Representative Sample Extraction from Web Data Streams*

Michael Scriney[1], Congcong Xing[1], Andrew McCarren[2], and Mark Roantree[1]

[1] VistaMilk Research Centre, School of Computing, Dublin City University, Ireland
[2] Insight Centre for Data Analytics, School of Computing, Dublin City University, Ireland
michael.scriney@dcu.ie, congcongxing2@mail.dcu.ie, Andrew.McCarren@dcu.ie, mark.roantree@dcu.ie

Abstract. Smart or digital city infrastructures facilitate both decision support and strategic planning with applications such as government services, healthcare, transport and traffic management. Generally, each service generates multiple data streams using different data models and structures. Thus, any form of analysis requires some form of extract-transform-load process normally associated with data warehousing to ensure proper cleaning and integration of heterogeneous datasets. In addition, data produced by these systems may be generated at a rate which cannot be captured completely using standard computing resources. In this paper, we present an ETL system for transport data coupled with a smart data acquisition methodology to extract a subset of data suitable for analysis.

1 Introduction

Data from the digital city can be generated from a wide variety of sources across many services such as housing, healthcare, transport, the environment etc. Similar to traditional web data, this data is available on-line, typically in either XML, JSON or CSV format, meaning it must be processed in some form before it can be properly used. These processing tasks include acquisition and interpretation, transformation, integration, and analysis or machine learning of data from sensors, devices, vehicles etc. There have been a number of approaches to building smart city applications [5] and to cluster or integrate query graphs on smart city data [10, 12]. Decision makers who use smart city applications require OLAP type services to generate the datasets from a warehouse upon which they make their decision. OnLine Analytical Processing (OLAP) queries offer the richest form of data extraction with dimensional data providing powerful dimensional queries. The challenge for smart city researchers lies in incorporating web, sensor and streaming data to these datasets usually in RDF format [2]. Sources are often new, not always available, are not suited to integration, and use heterogeneous data models.

* This research is supported by Science Foundation Ireland (SFI) and the Department of Agriculture, Food and Marine on behalf of the Government of Ireland under Grant Number 16/RC/3835.

Motivation. DublinBus [4] provides a range of online services related to the running of their bus network for the city of Dublin. One of these services provides a real-time location for all buses on the network. Occasionally, times are provided for buses that do not exist. While these can be detected using an analysis of the real-time data streams, there is no indication as to why buses may appear and disappear from the system. In order to facilitate a deeper analysis, it requires a methodology to acquire and transform data into a usable format, and at sufficiently regular intervals so as not to lose any real-time information. This poses two problems. Firstly, the system requires considerable resources to ensure that all of the data, for the entire network of bus journeys is extracted. Secondly, sufficiently flexible OLAP functionality is required to manipulate datasets for the types of analyses required.

Contribution. In this research, we present a framework for manipulated transport data streams so as to create datasets for machine learning algorithms. Our data model represents one subset of a larger smart city application where our bus transport data can be integrated with other smart city datasets using selected attributes from time and geo dimensions. Our contribution is to tackle the problem of big data by using an algorithm to extract smaller representative samples. We also provide an extended OLAP which uses keyword extensions to present data so as to meet specific requirements for machine learning algorithms. A longer version of this paper can be found at [13].

Paper Structure. The paper is structured as follows: in §2, we provide a discussion on state of the art; in §3, we provide a description of our Extract-Transform-Load architecture which manages source transport data into our multidimensional model; in §4, we present an algorithm to optimise the acquisition of data; in §5, we present our evaluation and discussion; and finally in §6, we conclude the paper.

2 Related Research

In [11], the authors present a system to construct a data warehouse from user requirements. The system is given a domain ontology which is used to derive facts which are subsequently presented to the user. Once the user selects their desired fact, the data is extracted and presented to them. Similar to our approach, the authors attempt to identify facts without user interaction. However, the authors' approach requires a domain ontology to discover these facts while ours is static. In addition, our system provides several OLAP extensions to provide datasets in an analysis-ready format. In [1], the authors present an automatic ETL system. The authors use an ontology to provide semantic data integration, where that ontology is derived from a data warehouse schema and a lexicon. Similar to our approach, the authors then use clustering to determine the similarity between data sources while we use similarity matrices in order to determine the best possible subset of data to be extracted. The authors in [9] present an ETL approach which uses domain-specific modelling. Their own language (DSL) is used to model the different steps in the ETL process. A domain expert using this language designs the ETL framework which is subsequently deployed. Domain modelling is used to describe the data sources and these sources are then linked

