Using an Incomplete Data Cube as a Summary Data Sieve

Curtis Dyreson
Department of Computer Science
James Cook University
Townsville, QLD 4811
AUSTRALIA
`curtis@cs.jcu.edu.au`
`http://www.cs.jcu.edu.au/~curtis`

Abstract

An incomplete data cube is a multidimensional hierarchy of aggregate values in which regions of the hierarchy, and the source data from which those regions are derived, are missing. We model an incomplete cube as a collection of complete sub-cubes called cubettes. Each cubette is defined using a precise, simple specification. The set of cubette specifications concisely characterises the information content of the cube. We discuss how these simple specifications can be used to generate a parser to sieve data from an underlying data source and populate an incomplete cube.

1 Introduction

Data cubes are a relatively recent, and popular phenomenon. A brief description of a data cube is that it is a multidimensional hierarchy of aggregate values. Values higher in the hierarchy are further aggregations of those lower in the hierarchy. The utility of the hierarchical organisation is that the user can easily navigate between high and low precision views of the same aggregate data. The hierarchical organisation supports *drill-down*, an operation that increases the precision of the aggregate data being viewed, and *roll-up*, which decreases that precision. For instance, suppose that a store manager is using a data cube to look at monthly sales for shoes and notices that sales in January were low. To analyse the poor sales the manager might drill-down to look at monthly sales by type of shoe or she might roll-up to look at sales for all product types combined. Several vendors already have cube products on the market, either as add-ons to existing databases or as stand-alone tools, and a "cube" operator has been proposed for inclusion in future SQL standards [3].

A cube can be implemented using a *lazy*, *eager*, or *semi-eager* strategy [6]. The eager strategy materialises every aggregate value in the cube hierarchy. The advantage of the eager strategy is that values can be quickly fetched from the cube during a query. The primary disadvantage is high storage cost (Shukla et al. present algorithms for estimating cube size [5]). For many applications eager cubes

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

are just too big. A lazy implementation strategy does not materialise values. Instead, the values in the cube are computed from the underlying relations during a query. The disadvantage of the lazy strategy is that it slows query evaluation (faster evaluation algorithms are being researched [1]). The semi-eager strategy eagerly materialises only some regions in the cube, and lazily computes others when needed during a query [4].

An *incomplete* data cube is also a multidimensional hierarchy of aggregate values. But in an incomplete data cube regions of the hierarchy, and the source data from which those regions are derived, are missing. For example, a data cube administrator may decide that hourly sales data from two years ago is no longer needed, daily sales data will suffice. The administrator can remove the aged, hourly data from the cube. The missing region makes the data cube incomplete and some queries (e.g., what are the hourly sales figures over the lifetime of the enterprise) can no longer be satisfied. Incomplete cubes have mechanisms for handling queries in the missing regions, such as suggesting alternative, complete queries and computing partial results.

In terms of storage, an incomplete cube has the same desirable behaviour as lazy and semi-eager cubes. Each materialises only part of what would be stored in an eager cube; the incomplete or unmaterialised portions incur no storage cost. For example, assume that a regional sales officer wants aggregate data for sales at stores in her region for every hour in 1995, but for stores in other regions, aggregate data for each day will suffice. In an eager cube an aggregate value for every combination of store and hour must be stored resulting in a much larger cube than needed. In contrast, an incomplete cube only stores the relatively small amount of data specified as needed, the hourly data for the other stores forms an incomplete region. Incomplete, lazy, and semi-eager cubes also scale well, new dimensions can be added to the cube and existing dimensions can increase in size (i.e., a more precise measure can be added to the dimension) with no adjustment to the existing cube storage. The resulting cube is merely incomplete in the new dimension, and can be populated as needed later.

But in one important respect an incomplete data cube is like an eager data cube, and unlike a lazy or semi-eager cube. Eager and incomplete cubes do not need the source data from which aggregate values in the cube are derived. Both lazy and semi-eager cubes presume that the source data is still available, so that an aggregate value which is not stored in the cube can be computed when needed. Both strategies *tightly couple* the cube to a data source. Eager and incomplete cubes, on the other hand, *uncouple* the cube from the source data.

