWEB) 5(3):14
Zimmermann A, Lopes N, Polleres A, Straccia U (2012) A general framework for representing, reasoning and querying with annotated semantic Web data. JWS 12:72–95

RDF Stream Processing

▶ Semantic Stream Processing

RDF Syntaxes

▶ RDF Serialization and Archival

Real-Time Big Spatial Data Processing

▶ Streaming Big Spatial Data

Real-Time Streaming Benchmarks

▸ Analytics Benchmarks

Reasoning at Scale

Jacopo Urbani
Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

Synonyms

Inference; Knowledge base; Rule-Based processing

Definitions

Reasoning is the process of deriving new conclusions from knowledge bases using a series of logical steps. Reasoning at scale refers to the ability of applying this process to very large knowledge bases, such as modern knowledge graphs that are available on the Web.

Overview

The Web contains a very large amount of semi-structured datasets that cover encyclopedic knowledge, social or co-authorship networks, experimental results, etc. This data is encoded using RDF (Brickley et al. 2014), and it is interlinked to each other following the principles of linked open data, thus forming a large network of datasets called the Web of Data (WoD) (Bizer et al. 2009).

Automated reasoning can derive a wealth of nontrivial knowledge from these datasets, which can be used, for instance, to augment the WoD with new knowledge or to detect inconsistencies. Unfortunately, the large size of the WoD makes reasoning a challenging task. In fact, the datasets might be too large to be stored in a single machine, requiring thus some form of distributed computing. Moreover, parallelizing the computation is not trivial due to factors like the input skewness or special corner cases that require sequential processing.

What constitutes the state of the art for reasoning on a large scale? To answer this question, this chapter offers a broad overview of the most recent efforts to execute rule-based reasoning on large inputs. More in particular, it describes the most important optimizations that can be applied to improve the efficiency of reasoning. Some of these optimizations work only with specific rules, while others are more generic. Even though none of them work with all possible inputs, in practice they turned out to be very effective as they enabled reasoning on large knowledge bases, with up to 100 billion triples in the largest experiments.

What Is Scale?

Generally speaking, the term scalability refers to the ability of a system to handle larger instances of a given problem. There can be several reasons that hinder the scalability of a system. First, the problem might have an unfavorable computational complexity which precludes termination within a reasonable time. Second, the algorithms might be poorly implemented. In this case, the system cannot scale well despite the problem is tractable. Third, the hardware might not have enough resources to carry on the computation. Scalability is thus a property that can be judged from three different angles: the theoretical complexity, the implementation, and the hardware requirements.

Before discussing the state of the art and describe how it deals with these challenges, it is important to define more precisely what reasoning is supposed to compute. Let KB be a generic RDF dataset, that is, a set of RDF statements. This dataset can be represented as a labeled directed graph where each triple $\langle s, p, o \rangle$ maps to an edge that connects s to o and is labeled with p. These graphs are typically called *knowledge graphs*. Let

$KG = (V, E)$ be such a knowledge graph where V is the set of entities and E the labeled edges that connect the entities. Given in input a knowledge graph KG, the goal of reasoning is to derive new knowledge that can be inferred from KG. This knowledge takes the form of new triples which can be logically deduced from the KG. For now, it is assumed that V is complete, i.e., KG already contains all the entities of interest. With this assumption in mind, then the triples derived from reasoning can be represented by new edges in KG.

The derivation of new triples is determined by a set of rules, which must be provided as input. Rules are expressions of the form

$$B_1, \ldots, B_n \rightarrow H \tag{1}$$

where $B_1, \ldots, B_n$ are called *atoms*. An *atom* is an expression $p(\mathbf{x})$ where p is a *predicate* and $\mathbf{x} = x_1, \ldots, x_m$ is a tuple of terms that can be either variables or constants. A *fact* is an atom without any variable. In our context, facts are used to represent the KG. They can be unary (e.g., to express the *isA* relation – $Person(Mark)$), binary (e.g., $livesIn(Mark, Amsterdam)$), or ternary (e.g., $T(Mark, livesIn, Amsterdam)$). The set of atoms $B_1, \ldots, B_n$ is called the rule's *body*, while H is the rule's *head*.

Notice that rules can be more complex than in (1). For instance, some body atoms might be negated, or the head might contain a conjunction of multiple atoms. All scalable approaches which will be discussed in this chapter assume that rules only contain positive atoms and only one atom occurs in the head of the rule. Moreover, they assume that every variable in the head must also appear in the body (safeness condition). The reason behind these constraints is that they simplify the computation. For instance, negation can introduce non-determinism, while dropping safeness might lead to nontermination.

The computation of the rules can be formalized as follows. Let I be a generic database of facts (i.e., the input KG); σ be a *substitution*, i.e., a partial mapping from variables to other variables or constants; and $r \in P$ be a rule of the form (1) in the program P. Then, $r(I) = \{H\sigma \mid B_1\sigma, \ldots, B_n\sigma \in I\}$ is the set of derivations that can be derived from I using r and $P(I) = \bigcup_{r \in P} r(I)$ is its extension to all rules in the program. The exhaustive application of all rules can be defined recursively by setting $P^0(I) = I$ and $P^{i+1}(I) = P^i(I) \cup P(P^i(I))$. Since the rules are safe and the set of constants is finite, there will be a j s.t. $P^{j+1}(I) = P^j(I)$. In this case, $P^j(I)$ is called the *closure* or *materialization* of I with P.

Materialization with Fixed Rules

Ontological languages are used to serialize semantic relations in RDF knowledge bases in a machine-readable format. For instance, they allow the user to define various semantic relations like subsumption between classes (e.g., *Student* is a subclass of *Person*) or specify that a relation is transitive (e.g., *ancestorOf* or *partOf*).

Ontological statements like the previous two examples can be translated into rules by either considering the standard constructs of the language or by also including the ontology at hand. The following example is useful to understand this difference.

Example 1 Let us assume that the KG contains the following ontological statements: $\langle$`:Bob`, `isA`, `:Actor`$\rangle$ and $\langle$`:Actor`, `soc`, `:Man`$\rangle$ where `soc` is an abbreviation for the standard RDF schema IRI of class subsumption.

A rule of the first type could be

$$T(A, \texttt{isA}, B), T(B, \texttt{soc}, C) \rightarrow T(A, \texttt{isA}, C) \tag{2}$$

while a rule of the second type could be

$$isA(A, \texttt{:Actor}) \rightarrow isA(A, \texttt{:Man}) \tag{3}$$

In the first case, rule (2) simply translates the inference that it is possible to obtain considering the isA and soc relations. The rule is domain-independent and can be applied to any KG. In contrast, rule (3) is simpler because it does not require any data join but it has the disadvantage that it can be applied only to the input dataset.

In this chapter, rules of the first type are called *standard rules* since they are derived by standard ontological languages like RDF schema (Brickley et al. 2014) or OWL (Motik et al. 2009). Standard rules are important because they are universal in the sense that they do not depend on a particular input. Therefore, it is possible to introduce tailor-made optimizations to speed up their execution without any loss of generality. However, there are cases when nonstandard rules are preferable since they might be easier to execute. The remaining of this section will describe five optimizations which are crucial to enhance the scalability of reasoning using standard rules. The next section will address scalability with nonstandard rules.

Split instance/schema triples. The first, and perhaps most effective, optimization on current knowledge bases consists of splitting the input statements between the ones that describe instances and the ones that describe the schema. The last type of statements is typically ontological statements that use constructs from the language (e.g., OWL). One key property of current large KGs is that they contain many more instance statements than schema ones, and this is important because there are many standard rules which have two body atoms, one which matches instance statements while the other matches schema ones (Urbani et al. 2012). Rule (2) is such an example: Here, typically there will be many more triples of the form $T(A, \mathtt{isA}, B)$ than of the form $T(B, \mathtt{soc}, C)$.

A strategy to parallelize the computation of such rules is to simply range-partition the instance triples and assign each range to a different processor. If the processors operate in separated memory spaces (e.g., different machines), then schema triples can be replicated on each space.

After the partitioning is done, the rule can be executed in parallel without any intermediate node communication. An example of such computation is graphically depicted in Fig. 1a.

Reducing duplicates. There are cases where different rules might produce the same derivation. For instance, the two standard rules

$$T(A, P, B), T(P, \mathtt{domain}, C) \rightarrow T(A, \mathtt{isA}, C) \quad (4)$$

$$T(B, P, A), T(P, \mathtt{range}, C) \rightarrow T(A, \mathtt{isA}, C) \quad (5)$$

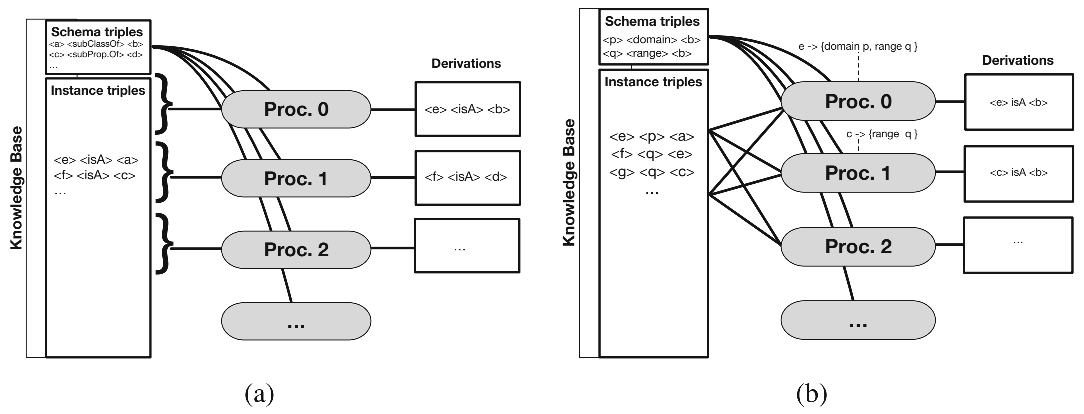
can derive the same information for an entity which is used both as a subject and as an object in different triples. For instance, $\langle Bob, worksIn, Amsterdam\rangle$ and $\langle Alice, daughterOf, Bob\rangle$ can both lead to the derivation $\langle Bob, isA, Person\rangle$ if the domain and range of the two predicates are $Person$.

A strategy to remove these types of duplicates is to group instance triples by the list of terms which are used in the head. In the previous case, triples can be grouped either by subject of by object, depending on the rule. Then, rules can be executed in parallel on each group. In this way, it is impossible that two different groups will produce the same derivation because the grouping criterion ensures that derivations must differ by at least one term (i.e., the grouping key).

During the rule computation, duplicates can still be produced within the same group. However, in this case they can be removed in parallel without any synchronization between the various processors. Figure 1b shows an example of such computation for rules (4) and (5).

***sameAs* table.** One category of rules which is widely used in modern knowledge bases encodes reasoning over the equality of concepts, which is a relation that is stated with the predicate *owl:sameAs*. Equality is transitive (if a is the same as b and b is the same as c, then a is the same as c) and symmetric (if a is the same as b, then b is the same as a); thus rules in this category produce a large number of materializations.

To improve the performance, reasoners typically avoid materializing all conclusions but instead build a dedicated table where groups of



Reasoning at Scale, Fig. 1 (**a**) Parallel execution of a rule that requires a join between schema and instance triples. (**b**) Execution of a group of rules: in this example, processor 0 receives all information regarding the entity "e," which allows it to apply both rules (4) and (5) and remove duplicates locally

equal terms receive a unique representative ID, and all occurrences of these terms in the KB are replaced by the corresponding ID. During the materialization, the reasoner might derive more equality relations. If this happens, then the table needs to be updated, and more occurrences must be replaced with the corresponding ID. Notice that also rules need to be rewritten if they contain constants in the table.

After the materialization is computed, the augmented $P^j(I)$ will contain a compressed version of the full materialization since further derivations can be obtained by simply replacing the occurrences of representative IDs with each of the members of their equality group. However, in practice this operation is often omitted as it can be trivially computed on-the-fly whenever is needed.

Sorting rule execution. For some standard rulesets like the rules from RDF schema, it is possible to define a rule application order to reduce to the minimum the chance that the output of one rule can be used as input for another one. Unfortunately, this optimization guarantees a complete output only on some specific cases, and it is not possible to define an execution order which avoids repeated executions with more complex rulesets like, for instance, the OWL2 RL/RDF ruleset (Motik et al. 2009).

Memoization. Another technique that can be applied to improve the performance is *memoization* (Urbani et al. 2014). Memoization is a special type of caching which consists of storing the output of expensive functions to reduce the cost of repeated executions. For the problem of reasoning, memoization can be used to precompute all answers of some particular atoms. This enables a faster computation of data joins.

An example is useful to clarify this optimization. Let us consider, once again, rule (2). This rule contains two body atoms: One atom matches instance triples while the other matches schema triples. During the materialization, the number of facts that match the second atom will change if other rules derive new triples with $\texttt{soc}$ as predicate. With memoization, before the materialization starts a query-driven materialization procedure like QSQR (Abiteboul et al. 1995) or Magic set (Bancilhon et al. 1985), two well-known query-driven algorithms are invoked to compute all answers for the query $T(B, \texttt{soc}, C)$. Once this procedure is terminated, we can safely assume that the collection of facts that match this atom is immutable; thus the engine can index this collection more efficiently to facilitate the execution of the rule. In some cases, memoization does not lead to any reduction of the materialization runtime, while in other cases, the advantage is significant.

Materialization with Generic Rules

Nonstandard rules are typically easier to execute since they require less joins and predicates have a smaller arity (i.e., they are unary or binary only). On these rules, however, the previous optimizations might not be applicable. Still, the execution can be improved in two ways: Either by applying more general parallel algorithms or by considering multiple facts at the same time. Both types of improvements are described below.

Parallelizing rule execution. Three different types of parallelism can be applied to the rule execution: *Intra-rule parallelism*, *inter-rule parallelism*, and *instance-based parallelism*.

With intra-rule parallelism, the goal is to distribute the execution of a single rule among different processors. For example, let us consider rule (1). In this case, facts that match B_1 could be partitioned into n different partitions depending on the value of the terms that should be joined with B_2. Similarly, facts that match B_2 could be partitioned in an equivalent number of partitions in so that "B_1" and "B_2" facts with the same join terms will be in the same partition. In this way, each partition can be processed simultaneously by concurrent processors.

With inter-rule parallelism, the idea is to let concurrent processors execute different rules at the same time. For instance, one processor could execute rule (4) while another one execute rule (5). Notice that neither of these two types of parallelism is perfect: With intra-rule parallelism, the computation could be unbalanced if some partitions are much bigger than others. With inter-rule parallelism instead, the maximum number of concurrent processors is bound by the number of rules.

Intra- and inter-based parallelism are well known in literature and are also used in other scenarios. The third type of parallelism is a more recent variant which was first introduced in the RDFox system (Nenov et al. 2015). The idea is to let a number of concurrent processors to continuously pull not-yet-considered facts from a queue and verify whether they instantiate the body of a rule. If this occurs, then the system searches for other atoms in the database to compute a full rule instantiation. If this process succeeds, then the processor produces a new derivation and puts it back in the queue and database so that it can be further considered, possibly by other processors.

This type of parallelism is significantly different than the other two because here the processors do not receive a predefined amount of work but are free to "steal" computation from each other whenever they become idle. A limitation of this technique is that it requires a number of data structures that allow a fast concurrent access. While hash tables can provide this functionality, they have the disadvantage that they are not cache-friendly, that is, they do not use efficiently the CPU cache.

Set-based rule execution. Another technique for improving the performance consists of generating meta-facts which represents multiple sets of facts. This technique can be intuitively explained with a simple example. Let us consider the rule:

$$P(X, Y) \to Q(Y, X) \tag{6}$$

and assume that the input database does not contain any q-facts. In this case, it is clear that each fact that matches the body will generate a new q-fact. Thus, the engine can simply create one single fact $q(y^*, x^*)$ where x^* and y^* are special terms which point to set of terms, namely, all first and second terms that appear in p-facts. For instance, if the database equals to $\{p(a, b), p(c, d)\}$, then $y^* \to \langle a, c\rangle$ and $x^* \to \langle b, d\rangle$. Notice that the engine does not need to explicitly materialize the lists $\langle a, c\rangle$ and $\langle b, d\rangle$ but can simply store instructions to compute this list on-the-fly in case it is needed.

This technique was first introduced in the VLog system (Urbani et al. 2016), and empirical results show excellent performance against the state of the art, especially because this technique becomes more effective with larger databases as potentially larger sets of facts can be compressed in a single meta-fact.

References

Abiteboul S, Hull R, Vianu V (1995) Foundations of databases, vol 8. Addison-Wesley, Reading
Bancilhon F, Maier D, Sagiv Y, Ullman JD (1985) Magic sets and other strange ways to implement logic programs. In: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on principles of database systems. ACM, pp 1–15
Bizer C, Heath T, Berners-Lee T (2009) Linked data-the story so far. Int J Semant Web Info Syst 5(3):1–22
Brickley D, Guha RV, McBride B (2014) RDF schema 1.1. W3C Recomm 25:2004–2014
Motik B, Grau BC, Horrocks I, Wu Z, Fokoue A, Lutz C, et al (2009) Owl 2 web ontology language profiles. W3C Recomm 27:61
Nenov Y, Piro R, Motik B, Horrocks I, Wu Z, Banerjee J (2015) RDFox: a highly-scalable RDF store. In: International semantic web conference. Springer, pp 3–20
Urbani J, Kotoulas S, Maassen J, Van Harmelen F, Bal H (2012) WebPIE: a web-scale parallel inference engine using mapreduce. Web Semant Sci Serv Agents World Wide Web 10:59–75
Urbani J, Piro R, van Harmelen F, Bal H (2014) Hybrid reasoning on OWL RL. Semantic Web 5(6):423–447. https://doi.org/10.3233/SW-130120
Urbani J, Jacobs C, Krötzsch M (2016) Column-oriented datalog materialization for large knowledge graphs. In: Proceedings of AAAI, pp 258–264

Recommender Systems Over Data Streams

András A. Benczúr[1], Levente Kocsis[1], and Róbert Pálovics[2]
[1]Institute for Computer Science and Control, Hungarian Academy of Sciences (MTA SZTAKI), Budapest, Hungary
[2]Department of Computer Science, Stanford University, Stanford, CA, USA

Synonyms

Incremental learning; Online machine learning on streams; Stream learning

Definitions

- Data stream algorithms process a continuous stream of data with only a limited possibility to store past records.
- Online machine learning covers methods that update their models after observing a new event and can immediately serve predictions based on the updated model.

Overview

In this chapter, we investigate online learning based recommender algorithms that can efficiently handle nonstationary datasets. We show that online learning for recommendation is rather usual than the exceptional task: For example, if no user history is available, we have to build a user model on the fly, based on the interactions in the live user session.

To the best of our knowledge, this is the first survey with a comprehensive overview of the ideas for recommendation over streaming data and their implementation in various distributed data stream processing systems.

The chapter is based on the notions of online learning, as introduced in the chapter "Overview of Online Machine Learning in Big Data Streams" of this Encyclopedia.

Introduction

Recommender systems (Ricci et al. 2011) serve to predict user preferences regarding items such as music tracks (Spotify), movies (Netflix), products, books (Amazon), blogs, or microblogs (Twitter), as well as content on friends' and personal news feeds (Facebook).

