D|ONE
WE MAKE SENSE.

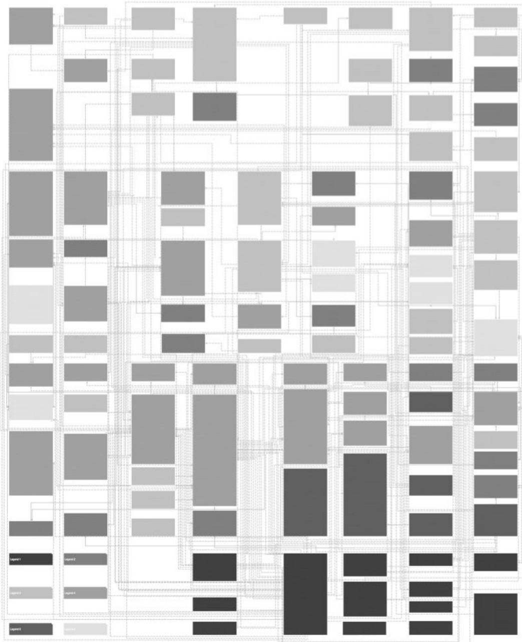
Topological sorting and the ETL process

A D|ONE insight brought to you by Joonas Asikainen

Abstract

Consider the data warehouse of Figure 1. It consists of a hundred tables with over two hundred relations. The question any ETL developer will ask is, in which order do I load the tables?

Figure 1: Big data warehouse.



In this paper, we point out that a typical ETL process to populate a database can be thought of as an acyclic directed graph and that existing graph algorithms can thus be used to resolve the order of processing the tables in the load process. While simple databases can easily be managed manually, such methods prove very useful with big and complex data warehouses. We extend an existing sorting algorithm to provide with information on which tables can be loaded in parallel.

Introduction

The ETL (extract-transform-load) process to populate a DWH (data warehouse) implemented as a dimensional model has typically a clear constraint that tables need to be loaded in a specific order. In a dimensional model [1] tables are classified as facts (measures) and dimensions (context) and their relationships are enforced by foreign keys. This implies that, for instance, any dimension table referenced by a fact table must be loaded (inserted into) before the said fact table can be loaded to avoid foreign key violations.

While for small data warehouses and simple star models [2] this order of dependencies is trivial and easy to maintain, in snowflake [4] or other dimensional models and 3NF models (third normal form) this becomes an increasingly complex task, especially when multiple interconnected fact tables are present. In this paper, we point out the analogy to the well known problem of topological sorting [4] in graph theory and apply the Kahn algorithm [5] to figure out the ordering of the ETL process for a sample DWH. We also append the Python code which implements the sorting. We extend the algorithm to output the level in the hierarchy of nodes within the graph as dictated by the edges. This information can be used to optimally parallelize the load process.

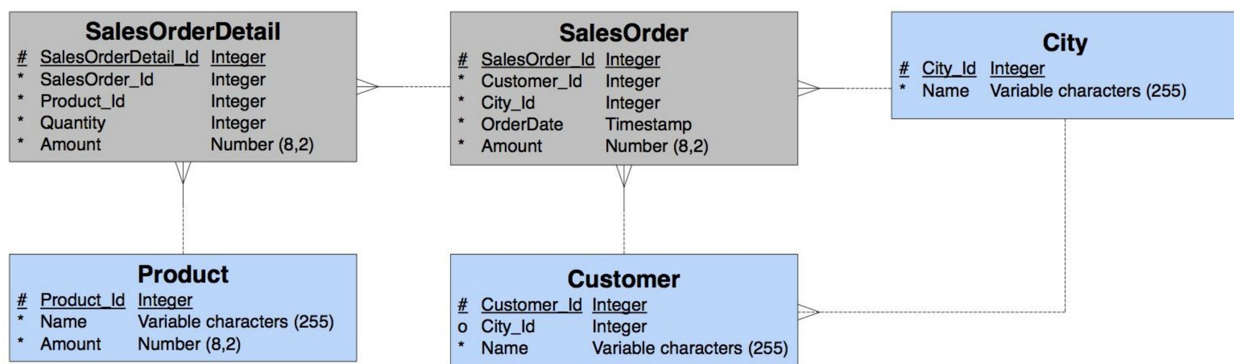
Topological sorting

In the field of computer science, a **topological sort** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG) [4].

Data Warehouse

We illustrate the use of topological sorting by a sample data warehouse shown in Figure 2. It consists of five tables which are connected through five foreign key relations.

Figure 2: Sales database - fact tables are gray whereas dimension tables are light blue.



The relations defined in the data model diagram above define edges in a graph consisting of nodes represented by the tables.

ETL & Kahn

In order to resolve the order in which the tables in the sample DWH of Figure 1 need to be loaded, we apply the Kahn algorithm [5] to the underlying directed graph. Kahn algorithm works as follows. Let *L* be a list of nodes that will be by the completion of the algorithm in the correct topological order. Let *S* be a list of nodes that are being processed and which is filled and emptied during the process [4, 5]. Further, let *nxt* be the hierarchy level of the next node to be inserted in the sorted list *L* and *cur* be the hierarchy level of the current node being processed.

```
nxt ← 0
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges with level nxt
while S is non-empty do
   remove a node [n,cur] from S
   if cur equals nxt then
     increment nxt
   add [n,cur] to tail of L
   for each node m with an edge e from n to m do
      remove edge e from the graph
      if m has no other incoming edges then
         insert [m,nxt] into S
if graph has edges then
   return error (graph has at least one cycle)
else
   return L (a topologically sorted order)
```

For details of our implementation, see the Python code in the Appendix.

