Web Performance Optimization: Analytics

Wim Leers

Thesis proposed to achieve the degree of master
in computer science/databases

Promotor: Prof. dr. Jan Van den Bussche

Hasselt University
Academic year 2009-2010

Abstract

The goal of this master thesis is to make a useful contribution to the upcoming *Web Performance Optimization* field, or *WPO* for short. The importance of WPO is only growing, and as it grows, the need for tools that can assist developers in making the right decisions also grows. Hence that is the goal of this thesis: to build a tool that can be used for the continuous profiling of a web site's performance.

The developer begins by integrating *Episodes* (a tool for measuring how long the various episodes of the page loading process take) with the web site, which will log the measured results to an Episodes log file. This log file by itself is a good set of data that can be interpreted, but it would be too time-intensive to manually analyze it. Hence we want to automate this analysis, and this is why the thesis is titled "Web Performance Optimization: Analytics".

We call this analysis *Episodes log mining*, which is a specialization of *web usage mining*. However, web usage mining is only designed to work with static data sets (that are updated in batches), while an Episodes log file is updated continuously: it should be considered a *data stream*. Hence *data stream mining* has also been studied: both frequent item mining and frequent item*set* mining algorithms have been looked into. However, frequent pattern mining algorithms can only find problems that persist over relatively long periods over time. We also want to detect brief problems, that are typically caused by traffic spikes; i.e. *infrequent issues*. To achieve this, *anomaly detection* has been investigated as well.

Finally, automatically detecting problems and presenting them to the user is great, but the user may also want to inspect all measurements himself. That can be achieved with *OLAP* techniques and more specifically the *data cube*, which is a data structure designed to be able to quickly answer queries about multidimensional data.

Preface

This thesis has grown from a custom proposal that I wrote, which continues where I left off with my bachelor thesis. My personal goal is to create a useful contribution to the upcoming field of *Web Performance Optimization*, to hopefully help the field move forward, towards a faster and more pleasant Internet experience.

I thank Steve Souders—evangelist and coiner of the term *Web Performance Optimization*, or *WPO* for short—for giving feedback on my initial suggestions on doing a meaningful master thesis in the WPO field. There likely is nobody in a better position to judge this than him.
It is my hope that choosing an idea that has his approval, maximizes the chance of it being a useful contribution to the field and it making a difference in real-world situations.

My deepest gratitude goes to my promotor, Prof. dr. Jan Van den Bussche, for making time in his already overfull schedule for guiding me through the various steps of this master thesis. Our rare—yet intense—meetings have often triggered my disbelief and raised eyebrows at his memorable thoroughness and insight. They brought interesting facts & trivia, and made my view on computer science broader. The end of office hours did not imply that it was time to stop a meeting. Even late at night in the weekend, I would get an e-mail explaining his interpretation on an algorithm. If I had to give one word to describe him, it would be *dedication*. I'm very grateful for his indispensable help.

Special thanks also go to Prof. dr. Benjamin Schrauwen, whom pointed me in the right direction when I was looking into anomaly detection literature. He saved me *a lot* of time.

Finally, I would like to thank my parents Etienne & Noëlla and my brother Tim, whose support has been invaluable.

Contents

1 Introduction

My bachelor thesis [1] was about making Drupal [2] web sites load faster. 80 to 90% of the response time (as observed by the end user) is spent on downloading the components of a web page [4]. Therefor this is also the part where optimizations have the largest effect—optimizing the code that renders the pages has far less effect.

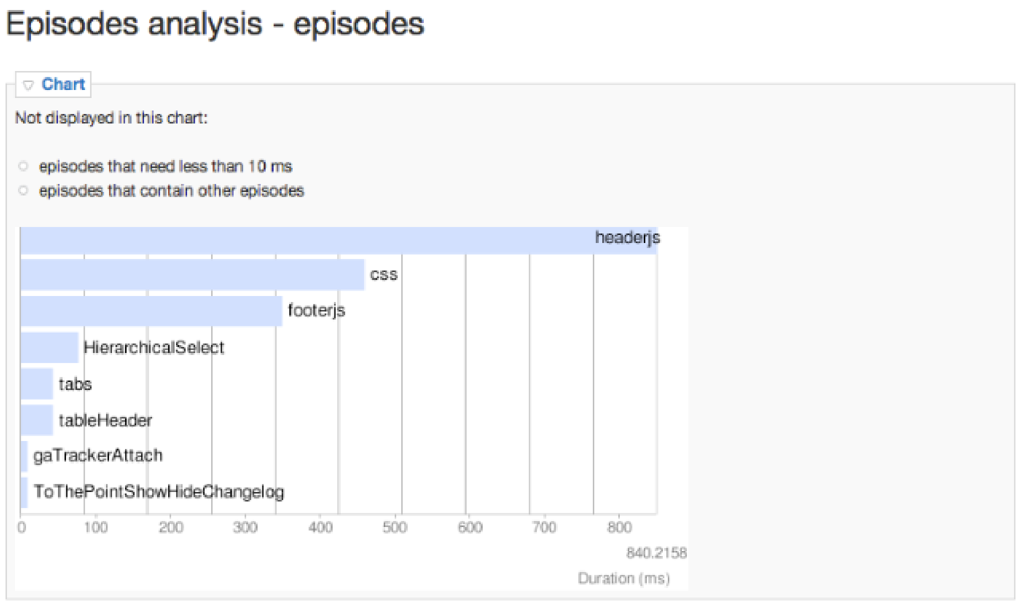
To be able to prove the positive impact of optimizing the loading of the components of a web site—thereby proving that the work I was going to have done had a positive impact—I researched existing page loading profiling tools. Episodes [5, 6] (which refers to the various *episodes* in the page loading sequence) came out as a clear winner:

- Episodes aims to become an industry standard;
- Episodes is open source;
- Episodes is a piece of JavaScript that runs in the browser on each loaded page, thus for each real visitor, thus it represents the real-world performance (all existing solutions [7, 8, 9, 10] require simulations, which implies they're also only suitable for simulating traffic on a new version of a web site before it goes live—they required simulations when I wrote my bachelor thesis in 2009, and still do at the time of writing this, in May 2010);
- Episodes does not require any hardware other than a server to log to.

Also as part of my bachelor thesis, I wrote a simple Drupal module—the Episodes module [11]—that could create simple charts to compare the average page loading time per day per geographic region. For my test case, with two weeks of collecting data, this was the resulting dataset:

> About two weeks, or 100 MB worth of statistics, had been logged. These were then imported on June 25, resulting in a database table of 642.4 MB. More than 2.7 million episodes were collected over more than 260,000 page views.

While my test case was a fairly big web site (500,000-1,000,000 page views per month), that is nothing when compared with the top-100 web sites. And even for these mere 2.7 million recorded episodes, it took several minutes to



Episodes analysis - episodes

▽ Chart

Not displayed in this chart:

○ episodes that need less than 10 ms
○ episodes that contain other episodes

Figure 1: Episodes analysis charts about episodes generated by the Drupal Episodes module.

generate simple charts (see figures 1 and 2). And that doesn't include the time for importing the log file into the database.

That is of course largely due to the fact that the database schema used was extremely inefficient: it was in fact a verbatim copy of the log file. The database schema should be optimized for the queries that are necessary to generate the charts. In that implementation, multiple full table scans were required, which is something that should be absolutely avoided when building an application on top of an RDBMS, because it guarantees poor performance.

Despite its obvious (intended) lack of optimizations, it was sufficient to prove that File Conveyor [3]—the daemon that I wrote to automatically sync files to any CDN, regardlesss of the file transfer protocol used—when integrated with a Drupal web site and thus providing CDN integration for that web site, had a positive impact: the test web site consistently loaded about *twice as fast*, especially for visitors with slower internet connections, such as visitors from Brazil. Without this proof-of-concept implementation, I would never have been able to prove the positive impact on performance.

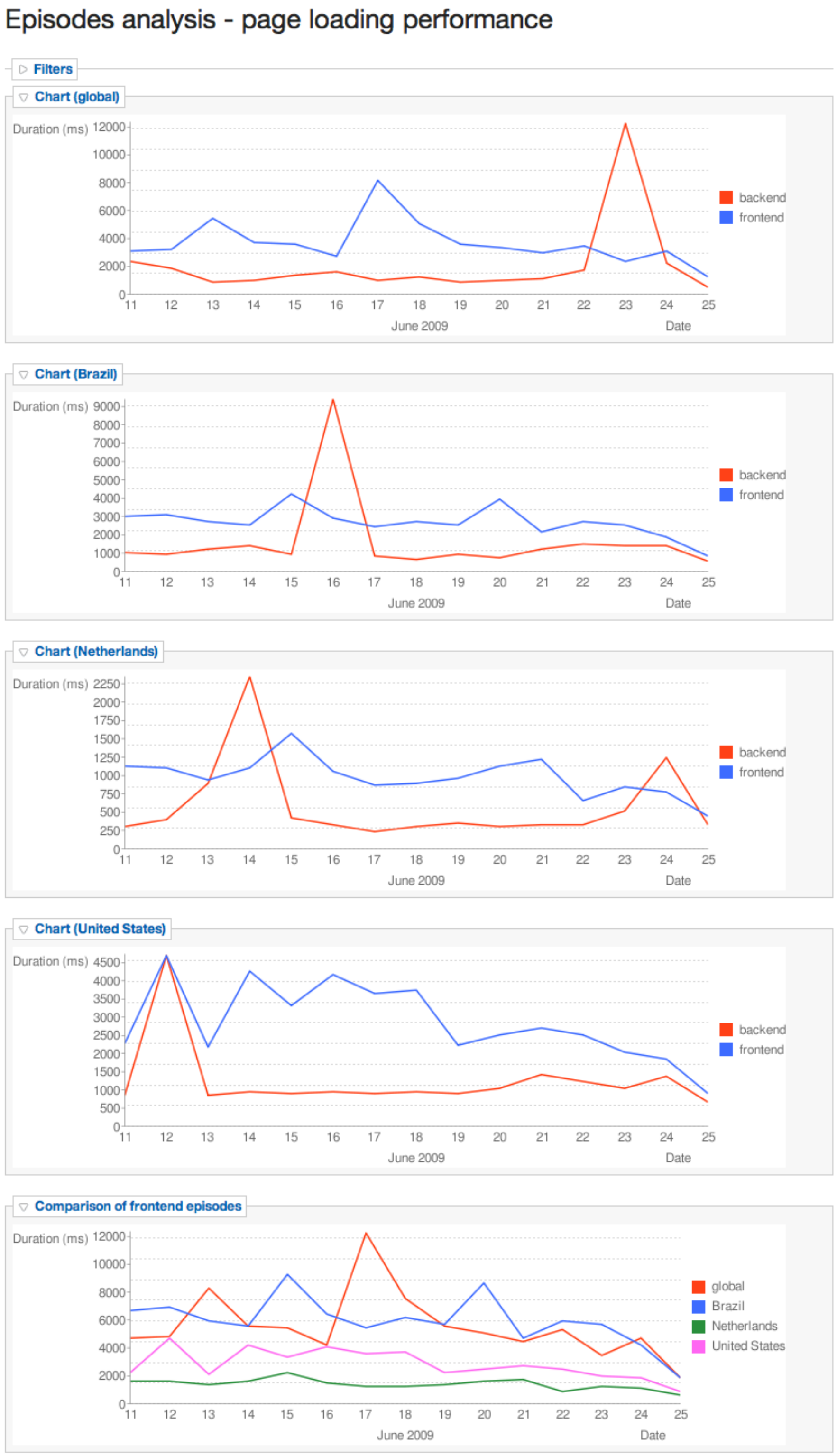


Figure 2: Episodes analysis charts about page loading performance generated by the Drupal Episodes module.

1.1 Continuous Profiling

The main problem is that sites are too slow. In my bachelor thesis, I implemented a daemon to synchronize files to a CDN, which is one of the most important ways to speed up the loading of a web site.

However, simply implementing all known tricks is not enough, because using a CDN might speed up your web site for half your visitors and slow it down for the other half—although that is an extremely unlikely scenario. That is why you need to be able to do Continuous Profiling (cfr. Continuous Integration).

Continuous Profiling means that you are continuously monitoring your real-world web performance: you must track the page loading characteristics of each loaded page! That by itself is easy: all it requires is to integrate Episodes with your web site. The actual problem lies in analyzing the collected data. To be able to draw meaningful conclusions from the collected data, we need to apply data mining techniques as well as visualizing the conclusions that are found. E.g. pages may be loading slower from South-Africa because the CDN's server there (a *PoP*) is offline, or your shopping cart checkout page may be loading slow in Firefox because of a JavaScript issue, or a particular page may be loading slow in all web browsers because of bad CSS on that page, or maybe your site is loading very slow for all users of a certain ISP because their DNS server has poor performance. All of these problems (and more) could be pinpointed (albeit partially) automatically.

Hence, that is what the goal is of this thesis: to build something like Google Analytics, but for *web performance (page loading performance)* instead of just *page loads*. An analytics suite for tracking web performance. An application that can automatically extract conclusions out of Episodes logs and visualize them. This application should be very scalable (as the number of recorded episodes is typically an order of magnitude higher than the number of page views) and possibly also distributed. You should also be able to go back to any point in the past and view the web performance at that time. Thus, efficient storage is also a requirement. Finally, it should be an open source application that can be developed further by others after I finish my master thesis.

I told Steve Souders about my idea for my master thesis—he is the most prominent speaker, researcher and evangelizer in the web performance optimization scene and on Google's payroll to push this forward—and asked him for feedback. His response:

> I did a mini performance conference in LA last month and

> heard three big companies (Shopzilla, Edmunds, and Google PicasaWeb) get up and say they had regressed in their web site performance because they weren't tracking latency. I realized that most companies aren't even at the point where they have good metrics. I think the first idea—Google Analytics for latency—is the best idea. [...] It would be great if this lived on Google AppEngine. Users could take the code and spin up their own instance—for free! You could also host a shared instance. I will say that the work [...] on AppEngine has been hard because of the datastore—my officemate does the programming and it is taken him months to do what I did in a few days on the LAMP stack.