Recommenders give a clear, industry-relevant example of the requirements for online machine learning introduced in the Chapter "▶ Overview of Online Machine Learning in Big Data Streams" of this Handbook. In a typical implementation, users interact with the system by requesting recommendations and then providing feedback by clicking on some of the displayed items. In this way, the users produce a continuous stream of events that can be used for model update and immediate evaluation, for example, by click-through rate. Note that in Žliobaite et al. (2012), it is observed that adaptive learning models are still rarely deployed in industry. Recommender systems are the main exception: A special class, the session-based recommendation task appears frequently in practice when no past user history is available and user models are built on the fly, using recent interactions in the session.

Recommender systems can be categorized by the type of information they infer about users and items. Collaborative filtering (Linden et al. 2003; Sarwar et al. 2001) builds models of past user-item interactions such as clicks, views, purchases, or ratings, while content-based filtering Lops et al. (2011) recommends items that are similar in content, for example, share phrases in their text description. Context-aware recommenders (Adomavicius and Tuzhilin 2011) use additional information on the user and the interaction, for example, user location and weather conditions. Recent events in a user session (Koenigstein and Koren 2013) serve as a special context.

A milestone in the research of recommendation algorithms, the Netflix Prize competition (Bennett and Lanning 2007), had high impact on research directions. The target of the contest was based on the one- to five-star ratings given by users, with one part of the data used for model training and the other for evaluation. As an impact of the competition, tasks now termed batch rating prediction were dominating research results.

Recommendation models rely on the feedback provided by the user, which can be **explicit**, such as one- to five-star movie ratings on Netflix (Adhikari et al. 2012). However, most recommendation tasks are **implicit**, as the user provides no like or dislike information. Implicit feedback can be available in the form of time elapsed viewing an item or listening to a song, or in many cases, solely as a click or some other form of user interaction. In Pilászy et al. (2015), the authors claim that 99% of recommendation industry tasks are implicit.

As a main difference between recommendation and classification, classifiers usually work independently of the event whose outcome they predict. Recommender systems, on the other hand, may directly influence observations: They present a ranked top list of items (Deshpande and Karypis 2004), and the user can only provide feedback for the items on the list. Moreover, real systems process data streams where users request one or a few items at a time and get exposed to new information that may change their needs and taste when they return to the service next time. Furthermore, an online trained model may change and return completely different lists for the same user even for interactions very close in time.

By the above considerations, real recommender applications fall in the category of top item recommendation by online learning for implicit user feedback, a task that has received less attention in research so far. In this section, we show the main differences in evaluating such systems compared to both classifiers and batch systems, as well as describe the main data stream recommender algorithms.

Online recommenders seem more restricted than those that can iterate over the data set several times, and one could expect inferior quality from the online methods. By contrast, in Pálovics et al. (2014) and Frigó et al. (2017), surprisingly strong performance of online methods is measured.

As an early time-aware recommender system example, the item-based nearest neighbor (Sarwar et al. 2001) can be extended with time-decay (Ding and Li 2005). Most of the early models, however, are time-consuming to compute and difficult to update from a data stream and hence need periodical batch training. Probably the first result in this area, the idea of processing transactions in chronological order to incrementally train a recommendation model first appeared in Takács et al. (2009, Section 3.5). Streaming gradient descent matrix factorization methods were also proposed in Isaacman et al. (2011) and Ali and Johnson (2011), who use Netflix and MovieLens data and evaluate by root mean square error (RMSE).

The difficulty of evaluating streaming recommenders was first mentioned in Lathia et al. (2009), although the authors evaluated models by offline training and testing split.

Ideas for online evaluation metrics appeared first in Pálovics and Benczúr (2013), Vinagre et al. (2014), and Pálovics et al. (2014). In Vinagre et al. (2014), incremental algorithms are evaluated using recall. In Pálovics et al. (2014), recall is shown to have undesirable properties, and other metrics for evaluating online learning recommenders are proposed.

Finally, we note that batch distributed recommender systems were surveyed in Karydi and Margaritis (2016).

Prequential (Online) Evaluation for Recommenders

To train and evaluate a time-sensitive or online learning recommender, we can use the prequential or online evaluation framework that is described in detail for classifier evaluation in the chapter "▶ Online Machine Learning Algorithms over Data Streams" of this Handbook. As seen in Fig. 1, online evaluation for a recommender system includes the following steps:

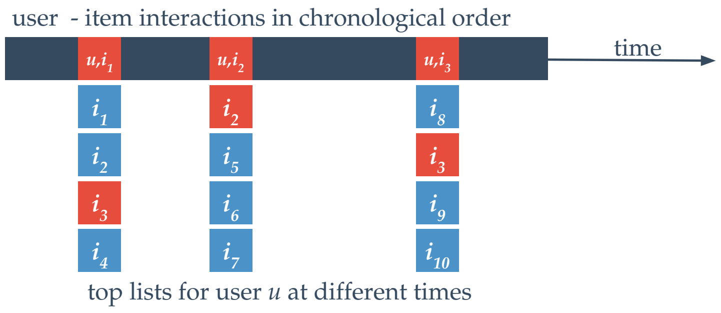
1. We query the recommender for a top-k recommendation for the active user.
2. We evaluate the list in question against the single relevant item that the user interacted with.
3. We allow the recommender to train on the revealed user-item interaction.

Since we can potentially retrain the model after every new event, the recommendation for the same user may be very different even at close points in time, as seen in Fig. 1. The standard recommender evaluation settings used in research cannot be applied, since there is always only a single relevant item in the ground truth.

In one of the possible recommender evaluation settings, the **rating prediction** problem, which is popular in research, we consider a user u and an item i. The actual user preference in connection with the item is expressed as a value r_{ui}, for which the system returns a prediction $\hat{r}_{ui}$. This explicit rating can be a scale such as one to five stars for a Netflix movie, while implicit rating can be the duration of viewing a Web page in seconds. Implicit rating is binary when the only information is whether the user interacted with the item (clicked, viewed, purchased) or not. Depending on whether r_{ui} is binary or scale, the same prequential metrics, such as error rate or mean squared error (MSE), can be applied as for classification or regression. For example, in the Netflix Prize competition, the target was the square root of MSE between the predicted and actual ratings.

Another possible way to evaluate recommenders is **ranking prediction**, where performance metrics depend on the list of displayed items. We note that given rating prediction values $\hat{r}_{ui}$ for all i, in theory, ranking prediction can be solved by sorting the relevance score of all items. For certain models, heuristics to speed up the selection of the highest values of $\hat{r}_{ui}$ by candidate preselection exist (Teflioudi et al. 2015).

To evaluate ranking prediction, we have to take into consideration two issues that do not exist for classifier evaluation. In the case of prequential evaluation, as shown in Fig. 1, the list for user u may change potentially after every


user - item interactions in chronological order
time
u,i1
u,i2
u,i3
i1
i2
i3
i4
i2
i5
i6
i7
i8
i3
i9
i10
top lists for user u at different times

Recommender Systems Over Data Streams, Fig. 1 Prequential evaluation of the online ranking prediction problem interaction with u. As soon as u provides feedback for certain item i, we can change model parameters and the set of displayed items may change completely. Most of the batch ranking quality measures focus on the set of items consumed by the same user, under the assumption that the user is exposed to the same list of items throughout the evaluation. As this assumption does not hold, we need measures for individual user-item interactions.

Another issue regarding ranking prediction evaluation lies in a potential user-system interaction that affects quality scores. Typically, the set of items is very large, and users are only exposed to a relatively small subset, which is usually provided by the system. The form of user feedback is usually a click on one or more of these items, which can be evaluated by computing the click-through rate. Since users cannot give feedback on items outside the list, the fair comparison of two algorithms that present different sets for the user can only be possible by relying on live user interaction. This fact is known by practitioners, who use **A/B testing** to compare the performance of different systems. In A/B testing, the live set of users is divided into groups that are exposed to the results of the different systems.

Most traditional ranking prediction metrics, to a certain level, rely on the assumption that the same user is exposed to the same list of items, and hence the interactions of the same user can be considered to be the unit for evaluation. For online evaluation, as noted in Pálovics et al. (2014), the unit of evaluation will be a single interaction, which usually contains a single relevant item. Based on this modification, most batch metrics apply in online learning evaluation as well. Note that the metrics below apply not just in A/B testing but also in experiments with frozen data, where user feedback is not necessarily available for the items returned by a given algorithm. For example, if the item consumed by the user in the frozen data is not returned by the algorithm, the observed relevance will be 0, which may not be the case if the same algorithm is applied in an A/B test. Note that attempts to evaluate research results by A/B testing have been made in the information retrieval community (Balog et al. 2014); however, designing and implementing such experiments is cumbersome.

Next, we list several metrics for the quality of the ordered top-K list of items $L = \{i_1, i_2, \ldots, i_K\}$ against the items E consumed by the user. We will also explain how online evaluation metrics differ from their batch counterparts. For the discussion, we mostly follow Pálovics et al. (2014).

Click-through rate is commonly used in the practice of recommender evaluation. It is defined as the ratio of clicks received for L:

$$\text{Clickthrough@K} = \begin{cases} 1 & \text{if } E \cap L \neq \emptyset; \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

For precision and recall, similar to click-through, the actual order within L is unimportant:

$$\begin{aligned} \text{Precision@K} &= \tfrac{|E \cap L|}{K}, \\ \text{Recall@K} &= \tfrac{|E \cap L|}{|E|}. \end{aligned} \quad (2)$$

For batch evaluation, E is the entire set of items with positive feedback from a given user who is exposed to the same L for each interaction. The overall batch system performance can be evaluated by averaging precision and recall over the set of users. For online evaluation, typically $|E| = 1$, where Precision@K is 0 or $1/K$ and Recall@K is 0 or 1 depending on whether the actual item in E is listed in L or not. Precision and recall are hence identical to click-through, up to a constant. As a consequence, the properties of online precision and recall are very different from their batch counterparts. The main reason for the difference lies in the averaging procedure of prequential evaluation: We cannot merge the events of the same user; instead, we average over the set of individual interactions.

Measures that consider the position of the relevant item i in L can give more refined performance indication. The first example is reciprocal rank:

$$RR@K = \begin{cases} 0 & \text{if rank}(i) > K; \\ \dfrac{1}{\text{rank}(i)} & \text{otherwise.} \end{cases} \quad (3)$$

Discounted cumulative gain (DCG) is defined similarly, as

$$\mathrm{DCG@K} = \sum_{k=1}^{K} \frac{\mathrm{rel}(i_k)}{\log_2(1+k)} \qquad (4)$$

where $\mathrm{rel}(i_k)$ indicates the relevance of the i-th item in the list. For the implicit task, relevance is 1 if the user interacted with the item in the evaluation set, 0 otherwise. For batch evaluation, we can consider all interactions of the same user as one unit. If we define iDCG@K, the ideal maximum possible value of DCG@K for the given user, we can obtain nDCG@K, the normalized version of DCG@K, as

$$\mathrm{nDCG@K} = \frac{\mathrm{DCG@K}}{\mathrm{iDCG@K}}. \qquad (5)$$

Note that for online learning, there is only one relevant item, hence iDCG $= 1$. For emphasis, we usually use the name nDCG for batch and DCG for online evaluation.

Session-Based Recommendation

Previous items in user sessions constitute a very important context (Hidasi and Tikk 2016). In e-commerce, the same user may return next time with a completely different intent and may want to see a product category completely different from the previous session. Algorithms that rely on recent interactions of the same user are called **session-based item-to-item** recommenders. The user session is special context, and it is the only information available for an item-to-item recommender. In fact, several practitioners (Koenigstein and Koren 2013; Pilászy et al. 2015) argue that most of the recommendation tasks they face are without sufficient past user history. For example, users are often reluctant to create logins and prefer to browse anonymously. Moreover, they purchase certain types of goods (e.g., expensive electronics) so rarely that their previous purchases will be insufficient to create a meaningful user profile. Whenever a long history of previous activities or purchases by the user is not available, recommenders may propose items that are similar to the most recent ones viewed in the actual user session.

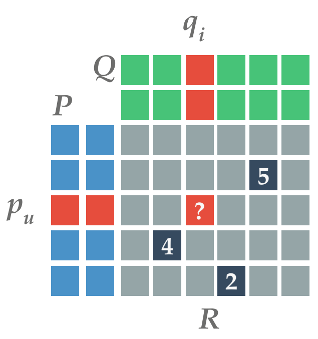
Session-based recommendation can be served by very simple algorithms, most of which are inherently online. A comparison of the most important such online algorithms in terms of performance is available in Frigó et al. (2017). Data stream processing algorithms can retain items from the most recently started sessions as long as they fit in their memory. Recommendation is based on the recent items viewed by the user in the actual shopping session. For example, we can record how often users visited item i after visiting another item j. Since fast update to transition frequencies is usually possible, the method is online.

In an even simpler algorithm that is not strictly session-based, we recommend the most popular recent items. This method can be considered batch or online depending on the granularity of the item frequency measurement update. Both algorithms can be personalized if we consider the frequency of past events involving the user. If items are arranged hierarchically (e.g., music tracks by artist and genre), personal popularity and personal session data can involve the frequency of the artists or genres for recommending tracks. More session-based algorithms are described in Koenigstein and Koren (2013).

Online Matrix Factorization

Most nontrivial online recommender algorithms are based on matrix factorization (Koren et al. 2009), a popular class of collaborative filtering methods. Given the user-item utility matrix $R = [r_{ui}]$ shown in Fig. 2, we model R by decomposing it into the two dense matrices P and Q. For a given user u, the corresponding row in P is user vector p_u. Similarly, for item i, the corresponding column of Q is item vector q_i. The predicted relevance of item i for user u is then

$$\hat{r}_{ui} = p_u q_i^T. \qquad (6)$$



Recommender Systems Over Data Streams, Fig. 2 Utility matrix R and the matrix factorization model built from matrices P and Q

Note that we can extend the above model by scalar terms that describe the biased behavior of the users and the items (Koren et al. 2009).

One possibility to train model parameter matrices P and Q is by gradient descent (Koren et al. 2009; Funk 2006), which can be applied to online learning as well (Pálovics et al. 2014). For a set of interactions E, we optimize Eq. (6) for MSE as target function:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{r}_{ui} - r_{ui})^2, \tag{7}$$

where r_{ui} is the actual and $\hat{r}_{ui}$ is the predicted rating for user u and item i and N is the current size of the data stream.

In one step of gradient descent, we fit P and Q in Eq. (6) to one of the ratings in E. Unlike in batch training, where we can use the ratings several times in any order, in online learning, we have the most recent single item in E. In other words, in online gradient descent, we fit the model to the events one by one as they arrive in the data stream.

For a given (explicit or implicit) rating r_{ui}, the steps of gradient descent are as follows. First, we compute the gradient of objective function F with respect to the model parameters:

$$\frac{\partial F}{\partial p_u} = -2(r_{ui} - \hat{r}_{ui})q_i, \quad \frac{\partial F}{\partial q_i} = -2(r_{ui} - \hat{r}_{ui})p_u. \tag{8}$$

Next, we update the model parameters in opposite direction of the gradient, proportionally to learning rate η, as

$$p_u \leftarrow \eta(r_{ui} - \hat{r}_{ui})q_i,$$
$$q_i \leftarrow \eta(r_{ui} - \hat{r}_{ui})p_u.$$

Overfitting is usually avoided by adding a regularization term in the objective function (Koren et al. 2009).

In the case of implicit feedback, the known part of the utility matrix only contains elements with positive feedback. To fit a model, one requires negative feedback for training as well. Usually, such elements are selected by sampling from those that the user has not interacted with before Rendle and Freudenthaler (2014). We can also introduce confidence values for ratings and consider lower confidence for the artificial negative events (Hu et al. 2008).

Gradient descent can also be used in a mix of batch and online learning, for example, training batch models from scratch periodically and continuing the training with online learning. We can also treat users and items differently, for example, updating user vectors more dynamically than item vectors, as first suggested by Takács et al. (2009).

Another use of online gradient descent is to combine different recommendation models (Pálovics et al. 2014). We can express the final prediction as the linear combination of the models in the ensemble whose parameters are the linear coefficients and the individual model parameters. In Pálovics et al. (2014), two online gradient descent methods are described with regard to whether the derivative of the individual models is available, where all parameters can be trained through the derivative of the final model or otherwise by learning the coefficients and the individual models separately.

Variants of Matrix Factorization

Several variants of matrix factorization that can be trained by gradient descent both for batch and online learning tasks have been proposed. Bayesian Personalized Ranking (Rendle et al. 2009) has top list quality as target instead of MSE. In asymmetric matrix factorization (Paterek 2007), we model the user by the sum of the item vectors the user rated in the past.

Recently, various factorization models have been developed that incorporate context information (Hidasi and Tikk 2016). Context data can be modeled by introducing data tensor D instead of the rating matrix R. In a simplest case, the data includes a single piece of additional context information (Rendle and Schmidt-Thieme 2010): for example, music tracks can have artist as context.

Alternating least squares (Koren et al. 2009; Pilászy et al. 2010) (ALS) is another optimization method for matrix factorization models, in which for a fixed Q, we compute the optimal P, then for a fixed P, the optimal Q, repeatedly until certain stopping criteria are met. Hidasi et al. Hidasi and Tikk (2012); Hidasi (2014); Hidasi and Tikk (2016) introduced several variants of ALS-based optimization schemes to incorporate context information. By incremental updating, ALS can also be used for online learning (He et al. 2016).

Conclusions

Recommendation differs from classification in that in recommendation, there are two types of objects, users, and items, and a prediction has to be made for their interaction. A practical recommender system displays a ranked list of a few items for which the user can give feedback. In an online learning system, the list shown to the same user at different times may change completely for two reasons. First, as in the prequential classifier training and evaluation setting described in detail for classifier evaluation in the chapter "▶ Online Machine Learning Algorithms over Data Streams" of this Handbook, the list of recommendations may change because the model changes. Second, the user feedback we use for evaluation depends on the actual state of the model, since the user may have no means to express interest in an item not displayed. Hence for online learning evaluation, metrics that involve the notion of a volatile list have to be used.

Online learning, as introduced in the chapter "▶ Overview of Online Machine Learning in Big Data Streams" of this Handbook, is very powerful for recommender systems due to their advantage of having much more emphasis on recent events. For example, if we update models immediately for newly emerged users and items, trends are immediately detected. The power of online learning for recommendation may also be the result of updating user models with emphasis on recent events, which may be part of the current user session. User session is a highly relevant context for recommendation, and most session-based methods are inherently online.

Cross-References

- ▶ Online Machine Learning Algorithms over Data Streams
- ▶ Overview of Online Machine Learning in Big Data Streams
- ▶ Reinforcement Learning, Unsupervised Methods, and Concept Drift in Stream Learning

Acknowledgements Support from the EU H2020 grant Streamline No 688191 and the "Big Data—Momentum" grant of the Hungarian Academy of Sciences.

