OBJECTCUBE – A GENERIC MULTI-DIMENSIONAL MODEL FOR MEDIA BROWSING

Grímur Tómasson
Master of Science
Computer Science
January 2011
School of Computer Science
Reykjavík University

M.Sc. RESEARCH THESIS

ISSN 1670-8539

ObjectCube – A Generic Multi-Dimensional Model for Media Browsing

by

Grímur Tómasson

Research thesis submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science in **Computer Science**

January 2011

Research Thesis Committee:

Björn Þór Jónsson, Supervisor
Associate Professor, Reykjavík University

Laurent Amsaleg
Research Scientist, IRISA-CNRS

Hannes Högni Vilhjálmsson
Associate Professor, Reykjavík University

ObjectCube – A Generic Multi-Dimensional Model for Media Browsing

Grímur Tómasson

January 2011

Abstract

Since the introduction of personal computers, personal collections of digital media have been growing ever larger, with users facing an ever increased difficulty in organizing and retrieving the contents of their computers. It is therefore increasingly important to provide effective tools to organize and browse such collections. We propose a multi-dimensional model for media browsing, called ObjectCube, based on the multi-dimensional model commonly used in OLAP applications for business intelligence. We then describe a prototype of a media browser based on the ObjectCube model, and evaluate its performance using three different underlying data stores and image collections of up to one million images.

ObjectCube –
Almennt Fjölvítt Líkan fyrir Gagnaskoðun

Grímur Tómasson

Janúar 2011

Útdráttur

Frá því að einkatölvuvæðingin hófst hafa söfn stafrænna upplýsinga vaxið óðfluga, sem og vandamál notenda við að skipuleggja og finna upplýsingar. Það hefur því orðið stöðugt mikilvægara að þróa skilvirkar aðferðir til þess að skipuleggja og skoða slík söfn. Okkar tillaga að lausn er fjölvítt líkan ætlað fyrir gagnaskoðun, nefnt ObjectCube, sem byggir á því fjölvíða líkani sem almennt er notað í OLAP hugbúnaði fyrir viðskiptagreiningu. Við lýsum einnig frumgerð hugbúnaðar sem útfærir ObjectCube líkanið og metum svo frammistöðu frumgerðarinnar fyrir þrjár mismunandi gagnageymslur, með myndasöfnum sem innihalda allt upp í milljón myndir.

To Helga Hrönn Jónasdóttir, who's support made this possible,
and to Freyr & Urður, for putting up with a frequently absent father.

Acknowledgements

I would like to thank Björn Þór Jónsson, Laurent Amsaleg and Kári Harðarson for invaluable input into the project this thesis represents. Their enthusiasm for multi-dimensional browsing is inspiring.

The input and feedback of Hlynur Sigurþórsson, the creator of ObjectCube's image browsing interface, and the constructive criticism of Gylfi Þór Guðmundsson and Haukur Pálmason have also made this project and thesis much better than it otherwise would have been.

The project has been supported by grant 070005023, *Integrated Browsing and Searching of Personal Digital Image Collections*, from the Icelandic Research Fund.

Publications

Part of the material in this thesis has been submitted to an international conference. Co-authors are Björn Þór Jónsson (Reykjavík University) and Laurent Amsaleg (IRISA-CNRS). While Björn and Laurent contributed significantly to the writing of the conference submission, the implementation and experimentation is entirely my work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the introduction of personal computers, and digital recording devices in particular, personal collections of digital media have been growing ever larger. A typical personal computer now contains a multitude of files, images, videos and music. Keeping track of the location and contents of all these media files is turning into a major headache for most people, and users find it increasingly difficult to organize and retrieve the contents of their computer.

Consider, for example, the particular case of personal digital image collections. The number of images in a typical household collection is likely to be in the thousands and photography enthusiasts may easily have tens of thousands of images. It is therefore increasingly important to provide effective tools for browsing and searching such personal image collections.

What, one might ask, makes a browser effective? A simple answer to that question might be: The capability to define and present the set of images that the user is interested in. Many of the current browsers indeed excel at the presentation aspect; the sets of interest are arranged nicely on the screen, and can easily be exported to web-pages, slide shows, or e-mails. Those browsers, however, are unfortunately quite limited in their capabilities for defining these sets of images of interest. Consider for example the following two simple—and very plausible—scenarios:

Scenario 1: A user wishes to retrieve all images containing one or more of her friends, that have been taken during summer, and are brightly lit. Furthermore, the images should be arranged by the location where they were taken.

Scenario 2: A user wishes to retrieve all images taken somewhere in Europe, which contain at least one of his children and at least one of their grandparents. Viewing those

images in a three-dimensional display, grouped by child, grandparent and location, might then make sense.

Unfortunately, however, the browsing tools available today are ill suited for defining the set of images of interest for browsing scenarios such as these. These tools typically support many other features, as mentioned earlier, but it is the browsing part that is the focus of this thesis.

1.1 Current Browsers

Current image browsers can roughly be split into two categories. The first category contains simple browsers that allow browsing of file system folders and viewing of images contained in those folders. The only way to organize images using this type of browser is by creating the appropriate folder structures; they do not support tagging or other meta-data. While the folder structure allows categorization of images, it only allows categorization according to a single criterion. We might for example, referring to the scenarios above, categorize them by location or by time, but we cannot categorize them by both location and time, as well as by the people in the images. Examples of products that fall into this category include ACDSee, FastStone and file browsers.

The second category contains more powerful browsers that allow the application of tags or other meta-data to images, or even parts of images. Some of these browsers support folder hierarchies and handle time quite well, and some of them even support automated tagging using face recognition. Furthermore, the support for tagging adds the ability to search for images containing a certain tag. Products like Flickr, iPhoto and Picasa fall into this category.

In both cases there is no ability to browse images using other categories than the one supplied by folder structure, meaning that an image can only be assigned to a single category. While multiple tags can be added to a single image, tags are essentially attributes, but not categories, since there is no mechanism for defining any relations between tags. In order to find all images with some friend in them, the user must type in the names of all her friends and search. There is limited or no support for filtering images according to a value range, for example a time period or brightness. On top of that, it is not possible to view images defined by more than, at best, a few criteria (tags) and a single category, and there is limited support for grouping images by contents. The tools we have today are thus less than satisfactory.

In addition, some research projects have considered browsing. These include PhotoMesa (Bederson, 2001), Pibe (Bartolini, Ciaccia, & Patella, 2004), PhotoFinder (Kang & Shneiderman, 2000), Scenique (Bartolini & Ciaccia, 2009) and Camelis (Ferré, 2009). Overall, these projects are either quite rigid in defining hierarchies and/or tags, or it is quite unclear whether they can handle realistic collections of images. While interesting those research projects do not provide an acceptable solution to image browsing either.

1.2 Multi-Dimensional Browsing

The multi-dimensional model of data, which is commonly used in OLAP (On-Line Analytical Processing) applications, has been used very effectively in Business Intelligence applications to group and aggregate numerical data, such as sales and profits. For example, such a model could be used to view total sales broken down by products, store locations, and years. While the multi-dimensional model is not immediately suited to image browsing, we believe that the similarity of the problems is significant enough to use the concepts of that model as the foundation of a powerful and flexible solution to media browsing.

The two key concepts of the multi-dimensional model are dimensions—including hierarchies—intended for defining interesting sets of data, and facts, or numerical attributes, which can be aggregated to present an easy-to-understand view of these sets of interest. Considering, for example, the two scenarios above, the following three dimensions are immediately apparent: time, people, and location. The facts are that which we are interested in; images in our case. Aggregating images is, however, not straightforward, but a stack of images may be thought of af an aggregation of sorts.

1.3 Contributions of the Thesis

The work described in this thesis, is part of a larger project with the goal of creating a an efficient and effective media browser based on the concepts of the multi-dimensional model. It is inspired by Hardarson and Jonsson (2007). In this thesis, the focus is on the architecture of the underlying application model, and the corresponding APIs. This thesis makes the following three major contributions:

- First, we define a new generic multi-dimensional model suitable to media browsing, called ObjectCube, based on the concepts of the multi-dimensional model commonly used in OLAP applications.

- Second, we describe a prototype of a media browser based on the ObjectCube model. The prototype is architected to be highly flexible, supporting multiple operating systems and multiple underlying data stores. It supports automated tagging of media through a plug-in architecture, and provides both C++ and Python APIs for user-interface programmers.
- Third, we present detailed measurements of the performance of the prototype, using three different data stores and image collections of up to one million images.

In parallel, to the work described here, a graphical user interface is under development. Using a highly successful model, OLAP, as a starting point leads us to high expectations regarding usability. Usability studies must, however, wait for a completed user interface.

Note that neither ObjectCube the model nor prototype presented in this thesis are limited to images, but can be applied to arbitrary browsing scenarios, such as file browsing, audio browsing, or document browsing. In order to ground the discussion, however, we have chosen to focus on image browsing in our presentation and experiments.

1.4 Overview of the Thesis

In Chapter 2, we present the multi-dimensional model used in OLAP applications. In Chapter 3, we define the new ObjectCube model, and describe its relationship to the multi-dimensional model. In Chapter 4, we describe the architecture of the prototype implementation. In Chapter 5, we describe the experimental analysis of the performance of the prototype and, in Chapter 6, we describe related work, focusing in particular on image browsing. Finally, in Chapter 7, we conclude and discuss possible avenues for future work.

Chapter 2

Multi-Dimensional Analysis

A multi-dimensional analysis is a way of looking at large amounts of data using aggregation and grouping, and thus enabling the discovery of trends and patterns. The best known usage of multi-dimensional analysis is in Business Intelligence.

The ObjectCube model is the foundation of the work of this thesis, and since it is based on key concepts from multi-dimensional analysis, or OLAP, it is imperative to examine those concepts in some detail.

In Section 2.1 we take a brief look at the history of multi-dimensional analysis. In Section 2.2 we give an example of a typical multi-dimensional analysis usage. In Section 2.3 we look at the definition of multi-dimensional analysis and define our scope. In Section 2.4 we first describe a running example, and then examine each key multi-dimensional analysis concept in the context of that example. Finally, we summarize in Section 2.5.

2.1 A Very Short History

The origins of the concept of multi-dimensional analysis goes back to 1957, when Kenneth E. Iverson created the notation later published in his book *A Programming Language* (Iverson, 1962) in 1962. That language, APL, supported multi-dimensional variables, was mathematically defined and used Greek symbols as operators. Despite being considered somewhat elitist and being hard to maintain it was relevant all the way up to the 1980s and even today IBM is selling it's successor APL2.

The acronym OLAP, which stands for On-Line Analytical Processing, is much more recent. It originates in a white-paper Edgar F. Codd et al wrote and published in 1993 (Codd, Codd, & Salley, 1993).

2.2 Typical Usage

The most common usage of multi-dimensional analysis is without a doubt analysis of grouped aggregations of numerical business data, such as sales and profits; as it is one of the key tools in the Business Intelligence (BI) domain. A typical example would be to view aggregations of sales information in groups defined by products, store locations and time. A good starting granularity might be looking at this data by product categories, countries and years. That way, a business analyst or a manager can look at a cube of information where each cell sums up the sales of a certain product category for all stores in a certain country and for a certain year. The analyst can then manipulate the cube to follow the trends or patterns he observes in the information presented in the cube.

2.3 Definition of a Multi-Dimensional Analysis System

Defining a multi-dimensional analysis system is not straightforward, since there has been some controversy regarding what constitutes one. In Codd's white-paper, he set forth twelve rules a system must abide by, to be considered an OLAP solution. It was later discovered that the paper was sponsored by Arbor, a company selling a multi-dimensional data analysis solution. Unsurprisingly, that discovery devalued Codd's definition. Nigel Pendse has criticized Codd's rules for being unsuitable to measure the OLAP compliance of a solution and that they should rather be considered a vendor sponsored brochure than a serious research paper (Pendse, 2008). Pendse has set forth his own five rules defining an OLAP solution. Those five rules are known by the acronym FASMI, which stands for Fast Analysis of Shared Multi-dimensional Information. Aside from those two definitions, there have been other attempts at defining OLAP, for example the glossary of the short lived OLAP Council, which was a vendor driven organization.

While we do not have a firm, universally agreed upon definition of OLAP the situation is much improved when we look at the concepts that describe the multi-dimensional view and how it can be manipulated, all OLAP definitions share a focus on that. For the purposes of this thesis we are only interested in those concepts, while OLAP implementation details and yardsticks for OLAP solution compliance will be brushed aside.

2.4 Multi-Dimensional Analysis Concepts

Before examining each key multi-dimensional concept we define a running example using relational tables.

2.4.1 A Running Example

Figure 2.1 shows the definition of the four relational tables we use as the basis of a running example that is used to explain the key multi-dimensional analysis concepts.

The tables in Figure 2.1 are quite simple and have not been normalized. The *Sales* table in this example is the *fact* table, which contains the information we are interested in examining further. The other three tables, *Product*, *Time* and *Location*, are look-up tables, which provide additional information about the facts of the fact table.

Having set up a running example we now examine each key concept of the multi-dimensional analysis in turn.

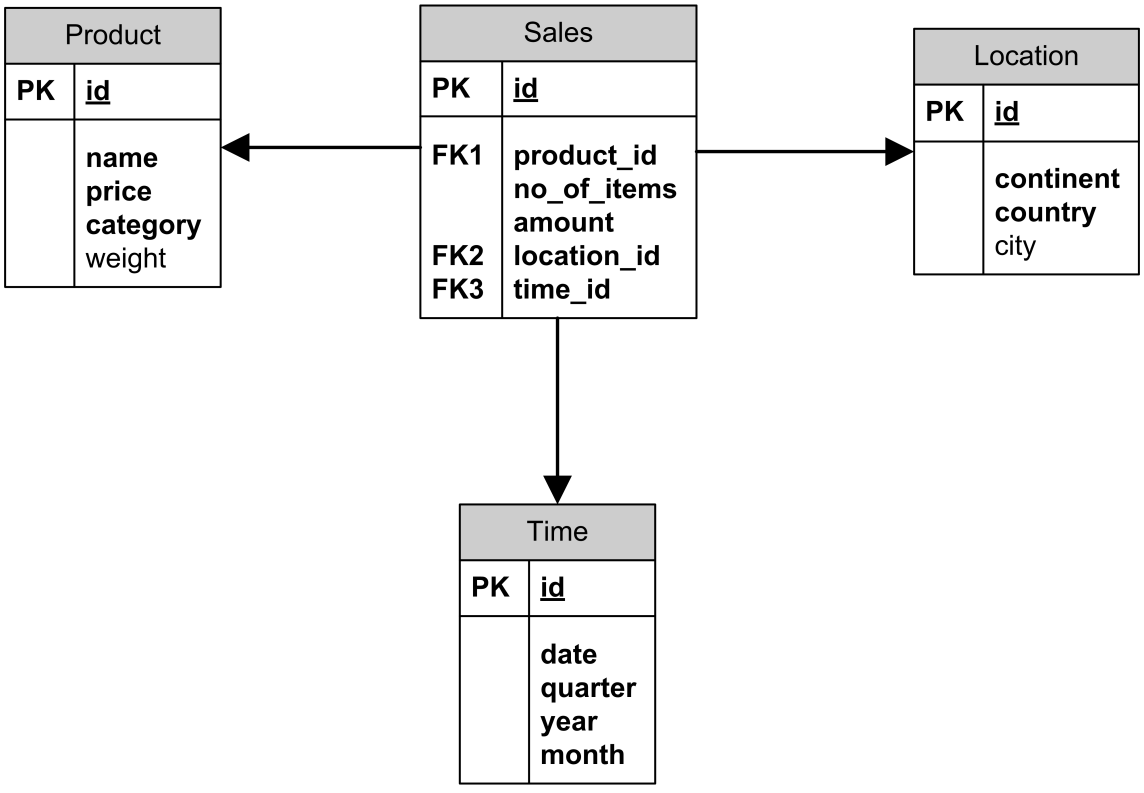


Figure 2.1: Relational tables.

2.4.2 Fact

A *fact* is any numerical attribute we are interested in aggregating and examining further. As we have mentioned earlier, the *Sales* table is our fact table. The facts we are interested in are the contents of the *amount* column in that table, those are our facts.

2.4.3 Dimension & Member

A *dimension* is a category of information. Another way to describe it is as a set of *members* the user perceives to be similar. Each look-up table in our example encompasses a dimension. Consider, for example, the *Time* table, any valid dates, years, quarters or months can be members of the time dimension. A *member title* is the name by which it is known, for example 'month'. An instance of a member is known as a *member value*, for a member with the title 'month', 'January' is a valid instance. In the context of relational tables, the member title is the name of the column, here the *month* column in the *Time* table, and an instance is the value of that column in a certain row, nulls aside. Members are also known as meta-data or labels.

2.4.4 Attribute

An *attribute* is a piece of information that further describes a member, considering the *Product* table, the *id*, or the *name* were it unique, of the product could be a member in the product dimension and the *weight* an attribute of that member.

2.4.5 Hierarchy

A dimension may, and preferable should, contain differing levels of information so that it can be used to examine information with varying granularity. Considering the columns of the *Time* table, we observer that they are of different levels of information, as a single date is not equivalent to a quarter, a month or a year. A *hierarchy* structures the information of a dimension, by organizing a subset of the members of a dimension into parent-child relationships where the parent vertex is an aggregation of its child vertices. That relationship is transitive. Figure 2.2 shows one valid hierarchy in the time dimension. In this (partially completed) hierarchy we have four different levels, or categories, of information. Here, the year 2009 is an aggregation of its quarters, each quarter is an aggregation of its months,

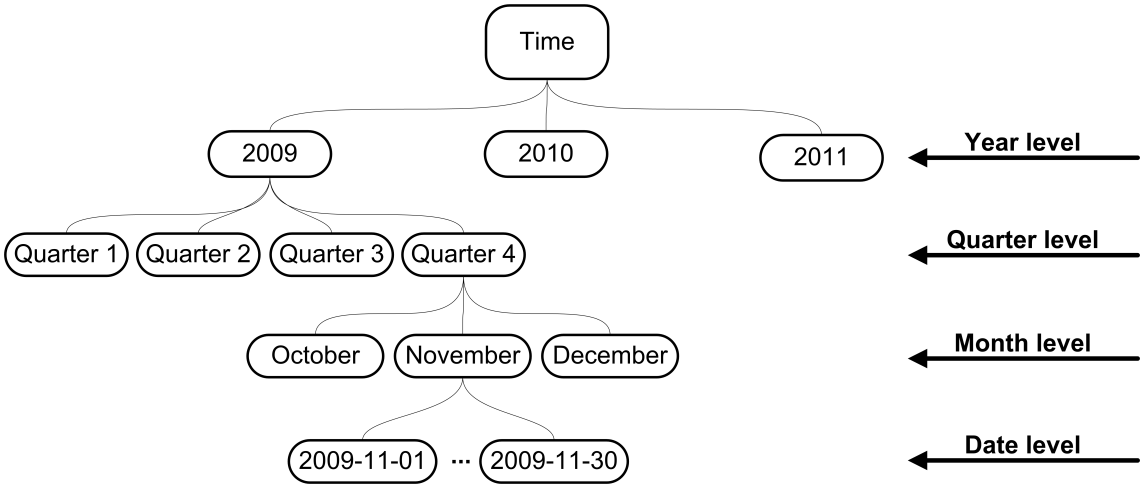


Figure 2.2: Level based hierarchy.

and each month is an aggregation of the dates. This hierarchy is an example of a level based hierarchy, where all the vertices on each level belong to the same category.

While *level based hierarchies* are the most common we also have another basic hierarchy type within the multi-dimensional model, the *value based hierarchy*. A value based hierarchy has one or more levels, where there are at least two vertices that do not belong to the same category. Figure 2.3 shows a typical example of a value based hierarchy; an organizational chart. In this case we still have a normal aggregation, the CTO is, for example, responsible for all the staff below him in the hierarchy, but there are instances of the vertices of a level not belonging to the same category. A case in point is that the CTO does not belong to the same staff category as the CEO's personal assistant.

Hierarchies relate to dimensions in the following way: Each dimension can have zero or more hierarchies and a hierarchical level can be in multiple hierarchies in the same dimension. In other words, we could conceivably have two or more hierarchies in the time dimension that all had a *Month* level.

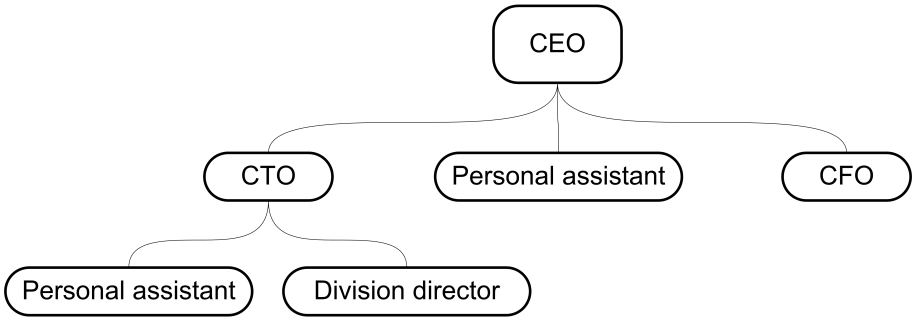


Figure 2.3: Value based hierarchy.

2.4.6 Hypercube

The *hypercube* is the central concept of the multi-dimensional model. We create a hypercube, often shortened to cube, by picking one or more dimensions we want to examine our facts in relation to. While we tend to regard it as a three dimensional thing the dimensionality of the cube is not limited.

Figure 2.4 shows a hypercube view of hypothetical data from the running example tables, viewing aggregations of the *amount* in the *Sales* table in the *Product*, *Time* and *Location* dimensions.

2.4.7 Multi-Dimensional View

When a user browses a hypercube, he is working with a *multi-dimensional view* of the underlying, persistent, hypercube. The users browsing operations do not modify the hypercube, only his view of it.