In general, an incomplete cube is useful in situations where a complete, eager cube would be unnecessarily large, but where a lazy or semi-eager cube cannot be used because the source data is not available or expensive to query. We conjecture that an incomplete data cube would be useful in the following scenarios, among others.

- One reason that data cubes are popular is that many data collections are characterised by the property that as data in the collection ages, each datum individually becomes less relevant, but remains relevant in aggregate. For such data collections, a data cube can be used to store the aggregated historical data, allowing the original data to be archived or deleted and resulting in considerable savings in space.

- A data cube is used to summarise data from a log file or flat file. For example, suppose that a data cube is used to store aggregate data from a log file of sales transactions rather than a sales relation. To search a large log file and retrieve data during query evaluation imposes a heavy burden on system resources, so the data cube's administrator decides to use an incomplete data cube and package requests for more data in an overnight cron job.

- Aggregate data is broadcast on a network by various sites. The aggregate data from external sites is collected and inserted into a cube at each site, but the source data is not shipped across the network for a number of reasons (privacy, cost of broadcasting and duplicating the source data at each site, etc.).

- The cube contains regions of secret data and the authorisation to view the secret data varies from user to user, that is, some users can see all of the data, others only a portion, still others a different portion, etc. In an incomplete cube, it is easy to create a different, incomplete view of the same complete cube for each class of authorised user. The data can be kept secret by hiding it in an incomplete region.

This paper describes how an incomplete data cube can be used to sieve relevant summary data from an unstructured text file which is too large to store and query, such as a rapidly-growing log file. The next section briefly describes a real-world problem that could benefit from the application of incomplete data cube technology. We will use this example problem in the remainder of the paper. We then discuss an incomplete data cube in more detail. An incomplete cube is a federation of complete sub-cubes, which we call *cubettes*. Each cubette is defined using a precise, simple specification. The set of cubette specifications concisely characterises the information content of the cube. We then discuss how these simple specifications can be used to generate a parser to sieve data from an underlying data source and populate an incomplete cube.

More information on incomplete data cubes as well as a prototype implementation in the Java programming language is available at `http://www.cs.jcu.edu.au/~curtis/IncompleteCube.html`.

2 Motivating example - analysis of the World-Wide Web access log

A very rough description of the World-Wide Web (WWW) is that it is a network of information servers. A server responds to a client (browser) request by supplying the appropriate information, typically a page in a hypertext document. The authors of these documents usually want to know how their documents are being used. In particular, they would like to ascertain how frequently their pages are being requested and what kinds of users are requesting them. Of course there will likely be many different authors at a site and each will want a very detailed analysis of their materials. In addition, others at the site will want other kinds of summary information, for instance the server administrator might want to monitor the overall daily traffic from each country. So in general both detailed and abstract summaries must be collected and maintained.

The starting place for any quantitative analysis of WWW page usage is the server *access log*. The access log is a history of server requests. Each record in the log has a timestamp, the requesting machine IP name or number, and the resource requested. For example, in the following access log record:

```
crab.jcu.edu.au - - [01/Jan/1997:17:55:24] "GET /web/home.html HTTP/1.0" 200 4748
```

the timestamp is `[01/Jan/1997:17:55:24]`, the requesting machine is `crab.jcu.edu.au`, and the resource is `/web/home.html`. Although each record in the access log also has other information, such as the response code, without loss of generality, we ignore that information in this paper.

A single access log, however, will never contain the entire request history for pages on a server. Many clients will use a proxy cache server. A proxy cache server keeps copies of frequently requested pages in its cache. A page request for a cached page will be handled entirely by the proxy, and so will be logged in its access log. So the complete history of requests for a page is commonly distributed over many log files. In order to more accurately assess the use of WWW pages, the data in these distributed log files needs to be integrated.