References

Adhikari VK, Guo Y, Hao F, Varvello M, Hilt V, Steiner M, Zhang ZL (2012) Unreeling netflix: understanding and improving multi-cdn movie delivery. In: INFOCOM, 2012 Proceedings IEEE. IEEE, pp 1620–1628

Adomavicius G, Tuzhilin A (2011) Context-aware recommender systems. In: Ricci F, Rokach L, Shapira B, Kantor PB (eds) Recommender systems handbook. Springer, Boston, pp 217–253

Ali M, Johnson CC, Tang AK (2011) Parallel collaborative filtering for streaming data. University of Texas Austin, Technical Report

Balog K, Kelly L, Schuth A (2014) Head first: living labs for ad-hoc search evaluation. In: Proceedings of the 23rd ACM international conference on conference on information and knowledge management. ACM, pp 1815–1818

Bennett J, Lanning S (2007) The netflix prize. In: KDD Cup and workshop in conjunction with KDD 2007

Deshpande M, Karypis G (2004) Item-based top-n recommendation algorithms. ACM Trans Inf Syst (TOIS) 22(1):143–177

Ding Y, Li X (2005) Time weight collaborative filtering. In: Proceedings of the 14th ACM international conference on Information and knowledge management. ACM, pp 485–492

Frigó E, Pálovics R, Kelen D, Benczúr AA, Kocsis L (2017) Online ranking prediction in non-stationary environments. In: Proceedings of the 1st workshop on temporal reasoning in recommender systems, co-located with 11th international conference on recommender systems

Funk S (2006) Netflix update: try this at home. http://sifter.org/simon/journal/20061211.html

He X, Zhang H, Kan MY, Chua TS (2016) Fast matrix factorization for online recommendation with implicit feedback. In: Proceedings of the 39th international ACM SIGIR conference on research and development in information retrieval. ACM, pp 549–558

Hidasi B (2014) Factorization models for context-aware recommendations. Infocommun J VI(4):27–34

Hidasi B, Tikk D (2012) Fast ALS-based tensor factorization for context-aware recommendation from implicit feedback. In: Machine learning and knowledge discovery in databases. Springer, pp 67–82

Hidasi B, Tikk D (2016) General factorization framework for context-aware recommendations. Data Min Knowl Discov 30(2):342–371

Hu Y, Koren Y, Volinsky C (2008) Collaborative filtering for implicit feedback datasets. In: Eighth IEEE international conference on data mining, 2008. ICDM'08. IEEE, pp 263–272

Isaacman S, Ioannidis S, Chaintreau A, Martonosi M (2011) Distributed rating prediction in user generated content streams. In: Proceedings of the fifth ACM conference on recommender systems. ACM, pp 69–76

Karydi E, Margaritis K (2016) Parallel and distributed collaborative filtering: a survey. ACM Comput Surv (CSUR) 49(2):37

Koenigstein N, Koren Y (2013) Towards scalable and accurate item-oriented recommendations. In: Proceedings of the 7th ACM conference on recommender systems. ACM, pp 419–422

Koren Y, Bell R, Volinsky C (2009) Matrix factorization techniques for recommender systems. Computer 42(8):30–37

Lathia N, Hailes S, Capra L (2009) Temporal collaborative filtering with adaptive neighbourhoods. In: Proceedings of the 32nd international ACM SIGIR conference on research and development in information retrieval. ACM, pp 796–797

Linden G, Smith B, York J (2003) Amazon.com recommendations: item-to-item collaborative filtering. Internet Comput IEEE 7(1):76–80

Lops P, De Gemmis M, Semeraro G (2011) Content-based recommender systems: state of the art and trends. In: Ricci F, Rokach L, Shapira B, Kantor, PB (eds) Recommender systems handbook. Springer, Boston, pp 73–105

Pálovics R, Benczúr AA (2013) Temporal influence over the Last.fm social network. In: Proceedings of the 2013 IEEE/ACM international conference on advances in social networks analysis and mining. ACM, pp 486–493

Pálovics R, Benczúr AA, Kocsis L, Kiss T, Frigó E (2014) Exploiting temporal influence in online recommendation. In: Proceedings of the 8th ACM conference on recommender systems. ACM, pp 273–280

Paterek A (2007) Improving regularized singular value decomposition for collaborative filtering. In: Proceedings of KDD Cup workshop at SIGKDD'07, 13th ACM international conference on knowledge discovery and data mining, pp 39–42

Pilászy I, Serény A, Dózsa G, Hidasi B, Sári A, Gub J (2015) Neighbor methods vs matrix factorization – case studies of real-life recommendations. In: LSRS2015 at RECSYS

Pilászy I, Zibriczky D, Tikk D (2010) Fast ALS-based matrix factorization for explicit and implicit feedback datasets. In: Proceedings of the fourth ACM conference on recommender systems. ACM, pp 71–78

Rendle S, Freudenthaler C (2014) Improving pairwise learning for item recommendation from implicit feedback. In: Proceedings of the 7th ACM international conference on web search and data mining. ACM, pp 273–282

Rendle S, Freudenthaler C, Gantner Z, Schmidt-Thieme L (2009) Bpr: Bayesian personalized ranking from implicit feedback. In: Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence. AUAI Press, pp 452–461

Rendle S, Schmidt-Thieme L (2010) Pairwise interaction tensor factorization for personalized tag recommendation. In: Proceedings of the third ACM international conference on web search and data mining. ACM, pp 81–90

Ricci F, Rokach L, Shapira B (2011) Introduction to recommender systems handbook. In: Ricci F, Rokach L, Shapira B, Kantor PB (eds) Recommender systems handbook. Springer, Boston

Sarwar B, Karypis G, Konstan J, Reidl J (2001) Item-based collaborative filtering recommendation algorithms. In: Proceedings of the 10th international conference on World Wide Web (WWW'01). ACM Press, New York, pp 285–295. https://doi.org/10.1145/371920.372071. http://portal.acm.org/citation.cfm?id=372071

Takács G, Pilászy I, Németh B, Tikk D (2009) Scalable collaborative filtering approaches for large recommender systems. J Mach Learn Res 10:623–656

Teflioudi C, Gemulla R, Mykytiuk O (2015) Lemp: fast retrieval of large entries in a matrix product. In: Proceedings of the 2015 ACM SIGMOD international conference on management of data. ACM, pp 107–122

Vinagre J, Jorge AM, Gama J (2014) Evaluation of recommender systems in streaming environments. In: Workshop on recommender systems evaluation: dimensions and design (REDD 2014), held in conjunction with RecSys 2014, Silicon Valley, Oct 10, 2014

Žliobaite I, Bifet A, Gaber M, Gabrys B, Gama J, Minku L, Musial K (2012) Next challenges for adaptive learning systems. ACM SIGKDD Explor Newsl 14(1): 48–55

Record Linkage

Anja Gruenheid
Google Inc., Madison, WI, USA

Synonyms

Duplicate detection; Entity resolution

Definitions

Record linkage refers to the task of extracting record information from various input data sources and combining them in such a way that each output record corresponds a distinct real-world entity.

Overview

Record linkage is part of the broader area of data integration and more specifically data cleaning. It is most commonly used as a means to identify duplicates in a dataset or multiple datasets. The biggest challenge when executing record linkage algorithms is the trade-off between quality and performance. That is, record linkage is often run on high- volume datasets for which even sophisticated algorithms will not be able to provide high- quality results in a suitable timeframe. Thus, techniques such as blocking or incremental computation are applied to improve performance at the cost of decreased result quality. Additional challenges in record linkage include various types of input sources that are not necessarily structured. For example, human-generated data for record linkage has been extensively studied in recent years under the assumption that human-generated data on the similarity of records has better quality than machine-generated data.

Key Research Findings

Record linkage is a decades- old problem that has been studied extensively. It can be split into several key research areas. First, *similarity computation* describes the challenge of identifying whether two records are similar. Second, there exists a variety of *algorithms* that take these similarities as input and output (sets of) records where each record represents a real-world entity. Finally, *performance* challenges are have arisen in the context of record linkage due to an increase in data volume and velocity and the inherent quadratic scaling of exhaustive record linkage algorithms.

Similarity Computation

To identify whether two records refer to the same entity, it is crucial to measure their similarity. For example, Table 1 shows different business addresses that can be found in datasets obtained from (semi-)manually curated business listings. Here, different records r_i have different formats, some have erroneous values, and others may have missing data. This exemplifies that comparable similarity computation is not trivial. For example, is r_1 as similar to r_2 (different formatting) as it is to r_3 (wrong phone number)? The most common similarity computation techniques are based on the similarity of the characters in the record strings, Elmagarmid et al. (2007), when comparing two input records. Examples for pair-wise character-based similarity computation techniques are edit distances such as the Levenshtein distance, Levenshtein (1966), or metrics such as the Jaro similarity metric, Jaro (1978). Alternatively, some record linkage systems also use token-based similarity, first proposed by Monge et al. (1996), or phonetic similarity, Russell (1922). The chosen similarity metric often depends on the use case that the record linkage mechanism is applied in.

Given these similarity metrics, it is obvious that resolving the similarity between records r_i and r_j is not always certain. That is, independent of the actual metric, the pair-wise similarity of records r_i and r_j is typically represented as a probability $p \in [0, 1]$ where $p(r_i, r_j) = 1$ signi-

R

Record Linkage, Table 1 Example business addresses

ID	Name	Address	City	Phone
r_1	Peet's coffee	2124 Vine St Ste1	Berkeley	(510) 841-0564
r_2	Peet's coffee	2124 VINE ST	BERKELEY	5108410564
r_3	Peet's coffee	2124 Vine St	Berkeley	5102257700
r_4	Peet's	2501 Telegraph Ave	Berkeley	(510) 225-7700
r_5	Peet's	2501 Telegraph Ave		(510) 225-7700
r_6	Peet's coffee		Berkeley	

fies that both records are identical. Uncertainty in the similarity computation can thus be expressed explicitly, i.e., $p(r_i, r_j) \approx 0.5$.

Algorithms

Algorithms for record linkage take as input the pair-wise similarity values, evaluate them, and output a partitioning of records $R = \bigcup R_k$. In this partitioning, each record $r_i^k \in R_k$ points to the same real-world entity e_k. This problem has been approached from various angles. Amongst the most significant are techniques such as *probabilistic modeling*, *graph clustering*, and *(semi-)supervised learning* described in detail next. Note that this is not a complete list of record linkage algorithms many of which are explained and evaluated by Hassanzadeh et al. (2009) or discussed in Dong and Srivastava (2015).

Probabilistic Modeling. First formalized by Fellegi and Sunter (1969), the core idea of probabilistic modeling for record linkage is to assign a record pair $\gamma = (r_i, r_j)$ to either a set of matches M or non-matches U based on the computational similarity of γ. Specifically, assigning the pair to M signifies that $r_i = r_j$, while if the pair is in U, then $r_i \neq r_j$ holds. The record pair is assigned to M if the probability of assigning γ to M based on the agreement $m(\gamma)$ is higher than the probability of assigning γ to U based on disagreement $u(\gamma)$, i.e., $P(M|m(\gamma) > P(U|u(\gamma)))$. Using the Bayes rule, this can be rewritten as

$$\gamma \in \begin{cases} M, \text{ if } \frac{P(m(\gamma)|M)}{P(u(\gamma)|U)} > \frac{P(U)}{P(M)}, \\ U \text{ otherwise.} \end{cases} \quad (1)$$

Probabilistic modeling assumes, as the name indicates, that the decision whether γ is a match is non-trivial and has to be expressed as a likelihood. Thus, by design, there will be cases where there exists evidence for γ being in M and U at the same time. To resolve this problem, probabilistic modeling assumes that assigning γ to M even though there exists some evidence that it should be in U incurs an error reflected in a penalty. Finding the assignment of all possible record pairs thus becomes an *error minimization* problem. Furthermore, Fellegi and Sunter (1969) observed that the similarity functions that determine the agreements and disagreements of γ may contain errors. Thus, they loosened their categorization of γ to allow it to be classified as a *possible pair* if there is evidence for both a match and a non-match. Pairs in that group are confirmed by human workers after the automated record linkage has concluded.

Correlation Clustering. Correlation clustering is a popular graph clustering technique for record linkage, Bansal et al. (2004). At its core, it builds upon the idea of probabilistic modeling to determine whether a record pair is a match or non-match, but instead of assigning the pair to distinct classes, correlation clustering assigns them to distinct record clusters where each cluster represents a real-world entity. The assignment is done analogous to the idea of error minimization in probabilistic modeling, i.e., if the penalty of adding a record r_i to a cluster c_k is smaller than the penalty of adding it to any other cluster or keeping it as a singleton cluster, then the record is added to c_k. Note that correlation clustering can also be formulated as an *agreement maximization* problem which may be easier to implement for some use cases.

Formally, correlation clustering works as follows. Assume that F is the penalty for the a clustering C and $p(r_i, r_j)$ is the probability of r_i and r_j belonging to the same cluster $c_k \in C$. An optimal clustering C^* is the clustering that has the smallest value of F, i.e., F^*, given any possible clustering. Recall that its value is computed as the global penalty. In other words, if records r_i and r_j are in the same cluster c_k but $p(r_i, r_j) < 1$, then this clustering incurs a penalty of $1 - p(r_i, r_j)$. Similarly, if r_i and r_j are not in c_k, the clustering incurs a penalty of $p(r_i, r_j)$. F is thus computed as:

$$\begin{aligned} F(C) = \textstyle\sum_{c_k \in C \wedge r_i, r_j \in c_k} & 1 - p(r_i, r_j) \\ + \textstyle\sum_{c_k \in C \wedge r_i \in c_k \wedge r_j \notin c_k} & p(r_i, r_j) \end{aligned} \tag{2}$$

Note that there may exist multiple clusterings that all have the same value F^* and are thus considered optimal.

Calculating C^* exhaustively is computationally infeasible. Thus, Bansal et al. (2004) propose an approximation algorithm called *cautious* correlation clustering with an approximation parameter δ. It proceeds iteratively as follows on a set of records R:

1. Pick a node $r_i \in R$ at random.
2. Add the neighborhood of r_i, i.e., all records connected to r_i, to cluster c_k.
3. Remove all records in c_k that are not 3δ-good.
4. Add all records in the neighborhood of c_k that are 7δ-good.
5. Remove all records in c_k from R. Jump to 1.

The parameter δ is specific to cautious correlation clustering and is used to give a guarantee of how much the approximated solution diverges from the optimal solution. Specifically, the authors show that with the above algorithm, a $9(\frac{1}{\delta^2} + 1)$ approximation can be found in $O(R^2)$.

(Semi-)Supervised Learning. Learning techniques have shown potential for improvement of record linkage solutions especially for record linkage scenarios where the similarity of records cannot be captured with simpler similarity comparison techniques such as character-based similarities. Amongst others, there has been work on applying SVMs, Bilenko et al. (2003), and learning how to cluster records that refer to the same real-world entity in a supervised manner, Cohen and Richman (2002). More recently, research in the area of record linkage has seen an increase in work combining crowdsourcing, i.e., human responses, and automated entity resolution techniques. For example, Wang et al. (2012) and as follow-up Wang et al. (2014) discuss strategies to minimize the number of crowd requests while at the same time increasing the quality of the record linkage solution. Specifically, the idea in this line of work is to leverage positive responses of the crowd workers to reduce the search space based on transitivity. That is, if records r_i and r_j represent the same real-world entity, and r_j and r_l refer to different entities, then the crowd does not need to confirm the relationship of r_i and r_l. Other work in this area has also utilized similar techniques as applied for probabilistic modeling to maximize the quality of a crowdsourced record linkage solution, Verroios and Garcia-Molina (2015).

Performance Improvements

With an increase in stored data and processing capabilities, new challenges have arisen for executing traditional record linkage efficiently. For example, large datasets in the 1960s for which some of the original record linkage algorithms were developed, are multiple orders of magnitude smaller than current linkage datasets. Thus, several performance improvement techniques such as *blocking* or *incremental data processing* have been applied in the context of record linkage to make this process computationally feasible. A drawback of these algorithms and mechanisms is that they often trade off quality for performance.

Blocking. The purpose of blocking is to parallelize computationally expensive steps in the record linkage process. Basically, blocking partitions the record set R according to a pre-defined input function f that maps a record $r_i \in R$ to a block B_k, Jaro (1989). The goal of f is to map all records that point to the same real-world entity to the same B_k. Techniques such as similarity comparison which would have to be run in $O(R^2)$ before can then be run in $O(|B_k|^2)$. The applied mapping function is either based on the record characteristics such as the words, tokens, or k-grams it contains, domain knowledge of the record linkage context, user-specified constraints, Arasu et al. (2009), or techniques such as sorted neighborhood, Hernández and Stolfo (1998). The drawback of defining such functions is that there are no guarantees whether any blocking function will correctly map all records of the same real-world entity to the same block. Approaches to address this problem are, for example, to deploy multiple blocking functions or compute overlapping canopies instead of distinct blocks, McCallum et al. (2000).

Incremental Record Linkage. Datasets that are deduplicated are often static and only occasionally updated. The idea behind incremental record linkage is to leverage that observation and to only (re)compute those parts of the record linkage solution that have been directly or indirectly modified by updates in the input dataset. First discussed by Benjelloun et al. (2009) as agglomerative clustering, the basic idea of incremental record linkage was extended by Gruenheid et al. (2014) to allow any kind of data modification of the input dataset and to provide theoretical guarantees for graph record linkage techniques.

In practice, incremental record linkage does not modify the idea of the original batch linkage algorithm, for example, correlation clustering, but modifies these algorithms to fit an iterative linkage use case. Depending on the applied batch linkage algorithm, this may lead to a decrease in the quality of the linked results if the record linkage computation is global and cannot be localized, i.e., when the whole dataset is required as input to form an optimal solution.

Applications of Record Linkage

Entity resolution is a task important for many different data integration and more recently machine learning systems. Thus, there exist dedicated systems such as IBM InfoSphere that have been commercializing record linkage for many decades as part of so-called ETL (extract, transform, load) processes. The most common use case for commercial system is that legacy data in companies is stored in various places such as databases, file systems, etc. and needs to be combined efficiently and without losing any of the input data. Deduplication issues arise also if companies merge and the data of the acquired company has to be integrated into the existing internal database. Next to commercial systems, there also exist open source alternatives that can be used for entity resolution such as Konda et al. (2016).

Furthermore, record linkage is part of pipelines used for data cleaning. The goal is to find "dirty" data which is the same record inserted into the system multiple times with slight variations,; see Table 1 for an example. It is crucial to have good- quality linkage results for such pipelines as the consolidated entities are often shown in user interfaces. For example, information about restaurants, attractions, etc. can be obtained from a number of input data sources such as the place's website, community boards discussing the place, or third parties storing information on businesses. Thus, it needs to be consolidated before appearing on platforms such as Facebook or Google Maps.

Future Directions for Research

Although record linkage has been extensively studied, developments in data processing and learning continuously change its scope. For example, with the construction of knowledge bases, record linkage became a focus of those that wanted to interpret user feedback and content. They then developed new techniques for the new context in which record linkage was applied in. Next to specific applications of record linkage, there also exists a general trend to make record linkage part of existing data integration pipelines and to provide tools that enable users to execute record linkage with low overhead and in near real -time. As a result, one of the challenges and still an open problem of such an integration is interactive record linkage given that most traditional linkage algorithms have been designed for offline processing.