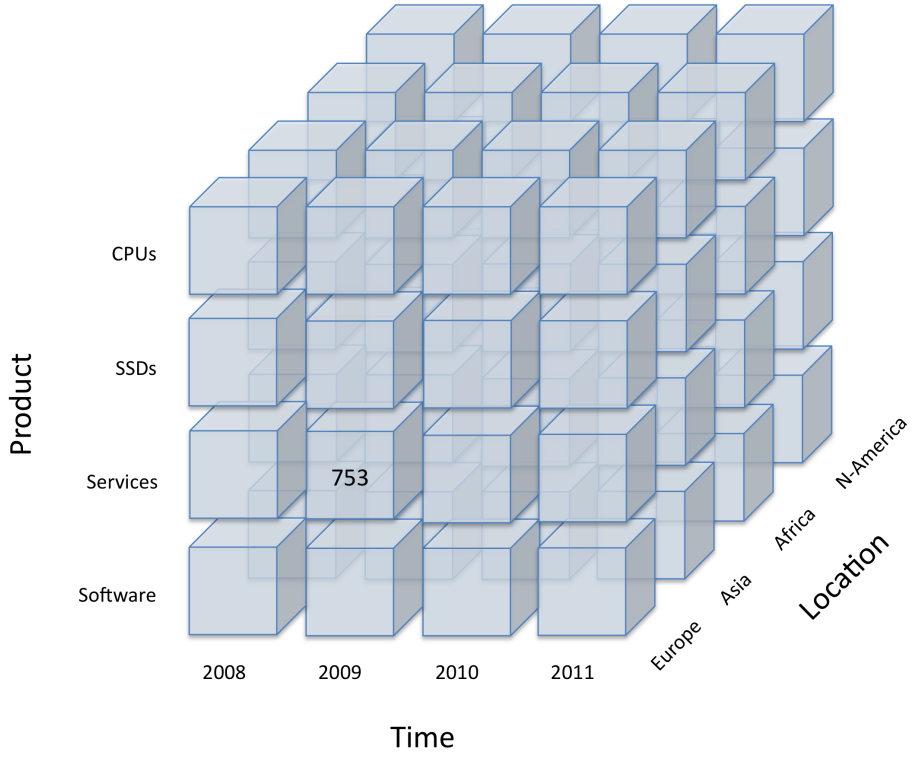


Figure 2.4: Hypercube.

2.4.8 Cell & Measure

At the intersection of a single value from each of the dimensions making up the cube, is a *cell*. All the small cubes in Figure 2.4 are cells. If we use that cube and pick *Services* from the *Product* dimension, *2009* from the *Time* dimension and finally *Europe* from the *Location* dimension, it yields us the cell containing the number 753. That value, and the value of every other cell, is the *measure*. A measure is a numeric fact that has a set of members, for which the dimensions provide further information, associated with it. Another way to describe what a measure is, would be as an aggregation of the facts in the fact table, in our case the amount, defined by the dimensions. In the case described here the total sales amount for services in Europe for the year 2009 is 753, here 753 is the measure.

2.4.9 Drill Down

Drilling down is the act of going from a more generic level in a hierarchy to a more specific one, or more exactly moving from a parent vertex, in a hierarchy of a dimension, to a child vertex. It can also be described as moving from a more aggregated view of the data to a more detailed one. Given the definition of drilling down, it should come as no surprise that having a hierarchy within the dimension we are interested in drilling down in is a prerequisite.

Figure 2.5 shows the multi-dimensional view, of the underlying hypercube, resulting from drilling down in the *Time* hierarchy, from Figure 2.2, of the *Time* dimension and selecting the vertex representing the year 2009.

2.4.10 Roll Up

Rolling up is the exact opposite of drilling down. It is moving from a child vertex in a hierarchy to its parent. A more generic way of describing it is moving from a more detailed view of the data to a more aggregated one. Starting with the multi-dimensional view in Figure 2.5 and rolling up in the *Time* hierarchy of the *Time* dimension would net us the multi-dimensional view (cube) shown in Figure 2.4, the multi-dimensional view as it was before drilling down.

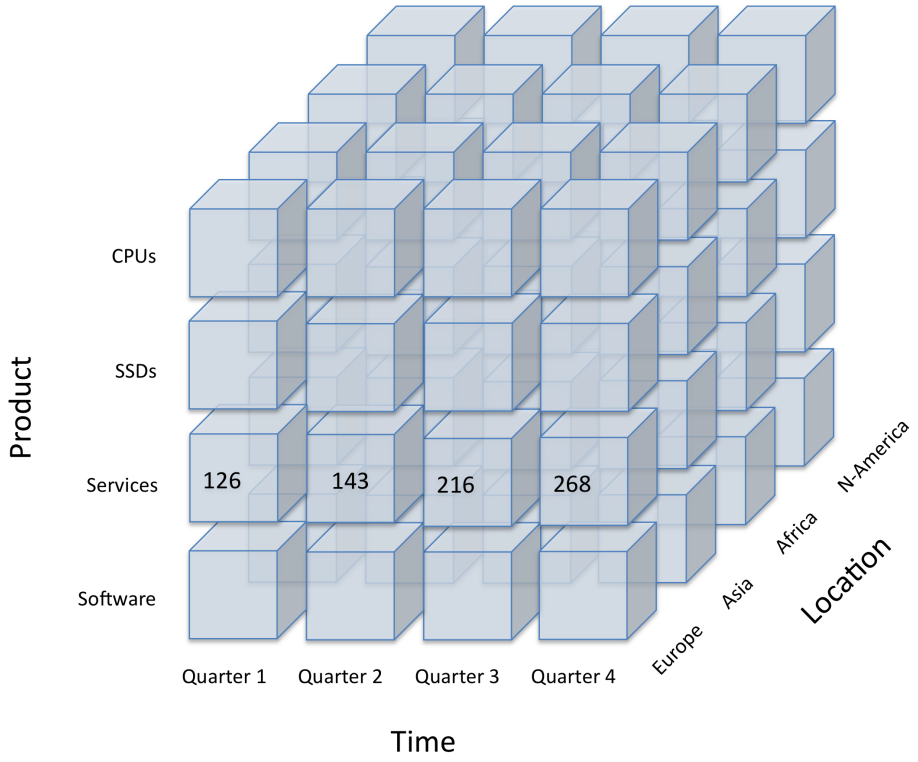


Figure 2.5: Multi-Dimensional view - Drill down.

2.4.11 Slicing

Slicing has the following formal definition: *"A slice is a subset of a multi-dimensional array corresponding to a single value for one or more members of the dimensions not in the subset."* (Council, 1995). Another way of describing a slice, is by selecting a single value for one or more dimensions of a cube.

Figure 2.6 shows the slice defined by choosing *SSD's* in the *Product* dimension, wishing only to see the sales for *SSD's*.

Slicing in a single dimension reduces the dimensionality of the multi-dimensional view by one, as the dimension used for slicing is no longer a dimension in the cube. Slicing can, in other words, be used to extract a two dimensional page from a cube. If the cube has more than three dimensions we must slice in more than one dimension to get a two dimensional page.

Figure 2.7 shows the page sliced from the cube; notice that there is no longer a *Product* dimension in it.

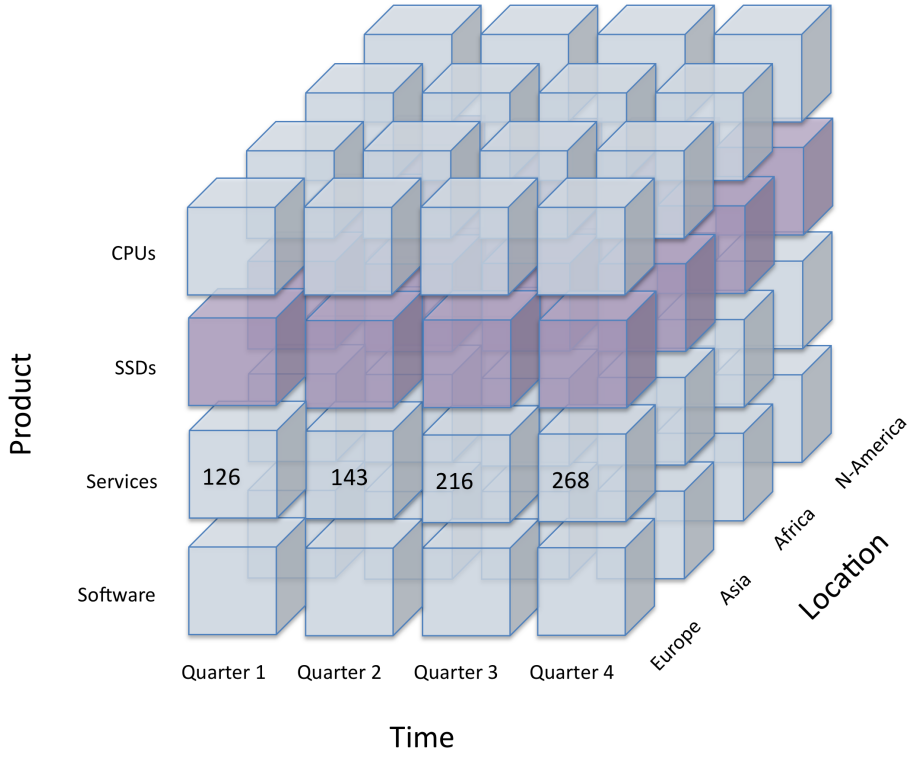


Figure 2.6: Multi-Dimensional view - Slicing.

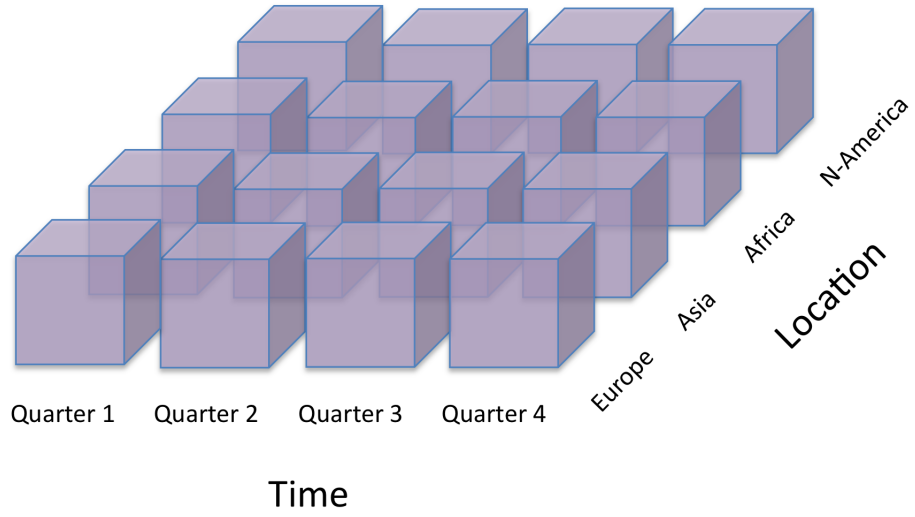


Figure 2.7: Multi-Dimensional view - A slice.

2.4.12 Dicing

Dicing is the act of defining a sub-cube of a cube by performing a slice in more than two dimensions, or performing more than two consecutive slices.

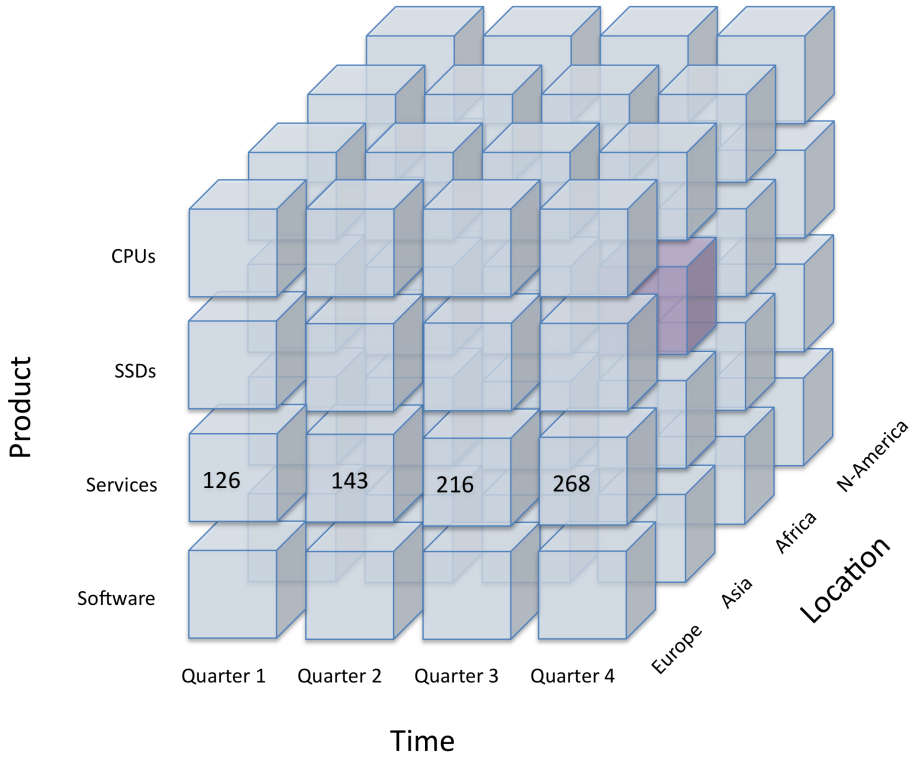


Figure 2.8: Multi-Dimensional view - Dicing.

Figure 2.8 shows the result of dicing the cube shown in Figure 2.5, selecting *SSD's* in the *Product* dimension, *Quarter 4* in the *Time* dimension and *Asia* in the *Location* dimension.

Figure 2.9 shows the result of, again using hypothetical data, starting with the die cube in Figure 2.8 and then drilling down in all three dimensions, using their hierarchies.

2.4.13 Pivoting

Pivoting is the act of choosing dimensions, or hierarchical levels within a dimension, for each of the axis of the cube, of the multi-dimensional view, and thereby reordering the cube. It can be done by either reordering the dimensions already in the cube or exchanging a dimension in the cube for one that is not. Pivoting is also known as rotating.

Thus far we have put the *Product* dimension on the y-axis, the *Time* dimension on the x-axis and the *Location* dimension on the z-axis. Figure 2.10 shows the result of pivoting the cube shown in Figure 2.5 by exchanging the orientation of the *Location* and *Product* dimensions, putting *Product* on the z-axis and *Location* on the y-axis.

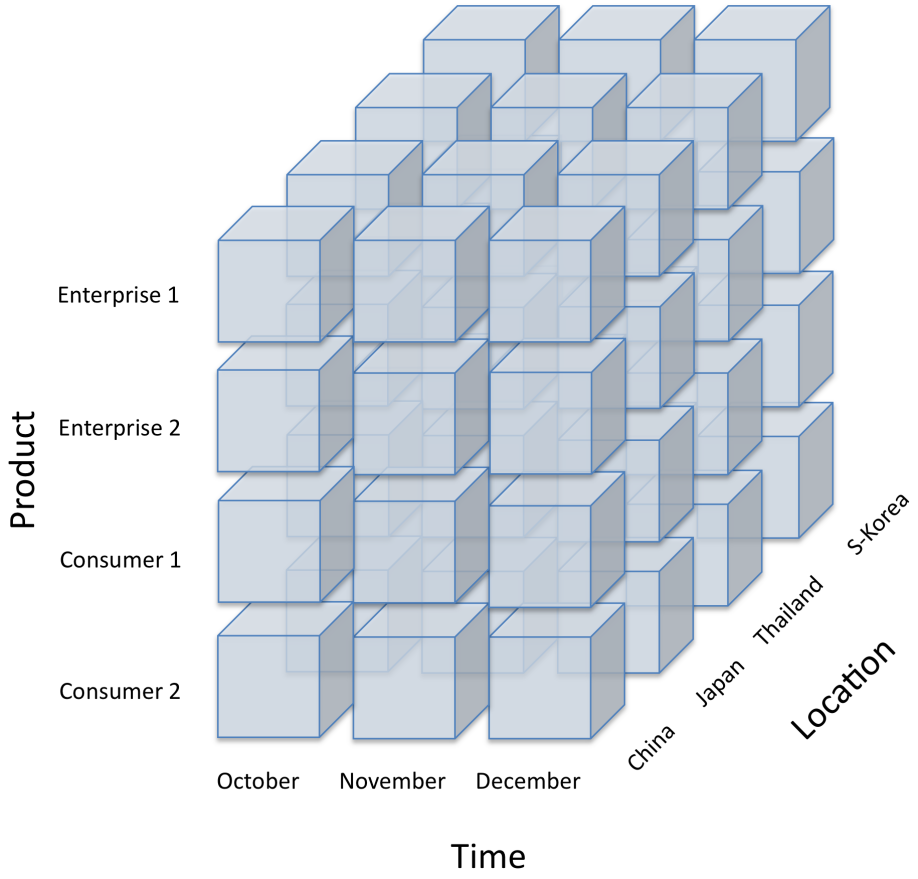


Figure 2.9: Multi-Dimensional view - Dicing & Drilling down.

Notice that the measures for sales of services in Europe have moved in the cube due to the pivoting.

A common use of pivoting is slicing a cube and then pivoting that slice to present it as a two dimensional page. Figure 2.11 shows the result of pivoting the slice shown in Figure 2.7 by putting *Location* on the y-axis in order to present the slice as a two dimensional page.

2.4.14 Page Dimension

As we have seen, we restrict what information is in the multi-dimensional view by choosing dimensions for the axis of the cube. Only information with the relevant members from the dimensions being used will be shown. Another way of restricting the information is setting one or more *page dimensions*. A page dimension is a dimension that is not shown on an axis of the cube but for which a value has been chosen; it thus restricts what information is in the cube. Using a page dimension is equivalent to slicing in a dimension that is not shown in the cube. If we had a *Sales Type* dimension an example would be selecting the *Online* member value as a page dimension. The dimensions of the cube would be the same

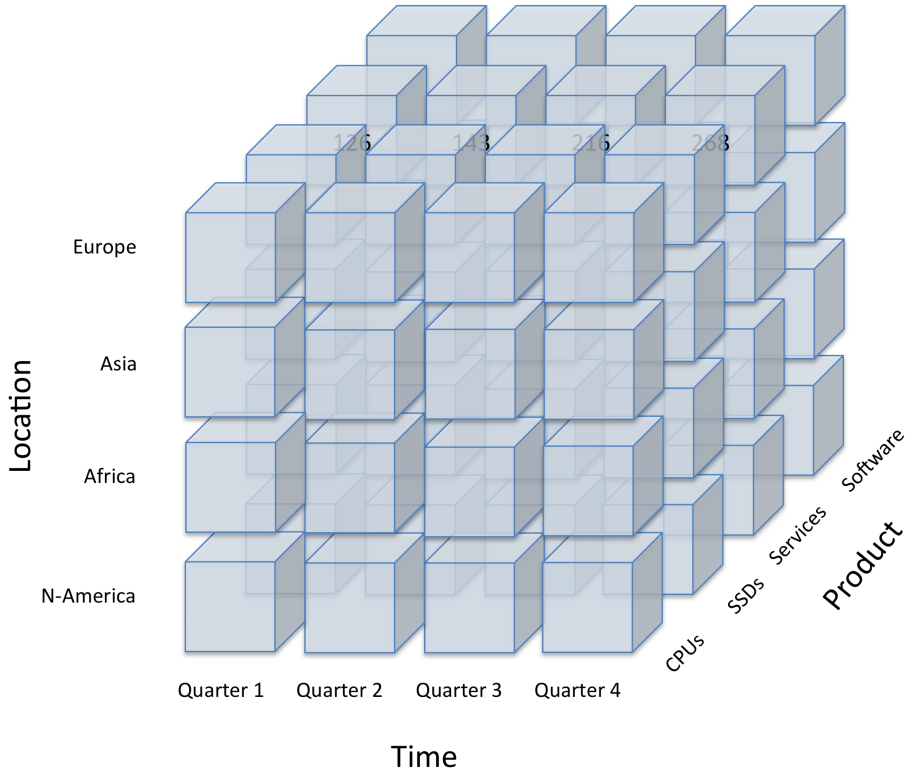


Figure 2.10: Multi-Dimensional view - Pivoting.

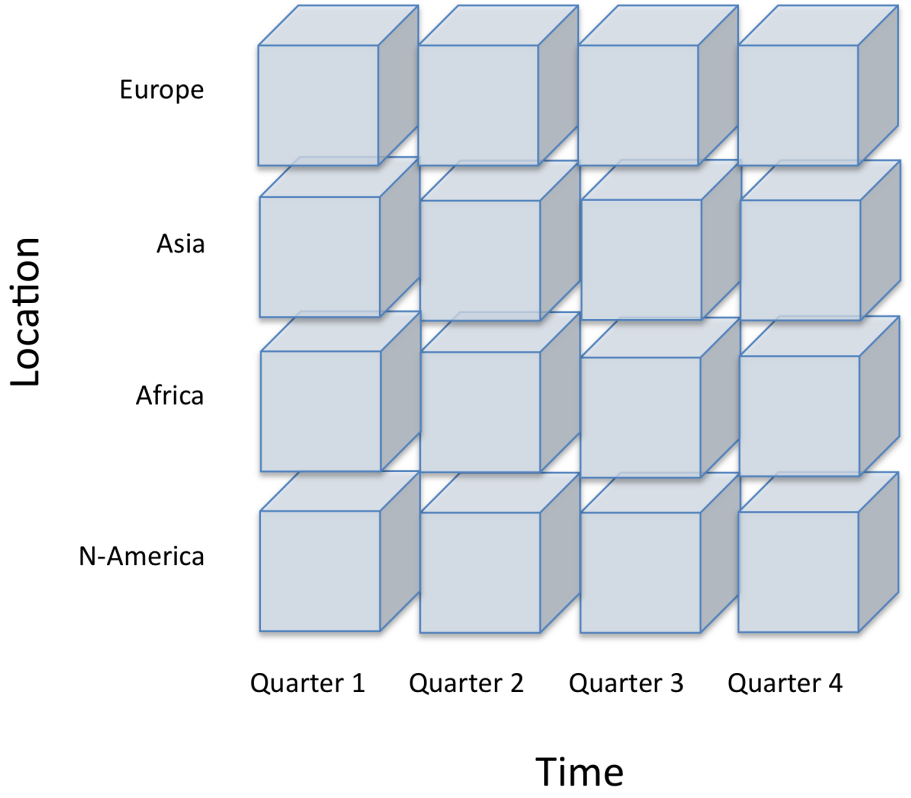


Figure 2.11: Multi-Dimensional view - Slicing & Pivoting.

but the data would not, presuming there were any online sales within the set defined by the dimensions of the cube.