using an ontology. We also use an ETL process to extract data, however we also have a pre-processing step before data acquisition to determine a suitable subset of the data which can be acquired. In [8], the authors use RDF and OWL ontologies to create integrated data marts. Each data source in the mart is provided with an ontology which is used to construct an RDF version of the data. From this, an OWL definition of the mart uses RDF queries to extract the required data from each source. In contrast, our approach uses the original source format to extract dimensional data. Our system provides integration dimensions for integrated data marts. In addition, we provide OLAP extensions to convert the data when being queried from the data warehouse.In [3] the authors present a big data architecture for smart city data. The system is composed of a series of layers, starting from the data source to a presentation layer providing analysis applications. Similar to our approach the authors use transport as a use case and a custom ETL workflow to deliver a k-means analysis, while our approach presents a generic model for transport data coupled with a series of transport specific OLAP extensions. Finally, the authors in [7] present a smart city platform which allows a user to search and integrate smart city data to suit their end requirements. A user selects the data source(s) they wish to integrate and the data undergoes an ETL process to convert all source data into RDF triplets. The authors use RDF as their common data model while our approach uses a data model of our own design. Furthermore, our system strives to provide data which is suitable for analysis.
The state of the art in ETL and data warehousing shows that some form of common data model either on a data source layer or globally is required to provide holistic integration. We adopt this approach through the use of our data model. However, in limiting ourselves to a particular domain (i.e. transport) we are able to provide a set of domain specific operations to facilitate OLAP analysis.

3 Data Acquisition and Transformation

In this section, we begin with a description of the Extract-Transform-Load (ETL) process to acquire and transform raw data. We then describe our multi-dimensional (cube) data model, and provide a description of the data acquisition and transformation process.

3.1 System Architecture

In figure 1, we show the system workflow as a series of layers, each containing data in different formats, where layers are connected by data transformation processes. The *Source Data* layer is generally a web service providing an API to facilitate JSON or XML data extraction. In this instance, the DublinBus Real Time Passenger Information (RTPI) API generates XML elements based on bus routes or stops. The Data Acquisition process is effectively a purpose built wrapper to the *Data Staging Layer*, which manages JSON, XML or relational data. The Data Transformation processor is a generic process converting one of 3 data types to the system data model (the predefined smart city data mart defined in section 3.2). Our customised OLAP (OnLine Analytical Processing) API described in section 4, generates the datasets for analyses.

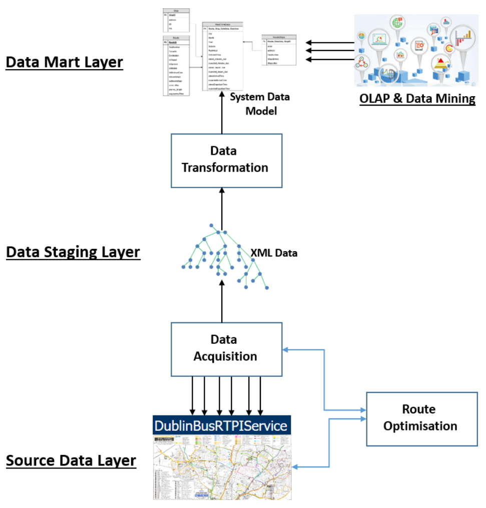


Fig. 1. System Architecture

The cube structure for bus transport data is presented in figure 2, in the form of a star schema, centred around the `FtRealTimeData` fact. It comprises 5 dimensions: `Stop`, `RouteStops`, `Route`, `Time` and `Date`. The Stop, Date and Time dimensions are used for integration with other Cubes through the attributes *latitude*, *longitude*, *Date* and *Time*. These are underlined within figure 2.