Cross-References

▶ Large-Scale Entity Resolution

References

Arasu A, Ré C, Suciu D (2009) Large-scale deduplication with constraints using dedupalog. In: Proceedings of the 25th international conference on data engineering, ICDE, 29 Mar–2 Apr 2009, Shanghai, pp 952–963. https://doi.org/10.1109/ICDE.2009.43

Bansal N, Blum A, Chawla S (2004) Correlation clustering. Mach Learn 56(1–3):89–113

Benjelloun O, Garcia-Molina H, Menestrina D, Su Q, Whang SE, Widom J (2009) Swoosh: a generic approach to entity resolution. VLDB J Int J Very Large Data Bases 18(1):255–276

Bilenko M, Mooney R, Cohen W, Ravikumar P, Fienberg S (2003) Adaptive name matching in information integration. IEEE Intell Syst 18(5):16–23

Cohen WW, Richman J (2002) Learning to match and cluster large high-dimensional data sets for data integration. In: Proceedings of the eighth ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 475–480

Dong XL, Srivastava D (2015) Big data integration. Synth Lect Data Manag 7(1):1–198

Elmagarmid AK, Ipeirotis PG, Verykios VS (2007) Duplicate record detection: a survey. IEEE Trans Knowl Data Eng 19(1):1–16

Fellegi IP, Sunter AB (1969) A theory for record linkage. J Am Stat Assoc 64(328):1183–1210

Gruenheid A, Dong XL, Srivastava D (2014) Incremental record linkage. Proc VLDB Endow 7(9):697–708

Hassanzadeh O, Chiang F, Miller RJ, Lee HC (2009) Framework for evaluating clustering algorithms in duplicate detection. PVLDB 2(1):1282–1293

Hernández MA, Stolfo SJ (1998) Real-world data is dirty: data cleansing and the merge/purge problem. Data Min Knowl Disc 2(1):9–37

Jaro MA (1978) Unimatch: a record linkage system: users manual. Bureau of the Census, Washington DC

Jaro MA (1989) Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. J Am Stat Assoc 84(406):414–420

Konda P, Das S, C PSG, Doan A, Ardalan A, Ballard JR, Li H, Panahi F, Zhang H, Naughton JF, Prasad S, Krishnan G, Deep R, Raghavendra V (2016) Magellan: toward building entity matching management systems. PVLDB 9(12):1197–1208. http://www.vldb.org/pvldb/vol9/p1197-pkonda.pdf

Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics Doklady, vol 10, pp 707–710

McCallum A, Nigam K, Ungar LH (2000) Efficient clustering of high-dimensional data sets with application to reference matching. In: Proceedings of the sixth ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 169–178

Monge AE, Elkan C et al (1996) The field matching problem: algorithms and applications. In: KDD, pp 267–270

Russell R (1922) Index. US Patent 1,435,663. https://www.google.com/patents/US1435663

Verroios V, Garcia-Molina H (2015) Entity resolution with crowd errors. In: 31st IEEE international conference on data engineering, ICDE 2015, Seoul, 13–17 Apr 2015, pp 219–230. https://doi.org/10.1109/ICDE.2015.7113286

Wang J, Kraska T, Franklin MJ, Feng J (2012) Crowder: crowdsourcing entity resolution. Proc VLDB Endow 5(11):1483–1494

Wang J, Li G, Kraska T, Franklin MJ, Feng J (2014) Leveraging transitive relations for crowdsourced joins. CoRR abs/1408.6916. http://arxiv.org/abs/1408.6916

Regular Path Queries

▶ Graph Path Navigation

Reinforcement Learning, Unsupervised Methods, and Concept Drift in Stream Learning

András A. Benczúr[1], Levente Kocsis[1], and Róbert Pálovics[2]
[1]Institute for Computer Science and Control, Hungarian Academy of Sciences (MTA SZTAKI), Budapest, Hungary
[2]Department of Computer Science, Stanford University, Stanford, CA, USA

Synonyms

Incremental learning; Online machine learning on streams; Stream learning

Definitions

- Data stream algorithms process a continuous stream of data with only a limited possibility to store past records.
- Online machine learning covers methods that update their models after observing a new event and can immediately serve predictions based on the updated model.

Overview

In this chapter, we give a brief overview the following special topics in online machine learning: Reinforcement learning; unsupervised data mining methods, including clustering, frequent itemset mining, dimensionality reduction, and topic modeling; finally, we list the most important concept drift adapting learning methods.

This Chapter is an extension of the other chapters in this Handbook, "Overview of Online Machine Learning in Big Data Streams" , "Online Machine Learning Algoriths over Data Streams", and "Recommender systems over Data Streams"

Reinforcement Learning

Reinforcement learning is an area of machine learning concerned with agents taking actions in an environment with the aim of maximizing some cumulative reward. It is different from supervised learning in that the environment does not provide a target behavior, only rewards depending on the actions taken.

The environment is typically assumed to be a Markov decision process (MDP). Formally, we assume a set of states, S; a set of actions, A; and a transition probability function $P(s, a, s')$ denoting the probability of reaching state s' after taking action a in state s and a reward function $R(s, a)$ denoting the immediate reward after taking action a in state s.

While there is a wide range of reinforcement learning algorithms (see, e.g., Sutton and Barto 1998), we focus here on algorithms that fit the streaming model and (possibly) deal with non-stationary environments. The streaming model of reinforcement learning is constrained not only by a continuous flow of input data but also by a continuous requisite to take actions.

Algorithms for Stationary Environments

Most reinforcement learning algorithms estimate the value of feasible actions and build a policy based on that value (e.g., by choosing the actions with the highest estimates with some additional exploration). An alternative to value prediction methods are policy gradient methods that update a parameterized policy depending on the performance.

Value Prediction

The value of a state is the expected cumulative reward starting from a given state and following a particular policy. In a similar way, the action value is the expected reward starting from a given state with a particular action.

Value prediction methods estimate the value of the state or the value of the actions in particular states. In the former case, to build a policy from the estimated values, an additional transition model is needed as well. Such a model is provided for some domains (e.g., by the rules of a game), but in many cases, the transition model needs to be learned as well. Action values can be used directly for constructing a policy without the need for a model.

Temporal difference (TD) learning learns the state value estimate $V(s)$ by the following update rule after each state transition (S_t, S_{t+1}):

$$V(S_t) \leftarrow (1-\alpha)V(S_t) + \alpha(R_t + \gamma V(S_{t+1})),$$

where α is a step-size and γ is the discount factor. TD learning was used in one of the first breakthroughs for reinforcement learning, that is, Tesauro's backgammon program (Tesauro 1995).

The best-known action-value prediction algorithm is Q-learning (Watkins and Dayan 1992). For each occurrence of a transition (S_t, A_t, S_{t+1}), the algorithm updates the action-value $Q(S_t, A_t)$ by

$$Q(S_t, A_t) \leftarrow (1-\alpha)Q(S_t, A_t)$$
$$+\alpha(R_t + \gamma \max_a Q(S_{t+1}, a)).$$

Q-learning using deep neural network to approximate the action-values has been successfully applied to playing some Atari games at human expert level (Mnih et al. 2015).

Another algorithm that learns action-values is Sarsa (Sutton 1996). For each sequence $S_t, A_t, S_{t+1}, and A_{t+1}$, the algorithm updates its estimates by

$$Q(S_t, A_t) \leftarrow (1-\alpha)Q(S_t, A_t) \\ +\alpha(R_t + \gamma Q(S_{t+1}, A_{t+1})).$$

Sarsa was successfully used by Ipek et al. (2008) for optimizing a DRAM memory controller.

The value prediction algorithms above were described with update rules for a tabular representation. In most cases, function approximation is used, and the update rules rely on a gradient step. Online enhancements of gradient descent as well as eligibility traces (Sutton and Barto 1998) can be applied to all variants.

Policy Gradient

While using value functions is more widespread, it is also possible to use a parameterized policy without relying on such functions. Parameterized policies are typically optimized by gradient ascent with respect to the performance of the policy.

A policy gradient algorithm, the REINFORCE algorithm (Williams 1992), was used to optimize policy in a Go playing program that outperforms the best human players (Silver et al. 2016).

Algorithms for Nonstationary Environments

Most reinforcement learning algorithms, including those discussed in the previous section, assume that the environment does not change over time. While incremental algorithms such as Q-learning can adapt well to nonstationary environments, it may be necessary to devise more explicit exploration strategies that can cope with changes, for example, in reward distribution.

A special case of reinforcement learning is the multiarmed bandit problem. In this case, the agent repeatedly selects an action from K possible choices, obtaining a reward after each choice. This problem retains the notion of reward; however, there are no states and consequently no state transitions. In the non-stochastic variant (Auer et al. 2002), the distribution of the rewards may change over time arbitrarily. Standard algorithms for this problem are Exp3 and its variants (Auer et al. 2002), which rely on an exponential selection algorithm, including some exploration terms as well. Contextual bandits extend the bandit setting with the notion of state (or context); however, state transitions are still missing. This framework was used, for instance, in (Li et al. 2010) to select personalized new stories. We note that the distinguishing feature of recommendation in a bandit setting is that the user can provide feedback only on the recommended items.

Unsupervised Data Mining

The most prominent class of unsupervised learning methods is **clustering** where instances have to be distributed into a finite set of clusters such that instances within the cluster are more similar to each other than to others in different clusters (Pang-Ning 2006). Batch clustering algorithms have been both studied and employed as data analysis tools for decades (Jain et al. 1999; Wunsch 2008). One frequently applied clustering method is **k-means** (Hartigan and Hartigan 1975) where cluster center selection and assignment to nearest centers are iteratively performed until convergence. Another is **DBSCAN** (Ester et al. 1996), a density-based method that groups points that are closely packed together.

Online clustering algorithms are surveyed among other places in Mahdiraji (2009), Kavitha and Punithavalli (2010), Aggarwal (2013), and Silva et al. (2013). The majority of the most relevant methods are data stream versions of k-means or its variants such as k-medians (Zhang et al. 1996; Bradley et al. 1998; Farnstrom et al. Farnstrom; O'callaghan et al. 2002; Guha et al. 2003; Aggarwal et al. 2003; Zhou et al. 2008; Gama et al. 2011; Kranen et al. 2011; Ackermann et al. 2012). Another set of results describes the data stream implementation of DBSCAN (Cao et al. 2006; Chen and Tu 2007; Kranen et al. 2011). Finally, an online hierarchical clustering algorithm that maintains similarity measures and hierarchically merges closest clusters is described in Rodrigues et al. (2006).

Finding **frequent itemsets** Agrawal et al. (1993) is another central unsupervised data mining task, both static and streaming. In brief, for a table of transactions and items, the task is to find all subsets of items that occur together in transactions with at least a prescribed frequency. Several variants of the task are described in Aggarwal and Han (2014). Online frequent itemset mining algorithms are surveyed in Cheng et al. (2008b). Algorithms based on counts of all past data in the stream (Chang and Lee 2003; Giannella et al. 2003; Li et al. 2004; Yu et al. 2004; Lee and Lee 2005) are also called landmark window-based approaches. In some of these algorithms, time adaptivity is achieved by placing more importance on recent items (Chang and Lee 2003; Giannella et al. 2003; Lee and Lee 2005). Sliding window-based approaches (Chang and Lee 2003; Chi et al. 2006; Chang and Lee 2006; Song et al. 2007; Cheng et al. 2008a; Li et al. 2009; Yen et al. 2011; Calders et al. 2014) are particularly suitable for processing data with concept drift. For a comparative overview, see, for example, how MOA's algorithm was selected (Quadrana et al. 2015). Note that a special subtask, finding frequent items in data streams, is already challenging and requires approximate data structures (Charikar et al. 2004).

Principal component analysis (PCA) is a powerful tool for dimensionality reduction (Jolliffe 1986) based on matrix factorization. Online variants are based on ideas to incrementally update the matrix decomposition (Bunch and Nielsen 1978; Hall et al. 2000; Brand 2002). The first PCA algorithms suitable for online learning are based on neural networks (Oja 1982; Sanger 1989; Oja 1992). Similar to linear classification and regression models, PCA can also apply the kernel trick to involve nonlinear modeling (Schölkopf et al. 1998). Iterative **kernel PCA** is described in Kim et al. (2005) and Günter et al. (2007) and online kernel PCA in Honeine (2012). We note that for nearest neighbor search in the low-dimensional space provided by PCA, the heuristics for selecting large inner products is applicable (Teflioudi et al. 2015).

Probabilistic topic modeling fits complex hierarchical Bayesian models to large document collections. A topic model reveals latent semantic structure that can be used for many applications. While PCA-like models can also be used for latent semantic analysis (Deerwester et al. 1990), recently the so-called **Latent Dirichlet Allocation (LDA)** (Blei et al. 2003) has gained popularity. Most topic model parameters can only be inferred based on Markov Chain Monte Carlo sampling, a method difficult to implement for online learning. LDA inference is possible based on either online Gibbs sampling (Song et al. 2005; Canini et al. 2009) or online stochastic optimization with a natural gradient step (Hoffman et al. 2010). Several online LDA variants are described in Smola and Narayanamurthy (2010), Ho et al. (2013), Li et al. (2014), Yuan et al. (2015), Yu et al. (2015), Jagerman et al. (2017).

Concept Drift and Adaptive Learning

In dynamically changing and nonstationary environments, we often observe concept drift as the result of data distribution change over time. The phenomenon and mitigation of concept (or dataset) drift for online learning are surveyed in several articles (Widmer and Kubat 1996; Tsymbal 2004; Quionero-Candela et al. 2009; Žliobaite et al. 2012; Gama et al. 2014). The area of transfer learning where the (batch) training and the test sets are different (Pan and Yang 2010) is closely related to concept drift (Storkey 2009) but more difficult in the sense that adaptation by learning part of the new data is not possible.

Adaptive learning refers to the technique of updating predictive models online to react to concept drifts. One of the earliest active learning systems is STAGGER (Schlimmer and Granger 1986). In Žliobaitė (2009), the main steps of online adaptive learning are summarized as (1) making assumptions about future distribution, (2) identifying change patterns, (3) designing mechanisms to make the learner adaptive, and (4) parameterizing the model at every time step.

A comprehensive categorization of concept drift adaptation techniques is found in Gama et al. (2014). Online learning algorithms can naturally adapt to evolving distributions. However, adaptation happens only as the old concepts are diluted due to the new incoming data, which is more suitable for gradual changes (Littlestone 1988; Domingos and Hulten 2000). For sudden changes, algorithms that maintain a sliding window of the last seen instances perform better (Widmer and Kubat 1996; Gama et al. 2004; Kuncheva and Žliobaitė 2009). Another option is to include explicit forgetting mechanisms (Koychev 2000; Klinkenberg 2004; Elwell and Polikar 2009). The most important distinction is whether changes are explicitly or implicitly detected: **Trigger-based** methods aim at detecting when concept drift occurs to build a new model from scratch (Gama et al. 2004). **Evolving learners**, by contrast, do not aim to detect changes but rather maintain the most accurate models at each time step. Evolving learners are method-specific, most of them are based on ensemble methods (Wang et al. 2003; Kolter and Maloof 2003).

A few papers (Minku et al. 2010; Moreno-Torres et al. 2012) give overviews of different types of environmental changes and concept drifts based on speed, recurrence, and severity. Drift can happen gradually or suddenly, in isolation, in tendencies or seasonally, and predictably or unpredictably, and its effect on classifier performance may or may not be severe. In Schlimmer and Granger (1986), Gama et al. (2004), several artificial data sets with different drift concepts, sudden or abrupt, and gradual changes are described.

A large variety of single classifier and ensemble models capable of handling concept drift are described in Tsymbal (2004). Perhaps the majority of the results consider tree-based methods Alberg et al. (2012). For example, concept drift adaptive online decision trees based on a statistical change detector that works on sliding windows are described in Bifet and Gavald (2009), Bifet (2010). More examples include Bayesian models (Gama et al. 2003; Bach and Maloof 2010), neural networks (Gama and Rodrigues 2007; Leite et al. 2013), and SVM (Syed et al. 1999; Klinkenberg and Joachims 2000). Concept drift adaptation methods exist for clustering (Rodrigues et al. 2006; Silva et al. 2013). Sliding window-based data stream frequent itemset mining is also adaptive (Quadrana et al. 2015). Some of the results do not follow the data stream computational model but rather use computational resources with little restriction. One class of such methods are incremental algorithms with partial memory (Maloof and Michalski 2004). We also note that there is a MOA-based software system for concept drift detection (Bifet et al. 2013).

Cross-References

- ▶ Overview of Online Machine Learning in Big Data Streams
- ▶ Online Machine Learning Algorithms over Data Streams
- ▶ Recommender systems over Data Streams

Acknowledgements Support from the EU H2020 grant Streamline No 688191 and the "Big Data—Momentum" grant of the Hungarian Academy of Sciences.

References

Ackermann MR, Märtens M, Raupach C, Swierkot K, Lammersen C, Sohler C (2012) Streamkm++: a clustering algorithm for data streams. J Exp Algorithmics (JEA) 17:2–4

Aggarwal CC (2013) A survey of stream clustering algorithms. In: Aggarwal CC, Reddy CK (eds) Data clustering: algorithms and applications. Chapman and Hall/CRC, Boca Raton, p 231

Aggarwal CC, Han J (2014) Frequent pattern mining. Springer, Cham

Aggarwal CC, Han J, Wang J, Yu PS (2003) A framework for clustering evolving data streams. In: Proceedings of the 29th international conference on very large data bases, vol 29. VLDB Endowment, pp 81–92

Agrawal R, Imielienski T, Swami A (1993) Mining association rules between sets of items in large databases. In: Bunemann P, Jajodia S (eds) Proceedings of the 1993 ACM SIGMOD conference on management of data. ACM Press, New York, pp 207–216

Alberg D, Last M, Kandel A (2012) Knowledge discovery in data streams with regression tree methods. Wiley Interdiscip Rev Data Min Knowl Disc 2(1): 69–78

Auer P, Cesa-Bianchi N, Freund Y, Schapire RE (2002) The nonstochastic multiarmed bandit problem. SIAM J Comput 32(1):48–77

Bach S, Maloof M (2010) A Bayesian approach to concept drift. In: Advances in neural information processing systems. Curran Associates, Inc., New York, pp 127–135

Bifet A (2010) Adaptive stream mining: Pattern learning and mining from evolving data streams. In: Proceedings of the 2010 conference on adaptive stream mining: pattern learning and mining from evolving data streams. IOS Press, pp 1–212

Bifet A, Gavaldà R (2009) Adaptive learning from evolving data streams. In: International symposium on intelligent data analysis. Springer, pp 249–260

Bifet A, Read J, Pfahringer B, Holmes G, Žliobaitė I (2013) CD-MOA: change detection framework for massive online analysis. In: International symposium on intelligent data analysis. Springer, pp 92–103

Blei DM, Ng AY, Jordan MI (2003) Latent dirichlet allocation. J Mach Learn Res 3:993–1022

Bradley PS, Fayyad UM, Reina C et al (1998) Scaling clustering algorithms to large databases. In: Proceedings of the 4th international conference on knowledge discovery and data mining, pp 9–15

Brand M (2002) Incremental singular value decomposition of uncertain data with missing values. In: Computer vision–ECCV 2002, pp 707–720

Bunch JR, Nielsen CP (1978) Updating the singular value decomposition. Numer Math 31(2):111–129

Calders T, Dexters N, Gillis JJ, Goethals B (2014) Mining frequent itemsets in a stream. Inf Syst 39:233–255

Canini K, Shi L, Griffiths T (2009) Online inference of topics with latent dirichlet allocation. In: Proceedings of the twelth international conference on artificial intelligence and statistics, in PMLR, Clearwater Beach, vol 5, pp 65–72

Cao F, Estert M, Qian W, Zhou A (2006) Density-based clustering over an evolving data stream with noise. In: Proceedings of the 2006 SIAM international conference on data mining. SIAM, pp 328–339