2.4.15 Selection

Another way of restricting the information selection is defining a criterion to filter the data, this is known as *selection*. An example would be to only retrieve data for the top 5 products by sales amount, or to only retrieve data for products that weigh more than 5 kilograms.

2.5 Summary

We have looked at the origins of multi-dimensional analysis and mentioned its most common usage in Business Intelligence. We have also discussed the controversy regarding the definition of a multi-dimensional analysis system and the fact that there is more of an agreement on the concepts of the multi-dimensional view and how it can be manipulated. Finally we have set up a running example and examined the key concepts of multi-dimensional analysis in its context.

Chapter 3

The ObjectCube Model

The concepts of multi-dimensional analysis have been successfully used in various OLAP products to view and analyze numerical data. As our goal is to create a generic multi-dimensional analysis solution for non-numerical data, the concepts of multi-dimensional analysis are a natural starting point for the definition of the conceptual model we needed.

While the concepts of our model are based on multi-dimensional analysis concepts they are not exactly identical.

In Section 3.1 we discuss the core concepts of our model. In Section 3.2 we look at the concepts that make up the multi-dimensional element in ObjectCube. In Section 3.3 we examine concepts having to do with retrieval. In Section 3.4 we analyze the correspondence between the concepts of ObjectCube and those of the multi-dimensional model, after which we summarize in Section 3.5.

3.1 Core Concepts

In this section we describe the two core concepts of *objects* and *tags*, which apply to the media items themselves, independent of any potential browsing scenarios.

3.1.1 Object

An object is any entity that a user is interested in storing information about. In many cases it will be a file of some sort and, while it can be any type of file, the most obvious candidate may be a binary file since we cannot easily search those for relevant information.

3.1.2 Tag

A tag is any meta-data that can be associated with objects. There is no limitation on how many objects a tag can be associated with; one tag could conceivably be associated with no objects, while another one is associated with all objects in the system. There are also no limitations on how many tags can be associated with a single object; one object could have no tags associated with it, while another object has all tags in the system associated with it. The relationship between objects and tags is thus many to many. Figure 3.1 shows an example with four objects and a few tags in the center. In this example we have both an object with no tags associated with it and a tag associated with no objects.

3.2 Multi-Dimensional Concepts

In this section we define four concepts that concern categorization and grouping of data, and are at the heart of the multi-dimensional nature of our model. They are the *tag-set*, *hierarchy*— and its variations —, *hypercube* and *cell*.

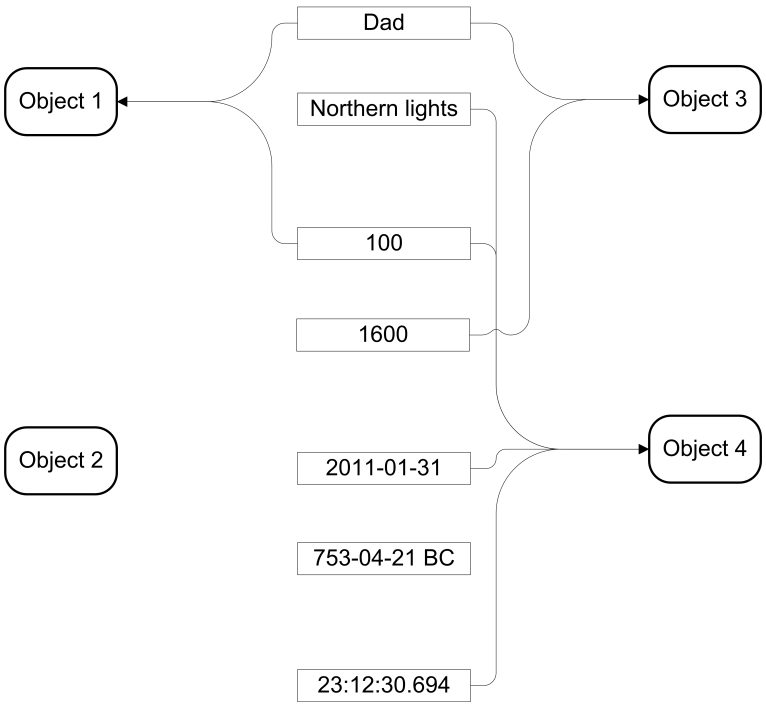


Figure 3.1: Objects & Tags.

3.2.1 Tag-Set

A tag-set is a set of tags that are in some way cohesive, the user perceives them to be similar; we can think of the tag-set as a category of sorts. In a tag-set instance named ‘People’ the tags can, for example, be the names of people or the names of subcategories such as ‘Family’ or ‘Friends’. Tag-sets are mathematical sets as the elements (tags) are distinct and the order of the elements is irrelevant. There is no explicit or implied order of the tags of a tag-set.

Figure 3.2 shows a tag-set instance example named ‘People’, consisting of tags representing the names of individuals and groups.

Note that this example includes both the names of individuals and subcategories (of people) like ‘Family’. Each and every tag in a tag-set can be associated with objects, including subcategories. An example would be tagging an image with the tag ‘Family’ from the ‘People’ tag-set shown in Figure 3.2.

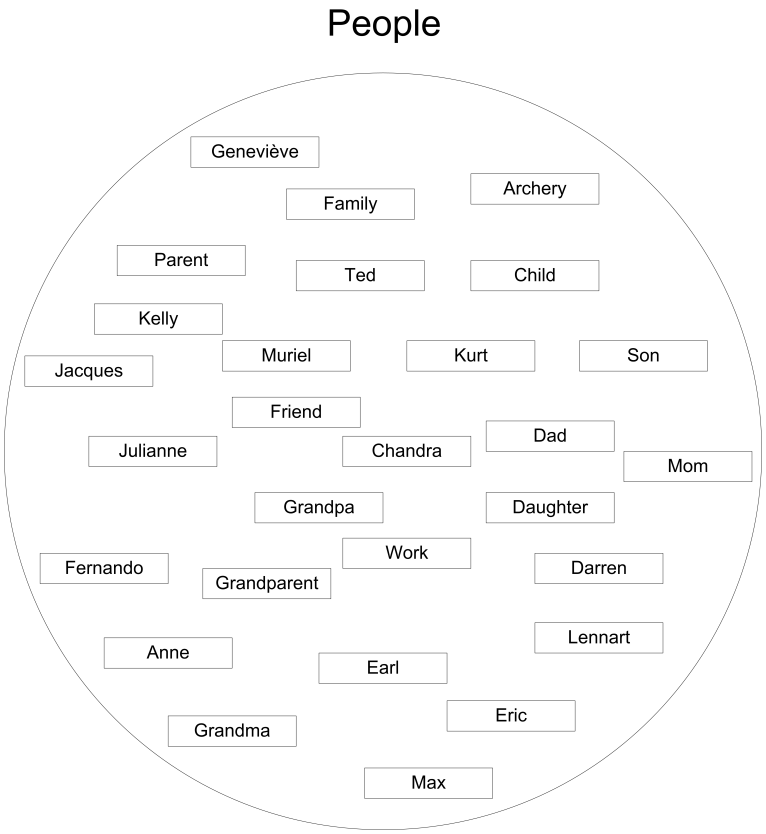


Figure 3.2: The ‘People’ tag-set.

3.2.2 Hierarchy

A hierarchy is a tree structure that adds structure and order to a non-strict subset of the tags of a tag-set. More informally, it serves to categorize some of the tags of a single tag-set. A hierarchy is derived from a single tag-set and only contains tags from that tag-set. Each tag-set, however, may have zero, one, or more hierarchies derived from it and its hierarchies belong to it in much the same way as the tags.

While the tags of a tag-set are not ordered, the vertices of a hierarchy are explicitly ordered. Hierarchies can therefore be used to enforce the order of tags and create a tag sequence. Think, for example, of the months of the year in an instance of a tag-set of dates, we want them listed in their natural order.

A vertex in a hierarchy may optionally have a title that applies to its children, a *child category title*. The children (vertices below the chosen vertex in the hierarchy) are then instances of the category that the title names. To use the example above, 'Month' could be the child category title while the months themselves are the children. What has been said so far, about hierarchies, applies to all variants of hierarchies. There are two variants, *tag-based hierarchies* and *tag property-based hierarchies*.

Tag-Based Hierarchies

A tag-based hierarchy is a hierarchy where each vertex in the hierarchy encapsulates a single tag in the tag-set. It defines a parent-child relationship between a subset of the tags of a single tag-set. Figure 3.3 shows an example of two tag-based hierarchies in the 'People' tag-set shown in Figure 3.2.

The tag-based hierarchies only allow the same tag to appear once in each hierarchy. A single tag can, however, appear in multiple hierarchies of the same tag-set. The implication is that, were we to create a 'Friends' hierarchy in the 'People' tag-set, including both 'Related' and 'Workplace' vertices, we cannot add the same person (tag) to both, even if that person both works with us and is related to us.

Note that this limitation is due to an early design decision of keeping things simple. The hierarchies are rooted trees and each child has but one parent. In the general case, relaxing that requirement, e.g., by allowing rooted DAGs, is quite complex. Allowing the same leaf (tag) to appear in multiple subtrees of the same hierarchy is, however, much less complex and certainly a viable approach.

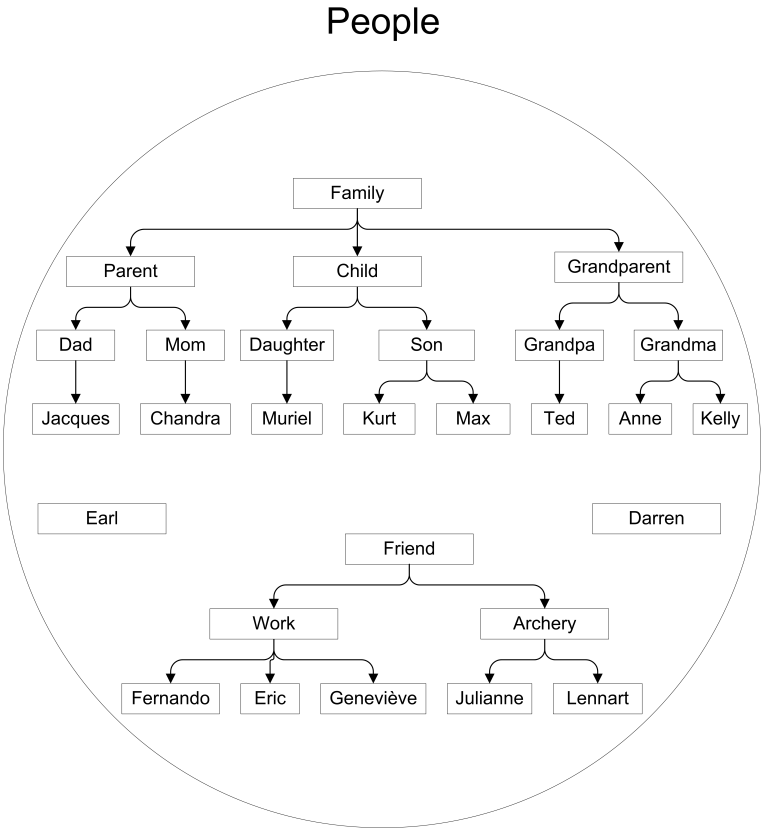


Figure 3.3: Tag-based hierarchies.

Tag Property-Based Hierarchies

Each tag property-based hierarchy is based around a single property of the tags in the tag-set the hierarchy is derived from. That property is represented by the root of the hierarchy. An example of a property might be the month part of a tag representing a date. The second level in a tag property-based hierarchy denotes the values of the property the hierarchy is based around. In the case of the month example it might be the number for each month, from 1 to 12. The values in the second level are only the values we are interested in, not necessarily the whole range. And finally, the third level in the hierarchy contains the tags of the hierarchy that have a value of the property equal to that of their parent vertex. For our month example, and a second level vertex with the value 4, the child vertices would be all tags in the tag-set with April as their month, regardless of year and day of month. There are always exactly three levels, the root, the property value level and finally the tag level. The first two levels, the root and the property value level do not represent tags, as the vertices in them do not need to have a tag representation in the tag-set. A tag property-based hierarchy checks all the tags, of the tag-set it is derived from, for its property, and if they have it, the value of the property. Each tag can only have a

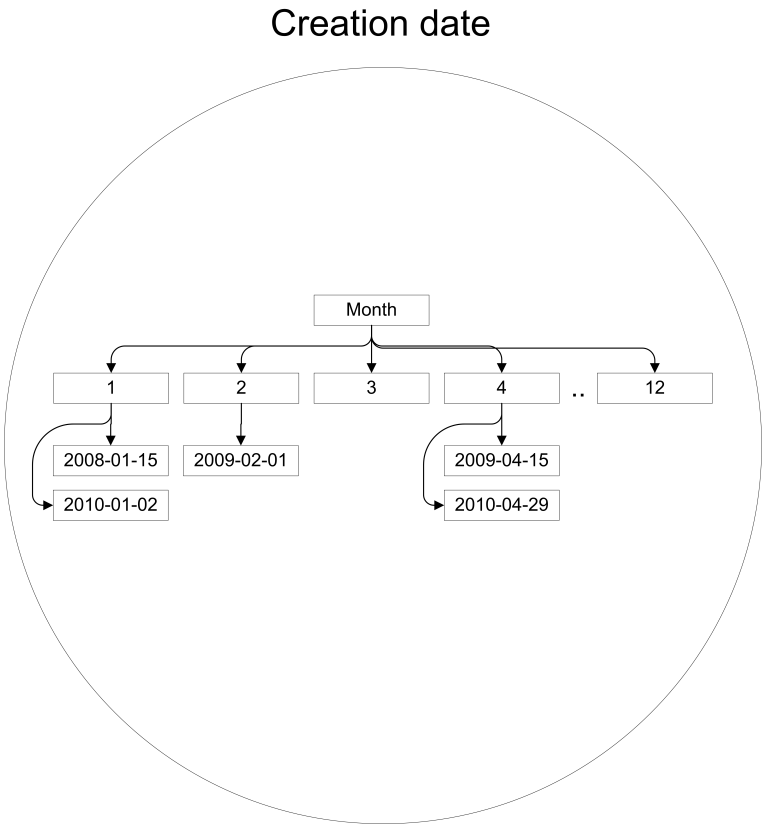


Figure 3.4: A tag property-based hierarchy.

single value for the property of a single tag-property based hierarchy and can therefore only appear once in each tag property-based hierarchy.

A characterizing attribute of the tag property-based hierarchies is that they can be used to group together tags that would belong to a number of sub-trees in a tag-based hierarchy. Consider for example a date hierarchy with multiple years, all the months for each year, and then the actual date tags beneath. A tag property-based month hierarchy like we have been describing would contain, as children of the same month value vertex (second level), tags from multiple sub-trees of the tag-based hierarchy.

Figure 3.4 shows the month tag-property based hierarchy we have been using as an example. In the hierarchy shown in Figure 3.4 we have a tag-set that only contains five tags, given that they all have the property and the property value level covers all values from 1 to 12, all the tags in the tag-set are in the month tag property-based hierarchy.

Purpose of Hierarchy Variants

The two variants of hierarchies serve two different purposes. The tag-based hierarchy serves the purpose of a general multi level categorization, from the most general to the most

specific, of tags. A tag-based hierarchy only considers the whole tag, never it's properties. The tag property-based hierarchy on the other hand serves the purpose of a specific property driven categorization where the hierarchy is based around a single property and the values of that property we are interested in. Tag property-based hierarchies only consider a tags property value, not the whole tag. While a property might be defined to look at whole tags, the hierarchies created would be fairly useless since each value level vertex would be the aggregation of at most a single tag in the tag-set.

A Formal Definition of Hierarchies

With a more formal definition, the tag-based hierarchy is a rooted, directed, labeled, ordered (strict total order) tree. The tree is rooted, since the hierarchy is derived from a single tag and has the natural direction of a parent-child relationship. The definition of a vertex-labeled graph is as follows: "Normally, the vertices of a graph, by their nature as elements of a set, are distinguishable. This kind of graph may be called vertex-labeled." (Misc., 2010a). Since each vertex in a tag-based hierarchy represents a tag in a tag-set it fulfills this requirement. The definition of an ordered tree is as follows: "An ordered tree is a rooted tree for which an ordering is specified for the children of each vertex." (Misc., 2010b). The tree is ordered since each vertex has a fixed location in the graph. In tag-based hierarchies the direction of the edges defines a parent child relationship between the connected vertices, which eliminates the possibility of there being any cycles in the graph. A requirement for a graph being a tree is that it does not contain any cycles.

The tag property-based hierarchy is a rooted, directed and ordered (strict total order) tree. It is rooted since it is derived from a single property and it has the natural direction of a parent-child relationship. It is ordered and a tree for the very same reasons as the tag-based hierarchy. Unlike the tag-based hierarchy, the tag property-based hierarchy is not vertex-labeled since a part of the vertices are not elements of the set.

3.2.3 Hypercube

The *hypercube* is the combination of all the tag-sets, tags and objects in the model. It has as many dimensions as there are tag-sets. The number of tags in a tag-set define the cell count for that dimension.

3.2.4 Cell

A *cell* in the hypercube is defined by the intersection of a single tag, or tag property in the case of tag property-based hierarchies, from each of the tag-sets in the hypercube. A cell can contain zero or more objects.

3.3 Retrieval Concepts

So far, we have defined concepts that describe and organize objects, but now we turn to the retrieval of the objects. Unlike typical search applications, the retrieval is based on browsing, where the retrieved object sets are defined incrementally, based on the users interest. In the ObjectCube model, different *filters* can be applied to the various tag-sets or hierarchies, resulting in a *browsing state* (hereafter shortened to *state*).

3.3.1 Filter

A filter is a constraint describing a sub-set of objects that the user wishes to retrieve. Each filter applies to a single tag-set, but many filters may be applied to the same tag-set.

An object passes through a filter if it has one, or more, tags associated with it that pass through the filter.

Note that a filter can be applied to any dimension — tag-set or hierarchy — regardless of whether that dimension is being shown in a user interface or not. Furthermore, a filter continues to restrict the retrieval until it is explicitly revoked by the user. A browsing session thus consists of applying filters and retrieving the objects that pass through all the applied filters.

There are three different filter variants: *tag filter*, *range filter*, and *hierarchical filter*.

Tag Filter

The tag filter is a filter that encapsulates a single tag, which must exist in the tag-set being filtered. It is used to retrieve only objects associated with that particular tag.

An example could be a filter on the tag ‘Muriel’ in the ‘People’ tag-set, shown in Figure 3.2. All objects who have the tag associated with them pass through this filter.

Range Filter

A range filter defines a value range by two boundary values, where both boundary values are included in the range. The boundary values themselves do not need to exist as tags in the tag-set the filter is being applied to. Since a filter only applies to a single tag-set, only tags from that tag-set can be considered to be within the range. A range filter is used to retrieve only objects that have at least one tag associated with them, that falls within the range of the filter.

An example would be a filter on the dates 2009-01-01 and 2009-06-30 in the 'Creation date' tag-set, shown in Figure 3.4. All objects created during the first half of 2009, and thus tagged, pass through this filter.

Hierarchical Filter

A hierarchical filter encapsulates a single vertex in a hierarchy. All objects associated with one or more tags in the sub-hierarchy, with the chosen vertex as its root, are said to pass through the filter.

An example would be a hierarchical filter encapsulating the 'Child' tag in the 'Family' hierarchy of the 'People' tag-set, shown in Figure 3.2. All objects associated with any tags from the sub-tree of 'Child' ('Child', 'Daughter', 'Son', 'Muriel', 'Kurt', 'Max') pass through this filter.

3.3.2 State

As mentioned earlier, a browsing session consists of applying filters and retrieving the objects that pass through all the applied filters. The state is a, transient, multi-dimensional view of the, persistent, underlying hypercube, defined by the set of filters in effect. Another way to look at it is as the set of objects and associated meta-data that corresponds to the current state of the filters.

Notice that the ObjectCube model does not define what operators are applied between filters — union, intersection etc. — while it might be added to future versions it is currently left to implementation. There is also no notion of scope.

Each filter only applies to a single tag-set, therefore the state of a single tag-set is the set of objects and associated meta-datum that corresponds to the state of its filters.

The state of the whole system is the combination of the states of those of its tag-sets with any filters in effect. To clarify, if any filters are in effect in the system, tag-sets with no filters in effect are not included in the combination to get the state of the system. Otherwise, only objects with tags from all tag-sets would ever be included in the state. If no filters are in effect, any state retrieved will include all objects in the system.

The state contains the following data[1]:

- **Objects**
  All objects that pass through the combination of filters in effect are included. It is sufficient that one tag that the object is associated with passes through a filter, and different tags may pass through different filters.
- **Tags**
  For each object in the state all its tags get included, with the tags resulting in inclusion being marked out. Each tag resulting in inclusion knows which filters it passed through.
- **Hierarchies**
  All hierarhies used for hierarchical filtering are included. In other words, for each of the hierarchies in the state there are one or more hierarchical filters, encapsulating one of its vertices, in effect.

  The hierarchy of a state is a mirror image of the tag-based or tag-property based hierarchy it represents, but in addition it knows all the objects, from the object set of the state, associated with each tag encapsulated by a vertex of the hierarchy. A state hierarchy contains all the vertices of the hierarchy it represents irrespective of whether a vertex, through its tag, is associated with any objects in the state.

The state can present its objects in two ways. First, as a multi-dimensional view of the underlying hypercube, where the user picks tag-sets or hierarchies for each dimension. Second, as a list of objects.

3.4 Comparison of Models

Figure 3.1 summarizes the concepts of ObjectCube and corresponding concepts from multi-dimensional analysis. We now discuss that correspondence or lack thereof.

[1] It should also contain tag-sets due to range filtering, future work.