3.2 Transport Data Model

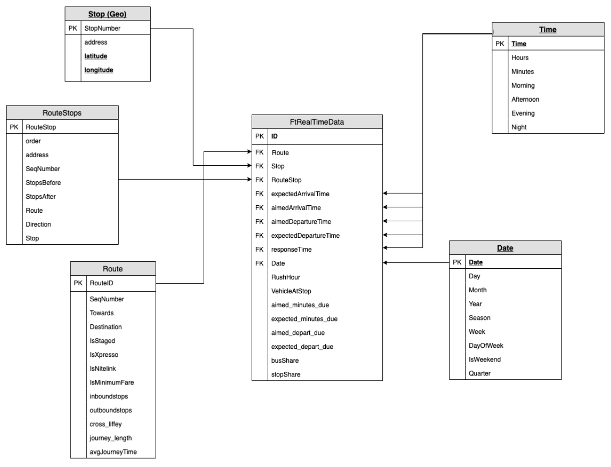


Fig. 2. Transport Data Schema

It is important to note, while we use Dublin Bus for our case study, the model and algorithms presented are sufficiently generic for use in analysing any transport network provided the network can be modelled as a graph; where nodes indicate points of interest (e.g. bus stops, train stations) connected by edges. A route indicates a specific path through the network. This coupled with the availability of a real-time data stream providing arrival times to points of interest could be used by our model for analysis. The `Stop` dimension describes every bus stop across the network with dimensional values for: stop identifier (StopID); a textual address for the stop (address); and the co-ordinates of the stop (latitude, longitude) effectively making this the `Geo` dimension. The `Route` dimension contains *static* information for bus routes such as the start and end points of the route and a number of flags indicating the type of route (e.g. `IsXpresso` indicating if the bus is a commuter route). The `RouteStops` dimension models the stops for each route. This is required in order to examine the route for a specific bus. Information on each stop and their related route lines (stops of one route, stop numbers, stop orders, destinations, directions, etc.) are stored in this dimension. The `RouteStops` dimension models the many to one relationship held between a stop and a route. It comprises: the stop number within that route `order`; a full textual address of the stop `address`; the number of stops preceding the current stop on the route `StopsBefore`; the number of stops after the current stop on the route `StopsAfter`; the route number `Route`; the direction of the route `Direction` (this can either be 'Inbound' or 'Outbound'); and the stop number itself `Stop`. The `Time` dimension is a `role-playing dimension` [6], providing a *many-to-1* relationship to the fact table `FtRealTimeData`. This is used where we wish to analyse across any of the time values extracted from real time information. It stores time in a 24-hour format with each dimensional instance populated with additional flags such as `Morning`, which can be used in different types of analyses. The `Date` dimension is similar to the Time dimension, capturing data such as `Day`, `Week` and `DayOfWeek`. Finally, the `FtRealTimeData` fact table stores for each instance of real-time data: if it is rush hour; if the bus is currently at the stop; the schedule time the bus is due at the stop; the *real-time* due time; the scheduled departure time; and *real-time* departure time. In terms of persistence the dimensions `Stop (Geo)`, `Route` and `RouteStops` are largely static and are harvested once. However, they may change as Routes are added and removed or a specific route is changed. This in turn introduces a historical aspect to these dimensions, meaning they are slowly changing dimensions [6].

3.3 Data Extraction

Data is extracted from the RTPI service [4] in 2 minute intervals: this is our limit for data acquisition without losing real-time information. Of the functions provided by the service, our system uses: `GetRoutes`, which provides a high level overview of a route, including the start and end address of the route and the name of the route; `GetStopDataByRoute`, which provides data for each stop, on a particular route.; and `GetRealTimeStopData`, which provides the realtime component of the system. It takes a stop number as input and returns a list of routes which are due at the stop. It provides 28 attributes for each bus due, most of which are repeated metadata values. The data used for our work are: the bus number due at the stop, a flag indicating whether or not the bus is in congested traffic, a flag indicating whether or not the bus is at the stop, the time of the server response and the scheduled and actual arrival times for the bus.

3.4 Data Transformation

The data is obtained from the service in XML format. For some attributes, the data is taken directly from the raw source and placed into the data warehouse. However, a number of attributes are provided by data enrichment. They are as follows: `Route.cross_liffey` indicates if the route travels across the city, this is manually annotated for each route. `Route.inboundstops & Route.outbound stops` are the number of stops on the route. `Route.journey_length` approximates the length of the route based on the distances between all stops on the route. `Route.avgJourneyTime` is populated from historical data denoting the travel time in minutes of the route. Finally, the attributes `busShare` and `stopShare` in the fact table denote how many buses are due at a particular stop within 5 minutes and the total number of routes which share this stop respectively.