He agrees on the necessity for such an application and immediately proposes to make it run on Google AppEngine [24], which is a free platform for web applications with its own, apparently complicated, datastore that is schemaless. The idea is that anybody can create a free AppEngine account, install this application and get a Continuous Profiling application for free!

Whether it would run or Google AppEngine or not, it is certain that an open source Continuous page loading performance profiling would be very valuable, which is exactly what I'll try to build for my master thesis.

1.2 Context

Ever since Steve Souders' *High Performance Web Sites* book [4], interest in making web sites load faster has been increasing. More and more big companies with a strong *web presence* are paying attention to page loading performance: the well-known ones such as Microsoft, Yahoo, Google, but also big companies that are not technology companies such as Amazon, White Pages, Shopzilla, Edmunds, Netflix ...

Page Loading Profiling Tools

As a result of this trend, a large number of advanced page loading profiling tools are being developed:

- Deep tracing of the internals of Internet Explorer, by using dynaTrace Ajax [12]

- JavaScript memory heap profiler and sample-based CPU profiler in WebKit/Google Chrome [13]
- Firefox has been leading the way with the development of the Firebug extension and the Yahoo! YSlow [14] & Google Page Speed [15] Firebug plug-ins

Proposals

Recent proposals (in the last three months of 2009 alone) for web performance optimization include:

- SPDY [16], a new application-level protocol that learns from the mistakes of HTTP (which is ten years old). This protocol specification is currently in draft state, but tests of the researchers (at Google) show that pages of the top 25 web sites loaded up to 55% faster.
- Resource Packages [17, 18]. A resource package is a zip file that bundles multiple resources into a single file and therefor requires only a single HTTP response and avoids multiple round trip delays. Browsers typically only take advantage of about 30% of their bandwidth capacity because of the overhead of HTTP and TCP and the various blocking behaviors in browsers. This proposal would result in less bandwidth being consumed by overhead. Plus, it is backwards compatible: browsers that don't support it load the page the same way as today.
- Web Timing [19]. This is a proposal presented to the W3C and welcomes feedback from browser vendors. It effectively means that Episodes is being moved into the actual browser partially, to get rid of the latency of loading Episodes' JavaScript and the relatively inaccurate time measurements of JavaScript. It would also allow us to get a complete picture of the end-to-end latency, which is impossible to do with Episodes (which can only rely on what JavaScript can do). This proposal is only a working draft and requires interacting with browser vendors to ensure all current major browsers will implement this. Even in the best case scenario, it will take *years* until the majority of the installed browsers will support this. Until then, we will be limited in what we can measure. Hence this proposal should move forward as fast as possible.

All of these would strongly affect browser implementations, which indicates the willingness and likeliness to change the way data is transferred over the internet to make web sites load faster.

Search Engine Ranking

The importance of web performance is lifted to an even higher level by the fact that Google is now using the page loading performance (they call it "page speed" or "site speed") of a web page to calculate its ranking.

They announced that they would likely let page speed influence the ranking of web pages in December 2009 [20] and activated it in April 2010. This effectively means that all companies whom have been paying for SEO (search engine optimization) will also have to consider web performance optimization.

1.3 Conclusion

Given the aforementioned context, it is clear that the importance of web performance optimization is only growing. And as it grows, the need for tools that can assist developers in making the right decisions of course also grows. Because new performance issues may occur at any point of time, there is a need for continuous profiling.

That's why it is my goal to build a tool that can be used for continuous profiling that, if well-written, can become a very useful tool in the day-to-day life of the web developer, to help keep the web developer's live web sites loading fast. It *could* make a real difference, and that is what I'm aiming for.

2 Justification of Literature Study Subjects

Episodes Log Mining

For *Episodes log mining*, (see the next section), I have used *web usage mining* as a basis. However, it was clear that this would be too "applied" to qualify as a true member of this literature study. Hence it was decided to postpone the full details to part two of this thesis.

Data Stream Mining

The main task of this thesis consists of mining patterns in the Episodes log file. However, this Episodes log file is continuously being updated: new log entries are appended as pages are being viewed on the web site. So we are in fact not dealing with a static data set that occasionally receives a batch of new log entries: we are dealing with a *data stream*! Therefore, data stream mining is precisely what is needed; more specifically: *frequent pattern stream mining*, because from there it is a simple step to association rules, which are exactly what we need. This is discussed in section 4.

Anomaly Detection

Data stream mining can only find frequently occuring patterns, because that is exactly what frequent pattern mining is about. However, we also want to be able to detect occasional spikes instead of just the persistent problems. For example, spikes may occur only on the first day of the month (because people can enter the monthly contest on that day), which the web server may not be able to cope with properly. Detecting these *infrequent problems* is exactly what *anomaly detection* is for. Anomaly detection is discussed in section 5.

OLAP

OLAP (see section 6), and more specifically the *data cube*, is necessary to be able to quickly answer queries about multidimensional data. The data that needs to be presented to the user (and browsed, queried, interacted with) in the context of web performance optimization is very multidimensional, as is explained in section 3.2.

3 Episodes Log Mining

3.1 Introduction

3.1.1 Web Usage Mining

Episodes log mining is a specialized form of *web usage mining*, which in turn is a type of *web mining*. But what is *web mining*? According to [27]:

> **Web mining aims to discover useful information or knowledge from** the web hyperlink structure, page content and **usage data**. Although web mining uses many data mining techniques, it is n**ot purely an application of traditional data mining due to the hetereogeneity and semi-structured or unstructured nature of the web data**. Many new mining tasks and algorithms were invented in the past decade. Based on the primary kinds of data used in the mining process, web mining tasks can be categorized into three types: web structure mining, web content mining and **web usage mining**.

The web mining process is similar to the traditional data mining process, however, there usually is a difference in the data collection step. In traditional data mining, the data is often already collected (and stored in a data warehouse). In the cases of web structure mining and web content mining, collecting data can be a large and daunting undertaking. Fortunately, in the case of web usage mining, it is fairly simple: most web servers keep log files already (e.g. Apache server logs).
And as indicated at the beginning of this section, it is only web usage mining that we need, the other types of web mining are irrelevant for this thesis.

Again according to [27], web *usage* mining is:

> **Web usage mining** refers to the automatic discovery and analysis of patterns in clickstream and associated data colected or generated as a result of user interactions with web resources on one or more web sites. **The goal is to capture, model and analyze the behavioral patterns** and profiles of users interacting with a web site. The **discovered patterns** are usually represented as **collections of pages**, objects, or resources **that are frequently accessed by groups of users with common needs or interests**.

3.1.2 Web Usage Mining Versus Episodes Log Mining

However, in the context of web performance optimization analytics (which is what this thesis is about), typical web server logs are not sufficient: they only capture which resources were requested by user agents and some metadata (date and time, IP address, referrer, etc.). That is by itself not enough information on how the actual page loading performance of the browser as perceived by the end user was. It only provides sufficient information for other kinds of analysis, such as typical navigation paths, popular pages, and so on.
While that is interesting in itself and *can* be useful for suggesting advanced page loading performance improvements, it doesn't provide enough information to be able to perform page loading performance analysis.

That is why Episodes was developed. As explained earlier, Episodes records the durations of the various episodes during the loading of the page and when the page has finished loading, it sends this information to a web server log. It does this by means of a specially formatted URL—the URL contains the names and durations of the recorded episodes (in order of being recorded and occurrence) as a single long GET parameter. This GET parameter can then be parsed to easily extract the episodes that were recorded.

The additional information that is virtually always included in web server log files, such as IP address, date and time and user agent can then be used to apply web performance optimization analysis: IP addresses can be mapped to locations/ISPs to pinpoint bad performance to a specific location/ISP, date and time can be used to detect bad performance during specific times during the day (indicating overloaded web or application servers) and finally the browser and operating system can be used to detect performance issues with a specific browser, possibly a specific version of that browser and even on a specific operating system.

And, of course, any web performance issues that are a combination of the above can also be detected: web performance problems that only occur for a specific browser/ISP combination, for example (which might be caused by a badly configured web proxy server for example).

Examples

Examples of web performance issues that need to be detected automatically are, for example:

- ***http://example.com/*** *is slow in* ***Belgium****, for users of the ISP* ***Telenet***
- ***http://example.com/path*** *and all pages in this directory have slowly loading* ***CSS***
- ***http://example.com/path/b*** *has slowly loading* ***JS*** *for visitors that use the browser* ***Internet Explorer 6 or 7***

The Definition of 'Slow'

Of course, "slow" is a subjective quality. There are many possible methods for defining "slow". Here are two examples (*both* of which I would like to support in the implementation that will accompany this thesis):

1. The top x percent slowest episodes are considered "slow". x would be configurable by the user.
2. There is a threshold y defined for each episode; durations for this episode higher than y would be marked slow.

Analogously, one could define *multiple* "speeds": very slow, slow, acceptable, fast, very fast, for example. This would need to come with sane defaults, but should be configurable by the user in the end.

Note that in the first example definition for "slow", the threshold for a "slow" episode constantly changes, as new episodes are being added. This can be worked around by using *data stream mining*, as opposed to "regular" data mining (see section 4).

3.1.3 The Mining Process

The overall web usage mining process (and therefore Episodes log mining, which is merely a specialization) can be seen as a three-stage process. Below I have provided a high-level comparison of the differences between web usage mining and Episodes log mining.

1. **data collection and pre-processing**
    - Web usage mining: this would consist of partitioning the log entries into a set of user transactions. In pre-processing, knowledge

  about the site content or structure, or semantic domain knowledge (from the used ontologies) may be used to enhance the transaction data.

- Episodes log mining: here, it is quite different: data collection is not an issue; and pre-processing consists of mapping the IP address of each log entry to a location and an ISP (if possible), extracting the various episodes from the specially formatted URL, normalizing the user agent string, and so on.
  The data collection has already been implemented in [1] (as already indicated in section **??**). *The pre-processing will be implemented as part of this master thesis.*

2. **pattern discovery**

   - Web usage mining: find hidden patterns reflecting typical behavior of users and generate summary statistics on components, sessions and users.
   - Episodes log mining: find hidden patterns related to web performance and summary statistics such as average page loading time per country or browser.
     The discovery of these patterns will be implemented as part of this master thesis. For this, data stream mining—see section 4—and anomaly detection—see section 5—will be used.

3. **pattern analysis**

   - Web usage mining: the discovered patterns and statistics are further processed, filtered, and then used in recommendation engines, visualization tools or analytics/report generation tools.
   - Episodes log mining: the discovered patterns and statistics are displayed in a tool that provides visualizations and automatically makes suggestions as how to solve automatically detected web performance issues.
     This visualization tool will be implemented as part of this master thesis. For this, OLAP will be used—see section 6.

Simultaneously, this overview of course also gives a high-level idea of what the implementation that will accompany this thesis will entail.

3.2 The Attributes

As explained before, essentially the goal of this thesis is analyzing Episodes log files. Each log entry is stored in a format which has been optimized to store only the information that ever might be useful for Episodes log mining instead of regular web usage mining. The format is as follows:

```
211.138.37.206 [Sunday, 21-Jun-2009 06:23:37
   +0200] "?ets=css:63,headerjs:4453,footerjs:16,
   domready: 7359,tabs:31,
   ToThePointShowHideChangelog:0,gaTrackerAttach
   :16,DrupalBehaviors:47,frontend:8015" 200 "http
   :// driverpacks.net/applications" "Mozilla/4.0
   (compatible; MSIE 6.0; Windows NT 5.1; SV1; (R1
    1.6); .NET CLR 2.0.50727)" "driverpacks.net"
```

Each such log entry (of which there is one for each pageview!) can be transformed into a long list of categorical attributes: IP address, location (by mapping the IP address to a location), date, episode names, browser, operating system, and so on. There also is an important list of numerical attributes: the episode durations.

Episodes log mining in general and these attributes in specific of course need more explaining, but it cannot be considered literature study, since it only requires a very “applied” text to be meaningful. General web usage mining is irrelevant to this thesis, since that is mostly about finding patterns in visitor’s activities, which is something we don’t care about from a web performance optimization perspective.
Therefore, these additional explanations will be added in part two of this thesis.

4 Data Stream Mining

This section is based mostly on [41, 46], at least for the introduction and general information about the various methodologies. The details about the various algorithms originates from their corresponding original (or related) papers.

To achieve this, we must dive deeper into the field of *data stream mining*. The goals are the same as for data mining, but the difference is that we do not operate on a fixed set of data, but on a *stream* of incoming data, that is generated *continuously*, and with varying update rates. Data streams are *temporally ordered*, *fast changing*, *massive*, and *potentially infinite*. Because not all data is known before starting the mining process, and because the size of the (stream of) data is potentially infinite, this implies that we can no longer use algorithms that require multiple scans: instead, it is necessary to use single-scan algorithms (it may even be impossible to store the entire data stream).
Even for non-stream data this may be necessary: if the dataset is so enormous that it is not feasible to perform multiple scans (e.g. when one needs to perform Episodes log mining on months worth of Episodes logs), then algorithms developed for data streams are equally applicable.

4.1 Methodologies for Stream Data Processing

As discussed before, it is impractical (or even unrealistic) to scan through an entire data stream multiple times—sometimes it even might be impossible to evaluate every element of the stream due to the update rate. The size of the data is not the only problem: the universes[1] that need to be tracked can be very large as well (e.g. the universe of all IP addresses is enormous).

Clearly, new data structures, techniques and algorithms are needed for effective processing of stream data. Because it is impossible to store all stream data (which would require an infinite amount of storage space), it is often necessary to consider a trade-off: accuracy versus storage. In other words: approximate instead of exact answers are often sufficiently accurate.