3.4.1 Analysis of Correspondence

Object vs. Fact

In both cases they are the information we are interested in analyzing and both have meta-data associated with them to describe them further. Unlike numerical facts, however, objects are typically quite complex, making operations such as aggregation very difficult.

ObjectCube	**Multi-dimensional analysis**	**Correspondence**	**Details**
Object	Fact	Approximate	3.4.1 [Object vs. Fact]
Tag	Member	High	3.4.1 [Tag vs. Member]
Tag-set	Dimension	High	3.4.1 [Tag-set vs. Dimension]
Hierarchy	Hierarchy	High	3.4.1 [Hierarchy vs. Hierarchy]
Hypercube	Hypercube	High	3.4.1 [Hypercube vs. Hypercube]
Cell	Cell	Approximate	ObjectCube cells contain zero or more objects, not an aggregation of facts.
Filter	Page dimension	Approximate	3.4.1 [Filter]
Filter	Selection	Approximate	3.4.1 [Filter]
State	Multi-dimensional view	Approximate	3.4.1 [State vs. Multi-Dimensional View]
-	Attribute	None	3.4.2 [Attribute]
-	Measure	None	3.4.2 [Measure]
-	Drill down	Indirect	
-	Roll up	Indirect	All of those are achievable
-	Slicing	Indirect	using our concepts, see 3.4.2
-	Dicing	Indirect	[Operations on a State]
-	Pivoting	Indirect	

Table 3.1: ObjectCube and multi-dimensional analysis concept correspondence.

Tag vs. Member

Both are associated with the objects/facts, both provide additional information about them. A member title, that gets represented as a column name in the source table in multi-dimensional analysis, is represented as a tag, or a child category title in a hierarchy, in ObjectCube. The instances of a member get represented as tags too. The tag and member concepts are, in other words, similar.

Tag-Set vs. Dimension

In both cases, the user perceives the members/tags of the set as being strongly related to each other, and in both cases hierarchies can be constructed to organize their contents.

Hierarchy vs. Hierarchy

The hierarchy concepts are highly similar: Both can represent either level or value based hierarchies, and in both cases a vertex is the aggregation of its children. For the multi-dimensional analysis model, aggregation is typically based on mathematical functions, such as sum or average, while in the ObjectCube, aggregation takes some form of grouping.

A minor difference between the two models is that in multi-dimensional analysis, a column name supplies a level name in a hierarchy, but in ObjectCube there is a *child category title* that only applies to the children of that vertex, not the entire level.

While the tag property-based hierarchy is a special case in our concepts, it is a part of the hierarchy of the multi-dimensional analysis. Using properties such as the month of a date is well known in Business Intelligence, and can be done by adding a column, for the property, into the appropriate look-up table.

Hypercube vs. Hypercube

In multi-dimensional analysis, the hypercube is typically stored in a data structure specifically designed for efficient access. This data structure stores both base facts and pre-calculated aggregations of facts for higher levels of the hierarchies. Due to the differences between objects and facts, however, only base data can be stored for the ObjectCube model.

Filter

The filter concept corresponds to two multi-dimensional analysis concepts: Selection and page dimension. First, it serves the same purpose as selection, both can be used to define a filter to restrict the data retrieved. Second, it corresponds to the page dimension concept in the way that both can be used to restrict the data being retrieved through a dimension not in the state. The filter is a more general concept than either the selection or the page dimension.

State vs. Multi-Dimensional View

The state differs from the multi-dimensional view, of multi-dimensional analysis, in that it contains all the objects per each cell, not aggregations, and each object stores its tags (members). It contains more data, objects, and meta-data then the multi-dimensional view.

3.4.2 Lack of Correspondence

There are some concepts, covered in Chapter 2, that do not have a corresponding concept in ObjectCube.

Attribute

The multi-dimensional analysis attribute is not represented directly. In multi-dimensional analysis, an attribute is used to provide additional information about a fact. In ObjectCube we use additional tag-sets and tags to do this. It is both more flexible and powerful, as each 'attribute' tag-set can be used as a dimension in a state.

Measure

A measure is a numerical value, that is an aggregation of all the facts within a cell. The ObjectCube model does not, currently, support the aggregation of objects. Supporting that is one possible avenue of future work.

Operations on a State

Operations on a cube are only relevant in the context of whether the same results are achievable or not using the ObjectCube model. Drill down can be achieved by removing a filter on a vertex, in a hierarchy, and creating a filter on a child vertex. Conversely, roll-up is achieved by removing a filter on a child vertex, and putting a filter on its parent instead. Slicing is achieved by creating a tag filter on a tag in a tag-set being used as a dimension in a hypercube. Dicing is achieved by creating tag filters on more than two tag-sets, being used as dimensions in a state. Pivoting is achieved by picking tag-sets or hierarchies for the hypercube of the state.

Notice that all operations requiring addition or removal of filters require the retrieval of a new, current, state to show the changed state of the model.

3.5 Summary

In this section we have defined the generic multi-dimensional model, ObjectCube, and contrasted it with the well-known multi-dimensional analysis model. All the major differences stem from the fact that the multi-dimensional analysis model is applied to simple facts, which are typically numerical data items, while the ObjectCube model is applied to complex media objects which cannot be easily manipulated or aggregated. In the next chapter we present a prototype software implementation of ObjectCube.

Chapter 4

Prototype Architecture

In this chapter we present the prototype software implementation of the ObjectCube model. In Section 4.1, we discuss key design features of the prototype. In Section 4.2, we present the overall architecture. In Section 4.3, we discuss entities stored persistently, and those that are not in Section 4.4. In Section 4.5 we discuss the plug-in architecture. In Section 4.6, we discuss implementation details, and we summarize in Section 4.7.

4.1 Design Features

We set out to implement the ObjectCube model in software and thereby to create the back-end of a media browser. In this section we discuss the key design features of the prototype. They can be split into three broad categories; usability, flexibility, and availability.

4.1.1 Usability

In this category are features that should make our software better meet the end user needs.

- **It is stateful.**
  This is a single user browsing solution; being able to incrementally add and remove filters to refine the image set viewed enhances the user experience.
- **Automated tagging of objects is supported.**
  In order to automate tagging, we support plug-ins; limiting the user workload when adding objects.

- **It is efficient.**
  ObjectCube is written in C++, and while the focus is on architecture (e.g. layering, dependency management and ease of maintenance) it is also written with performance in mind. This enables optimizations using low level code where needed. Among optimizations performed, is caching a part of ObjectCube's data to deliver the state efficiently, and linking objects to the vertices of state hierarchies. The goal is the limitation of delays.

- **Tags are typed.**
  This is done for two reasons. The first is that it enables us to make efficient use of the capabilities of the data stores used when filtering, especially for ranged filtering. The second is that it enables type-checking of tags; e.g. checking that a date tag contains a valid date.

- **Tag-sets are typed.**
  This is done to help end users create tag-sets that have a high level of cohesion between their tags. Consider, for example, the 'People' tag set shown in Figure 3.2. This tag-set should include alphanumerical string tags only, and date tags should be excluded.

4.1.2 Flexibility

In this category are features that should make our software more flexible, and better able to support diverse usage scenarios.

- **Automated tagging of objects is supported through a flexible plug-in architecture.**
  In order to be both data type invariant and to enable easy addition of automated tagging functionality, ObjectCube has a flexible plug-in architecture that supports plug-ins through a well defined interface.

- **It is storage layer invariant.**
  It is unclear which data storage architecture is best suited to the ObjectCube model. This is vital for experimentation with different data storages, using different architectures, as they can be exchanged in isolation.

  The three data storage architectures that most interest us are: multi-dimensional data storage — storing predefined hypercubes and aggregations —, column-stores — storing the columns of a relational table contiguously —, and finally row-stores — storing the rows of a relational table contiguously —. The first of those is one

possible, and highly interesting, avenue for future work. For the latter two, we have implementations.

While the data stores used are relational databases, there is nothing in the storage layer interface that requires that. Note, ObjectCube also suports multiple storage layers in the same binary.

- **Dual interface.**
  The prototype supports both a C++ interface and a Python interface that mirrors the C++ one. This way, users of the software can write user interfaces at a suitable level.

4.1.3 Availability

In this category are features that aim to maximize the availability of the software.

- **It is multi-platform.**
  Currently, ObjectCube runs on OS X and Linux. There are plans to port it to Windows.
- **It is suitable for open source licensing.**
  Aside from the data stores used, of which one is open source and another public domain, there are no third party code or software dependencies in the core C++ solution.

  The Python interface depends on Boost.Python, an open-source C++ library. This limitation of third party dependence of any sort also helps in porting ObjectCube between platforms.
- **It is multi-lingual**
  The prototype only supports Icelandic and English, but adding languages is simple.

Having discussed the key design features we now examine the overall architecture.

4.2 Overall Architecture

Figure 4.1 shows the overall architecture of the ObjectCube prototype, including plug-ins and data stores.

Before discussing the purpose of each software layer it should be noted, as Figure 4.1 shows, that strictly speaking the plug-ins are not a part of the core ObjectCube software. They will, however, be a part of most practical applications of it.

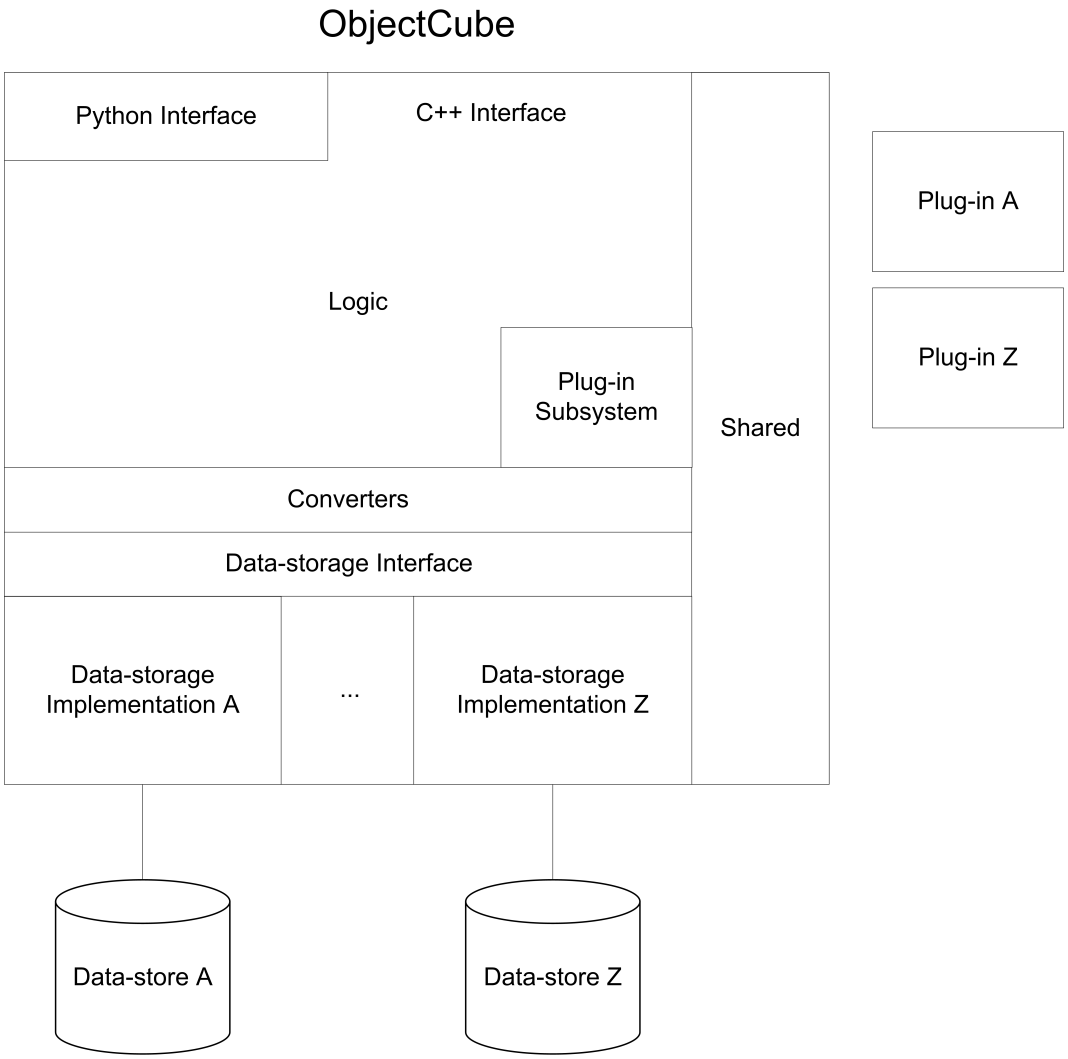


Figure 4.1: Architectural overview.

Shared

The sole purpose of this layer is to limit dependencies between different layers of the system. The shared layer includes utility functions, error management, memory management, a logger, handling of input parameters, enumerations and the like.

Logic

This is the key layer of the prototype, which contains the implementations of all the concepts of the ObjectCube model. During a browsing session, the logic layer is responsible for maintaining and returning the state. All caching is done in this layer.

The public interface of all the classes in this layer make up the C++ interface of ObjectCube.

Python Interface

The Python interface layer is a thin wrapper on top of the Logic layer, using Boost Python. It mirrors the interface portion of the Logic layer. Those two interfaces are equivalent.

Converters

The purpose of this layers is to contain all data conversion code between the *Logic* layer and the *Data Storage Interface* layer in a single place, thereby avoiding duplication of the conversion code, isolating the *Logic* layer better from the *Data Storage Interface* layer, and making the code of the *Logic* layer cleaner.

Data Storage Interface

This is an abstract interface implemented by all data storage implementations. It isolates the *Logic* layer from the actual data storage implementations.

Data Storage Implementation

Each data storage implementation implements the abstract interface of the data storage interface for a single data store. Also, a part of each data storage implementation is converting the set of filters in effect into a query suitable for that data store, upon state retrieval.

Plug-in Subsystem

The Plug-in-subsystem loads, manages and isolates plug-ins from the rest of the layers. It is covered in some detail in Section 4.5.

Plug-ins

Each plug-in performs data specific analysis on an object and returns tags. While only two plug-ins are drawn there is no limit on their number.

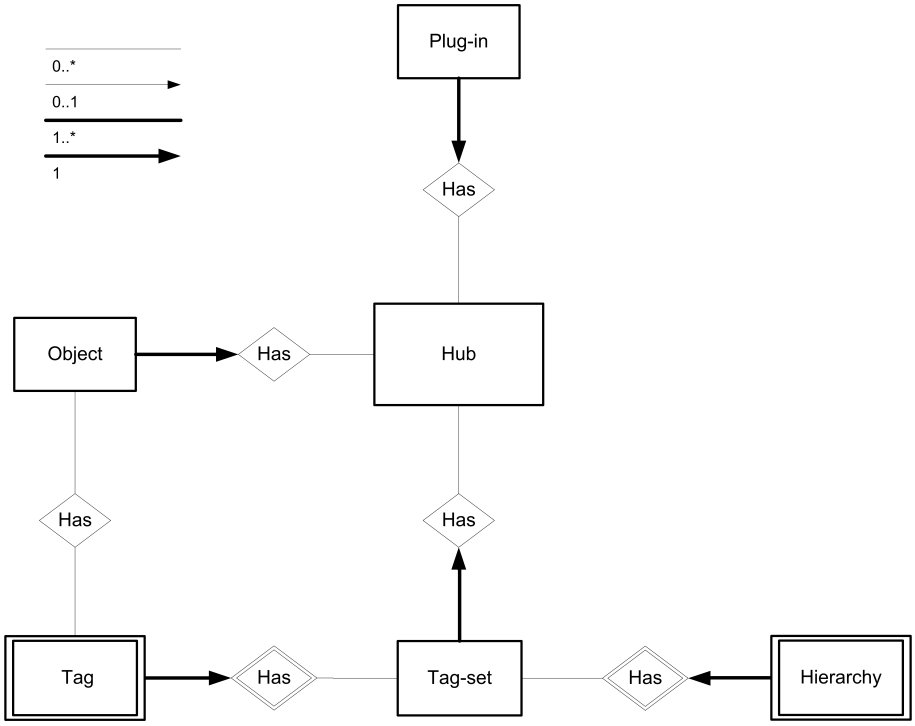


Figure 4.2: Persistent entities.

4.3 Persistent Data

Having examined the key concepts of our multi-dimensional model, we now present that model as an entity diagram.

Please notice that the entity diagrams presented in this chapter have been simplified to give a clear high level view of the model. In our implementation, there are multiple types of tags, tag-sets, hierarchies and filters. To keep the focus on how the entities related to each other, the structure of our model, attributes of entities are also left out.

In this section, we discuss the part of ObjectCubes data that is persistently stored, using an entity diagram. We also discuss the supported types of tags and tag-sets.

4.3.1 Model

Figure 4.2 shows all ObjectCube entities which are stored in a persistent data store. Aside from the ‘Plug-in’ and ‘Hub’, the entities in Figure 4.2 represent the concepts of the ObjectCube model. The hub is a construct that serves as the central part of our software solution.

4.3.2 Supported Types

As mentioned in Section 4.1.1 tags and tag-sets are typed. We now discuss the types that are supported.

Tag Types

As the focus is on a proof of concept rather than a large feature set, the prototype only implements tag types for the most common data types. The current set is:

- **Alphanumerical Tag**
  This is the most generic tag, which consists of a single string of letters, potentially including numbers and special characters, such as punctuation marks and white-spaces.
- **Numerical Tag**
  The numerical tag consists of a single integer. It is useful for anything from street numbers to light sensitivity.
- **Time Tag**
  The time tag is data store and system independent. It consists of four integers, one for each of hours, minutes, seconds and milliseconds. The allowed value ranges are 0..23 for the hour, 0..59 for the minutes and seconds and 0..999 for the milliseconds.
- **Date Tag**
  The date tag is also data store and system independent. It consists of three integers, one for each of year, month and day of month. The value range for the year is unbounded negative to unbounded positive; negative integers are used to represent dates BC. The value range for the month is 1..12 and 1..31 for the day of month.

In the current data store implementations, there is a base tag table and then a separate table for each of the supported tag types, which extends the base tag table. Note that due to the tag caching, we do not select any data from the typed tag tables when retrieving a state; they are, however, accessed for range filtering.

Tag-set Types

For each tag type, there is a corresponding tag-set type. Unlike the tags, however, this does not affect the tag-set itself, merely its members. All tag-set types are therefore stored in the same table.

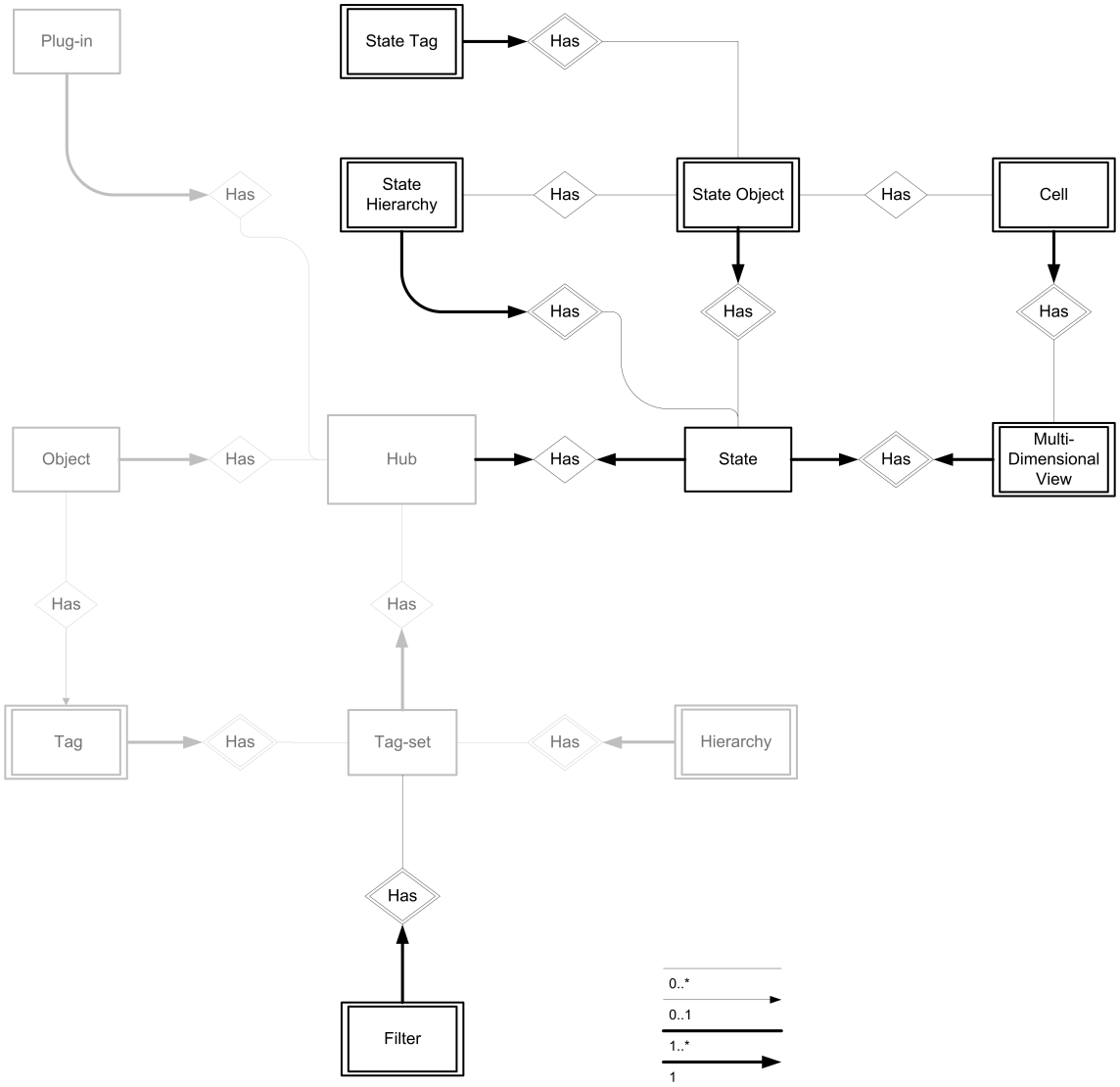


Figure 4.3: Non-persistent entities.