4 Extracting Representative Samples

There are close to 4,000 bus stops in the Dublin Bus network, each updated every minute. In order to extract the entire network, this requires between 12,000 and 20,000 queries and requires up to 30 minutes on a typical quad-core workstation. If we limit each extraction to 2 minutes, this allows for up to 600 bus stops to be extracted. If we were to limit extraction to one minute intervals we would effectively halve the size of our dataset, limiting the number of complete routes that can be obtained, providing us with less data for analysis. Therefore, a strategy is required to maximise the extracted dataset. A small number of key steps are needed to obtain the *best* subset of the data. By *best*, we meant to maximise the amount of data acquired while minimising the number of queries (the set of bus stops). In order to maximise the dataset for the minimum queries, a route similarity matrix is constructed, which compares routes based on their number of *shared* stops. When constructing the similarity matrix, each route can be considered as a distinct set of stops which a bus must visit, in a specific order. The routes were then compared to each other by comparing the number of stops they share and the total number of stops for both routes. This produces the matrix shown in Table 1. We can use this matrix to determine how similar two routes are. For example, the routes `120` and `116` share 60% of their stops. Using this matrix, the algorithm then selects routes with the highest degree of interaction. The Data Acquisition command is specified in Definition 1 using a sql-like syntax. It informs the system how to acquire the required dataset based on routes or specific bus stops.
In Definition 1, the acquisition expression uses 3 sub-expressions: INSERT INTO which must specify the name of the fact (always `FtRealTimeData` in our case); the optional WHERE clause is used to provide a set of routes or a set of stops; and the optional LIMIT clause contains either a time limit expressed in seconds

Table 1. Route similarity Matrix

Route	**1**	**11**	**116**	**118**	**120**	**122**	**123**	**13**
1	1	0.45	0.43	0.65	0.54	0.46	0.48	0.34
11		1	0.57	0.76	0.63	0.56	0.58	0.44
116			1	0.85	0.6	0.52	0.55	0.38

or a bus stop limit. If the user does not specify a limit, the default limit, the number of stops which can be obtained in 2 minutes, will be used. Additionally, if no route or stop list is presented the system will choose two stops with the highest number of routes as default. Example 1 provides a sample query which would be used to populate the fact table for the routes 145, 42 and other routes which have the highest degree of interaction. Example 2 allows a user to select specific stops they wish to obtain. The system examines routes which pass through these stops, and then, using the route matrix selects the additional stops to be obtained.

Definition 1. *Data Acquisition Command*

```
INSERT INTO <FACT>
WHERE [ROUTE IN <ROUTELIST>] || [STOP IN <STOPLIST>]
LIMIT [TIME_LIMIT] || [STOP_LIMIT]
```

Example 1. Data acquisition for route

```
INSERT INTO FtRealTimeData
WHERE ROUTE IN (145,42)
```

Example 2. Data acquisition by Stops

```
INSERT INTO FtRealTimeData
WHERE STOP IN (4331, 7475)
```

The minimum value for the stop limit is 5, as that is the smallest number of stops which make up a particular route. The system starts by examining the list of potential routes which can be obtained. As each route can be considered as a set of stops, the system seeks to obtain the maximum number of stops by examining commonalities across routes using the route similarity matrices. When a route is selected, it is added to a list and the similarity matrix is consulted to add the next route with the highest similarity to those already selected. This process continues until either the upper limit of stops is reached, or that there are no more routes which can be added without passing the upper limit. For our experiments, the query used to gather the data used was based on the default values, effectively making our query `INSERT INTO FtRealTimeData`. Due to the nature of the bus network, it is inevitable that we will also obtain information on incomplete routes. In total, there are 10 full routes and 105 routes in total (95 route fragments).

5 Evaluation

5.1 OLAP Extensions

OLAP functionality (such as Slice, Dice and Pivot) can be used on the fact table to extract and view data for further analysis. However, due to the complexity of the data, these commands may not generate datasets suited to the required analyses to extract insights from the dataset. Because of this, we introduce three OLAP extensions which are used to produce analysis-ready datasets. These are:

`series`, `lag` and `interval`. **Series**: If an analyst wishes to examine a particular route and direction a query, such as the one shown in Ex 3 would be used.