Synopses can be used to calculate approximate answers, by providing *summaries* of data: they use *synopsis data structures*, which are data structures that are significantly smaller than their base data set (here: stream data). We want our algorithms to be efficient both in space and time. Instead of

---

[1] A universe is the domain of possible values for an attribute.

storing all elements seen so far (requires $O(N)$ space), it is more desirable to only use polylogarithmic space ($O(\log^k N)$).

The synopses below are explained succinctly, either because they're fairly easy to comprehend or because explaining them in-depth would lead us too far.

4.1.1 Random Sampling

Rather than storing (and processing) the entire data stream, another option is to sample the stream at periodic intervals. However, to obtain an unbiased sampling of the data, it is necessary to know the length of the stream in advance, to determine the periodic interval. But for many data streams it is impossible to know the length, or indeed it will be infinite. Hence another approach is necessary.

An alternative method is *reservoir sampling*: it achieves an unbiased sample by selecting s elements randomly and without replacement. In reservoir sampling, a sample of size *at least* s is maintained, which is called the reservoir. From this reservoir, a random sample of size s can be generated. To avoid the cost of generating a sample from the possibly large reservoir, a set of s *candidates* in the reservoir is maintained. These candidates form a true random sample of the elements seen so far in the stream.

As new data flows in from the data stream, every new element in the stream can replace a random old element in the reservoir with the probability $\frac{s}{N}$.

4.1.2 Sliding Windows

Instead of working with all data ever flown in through the data stream, we make decisions based only on *recent data*. More formally: the element that arrives at time t expires at time $t + w$, with w the window size.

4.1.3 Histograms

A histogram is a synopsis data structure, which can be used to approximate the frequency distribution of element values in a a stream. It partitions the data into a set of contiguous buckets. Various partition rules are possible, among which *equal-width* (equal value range for all buckets) and *V-Optimal* (minimizes the frequency variance within each bucket, which better captures the distribution of the data).

However, histograms require at least two passes: at least one to decide the size of the buckets and then another to associate each value with a bucket. This makes histograms unsuitable for use with data streams.

4.1.4 Multiresolution Methods

A multiresolution method is an example of a data reduction method—a data reduction method can be used to achieve smaller data storage requirements, yet closely maintain the integrity of the original data.
Multiresolution methods also offer, on top of the aforementioned, the ability to look at the data stream in multiple levels of detail, which may be a desirable property when processing a data stream.

We look at one example of a multiresolution data reduction method: wavelets.

Wavelets

Wavelets are a technique from the field of signal processing, but can also be used to build a multiresolution hierarchy over a signal, which would be the data stream in our case. Wavelets coefficients are projections of the given signal (again, the data stream in our case) onto an orthogonal set of basis vector. Which wavelets can be used depends on the choice of basis vectors. Using the Haar wavelet (often chosen for their ease of computation) for example, we can recursively perform averaging and differencing at multiple levels of resolution.

An example of the one-dimensional Haar wavelet should clarify this. Let A be a one-dimensional data vector, with $A = [22, 14, 16, 12]$. We now first average each pair of values to get a new data vector with a "lower resolution": $A' = [\frac{22+14}{2}, \frac{16+12}{2}] = [18, 14]$. Clearly we cannot generate A from A': not enough information is available. To be able to restore the original values, we need to store the *detail coefficients*, which capture the information that has been lost. For Haar wavelets, these are simply the differences of the second original value with the averaged value, in our example that would be: $18 - 14 = 4$ and $14 - 12 = 2$. Note that it now is possible to restore the original four values. If we now apply this process of averaging and differencing recursively, we get the following full decomposition:

Resolution	Averages	Detail coefficients
2	$[22, 14, 16, 12]$	n/a
1	$[18, 14]$	$[4, 2]$
0	$[16]$	$[2]$

The *wavelet transform of A* (or *wavelet decomposition*) is defined to be the single coefficient representing the overall average of the values in A, followed by the detail coefficients in the order of increasing resolution. Thus, the Haar wavelet transform of A is $W_A = [16, 2, 4, 2]$. Each entry in W_A is called a *wavelet coefficient*.

We can then achieve a more compact data representation by either only including the lower resolution detail coefficients or by applying compression techniques such as run-length encoding (run-length encoding [51] can be applied because the information is statistically concentrated in just a few coefficients).

Wavelets have been used as approximations to histograms for query optimizations [42].

Unfortunately, wavelets also require multiple passes, rendering them too unsuitable for use with data streams.

4.1.5 Sketches

The aforementioned techniques either focus on a small partition of the data (sampling & sliding windows) or summarize the entire data (histograms), possibly at multiple resolutions (wavelets).

A histogram requires multiple passes and stores only a single resolution. A wavelet is an approximation of a histogram also requires multiple passes but can store multiple resolutions. Next in that row is a *sketch*: it can maintain an approximation of a full histogram in a single pass, and if desired can be used to store multiple resolutions.

A sketch can be used to maintain the full histogram over the universe of elements in a data stream in a single pass. Define the universe as $U = \{1, 2, \ldots, v\}$ (with v the universe size) and the elements in the data stream as $A = \{a_1, a_2, \ldots, a_N\}$ (with possibly $N = \infty$). For each value i in the universe, we want to maintain the frequency of i in the sequence of elements A. If the universe is large, the required amount of storage can be large as well. To achieve a smaller representation, we consider the *frequency moments* of A. These are the numbers F_k:

$$F_k = \sum_{i=1}^{v} m_i^k$$

where m_i is the frequency of i in the sequence and $k \geq 0$.

This can be interpreted as follows. Each example result is calculated over the sequence `131113342`.

- F_0 is the number of distinct elements in the sequence, i.e.: $0 \leq F_0 \leq v$.
  Applied to the example: $F_0 = 4$.
- F_1 is the length of the sequence, i.e.: $F_1 = N$.
  Applied to the example: $F_1 = 4 + 1 + 3 + 1 = 9$.
- F_2 is the so-called *self-join size*[2], or also known as *repeat rate* or *Gini's index of homogeneity.*
  Applied to the example: $F_2 = 4^2 + 1^2 + 3^2 + 1^2 = 27$.

The frequency moments of a data stream (or any data set of fixed size) provide useful information about this data for database applications, one of which is the *skew* (or asymmetry) of the data. The skew can be used to decide how to partition the data set for parallel or distributed database systems.

When the amount of available memory is smaller than v (the universe size), we need to employ a synopsis. The estimation of the frequency moments can be performed by *sketches*, which build a summary (requiring less space) for a distribution vector (e.g. a histogram) using randomized linear projections (i.e. linear hash functions) of the data they are fed (i.e. the data stream). Sketches provide probabilistic guarantees on the quality of the approximate answer. For example: the answer to the given query is 93±1 with a 95% probability. Given N elements and a universe U of v values, such sketches can approximate F_0, F_1 and F_2 in $O(\log v + \log N)$ space [43].

The most complex and interesting sketch is the one for approximating F_2, thus only that one will be explained more in-depth here.
The key idea behind the F_2 sketching technique is as follows: *every element i in the domain D is hashed uniformly at random onto a value $z_i \in \{-1, +1\}$. Define the random variable $X = \sum_i m_i z_i$ and return X^2 as the estimator of F_2.* Clearly, this estimator can be calculated in a single pass. Note that we do not actually calculate m_i in the formula for X: each time we encounter i, we just *update* X by adding another iteration of $m_i z_i$ (which is why it can work in a single pass). Hashing can be used because the actual value of each i is irrelevant: we only want to know the frequency.
To explain why this works, we can think of hashing elements to either -1

---

[2] The self-join size F_2 is also used to estimate the join size for RDBMSes in limited space, see [47].

or $+1$ as assigning each element value to an arbitrary side of a tug of war. When we sum up to get X, we can think of measuring the displacement of the rope from the center point. By squaring X, we square this displacement, thereby capturing the data skew F_2.

The sketching technique to compute F_0 was presented in [70] (which is referenced again in section 6.7.2), however, this required explicit families of hash functions with very strong independence properties. In [43], this requirement was relaxed; it explains how F_0, F_1 and F_2 can be approximated in logarithmic space by using linear hash functions (which is why sketches hold an advantage over wavelets in terms of storage). A single pass algorithm for calculating the k-th frequency moment of a data stream for any real $k > 2$ is given in [44], with an update time of $O(1)$. Finally, in [45], a simpler algorithm (but with the same properties) is given. Another interesting sketching method is given in [52].

4.1.6 Randomized Algorithms

Random sampling and sketching are examples of randomized algorithms.

Randomized algorithms that always return the correct answer but whose running times vary are known as *Las Vegas algorithms*. In contrast, there are also randomized algorithms that are bounded on running time but may not return the correct answer; these are called *Monte Carlo* algorithms.

In the context of data stream mining, where the time to process incoming data is obviously limited, we consider mainly Monte Carlo algorithms. A randomized algorithm can be thought of as simply a probability distribution over a set of deterministic algorithms.

4.2 Frequent Item Mining

A typical goal in data mining is *pattern mining*, from which it is easy to generate association rules. Association rules describe correlations between items, such as “people who buy both milk and beer also tend to buy diapers with 70% probability”. To find meaningful patterns, it is necessary to find which itemsets occur frequently in a dataset, where an itemset is considered frequent if its count satisfies a *minimum support*.

Fast algorithms for mining frequent itemsets have been developed for *static data sets*, such as Apriori and FP-growth. However, mining itemsets in *dynamic data sets* (i.e. data streams) creates a whole new set of challenges. Existing algorithms such as Apriori [60] and FP-growth [61] (and many others) depend on the ability to scan the entire data set (which may impossible for data streams, since they might be infinite), *and* typically require multiple passes. So how can we perform incremental updates of frequent itemsets, while an infrequent itemset can become frequent at a later point in the data stream, and vice versa? The number of infrequent itemsets also is exponential, which makes it impossible to track all of them[3]. Thus, a synopsis data structure (as explained in section 4.1) is obviously needed, or more accurately: an algorithm that builds such a data structure.

There are two possible approaches to overcome this difficulty:

1. Only keep track of a *predefined*, limited set of item(set)s. This method of course has very limited usage, because it will be unable to find frequent item(set)s beyond the predefined scope.
2. Derive an *approximate* answer—while this won’t be 100% correct, it is often sufficient in practice.

Now, an itemset of course consists of items. Hence we will focus in *frequent item mining* algorithms in this section and then look into *frequent itemset mining* algorithms in the next. Note that by *frequent* item counting, we are actually referring to *highly frequent* item counting. In the field of network traffic flows, the problem of finding the largest traffic flows is also known as *the heavy hitter problem* [56], so *frequent item mining algorithms* are sometimes also called *heavy hitter algorithms.*

All algorithms in this section and the next provide *approximate* answers.

---

[3] It has been shown [48] that it is impossible to find the exact frequency of frequent items using an amount of memory resources that is sublinear to the number of distinct elements.

4.2.1 Window Models

A data stream consist of elements, i.e. item(set)s, which arrive in a particular order over time. There are several ways one can deal with this sequence nature, existing models are [58]:

1. The *landmark* model: frequent item(set)s are mined in data streams by assuming the item(set)s are measured from the beginning of the stream until the current moment.
   This model may not be desirable when *changes of patterns (itemsets)* and their *trends* are more interesting than the patterns themselves. E.g. a series of shopping transactions could start a long time ago (e.g. a few years ago), but patterns found over the entire time span may be uninteresting due to fashion, seasonal changes, and so on.
2. The *sliding window* model: frequent item(set)s are mined over only the last w transactions, with w the window size.
3. The *tilted-time window* model: frequent item(set)s are mined over the last w transactions, but only the most recent frequent item(set)s are stored at fine granularity—frequent item(set)s in the past are stored at coarser granularity.
4. The *damped window* model: a decay function is applied to the data stream, to give more weight to recent data than to old data.

If this wasn't clear yet: this classification is both applicable to both single items (which is discussed in this section) and itemsets (discussed in section 4.3).

All frequent item mining algorithms in the remainder of this section are of the *landmark window* model, the window model for the frequent itemset mining algorithms in section 4.3 vary and are indicated on a per-algorithm basis.

Tilted-Time Window

The tilted-time window model needs a little bit more explaining.

The design of the tilted-time window is based on the fact that often the *details* of recent changes are interesting, but over a longer period, *less detail is necessary.*

Several ways exist to designed a tilted-time window. Here are two common examples:

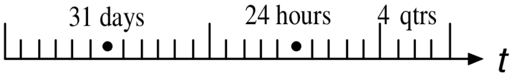


Figure 3: Natural tiled-time window.

(Figure courtesy of [58].)

1. *Natural tilted-time window model.* [41, 58] The time window is structured in multiple granularities, based on the "natural" (for humans) time scale: the most recent 4 quarters of an hour, then the last 24 hours, 31 days and then 12 months. This can of course vary, depending on the application. See figure 3 for an example of what that looks like.
   Based on this model, we can compute frequent itemsets in the last hour with the precision of a quarter of an hour, the last day with the precision of an hour, and so on. This model registers only $4 + 24 + 31 + 12 = 71$ units of time (quarters, hours, days, months respectively) instead of $365 \times 24 \times 4 = 35,040$ units of time—with the trade-off of coarser granularity for the distant past.

2. *Logarithmic tilted-time window model.* [41] In this model, the granularity decreases towards the past at an exponential rate. If the most recent slot holds data for the last quarter, then the one before that also holds data for one quarter (the one before the most recent), then for 2 quarters, 4, 8, 16, and so on. In this model, only $\lceil \log_2(365 \times 24 \times 4) + 1 \rceil = \lceil 16.1 \rceil = 17$ units of time are needed.

4.2.2 Algorithm Classification

Currently known frequent item mining algorithms all rely on one of three basic techniques [50, 55]:

- Counting: lossy counting (4.2.7), Karp/Demaine algorithm [59], probabilistic lossy counting (4.2.9). In general, they use 1) a fixed or bounded number of counters for tracking the size of frequent elements and 2) a condition to periodically delete or reallocate counters of infrequent elments.
  Counting algorithms have low per-element overhead, as they only require incrementing a counter, along with a potentially high periodic housekeeping step that may sort and delete counters.