4.4 Logic Layer

The logic layer contains implementations of our concepts, thus embodying the ObjectCube model, including all the entities discussed in Section 4.3.1. In addition to those persistent entities, there are also entities that are not persistently stored. In this section we discuss the entities that are not persistently stored and the filter types implemented.

4.4.1 Model

Figure 4.3 shows an entity diagram for the logic layer. The shaded entities are the ones stored persistently, the others are lost when execution of the software is terminated.

The diagram shows us first and foremost that neither filters nor the state of ObjectCube are persistently stored. In addition to that it shows how the state is broken up into entities.

The state can have zero or more state objects and zero or more state hierarchies. Each state hierarchy can have zero or more state objects included in the hierarchy, by its vertices. The state can present its data in a multi-dimensional view. That view has zero or more cells and each cell contains zero or more objects. Each object can be in more than one cell. We can simplify this down to the state having two representations of its data, a single dimensional one, in a state object list, and a multi-dimensional one, in its multi-dimensional view. The state hierarchies are both used to create the multi-dimensional view and to assist in browsing it, drilling down and rolling up.

Looking at the placement of the filter in the diagram, we can see that a single filter can only apply to a single tag-set, and that a single tag-set can have zero or more filters in effect. What it does not tell us is that filters are actually typed like the tags and tag-sets. Note, the prototype only supportes intersection between filters, the state of the system is the intersection of the object sets defined by the filters in effect.

4.4.2 Typed Filters

As mentioned above, there are three filter types: tag, range, and hierarchical. These behave slightly differently with respect to tag types. Both the tag- and hierarchical filter encapsulate a single tag or a hierarchical vertex by refering to the identifier of the tag or vertex. The type of that tag or hierarchy thus has no effect on them, there are no typed implementations of those two filter types. Range filter, however, differ based on the type of the tag-set, since the range filters do not use tags to define their boundaries.

In the prototype we have implemented typed range filters for numerical-, time- and date tag-sets, using the data definitions of the respective tag type as their boundary values. We did not implement an alphanumerical range filter, as it was considered less essential than the others.

4.5 Plug-in Architecture & Plug-ins

The plug-in architecture is highly flexible and supports multiple types of plug-ins. It is an evolution of the one suggested by Markus Ewald in (Ewald, 2007).

When a new object is added to ObjectCube, each plug-in is handed the binary data of the object. The plug-in is a piece of software that performs a data-specific analysis of the object and, with our current interface, returns tags to attach to that object. ObjectCube then creates, if they do not already exist, and attaches those tags to the object.

The plug-in subsystem makes two requirements of a plug-in. First, that it implements an interface for the plug-in subsystem, so that it can be loaded and version checked. Second, that it implements one of the supported object analysis interfaces.

To clarify, a plug-in only has access to what data ObjectCube gives it as a part of the object analysis interface it implements. The plug-ins do not have any access to either ObjectCube's internal data nor its data stores. Similarily, ObjectCube does not have any access to, or knowledge of, databases kept by plug-ins. The object analysis interfaces are the only means of passing data between ObjectCube and its plug-ins.

While there is no limit on the number of plug-ins that can be attached to the prototype, each plug-in increases the time it takes to add an object. This puts a practical limit on the number of plug-ins attached.

We foresee there being plug-ins implementing different interfaces. In the case of face recognition we would, for example, most likely need to support a bounding box in the interface. This flexibility is important in enabling us to further extend the system without undue hardship.

For our image browser we have created two plug-ins. One extracts about twenty different categories of EXIF (exchangeable image file format; a format for storing meta-data in image files) meta-data, such as date, time, ISO rating, and exposure time. The other plug-in extracts basic color information (average levels of red, green and blue).

4.6 Implementation Details

The ObjectCube prototype is written completely in C++ and makes extensive use of its standard library. It also makes use of the TR1 C++ library extensions. The current implementation, excluding plug-ins but including unit tests, consists of approximately forty thousand lines of C++ code.

As of now, the prototype runs, and has been tested, on Mac OS X 10.6 and Ubuntu Linux 10.04. As mentioned earlier, there are plans to port it to Windows 7. For both OS X and Ubuntu, the GNU GCC compiler was used to compile it, using the same GNU Make

makefile. We used GNU GCC version 4.2.1 on OS X and version 4.4.3 on Ubuntu. The Python interface uses Python 2.6 and is created using Boost.Python (version 2).

We make extensive use of automated unit tests, and have developed unit tests for all the functionality of the C++ interface. This is simply best practice; and with a highly integrated solution it is imperative that the code can be modified with a modicum of certainty that the solution does work as intended. For unit tests we use CppUnit, and SVN for source control. Finally we have data store implementations for the three following relational databases:

- **SQLite** - A public domain row-store that claims to be "the most widely deployed SQL database engine in the world." (SQLite, 2010).

- **MonetDB** - An open-source column-store from CWI (Centrum Wiskunde & Informatica) in Amsterdam that is known for its focus on performance (Manegold, Boncz, & Kersten, 2000; Zukowski, Boncz, Nes, & Héman, 2005; MonetDB, 2008).

- **System X** - A widely used commercial row-store.

4.7 Summary

In this chapter, we have presented the prototype implementation of ObjectCube. We have discussed its key design features — focusing on usability, flexibility, and availability — and analyzed its overall architecture, discussing each of its layers in turn. We have also looked at an entity diagram representation of ObjectCube, focusing on persistent and non-persistent entities. Additionally, we have discussed why some entities are typed, and the types supported by the ObjectCube prototype. In the next chapter we evaluate the performance of the prototype.

Chapter 5

Evaluation

In this chapter we evaluate the performance of the prototype by running three experiments, one for each filter type, designed as a "stress-test" to uncover any scalability or query complexity weaknesses of either the prototype or underlying data stores.

We ran the experiments using each of the three data stores the prototype supports. It is of interest to see how the prototype performs using different data stores, especially due to the different architectures, row-store vs. column-store, given that the workload is somewhat analytical in nature. A very large percentage of the data is processed in some of the queries. In addition we evaluate the performance overhead of the prototype; what it adds to the time taken by the data stores.

While the performance of every operation of the system is of importance, the time it takes to retrieve the state of the system is both the most time consuming and perhaps most important operation of the system. In this performance evaluation we focus solely on that.

In Section 5.1, we discuss the experiment setup and methodology. In Sections 5.2 to 5.4 we examine the performance of state retrieval using, in turn, tag-filtering, range-filtering and hierarchical-filtering. In Section 5.5 we examine the performance overhead of ObjectCube and in Section 5.6 we summarize.

5.1 Experimental Setup

The following setup was used in all the experiments.

5.1.1 Experimental Platform

The experiments were run on a Dell Precision T5400 Workstation with the following characteristics:

- Two Intel E5430 quad core CPU's running at 2.66GHz, each CPU having a 12MB L2 Cache.
- 4GB of memory.
- 500GB, 7200 RPM hard drive using an ATA interface.

We formatted the hard drive of the workstation used for the experiments, and installed a 32bit Ubuntu 10.04 Linux operating system using kernel version 2.6.32.24-generic. It was not modified in any way, settings or otherwise. We also used the default file-system, Ext4. Notice that since this is a 32bit operating system the usable memory was down to 3.2GB, from the 4GB installed.

The experiments were run using a virtual machine. This allowed us to use a single installation with all the relevant software and data stores, regardless of the underlying hardware. A loss of approximately 15% of CPU performance and 30% of I/O performance (Domingues, Araujo, & Silva, 2009) from native performance does not affect our conclusions significantly, as our goal is the comparison of different data stores and an analysis of the overhead of the prototype. The virtual machine used for the experiments was created using VMWare Fusion 3.1, on Mac OS X, and run using VMWare Player version 3.1.2. It ran Ubuntu 10.04, fully up to date, using kernel version 2.6.32.25-generic. The virtual machine was allocated 4 processors and 2684MB of memory, the maximum recommended by VMWare Player. It had 4GB of free hard drive space and before any experiments were run we ran the virtual hard drive defragmentation in VMWare Player.

5.1.2 Software

In the experiments, the prototype used the following data stores:

- SQLite, version 3.7.1
- MonetDB, version 1.38 June 2010 SP2
- System X, most recent Enterprise version.

Those are the most recent version of each database for which there was an Aptitude package or an installer available at the time of the experiments. In all cases, we ran the databases as

they were, out of the box, with no tuning of any kind performed. The databases have a table structure, which it is identical for all three, except for minor variance in type support. All tables are thoroughly indexed and all databases have identical indexes.

SQLite and System X are classical relational databases where entire rows are stored contiguously on disk. In contrast, MonetDB is a column-store for which each column is stored contiguously, with rows being scattered. Column-stores have been shown to outperform row-stores for analytical workloads where a large number of rows, and often only a few columns, are processed. This is, among other things, due to column-stores being I/O and compression friendly (Abadi, Madden, & Hachem, 2008), and CPU cache friendly (Manegold et al., 2000; Zukowski, Heman, Nes, & Boncz, 2006).

The queries the data stores execute in the experiments process a large amount of rows but they also process all the columns of the tables queried and return a large number of rows. Processing all columns lessens the I/O advantage of column-stores and returning a large number of rows is not advantageous to column-stores due to tuple-reconstruction cost (Idreos, Kersten, & Manegold, 2009).

5.1.3 Data Sets

In order to examine the scalability we used four data sets of images downloaded from Flick'r, each one an order of magnitude larger than the other. The properties of the image collections are shown in Table 5.1.

The largest of those data sets was created by adding one million images to ObjectCube through its Python interface. The bulk of the tags and image/tag associations in this set are automatically generated by the EXIF plug-in; of the individual tags in the largest data set about 85% are different time tags. Each of the smaller sets was then created from a correctly sized sub-set of the largest set.

Images	**Tags**	**Image/Tag associations**	**Average tags per image**
1,000	3,252	13,905	13.90
10,000	20,819	136,768	13.68
100,000	113,185	1,350,198	13.50
1,000,000	201,277	13,621,005	13.62

Table 5.1: Experimental data sets statistics.

Having created correctly sized sets we then created tag-sets, tags and a hierarchy, as required by the experiments, for each data set. That data was also added using the Python interface of ObjectCube. We give details of this additional data in the discussion of each relevant experiment. Each of the data sets contains the data needed for all three experiments.

The selectivity is the percentage of objects retrieved from the data store. Note that the number of rows processed can be approximated by multiplying the number of objects retrieved by the *average tags per image* for the relevant data set, shown in Table 5.1.

There is a separate database for each data set in both SQLite and MonetDB. In System X there is a schema for each data set.

Examples of queries for all filter types can be seen in Appendix A.

5.1.4 Methods

As mentioned earlier the experiments focus on the retrieval of the state using all filter types. We have chosen to ignore the time to retrieve the images themselves, as that time is independent of the method used to retrieve the state.

We are interested in all aspects of scalability, i.e., in terms of data set size, the selectivity of the retrieved state, and the complexity of the queries required to retrieve the state. As we observe, different data stores react differently to different scalability aspects.

Each experiment was run for all the data sets, from the smallest to the largest, using a single data store. Before running an experiment for a single data store, the workstation (and hence the virtual machine) was restarted. We did not perform a restart when going between different sized data sets using the same data store. Before running an experiment using a certain sized data set, we issued SQL queries that loaded all the data from the relevant tables into memory[1].

Each experiment was run twice on each data set. The first run gives an indication of how the prototype performs at the start of a browsing session, and the second run gives an indication of how it performs during continuous browsing. The second run was executed approximately a minute after the first one.

All the experiments use a varying selectivity and when running them, we always go from the lowest to the highest.

[1] Note that we did not load the tagging table for the largest data set, as it took a prohibitive amount of time on some of the data stores.

Only the data store being used was run on the virtual machine and nothing but VMWare Player ran on the workstation, which was not connected to a network to avoid interference. All experiments were run using Python scripts that use the Python interface of ObjectCube.

5.1.5 Measures

We focus on the time, measured at a granularity of milliseconds, to retrieve a state. All time measurements are made using wall clock time. In all cases, we consider retrieval time below one second to be satisfactory. In the case of a state retrieval taking more than 90 seconds, the runs were aborted.

When considering scalability with respect to data set size, we compare the scaling of individual data stores to linear scaling. For each data store, we assign a value of 100% to the time required for the 1,000 image data set. Then we scale the retrieval time for each of the other data sets with the time for the 1,000 image data set and the relative size of the data set. If the system scales linearly, this will result in a straight and horizontal line, while sub-linear scaling results in a downward slope.

5.2 Experiment I: Tag Filtering

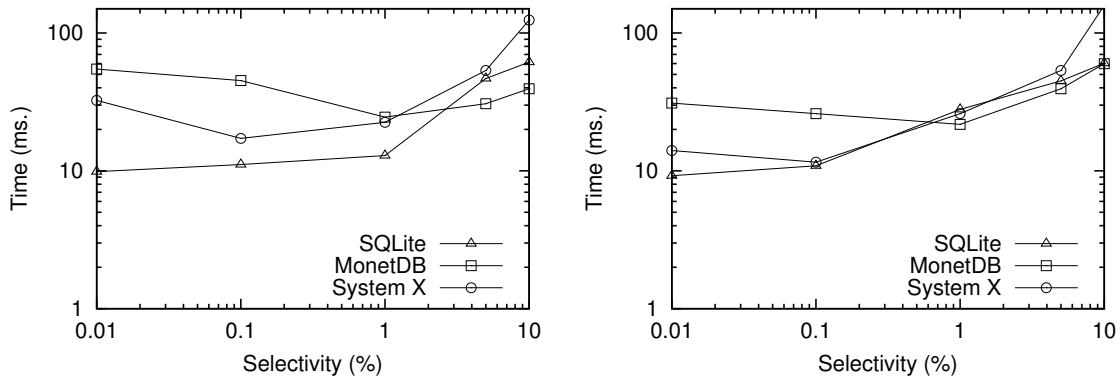
In this experiment, a single tag filter is used to retrieve all images that have the tag encapsulated by the tag filter associated with them.

5.2.1 Preparation

In order to perform this experiment we created five tags with a varying level of selectivity, as shown in Table 5.2. The underlying query to the data store, however, is independent of the selectivity. See Appendix A for details.

We tagged images randomly in order to create a worst case scenario, so there are no dense clusters of tagged images. Before running this experiment, on each data store and data set, we ran the following two queries to enable the data stores and operating system to cache at least some of the data.

```
select * from object
select * from object_tag
```



(a) First run (b) Second run

Figure 5.1: State retrieval using a tag filter and a 1,000 image set.

For tag filtered state retrieval, those are the only two tables accessed. Note, as mentioned before, the second query was not executed for the largest data set, as it took a prohibitive amount of time on some data stores. We do not query any tag tables since tag information is added from cache.

5.2.2 Impact of Selectivity

Figure 5.1 shows the results for the first and second run using the 1,000 image data set. The x-axis shows the selectivity, percentage of objects in the data set retrieved, and the y-axis the time it took to retrieve the state in milliseconds. Note the logarithmic scale on both axes.

Recall that the first run gives an indication of how the prototype performs at the start of a browsing session, and the second how it performs during continuous browsing. Looking at the first run, the performance of the prototype is satisfactory using all three data stores. That said, it is of interest that the performance gets better in some cases as the selectivity rises, in

Selectivity (%)	**Images in retrieved state**			
0.01	0	1	10	100
0.1	1	10	100	1,000
1	10	100	1,000	10,000
5	50	500	5,000	50,000
10	100	1,000	10,000	100,000
Data set size in images	1,000	10,000	100,000	1,000,000

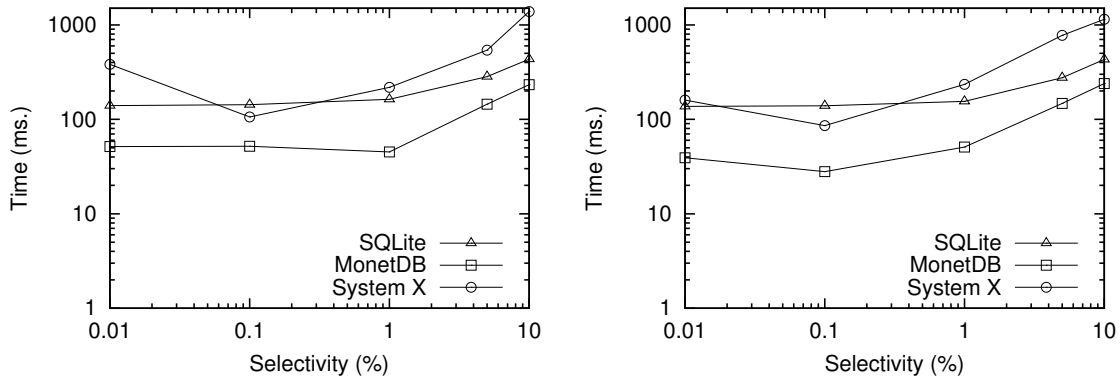
Table 5.2: Selectivity of tags in tag filtering experiment.

particular in the case of System X. This appears to be due to aggressive caching performed by System X. Those initial performance anomalies aside, there is little difference between the first and second run. This is not surprising as we are working with a small data set that is most likely all cached by the file-system.

It is also of interest that SQLite seems to have the least query time overhead, followed by System X. The fastest state retrieval time for SQLite is 9.2ms and 11.5ms for System X; by comparison the fastest time for MonetDB is 21.7ms.

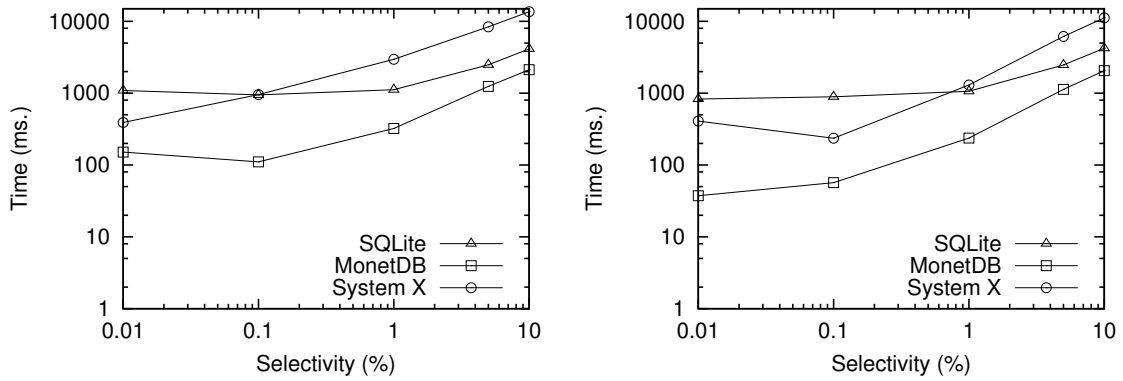
In both the first and second run System X does stand out, as it does seem to suffer worse than the other two as the selectivity rises. Results from this experiment using larger data sets support this.

Figure 5.2 shows the results for the 10,000 image data set. Its axis and scale are the same as in Figure 5.1. Looking at the first run, we see that System X again takes an initial performance hit on the first state retrieval. We also see that there is no great difference between the first and second run using this data set. What stands out here are two things: first that System X again suffers worse from an increase in selectivity than the other two, and second that MonetDB stands out performance wise. Using the smallest selectivity, state retrieval takes 28.5% (39.2ms vs. 137.3ms) of the time needed by the second fastest data store, SQLite. Using the largest selectivity, the difference is much smaller, but still MonetDB only takes 55.6% of the time needed by the second fastest data store. The performance of SQLite and MonetDB is in all cases satisfactory, using our one second limit, while the performance of System X is not for the largest selectivity. This applies to both runs.



(a) First run (b) Second run

Figure 5.2: State retrieval using a tag filter and a 10,000 image set.



(a) First run (b) Second run

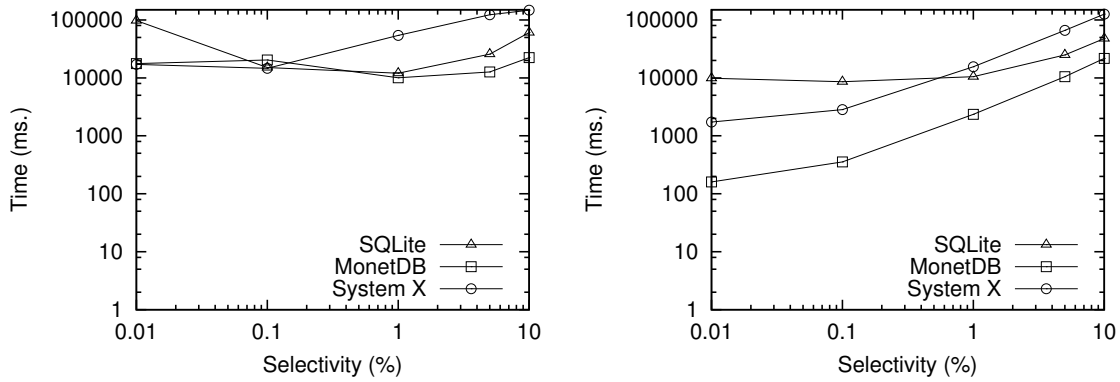
Figure 5.3: State retrieval using a tag filter and a 100,000 image set.

Figure 5.3 shows the results for the 100,000 image data set. Looking at the first run, it shows the same trends as before but with a single exception: in this case System X does not show a performance hit on the first state retrieval on the first run. The reason for this is not known, it might be due to cache size, lack of available memory or a combination of various factors. Note that it is no faster than it was using the 10,000 image set and in both cases there are 10 or fewer objects in the state.

In the case of the second run, the performance of the prototype, using a very large image collection, is acceptable up to a 0.1% selectivity using System X. That number rises to 1% for SQLite, and MonetDB only slightly passes our one second limit for a 5% selectivity (1.1 second). After that, the performance of the prototype, using all data stores, gets sub-linearly worse. As before, MonetDB performs the best, but at 10% selectivity takes 2.1 seconds to retrieve the state. It should be noted, however, that at this point the state contains 10,000 objects, which likely will present far more images than any user can digest on screen and would, in fact, require a very long time to load from disk.