Few, if any, of these log files are kept entirely in situ. Logically, the access log is an append-only file since server history only accumulates. The access log for the Department of Computer Science server at James Cook University grows by about a megabyte a day. Physically, some technique must be used to limit the file's growth, otherwise, at many sites, it will quickly exceed storage capacity. Typically, as the access log grows, it is periodically moved to a different storage medium, e.g., from disk to tape, or from an uncompressed to a compressed version, and the log is restarted.

Even if the entire access log is kept, access to the log file at many sites must be restricted to protect the privacy of both clients and authors. So it would be beneficial if the relevant access log data could be hidden or made available on an individual basis.

In summary, a data analysis tool is needed that can generate a parser to sieve data from a text file prior to the file being archived, allows the piecemeal specification and retention of relevant summaries, is able to integrate several data sources, and supports data hiding. While a complete data cube can satisfy some of these requirements, the resulting data cube will likely be too large. An incomplete data cube can satisfy all the requirements in a minimal amount of space since only the data specifically requested is put into the cube (and just like in a complete cube, that data can still be compressed and compacted).

3 Measures and units

The three kinds of information in an access log record come from separate domains. The time is a value from the temporal domain, the source machine is from the domain of machine names (a spatial domain), and the file name is from the domain of file names (a second spatial domain). Each kind of information is in fact a unit of measurement, and given to an implied system of measurement. For example, the temporal information in an access log record is measured in *seconds*. Individual units in a measurement by seconds are `01/Jan/1997:17:55:24`, `01/Jan/1997:17:55:25`, etc. In the remainder of this paper we will refer to a system of measurement as a *measure*, e.g., seconds, and a unit of measurement as a *unit*, e.g., a particular second.

There can be many different measures in a single domain. For instance, in the temporal domain, a page request might be measured by the second, hour, day, week, month, or year in which it occurs. Most of these measures are related insofar as some are strictly more precise than others. For example, a temporal measurement given in days is more precise than one given in weeks since a measurement in days pinpoints the week but not vice-versa (since there are seven days in every week). In the next section we discuss how units, measures, and the relationships among them are used in an incomplete data cube.

4 Data cubes

In this section we describe an incomplete data cube in more detail.

A useful way of understanding a data cube is to conceive of it as a dataflow graph. An edge in the graph represents a data dependency; the data at the "to" node is derived from the data at the "from" node. In the graph there are two kinds of nodes: *source* nodes and *derived* nodes.

A source node represents a group of facts drawn from some underlying data source, usually a database relation. Facts are precisely measured in several dimensions and those with the same measurements are grouped at the same source node. The most common measurement dimensions are space and time. For example, facts could be grouped based on the temporal measure of days and the spatial measure of countries. In such a grouping there would be one source node for every combination of country and day; facts having the same country and day measurements would be placed in the same source node.

A derived node, on the other hand, holds the result of an aggregate operation applied to the data at the incoming nodes. A derived node on an edge from a source node holds the result of computing an aggregate, e.g., **count**, on the group of facts at the source node. Other derived nodes hold the result of an aggregate operation, e.g., **sum**, applied to the aggregate values on the incoming edges.

An example graph is shown in Figure 1. The graph assumes that there is only one domain of measurement: time. The lone source node depicted is the group `1 jan 1997`. Facts that share a temporal measurement of 1/Jan/1997 are in the group at this source node. Above the source node is a derived node, the unit `1 jan 1997`. The aggregate value at this node is a count of all the page requests for the first of January. An edge connects the source node to this node because the aggregate value is computed using the data at the source node. Above that derived node are other derived nodes, with the data dependencies indicated by the edges.

The dependencies in the graph are given by the relationships among the measures in each dimension. In general, units in less precise measures are dependent on those in a more precise measure. For example, the measure of months is less precise than that of days. The graph in Figure 1 shows that the unit `jan 1997` (which is in the measure of months) is dependent on the units `1 jan 1997` through `31 jan 1997`.