- Hashing: count sketch (4.2.8), min-count sketch. They all use variants of the same data structure, which most of them call a *sketch*, which is a one- or two-dimensional array of hash buckets.
  Hashing algorithms use fixed memory resources to estimate the frequency of an arbitrary element of a data stream and provide probabilistic guarantees on the estimation errors.

- Sampling: basic sampling (4.2.3), concise sampling (4.2.4), counting sampling (4.2.5), sticky sampling (4.2.6).
  Sampling algorithms reduce the required memory resources and the processing overhead for identifying frequent items. The downside is that they typically have a lower estimation accuracy.

The algorithms are explained in the next subsections, in order of being published—with the additional goal of providing an (approximation of) the timeline over which new, improved algorithms have been invented.
By including the older algorithms upon which the newer ones are based, it also becomes more clear how we ended up with the current state-of-the-art algorithms.

4.2.3 Basic Sampling

Note that this algorithm is the most basic sampling algorithm [55] and that other algorithms such as concise sampling (see section 4.2.4), count sampling (see section 4.2.5) and sticky sampling (see section 4.2.6) build upon it. It requires the size of the data set to be known in advance, which renders it useless for use with data streams. It is only listed here fore reference.

This algorithm is the most straightforward solution for counting item frequencies: it keeps a uniform random sample of the elements, stored as a list $\mathcal{L}$ of items, with a counter for each item. If the same element is added multiple times, its counter is incremented (the element is not added multiple times to $\mathcal{L}$).

If x is the size of the sample (counting repetitions) and N the size of the data set, then the probability of being included in the sample is $\frac{x}{N}$, the count of the k^{th} most frequent element is denoted n_k (i.e. $n_1 \geq n_2 \geq \ldots \geq n_k \geq \ldots \geq n_m$) and let $f_i = \frac{n_i}{N}$. To guarantee that all top k elements will be in the sample, we need $\frac{x}{N} > O(\log \frac{N}{n_k})$, thus $x > O(\log \frac{N}{f_k})$.

4.2.4 Concise Sampling

This is a variant of the basic sampling algorithm given in section 4.2.3. Introduced by P. B. Gibbons and Y. Matias in 1998 [49], the concise sampling algorithm keeps a uniformly random sample of the data, but does not assume that the length of the data set is known beforehand (which the general sampling algorithm of section 4.2.3 does assume), hence making this algorithm suitable for use with data streams.

Again a list of items with a counter for each item is kept, i.e. a list $\mathcal{L}$ of (e, c) pairs with e the element and c its count.

It begins optimistically, assuming that we can include elements in the sample with probability $\frac{1}{r}$, with threshold $r = 1$. As it runs out of space, the threshold r is increased to r' repeatedly; until some element is deleted from the sample: each of the sample points in $\mathcal{L}$ is evicted with probability $\frac{r}{r'}$. We then continue with this new, higher r'.

The invariant of the algorithm is that at any point, each item is in the sample with the current probability $\frac{1}{r_c}$. At the end of the algorithm (i.e. the end of the data stream, if there is an end), there is some final probability $\frac{1}{r_f}$.

No clean theoretical bound for this algorithm is available: it can only be calculated for specific distributions. E.g. for exponential distributions, the advantage is exponential: this means that the sample size is exponentially larger than the memory footprint for this sample size.

Note: the reader familiar with data compression techniques may have aptly noted that this is indeed very similar to the simple, yet widely utilized run-length encoding technique [51]!

4.2.5 Counting Sampling

Counting sampling is merely a small optimization to concise sampling (and is discussed in the same paper by P. B. Gibbons and Y. Matias from 1998 [49]); it is based on the simple observation that so long as space is set aside for a count of an item in the sample anyway, we may as well keep an exact count for the occurrences.

This change improves the accuracy of the counts of items, but does not change which elements will actually get included in the sample.

Since this is only an optimization and the essence of the concise sampling remains untouched, no clean theoretical bound on the space complexity of this algorithm exists either.

4.2.6 Sticky Sampling

The *sticky sampling* algorithm is an enhanced version of the counting sampling algorithm. The difference is that in sticky sampling, the sampling rate r increases logarithmically, proportional to the size of the stream. Additionally, it guarantees to produce all items whose frequency exceed a user-specified minimum support parameter s, instead of just the top k. The user can also specify an acceptable error margin $\epsilon \in [0, 1]$ and an acceptable probability of failure $\delta \in [0, 1]$ to meet this error margin.
It was presented in 2002 by G. S. Manku and R. Motwani [53].

Guarantees

A very clear set of guarantees is given for this algorithm:

1. All items whose true frequency exceeds sN are output. There are *no false negatives.*
2. No items whose true frequency is less than $(s - \epsilon)N$ are output.
3. Estimated frequencies are *less* than the true frequencies *by at most* ϵN with probability $1 - \delta$.

We say that the algorithm maintains an *ϵ-deficient synopsis* if its output satisfies these guarantees.

Guarantees Example

For example, if the goal is to identify all items whose frequency is at least 1%, then $s = 1\%$. The user is allowed to set the error margin ϵ to whatever value is considered acceptable. Let's assume a 5% margin of error is acceptable, then $\epsilon = 0.05\% = 5\% \times s$. Then, as per guarantee 1, all elements with frequency exceeding $s = 1\%$ will be output, and there will be no false negatives. As per guarantee 2, no element with frequency below 0.95% will be output. This leaves elements with frequencies between 0.95% and 1%. These might or might not form part of the output. Those that make their way to the output are false positives. Further, still as per guarantee 3, all individual frequencies are less than their true frequencies by at most 0.05%.

The approximation in this algorithm has two kinds of errors: 1) false positives still have high frequencies, 2) individual frequencies have small errors. Both kinds of errors are tolerable in the context of frequent item mining.

Algorithm

The algorithm *in se* is the same as the one for concise sampling, with a different method for changing the sampling rate r: it increases logarithmically. Formally: let $t = \frac{1}{\epsilon}\log(s^{-1}\delta^{-1})$. The first $2t$ elements are sampled at $r = 1$, the next $2t$ elements are sampled at rate $r = 2$, the next $4t$ at $r = 4$, and so on.

Whenever the sample rate changes, we also scan $\mathcal{L}$'s entries and update them as follows: for each entry (e, c), we repeatedly toss an unbiased coin until the coin toss is successful, diminishing c by one for every unsuccessful outcome. If c becomes 0 during this process, we delete the entry from $\mathcal{L}$. The number of unsuccessful coin tosses follows a geometric distribution, which can be efficiently computed [54].

Effectively, this will have transformed $\mathcal{L}$ to the state it would have been in if we had been sampling with the new rate from the start.

When a user requests a list of items with threshold s, we output the entries in $\mathcal{L}$ where $c \geq (s - \epsilon)N$. One can prove that the true supports of these frequent items are underestimated by at most ϵ with probability $1 - \delta$.

Space

Its name is derived from the analogy with a magnet: $\mathcal{L}$ sweeps over the data stream like a magnet, attracting all elements which already have an entry in $\mathcal{L}$. Note that the space complexity of sticky sampling is *independent* of N: the space requirements are $2t$ as said before, t is known, thus the space bound is $O(\frac{2}{\epsilon}\log(s^{-1}\delta^{-1})$. Consult [53] for the proof.

4.2.7 Lossy Counting

This is the first algorithm in our list that is deterministic instead of probabilistic. It was presented in the same paper that introduced sticky sampling, by G.S. Manku and R. Motwani, in 2002 [53]. It uses at most $\frac{1}{\epsilon}\log(\epsilon N)$ space, where N denotes the length of the stream so far—contrary to the sticky sampling algorithm described in the previous section, this algorithm is *not independent* of N. This algorithm performs better than sticky sampling *in practice*, although *in theory*, it is worst-case space complexity is worse.

Guarantees

A very clear set of guarantees is given for this algorithm:

1. All items whose true frequency exceeds sN are output. There are *no false negatives.*
2. No items whose true frequency is less than $(s - \epsilon)N$ are output.
3. Estimated frequencies are *less* than the true frequencies *by at most* ϵN.

We say that the algorithm maintains an *ϵ-deficient synopsis* if its output satisfies these guarantees.

Note that guarantee 3, unlike the third guarantee for 4.2.6, does not have a failure probability.

Guarantees Example

The same guarantees example as for sticky sampling applies to lossy counting.

Definitions

The incoming stream is conceptually divided into *buckets* of width $w = \lceil \frac{1}{\epsilon} \rceil$ transactions each. Buckets are labeled with *bucket ids*, starting from 1. The *current bucket id* is denoted by $b_{current}$, whose value is $\lceil \frac{N}{w} \rceil$, with N again the length of the data stream so far. For an element e, we denote its true frequency in the stream so far by f_e.
Note that ϵ and w are fixed while N, $b_{current}$ and f_e are variables whose values change as the stream flows in.

Our data structure $\mathcal{D}$ is a set of entries of the form (e, f, Δ), where e is an element in the stream, f is an integer representing the estimated frequency of e, and Δ is the maximum possible error in f.

In this algorithm, the stream is divided into buckets, but in other algorithms they are typically called *windows*: in the context of this algorithm, they are equivalent concepts.

Algorithm

Initially, $\mathcal{D}$ is empty.

Whenever a new element e arrives, we first scan $\mathcal{D}$ to check if an entry for e already exists or not. If an entry is found, we update it by incrementing its frequency f by one. Otherwise, we create a new entry of the form $(e, 1, b_{current} - 1)$. Why the value for Δ is being set to $b_{current} - 1$ will be explained later on.
So far, the frequency counts hold the actual frequencies rather than approximations. They will become approximations because of the next step.

We also prune $\mathcal{D}$ by deleting some of its entries at bucket boundaries, i.e.: whenever $N \equiv 0 \mod w$. In other words: we prune $\mathcal{D}$ when the next bucket in the stream begins. The rule for deletion is simple: an entry (e, f, Δ) is deleted if $f + \Delta \leq b_{current}$. In other words: elements with a small frequency are deleted; or more accurately: e is deleted if it occurs *at most once per bucket on average.*
Because of this step, the frequency counts now contain approximations of the actual frequencies. Note that these approximations will *always* be *underestimations.*

At any point of time, the algorithm can be asked to produce a list of items, along with their estimated frequencies. When such a request is made by the user, we output those entries in $\mathcal{D}$ where $f \geq (s - \epsilon)N$. This condition guarantees that *all* items whose true frequency exceeds sN are output, but allows for *some* false positives to leak through, although they have a frequency that is almost high enough to qualify as truly frequent.

Insight in How the Algorithm Works

For an entry (e, f, Δ), f represents the exact frequency count of e ever since this entry was *last* inserted into $\mathcal{D}$. The value of Δ assigned to a new entry is the *maximum* number of times e could have occurred in the first $b_{current} - 1$ buckets. This value is exactly $b_{current} - 1$, because otherwise e would *not* have been deleted. Once an entry is inserted into $\mathcal{D}$, its Δ value remains unchanged.

Upon insertion, Δ is being set to $b_{current} - 1$, which is the maximum number of times e *could* have occurred in the first $b_{current} - 1$ buckets, but was deleted at some point in the past because its maximum frequency $(f + \Delta)$ was not sufficiently high $(f + \Delta \not> b_{current})$. Therefore, the average frequency of e over the past buckets must have been less than 1: $\frac{f_e}{b_{current}} \leq 1$.

We can deduct this minimum average occurrence from the fact that the deletion rule is $f + \Delta \leq b_{current}$: this is not satisfied as soon as the f is incremented by at least one for every observed bucket. This effectively means that this algorithm will store all elements which occur more than once per bucket on average.
Since an element is deleted when $f + \Delta \leq b_{current}$, and we know that $b \leq \frac{N}{w} = \frac{N}{\frac{1}{\epsilon}} = \epsilon N$, we can conclude that an item can be underestimated at most by ϵN.

Space

Lossy counting uses at most $\frac{1}{\epsilon}\log(\epsilon N)$ entries, where N is again the current stream length. If elements with very low frequency (at most $\frac{\epsilon N}{2}$) tend to occur more or less uniformly at random, then lossy counting requires no more than $\frac{7}{\epsilon}$ space. Proofs can be found in [53].

4.2.8 Count Sketch

Count Sketch is in fact not the name of this algorithm that was published in 2002 [55], but of the data structure it relies on to estimate the most frequent elements in a data stream in a single pass. A nice side-effect is that this algorithm leads directly to a two-pass algorithm for estimating the elements with the largest (absolute) change in frequency between two data streams.

Intuition

We begin from a very simple algorithm and go to the final algorithm on a step-by-step basis.

Let $\mathcal{S} = q_1, q_2, \ldots, q_n$ be a data stream, with each $q_i \in \mathcal{U} = \{e_1, e_2, \ldots, e_m\}$ (i.e. m different elements in the universe). If each element e_i occurs n_i times in $\mathcal{S}$, then that is so that $n_1 \geq n_2 \geq \ldots \geq n_m$, i.e. n_1 is the most frequent element, n_2 the second most frequent, and so on.

First, let s be a hash function from elements to $\{+1, -1\}$ and let c be a counter. As we process the incoming objects of the stream, each time we encounter an element e_i, we update the (single) counter $c = c + s(q_i)$. We can then estimate the i^{th} most frequent item n_i as follows: $E[c \cdot s[q_i]] = n_i$. However, the variance of every estimate is obviously very large.

A natural solution to this problem is to use more counters. I.e. use t hash functions $s_1, \ldots, s_t$ and maintain t counters $c_1, \ldots, c_t$. Then to process an element q_i, we need to update all counters: $c_j = c_j + s_j(q_i)$, for each j. Now we have $E[c_i \cdot s_i[q_i]] = n_i$. We can then take the mean or median of these estimates to achieve a new estimate with a lower variance than in the previous approach.