Figure 5.4 shows the results for the 1,000,000 image data set. Here, the first run stands out. This is due to the fact that we could not run the second pre-experiment select as mentioned earlier. This run does tell us that SQLite is initially more affected by this and that System X's selectivity scaling problems start earlier. While interesting, the first run is not comparable to our earlier first runs, so we focus on the second run which is comparable.

What the second run shows is that SQLite does not seem to handle this amount of data well and its performance is unacceptable: the best time is 8.6 seconds on the 0.1% selectivity. System X shows a markedly better performance than SQLite for the first two selectivity percentages, after that its scaling issues regarding increased selectivity make its



(a) First run (b) Second run

Figure 5.4: State retrieval using a tag filter and a 1,000,000 image set.

performance worse than SQLite's. System X's best performance is an unacceptable 1.7 seconds for the 0.01% selectivity.

MonetDB performs the best, but it still only provides acceptable performance for the first two selectivity percentages. For a 1% selectivity it takes 2.3 seconds; at this point the state contains 10,000 objects, and as mentioned before that may not be reasonable.

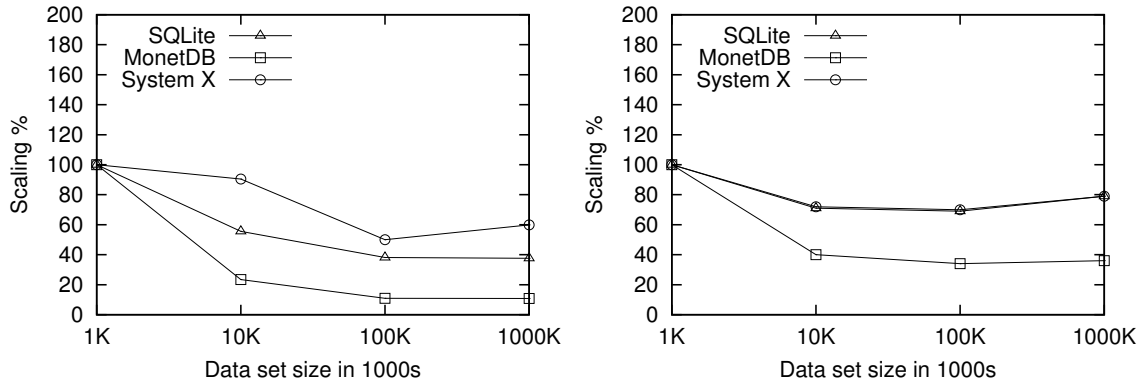
While the performance of all data stores falls below our one second mark, it should be noted that this may be, to a degree, a function of the number of objects retrieved. Using MonetDB, it takes 355ms to retrieve a state containing 1,000 objects — this represents many images to view at once. Limiting the maximum number of objects in a state is an interesting future research avenue.

5.2.3 Impact of Scaling

Figure 5.5 shows how the prototype scales with the size of the data set, for 1% and 10% selectivity. The x-axis shows the data set size, while the y-axis shows the performance relative to the 1,000 image data set, scaled by the size of the data set. Recall that, with this methodology, linear scaling would result in a horizontal line. The data used is from second runs.

Of the experiments, the queries for tag-filtering are the simplest, so this experiment puts the greatest focus on the impact of the data set size.

Looking at scaling for 1% selectivity, it is clear that all three solutions show considerably sublinear scaling, with MonetDB scaling the best. In the case of 10% selectivity, SQLite and System X show a slightly sublinear scaling, which degrades with data set size. In both



(a) 1% Selectivity (b) 10% Selectivity

Figure 5.5: Scaling of state retrieval using a tag filter and varying selectivity.

cases MonetDB shows much better scaling than the other two. Here, MonetDB's column-store architecture seems advantageous despite the fact that we are retrieving up to 1.3 million rows and tuple-reconstruction being a significant cost component for column-stores (Idreos et al., 2009).

5.3 Experiment II: Range Filtering

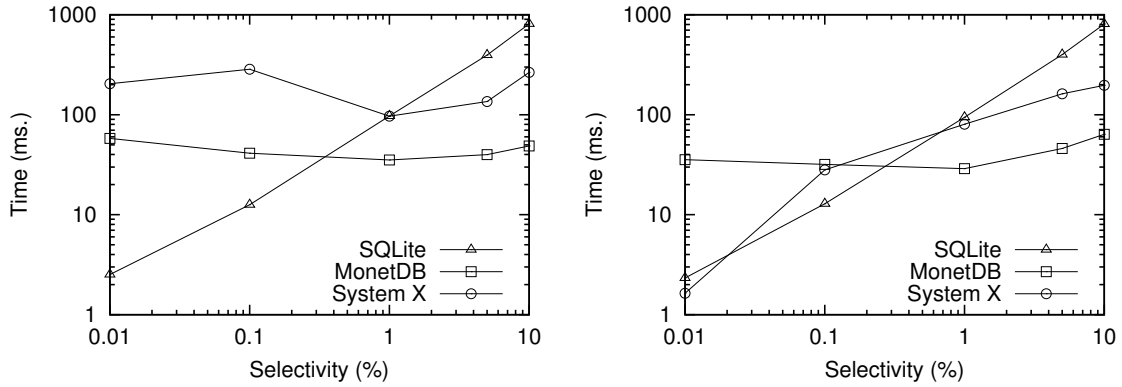
In this experiment we evaluate the performance of the prototype using a single range filter. All images associated with a tag with a value within the range of the range filter are retrieved.

5.3.1 Preparation

In this experiment we use contiguous time ranges in the tag-set representing the creation time of images, so no data was added or edited for this experiment. In the implementation for our current data stores, there is a database table for each tag type and time tags make up the largest portion; in the largest data set over 85% of the total tags are time tags.

We use the same selectivity percentages as we did in the tag filter experiment, so the information in Table 5.2 applies to this experiment too. Before running this experiment, on each data store and data set we ran the following three queries to enable the data stores and operating system to cache at least some of the data.

```
select * from object
select * from object_tag
```



(a) First run (b) Second run

Figure 5.6: State retrieval using a range filter and a 1,000 image set.

```
select * from time_tag
```

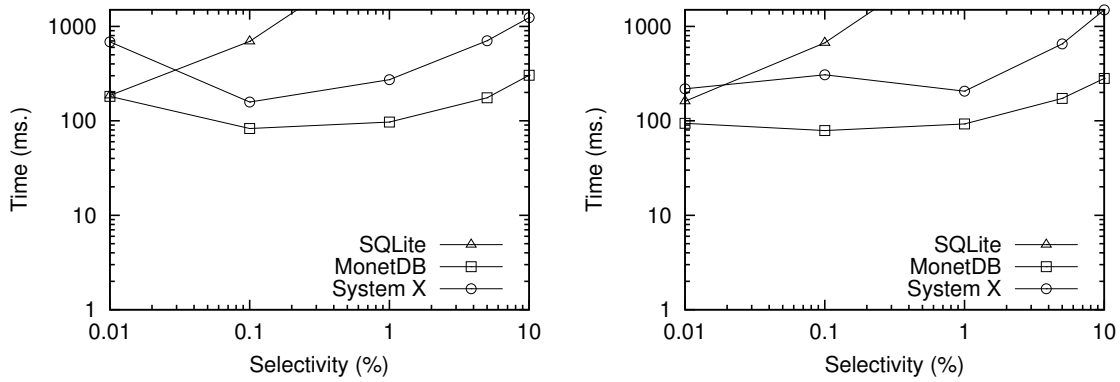
Only those three tables are accessed for a time range filtered state retrieval.

5.3.2 Impact of Selectivity

Figure 5.6 shows the results for both the first and second run using the 1,000 image data set. As before, the first run gives an indication of how the prototype performs at the start of a browsing session, and the second how it performs during continuous browsing.

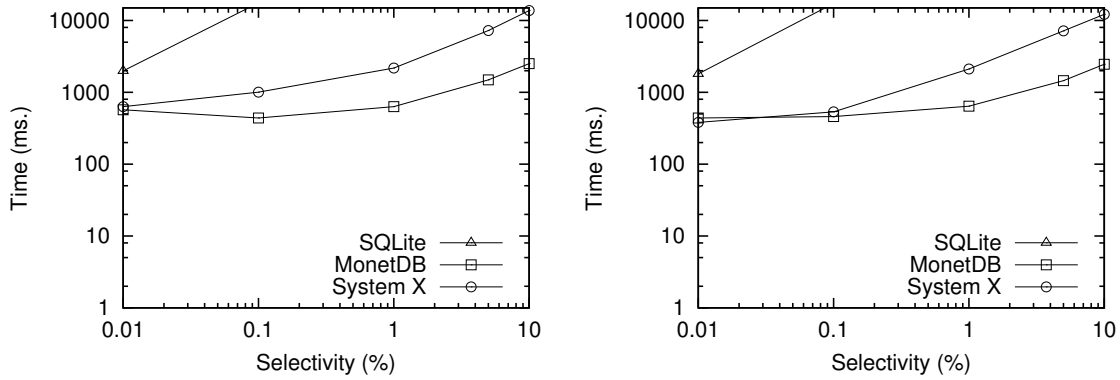
Here we can see that System X behaves in the same way as in the tag filter experiment; on the first run its performance gets better with increased selectivity. The fact that it does not show this behavior in a warm run again points to caching. This aside, what stands out is that SQLite's performance scales much worse with increased selectivity than the other two data stores. The performance disparity between System X and MonetDB is also greater using a range filter than a tag filter. MonetDB performs consistently better. That said, all data stores perform acceptably using this data set, within our one second limit. In the case of SQLite it is close, as it takes 0.8 seconds to retrieve the state for the largest selectivity.

Figure 5.7 shows the results for the 10,000 image date set. It shows us an exaggerated version of the results for the 1,000 image set. On the one hand, both MonetDB and System X perform relatively well across the range, although System X requires about two seconds for the 10% selectivity. The difference between System X and MonetDB is considerable, however, as MonetDB only needs about 20% of the time used by System X for the largest selectivities. It is clear, using MonetDB as a data store, that ObjectCube performs well



(a) First run (b) Second run

Figure 5.7: State retrieval using a range filter and a 10,000 image set.



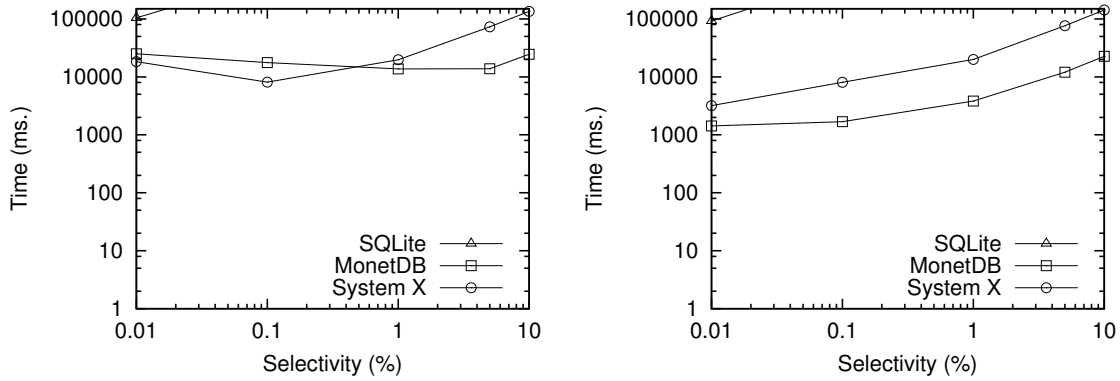
(a) First run (b) Second run

Figure 5.8: State retrieval using a range filter and a 100,000 image set.

using both tag- and range filters using a collection of 10,000 images. SQLite, on the other hand, performs significantly worse than before and is unusable above 0.1% selectivity. It takes in excess of 72 seconds to retrieve the state using 10% selectivity. There is no need to discuss the range filter performance using SQLite any further.

The major difference between the queries executed for tag- and range queries is that, in the case of a tag filter, the object/tag association table is scanned for a single value (tag), but in the case of the range filter a tag type table is scanned for multiple values, for which the object/tag association table is then scanned. SQLite seems especially ill suited to this type of a query.

Figure 5.8 shows the results for the 100,000 image data set. Looking at the first run we observe the same behavior for System X as in the tag filter experiment using the same image set. There is no visible initial performance hit. Using the same, one second, limit as



(a) First run (b) Second run

Figure 5.9: State retrieval using a range filter and a 1,000,000 image set.

before, System X is only usable up to 0.1% selectivity. MonetDB is usable for a selectivity of up to 1%.

Looking at the largest selectivity, 10%, we see that MonetDB takes 2.5 seconds to retrieve the state. The same selectivity using a tag filter, took 2.1 seconds. This can be put down to increased query complexity. System X is similarly affected. Again, the number of objects in the state at this point is beyond the point of practicality.

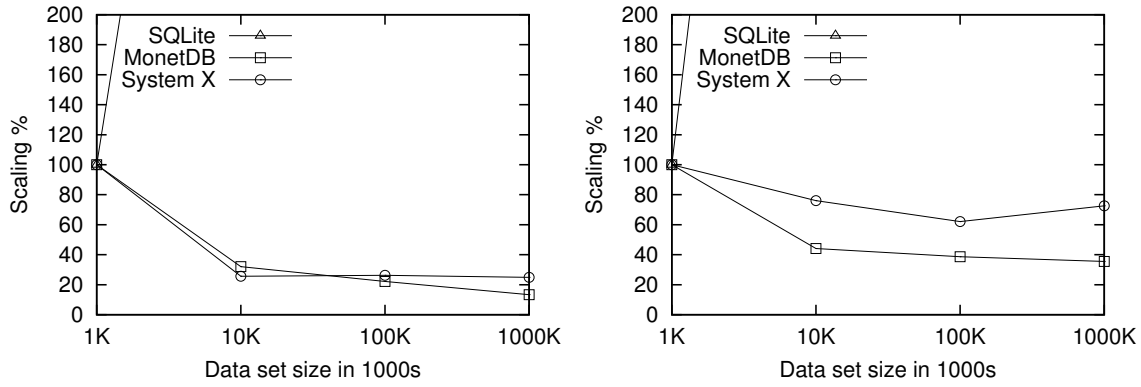
Figure 5.9 shows the results for the 1,000,000 image data set. For the first run, we see the same trend as in the tag filter experiment; all data stores take an initial hit. This is due to the same reason as before.

Even if we only look at the second run, using this very large data set, only MonetDB comes close to matching our definition of usability, and only for small selectivities. As before System X performs significantly worse for range filters, and SQLite is extremely slow, requiring more than 90 seconds even for 0.01% selectivity.

5.3.3 Impact of Scaling

Figure 5.10 shows how the prototype scales with the size of the data set. The x-axis shows the data set, while the y-axis shows the performance relative to the 1,000 image data set, scaled by the size of the data set. Recall that with this methodology, linear scaling would result in a horizontal line. We are using the data from the second run.

Given the poor performance of SQLite in this experiment, its superlinear scaling is not surprising. For 1% selectivity, MonetDB scales best, but slightly worse than it did in the



(a) 1% Selectivity (b) 10% Selectivity

Figure 5.10: Scaling of state retrieval using a range filter and varying selectivity.

tag filter experiment. System X also scales well, and even better than it did in the tag filter experiment.

If we look at scalability at 10% selectivity, both System X and MonetDB show similar or even better relative salability than in the tag filter experiment, and both are significantly sublinear. In addition, MonetDB scales about twice as well as System X. For large collections and selectivities, however, this is not sufficient for good performance, as we have seen.

5.4 Experiment III: Hierarchical Filtering

In this experiment we evaluate the performance of the prototype using a single hierarchical filter. All objects associated with a tag that is in the sub-hierarchy being filtered are retrieved.

5.4.1 Preparation

For this experiment, we wanted to create a worst case scenario for hierarchical filtering. In a worst case scenario, all the images in the system are associated with one or more tags encapsulated by the vertices in the hierarchy. This is quite relevant since it is reasonable to tag all images with a tag from a ‘Location’ tag-set, for example. The larger the hierarchy, the more complex the database queries become. With this in mind we created a tag-set, populated it with 400 different tags, and then created a four level hierarchy where the root and each child, aside from the leaves, has seven children. This leads to a hierarchy with

400 vertices, where each vertex encapsulates a tag from the tag-set. We then associated every image in each data set with a random vertex in this hierarchy. Due to this random association, the selectivity percentages vary slightly between the data sets. Please notice that the selectivity goes significantly higher than in previous experiments. Table 5.3 shows selectivity details.

Before running this experiment, on each data store and data set we ran the following two queries to enable the data stores and operating system to cache at least some of the data.

```
select * from object
select * from object_tag
```

Only those two tables are accessed for a hierarchically filtered state retrieval.

5.4.2 Impact of Selectivity

Figure 5.11 shows the performance of the prototype retrieving a state using hierarchical filtering and a 1,000 image set. The selectivity of the points can be seen in Table 5.3.

It is obvious that the performance of this type of filtering is far worse than that of tag and range filtering. The performance of all the data stores is worse than what we consider acceptable. SQLite performs the best but still takes 1.3 seconds to retrieve all the images in a state, using the root of the hierarchy.

We have reason to believe that the reason for this is twofold. The first reason is that the selectivity is higher and the second one has to do with the complexity of the queries executed. Hierarchical filters are by nature a hierarchy of tag filters, as each vertex in the

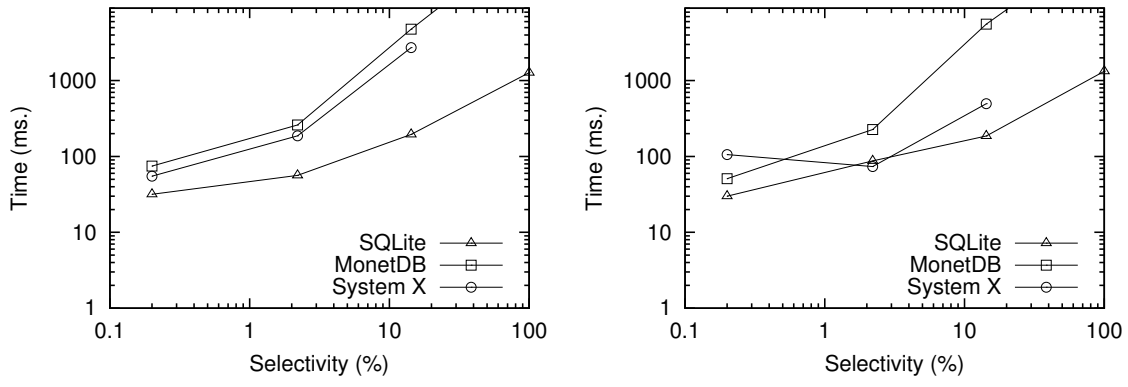
Hierarchy level of vertex	**Sel.**	**Obj.**	**Sel.**	**Obj.**	**Sel.**	**Obj.**	**Sel.**	**Obj.**
1	0.2%	2	0.23%	23	0.30%	302	0.29%	2,878
2	2.2%	22	1.84%	184	2.02%	2,023	2.02%	20,225
3	14.3%	143	13.74%	1,374	14.28%	14,283	14.29%	142,903
4 (root)	100%	1K	100%	10K	100%	100K	100%	1,000K
Data set size in images	1,000		10,000		100,000		1,000,000	

Table 5.3: Selectivity and objects in state in hierarchical filtering experiment.

hierarchy encapsulates a tag. Current data store implementations union together selects for all the vertices, as we need to know which objects are associated with the tag encapsulated by each vertex. In case of a hierarchical filter encapsulating the root of the hierarchy we are looking at a union of 400 selects.

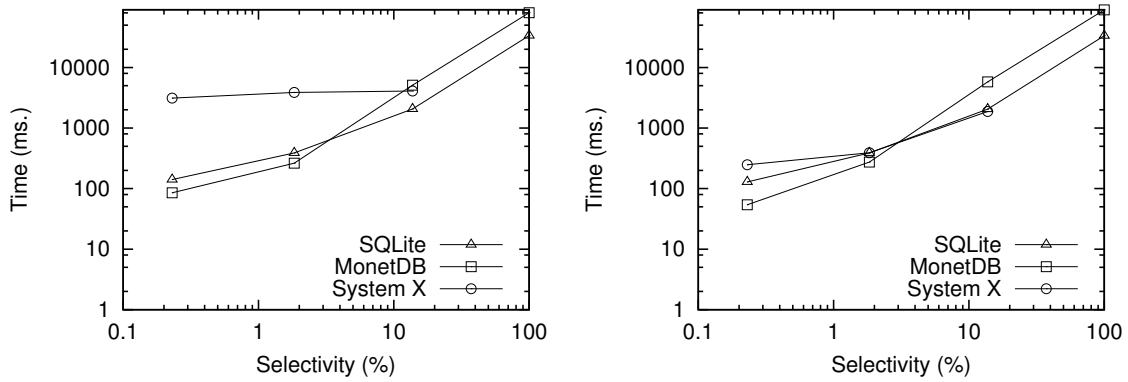
Comparing the results here to its closest relative, the tag filter results, is in some ways interesting. While SQLite performs the best here, the time it takes to retrieve a state of 1,000 images, on a second run, is 431ms, using a tag filter, but 1,330ms using a hierarchical filter. Note that retrieving the tag filtered state with 1,000 objects was done using the 10,000 image set so the cost of query complexity is likely higher than the disparity between those numbers.

Somewhat surprisingly, given the range filter experiment results, both MonetDB and System X suffer worse from this type of query complexity than SQLite. MonetDB retrieved the whole 1,000 image set as a state in more than 71 seconds, on a second run. We can safely put this down to query complexity since MonetDB takes 652ms to retrieve a state of 2,023 objects using a hierarchical filter on a level 2 vertex using the 100,000 image set. Each step up the hierarchy, used here, increases the number of unions seven fold or more. If we ignore the selectivity in these results, and look at MonetDB's performance as a function of query complexity (union count) it shows us that going from no unions to 7 gives a 4.4 fold increase in time (50.9ms to 226.4ms), going from 7 to 57 unions gives a 24.5 fold increase (226.4ms to 5544.7ms) and from 57 to 399 unions a 12.9 fold increase (5,544.7ms to 71,491.6ms). MonetDB's union scaling is superlinear. System X suffers quite a bit worse than SQLite from union scaling but not as badly as MonetDB. It was, however, not able to run the 100% selectivity state retrievals on any data set, as it returned an error indicating that it had run out of memory.