We saved two text files nodes.txt and edges.txt corresponding to the edges and vertices defined by the tables and relations in the sample sales database and ran our Python code on the graph thus defined. Note that lines beginning with a hash "#" are treated as comments.

edges.txt

```
# NodeTo     NodeFrom
City  Customer
City  SalesOrder
Customer     SalesOrder
SalesOrder   SalesOrderDetail
Product      SalesOrderDetail
```

nodes.txt

```
# Node
SalesOrder
Customer
City
SalesOrderDetail
Product
```

We executed the Python code by invoking the following command on a shell window in a folder containing the input files as well the source code:

```
python toposort.py nodes.txt edges.txt
```

The resulting topologically sorted list of tables is shown in the table below. The column “level” indicates the level in the hierarchy within the graph dictated by the edges.

Level	Table
0	Product
0	City
1	Customer
2	SalesOrder
3	SalesOrderDetail

Note that the (topological) order is not unique, as, e.g., tables City and Product are interchangeable (neither table has no incoming connections and thus they can be independently loaded). From the perspective of the ETL load process, the loading (insert into) order of the tables is from top to bottom. This ensures that the foreign key constraints are not violated. Further, the tables sharing the value of column “level” can be loaded in parallel (tables Product and City in this case).

References

1. Dimensional modeling of a Data Warehouse
2. Star schema
3. Snowflake schema
4. Topological sorting
5. Kahn, Arthur B. (1962), "Topological sorting of large networks", *Communications of the ACM* **5** (11): 558–562, doi:10.1145/368996.369025.

Appendix - source code

[Source code highlighting by http://markup.su/highlighter/]

```
#!/usr/bin/env python
# Date:        2015-06-09
# Author:      Joonas Asikainen
# Description: Kahn algorithm - topological sorting
# Usage:       python toposort.py node_file_name edge_file_name
# Sample:      Sample input data (tab-separated files edges.txt, nodes.txt)
#              below.

# edges.txt:
"""
# NodeTo    NodeFrom
City    Customer
City    SalesOrder
Customer      SalesOrder
SalesOrder    SalesOrderDetail
Product       SalesOrderDetail

"""
# nodes.txt:
"""
# Node
SalesOrder
Customer
City
SalesOrderDetail
Product
"""

#####
import sys
import math
import numpy
import time

### Kahn algorithm - enhanced with level info (e.g., parallel loading group). ###
def getTopologicalSorting(matrix) :
    nxt = 0

    slist = []
    for i in range(size) :
        refs = matrix[i].sum()
        if (refs == 0) :
            slist.append([i,nxt])
    llist = []

    # topological sort algorithm
    nxt = nxt + 1
    while (slist) :
        [nn, cur] = slist.pop()
```

```
        if (nxt == cur) :
            nxt = nxt + 1
        llist.append([nn,cur])
        for mm in range(size) :
            # remove edge
            if (matrix[mm][nn] == 1) :
                matrix[mm][nn] = 0
                refs = matrix[mm].sum()
                if (refs == 0) :
                    slist.insert(0, [mm, nxt])
    # done
    return llist

### main ###
if __name__ == '__main__':

    # check args
    args = sys.argv[1:]
    nargs = len(args)
    if (nargs < 2) :
        print '-- usage: toposort.py node_file_name edge_file_name'
        sys.exit()

    # info
    print '-- toposort.py:', args

    # timing
    start = int(round(time.time() * 1000))

    # extract nodes
    fn = args[0]
    f = open(fn, 'r')
    nodes = []
    for line in f :
        if (not line.startswith('#')) :
            line = line.strip()
            nodes.append(line)
    f.close()
    nodes = sorted(nodes)

    # system size and the edge matrix
    size = len(nodes)
    matrix = numpy.zeros([size, size], dtype=numpy.int)

    # extract edges
    fn = args[1]
    f = open(fn, 'r')
    edges = []
    for line in f :
        if (not line.startswith('#')) :
            # edge
            edge = line.strip().split('\t')
            # exclude self-references
```

```
            if (not edge[0] == edge[1]) :
                edges.append(edge)
    f.close()
    edges = sorted(edges)

    # prepare the edge matrix
    for edge in edges :
        efrom = nodes.index(edge[1])
        eto = nodes.index(edge[0])
        matrix[efrom][eto] = 1

    # timing
    print '-- number of nodes =', len(nodes)
    print '-- number of edges =',len(edges)
    end = int(round(time.time() * 1000))
    print '-- read time =', (end-start), 'ms.'

    # sorting topologically
    start = int(round(time.time() * 1000))
    srtd = getTopologicalSorting(matrix)
    end = int(round(time.time() * 1000))
    print '-- computation time =', (end-start), 'ms.'

    # done sorting
    cnt = matrix.sum()
    if (cnt > 0) :
        print '-- not an ADG (asyclic directed graph)'
        print '-- number of edges remaining = ', cnt
    else :
        print '-- level vs node (topologically sorted):'
        for [node, lvl] in srtd :
            print lvl, nodes[node]
```