However, high frequency elements can spoil the estimates of low frequency elements, because for each element that is encountered, all counters are updated. Therefore we propose an alternative: we replace each of the t counters by a hash table of b counters and have all elements update *different subsets* of counters, one per hash table (i.e. all t "counter hash tables" are updated, but only *one* counter per hash table). This way, every element will get a sufficient amount of high-confidence estimates (since only a few will have large variance thanks to this randomized counter updating process) and therefore all elements can be estimated with sufficient precision. Now we have $E[h_i[q] \cdot s[q]] = n_q$. Note that by increasing the number of counters per hash table b to a sufficiently large amount, the variance can be decreased to an acceptable level and by making the number of hash tables t sufficiently large, we will make sure that each of the m estimates (i.e. one for every element in the universe) has the desired variance.

Algorithm

Let $h_1, \ldots, h_t$ be hash functions from objects to $\{1, \ldots, b\}$ and $s_1, \ldots s_t$ also be hash functions from objects to $\{+1, -1\}$. The CountSketch data structure consists of these hash functions h_i and s_i, along with a $t \times b$ array of counters, which should be interpreted as an array of t hash tables that each contain b buckets. Both t and b are parameters to the algorithm and their values will be determined later.

Note that the idea of hashing elements onto -1 and $+1$ for estimation has already been used and explained before, for approximating the F_2 frequency moment—see section 4.1.5.

The data structure supports two operations:

- `add(`C`,`q`):  for `i`=1 to `t` do `h_i`[`q`] += `s_i`[`q`]`
- `estimate(`C`, `q`):  return median`$_i$ $\{h_i[q] \cdot s_i[q]\}$

We use the median instead of the mean because the mean is—as is well-known—very sensitive to outliers, whereas the median is more robust.

Once this data structure is implemented, the algorithm that belongs with it is straightforward and simple to implement. The CountSketch data structure is used to estimate the count each element in the data stream; to keep a heap of the top k elements seen so far. Formally: given a data stream $q_1, \ldots, q_n$, for each $j = 1, \ldots, n$:

- `add(`C`,`q_j`)`
- If q_j is in the heap, increment its count. Else, add q_j to the heap, but only if `estimate(`C`, `q`)` is greater than the smallest estimated count in the heap; this smallest estimated count should then be deleted from the heap, to make room for q_j.

The algorithm requires $O(tb + k)$ space. It is also possible to bound t and b, but that would involve several proofs, thereby leading us too far—consult [55] for that.

4.2.9 Probabilistic Lossy Counting

One of the most efficient and well-known algorithms for finding frequent items is lossy counting (see section 4.2.7). In [56], published in 2008, a probabilistic variant of lossy counting was introduced, with the unsurprising name Probabilistic Lossy Counting (PLC). It uses a tighter error bound on the estimated frequencies and provides probabilistic rather than deterministic guarantees on its accuracy.

The probabilistic-based error bound substantially improves the memory consumption of the algorithm: it makes PLC less conservative in removing state for elements with a low frequency. In data streams with a large amount of low-frequency elements, this drastically reduces the required memory.

On top of this, PLC also reduces the rate of false positives and still achieves a low, although slightly higher estimation error.

When they applied PLC to find the largest traffic flows (which in the network traffic flow context are typically called *heavy hitters*) show that PLC has between 34.4% and 74% lower memory consumption and between 37.9% and 40.5% fewer false positives, while maintaining a sufficiently small (but as already mentioned, slightly higher) estimation error. Note that these tests were conducted with a very large proportion of small traffic flows (98.9%).

In the original PLC paper, network traffic flows are used to compare PLC with LC. The researchers want to identify the largest traffic flows, to be able to identify denial of service (DoS) attacks, to monitor traffic growth trends, to warn heavy network users, and so on.

Observations Leading to PLC

Remember, LC uses a data structure $\mathcal{D}$ which consists of a set of entries. Each entry is of the form (e, f, Δ). Look at 4.2.7 again to refresh your memory if necessary.

The maximum possible error Δ associated with each element is used when determining which elements to remove from $\mathcal{D}$. An entry is deleted if $f+\Delta \leq b_{current}$. Since Δ is initialized to $b_{current} - 1$ (to adjust for *all* possible buckets in which e might have occurred), this maximum possible error Δ may be large so that the entry stays in $\mathcal{D}$ unnecessarily long. That is, when an entry for an element stays in $\mathcal{D}$ for more buckets, then according to Little's law [57], the average size of $\mathcal{D}$ increases. Thus, *the value of the maximum possible error Δ has a direct impact on the memory consumption of the algorithm.* This is the key observation.

The main improvement of PLC over LC is then to make Δ substantially smaller by providing probabilistic guarantees (versus LC's deterministic error bound). The probabilistic value for Δ as generated by PLC guarantees with a desired probability $1 - \delta$ (with $\delta \ll 1$) that the error of the frequency of an element is smaller than the bound.

In figure 4, the difference in maximum error bound between PLC and LC is demonstrated for a data stream with a very large proportion (98.9%) of low-frequency elements. While this may be considered an extreme example, it still shows the potential for improvement that PLC entails: since there is a large number of low-frequency elements, the decrease in Δ that PLC promises can drastically reduce the size of $\mathcal{D}$.

Guarantees

The user can still specify an acceptable error margin $\epsilon \in [0, 1]$, but unlike LC an acceptable probability of failure $\delta \in [0, 1]$ to meet this error margin can be set once again (like sticky sampling, see 4.2.6).

A very clear set of guarantees is given for this algorithm:

1. All items whose true frequency exceeds sN are output. There *may be false negatives*, although [56] found that false negatives are unlikely in practice. The probability of false negatives can be controlled using the δ parameter.
2. No items whose true frequency is less than $(s - \epsilon)N$ are output.

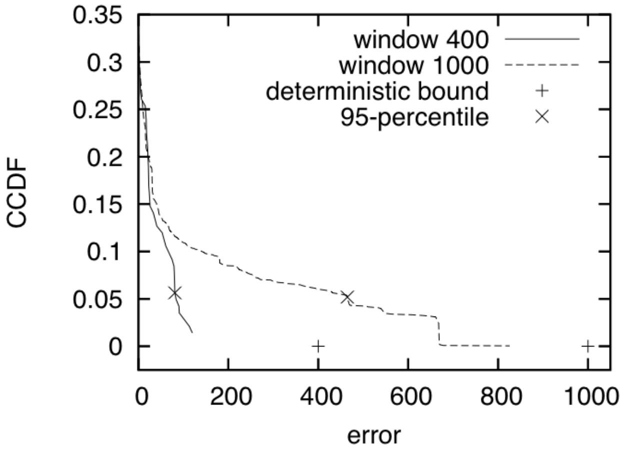


Figure 4: Cumulative error distribution of elements entering $\mathcal{D}$ at buckets (or *windows*) 400 and 1000, 95-percentile of error distribution, and deterministic error bound of LC.
Note that the deterministic bound is significantly larger than the 95 percentile. The data stream is a trace of network traffic flow.
("CCDF" in the chart corresponds to δ and "error" corresponds to Δ.)
(Figures courtesy of [56].)

3. Estimated frequencies are *less* than the true frequencies *by at most* ϵN with probability $1 - \delta$.

Algorithm

The algorithm is identical to the one of LC. The only exception is the value of the maximum possible error Δ. To find this value, [56] assumes that the data stream's element frequencies follow a power-law distribution (they don't give a solution for non-power-law distributions).

In their case of network flow traffic, they have empirically observed that it follows a Zipfian distribution. Providing the entire proof would lead us too far, thus consult [56] for full details.

If Y is a random variable that denotes the true frequency of an element, then $Pr(Y > y) = \alpha y^{\beta}$, where α ($\alpha \leq 1$) and β are the parameters of the power-law distribution. Then we end up at:

$$\Delta = \sqrt[\beta]{\delta(1 - (b_{current} - 1)^{\beta} + (b_{current} - 1)^{\beta}}$$

We still need to calculate β. With probability $1 - \delta$, the set of entries $\mathcal{D}$ contains all the elements with true frequency larger than $b_{current} - 1$. The frequency distribution of these elements is:

$$Pr(Y > y | Y > b_{current} - 1) = \frac{Pr(Y > y)}{Pr(Y > b_{current} - 1)} = \frac{y^{\beta}}{(b_{current} - 1)^{\beta}}$$

Note that this frequency distribution also follows a power-law with the same parameter β as the overall frequency distribution of the data stream. Thus, we can estimate β on-line by fitting a power-law on the frequency distribution of elements in $\mathcal{D}$ with $f > b_{current} - 1$. This of course has the limitation that we are using the estimated frequency f instead of the true frequency. In practice, they found that the estimated frequencies are almost identical to the true frequencies, with a very small error, thereby introducing a negligible error.

Space

The worst-case memory bounds for PLC are the same as those for LC. The average case has the potential to use far less space though, thanks to the more aggresive pruning step.

Evaluation

PLC exploits data streams that tend to have a lot of low-frequency items. For such data streams, PLC is an optimization worth pursuing since the memory consumption savings can be significant.
However, for data streams with relatively equally divided frequencies, there is no memory footprint to gain, but some accuracy is lost and additional computations are necessary.

Clearly, PLC should only be used for data streams with a large proportion of low-frequency items.

4.3 Frequent Pattern *(Itemset)* Mining

Several frequent pattern mining algorithms have been investigated, and they are again presented in order of appearance. Pattern mining works with item*set*s (there are no patterns to be found in single items), which are often called *transactions* in the context of market basket analysis.

Note that the introduction of frequent item mining is still applicable (section 4.2), as are the explanations about window models (section 4.2.1) and the algorithm classification (section 4.2.2).

4.3.1 Lossy Counting for Frequent Itemsets

This algorithm (which is one of the *landmark* model) builds upon the lossy counting (LC) algorithm (see section 4.2.7), to add support for frequent itemset mining. It was introduced by the same paper [53].

However, it clearly is much more difficult to find frequent *itemsets* than *items* since the number of possible itemsets grows exponentially with the number of different items: many more frequent itemsets are possible than the items they consist of.

Changes

The set of entries $\mathcal{D}$ does no longer contain entries of the form (e, f, Δ), but of the form (set, f, Δ), where *set* is a subset of items.

We no longer process the stream transaction per transaction, because then memory consumption would rise significantly. Instead, we try to fill available main memory with as many transactions as possible and then process such a *batch* of transactions together. Let β denote the number of buckets in main memory in the current batch being processed. We then update $\mathcal{D}$ as follows:

- `update_set`: For each entry (set, f, Δ) that exists in $\mathcal{D}$, update f by counting the occurrences of *set* in the current batch.
  The updated entry is deleted if $f + \Delta \leq b_{current}$, just like in LC.
- `new_set`: If a set *set* in the current batch has frequency $f \geq \beta$, and does not yet exist in $\mathcal{D}$, add a new entry $(set, f, b_{current} - \beta)$ to $\mathcal{D}$.
  This too, is analogous to what happens in LC, and is merely adjusted to work with itemsets instead of items.

It is important that β is a large number: this will save memory because all itemsets with a frequency less than β will never enter $\mathcal{D}$ and therefore save memory. For smaller values of β (such as $\beta = 1$ when working with frequent items instead of frequent itemsets), more spurious subsets will enter $\mathcal{D}$, which would drastically increase the average size of $\mathcal{D}$, as well as drastically increase the refresh rate—effectively harming the algorithm in both time and space.

4.3.2 FP-Stream

FP-stream, published in 2003 [58], is designed to mine time-sensitive data streams. It actively maintains frequent *patterns*[4] under a tilted-time window framework (explained a couple of paragraphs further) in order to answer time-sensitive queries. The frequent patterns are compressed and stored using a tree structure similar to FP-tree[5], and updated incrementally as new data flows in.

The task FP-stream wants to solve is to *find the complete set of frequent patterns in a data stream*, with the limitation that one can only see a limited set of transactions (those in the current window) at any moment.

In the FP-growth algorithm [61], the FP-tree provides a structure to facilitate mining in a static data set environment (or a data set that is updated in batches).
In the FP-stream algorithm, two data structures are used:

1. A *FP-tree* in main memory for storing transactions of the *current window.*
2. A *pattern-tree*, which is a tree structure similar to an FP-tree, but with tilted-time windows embedded in it, for storing frequent patterns of the *windows in the past.*

Incremental updates can be performed on both of these parts. Incremental updates occur when some infrequent patterns become subfrequent or frequent, or vice versa. At any point in time, the set of frequent patterns over a period can be obtained from the pattern-tree in main memory.

---

[4] In [58], frequent itemsets are called frequent *patterns*, a name that was kept throughout this section on FP-stream for clarity because some FP-stream-specific structures include "pattern" in their names.

[5] It is assumed the reader is already familiar with the FP-growth algorithm [61]—if not, that should be read first; note that a very clear explanation of FP-growth is available in [25], including excellent figures to explain the data structures it uses.

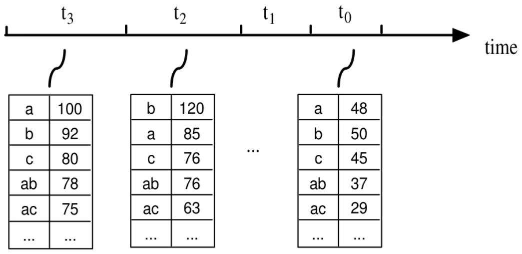


Figure 5: Frequent patterns for tilted-time windows.

(Figure courtesy of [58].)

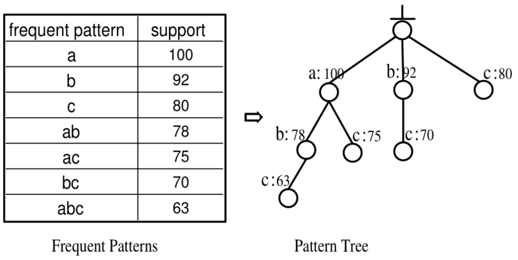


Figure 6: Pattern tree.