(a) First run (b) Second run

Figure 5.11: State retrieval using a hierarchical filter and a 1,000 image set.

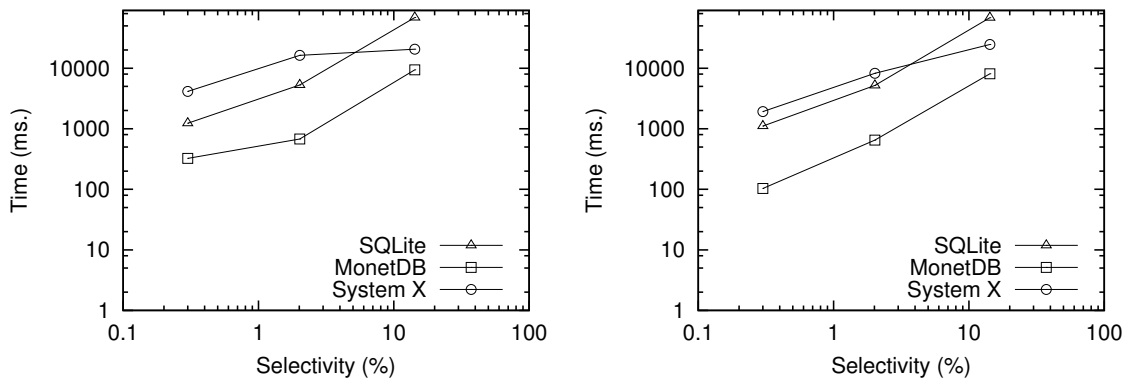


(a) First run (b) Second run

Figure 5.12: State retrieval using a hierarchical filter and a 10,000 image set.

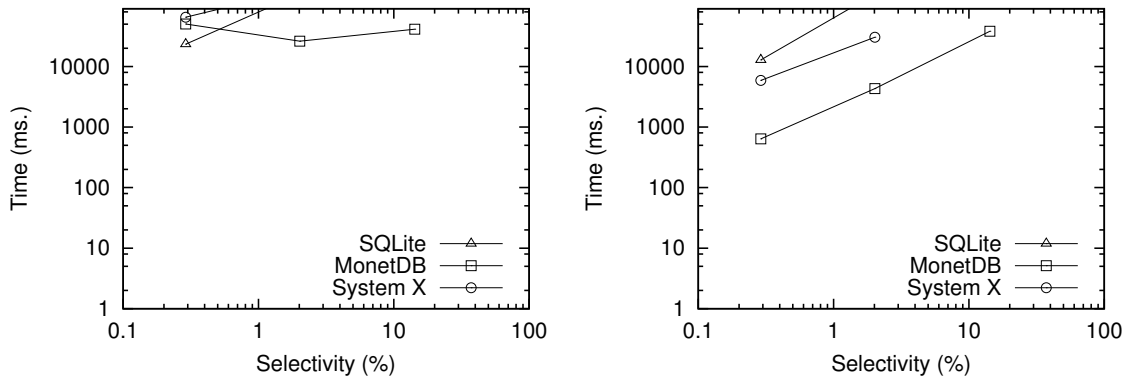
Figure 5.12 shows the results for the 10,000 image data set. It show us that in a worst case scenario the hierarchical filtering implementations, of all three data stores, of the prototype cannot handle an image set of 10,000 images. SQLite also suffers more from an increase in the data than the other two, this is a recurring trend. Another recurring trend is that System X suffers worse from an increase in data than MonetDB. The performance, while still unusable, of MonetDB does not change much although the data set is ten times larger.

Figures 5.13 and 5.14 show us the results for the 100,000 and 1,000,000 image sets, respectively. This only reinforces what we have already observed. MonetDB scales better than the other two with an increase in data set size and the time it takes is less a factor of the size of the data set than the complexity of our queries. None of the databases managed to finish the most complex query on the 100,000 image data set and only MonetDB managed



(a) First run (b) Second run

Figure 5.13: State retrieval using a hierarchical filter and a 100,000 image set.



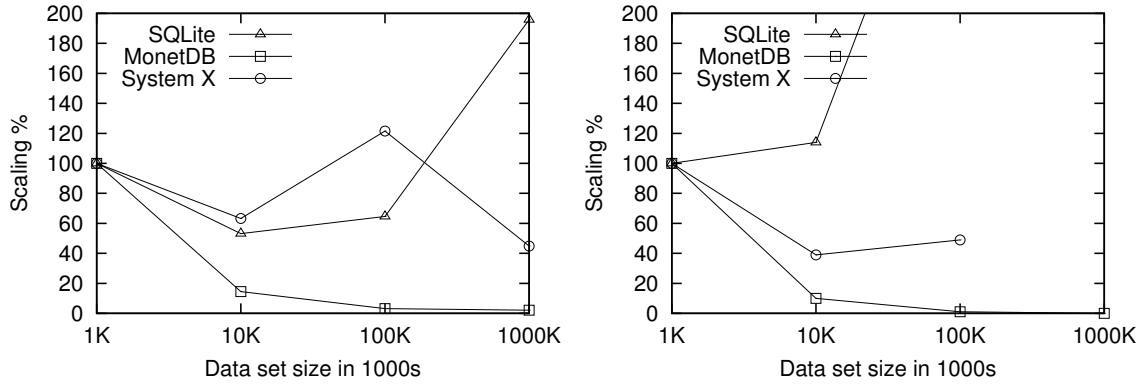
(a) First run (b) Second run

Figure 5.14: State retrieval using a hierarchical filter and a 1,000,000 image set.

the second most complex query on the largest data set. Again, the first run results for the largest set are incomparable with previous first run results result in this experiment.

5.4.3 Impact of Scaling

Figure 5.15 shows how the prototype scaled with the size of the data set. The x-axis shows the data set, while the y-axis shows the performance relative to the 1,000 image data set, scaled by the size of the data set. Recall that with this methodology, linear scaling would result in a horizontal line. Since the selectivity changes slightly between data sets we have also scaled the times to standardized selectivity, that of the largest set. We are using the data from the second run.



(a) Approx. 2% Selectivity (b) Approx. 14% Selectivity

Figure 5.15: Scaling of state retrieval using a hierarchical filter and varying selectivity.

The first diagram shows us scaling with a fairly low selectivity, about 2%, and a semi complex query; it contains eight vertices and therefore seven unions. MonetDB's great scaling is most likely a factor of it starting with a fairly bad performance due to query complexity. It is worse affected by query complexity than data size. SQLite scales badly with an image set larger than 10,000 images. Given the fact that it scaled decently with tag filtering, the cause of this poorer scaling is likely a much higher query complexity. System X's scaling is odd, since its scaling gets worse going from the 10,000 image set to the 100,000 image set but it improves going from the 100,000 to the 1,000,000 image set. The reason for this is unclear, but System X shows this pattern for both the first and the second run.

The picture shown for the larger selectivity, using 56 unions, is more in line with what we expected. SQLite scales badly, while both System X and MonetDB show an exaggerated version of their usual scaling behavior. This may again be due to query complexity overhead, as the initial reference point is much worse than in previous experiments. Only MonetDB managed to retrieve the state using the largest set in this experiment. Given the unusable performance of the data stores, in this experiment, we should not read too much into this data.

The conclusion of this experiment is that in order to get reasonable performance from hierarchical filters, using worst case hierarchies, we need to do two things. First and foremost we must make our queries less complex; one promising avenue for this is using materialized views, allowing us to embed the hierarchical vertex id's in the data and rid the queries of the unions. The second conclusion is that there must be a limit to the amount of objects in a state that are useful for presentation. If we could limit the number of objects returned for a single vertex in a hierarchy we would get considerably better performance. Consider, for example, if we had a limit of 5 objects per vertex in the experimental hierarchy; the maximum number of objects in a state would then be 2,000. In order for this to be useful we would first need to get rid of the unions. This sort of size restriction may also be considered for other types of filters. Any limitation of those sorts would need to be done in a manner that the user perceives as intuitive, seamless, and non-intrusive. One potential solution is to pre-calculate aggregates of hypercube cells in some fashion, for example using collages or slide-shows.

5.5 Analysis of ObjectCube Overhead

Up to now we have been evaluating the performance of the prototype with regards to the data stores. Now we discuss the overhead the prototype adds to the query execution time of the data stores for state retrieval. We discuss the cost of the prototype processing the data retrieved from data stores, the cost of tag caching, the cost of creating state hierarchies and finally the cost of using the Python interface. In this analysis we use data from the second run of the hierarchical filtering experiment. The items discussed here are not dependent on the performance of the underlying data store.

5.5.1 Data Processing

In order to evaluate the overhead of data processing, done by the prototype, we ran the queries executed by the prototype directly against the same data store and data set in the Python test code. That Python code uses the Python database access code for each database.

In order to retrieve a state we execute a query that joins together the objects and their tags. With any filters in effect, we get as many rows returned as there are image/tag associations in the set returned. In our C++ code we then process this data to create logical objects.

In order to measure the time taken to get all the data, and not just the first rows, the Python code loops through the rows and accesses the first column in each row. We only look at the second run and the 1,000 image set here; to increase the odds of both facing the same caching conditions. This is not a comparison of equals so at best this will give us ideas and insights.

Table 5.4 presents the C++ time, prototype time without the Python interface, of a state retrieval and the Python execution time of the same query that was used to retrieve the state.

Objects in state	SQLite		MonetDB		System X	
	C++	Python	C++	Python	C++	Python
2	12	20	31	40	86	119
22	69	32	205	224	54	31
143	162	144	5,515	5,274	468	123
1,000	1,264	1,172	71,417	78,308	Err.	Err.

Table 5.4: Data processing overhead (milliseconds).

It is in most ways indicative of the execution time of our C++ data retrieval vs. Python execution of the same query.

In the case of SQLite, there tends to be little between them but the Python version is slightly faster more often than not. In the case of MonetDB, the Python version is always slightly slower. The most interesting data is for System X, there the Python version seems much less affected by increased number of rows returned than the C++ version. This trend goes through all our data.

We must be careful drawing any conclusions from this data, we do not know the quality of the Python database communication layers nor their exact behavior. This data does indicate that, in the case of SQLite and MonetDB, there is not a great deal of overhead. In System X's case it may point to inefficiencies in our data processing code. While we cannot rule this out, without further testing, it should be pointed out that the code used is near identical to the one used for SQLite. There are no clear conclusions here.

5.5.2 Tag Caching & Layer Conversion

In order to provide fast access to tag-sets and hierarchies, and to assist in state retrieval, we cache all tag-sets, hierarchies and tags. In the case of our current data store implementations and tag types, this rids us of four outer joins in our state retrieval query. When a state is retrieved, we only retrieve unique id's of tags and then add the tags to the state from cache. Due to how the code is written, we convert from data storage types to logic layer types in the same loop we add the tags. Those two will therefore be discussed together. The cost discussed here is not a function of the data stores so we only look at data from a single data store. Since MonetDB managed to create the largest hierarchically filtered state we will use that.

Table 5.5 shows the overhead of data conversion and addition of tags from cache. It can be summarized as not negligible; it does, however, scale linearly, and will not be noticeable unless the state contains thousands of images. For 1,000 images from our largest set, with 13.6 tags per image on average, this equals about 45ms. For a prototype this is acceptable.

Objects in State	Approx. tags	Time	Time per 10,000 tags
2,878	39,198	129.34	33.00
20,225	275,464	889.78	32.30
142,902	1,946,325	6,570.15	33.76

Table 5.5: Tag caching & layer conversion overhead (milliseconds).

If we wish to improve this, then adding tags in parallel using multiple CPU cores might be an interesting approach.

5.5.3 Creation of State Hierarchies

Each vertex in a state hierarchy has a list of images associated with the tag it encapsulates. In order to achieve this, we gather data in a hash map and then traverse recursively through the hierarchy populating each vertex with the appropriate data. This is done using both data from the hash map and an aggregation of the objects associated with the child vertices of the vertex being processed. The cost of this is not a function of the data stores so we only look at data from a single data store. Again that will be MonetDB and for the same reasons as in Section 5.5.2.

Table 5.6 shows the time it takes to create a state hierarchy for two image sets. If we compare the results from those two data sets we can see two things. The first is, looking at the data for the 10,000 image data set, that increasing the vertice count from 57 to 400 has considerable effect on the time needed to create the state hierarchy. We can put this down to the vertice count rather than the object count by looking at the same seven fold increase in objects between the 1 and 8 vertices in the 1,000,000 image set. Increasing the vertice count seven fold, from 57 to 400 adds 77% to the time needed. In the first three smaller data sets there is only a marked increase in the time needed to construct the state hierarchy when going from 57 to 400 vertices. This is most likely due to the recursive aggregation of the images of the child vertices. The second interesting thing is the anomaly, when using the largest data set, when we go from 8 vertices to 57 vertices. This kind of increase does not happen in the smaller sets in this place. This may be due to scarcity of resources or simply the amount of data copied in the aggregation phase. We do not have numbers for 400 vertices on the largest set, as no data store managed to return one.

Data Set Size	Objects in State	Vertices in subtree	Time
10,000	23	1	15.22
	184	8	14.26
	1,374	57	16.08
	10,000	400	28.54
1,000,000	2,878	1	14.37
	20,225	8	15.59
	142,902	57	31.52

Table 5.6: State hierarchy creation overhead (milliseconds).

The overhead of state hierarchy construction is more than acceptable and scales well.

5.5.4 Python vs. C++ Interface

All the experiments were run through the Python interface and we measured both the time an operation took in the C++ code and then the total time in Python. Examining the data gave us a few interesting pieces of information. One is that the overhead is almost entirely related to the amount of data returned. In the tag filtering experiment, the first run on the smallest image set gave us a state with 0 images. Table 5.7 shows the overhead for that experiment.

When we run the hierarchical experiment, retrieving a state, we always create the 400 vertice state hierarchy. Table 5.8 shows the fixed cost of that, along with creating 2 state objects. As the table shows, the overhead is less than 3ms in all cases.

Table 5.9 finally shows the overhead of the Python interface using large amounts of data. The data is from the hierarchical filter experiment. Since MonetDB managed to create the largest hierarchically filtered state we use its results.

Data store	Python time	C++ time	Overhead
SQLite	9.21	9.05	0.16
MonetDB	31.03	30.37	0.66
System X	14.03	13.84	0.19

Table 5.7: Fixed overhead of Python interface (milliseconds).

Data store	Python time	C++ time	Overhead
SQLite	29.93	27.00	2.93
MonetDB	50.86	47.89	2.97
System X	105.8	102.85	2.95

Table 5.8: Fixed Python overhead of creating a 400 vertice hierarchy (milliseconds).

Objects in State	Python time	C++ time	Overhead	Overhead per 1.000 objects
2,878	640.76	621.94	18.82	6.54
20,225	4,318.65	4,190.36	128.29	6.34
142,903	38,229.19	37,318.8	910.39	6.37

Table 5.9: Python overhead as a function of the number of objects in a state (milliseconds).

The cost of using the Python interface can be summed up as negligible, for our purposes, if there is little data returned. When it comes to large amounts of data the overhead is a function of data rather than the time it takes the C++ code to execute. Paying between 6 and 7ms for every 1,000 objects in a state, through the Python interface, is acceptable. We consider the performance of the Python interface, of the prototype, quite acceptable.

5.6 Summary

Using a single second threshold for acceptable performance, and a very high selectivity of 10%, we can conclude that the prototype performs acceptably, using MonetDB as a data store, for data sets of 40-50,000 objects. And this only if we do not use hierarchical filtering. The other two data stores are usable for smaller data sets; the limit is around 1,000 objects. The prototypes implementations of tag- and range filtering can be considered acceptable.

The performance of hierarchical filtering can be summed up as unacceptable. As hierarchies and hierarchical filtering are very important to the usability of the prototype, we must redesign and re-implement hierarchical filtering. We plan to address the performance of hierarchical filtering using the standard relational tools of materialized views.

The three data stores were differently affected by different aspects of scalability. SQLite was the worst affected by increase in data set size, with System X coming second. When we look at how they scaled with regards to the size of the result set (rows), System X had the worst scaling, the other two were similar. The added query complexity between tag- and range filtering affected all the data stores to some degree, but it affected SQLite far worse than the other two. Hierarchical filtering on a large sub-hierarchy results in very complex queries and all the data stores were greatly affected, SQLite the least and MonetDB the greatest.

Summarizing our results, aside from our limited implementation of hierarchical filtering, the column-store MonetDB was the best data store.

If we consider that there may be a practical limit to the number of images, in a state, that can be retrieved and shown to the user in a timely manner, we can see that there are great opportunities in limiting the number of objects returned in a state. Any limitation of that sort would need to be done in a manner that the user perceives as intuitive, seamless, and non-intrusive. Consider, for example, that MonetDB retrieved a state of 1,000 objects from the 1,000,000 image data set in 1.7 seconds using a range filter and 0.35 seconds using a

tag filter. Any limitation on the number of objects returned has great potential regarding performance on large data sets.

Finally, we considered the overhead of the prototype. For 1,000 objects, it is 45ms for layer conversion and adding tags from cache, 15-30ms for each state hierarchy and 6-7ms for the Python interface. We expect to see 1-3 state hierarchies in a typical state, so we should typically be looking at 140ms or less. This overhead is quite acceptable.

Chapter 6

Related work

Our work is both related to previous academic work and current software solutions. In Section 6.1, we discuss current software solutions, both simple viewers and more advanced solutions in image browsing. In Section 6.2, we discuss related academic work, both having to do with problem domain specific multi-dimensional models, and image browsing in particular.

6.1 Current Browsers

When looking at how current software solutions relate to our work, we are only interested in what conceptual features they support. For instance what support they have for meta-data and it's automatic addition, categorization, methods of restricting the set viewed and the grouping — including aggregation — of images. Our intention is not to review them in any way. Image browsing solutions can, broadly, be split into two categories. The first one contains simple viewers and the second one more complete solutions. We discuss each one in turn.

6.1.1 Simple Viewers

We define simple viewers by their inability to attach meta-data to images. What follows is that they have no meta-data driven way of restricting the images viewed. The only way to organize images using this type of browser is by creating the appropriate folder structures. The folder structure allows categorization of images in a single dimension — according to

a single criterion. Examples of solutions that fall in this category includes products like ACDSee, FastStone and similar.

It must be said that the feature set of those products is much larger than only image browsing; it is simply only image browsing that is relevant to our work.

While being precursors of our work, their inability to support meta-data, multiple hierarchies and powerful restrictions on the set viewed makes them a distant relation at best.

6.1.2 Advanced Browsers

As opposed to simple viewers, the advanced solutions can attach meta-data to images. There is a myriad of advanced image browsing solutions available, here we discuss how our work relates to three well known image browsers: Flickr, Picasa and iPhoto. Flickr and Picasa both use untyped tagging; each tag is simply an alphanumerical string. iPhoto does support limited user tagging for the names of people in images and description. That being said, all three solutions support geotagging (adding geographical location meta-data to media) as a special case. While those solutions support attributes, or tags, of images, like description (all), title (Flickr) and event (iPhoto) that relate to the tag-sets of our work there is no generic solution to it. Tags are either a part of a predefined group, like image name or description, or they cannot be grouped. This compares poorly to our tag-sets.

All of the advanced solutions support automated creation of meta-data to some extent. Picasa and iPhoto support automated facial recognition, and all three of them are able to extract geographical location meta-data from images. Automated tagging is, however, limited to those two uses; there is no generic support for automated tagging.

Those solutions have a limited ability to group images; one can select a single group (album, group, event, photographer etc.) but there are no aggregations and no hierarhcies except for the folder structure of the file-system (Picasa). Unlike our work, none of them support relations between the tags of a group. There is no support for hierarchically categorizing the tags used. That leaves us with a limited ability to group and categorize images. Compared to our approach to hierarchies they have a poor support for grouping and categorization.

Both Flickr and Picasa support a fairly powerful text search through the meta-data they store. Both support restricting the set shown using multiple words. iPhoto, while supporting multiple words, has a more limited text search. This can be considered similar to our tag filtering. When it comes to range filtering only Flickr has something similar, where

you can define a date range for either the time the image was taken or uploaded. It is, however, limited to only those two. Our solution has no such limitations. If we look at hierarchical filtering, the only thing that resembles it is browsing through the file-system using Picasa. Using the file-system limits Picasa to a single hierarchy. Again, our work has no such limitations. There are browser solutions, like PicaJet, that do support hierarchies but in that case a user can only select a single vertex in a single hierarchy to restrict the set viewed.

When it comes to viewing the images, the browsers use a single dimension. Our work supports multiple dimensions and therefore supports multi-dimensional browsing.

In addition to the, aforementioned, browsers there is software like PivotViewer, which has a multi-dimensional model and fair grouping capabilities, but a complex static image collection, no automated tagging and no hierarchies. It seems better suited to web presentations of predefined, static, collections than dynamic media collection browsing.

To summarize, while current solutions can be considered rather impressive they are less flexible and lack many of the key features of our work; among others the ability to define and view a set of images using more than one dimension and more than a single hierarchy.

6.2 Academic Work

There exists prior work in creating multi-dimensional models for data that is not easily aggregated, for example for spatial data (Bimonte & Kang, 2010), and document analysis (Ravat, Teste, Tournier, & Zurifluh, 2007). Both of those are specific to their, single, problem domain. They are good indicators of the validity of our approach, but not candidates for a solution.

Work has also been done regarding efficient multi-dimensional operations on spatial data (Wang et al., 2003). This, mostly, shows the potential of designing data structures for multi-dimensional data that is not easily aggregated.

PhotoMesa (Bederson, 2001) and PhotoFinder (Kang & Shneiderman, 2000) have similar abilities, regarding image browsing, as the advanced browsers described in Section 6.1.2.