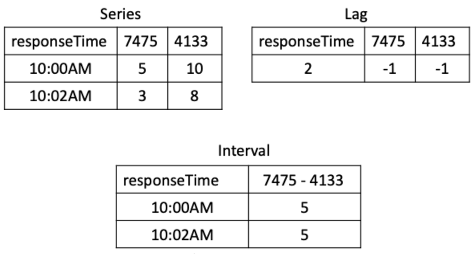
Example 3. Sample SQL for route and direction

```
SELECT *
FROM FtRealTimeData join RouteStops
where Route='145' and Direction ='I'
ORDER BY Date, responseTime;
```

Output for query shown in Ex 3

responseTime	Stop	aim_min_due
10:00 AM	7475	5
10:00 AM	4133	10
10:02 AM	7475	3

This would produce a list of all data grabbed from the real-time system based on the time they were scraped. For an individual bus, there are several measurements per individual stop on the route each taken at different times. Such data



Series

responseTime	7475	4133
10:00AM	5	10
10:02AM	3	8

Lag

responseTime	7475	4133
2	-1	-1

Interval

responseTime	7475 - 4133
10:00AM	5
10:02AM	5

Fig. 3. OLAP extension examples

would need to be pivoted in order to be useful. To accomplish this we provide an OLAP extension called 'series' which produces a time-series representation of the data. This is non-trivial for the dataset, as it requires a list of all stops in order for a route be constructed. This command returns millions of records, a small sample of the output of the `Series` command can be seen in the `Series` table in figure 3. In this example we can see the data has been pivoted in order of stops on the route, with the stops occupying the columns, each row represents all data gathered at a specific time, and each cell occupies the measure `aimed_minutes_due`. As the series works based on a set of ordered stops the extension *requires* a `route` and `direction` as input. **Lag**: A common method used in time-series analysis is *lagging*. This is used to quantify a features rate of change with respect to time. In conjunction with the `series` keyword this can be used to produce a dataset showing the rate of change by time for all measurements. The time taken to execute a series query for 2,291,621 rows was 239 seconds. A sample output can be seen in the `Lag` table in figure 3. The time taken to execute a lag query for a series of size $133124 * 73$ was 3 seconds. **Interval**: The series keyword displays the progression of a bus over time, however an analyst may wish to examine the relationships between stops. This functionality is provided by the keyword **interval**. It can be considered similarly to lagging data in standard time series analysis. This keyword is used in conjunction with `series` to produce a dataset listing the intervals between stops. A sample of the output can be seen in the `Interval` table of figure 3. The time taken to execute an interval query for a series of size $133124 * 73$ was 51 seconds.

5.2 Data Collection & Consistency

The purpose of our approach is to collect a subset of big data which is suitable for analysis. To determine the feasibility of our approach we must satisfy two conditions: that our data collection process is *consistent*, and that collected data is suitable for analysis. **Data Collection:** We collected data for 30 days and collected 31,589,416 instances of real-time data (fact table records). Table 2 details the number of instances scraped per day. Looking at the `Count` column we can see that Wednesday and Sunday have the lowest number of instances (a limited number of routes run on Sunday). Wednesday has the lowest number of records at 3762178. This is because the real time system was unavailable one Wednesday. On average we obtain 1000000 records per day. Using this figure, we estimate, had the system been available that we would have obtained 47000000 records, bringing the figure for Wednesday in line with other weekdays. The minimum records count for Tuesday at 1777 can be explained by the fact that the system was first initialised on Tuesday.

Table 2. Total data acquisition by weekday

Weekday	Count	Average	Max	Min	StDev
Monday	5104382	1701461	1783311	1589465	81958
Tuesday	4398690	1099673	1566792	1777	637228
Wednesday	3762178	1254059	1399659	990699	186570
Thursday	5252280	1050456	1501763	35223	558590
Friday	4577739	1144435	1539966	394970	464937
Saturday	4531959	1510653	1530066	1474605	25515
Sunday	3962188	1320729	1393328	1242274	61806