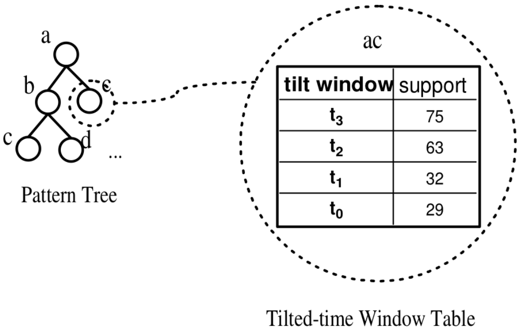
(Figure courtesy of [58].)

Mining Time-Sensitive Frequent Patterns in Data Streams

FP-stream can use any tilted-time window model (for more information about window models and the tilted-time window model in particular, please see 4.2.1). We focus on FP-stream with a natural tilted-time window model (see figure 3 on page 22 again).

For each tilted-time window, a *frequent pattern set* is maintained—see figure 5. This allows us to answer queries like:

- What is the frequent pattern set over the periods t_2 and t_3?
- What are the periods when the pattern (a, b) is frequent?
- Does the support of (a, b, c) change dramatically in the period from t_3 to t_0?
- ...



tilt window	support
t_3	75
t_2	63
t_1	32
t_0	29

Figure 7: Tilted-time windows embedded in pattern-tree.
(Figure courtesy of [58].)

That is, we have sufficient flexibility to mine a variety of types of frequent patterns associated with time—possibilities are:

- Mining frequent patterns in the current window—obviously this is the most basic requirement.
- Mining frequent patterns over time ranges with different granularities.
- Placing different weights on windows to mine weighted frequent patterns.
- Mining evolution of frequent patterns based on the changes of their occurrences in a sequence of windows.

However, we can store this frequent pattern set much more efficiently using a compact tree presentation, called a *pattern-tree*. See figure 6. Note the strong resemblance in structure with an FP-tree. The difference is that in an FP-tree, *all* incoming transactions (itemsets) are stored, whereas in a pattern-tree, only *frequent* patterns (itemsets) are stored. In fact, a pattern-tree (as described thus far) is the same as an FP-tree, but it gets fed different data: frequent transactions only instead of all transactions.

Finally, frequent patterns usually do not change significantly over time. Therefore the pattern-trees for different tilted-time windows will likely have a considerable amount of overlap. If we can *embed the tilted-time window structure into each node of the pattern-tree*, we can save memory. The important assumption here is that frequencies of items do not change drastically and thus the FP-tree structure (its hierarchical structure) does not need to change[6].

---

[6] This requires knowledge about the original FP-growth algorithm [61].

Thus, we *use only a single* pattern-tree where *at each node the frequency for each tilted-time window is maintained.* This final structure is what we call a *FP-stream.* See figure 7 for an example of this.

Maintaining Tilted-Time Windows

As new data flows in, the tilted-time window table grows. In the case of a natural tilted-time window, which is the running example, we need $4 + 24 + 31 + 12 = 71$ windows. For this tilted-time window model, it is very straightforward to perform maintenance: when 4 "quarter windows" have been collected and a fifth has begun, they are merged to form 1 new "hour window". Analogously, when 24 "hour windows" have been collected and a 25th has begun, these 24 windows are merged to form one new "day window", and so on.

Tail Pruning

Given a batch of transactions B, let $f_I(i, j)$ denote the frequency of I in $B(i, j)$.

Let $t_0, \ldots, t_n$ be the tilted-time windows which group the batches seen thus far, with t_n the oldest and t_0 the current. The window size of t_i is denoted w_i (the number of transactions in the window).

The goal of FP-stream is to mine all frequent itemsets whose support is larger than σ over period $T = t_k \cup t_{k+1} \cup \ldots \cup t_{k'}$ (with $0 \leqslant k \leqslant k' \leqslant n$). Then the size of T clearly is $W = w_k + w_{k+1} + \ldots + w_{k'}$. This goal can only be met if we maintain all possible itemsets over all these periods no matter if they are frequent or not[7]. However, this would require too much space.

Fortunately, there is a way to approximate this (and thus require less space). Maintaining only $f_I(t_0), \ldots, f_I(t_{m-1})$ for some m (with $0 \leqslant m \leqslant n$) and dropping the remaining tail sequences of tilted-time windows is sufficient. Specifically, we drop tail sequences $f_I(t_m), \ldots, f_I(t_n)$ when the following conditions hold:

$$\exists l, \forall i, l \leqslant i \leqslant n, f_I(t_i) < \sigma w_i$$

and

$$\forall l', l \leqslant m \leqslant l' \leqslant n : \sum_{i=l}^{l'} f_I(t_i) < \epsilon \sum_{i=l}^{l'} w_i$$

---

[7] Maintaining only frequent tilted-time window entries is not sufficient: as the stream progresses, infrequent itemsets may become frequent.

These conditions imply that all itemsets will be dropped that:

- have a frequency smaller than the minimum frequency per window (σw_i) in *any* window from window l until the n^{th}, i.e. first, i.e. most distant past window ($f_I(t_i) < \sigma w_i$), *and*;
- have a frequency over all windows l through n or l' through n that is lower than the average allowed error rate

As a result, we no longer have an exact frequency over T, but an *approximate frequency* $\hat{f}_I(T) = \sum_{i=k}^{min\{m-1,k'\}} f_I(t_i)$ if $m > k$ and $\hat{f}_I(T) = 0 \sim \epsilon W$ if $m \leqslant k$. The approximation is less than the actual frequency by at most as much as:

$$f_I(T) - \epsilon W \leqslant \hat{f}_I(T) \leqslant f_I(T)$$

Thus, if we deliver all itemsets I for which $\hat{f}_I > (\sigma - \epsilon)W$, we will not miss any frequent itemsets over the period T. As a side-effect, we may incorrectly return some itemsets whose *real* frequencies are between $(\sigma - \epsilon)W$ and σW. This is reasonable when ϵ is small.

We call this *tail pruning.*

Type I & II Pruning

For any itemsets $I \subseteq I'$, the following holds: $f_I \geq f_{I'}$. This is known as the *anti-monotone property*: the frequency of an itemset is always equal or larger than the the frequency of its supersets.
It can be shown that this still holds in the current context of approximate frequency counting and tilted-time windows [58].

From this, it immediately follows that if an itemset I is in the current batch B, but is not in the FP-stream structure, then no superset is in the structure. Therefore, if $f_I(B) < \epsilon\,|B|$, then none of the supersets need to be examined. So the mining of B can *prune its search* and not evaluate supersets of I.

We call this *type I pruning.*

The consequence in the other direction is that if an itemset I is being dropped from the FP-stream structure, then all its supersets can also be dropped.

We call this *type II pruning.*

Algorithm

For an in-depth explanation and evaluation of the algorithm, we refer to [58], sections 3.6, 3.7 and 3.8.

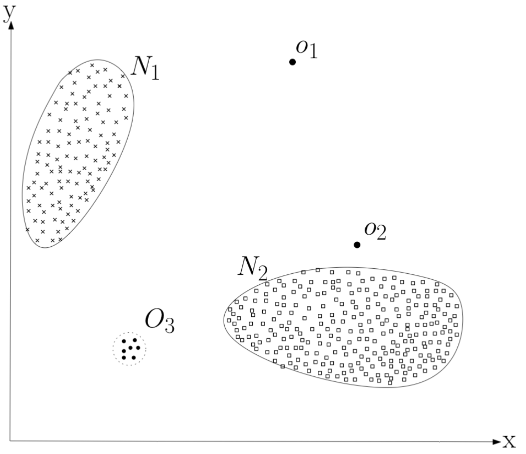


Figure 8: An example of anomalies in a 2D data set.

(Figure courtesy of [62].)

5 Anomaly Detection

This section is based on the comprehensive survey on anomaly detection by Chandola, Banerjee and Kumar [62].

5.1 What are Anomalies?

Anomalies are patterns in data that to not conform to a notion of "normal behavior". This can be easily illustrated through a figure: see figure 8. It illustrates anomalies in a simple 2D data set. The data set has two "normal" regions: N_1 and N_2. They are considered "normal" since most of the observations lie in these two regions. Points that are sufficiently far away from N_1 and N_2 are considered *anomalies*. In this example, that would be points o_1 and o_2, as well as all points in region O_3.

Anomalies can be triggered by a variety of causes, depending on their context; ranging from malicious activities (such as intrusions, credit card fraud, insurance fraud, attack of a computer system) to mere anomalous circumstances (such as an extremely long winter, an extreme amount of rainfall). All these anomalies have in common that they are *interesting* to the analyst—there must be *real life relevance* to make it into an anomaly.

Fields related to anomaly detection are *noise removal*, *noise accommodation* (both of which deal with removing uninteresting data points from a data set that are acting as a hindrance to data analysis) and *novelty detection* (detecting previously unobserved patterns in the data set).

5.2 Challenges

Conceptually, an anomaly is defined as a pattern that does not correspond to normal behavior. So, one would think that while looking at a specific region, one could easily discern the data that is not normal as an anomaly. Unfortunately, several factors make this simple approach impossible:

- When malicious actions cause anomalies, the malicious adversaries often try to adapt themselves to make the anomalous events appear normal, thereby making detecting them much more difficult.
- The definition of “normal behavior” may evolve over time, thus the current definition may no longer be representative in the future (cfr. people’s signatures that change over time).
- In one domain, a small fluctuation may be considered normal, and in another it may be considered an anomaly. Thus techniques of one domain are not necessarily easily applied in another domain.
- Data sets often contain noise that tends to be similar to the actual anomalies, which makes it difficult to detect the actual anomalies.

Due to the above challenges (and this list is not exhaustive), the anomaly detection problem in its most general form is hard to solve: a technique for one domain does not necessarily work for another. That is why existing anomaly detection techniques are often designed especially for one particular domain.

Concepts from other disciplines such as statistics, machine learning, data mining, information theory and spectral theory have been used to develop techniques for specific anomaly detection problems.

5.3 Types of Anomalies

There are three classes anomalies can be classified into:

5.3.1 Point Anomalies

If an *individual data point* can be considered anomalous in comparison with the rest of the data set, then this data point is called a *point anomaly.* This is the simplest type of anomaly, and the majority of the research is focused on this type.

The example (see figure 8 again) used in the introduction contains point anomalies.

For a real life example, let us look at a simple credit card fraud detection technique: if the *amount spent* in a transaction (the sole attribute of each data point) is very high compared to the average amount, that will be considered a point anomaly.

5.3.2 Contextual Anomalies

If a data point is anomalous *in a specific context (but not otherwise)*, then it is called a *contextual anomaly.*

A context is provided by the structure of the data set: each data point is defined using two sets of attributes:

1. *Contextual attributes.* These form the *context* for a data point. e.g. in spatial data sets, the longitude and latitude of a location are contextual attributes. In time-series data, time is a contextual attribute.
2. *Behavioral attributes.* These define the non-contextual properties of a data point. e.g. in a spatial data set that describes the average rainfall of the entire world, the *amount* of rainfall at any location is a behavioral attribute.

The anomalous behavior is then determined using the values for the behavioral attributes *within a specific context.* A data point may be a contextual anomaly in a given context, but another data point with identical behavioral attributes in another context (i.e. with different contextual attributes) may be considered normal.

Contextual anomalies are most commonly investigated in time-series data sets; figure 9 shows an example.
A similar example can be found in the credit card fraud detection domain, that was used for an example of point anomalies previously. Suppose that besides *amount spent* (which is of course a behavioral attribute), there is another, contextual attribute: *time of purchase.* A €50 weekly shopping bill is normal for a given individual, except in December, when he goes buying presents for Christmas and New Year's Eve, then a €200 bill is quite normal. Therefore a €200 bill in February will be considered a contextual anomaly, although a €200 bill in December will *not* be.

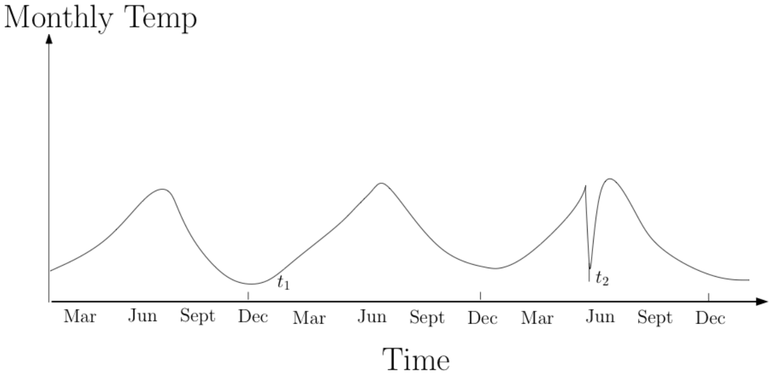


Figure 9: Example of a contextual anomaly. The temperature at time t_1 is the same as that at t_2, but occurs in a different context: the temperature at t_1 is considered normal, the temperature at t_2 is considered an anomaly. *(Figure courtesy of [62].)*

5.3.3 Collective Anomalies

If a *collection* of data points is anomalous when compared with the entire data set, it is called a *collective anomaly.* The individual data points in a collective anomaly may not be anomalies on their own, but their collective occurrence *is* anomalous.

In figure 10, a medical example is shown: it is the output of a human electrocardiogram. The highlighted region is a collective anomaly because the same low value exists for an abnormally long time, although by itself this low value is *not* an anomaly (i.e. one such data point with this low value is not an anomaly).

Note: while point anomalies can occur in *any* data set, collective anomalies can only occur in data sets whose data points are *related.* By including possible contextual information (i.e. if it is available), both a point anomaly detection problem and a collective anomaly detection problem can be transformed into a contextual anomaly detection problem.