As an aside, we note that units, measures, and the relationships among them are specified in internal cube tables (we assume finite, bounded domains). Figure 2 presents a simplified view of the internal tables which describe the dependencies in the graph for Figure 1. The *Edges* table is the set of edges between nodes (the edge from a source node to a derived node is implicit). The *Groups* table is the set of groups. The membership pattern is a regular expression which is used to identify members of the group when scanning the access log. To reconfigure the cube for a text file with a different date format, the *Groups* table must be edited. Although the cube is a multidimensional space, the tables for each dimension are specified independently.

In an incomplete data cube the values of some nodes in the graph are *incomplete*. An incomplete value is not stored and can not be computed from the values that it depends on (at least one of them must also be incomplete, otherwise the value could be computed).

4.1 Cubettes

Integral to an incomplete data cube is a concise, high-level description of the complete regions in the cube (and by omission, which regions are incomplete). We will call a complete region a *cubette*, to signify that it is a diminutive, complete cube within the incomplete cube hierarchy. A cubette is concisely specified as a combination of a unit, u, and a measure m, and is written $u@m$ (literally u at m). The cubette specification describes a subgraph that extends from an apex at u to those units in the measure of m that are on a path from a source node to u. The units at measure m in the subgraph form the *base* of the cubette. For example, consider the cubette specification `jan 1997@days`. The base of the cubette is all the days that are on a path from a source node to `jan 1997`, that is, all the days in the month of January 1997. All the nodes in the cubette from the base (the days in January 1997) to the apex (`jan 1997`) are complete.

The core of an incomplete cube is the set of cubette specifications. The set represents the information content of the incomplete cube. We anticipate that the set will be moderately large, between a thousand and a hundred thousand cubettes. Each cubette is an independent, complete cube within the incomplete cube and can be implemented using a lazy, semi-eager, or eager strategy. Elsewhere we give algorithms for inserting new cubettes (which is more involved than simply adding a new specification to the set) and deleting cubettes [2].

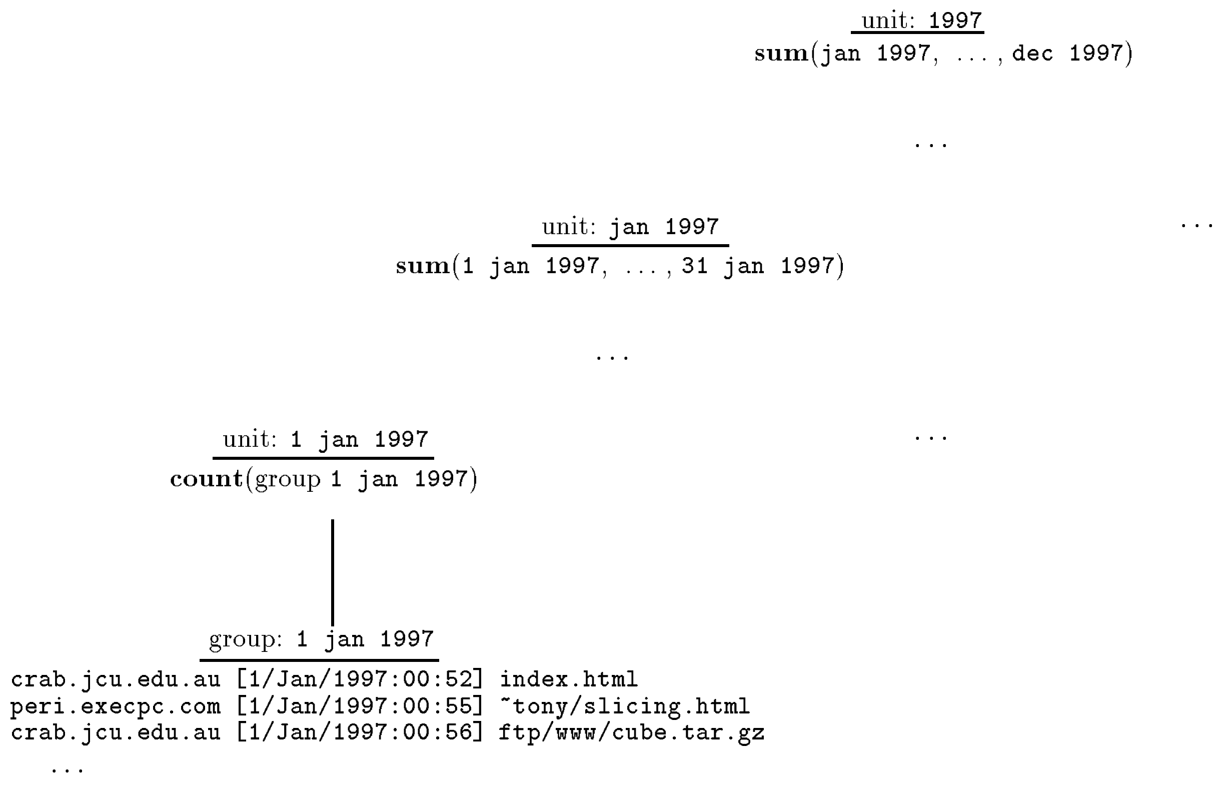
unit: 1997
sum(jan 1997, ... , dec 1997)
...
unit: jan 1997
sum(1 jan 1997, ... , 31 jan 1997)
...
...
unit: 1 jan 1997
count(group 1 jan 1997)
...
group: 1 jan 1997
crab.jcu.edu.au [1/Jan/1997:00:52] index.html
peri.execpc.com [1/Jan/1997:00:55] ~tony/slicing.html
crab.jcu.edu.au [1/Jan/1997:00:56] ftp/www/cube.tar.gz
...