Chang JH, Lee WS (2003) Estwin: adaptively monitoring the recent change of frequent itemsets over online data streams. In: Proceedings of the 12th international conference on information and knowledge management. ACM, pp 536–539

Chang JH, Lee WS (2003) Finding recent frequent itemsets adaptively over online data streams. In: Proceedings of the 9th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 487–492

Chang JH, Lee WS (2006) Finding frequent itemsets over online data streams. Inf Softw Technol 48(7):606–618

Charikar M, Chen K, Farach-Colton M (2004) Finding frequent items in data streams. Theor Comput Sci 312(1):3–15

Chen Y, Tu L (2007) Density-based clustering for real-time stream data. In: Proceedings of the 13th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 133–142

Cheng J, Ke Y, Ng W (2008a) Maintaining frequent closed itemsets over a sliding window. J Inf Syst 31(3): 191–215

Cheng J, Ke Y, Ng W (2008b) A survey on algorithms for mining frequent itemsets over data streams. Knowl Inf Syst 16(1):1–27

Chi Y, Wang H, Philip SY, Muntz RR (2006) Catch the moment: maintaining closed frequent itemsets over a data stream sliding window. Knowl Inf Syst 10(3): 265–294

Deerwester SC, Dumais ST, Landauer TK, Furnas GW, Harshman RA (1990) Indexing by latent semantic analysis. J Am Soc Inf Sci 41(6):391–407. www.citeseer.nj.nec.com/deerwester90indexing.html

Domingos P, Hulten G (2000) Mining high-speed data streams. In: Proceedings of the 6th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 71–80

Elwell R, Polikar R (2009) Incremental learning in nonstationary environments with controlled forgetting. In: International joint conference on neural networks, IJCNN2009. IEEE, pp 771–778

Ester M, Kriegel HP, Sander J, Xu X et al (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of ACM SIGKDD, pp 226–231

Farnstrom F, Lewis J, Elkan C (2000) Scalability for clustering algorithms revisited. ACM SIGKDD Explorations Newsletter 2(1):51–57

Gama J, Medas P, Castillo G, Rodrigues P (2004) Learning with drift detection. In: Brazilian symposium on artificial intelligence. Springer, pp 286–295

Gama J, Rocha R, Medas P (2003) Accurate decision trees for mining high-speed data streams. In: Proceedings of the 9th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 523–528

Gama J, Rodrigues PP (2007) Stream-based electricity load forecast. In: European conference on principles of data mining and knowledge discovery. Springer, pp 446–453

Gama J, Rodrigues PP, Lopes L (2011) Clustering distributed sensor data streams using local processing and reduced communication. Intell Data Anal 15(1):3–28

Gama J, Žliobaite I, Bifet A, Pechenizkiy M, Bouchachia A (2014) A survey on concept drift adaptation. ACM Comput Surv (CSUR) 46(4):44

Giannella C, Han J, Pei J, Yan X, Yu PS (2003) Mining frequent patterns in data streams at multiple time granularities. Next Gener Data Mining 212:191–212

Guha S, Meyerson A, Mishra N, Motwani R, O'Callaghan L (2003) Clustering data streams: theory and practice. IEEE Trans Knowl Data Eng 15(3):515–528

Günter S, Schraudolph NN, Vishwanathan S (2007) Fast iterative kernel principal component analysis. J Mach Learn Res 8:1893–1918

Hall P, Marshall D, Martin R (2000) Merging and splitting eigenspace models. IEEE Trans Pattern Anal Mach Intell 22(9):1042–1049

Hartigan JA, Hartigan J (1975) Clustering algorithms, vol 209. Wiley, New York

Ho Q, Cipar J, Cui H, Lee S, Kim JK, Gibbons PB, Gibson GA, Ganger G, Xing EP (2013) More effective distributed ML via a stale synchronous parallel parameter server. In: Advances in neural information processing systems. Neural Information Processing Systems Foundation, Inc., Lake Tahoe, pp 1223–1231

Hoffman M, Bach FR, Blei DM (2010) Online learning for latent dirichlet allocation. In: Lafferty JD, Williams CKI, Shawe-Taylor J, Zemel RS, Culotta A (eds) Advances in neural information processing systems. Curran Associates, Inc., New York, pp 856–864

Honeine P (2012) Online kernel principal component analysis: a reduced-order model. IEEE Trans Pattern Anal Mach Intell 34(9):1814–1826

Ipek E, Mutlu O, Martínez JF, Caruana R (2008) Self-optimizing memory controllers: A reinforcement learning approach. In: Proceedings of 35th international symposium on computer architecture, ISCA'08. IEEE, pp 39–50

Jagerman R, Eickhoff C, de Rijke M (2017) Computing web-scale topic models using an asynchronous parameter server. In: Proceedings of the 40th international ACM SIGIR conference on research and development in information retrieval. ACM

Jain AK, Murty MN, Flynn PJ (1999) Data clustering: a review. ACM Comput Surv (CSUR) 31(3): 264–323

Jolliffe IT (1986) Principal component analysis and factor analysis. In: Jolliffe IT (ed) Principal component analysis. Springer, New York, pp 115–128

Kavitha V, Punithavalli M (2010) Clustering time series data stream-a literature survey. arXiv preprint arXiv:1005.4270

Kim KI, Franz MO, Scholkopf B (2005) Iterative kernel principal component analysis for image modeling. IEEE Trans Pattern Anal Mach Intell 27(9):1351–1366

Klinkenberg R (2004) Learning drifting concepts: example selection vs. example weighting. Intell Data Anal 8(3):281–300

Klinkenberg R, Joachims T (2000) Detecting concept drift with support vector machines. In: ICML, pp 487–494

Kolter JZ, Maloof MA (2003) Dynamic weighted majority: a new ensemble method for tracking concept drift. In: Proceedings of the 3rd IEEE international conference on data mining, ICDM2003. IEEE, pp 123–130

Koychev I (2000) Gradual forgetting for adaptation to concept drift. In: Proceedings of the ECAI 2000 workshop on current issues in spatio-temporal reasoning

Kranen P, Assent I, Baldauf C, Seidl T (2011) The clustree: indexing micro-clusters for anytime stream mining. Knowl Inf Syst 29(2):249–272

Kuncheva LI, Žliobaitė I (2009) On the window size for classification in changing environments. Intell Data Anal 13(6):861–872

Lee D, Lee W (2005) Finding maximal frequent itemsets over online data streams adaptively. In: Proceedings of the 5th IEEE international conference on data mining. IEEE, pp 8–pp

Leite D, Costa P, Gomide F (2013) Evolving granular neural networks from fuzzy data streams. Neural Netw 38:1–16

Li HF, Ho CC, Lee SY (2009) Incremental updates of closed frequent itemsets over continuous data streams. Expert Syst Appl 36(2):2451–2458

Li HF, Lee SY, Shan MK (2004) An efficient algorithm for mining frequent itemsets over the entire history of data streams. In: Proceedings of the 1st international workshop on knowledge discovery in data streams, vol 39

Li L, Chu W, Langford J, Schapire RE (2010) A contextual-bandit approach to personalized news article recommendation. In: Proceedings of the 19th international conference on world wide web. ACM, pp 661–670

Li M, Andersen DG, Park JW, Smola AJ, Ahmed A, Josifovski V, Long J, Shekita EJ, Su BY (2014) Scaling distributed machine learning with the parameter server. In: Proceedings of 11th USENIX symposium on operating systems design and implementation (OSDI14). USENIX Association, pp 583–598

Littlestone N (1988) Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm. Mach Learn 2(4):285–318

Mahdiraji AR (2009) Clustering data stream: a survey of algorithms. Int J Knowl Based Intell Eng Syst 13(2):39–44

Maloof MA, Michalski RS (2004) Incremental learning with partial instance memory. Artif Intell 154(1–2): 95–126

Minku LL, White AP, Yao X (2010) The impact of diversity on online ensemble learning in the presence of concept drift. IEEE Trans Knowl Data Eng 22(5): 730–742

Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, Graves A, Riedmiller M, Fidjeland AK, Ostrovski G et al (2015) Human-level control through deep reinforcement learning. Nature 518(7540):529–533

Moreno-Torres JG, Raeder T, Alaiz-RodríGuez R, Chawla NV, Herrera F (2012) A unifying view on dataset shift in classification. Pattern Recog 45(1):521–530

O'callaghan L, Mishra N, Meyerson A, Guha S, Motwani R (2002) Streaming-data algorithms for high-quality clustering. In: Proceedings of 18th international conference on data engineering. IEEE, pp 685–694

Oja E (1982) Simplified neuron model as a principal component analyzer. J Math Biol 15(3):267–273

Oja E (1992) Principal components, minor components, and linear neural networks. Neural Netw 5(6):927–935

Pan SJ, Yang Q (2010) A survey on transfer learning. IEEE Trans Knowl Data Eng 22(10):1345–1359

Pang-Ning T, Steinbach M, Kumar V et al (2006) Introduction to data mining. Pearson Addison Wesley, Boston/Toronto

Quadrana M, Bifet A, Gavalda R (2015) An efficient closed frequent itemset miner for the MOA stream mining system. AI Commun 28(1): 143–158

Quionero-Candela J, Sugiyama M, Schwaighofer A, Lawrence ND (2009) Dataset shift in machine learning. The MIT Press: Cambridge

Rodrigues PP, Gama J, Pedroso JP (2006) ODAC: hierarchical clustering of time series data streams. In: Proceedings of the 2006 SIAM international conference on data mining. SIAM, pp 499–503

Sanger TD (1989) Optimal unsupervised learning in a single-layer linear feedforward neural network. Neural Netw 2(6):459–473

Schlimmer JC, Granger RH (1986) Incremental learning from noisy data. Mach Learn 1(3): 317–354

Schölkopf B, Smola A, Müller KR (1998) Nonlinear component analysis as a kernel eigenvalue problem. Neural Comput 10(5):1299–1319

Silva JA, Faria ER, Barros RC, Hruschka ER, de Carvalho AC, Gama J (2013) Data stream clustering: A survey. ACM Comput Surv (CSUR) 46(1):13

Silver D, Huang A, Maddison CJ, Guez A, Sifre L, Van Den Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M et al (2016) Mastering the game of go with deep neural networks and tree search. Nature 529(7587):484–489

Smola A, Narayanamurthy S (2010) An architecture for parallel topic models. Proc VLDB Endow 3(1–2): 703–710

Song G, Yang D, Cui B, Zheng B, Liu Y, Xie K (2007) Claim: an efficient method for relaxed frequent closed itemsets mining over stream data. In: International conference on database systems for advanced applications. Springer, pp 664–675

Song X, Lin CY, Tseng BL, Sun MT (2005) Modeling and predicting personal information dissemination behavior. In: Proceedings of the 11th ACM SIGKDD international conference on knowledge discovery in data mining. ACM, pp 479–488

Storkey A (2009) When training and test sets are different: characterizing learning transfer. In: Sugiyama C, Lawrence S (eds) Dataset shift in machine learning. MIT Press, Cambridge, pp 3–28

Sutton RS (1996) Generalization in reinforcement learning: successful examples using sparse coarse coding. In: Touretzky DS, Mozer MC, Hasselmo ME (eds) Advances in neural information processing systems, vol 8. MIT Press, Cambridge, pp 1038–1044

Sutton RS, Barto AG (1998) Reinforcement learning: an introduction, vol 16. MIT Press, Cambridge, pp 285–286

Syed NA, Liu H, Sung KK (1999) Handling concept drifts in incremental learning with support vector machines. In: Proceedings of the 5th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 317–321

Teflioudi C, Gemulla R, Mykytiuk O (2015) Lemp: fast retrieval of large entries in a matrix product. In: Proceedings of the 2015 ACM SIGMOD international conference on management of data. ACM, pp 107–122

Tesauro G (1995) Td-gammon: a self-teaching backgammon program. In: Applications of neural networks. Springer, Boston, pp 267–285

Tsymbal A (2004) The problem of concept drift: definitions and related work. Technical Report 2, Computer Science Department, Trinity College Dublin

Wang H, Fan W, Yu PS, Han J (2003) Mining concept-drifting data streams using ensemble classifiers. In: Proceedings of the 9th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 226–235

Watkins CJ, Dayan P (1992) Q-learning. Mach Learn 8(3–4):279–292

Widmer G, Kubat M (1996) Learning in the presence of concept drift and hidden contexts. Mach Learn 23(1):69–101

Williams RJ (1992) Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach Learn 8(3–4):229–256

Xu R, Wunsch D (2008) Clustering, vol 10. Wiley, Hoboken

Yen SJ, Wu CW, Lee YS, Tseng VS, Hsieh CH (2011) A fast algorithm for mining frequent closed itemsets over stream sliding window. In: 2011 IEEE international conference on fuzzy systems (FUZZ). IEEE, pp 996–1002

Yu HF, Hsieh CJ, Yun H, Vishwanathan S, Dhillon IS (2015) A scalable asynchronous distributed algorithm for topic modeling. In: Proceedings of the 24th international conference on world wide web, pp 1340–1350. International World Wide Web Conferences Steering Committee

Yu JX, Chong Z, Lu H, Zhou A (2004) False positive or false negative: mining frequent itemsets from high speed transactional data streams. In: Proceedings of the 13th international conference on very large data bases, vol 30. VLDB Endowment, pp 204–215

Yuan J, Gao F, Ho Q, Dai W, Wei J, Zheng X, Xing EP, Liu TY, Ma WY (2015) Lightlda: big topic models on modest computer clusters. In: Proceedings of the 24th international conference on world wide web. International World Wide Web Conferences Steering Committee, pp 1351–1361

Zhang T, Ramakrishnan R, Livny M (1996) Birch: an efficient data clustering method for very large databases. ACM SIGMOD Rec 25(2): 103–114

Zhou A, Cao F, Qian W, Jin C (2008) Tracking clusters in evolving data streams over sliding windows. Knowl Inf Syst 15(2):181–214

Žliobaitė I (2009) Learning under concept drift: an overview. Technical report, Vilnius University

Žliobaite I, Bifet A, Gaber M, Gabrys B, Gama J, Minku L, Musial K (2012) Next challenges for adaptive learning systems. ACM SIGKDD Explor Newsl 14(1): 48–55

Rendezvous Architectures

Ted Dunning and Ellen Friedman
MapR Technologies and Apache Software Foundation, Santa Clara, CA, USA

Definitions

Rendezvous architectures are a class of streaming microservice architecture designed to manage multiple implementations of a streaming function so that new implementations can be deployed easily and accurately, input data can be archived precisely, and past operations can be audited while maintaining strict service-level guarantees. Although suitable for more general application, rendezvous architectures are particularly useful for the special case of managing machine learning models. The discussion here follows the trend of associating rendezvous architecture with machine learning and is largely limited to that context.

Historical Background

The management of logistics in machine learning systems has always been notoriously difficult, particularly when there are multiple data sources and, as is almost always the case, multiple models being iteratively developed, evaluated, and deployed at the same time. Requirements include stability and reliability in production along with agility in response to changes. The challenges are even greater when this needs to be done at large scale, in a production setting, and in such a way as to meet critical service-level agreements that increasingly include guaranteed low latency (Böse et al. 2017; Dunning and Friedman 2017). People who are new to machine learning may think that machine learning models can be managed in the same way as normal software, by using continuous integration techniques to deploy a single high-quality model, but the reality for machine learning systems is far different. Successful projects involve a large number of models both in development and in production simultaneously. Simply determining which model is more accurate is difficult. Moreover, the proliferation of machine learning frameworks means that production models may not share common base technologies. Maintaining strict availability and latency service levels while simultaneously dealing with these other difficulties can be particularly difficult.

Another source of challenges lies in the fact that the process of machine learning model development and evaluation is iterative and needs to operate in a sufficiently flexible and agile way that allows for experimentation and timely response to the need for new or retuned models without compromising operational characteristics like worst-case latency or failure tolerance. Even when models are running well in production, there is still a need for continuous deployment in part because external conditions change. For example, customer behavior may change, fraudsters may develop new tricks, or business goals and requirements may be realigned, thus degrading the performance of a once-effective model.

Additional challenges in machine learning logistics arise from the need for consistency of conditions relative to the production environment during experimentation and evaluation. Even apparently trivial environmental differences between development and production can result in unwanted surprises upon deployment.

It is also important to allow large amounts of experimentation and trials in production settings, but at the same time, it is critical to bound the risk of failed experiments.

Solving these logistical problems with bounded risk is the primary motivation for rendezvous architectures.

Foundations

Rendezvous architectures are best understood in the context of managing machine learning logistics in large-scale systems even though they can be applied to any streaming transformation. In order to meet the challenges of handling input and output data and managing models, the rendezvous design takes advantage of several key techniques. These include stream-based architecture, a microservice approach, and the use of containers for isolation. The major goals of rendezvous architectures are to be able to:

- Preserve raw data that exactly reflects operational reality.
- Make it easier to manage multiple model versions.
- Provide for smooth and predictable model deployment into production.
- Meet stringent latency and availability guarantees with no planned violations and no accidental violations except under major failure scenarios.
- Work in an iterative and agile fashion.

Overview of Rendezvous Architecture

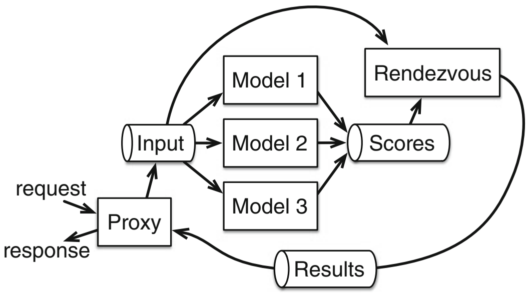
The distinctive architectural features of rendezvous architectures are the decomposition of a query-response microservice into a set of streaming microservices, the *rendezvous server* itself, and the way that all live models take inputs from and put their results to common streams.

The main rationale for using streaming as the backbone of the rendezvous architecture is that new models can start reading requests from the input stream and writing results into the output stream with no configuration or service discovery burden. This means that the overall management of running models is very simple and dynamic. New model implementations merely need to be told where to get their input and where to put results, and they can start operation. If a model is slow as it warms up or has other problems that prevent it from keeping up with production loads, there is no problem with partial results.

The rendezvous server itself is what makes this work. Results from models for each incoming request are buffered by the rendezvous server until one of the results is received that meets the requirements of a schedule that defines the trade-off between model preference and latency guarantees. That is, we might have a favored model, but we will only wait for a certain amount of time for that model to give us a result before using a less-favored model. The schedule can even specify a default result to be used in case no model produces a result in the required time.

The overall design of a rendezvous architecture is depicted in Fig. 1. In this figure, a request is accepted by any of a number of proxies and inserted into the `input` stream. All live models run in containers, evaluate as many requests as they can, and put all responses into the `scores` stream. The rendezvous server sees all incoming requests and starts a timer for each such request. As results are received from the live models, they are correlated against pending requests. The rendezvous server has a schedule of preferred model priorities versus latencies, and as soon as a result meets the requirements of the schedule, that result is put into the `results` stream and returned by the original proxy. All other results are ignored by the rendezvous server, although they are preserved in the `scores` stream for a configurable amount of time.