5.4 Anomaly Detection Modes

Labeling data points in an accurate manner, while ensuring that all types of behaviors are represented, may be prohibitively expensive. Labeling is often performed *manually* by a human expert—which clearly requires substantial effort. Typically, getting a labeled set of anomalous data that covers *all*

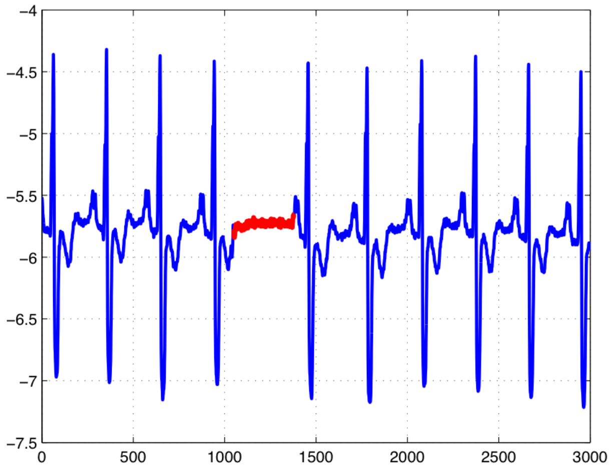


Figure 10: Example of a collective anomaly in a human electrocardiogram. *(Figure courtesy of [62].)*

possible types of anomalous behavior is more difficult than getting labels for normal behavior. Additionally, *new* anomalies may arise over time, for which there is no labeled training data.

Anomaly detection techniques can operate according to three possible modes. Which mode can be used depends on the availability of labels:

- *Supervised Anomaly Detection.* For supervised mode techniques, the availability of a a training data set with labels for normal *and* anomaly classes is a requirement.
- *Semi-Supervised Anomaly Detection.* Techniques that operate in this mode, training data has labeled data points for *only the normal class.* Because they do not need require labels for the anomaly class, they are more widely applicable than supervised techniques.
- *Unsupervised Anomaly Detection.* These techniques don't require *any* training data and therefore are most widely applicable. They do make the assumption, however, that normal instances are far more frequent than anomalies. If this assumption is false, then a high false alarm rate is the consequence.

5.5 Anomaly Detection Output

An obvious, yet important aspect of anomaly detection is the output of the technique used, which can be of either of the following two types:

- *Scores.* Scoring techniques assign an *anomaly score* to each data point in the data set, depending on the degree of anomalousness of that data point.
- *Labels.* Labeling techniques assign a label—either "normal" or "anomalous"—to each data point.

Note: scoring based anomaly detection techniques allow for a selection within all anomalies, e.g. to select the worst anomalies only.

5.6 Contextual Anomaly In Detail

There are many possible types of contextual attributes, some of which are:

1. *Spatial.* e.g. latitude and longitude
2. *Graphs.* The edges that connect nodes (with each node being a data point) define the neighborhood for each node (data point).
3. *Sequential.* The data set contains sequential data points, i.e. the contextual attributes of a data point define its position in the sequence. Note that there is an important difference between *time-series data* and *event sequence data*: time-series data haven *even inter-arrival times*, whereas event sequence data have *uneven inter-arrival times.*

While a lot of literature is available for point anomaly detection techniques, the research on contextual anomaly detection has been limited. Contextual anomaly detection techniques can be divided in two categories:

1. *Reduction to a point anomaly detection problem.* Contextual anomalies are individual data points (like point anomalies), but are anomalous only with respect to a *certain context.*
   An obvious generic reduction technique is then to first identify a context under which to operate and then perform a point anomaly detection technique.
2. *Model the structure of the data and then use this model to detect anomalies.* A generic technique in this category is the following. A model is learned from training data that is able to predict the expected behavior within a given context. If the *observed* behavior is significantly different from the *expected* behavior, the corresponding data point is declared

anomalous.
A simple example of this generic technique is regression in which the contextual attributes can be used to predict the behavioral attribute by fitting a regression line (sometimes also called a trend line) on the data.

Computational Complexity

The computational complexity of the training phase for techniques that use models of the data is typically higher than that of techniques that reduce the problem to point anomaly detection. However, structure model techniques have a relatively fast testing phase, thanks to the fact that each data point only needs to be compared to a single model.

Advantages and Disadvantages of Contextual Anomaly Detection Techniques

A *natural definition* of an anomaly is the main advantage of contextual anomaly detection techniques: in real life applications, data points tend to be similar within a given context. Also, these techniques are able to detect anomalies that may not be detected when using techniques that take a global view of the data set (which is exactly what point anomaly detection techniques do).

The main disadvantage is a very obvious one: contextual anomaly detection techniques are only applicable when a context is present in the data set.

5.7 Contextual Anomaly Algorithms

In the context of this thesis, we are clearly dealing with sequential data with contextual anomalies (with episode duration being the behavioral attribute and all other attributes contextual). However, we cannot assume even inter-arrival times, hence we need to look at techniques for *event sequence* data only.

After searching for papers on contextual anomaly detection algorithms that work on event sequences, two interesting papers stood out: the algorithm by Vilalta/Ma and the Timeweaver algorithm.

There is a strong reason for not examining point anomaly algorithms in more detail: to be able to reduce a contextual anomaly algorithm to a point

anomaly algorithm, it is necessary to consider *each combination of contextual attributes* and then look at the behavior attributes for that contextual attribute.
In the context of this thesis, the number of contextual attributes can grow very large, which then makes reduction to point anomaly detection rather inefficient.

5.7.1 Vilalta/Ma

Published in 2002, Vilalta & Ma [64] designed a system based on frequent itemset mining to find patterns in historical data. More specifically, their approach extracts temporal patterns from data to predict the occurrence of rare target events. They make two assumptions:

1. that the events are being characterized by categorical attributes and are occurring with uneven inter-arrival times, which makes this an algorithm to work on event sequence data and not time-series data;
2. that the target events are highly infrequent.

They have developed an efficient algorithm for this particular problem set that involves performing a search for all frequent eventsets (which are just a special type of itemsets: instead of "items" they contain "events types") that *precede* the target events. The patterns that are found are combined into a rule-based model for prediction.

Their approach differs from previous work that also uses the learning strategy: most learning algorithms assume even class distributions and adopt a *discriminant-description* strategy: they search for separators *(discriminants)* that best separate (discriminate) examples of different classes. Under skewed distributions (which is the case here: the target events are highly infrequent), separating the under-represented class is difficult. That is why they have opted for a *characteristic-description* strategy: instead of searching for separators, they search for common properties, and they do so by looking at the events preceding a target event, to find common precursor events.

The Event Prediction Problem, Formally

The fundamental unit of study is an *event*. An event is of the form $d_i = (e_i, t_i)$ where e_i indicates the event type and t_i indicates the occurrence time.

Events belong in a sequence $D =< d_1, d_2, \ldots, d_n >$.

We are interested in predicting certain kinds of events that occur in sequence D. We refer to this subset of events as *target events*: $D_{target} \subset D$. We assume that the relative frequency of target events in comparison with all events is low. Furthermore, target events do not represent a *global* property of D (such as a trend or periodicity), but rather a *local* property.

The user must specify a target event type e_{target} (e.g. all fatal events), that defines D_{target} as

$$D_{target} = \{d_i \in D \mid e_i = e_{target}\}$$

The framework assumes a dataset D of size n , containing a sequence of events (as defined before). Event types take on categorical values. We also assume we have identified a set of events $D_{target} \subset D$ with $|D_{target}| = m \ll n = |D|$.

The approach the Vilalta/Ma algorithm takes is to capture patterns that characterize the conditions that precede each target event (i.e. where $e_i = e_{target}$). Specifically, the goal is to find out what types of events frequently precede a target event, for the purpose of prediction. We look at those preceding events within a time window of fixed size W before a target event (as illustrated in figure 11).

Next, there is a whole series of definitions for an "eventset", that will be used in the remainder of this section:

- *Matching.* An eventset Z is a set of event types $\{e_i\}$. Eventset Z *matches* the set of events in window W if every event type $e_i \in Z$ is found in W.
- *Support.* An eventset Z has *support* s in D if s% of all windows of size W preceding target events are matched by Z. Eventset Z is *frequent* if s is above a minimum user-defined threshold.
- *Confidence.* An eventset Z has *confidence* c in D if c% of all windows of size W matched by Z precede a target event. Eventset Z is *accurate* if c is above a minimum user-defined threshold.
- *Specificity.* An eventset Z_i is said to be more specific than an eventset Z_j if $Z_j \subset Z_i$.
- *Order.* We impose a partial ordering over the space of eventsets. An eventset Z_i is marked as having a higher rank than eventset Z_j, denoted $Z_i \succ Z_j$ if any of the following conditions is true:

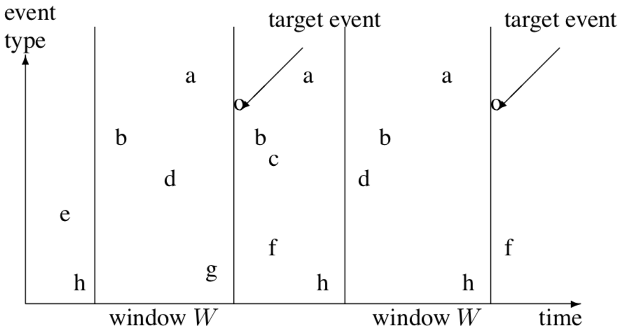


Figure 11: A plot of different event types versus time. Before each target event, there is a time window. This allows us to identify frequent sets of event types that act as indicators/precursors.

(Figure courtesy of [64].)

1. The confidence of Z_i is greater than that of Z_j.
2. The confidence of Z_i equals that of Z_j, but the support of Z_i is greater than the support of Z_j.
3. The confidence and support of Z_i equal that of Z_j, but Z_i is more specific than Z_j.

Prediction Strategy

Their prediction strategy takes the following steps:

1. *Characterize* target events by looking at a fixed time window that precedes the target event and then finding the types of events that frequently occur within that window. See figure 11 for an easy to understand graphical explanation.

2. *Validate* that the event types found in step 1 *uniquely* characterize target events, and that they do not often occur outside of the window directly preceding the target event.

3. *Combine* the validated event types found in step 2 into rules, to end up with a set of rules from which predictions can be made (i.e. a rule-based prediction system).

Algorithmically, these steps take the following shape:

1. *Frequent eventsets.* This employs the standard Apriori [60] frequent *itemset* mining algorithm over each window (but of course this could be replaced with any frequent itemset mining algorithm, such as FP-growth [61]) to find all frequent *eventsets.* E.g. in the case of figure 11, the eventset $\{a, b, d\}$ would be found as the only frequent eventset with a sufficiently high minimum support. Let's call the collection of frequent eventsets B, then $B = \{\{a, b, d\}\}$.
   Note that because thanks to the use of event*sets*, the order of events does no longer matter, nor do the inter-arrival times.

2. *Accurate eventsets.* With frequent eventsets calculated, the next step is filtering out those eventsets that do not meet minimum confidence. Here, the general idea is to look at the number of times each of the frequent eventsets occurs outside the time windows preceding the target events. We capture all event types within each window that does *not* overlap with the time windows that precede target events. We store these eventsets in a new database of eventsets B'. This database contains all eventsets that do *not* precede target events.
   Now we can calculate the confidence for the frequent eventsets in B. Let $f_Z(B)$ be the number of transactions in B that matches the eventset Z and $f_Z(B')$ that for B'. Then the confidence of the eventset Z is defined as follows: $confidence(Z, B, B') = f_z(B)/f_z(B) + f_z(B')$. Now we can filter the frequent eventsets to only keep those with high confidence, i.e. *accurate* eventsets. We store the result in $\mathcal{V}$.

3. *Building a rule-based model.* For this, we first need to order the eventsets in $\mathcal{V}$ depending on their rank. This allows us to find the most accurate and specific rules first. Then, we iterate over $\mathcal{V}$ as long as it is not empty. In each iteration, we select the next best eventset Z_i and removes all other eventsets Z_j in $\mathcal{V}$ that are more general than Z_i. This effectively eliminates eventsets that refer to the same pattern as Z_i but that are unnecessarily general. A rule for Z_i is generated, of the form $Z_i \rightarrow targetevent$ and is added to $\mathcal{R}$. Then the next iteration begins.

The resulting set of rules $\mathcal{R}$ can be used for prediction.

5.7.2 Timeweaver

Timeweaver is a genetic algorithm, published in 1998 [63], that is able to learn to predict rare events from sequences of events with categorical attributes. It achieves this by identifying predictive temporal and sequential patterns.

Because this algorithm is based on genetic algorithms, and explaining that too in full detail would lead us too far, this algorithm is only explained from a high level perspective. The explanation should be sufficient to grok the algorithm and put it into perspective next to the Vilalta/Ma algorithm (see section 5.7.1).

Prediction Pattern

A *prediction pattern* is a sequence of events connected by *ordering primitives* that define sequential or temporal constraints between consecutive events. The three ordering primitives are defined below, with A, B, C and D representing individual events:

- *Wildcard "*".* Matches *any number* of events, e.g. the prediction pattern A*D matches ABCD
- *Next ".".* Matches *no* events, e.g. the prediction pattern D.A.C. only matches DAC.
- *Unordered "|".* Allows events to occur in *any order* and is *commutative*, e.g. the prediction pattern A|C|D will match ACD, ADC, CDA, and so on.

The "|" primitive has the highest precedence. Each categorical attribute is allowed to take on the "?" value, which matches *any* value. A prediction pattern also has a *pattern duration*, of course represented by an integer.

Then a prediction pattern *matches* a sequence of events within an event sequence if:

1. events within the event sequence are matched by the prediction pattern, and;
2. ordering constraints in the prediction pattern are obeyed, and;
3. the events in the match occur within the pattern duration.

This prediction pattern language allows for flexible and noise-tolerant prediction rules. For example: "if 3 (or more) A events and 4 (or more) B events occur within an hour, then predict the target event".
This language was designed to be simple yet useful. Extensions are possible and would only require changes to timeweaver's pattern-matching logic.

Algorithm

First, the population is initialized by creating prediction patterns containing a single event, with the categorical attribute values set to the wildcard value "?" 50% of the time and to a randomly selected categorical attribute value the remaining 50% of the time.