Figure 1: A graph depicting the hierarchy of dependencies in a data cube

4.2 Queries

A data cube query, such as drill-down, is an operation that retrieves the value at a unit (or set of units). For instance, the user could query for the value of the unit `jan 1997`. This unit may be within a cubette and the query can be satisfied (elsewhere we give an algorithm for quickly determining whether the query can be satisfied [2]), or the unit may be in an incomplete region.

When the user queries for information in an incomplete region, the incomplete cube may have enough information to partially satisfy the query. For example, a query for the value of `jan 1997` can be partially satisfied since most of the values upon which it depends are complete (all save `1 jan 1997`). Various notions of query completeness can be defined and supported in an incomplete cube [2].

Queries in a cube will often fall into an incomplete region since users typically will not know a priori the extent of the complete information in a cube. However, a query in an incomplete region can be redirected to the "nearest" complete region. For example, a query for the incomplete value at `jan 1997` could be redirected to the complete value at `jan 1997` (or to one of the days in January). Elsewhere we give an algorithm for redirecting queries to complete regions[2].

5 Populating the cube

The cubette specifications describe which regions of the graph are complete. These regions can be populated from an underlying text file by parsing the file and matching only records that belong to source nodes that the region (transitively) depends on. In this section we sketch how to build the grammar for the parser.

Initially, the grammar just consists of productions for recognising units in a dimension. These

Edges

to	from
1997	jan 1997
1997	feb 1997
...	...
1997	dec 1997
jan 1997	1 jan 1997
...	...
jan 1997	31 jan 1997
	...

Groups

group	membership pattern
1 jan 1997	`"[1/Jan/1997"[^\]]+`
2 jan 1997	`"[2/Jan/1997"[^\]]+`
3 jan 1997	`"[3/Jan/1997"[^\]]+`
4 jan 1997	`"[4/Jan/1997"[^\]]+`
5 jan 1997	`"[5/Jan/1997"[^\]]+`
...	...
31 dec 1997	`"[31/Dec/1997"[^\]]+`
	...

Figure 2: An example of the tables that store the dependencies in a cube

productions are built from the tables described in Section 3. The terminals in the grammar are the groups (they are the tokens recognised by the lexical analyser). Groups have an associated regular expression that defines their members as shown in Figure 2. The nonterminals are the derived nodes. There is one production in the grammar for each derived node in the graph. The body of the production consists of all the nodes on incoming edges. An example grammar in BNF for the units depicted in the graph in Figure 1 is given below. The terminals are the day groups.