The final step of deploying a new model into production after it has been started and is evaluating requests is for the rendezvous server to stop ignoring the new model's results. This design makes it easy to roll out new models or to roll back to a previous model if performance of the



Rendezvous Architectures, Fig. 1 The general structure of a rendezvous architecture. Incoming requests are handled by one or more proxies that inject requests into a stream so that multiple models can evaluate each request. The rendezvous server sees the original request and selects a result to return to the proxy from the scores stream according to a policy designed to ensure service levels new model is not as expected. Moreover, if the new model crashes or slows down unacceptably, the rendezvous schedule will allow other models to take up the slack and avoid violation of service-level guarantees.

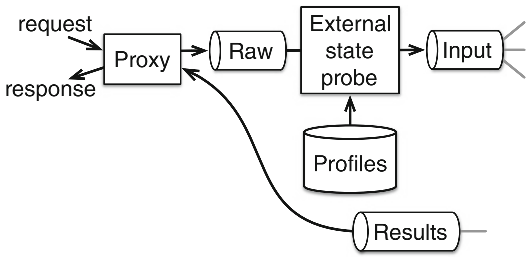
Note that this design also allows raw data to be augmented by additional variables or features before use as input data. This is illustrated in Fig. 2 where the raw request stream is transformed by the addition of common features or insertion of external state information before the models evaluate the requests. By adding common features or injecting external state at this point, all models are guaranteed to have exactly the same inputs.

In order to manage the conflicting environmental requirements for different models, it is a best practice to run each model in a container. This decreases the likelihood that the model behavior will change as it is put into production due to changes in the execution environment. Typically, an orchestration system such as Kubernetes (Burns et al. 2017) is used to manage such containers, but the very simple coordination and discovery for models in a rendezvous architecture make almost any orchestration system acceptable.

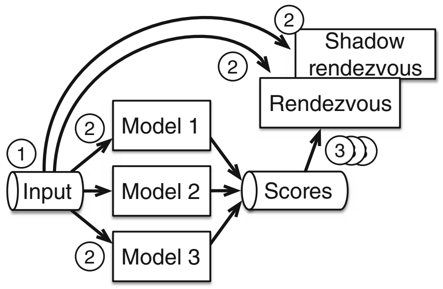
Versioning of the Rendezvous Components

All of the components of the rendezvous architecture can be upgraded during operation with no downtime or latency disturbances. This can be done by injecting transition tokens into the input of the system. As these tokens pass through the system, old versions of each component stop processing of messages, and new components take over. For elements such as the feature extraction or external state access which are pure transformations, this is straightforward since all messages are processed entirely by one version or the other. Figure 3 illustrates the more complex case of upgrading the rendezvous server itself. Extra complexity arises because the old rendezvous server has to keep track of any pending requests that arrived before the transition even if the results arrive after the transition, while the new rendezvous server must be sure to keep track of only requests that arrive after the transition token. The process starts with the token at position 1 in the figure at the input to the system. At this point, a shadow rendezvous server will have been started and will be listening to the same inputs as the production server. The token propagates to all models and to the inputs of the current and shadow rendezvous servers at the points marked 2. The models each pass the transition token unchanged so that multiple copies appear in the `scores` stream.

The production rendezvous server buffers results for all requests that arrive *before* the transition token is seen on the `input` stream. The



Rendezvous Architectures, Fig. 2 The incoming requests can be augmented with the result of probing external state (such as user profiles). The same module could be used to compute commonly used input features. This approach avoids race conditions between different models' probes of the external state and makes it easier to maintain exact consistency of all model inputs



Rendezvous Architectures, Fig. 3 Upgrading the rendezvous server using a transition token that passes through the system to locations marked 1, 2, and 3 at times t_1, t_2, and t_3, respectively. Control over processing transitions in stages from the production rendezvous server to the shadow server that will become the new production server after the transition token has passed all the way through shadow server takes over the production role by initiating coverage on all requests that are received after the transition token. The overall result is that only one rendezvous server ever produces a result for each request even if a rendezvous server is restarted during the upgrade. The old production rendezvous server can be retired once it reports results for all pending requests. In fact, it may be preferable to keep the old server around for a time in case it becomes necessary to fail back.

Since maximum response latencies for systems where a rendezvous server makes sense are typically less than a second, the hand-off process will appear to be nearly instantaneous once the transition token is introduced to the system.

This upgrade process is related to the Chandy-Lamport (Chandy and Lamport 1985) checkpointing algorithm and to the process that Apache Flink (Friedman and Tzoumas 2016) uses to upgrade program versions but is somewhat simpler.

Key Properties of Rendezvous Architectures

The rendezvous architecture uses streaming microservices (Dunning and Friedman 2016) internally to build a microservice that exposes a query-response interface externally. A rendezvous architecture is characterized primarily by a few properties:

Synchronous requests are evaluated internally in a streaming microservice style. This is highly unusual in large-scale machine learning systems. Much more common is to use a load balancer that distributes individual requests to individual models that evaluate requests synchronously. The use of streams to distributed requests is unusual, and it makes many operations in the rendezvous architecture much easier, largely because persistent streams allow simpler handling of failure modes.

Input requests are sent to all live models identically. In machine learning systems that do not use rendezvous techniques, secondary requests may be made by the load balancer to fallback models, mostly as hedges against latency violations. Similarly, requests are sometimes broken into pieces and sent to shared decision engines. Results are collected until complete or until a deadline looms and the results are "complete enough." Doing this with streaming and nearly universally for all queries as in the rendezvous server is very unusual but it deployment of models and the framework itself. It also makes it relatively easy to guarantee response latencies even in the presence of process failures or during the deployment and warmup of new models or the retirement of old ones.

All input requests are evaluated by multiple models. In non-rendezvous systems, evaluation of individual requests by multiple models at the same time, sometimes known as speculative evaluation, is the exception rather than the rule. Historically, high degrees of speculative execution were avoided due to worries about computational capacity, but this is much less of a consideration now, so the aggressive speculative execution of the rendezvous architecture is more viable.

All model results are put into a common stream of results. This is a unique characteristic of rendezvous architectures which allows the implementation of the rendezvous server to be substantially simplified.

A rendezvous server selects which result for each request to return Allowing the rendezvous server to select which result to return according to a trade-off schedule allows the concerns of accuracy and reliability to be separated. Accuracy, the primary goal of the model developer, is embodied in the models themselves. Reliability, particularly with respect to the satisfaction of service-level guarantees, is the focus of site reliability engineers and operations staff and is the key purpose of the rendezvous server. Separating these concerns makes it easier to satisfy both.

Key Applications

The primary application of rendezvous architectures is to allow continuous integration of machine learning models while maintaining promised service levels.

The types of machine learning for which the rendezvous pattern of architecture is most appropriate are those that involve decisioning, that is to say, applications that should return a "correct" answer as estimated by a model without much correlation between different decisions. These systems generally are synchronous in design in that they involve a query-response pattern of interaction with bounded request and response sizes. Examples include predictive analytics, image or speech recognition using deep learning techniques such as medical image analysis or speech-to-text, fraud detection in financial transactions, and IoT sensor data processing for manufacturing or for churn prediction in telecommunications and for web-based decision systems.

Cross-References

▶ Elasticity
▶ Streaming Microservices

References

Böse JH, Flunkert V, Gasthaus J, Januschowski T, Lange D, Salinas D, Schelter S, Seeger M, Wang Y (2017) Probabilistic demand forecasting at scale. Proc VLDB Endow 10:1694-1705

Burns B, Hightower K, Beda J (2017) Kubernetes: up and running dive into the future of infrastructure. O'Reilly Media Inc., Sebastopol

Chandy KM, Lamport L (1985) Distributed snapshots: determining global states of distributed systems. ACM Trans Comput Syst 3(1):63–75. http://doi.acm.org/10.1145/214451.214456

Dunning T, Friedman E (2016) Streaming architecture using Apache Kafka and MapR streams. O'Reilly Media. http://bit.ly/streaming-dunning

Dunning T, Friedman E (2017) Machine learning logistics: model management in the real world. O'Reilly, Sebastopol. https://mapr.com/ebook/machine-learning-logistics/

Friedman E, Tzoumas K (2016) Introduction to Apache flink: stream processing for real time and beyond. O'Reilly Media Inc. https://mapr.com/introduction-to-apache-flink/

Research Directions

▶ Big Data in Computer Network Monitoring

Resource Management

▶ Advancements in YARN Resource Manager

Review

▶ Auditing

Robust Data Partitioning

Alekh Jindal[1], Anil Shanbhag[2], and Yi Lu[2]
[1]Microsoft, Redmond, WA, USA
[2]MIT, Cambridge, MA, USA

Synonyms

Adaptive partitioning; Ad Hoc Query Processing; Distributed databases

Definitions

This chapter described new advancements in data partitioning for modern applications that are ad-hoc in nature and do not have any upfront query workload.

Overview

Data partitioning is a well-known technique for improving the performance of database applications. By splitting data into partitions and only accessing those that are needed to answer a query, databases can avoid reading data that is not relevant to the query being executed, often significantly improving performance. Additionally, when partitions are spread across multiple machines, databases can effectively parallelize query processing across them.

This chapter summarizes traditional data partitioning techniques, motivates the need for a more *robust* data partitioning over modern ad hoc query workloads, introduces the concept of hyper-partitioning for creating a robust partitioning tree and hyper-join to process join queries over such a partitioning tree, and finally discusses repartitioning techniques for adapting the partitioning tree in a robust manner.

Traditional Partitioning Approaches

The traditional approach to data partitioning is to split a table on some key, using hash or range partitioning. This helps queries that have selection predicates involving the key go faster, by only accessing the relevant portions of data. Likewise, for queries with joins, queries will benefit when the database is partitioned on attributes involved in the join, due to local co-partitioned join processing in each partition. Because of these performance gains, many techniques have been proposed in the literature.

Workload-Based Partitioning

The typical approach is to find a good data partitioning for a given query workload. These approaches assume that the query workload is either provided upfront or collected over time, and try to choose the best partitioning for that workload. Examples include fine-grained partitioning (Curino et al. 2010), hybrid of fine- and coarse-grained partitioning (Quamar et al. 2013), skew-aware partitioning (Pavlo et al. 2012), deep integration of partitioning with the query optimizer (Nehme and Bruno 2011), interdependence of different physical design decisions (Zilio et al. 2004), integrating vertical and horizontal partitioning decisions (Agrawal et al. 2004), and partitioning a B^+-Tree on primary keys (Graefe 2003). Workload-based partitioning need to be reconfigured every time the workload changes.

Multidimensional Partitioning

Several partitioning techniques have been proposed for multidimensional data, e.g., k-d trees, R-trees, and quadtrees. These are typically used for spatial data with two dimensions. Other approaches include binary search trees such as splay trees (Sleator and Tarjan 1985) and MAGIC to *decluster* data on multiple attributes (Ghandeharizadeh and DeWitt 1994). Recent approaches layer multidimensional index structures over distributed data in large clusters. This includes SpatialHadoop (Eldawy and Mokbel 2015), MD-HBase (Nishimura et al. 2011), and epiC (Wang et al. 2010) or adapting the multidimensional index to the workload in TrajStore (Cudré-Mauroux et al. 2010). Commercially, Oracle and MySQL support *sub-partitioning* to create nested partitions on multiple attributes. IBM DB2 supports multidimensional clustering tables to cluster data along multiple dimensions and build block-based indices on them.

Big Data Partitioning

Big data storage systems, such as HDFS, partition datasets based on size. Developers can later create attribute-based partitioning using a variety of data processing tools, e.g., Apache Hive and SCOPE (Zhou et al. 2012). However, such a partitioning is no different than traditional database partitioning since (i) partitioning is a static one time activity and (ii) the partitioning keys must be known a priori and provided by users. Recently, Sun et al. (2014) proposed to create data blocks in HDFS based on the features extracted from each input tuple. Again, the features are selected based on a workload, and the goal is to cluster tuples with similar features in the same data block. AQWA looks at adaptive data partitioning for spatial data (two dimensions). Their techniques do not scale to higher dimensions (Aly et al. 2015). Apart from single table partitioning, Hadoop++ (Dittrich et al. 2010) and CoHadoop (Eltabakh et al. 2011) propose to co-partition datasets in HDFS to speed up join queries. These systems still assume a workload.

Database Cracking

Database cracking (Idreos et al. 2007) is a technique to adapt the layout of data and indexes as queries arrive. Partial sideways cracking extends this idea to generate adaptive indexes on multiple columns (Idreos et al. 2009). Cracking is designed for in-memory column stores, and it adapts the data to *every* query in the system. It does not naturally apply to a distributed setting for two main reasons. First, the cost of repartitioning in a distributed setting is higher than in a main memory system. So, it is very expensive to repartition data on every access as cracking does. Second, cracking splits the data on every new predicate it encounters, which can result in a large number of blocks. However, in a distributed setting, the number of data blocks that can be created is limited because blocks must be a certain size to amortize latencies of disk and network access. As a result, adding a split for a new predicate involves merging existing partitions and re-splitting them to keep the number of blocks constant.

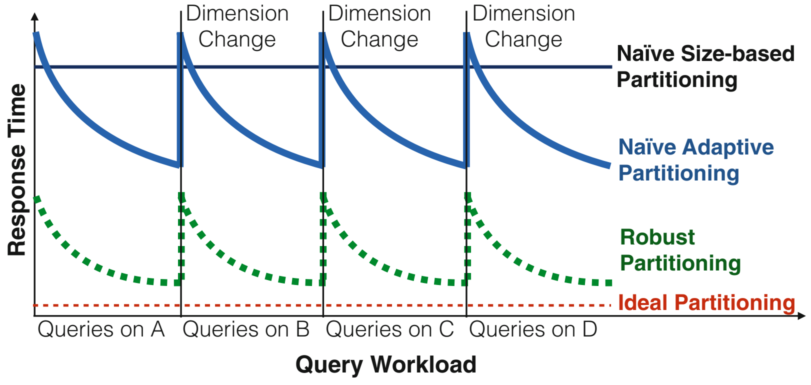
Robustness

Modern data analytics has newer data partitioning needs. Data science, for instance, often involves looking for anomalies and trends in data. There is no representative workload for this kind of ad hoc, exploratory analysis, and the set of tables and predicates of interest will often shift over time. For example, an analyst may look for patterns in a database of multidimensional web click events (with user history, demographic information, and platform information as dimensions). The analyst may want to view this data according to any of its dimensions – e.g., they may want to query according to the user's past browsing patterns, by their age or income or by whether they are using a mobile phone or a laptop. As the specific set of attributes of interest is not necessarily known upfront, workload-based partitioning techniques cannot be applied. Furthermore, as the workload is ad hoc in nature, database cracking cannot be applied as well. Figure 1a illustrates the data partitioning dilemma that analysts face with modern workloads.

Analysts are either stuck with naïve size-based partitioning that offers no data skipping capability and hence very poor performance (full scan). Or, alternatively, they could pick one of the more recent adaptive partitioning techniques, e.g., cracking (Idreos et al. 2007) that would make the first few queries even slower than full scan, but will gradually improve if successive queries are on the same dimension, i.e., having a selection predicate on the same attribute. In case the query dimension changes, the performance again goes back worse than full scan before gradually improving with successive queries on the new dimension (referred to as naïve adaptive partitioning). This is really painful for an analyst exploring multiple dimensions: analysts want a data partitioning scheme that is **robust** *to the ad hoc nature of the modern workloads and provides good performance from the first query itself, adaptively improving from there on.*

Hyper-partitioning

Distributed storage systems, such as HDFS, subdivide a dataset into chunks, called blocks, based on size (usually 128 MB). Workload-based partitioning techniques for such systems, including content-based chunking (Bhatotia et al. 2011) and feature-based blocking (Sun et al. 2014), create blocks such that irrelevant blocks could be quickly skipped for the specific query workload. Hyper-partitioning goes a step further by creating blocks based on a partitioning tree that allows to skip data over almost *all* ad hoc queries,



Robust Data Partitioning, Fig. 1 Need for robust data partitioning

without having any information about the query workload. Such a partitioning also serves as a good starting point for an adaptive query executor to improve upon.

Since hyper-partitioning partitions the data along several dimensions, it could end up de-clustering the data blocks across machines and performing random I/Os for each block. However, this is still fine; large block sizes in distributed file systems (Ghemawat et al. 2003) combined with fast network speeds lead to remote reads being almost as fast as local reads (Ananthanarayanan et al. 2011; Binnig et al. 2016). Essentially, hyper-partitioning sacrifices some data locality in order to quickly locate the relevant portions of the data on each machine in a distributed setting.

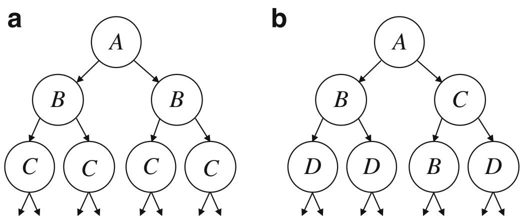
The rest of this section first introduces the notions of robust partitioning tree and attribute allocations in that tree and then describes how to construct and query such a tree.

Robust Partitioning Tree

The hyper-partitioning partitioning tree, or simply the *robust tree*, is represented as a balanced binary tree, i.e., the dataset is successively partitioned into two until it reaches the maximum partition size. For HDFS, hyper-partitioning takes the block size as the maximum partition size. The choice of binary tree is deliberate as it is more general (a four-way partitioning can be achieved by two successive two-way partitioning) as well as fine-granular when adapting the tree to workload changes later. Each node in the tree is represented as A_p, where A is the attribute being partitioned on and p is the cut point. All tuples with $A \leq p$ go to the left subtree and rest go to the right subtree. A leaf node in the tree is a **bucket**, having a unique identifier and a file name in the underlying file system. This file contains the tuples that satisfy the predicates of all nodes traversing upwards from the bucket to the root of the tree. Note that an attribute can appear in multiple nodes in the tree. Having multiple occurrences of an attribute in the same branch of the tree increases the number of ways the data is partitioned on that attribute.

Traditional binary partitioning trees, such as k-d tree (Bentley 1975), partition the space by considering the attributes in a round robin fashion, until the smallest partition size is reached. Hence, the tree can only accommodate as many attributes as the depth of the tree. Figure 2 shows a k-d tree where the three levels of the tree divide the dataset on attributes A, B, and C, respectively. In general, for a dataset size D, minimum partition size P, and n way partitioning over each attribute, the partitioning tree contains $\lfloor log_n \frac{D}{P} \rfloor$ attributes. With $n = 2$, $D = 1\,\text{TB}$, and $P = 64\,\text{MB}$, only 14 attributes can be accommodated in the partitioning tree. However, many real-world schemas have way more attributes.

In contrast to k-d tree, the robust tree performs *heterogeneous branching* in order to accommodate more attributes by partitioning



Robust Data Partitioning, Fig. 2 Multidimensional partitioning tree. (**a**) k-d tree. (**b**) Robust tree

different branches of the partitioning tree on different attributes. In other words, robust tree sacrifices the best performance on a few attributes to achieve robustness, i.e., improved performance over more attributes. This is reasonable as without a workload, there is no evident reason to prefer one attribute over another. Figure 2b shows a robust partitioning tree. After partitioning on attribute A, the left side of the tree partitions on B, while the right side partitions on C. Thus, the tree is now able to accommodate $four$ attributes, instead of $three$. However, attributes B and D are each partitioned on 75% of the data, while attribute C is partitioned on 50%. Ad hoc queries would now gain partially over all four attributes, which makes the partitioning more effective.