Data Consistency We have shown that we can collect data in a consistent manner, however the next step is to show that the data itself is *consistent.* Missing data is expected as the system may go down, or provide inconsistencies. However, in aggregate, all data for an individual bus should be highly correlated. With this as our metric, we extracted all data for 4 full bus routes as a time series using the `series` keyword. A matrix for all correlations for individual buses on a route were then calculated for each bus; the results are shown in Table 3. `RouteID` is the identifier of the bus route, `Records` is the count of records obtained for the route. `mean`, `std`, `min` and `max` are the average, standard deviation, minimum and maximum correlation found respectively and finally the values `25%`, `50%` and `70%` denote the correlation for their percentiles. The difference in records is due to the differing lengths of the routes. As we can see, the data is highly correlated with all routes having > 0.8 at the 50^{th} percentile and > 0.9 at the 75^{th}. As is common with real world datasets, missing data is the cause for the minimum values. The missing data can manifest either due to the fact that we are obtaining data at 2 minute intervals, or can be due to outages and anomalies within the real time system itself (the focus of our future work).

6 Conclusions

As more smart city services come online, more information becomes available to make strategic decisions for our cities. The velocity and volume of this data may

Table 3. Correlations for bus routes

RouteID	Records	mean	std	min	25%	50%	75%	max
1	2793896	0.77	0.37	-0.95	0.7	0.96	0.99	0.99
2	2136743	0.74	0.44	-0.91	0.73	0.97	0.99	0.99
3	129474	0.72	0.33	-0.447	0.54	0.88	0.98	0.99
4	114442	0.64	0.42	-0.675	0.34	0.85	0.98	0.99

prove too large to manage with limited computing resources. In this work, we present an ETL system for modelling a real-time smart city service and a process which can be used to extract a usable dataset from a service which provides high-volume data at speeds which cannot be captured completely without an investment in computing resources. Our evaluation shows that the data is being captured at a consistent rate and that the datasets themselves are `consistent` and suitable for analysis. Our future work is to conduct a deeper analysis into detecting anomalies across the bus network where delays are caused by an event not captured in traditional urban data streams.

References

1. Bergamaschi, S., Guerra, F., Orsini, M., Sartori, C., and Vincini, M. (2011) A semantic approach to ETL technologies. *Data Knowl. Eng.*, **70**, 717–731.
2. Cappellari, P, De Virgilio, R, Maccioni, A and Roantree, M (2011). A path-oriented rdf index for keyword search query processing. *Database and Expert Systems Applications.* Springer
3. Costa, C., Santos, M.Y. (2016) Basis: A big data architecture for smart cities. *SAI Computing Conference (SAI).*, IEEE
4. DublinBus RTPI Service WSDL site, (2019) http://rtpi.dublinbus.ie/DublinBusRTPIService.asmx
5. Hernández-Muñoz, J. M., Vercher, J. B., Muñoz, L., Galache, J. A., Presser, M., Gómez, Hernández, L.A. and Pettersson, J. (2011). Smart cities at the forefront of the future internet. Springer.
6. R. Kimball and M. Ross, The Data Warehouse Toolkit (2nd ed), Wiley, 2002.
7. Nesi, P., Po, L., Viqueira, J. R., and Trillo-Lado, R. (2017) An integrated smart city platform. *International KEYSTONE Conference on Semantic Keyword-Based Search on Structured Data Sources*, pp. 171–176. Springer.
8. Niinimaki, M. and Niemi, T. (2010) An etl process for olap using rdf/owl ontologies. *Journal on Data Semantics XIII*, **5530**, 97.
9. Petrović, M., Vučković, M., Turajlić, N., Babarogić, S., Aničić, N., and Marjanović, Z. (2017) Automating etl processes using the domain-specific modeling approach. *Information Systems and e-Business Management*, **15**, 425–460.
10. Roantree, M and Liu, J (2014). A heuristic approach to selecting views for materialization. *Software: Practice and Experience*, **44**, 10.
11. Romero, O. and Abelló, A. (2010) A framework for multidimensional design of data warehouses from ontologies. *Data & Knowledge Engineering*, **69**, 1138–1157.
12. Scriney M., O'Connor M., and Roantree M. Integrating Online Data for Smart City Data Marts. *31st British International Conference on Databases, Lecture Notes in Computer Science* 10365, pp. 23-35, 2017.
13. Scriney, M., Xing, C., McCarren, A. and Roantree, M. Using a Similarity Matrix to extract Sample Web Data Streams. *Dublin City University Online Repository, Article 23435, pp.1-15, April 2019.*