1997 ::= *Jan_1997* | *Feb_1997* | ... | Dec_1997;
Jan_1997 ::= *1_Jan_1997* | *2_Jan_1997* | ... | *31_Jan_1997*;
...
Dec_1997 ::= *1_Dec_1997* | *2_Dec_1997* | ... | *31_Dec_1997*;
1_Jan_1997 ::= `1_Jan_1997`;
...
31_Dec_1997 ::= `31_Dec_1997`;

So the token `1_jan_1997` would be recognised as *1_Jan_1997*, *Jan_1997*, and *1997*.

To complete the grammar, we need to add a production that identifies which records will be considered legal sentences. We only want records that provide information for a cubette. So we add a production that defines a record as the sequence of units in each cubette specification. For example, suppose that we have the following cubette specifications: `jan 1997@days`, `nov 1997@months`, and `1 dec 1997@days`. Then we would add the following production to the grammar.

Record ::= *Jan_1997* | *Nov_1997* | *1_Dec_1997*;

Record is the start symbol in the grammar. A parser for this grammar will only accept records pertinent to January, November, or December 1. Only these records will be needed to compute the values in the complete regions of the cube.

This grammar can be fed directly into a parser generator (e.g., yacc) to construct a parser for populating an incomplete data cube. The grammar itself can be automatically generated from the tables and set of cubette specifications. Although we have not discussed the actions that need to be associated with each production, these actions can also be automatically generated.

There are, however, several problems with this solution that all have to do with the fact that available parser generation packages tend to be limited. The first is that for an N-dimensional cube, the resulting grammar will require N lookahead symbols. Most parser generators can only generate single lookahead parsers. The second problem is that top-down parsers will be much too slow, they might explore the entire cube graph for every record, so some bottom-up parsing technique has to be utilised. But bottom-up parsers usually generate tables that are proportional in size to the product of the number of terminals and the number of parsing states. There will be a large number of terminals (one for every unit in a dimension). Finally, if overlapping cubette specifications are allowed the generated grammar will be ambiguous (since some records might satisfy two or more cubettes). To handle this kind of ambiguous grammar the constructed parser should allow "multiple passes" over the token stream for some productions.

6 Conclusions

An incomplete data cube is a data cube that allows incomplete regions to exist in the cube hierarchy. Such cubes are useful when the analytic power of the cube organisation is needed, a complete, eager cube is too large, and the underlying source data for the incomplete regions is unavailable. We briefly described a real-world problem that fits these criteria. To analyse the WWW access log, a site administrator can specify which summary data to put in the cube by giving a set of cubette specifications, e.g., to monitor the use of lecture notes for an Introduction to Database course from machines on campus week-by-week during the first semester, the site administrator would add the following cubette specification (assuming the relevant units and measures exist).

```
jcu@machines, first_semester_1997@weeks, database_lecture_notes@pages
```

The set of cubette specifications is a high-level description of the information content of the cube. It is essential to querying the cube, and can also be used to automatically generate a data sieve for populating the cube.

References

[1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the 22st Conference on Very Large Databases*, pages 506–521, Mumbai, India, September 1996.

[2] C. Dyreson. Information retrieval from an incomplete data cube. In *Proceedings of the 22st Conference on Very Large Databases*, pages 532–543, Mumbai, India, September 1996.

[3] J. Gray, Bosworth A., A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the 12th International Conference on Data Engineering*, pages 152–159, New Orleans, LA, 1996.

[4] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 311–322, Montreal, Canada, June 1996.

[5] A. Shukla, P. M. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proceedings of the 22st Conference on Very Large Databases*, pages 522–531, Mumbai, India, September 1996.

[6] J. Widom. Research problems in data warehousing. In *Proceedings of the 4th Int'l Conference on Information and Knowledge Management (CIKM)*, November 1995.