The number of attributes in the robust partitioning tree, with c as the minimum fraction of the data partitioned by each attribute and r as the number of replicas, is given as $\frac{1}{c} \cdot \lfloor \log_n \frac{D}{P} \rfloor$. With $n = 2$, $D = 1\,\text{TB}$, $P = 64\,\text{MB}$, and $c = 50\%$, the number of attributes that can be partitioned is 28. Note that the number of attributes that can be partitioned increases with the dataset size. This shows that with larger dataset sizes, hyper-partitioning is even more useful for quickly finding the relevant portions of the data.

Robust tree can further leverage the data replication in distributed storage systems, e.g., 3× replication in HDFS. Such replication mechanisms first partition the dataset into blocks and then replicate each block multiple times. Instead, first, the entire dataset is replicated, and then each replica is partitioned using a different partitioning tree. While the system is still fault tolerant (because it has the same degree of replication), recovery becomes slower because it needs to read several or all replica blocks in case of a block failure. Essentially, fast recovery time is sacrificed for improved ad hoc query performance. Such a scheme can either increase the number of attributes in the partitioning tree, or increase the data fraction covered per attribute. Both of these lead to improved query performance due to greater partition pruning.

Attribute Allocation

The goal of robust tree is to allocate attributes to nodes in the tree such that all attributes have similar advantage in terms of data skipping or parallel processing. Therefore, the allocation of an attribute is defined as the weighted sum of its fanout on each of the nodes it appears in the partitioning tree T, i.e., the allocation of attribute i is given as:

$$\text{Alloc}_i(T) = \sum_{n \in \text{nodes}(T,i)} \text{DataFraction}_n \cdot \text{Fanout}_n$$

The *allocation* defined above gives the granularity of data partitioning over an attribute. Higher allocation means more data skipping is possible. For example, in Fig. 2b, attribute B appears on two nodes, one covering 50% of the data while the other covering 25% of the data. Thus, B has an allocation of $(0.5*2+0.25*2) = 1.5$. With no query workload, the goal is to balance the benefit of partitioning across all attributes in the dataset. This means that same selectivity predicates on any two attributes X and Y should have similar speedups, compared to scanning the entire dataset. To achieve this, the total allocation is distributed equally among all attributes. Each attribute gets an allocation of $b^{1/|\mathbb{A}|}$, where $|\mathbb{A}|$ is the number of attributes and b is the number of buckets. For instance, if there are eight buckets, and three attributes, the allocation (average fanout) per attribute is $8^{1/3} = 2$. In case of prior workload information, users can provide relative weights of the attributes, and the attribute allocation will be distributed proportional to these weights. The intuition is then to compute the maximum per-attribute allocation and then place attributes into the tree so as to approximate this ideal allocation.

Hyper-partitioning Algorithm

Algorithm 1 shows the pseudocode to generate the robust partitioning tree. It first calculates the depth of the tree to be created (Line 3), then initializes the queue with the root node of the tree (Line 4), and starts a breadth-first traversal to assign an attribute to every node. The attribute to be assigned at a given node is given by the function `LeastAlloc`, which returns the attribute which has the highest allocation remaining. If two or more attributes have the same highest allocation remaining, the algorithm randomly chooses among the ones that have occurred the least number of times in the path from the node to the root. `Med` returns the median of the attribute assigned to this node by finding the median in the sampled data which comes to this branch. The algorithm starts with an allocation of 2 for the root node, since it partitions the entire dataset into two. Each time it goes to the left or the right subtree, it reduces the data it operates on by half. Once an attribute is assigned to a node, it subtracts from the overall allocation of the attribute (Line 13). The algorithm creates a leaf-level bucket in case it reaches the maximum depth (Line 18).

Algorithm 1: `CreateRobustTree`

```
   Input : Int $D$, Int maxPartitionSize, Float[] alloc,
           Tuple[] initSample
1  Tree tree;
2  numBuckets ← $\lfloor D/maxPartitionSize \rfloor$;
3  treeDepth ← $log_2$ (numBuckets) ;
4  Queue queue ← {(tree.root, treeDepth,
   initSample)};
5  while queue.size > 0 do
6      node,depth,sample ← queue.first();
7      if depth = 0 then
8          node ← NewBucket ();
9          Continue;
10     node.attr ← LeastAlloc (alloc) ;
11     node.val ← Med (sample,node.attr) ;
12     lS, rS ← SplitSample (node.attr,
       node.val) ;
13     node.left ← CreateNode () ;
14     node.right ← CreateNode () ;
15     alloc[node.attr] -= $2/2^{\text{maxDepth - depth}}$;
16     depth -=1;
17     queue.add((node.left, depth, lS));
18     queue.add((node.right, depth, rS));
```

Query Processing

A hyper-partitioning query processor considers the filter predicates in incoming queries and filters out partitions that do not match any of the query predicates. For example, if there is a node A_5 in the tree and one of the predicates in the query is $A \leq 4$, then any of the partitions in right subtree of the node don't need to be scanned.

Using Spark, for instance, a job can be constructed where relevant partitions are split into tasks, a set of partitions such that the total size is not more than 4 GB. Each task reads the blocks from HDFS in bulk and iterates over the tuples in main memory. A tuple is returned if it matches the predicates in the query. Tasks are executed independently by the Spark job manager across all machines, and the result is exposed to users as a Spark RDD. Users can use these RDDs to do more analysis using the standard Spark APIs, e.g., run an aggregation.

This section described hyper-partitioning and processing selection queries over a hyper-partitioned input. The following section describes techniques for processing join queries over two or more hyper-partitioned inputs.

Hyper-joins

Hyper-partitioning may end up partially partitioning tables on several different attributes, such that when two tables A and B are joined, a partition in A may join with several partitions in B, each located on HDFS. One option is to simply perform a shuffle join, i.e., repartition both A and B so that each partition of A joins with just one partition of B. However, this can be suboptimal if each partition of A only joins with a few partitions on B; instead, building a hash table over some partitions of A (or B) and probing it with partitions from B (or A) can result in significantly less network and disk I/O.

Example 1 Suppose table A has three partitions and table B has three partitions. Suppose A_1 joins with B_1 and B_2; A_2 joins with B_1, B_2, and B_3; and A_3 joins with B_2 and B_3, and each machine $\mathscr{M}_i$ has memory to hold 2 partitions to build hash tables on A. Consider building a hash table over A_1 and A_3 on $\mathscr{M}_1$; we will need to read B_1, B_2, and B_3. We then build another hash table over A_2 on $\mathscr{M}_2$ and again read B_1, B_2, and B_3. In total, we read six blocks. As an alternative, building a hash table over A_1 and A_2 on $\mathscr{M}_1$ and another one over A_3 on $\mathscr{M}_2$ requires reading just $B_1, 2 * B_2, 2 * B_3 = 5$ blocks.

Thus, building hash tables over different subsets of partitions will result in different costs. Unfortunately, finding the optimal collection of partitions to read is NP-Hard. However, heuristically, solving the problem can still provide significant performance gains over shuffling. To obtain these gains, partitions must be constructed such that, for a join between tables A and B, each partition of A only joins with a subset of the partitions of B. Hyper-join provides this property and is designed to move fewer blocks throughout the cluster than a complete shuffle join when tables are not co-partitioned.

The rest of this section formulates hyper-join as an optimization problem, presents an optimal solution based on mixed integer programming, introduces an approximate algorithm which can run in a much shorter time, discusses hyper-joins for multiple join predicates, and finally shows a two-phase partitioning technique to add join attributes into the robust partitioning tree.

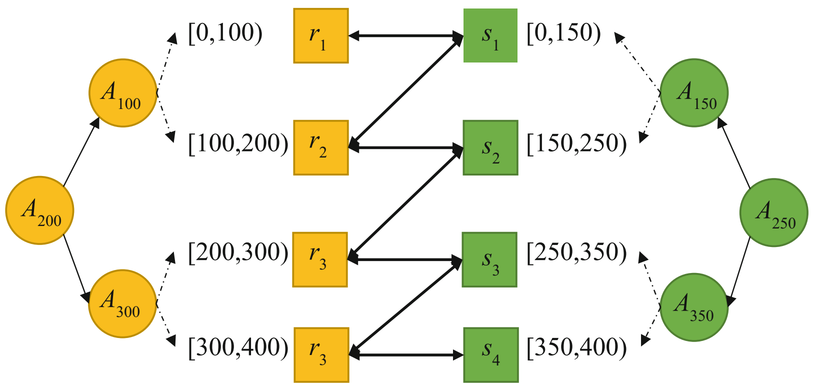
Problem Definition

Consider relations R and S, which can join on attribute t. Let $R = \{r_1, r_2, \ldots, r_n\}$ and $S = \{s_1, s_2, \ldots, s_m\}$ be the collection of data blocks obtained from hyper-partitioning. Let $V = \{v_1, v_2, \ldots, v_n\}$ be a collection of m-dimensional vectors, where each vector corresponds to a data block in relation R. The j-th bit of v_i, denoted by v_{ij}, indicates whether block r_i from relation R overlaps with block s_j from relation S on attribute t (these are the blocks that must be joined with each other). Let $\text{Range}_t(x)$ be a function which gives the range (min and max values) of attribute t in data block x and $\mathbb{1}(s)$ be a function which gives 1 when statement s is true. Given two relations R and S, and for each block r_i from R and s_j from S, let $v_{ij} = \mathbb{1}(\text{Range}_t(r_i) \cap \text{Range}_t(s_j) \neq \emptyset)$. A straightforward algorithm to compute V has a time complexity of $O(nm)$. The Range_t values for each block are stored with each block in the partitioning tree. Let $P = \{p_1, p_2, \ldots, p_k\}$ be a partitioning over R, where P is a set of disjoint subsets of the blocks of R and its union is all blocks in R. Each p_i is constrained to be able to fit into memory of the node performing the join. $\tilde{v}(p_i)$ is used to denote the union vector of all vectors in p_i, i.e., $\tilde{v}(p_i) = \bigvee_{r_j \in p_i} v_j$, where v_j is the vector for block r_j. Let $\delta(v_i) = \sum_{k=1}^{m} v_{ik}$ indicate the number of bits set in v_i. Given a partition p_i, $C(p_i)$ defines the cost of joining p_i with all partitions in S as the number of bits set in $\tilde{v}(p_i)$, i.e., $C(p_i) = \delta(\tilde{v}(p_i))$. This corresponds to the number of blocks to be read to join p_i. Next, the cost function $C(P)$ over a partitioning is defined as the sum of $C(p_i)$ overall p_i in P:

$$C(P) = \sum_{p_i \in P} C(p_i)$$

Thus, the problem of computing hyper join is finding the optimal partitioning P of relation R.

Consider the example in Fig. 3, with table $R = \{r_1, r_2, r_3, r_4\}$ and table $S = \{s_1, s_2, s_3, s_4\}$ and assume $|P| = 2$, i.e., that there is sufficient memory to store $|R|/|P| = 4/2 = 2$ blocks of R in memory at a time. The interval on each partition indicates the minimum and maximum value on the join attribute from all the records. The arrows in the figure indicate the two corresponding partitions overlapping on the join attribute. From the figure, r_1 needs to join with s_1, r_2 needs to join with s_1, s_2, etc. Therefore, $V = \{v_1 = 1000, v_2 = 1100, v_3 = 0110, v_4 = 0011\}$. A hash table could be built over multiple yellow partitions to share some disk access of green partitions. For example, a hash table could be built over the first two yellow blocks (r_1 and r_2) and another one over the last two yellow blocks



Robust Data Partitioning, Fig. 3 Illustrating hyper-join

(r_3 and r_4), so that only 5 green blocks need to be read from disk, assuming only one green block is in memory at a time. In this way, the partition $P = \{p_1 = \{r_1, r_2\}, p_2 = \{r_3, r_4\}\}$, which is optimal. The overall cost $C(P) = 5$, since $\tilde{v}(p_1) = 2$ and $\tilde{v}(p_2) = 3$.

Intuitively, the objective function $C(P)$ is the total number of blocks read from relation S, with some blocks being read multiple times. From the perspective of a real system, the size of p_i is constrained, both due to memory limits and to ensure a minimum degree of parallelism (the number of partitions should be larger than a threshold). If memory is sufficient to hold B blocks from relation R, then we need $c = \lceil n/B \rceil$ partitions. We now define the *minimal partitioning* problem.

Problem 1 Given a set of data blocks from relation R, find a partitioning P over R such that $C(P)$ is minimized, i.e.,

$$\begin{aligned} \underset{P}{\arg\min} \quad & C(P) \\ \text{subject to} \quad & |P| = c, \\ & |p_i| \le B, \forall p_i \in P. \end{aligned}$$

Optimal Algorithm

This section describes a mixed integer programming formulation which can generate the minimal partitioning. Given the maximum number of data blocks B that can be used to build a hash table due to available worker memory, the total number of hash tables to be built are $c = \lceil n/B \rceil$.

For each data block r_i from relation R and each partition p_k, the assignment of r_i to partition p_k is indicated with a binary decision variable $x_{i,k} \in \{0, 1\}$. Likewise, for each data block s_j from relation S, a binary decision variable $y_{j,k} \in \{0, 1\}$ indicates if the j-th bit of $\tilde{v}(p_k)$ is 1.

The first constraint in Problem 1 requires that the size of each partition p_k is under the memory budget B:

$$\forall k, \qquad \sum_{i=1}^{n} x_{i,k} \le B$$

The second constraint requires that each data block r_i from relation R is assigned to exactly one partition:

$$\forall i, \qquad \sum_{k=1}^{c} x_{i,k} = 1$$

Given a partitioning P, for each partition p_k, every overlapping data block from relation S must also be in partition p_k. Let J_k be the set of data blocks from relation R which overlaps with data block s_k from relation S.

$$\forall i, \forall k, \forall j \in J_k, \qquad y_{i,k} \ge x_{i,j}$$

We seek the minimal input size of relation S:

$$\min \sum_{j=1}^{m} \sum_{k=1}^{c} y_{j,k}$$

Solving integer linear programming (ILP) of this form is generally exponential in the number of decision variables; hence the running time of this algorithm may be prohibitive. The proof for NP-hardness of the problem can be found in Lu et al. (2017).

Approximate Solution

Taking B data blocks from relation R with smallest $\delta(\tilde{v}(\mathscr{P}))$ is NP-hard, and there is no algorithm for $n^{1-\epsilon}$-approximation for any constant $\epsilon > 0$. However, an approximate bottom-up algorithm, as shown in Fig. 4, can provide practical runtimes.

The algorithm starts from an empty set of partitions P and an empty partition $\mathscr{P}$. It iteratively adds a data block r_i into $\mathscr{P}$ with smallest $\delta(r_i \vee \tilde{v}(\mathscr{P}))$ until there are B blocks in partition $\mathscr{P}$ or no data block left in relation R. It then adds $\mathscr{P}$ into P until P contains all blocks from relation R. A straightforward implementation of this algorithm has a time complexity of $O(n^2)$ (where n is the number of blocks of R), since the minimum cost block (requiring a scan of the non-placed blocks) needs to be computed n times.

Joins Over Multiple Relations

Hyper-join technique can be extended to multiple inputs. Consider TPC-H query 3. If the join order is (`lineitem` $\bowtie$ `orders`) $\bowtie$ `customer` and the intermediate result of the first two tables is denoted by `tempLO`, then the relation `customer` needs to join with `tempLO` on `custkey`. If `custkey` is the join attribute in the `customer` partitioning tree, only `tempLO` needs to be shuffled based on `custkey`, and then hyper-join can be used instead of an expensive shuffle join, in which both `tempLO` and `customer` need to be shuffled.

With more relations to join, shuffle join over two intermediate outputs of hyper joins could be more efficient. Consider TPC-H query 8. If the join order is ((`lineitem` $\bowtie$ `part`) $\bowtie$ `orders`) $\bowtie$ `customer`, then the intermediate result with relation `lineitem` needs to be shuffled twice. Instead, changing the join order to (`lineitem` $\bowtie$ `part`) $\bowtie$ (`orders` $\bowtie$ `customer`) can use hyper-join twice and a shuffle join over the intermediate results.

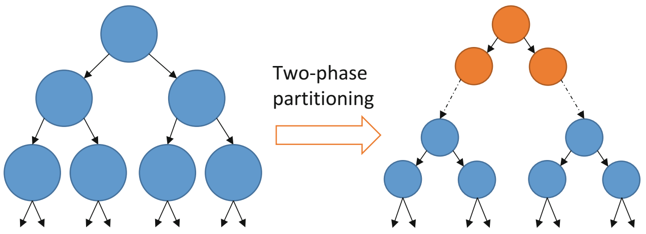
Two-Phase Hyper-partitioning

Hyper-join leverages hyper-partitioning; however, the robust partitioning tree described so far partitions data based solely on the selection predicates. Thus, it's unlikely to have the join attribute in very many nodes in the tree, and it's highly possible that every partition will overlap with a large number of partitions. Two-phase partitioning tackles this challenge by injecting the join attributes into the partitioning tree, as depicted in Fig. 5. The first phase splits on join attributes (shown in orange), while the second phase splits on selection attributes (shown in blue). During the first phase, median values of the join attributes are used to recursively split the dataset into two. During the second phase, the join partitions are further partitioned on selection attributes using the standard hyper-partitioning.

Consider the left partitioning tree in Fig. 3 as an example. There are two levels in the tree which are reserved for the join attribute, which, assuming data is uniformly distributed in the range [0, 400], leads to four disjoint partitions with range [0, 100), [100, 200), [200, 300), and [300, 400). The same procedure is also applied to the right partitioning tree, which creates four disjoint partitions with range [0, 150), [150, 250), [250, 350), and [350, 400).

Robust Data Partitioning, Fig. 4 A bottom-up approximate solution

```
$R \leftarrow \{r_1, r_2, \ldots, r_n\}$, $P \leftarrow \emptyset$, $\mathscr{P} \leftarrow \emptyset$
while $R$ is not empty:
    merge $\mathscr{P}$ with data block $r_i$ with smallest $\delta(r_i \vee \tilde{v}(\mathscr{P}))$
    if $|\mathscr{P}| = B$ or $r_i$ is the last one in $R$:
        add $\mathscr{P}$ to $P$ and $\mathscr{P} \leftarrow \emptyset$
    remove data block $r_i$ from $R$
return $P$
```



Robust Data Partitioning, Fig. 5 Illustrating two-phase partitioning

Robust Repartitioning

Hyper-partitioning and hyper-join allow an analyst to quickly get started with her ad hoc queries. However, the analyst also wants the partitioning to adapt as her analysis progresses, e.g., drilling down web click data into successively smaller age groups, to provide even better query performance. *Robust repartitioning* provides the mechanisms to achieve this. When a query is submitted, a repartitioning optimizer explores alternative partitioning trees to find the best one and decides whether repartitioning is worthwhile. The optimized plan only accesses data which is to be read by input queries, i.e., data that is not read by queries during repartitioning is not accessed. This has two benefits: (i) data that is not touched by any query is never repartitioned and (ii) query processing and repartitioning share scans reducing the cost of repartitioning.

The rest of this section describes the cost model used, introduces three basic transformations used to transform a given partitioning tree, describes a divide-and-conquer approach to consider all possible alternatives generated from the transformation rules for inserting a single predicate, discusses how to handle multi-predicate queries, and lastly shows a smooth repartitioning technique to adapt to changing join predicates. It is worth noting that the entire optimization process is transparent to users, i.e., users do not have to worry about making repartitioning decisions and their queries remain unchanged with the new access methods.