PIBE (Bartolini et al., 2004), uses visual feature based hierarchical clustering but lacks tagging. Its focus is on user modifiable dynamic hierarchies, based on visual similarity, an area we have yet to explore. While the work is interesting, PIBE is limited as a browser.

Perhaps its greatest limitation is relying solely on content-based search, which is known to yield imprecise results (Bartolini & Ciaccia, 2009).

Scenique (Bartolini & Ciaccia, 2009), PIBE's progeny, adds hierarchically ordered tags to PIBE's content based hierarchies; and thereby multiple categories, both visual and semantic. Scenique also adds facilities for suggesting tags for an image, based on image similarity, which is of great interest. Scenique is, however, rather inflexible, since each and every meta-datum must be a member in a hierarchy, each tag added therefore must be placed directly into a hierarchy. There is support for tag searching, but since Scenique has no concept of related tags it is a global search, without context, which is a limitation. In ObjectCube, the tag-sets provide this context. In addition to this, Scenique supports searches only within hierarchies, which do provide context. In this case, Scenique, uses hierarchies in a similar way to ObjectCube's tag-sets. This combination of the hierarchy and the tag-set, however, does not provide the same clarity of context as separating those two. There is no concept of deriving one or more hierarchies from the same set of cohesive tags. There is no mention of range searches either. In addition to this, it is also unclear whether Scenique can handle large collections of images, close to being realistic in size, as the largest collection mentioned contains only 5,000 images (Bartolini & Ciaccia, 2009).

Camelis (Ferré, 2009) is a logical property driven object browser that has hierarchies, of sorts, and concepts (meta-data). Camelis uses a single concept lattice which logics operate on. There is no split into logical sets like our tag-sets. Browsing is done by, both, selecting navigation links and manually editing a query string. We believe that the combination of those factors makes it more complex to use than ObjectCube. In addition to that it also lacks the ability to present images in stacks based on attached meta-data.

If we look at the browsing scenarios in Chapter 1, we see that, lacking both range filtering and concepts similar to tag-property based hierarchies, Scenique may have problems finding images taken during summer, and its ability to handle realistic collections is unknown. Camelis, using only 2-dimensional presentation and lacking the ability to present images in stacks based on attached meta-data, fares no better. Those research projects, while highly interesting in their own right, therefore do not provide an acceptable solution to image browsing either.

Chapter 7

Conclusion

In this thesis, we have pointed out the ever growing gap between the rather stagnant capabilities of current image browsers and the needs of users to be able to organize and browse ever larger collections of images in an effective way. Here, an effective way would be enabling the user to define and view the set of images, or more generally objects, the user is interested in.

Our solution to this problem is based on the multi-dimensional model, commonly used in OLAP applications in the Business Intelligence domain. A classical example of usage would be viewing total sales broken down by product, store location and years. A key feature of this multi-dimensional model is showing a scalar value in each cell of a multi-dimensional cube using aggregation. While it is clear that the multi-dimensional model is not immediately suitable to image browsing, our belief is that the similarity of the problems is significant enough to use the concepts of this model as the foundation of a powerful and flexible solution to image browsing.

7.1 Summary of Contribution

We have defined a generic multi-dimensional model called ObjectCube. That model stores references to objects, that can be of any file type, and supports adding tags of various types to objects. Each tag belongs to a single tag-set, which is a set of distinct tags that are of the same type. The tag-sets group the tags but to categorize them, or define relationships between tags, we use hierarchies. Each hierarchy defines the relationship between a subset of the tags of a single tag-set. Hierarchies can even define property based

relations between the tags of a tag-set. In order to define a set for viewing we use filters. The multi-dimensional view, thus defined, is the state.

There is a high degree of correspondence between the concepts of multi-dimensional analysis and the concepts of ObjectCube. All the major differences stem from the fact that the multi-dimensional analysis model is applied to simple facts, which are typically numerical data items, while the ObjectCube model is applied to complex media objects which cannot be easily manipulated or aggregated.

In addition to defining the ObjectCube model we have implemented it in a, synonymous, software prototype. The prototype is stateful, as it is designed for a single user browsing experience. It supports automated tagging through the use of plug-ins. It is also multi-platform, multi-lingual, and storage layer invariant. It has an interface in both C++ and Python and is suitable for open source licensing, as dependency on third party software has been kept to a minimum. ObjectCube has well defined layers, limited dependencies, and in its development we have made extensive use of automated unit tests. It consists of approximately forty thousand lines of C++ code.

While we focus on images in this thesis, to ground it, neither ObjectCube the model nor software has any data type dependency.

In order to evaluate the performance of the prototype we measured the time it takes to retrieve a state, the most complex and time consuming operation of the prototype, using all three filter types and all supported data stores. All experiments were created as stress tests, each is a worst case scenario. In addition we examined what performance overhead ObjectCube adds to the time taken by the data stores.

The column-store MonetDB performed best for both tag filters and range filters, and performance was quite acceptable for image collections of up to 100,000 images (answering goes over one second when managing more than 40-50,000 images). Hierarchical filtering, on the other hand, does not perform well in the current implementation due to the complexity of the queries retrieving the browsing state and the enormous amount of objects retrieved in a worst case scenario.

While the overriding factor in the performance of ObjectCube is the performance of the data store queries, there is also some overhead imposed by the prototype. For a state of 1,000 objects the overhead ranges from about 50 - 140 ms, in our experiments, mostly depending on the number of state hierarchies used for browsing. This overhead is quite acceptable.

7.2 Future Work

So far, our focus has been on the data model itself. It is, however, clear that we must also look into its life-cycle management. This is especially important for hierarchies, which may have to evolve to keep up with users' taxonomy needs. A user may wish to maintain hierarchies by merging or splitting them, adding or removing hierarchy levels, merging nodes and so on. This is something that must be analyzed further.

In general, user studies must be performed to validate the data model.

There are, however, many interesting avenues for future work that relate to the data model and the prototype. First, we address the current performance issues, in particular for hierarchical filtering. In order to improve performance, in particular for large object collections, we also plan to develop methods to reduce the maximum number of objects retrieved in a state, in a manner that the user perceives as intuitive, seamless and non-intrusive. One potential solution is to pre-calculate aggregates of hypercube cells in some fashion, for example using collages or slide-shows. This is a key subject of our future work.

Additionally, we plan to add support for dynamic browsing dimensions, for example based on similarity. Those dimensions should, preferably, be generic enough to work for more than a single similarity type.

Once the basic architecture is complete, however, most of the interesting work lies in the potential applications of the ObjectCube model. As mentioned earlier, we are already working on an image browsing application, and some interesting avenues of future work for this application include a face-recognition plug-in and geotagging support, which calls for a new tag type and a plug-in. Other applications of interest, which may require specialized plug-ins, are in music browsing and multi-dimensional file-system browsing. In fact, we believe that the ObjectCube model proposed in this paper can radically change the browsing experience for users of most media objects.

Bibliography

Abadi, D. J., Madden, S. R., & Hachem, N. (2008). Column-stores vs. row-stores: how different are they really? In *Proc. of the 34th ACM Int. Conf. on Management of Data.*

Bartolini, I., & Ciaccia, P. (2009). Integrating Semantic and Visual Facets for Browsing Digital Photo Collections. In *Proc. of the 17th Italian Symp. on Advanced Database Systems.*

Bartolini, I., Ciaccia, P., & Patella, M. (2004). The PIBE personalizable image browsing engine. In *Proc. of the 1st Int. Workshop on Computer Vision Meets Databases.*

Bederson, B. B. (2001). PhotoMesa: a zoomable image browser using quantum treemaps and bubblemaps. In *Proc. of the 14th ACM Symp. on User Interface Software and Technology.*

Bimonte, S., & Kang, M.-A. (2010). *Towards a Model for the Multidimensional Analysis of Field Data.*

Codd, E., Codd, S., & Salley, C. (1993, 09). *Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate.* Whitepaper.

Council, T. O. (1995, 1). *OLAP AND OLAP Server Definitions.* Webpage.

Domingues, P., Araujo, F., & Silva, L. (2009). *Evaluating the performance and intrusiveness of virtual machines for desktop grid computing.*

Ewald, M. (2007, 04). *Building a Better Plugin Architecture.* Webpage.

Ferré, S. (2009). Camelis: a logical information system to organize and browse a collection of documents. *Int. J. General Systems*, *38*(4).

Hardarson, K., & Jonsson, B. T. (2007). Breaking out of the Shoebox: Towards Having Fun with Digital Images. In *Proc. of the 3rd int. workshop on computer vision meets databases.*

Idreos, S., Kersten, M. L., & Manegold, S. (2009). Self-organizing tuple reconstruction in column-stores. In *Proc. of the 35th ACM Int. Conf. on Management of Data.*

Iverson, K. E. (1962). *A programming language*. New York, NY, USA: John Wiley & Sons, Inc.

Kang, H., & Shneiderman, B. (2000). Visualization Methods for Personal Photo Collections: Browsing and Searching in the PhotoFinder. In *IEEE Int. Conf. on Multimedia and Expo (III).*

Manegold, S., Boncz, P. A., & Kersten, M. L. (2000). Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, *9*(3), 231–246.

Misc. (2010a, 11). *Graph (mathematics).* Wikipedia.

Misc. (2010b, 11). *Tree (graph theory).* Wikipedia.

MonetDB. (2008, 09). *TPC-H Performance Study.* Webpage.

Pendse, N. (2008, 03). *What is OLAP?* Webpage.

Ravat, F., Teste, O., Tournier, R., & Zurifluh, G. (2007). *A Conceptual Model for Multidimensional Analysis of Documents.* Springer-Verlag Berlin Heidelberg.

SQLite. (2010, 11). *SQLite.* Webpage.

Wang, B., Pan, F., Ren, D., Cui, Y., Ding, Q., & Perrizo, W. (2003). *Efficient OLAP Operations for Spatial Data Using Peano Trees.* ACM.

Zukowski, M., Boncz, P. A., Nes, N., & Héman, S. (2005). MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, *28*(2), 17-22.

Zukowski, M., Heman, S., Nes, N., & Boncz, P. A. (2006). Super-Scalar RAM-CPU Cache Compression. In L. Liu, A. Reuter, K.-Y. Whang, & J. Zhang (Eds.), *Icde* (p. 59). IEEE Computer Society.

Appendix A

Table Structure and Sample Queries

In this appendix we present a part of the tables used by our data stores along with query samples for all three filter types.

A.1 Table Structure

Figure A.1 shows the central part of the table structure used by all three data stores the prototype uses for persistent storage.

Note, these are only the central tables, mostly having to do with the retrieval of state. Tables for hierarhies, plug-ins, language and translation, and others have been left out for simplicities sake.

A.2 Sample Queries

All of the following queries are actual samples of queries we ran as a part of the experiments. Those particular queries were generated by our SQLite data store implementation.

A.2.1 Tag Filter Query

The main reason for the complexity of this query is the need to retrieve all objects with a single tag, but with all their tags, in a single query. This applies to all the queries.

```
SELECT  c.ID, c.QUALIFIED_NAME, c.TAG_ID, c.FILTER_ID,
        c.DIMENSION_NODE_ID, c.UPPER_LEFT_X, c.UPPER_LEFT_Y,
        c.LOWER_RIGHT_X, c.LOWER_RIGHT_Y
```

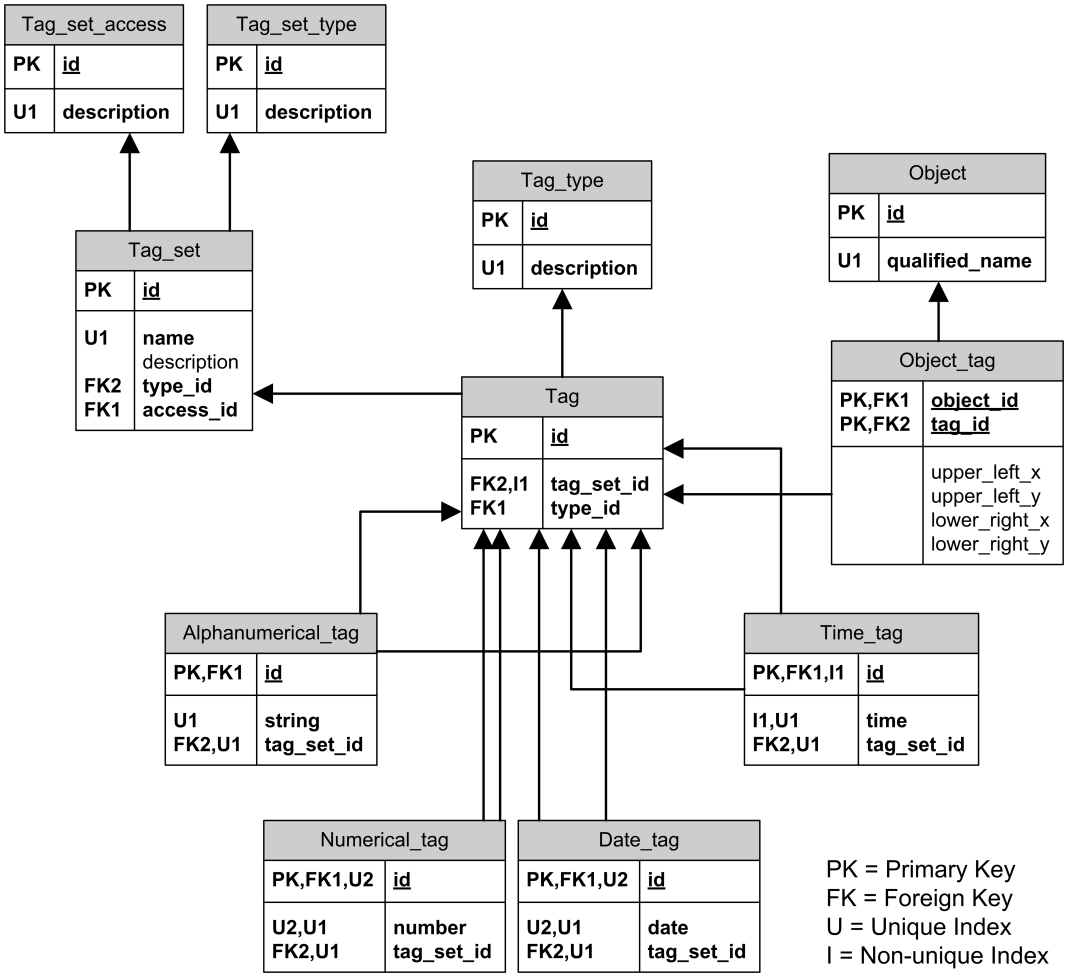


Figure A.1: Central part of table structure.

```
FROM
(
  SELECT b.ID, b.QUALIFIED_NAME, b.TAG_ID, b.FILTER_ID,
  b.DIMENSION_NODE_ID, b.UPPER_LEFT_X, b.UPPER_LEFT_Y,
  b.LOWER_RIGHT_X, b.LOWER_RIGHT_Y
  FROM
  (
    select  o.id, o.qualified_name, ot.tag_id,  1 as filter_id,
    -1 as dimension_node_id, ot.upper_left_x, ot.upper_left_y,
    ot.lower_right_x, ot.lower_right_y
    from object o, object_tag ot
    where o.id = ot.object_id and ot.tag_id = 200873
  ) as b,
  (
    select  distinct( ot.object_id ) as id
    from object_tag ot
    where ot.tag_id = 200873
  ) as a
  where b.id = a.id
  UNION ALL
  select  ot.object_id as id, '' as qualified_name, ot.tag_id,
          -1 as filter_id, -1 as dimension_node_id, ot.upper_left_x,
```

```
          ot.upper_left_y, ot.lower_right_x, ot.lower_right_y
  from object_tag ot
  where ot.object_id in
  (
    select  distinct( ot.object_id ) as id
    from object_tag ot
    where ot.tag_id = 200873
  )
) as c
order by c.ID, c.TAG_ID, c.FILTER_ID desc
```

A.2.2 Range Filter Query

```
SELECT  c.ID, c.QUALIFIED_NAME, c.TAG_ID, c.FILTER_ID, c.DIMENSION_NODE_ID,
        c.UPPER_LEFT_X, c.UPPER_LEFT_Y, c.LOWER_RIGHT_X, c.LOWER_RIGHT_Y
FROM
(
  SELECT b.ID, b.QUALIFIED_NAME, b.TAG_ID, b.FILTER_ID, b.DIMENSION_NODE_ID,
         b.UPPER_LEFT_X, b.UPPER_LEFT_Y, b.LOWER_RIGHT_X, b.LOWER_RIGHT_Y
  FROM
  (
    select o.id, o.qualified_name, ot.tag_id, 1 as filter_id,
          -1 as dimension_node_id, ot.upper_left_x, ot.upper_left_y,
          ot.lower_right_x, ot.lower_right_y
    from object o, object_tag ot, time_tag tt
    where o.id = ot.object_id and tt.id = ot.tag_id and tt.time between
          strftime( "%H:%M:%f", '00:00:01.000' ) and
          strftime( "%H:%M:%f", '00:00:01.000' ) and tt.tag_set_id = 11
  ) as b,
  (
    select distinct( ot.object_id ) as id
    from object_tag ot, time_tag tt
    where tt.id = ot.tag_id and tt.time between
          strftime( "%H:%M:%f", '00:00:01.000' ) and
          strftime( "%H:%M:%f", '00:00:01.000' ) and tt.tag_set_id = 11
  ) as a
  where b.id = a.id
  UNION ALL
  select  ot.object_id as id, '' as qualified_name, ot.tag_id,
          -1 as filter_id, -1 as dimension_node_id, ot.upper_left_x,
          ot.upper_left_y, ot.lower_right_x, ot.lower_right_y
          from object_tag ot
          where ot.object_id in
          (
            select distinct( ot.object_id ) as id
            from object_tag ot, time_tag tt
            where tt.id = ot.tag_id and tt.time between
                  strftime( "%H:%M:%f", '00:00:01.000' ) and
                  strftime( "%H:%M:%f", '00:00:01.000' ) and tt.tag_set_id = 11
          )
  ) as c
order by c.ID, c.TAG_ID, c.FILTER_ID desc
```

A.2.3 Hierarchical Filter Query

This hierarchical filter query is for a sub-hierarchy root and its seven children.

```
SELECT c.ID, c.QUALIFIED_NAME, c.TAG_ID, c.FILTER_ID, c.DIMENSION_NODE_ID,
       c.UPPER_LEFT_X, c.UPPER_LEFT_Y, c.LOWER_RIGHT_X, c.LOWER_RIGHT_Y
FROM
(
    SELECT b.ID, b.QUALIFIED_NAME, b.TAG_ID, b.FILTER_ID, b.DIMENSION_NODE_ID,
           b.UPPER_LEFT_X, b.UPPER_LEFT_Y, b.LOWER_RIGHT_X, b.LOWER_RIGHT_Y
    FROM
    (
      select o.id, o.qualified_name, ot.tag_id, 3 as filter_id, 247 as dimension_node_id,
      ot.upper_left_x, ot.upper_left_y, ot.lower_right_x, ot.lower_right_y
      from object o, object_tag ot
      where o.id = ot.object_id and ot.tag_id = 203082
  UNION ALL
  select o.id, o.qualified_name, ot.tag_id, 3 as filter_id, 248 as dimension_node_id,
        ot.upper_left_x, ot.upper_left_y, ot.lower_right_x, ot.lower_right_y
  from object o, object_tag ot
  where o.id = ot.object_id and ot.tag_id = 203083
  UNION ALL
  select o.id, o.qualified_name, ot.tag_id, 3 as filter_id, 249 as dimension_node_id,
              ot.upper_left_x, ot.upper_left_y, ot.lower_right_x, ot.lower_right_y
  from object o, object_tag ot
  where o.id = ot.object_id and ot.tag_id = 203084
  UNION ALL
  select o.id, o.qualified_name, ot.tag_id, 3 as filter_id, 250 as dimension_node_id,
        ot.upper_left_x, ot.upper_left_y, ot.lower_right_x, ot.lower_right_y
  from object o, object_tag ot
  where o.id = ot.object_id and ot.tag_id = 203085
  UNION ALL
  select o.id, o.qualified_name, ot.tag_id, 3 as filter_id, 251 as dimension_node_id,
              ot.upper_left_x, ot.upper_left_y, ot.lower_right_x, ot.lower_right_y
  from object o, object_tag ot
  where o.id = ot.object_id and ot.tag_id = 203086
  UNION ALL
  select o.id, o.qualified_name, ot.tag_id, 3 as filter_id, 252 as dimension_node_id,
              ot.upper_left_x, ot.upper_left_y, ot.lower_right_x, ot.lower_right_y
  from object o, object_tag ot
  where o.id = ot.object_id and ot.tag_id = 203087
  UNION ALL
  select o.id, o.qualified_name, ot.tag_id, 3 as filter_id, 253 as dimension_node_id,
            ot.upper_left_x, ot.upper_left_y, ot.lower_right_x, ot.lower_right_y
  from object o, object_tag ot
  where o.id = ot.object_id and ot.tag_id = 203088
  UNION ALL
  select o.id, o.qualified_name, ot.tag_id, 3 as filter_id, 254 as dimension_node_id,
        ot.upper_left_x, ot.upper_left_y, ot.lower_right_x, ot.lower_right_y
  from object o, object_tag ot
  where o.id = ot.object_id and ot.tag_id = 203089
  ) as b,
  (
    select distinct( ot.object_id ) as id
    from object_tag ot
    where ot.tag_id in( 203082, 203083, 203084, 203085, 203086, 203087, 203088, 203089 )
```

```
  ) as a
  where b.id = a.id
  UNION ALL
  select  ot.object_id as id, '' as qualified_name, ot.tag_id, -1 as filter_id, -1 as dimension_node_id,
         ot.upper_left_x, ot.upper_left_y, ot.lower_right_x, ot.lower_right_y
  from object_tag ot
  where ot.object_id in
  (
    select distinct( ot.object_id ) as id
    from object_tag ot
    where ot.tag_id in( 203082, 203083, 203084, 203085, 203086, 203087, 203088, 203089 )
  )
) as c
order by c.ID, c.TAG_ID, c.FILTER_ID desc
```

School of Computer Science
Reykjavík University
Menntavegi 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.reykjavikuniversity.is
ISSN 1670-8539