Cost Model

Consider a window (W) of queries that happened in the past X hours. X is a parameter in adaptive query executor, and it determines how quickly the system reacts to workload changes. For each query q in the query sequence, the cost of processing q using partitioning tree T is given as:

$$\mathrm{Cost}(T,q) = \sum_{b \in \mathrm{lookup}(T,q)} n_b$$

where $\mathrm{lookup}(T,q)$ returns the set of relevant buckets for query q in T and n_b is the number of tuples in bucket b. The cost of the query window is the sum of the cost of individual queries. For a query being executed, the optimizer might want to transform the partitioning tree to a new partitioning tree T' resulting in a set of buckets $B \subset \mathrm{lookup}(T,q)$ being repartitioned. The benefit of this transformation is

$$\mathrm{Benefit}(T') = \sum_{q \in W} \mathrm{Cost}(T,q) - \sum_{q \in W} \mathrm{Cost}(T',q)$$

and the added cost of repartitioning is given as

$$\mathrm{RepartitioningCost}(T,q) = c \sum_{b \in B} n_b$$

where c is the write multiplier, i.e., how expensive writes are compared to a read. Repartitioning is expensive; however, it only happens when the resulting decrease in the cost of the query window (benefit) is greater than the repartitioning cost. This check prevents constant re-paritioning due to a random query sequence and bounds the worst case impact. To illustrate, consider a single node in the tree and a query sequence of the form $\sigma_{A<2}, \sigma_{B<2}, \sigma_{A<2}, \sigma_{B<2} \ldots$. In this case, the data is not constantly repartitioned. After doing it once, say on A, the total cost goes down, and hence the repartitioning on B would not happen as Benefit $<$ Repartitioning-Cost.

Tree Transformations

A set of transformation rules allow exploring the space of possible plans when repartitioning the data. Consider a query predicate of the form $A \leq p$, denoted as A_p. Only partitioning transformations that are local, i.e., that do not involve rewriting the entire tree, are considered. These local transformations are cheaper and amortize the repartitioning effort over several queries. The three basic transformations are discussed below.

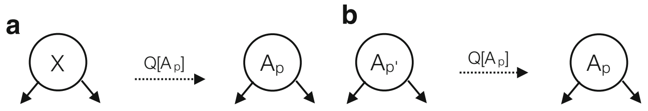
(1) Swap. Replaces an existing node in the partitioning with the incoming query predicate A_p. As only the accessed data is repartitioned, we consider swapping only those nodes whose left *and* right children are fully accessed by the incoming query. Applying swap on an existing node involves reading both sub-branches, and restructuring all partitions beneath the left subtree to contain data satisfying A_p and the right subtree to contain data that does not satisfy A_p. Swaps can happen between different attributes (Fig. 6a), in which case both branches are completely rewritten in the new tree. Swaps can also happen between two predicates of the same attribute (Fig. 6b), in which case the data moves from one branch to the other. For example, in the Fig. 6b, if node $A_{p'}$ is A_{10} and predicate A_p is $A \leq 5$, then data moves from the left branch to the right branch, i.e., the left branch is completely rewritten while the right branch just has new data appended.

Swaps serve the dual purpose of un-partitioning an existing (less accessed) attribute while refining on another (more accessed) attribute. As both the swap attributes as well as their predicates are driven by the incoming queries, they reduce the access times for the incoming query predicates. Finally, note that it is cheaper to apply swaps at lower levels in the partitioning tree because less data is rewritten. Applying them at higher levels results in a much higher cost.

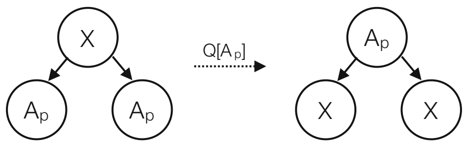
(2) Push-up. Pushes a predicate as high up the tree as possible. This can be done when both the left and the right child of a node contain the incoming predicate, as a result of a previous swap, as shown in Fig. 7. This is a logical partitioning tree transformation, i.e., it only rearranges the internal nodes without any modification to the leaf nodes.

A push-up transformation is checked every time a swap transformation is performed. The idea is to move important predicates (ones that have recently or frequently appeared in the query sequence) progressively up the partitioning tree, from the leaves right up to the root. This makes important predicates less likely to be swapped immediately, because swapping a node higher in the partitioning tree is much more expensive. Another advantage of push-up is that it causes a churn of the attributes assigned to higher nodes in the upfront partitioning. When such a dormant node is pushed down, subsequent predicates can swap them in an incremental fashion, affecting fewer branches, thus making the tree transformations more robust.

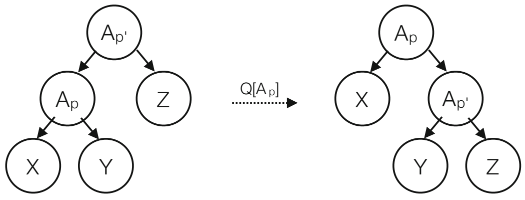
(3) Rotate. Transformation rearranges two predicates on the same attribute such that more important (recently accessed or frequently appearing in the query sequence) predicate appears higher up in the partitioning tree. Figure 8 shows a rotate transformation involving predicates p and p' on attribute A. The goal here is to churn the partitioning tree such that predicates on less important attributes are more likely to be replaced first. Similar to the push-up transformation, rotate is a logical transformation, i.e., it only rearranges the internal nodes of the parti-



Robust Data Partitioning, Fig. 6 Node swap in the partitioning tree (**a**) Different attribute. (**b**) Same attribute



Robust Data Partitioning, Fig. 7 Node pushdown in partitioning tree



Robust Data Partitioning, Fig. 8 Node rotation in partitioning tree

tioning tree and it is always performed wherever possible.

Above three partitioning tree transformations can be combined to capture a fairly general set of repartitioning scenarios. Figure 9 shows an example, where first nodes D_4 are swapped with incoming predicate A_2 at the lower level, then A_2 is pushed up one level above, and finally it is rotated with nodes A_5 and C_3. In the process, only half the leaves are repartitioned. Thus, in larger trees, repartitioning mostly happens on small fractions of the data modifying a few subtrees locally.

Divide-and-Conquer Repartitioning

Given a query with predicate A_p and a partitioning tree T, there are many different combinations of transformations that need to be considered. However, observe that the data access costs over a subtree T_n, rooted at node n, could be broken down into the access costs over its subtrees, i.e.,

$$\text{Cost}(T_n, q_i) = \text{Cost}(T_{n_{left}}, q_i) + \text{Cost}(T_{n_{right}}, q_i)$$

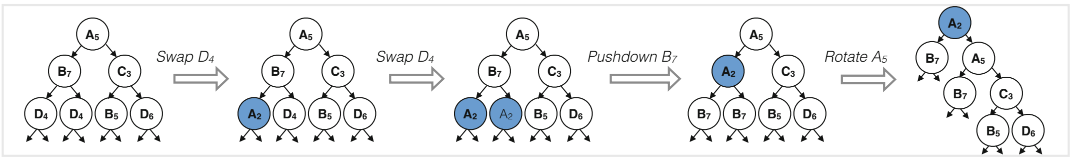
where $T_{n_{left}}$ and $T_{n_{right}}$ are subtrees rooted respectively at the left and the right child of n. Thus, finding the best partitioning tree can be broken down into recursively finding the best left and right subtrees at each level and considering parent node transformations only on top of the best child subtrees. For each transformation, the benefit and cost of that transformation is considered, and the one which has the best benefit-to-cost ratio is picked. Table 1 shows the cost and benefit estimates for the different transformations. For the swap transformation, denoted as $P_{\text{swap}}(n, n')$, the query costs are recalculated. However, push-up and rotate transformations, denoted as $P_{\text{swap}}(n, n')$ and $P_{\text{pushup}}(n, n_{\text{left}}, n_{\text{right}})$, respectively, inherit the costs from children subtrees. Applying none of the transformations at a given node is denoted as $P_{\text{none}}(n)$. This approach helps to significantly reduce the candidate set of modified partitioning trees.

Above divide-and-conquer algorithm has a complexity of $O(QN\log N)$, where N is the number of nodes in the tree and Q is the number of queries in the query window. More details on the algorithm can be found in Shanbhag et al. (2017).

Repartitioning with Multiple Predicates

A predicate of the form $A \leq p$ gets inserted in the tree as A_p, and on insertion, only the leaf nodes on the left side of the node are accessed. $A > p$ is also inserted as A_p with the right side of the node being accessed. For $A \geq p$ and $A < p$, let p' be $p - \delta$ where δ is the smallest change for p's data type. We insert $A_{p'}$ into the tree. $A = p$ is treated as combination of $A \leq p$ and $A > p'$.

Now consider a query with two predicates A_p and A_{p2}. The brute force approach is to consider choosing a set of accessed nonterminal nodes to be replaced by A_p, and then for every such choice, choose a set of remaining nodes to be replaced by A_{p2}. Thus, the number of choices grows exponentially with the number of predicates. A greedy approach is to try to insert each predicate in the query into the partitioning tree. The best among the best plans obtained for different predicates is picked, and the corresponding predicate is removed from the predicate set. Likewise, the remaining predicates are inserted into the best plan obtained so far. The algorithm stops when either all predicates have been inserted or when the tree stops changing. Doing this adds a multiplicative complexity of $O(|P|^2)$ where P is the set of query predicates.



Robust Data Partitioning, Fig. 9 Introducing predicate A_2 into the partitioning tree

Robust Data Partitioning, Table 1 The cost and benefit estimates for different partitioning tree transformations

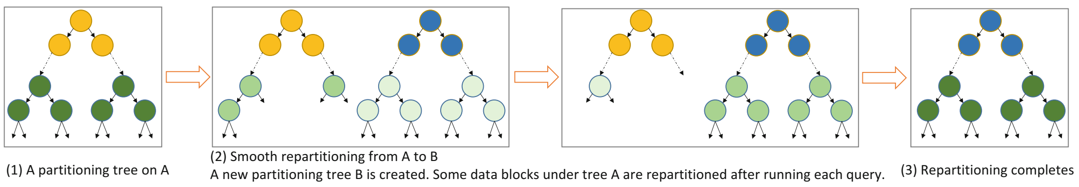
Transformation	Notation	Cost (C)	Benefit (B)
Swap	$P_{\text{swap}}(n, n')$	$\sum_{b \in T_n} c \cdot n_b$	$\sum_{i-0}^{k}[\text{Cost}(T_n, q_i) - \text{Cost}(T_{n'}, q_i)]$
Pushup	$P_{\text{pushup}}(n, n_{\text{left}}, n_{\text{right}})$	$C(P(n_{\text{left}})) + C(P(n_{\text{right}}))$	$B(P(n_{\text{left}})) + B(P(n_{\text{right}}))$
Rotate	$P_{\text{rotate}}(p, p')$	$C(P(n_{\text{left} \mid \text{right}}))$, for p' on $n_{\text{left} \mid \text{right}}$	$B(P(n_{\text{left} \mid \text{right}}))$, for p' on $n_{\text{left} \mid \text{right}}$
None	$P_{\text{none}}(n)$	$C(P(n_{\text{left}})) + C(P(n_{\text{right}}))$	$B(P(n_{\text{left}})) + B(P(n_{\text{right}}))$

Smooth Repartitioning

A key limitation of the repartitioning techniques presented so far is that they do not adapt in response to join queries. Instead, each table adapts independently, and tables end up being partitioned on different attributes and ranges, such that hyper-join would not provide a performance advantage over shuffle joins. After the initial two-phase partitioning, with new incoming queries containing a new join attribute, the partitioning tree should also shift to the new join attribute. However, repartitioning all of the data immediately would introduce a potentially very long delay and, when the workload is periodic, could lead to oscillatory behavior where it switches from one partitioning to another. Furthermore, a table with multiple foreign keys may join with multiple tables. For example, in TPC-H, queries join `lineitem` and `orders` on order_key, and `lineitem` and `supplier` join on supplier_key. Smooth repartitioning addresses these challenges by maintaining multiple partitioning trees, building each when a new popular join attribute is seen, and migrating blocks between them. The key goal is to adapt partitioning trees in a way that facilitates joins while still maintaining the performance advantages of partitioning for selection queries.

Smooth partitioning creates a new partitioning (initially empty) tree, when it observes a query with a new join attribute. The new tree's join attribute comes from the new query, and its predicates are used to build the lower levels of the tree. Smooth repartitioning also repartitions $1/|W|$ of the dataset from the old tree to the new tree, where $|W|$ is the length of the query window. This is accomplished by randomly choosing $1/|W|$ of the blocks in the old tree and inserting them into the new tree (because files are only appended in HDFS, it is possible to do this without affecting the correctness of any concurrent queries). To avoid doing repartitioning work when rare queries arrive, smooth repartitioning can be configured to wait to create a new partitioning tree until the query window contains some minimum frequency f_{min} of queries for a new join attribute; in this case once the tree is created, $f_{\text{min}}/|W|$ of the blocks will be moved.

As more queries arrive with the new join attribute, smooth repartitioning repartitions more data into the new partitioning tree using the following algorithm. It first calculates the percentage of two types of queries in the query window and the data in each of the partitioning trees. If the incoming query's join attribute is the same as the newly created partitioning tree and the fraction of data in the new partitioning tree is less than the fraction of its type in the query window, data from the old partitioning tree is moved to the new one, again by randomly selecting blocks and moving them.


(1) A partitioning tree on A
(2) Smooth repartitioning from A to B
A new partitioning tree B is created. Some data blocks under tree A are repartitioned after running each query.
(3) Repartitioning completes

Robust Data Partitioning, Fig. 10 Illustrating smooth repartitioning

Consider the example in Fig. 10. The algorithm starts from a partitioning tree optimized for join attribute A. When a query with new join attribute B comes, a new partitioning tree for B is created with two-phase partitioning and repartitions $1/|W|$ of the dataset from the old partitioning tree. The color of nodes from the lower levels of the partitioning trees indicates the size of data. The darker the color is, the larger the size of data is. After the new tree is created, both the partitioning trees are maintained with different join attributes. As more queries with join attribute B appear in the query window, more data from the old partitioning tree is repartitioned to the new one. The above procedure is iterated until the query window only includes queries with join attribute B. After the dataset finishes repartitioning, the old partitioning tree for join attribute A is removed, and only the partitioning tree for join attribute B is maintained, which is depicted by the last sub-figure in Fig. 10. (Of course, in many applications, there will not be a complete shift from one join to another, in which case multiple trees will be preserved.)

Conclusion

This chapter described new advancements in data partitioning for modern applications that are ad hoc in nature and do not have any upfront query workload. The key ideas presented include the notion of robustness, the concept of hyper-partitioning for creating a robust partitioning tree without upfront query workload, a hyper-join technique to efficiently process join queries over hyper-partitioned data, and a set of robust repartitioning techniques to steadily adapt the partitioning tree to changes in the workload.

Robust data partitioning revisits the design of a database in the face of modern ad hoc query workloads, recalibrating the database systems to the expectations of modern users – good performance from the first query itself and adaptively improving from there on.

Cross-References

- ▶ Big Data Indexing
- ▶ Hadoop
- ▶ Parallel Join Algorithms in MapReduce

References

Agrawal S, Narasayya V, Yang B (2004) Integrating vertical and horizontal partitioning into automated physical database design. In: SIGMOD

Aly AM, Mahmood AR, Hassan MS, Aref WG, Ouzzani M, Elmeleegy H, Qadah T (2015) AQWA: adaptive query workload aware partitioning of big spatial data. PVLDB 8:2062–2073

Ananthanarayanan G, Ghodsi A, Shenker S, Stoica I (2011) Disk-locality in datacenter computing considered irrelevant. In: HotOS

Bentley JL (1975) Multidimensional binary search trees used for associative searching. Commun ACM 18(9):509–517

Bhatotia P, Wieder A, Rodrigues R, Acar UA, Pasquin R (2011) Incoop: MapReduce for incremental computations. In: SoCC

Binnig C, Crotty A, Galakatos A, Kraska T, Zamanian E (2016) The end of slow networks: it's time for a redesign. PVLDB 9:528–539

Cudré-Mauroux P, Wu E, Madden S (2010) TrajStore: an adaptive storage system for very large trajectory data sets. In: ICDE

Curino C, Jones E, Zhang Y, Madden S (2010) Schism: a workload-driven approach to database replication and partitioning. PVLDB 3:48–57

Dittrich J, Quiané-Ruiz JA, Jindal A, Kargin Y, Setty V, Schad J (2010) Hadoop++: making a yellow elephant

run like a cheetah (without it even noticing). PVLDB 3:518–529
Eldawy A, Mokbel MF (2015) Spatialhadoop: a mapreduce framework for spatial data. In: ICDE
Eltabakh MY, Tian Y, Özcan F, Gemulla R, Krettek A, McPherson J (2011) CoHadoop: flexible data placement and its exploitation in hadoop. PVLDB 4: 575–585
Ghandeharizadeh S, DeWitt DJ (1994) MAGIC: a multiattribute declustering mechanism for multiprocessor database machines. IEEE Trans Parallel Distrib Syst 5:509–524
Ghemawat S, Gobioff H, Leung ST (2003) The google file system. SIGOPS Oper Syst Rev 37(5):29–43
Graefe G (2003) Sorting and indexing with partitioned B-trees. In: CIDR
Idreos S, Kersten M, Manegold S (2007) Database cracking. In: CIDR
Idreos S, Kersten M, Manegold S (2009) Self-organizing tuple reconstruction in column-stores. In: SIGMOD
Lu Y, Shanbhag A, Jindal A, Madden S (2017) AdaptDB: adaptive partitioning for distributed joins. PVLDB 10:589–600
Nehme R, Bruno N (2011) Automated partitioning design in parallel database systems. In: SIGMOD
Nishimura S, Das S, Agrawal D, El Abbadi A (2011) MD-HBase: a scalable multi-dimensional data infrastructure for location aware services. In: MDM
Pavlo A, Curino C, Zdonik S (2012) Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In: SIGMOD
Quamar A, Kumar KA, Deshpande A (2013) SWORD: scalable workload-aware data placement for transactional workloads. In: EDBT
Shanbhag A, Jindal A, Madden S, Quiané-Ruiz J, Elmore AJ (2017) A robust partitioning scheme for ad-hoc query workloads. In: SOCC, pp 229–241
Sleator DD, Tarjan RE (1985) Self-adjusting binary search trees. J ACM 32:652–686
Sun L, Franklin MJ, Krishnan S, Xin RS (2014) Fine-grained partitioning for aggressive data skipping. In: SIGMOD
Wang J, Wu S, Gao H, Li J, Ooi BC (2010) Indexing multi-dimensional data in a cloud system. In: SIGMOD
Zhou J, Bruno N, Wu MC, Larson PA, Chaiken R, Shakib D (2012) SCOPE: parallel databases meet MapReduce. PVLDB 21:611–636
Zilio DC, Rao J, Lightstone S, Lohman G, Storm A, Garcia-Arellano C, Fadden S (2004) DB2 design advisor: integrated automatic physical database design. In: PVLDB

Role-Based Access Control (RBAC)

▶ Security and Privacy in Big Data Environment

Routes

▶ Spatiotemporal Data: Trajectories

Rule Mining

▶ Decision Discovery in Business Processes

Rule-Based Processing

▶ Reasoning at Scale

Rule-Oriented Process Mining

▶ Declarative Process Mining