The genetic algorithm then repeatedly does the following until a stopping criterion is met: it selects 2 individuals from the population and applies the mutation operator on both individuals (which randomly modifies a prediction pattern: changing the categorical attribute values, ordering primitives or pattern duration) *or* crossover (which may result in offspring of different length from the parents, and thus may result in any size of pattern over time).

Now, of course it is impossible to keep adding new prediction patterns: after a certain amount of prediction patterns is being maintained, it becomes necessary to replace existing ones with new ones (i.e. offspring from crossover). We cannot use simple strategies such as FIFO here; it is necessary to balance two opposing criteria: maintaining a diverse population (to keep all options open) and focusing search in the most profitable areas. This can be achieved by evaluating prediction patterns on exactly those properties: weighing each pattern's fitness versus its uniqueness when compared to the other patterns.

For more details, please consult [63].

6 OLAP: Data Cube

OLAP—short for On-Line Analytical Processing—is an approach designed to be able to quickly answer queries about multidimensional data.

Some of the terminology and capabilities of OLAP systems can be found in today's spreadsheet applications, so it is in fact very likely that you're already (unwittingly) familiar with OLAP principles! OLAP systems are designed to make interactive analysis of (multidimensional) data possible and typically provide extensive visualization and summarization capabilities.

6.1 Multidimensional Data Representation

6.1.1 Fact Table

The starting point typically is a *fact table*: a tabular representation of the data set.

The Iris data set

In table 1, a *fact table* of the multidimensional Iris data set[a] can be found. It has been simplified[b] to serve as a simple, easy-to-grasp example that will be used throughout the OLAP section to demonstrate data transformations and manipulations.
For each of the 3 types of Irises that have been reviewed (Setosa, Versicolour and Virginica), the petal length and petal width have been analyzed. The lengths and widths that were found have then been marked[c] as "low", "medium" or "high". 50 flowers of each species were analyzed.
The table is split in three parts, one for each species (thus each of these parts' counts sums up to a total of 50).
In the remainder of this section, you will often see boxes like this one (with a double frame). Each of those apply the explanations in the preceding piece of text to the Iris data set. This should help the reader gain a deeper understanding much faster.

---

[a] A famous data set from 1936 by the statistician R.A. Fisher; can be obtained from the UCI Machine Learning Repository [26].

[b] Two attributes have been omitted: sepal length and sepal width.

[c] More accurately, the continuous attributes petal length and petal width have been *discretized*. They were numbers in the range $[0, \infty[$ (in centimers) that have been discretized to the intervals $[0, 0.75] \rightarrow$ "*low*", $[0.75, 1.75] \rightarrow$ "*medium*" and $[1.75, \infty[\rightarrow$ "*high*".

petal length	*petal width*	*species type*	*count*
low	low	Setosa	46
low	medium	Setosa	2
low	high	Setosa	0
medium	low	Setosa	2
medium	medium	Setosa	0
medium	high	Setosa	0
high	low	Setosa	0
high	medium	Setosa	0
high	high	Setosa	0
low	low	Versicolour	0
low	medium	Versicolour	0
low	high	Versicolour	0
medium	low	Versicolour	0
medium	medium	Versicolour	43
medium	high	Versicolour	3
high	low	Versicolour	0
high	medium	Versicolour	2
high	high	Versicolour	2
low	low	Virginica	0
low	medium	Virginica	0
low	high	Virginica	0
medium	low	Virginica	0
medium	medium	Virginica	0
medium	high	Virginica	3
high	low	Virginica	0
high	medium	Virginica	3
high	high	Virginica	44

Table 1: The Iris data set: a table representation. Contains data on a number of flowers having a particular combination of petal width, petal length and species type.

6.1.2 Multidimensional Array

A key motivation for using a multidimensional viewpoint of data is the importance of aggregating data from various perspectives. In sales, you might want to find totals for a specific product per year and per location for example. Or per day. Or for all products per location. Anything is possible.

To represent this input data as a multidimensional array, two steps are necessary:

1. identification of the dimensions (or *functional attributes*); these must be *categorical attributes*[8]
2. identification of the attribute that is the focus of the analysis (the *measure attribute*)—this attribute is called the *target quantity*; this must be a *quantitative attribute*

Note that it is possible to have multiple target quantities (i.e. analyze multiple quantitative attributes simultaneously). However, to keep the reasoning straightforward, we will impose a limit of a single target quantity.
One could simply analyze each target quantity separately, or apply an arbitrary formula to combine multiple quantitative attributes into a single target quantity.

The dimensions are categorical attributes. The values of an attribute serve as the indices into the array for the dimension corresponding to that attribute; the size of this dimension is equal to the number of different values for this attribute.

---

[8] Obviously, any attribute can be transformed into a categorical attribute by means of discretization. This is also what has been done for the example: the petal length and petal width examples have been discretized.

Dimensions of a multidimensional array representation

In the case of the Iris data set (see table 1), there are a single quantitative attribute (count) and 3 categorical attributes:

1. petal length
2. petal width
3. species type

Petal length and petal width range[a] over the same 3 values: “low”, “medium” and “high”. Hence 3 is the size of both the petal length dimension and the petal width dimension.

There are 3 different species and thus the species type dimension is also of size 3. Hence there are 3 × 3 × 3 indices, with 27 corresponding values.

---

[a] As already mentioned before, petal length and petal width originally also were quantitative attributes.

Each combination of attribute values (one for each attribute) defines a cell in the multidimensional array; each cell contains the value of the target quantity. The target attribute is a *quantitative* attribute because typically the goal is to look at aggregate *quantities* (total, average, minimum, maximum, standard deviation . . . ; the list can go on endlessly when adding domain-specific functions for physics, financial analysis, etc.).

Multidimensional array representation

There are three categorical attributes: petal length, petal width and species type. There is one quantitative attribute: the corresponding count. Since there are three categorical attributes, this can be represented in a three-dimensional array. See figure 12.
Note that this is *not* a data cube: it is merely a multidimensional representation. It has 3 dimensions and therefore it looks like and *is* a cube, but *not* a *data cube*. As long as not *all* aggregates are there, it is not a data cube! (Note that there is for example no aggregate count for all flowers by species type, amongst others.) At least in OLAP context.
It may be called a *data cube representation* though: it is just a way to represent a data set—no calculations are required. For the result of the data cube *operator*, calculations *are* required.

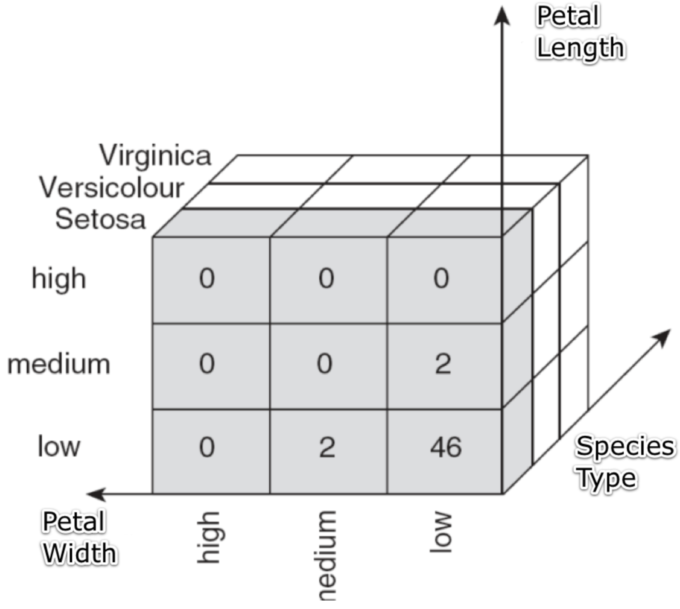


Figure 12: A multidimensional *representation* of the Iris data set—but *not* a data cube!

(Figure courtesy of [25].)

6.2 Slicing and Dicing

Slicing and *dicing* are both very straightforward. Slicing requires a specific value to be specified for one or more dimensions. Dicing does not require a single specific value to be chosen, but allows a *range* of attribute values to be specified.

Slicing

In the context of the Iris data set example: the "front" of the multidimensional representation (figure 12) is one of the three displayed slices (table 2), the other two possible slices (tables 3 and 4) are the "deeper" slices, when looking at the multidimensional representation from the same perspective.

Setosa

		petal width		
		high	medium	low
petal length	high	0	0	0
	medium	0	0	2
	low	0	2	46

Table 2: Slice where the species "Setosa" has been selected.

Versicolour

		petal width		
		high	medium	low
petal length	high	2	2	0
	medium	3	43	0
	low	0	0	0

Table 3: Slice where the species "Versicolour" has been selected.

Virginica

		petal width		
		high	medium	low
petal length	high	44	3	0
	medium	3	0	0
	low	0	0	0

Table 4: Slice where the species "Virginica" has been selected.

Dicing

A possible dice for the Iris data set can be seen in table 5: it is a *subset* of the “front” of the multidimensional representation (figure 12).

Setosa		petal width		
		high	medium	low
petal length	low	0	2	46

Table 5: Slice where the species “Setosa” and petal length “low” have been selected.

6.3 Data Cube

Before going into details about the data cube, let's start with an example—it will immediately be clear how a data cube can be used.

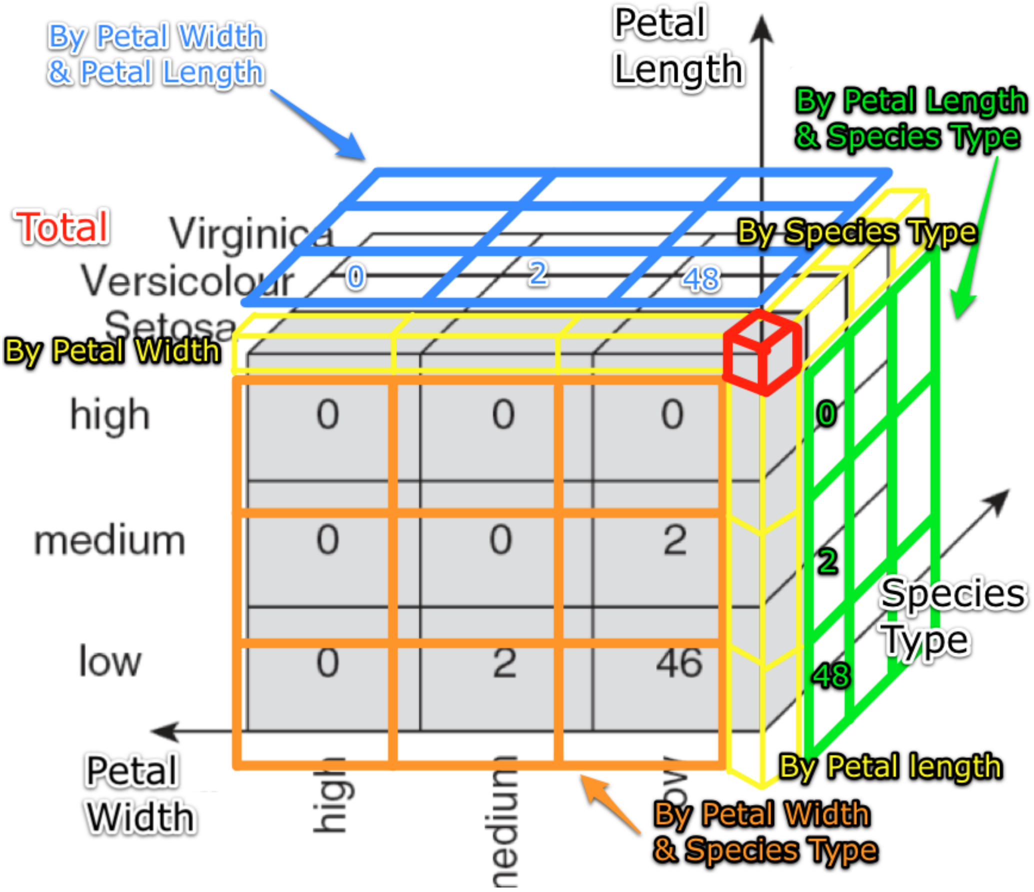


Figure 13: Annotated data cube.

6.3.1 Definition

As input, the data cube operator accepts a fact table T. T has n aggregation attributes $\mathtt{A}_1, \ldots, \mathtt{A}_n$ and 1 measure attribute `M`.

$$T(\mathtt{A}_1, \ldots, \mathtt{A}_n, \mathtt{M})$$

The aggregation function is applied to the measure attribute `M`, e.g. `SUM()`.

The SQL syntax for the data cube operator is:

```
SELECT A₁, ..., Aₙ, CSUM
FROM T
GROUP BY A₁, ..., Aₙ, SUM(*) AS CSUM
WITH CUBE
```

Now, let us consider the semantics behind the above. Consider a subset of the aggregation attributes $\mathtt{S} \subseteq \{\mathtt{A_1}, \ldots, \mathtt{A_n}\}$. Define the query $Q_{\mathtt{S}}$ as :

```
SELECT Â₁, ..., Âₙ, SUM(M)
FROM T
GROUP BY S
```

with

$$\hat{\mathtt{A}}_{\mathtt{i}} = \begin{cases} \mathtt{A_i} & \textit{if}\ \mathtt{A_i} \in \mathtt{S} \\ \mathtt{ALL} & \textit{otherwise} \end{cases}$$

(In the above, each `ALL` value is in fact an alias for a set: the set of *all* values of the attribute over which an aggregate is computed.)

Each $Q_{\mathtt{S}}$ defines aggregation over a specific combination of attributes. Then the entire cube is the union of all these $Q_{\mathtt{S}}$ (i.e. with all possible subsets `S`), of which there are 2^n (i.e. there are 2^n subsets `S` for n aggregation attributes).

If the cardinality of the n attributes are $C_1, C_2, \ldots, C_n$ (i.e. $\textit{cardinality}(\mathtt{A_i}) = C_i$), then the cardinality of the resulting cube relation is $\prod(C_i+1)$. The extra value in each attribute domain is the `ALL` value, which represents the set of values over which the aggregate is computed.

6.4 Generalized constructs

The data cube (or just *cube*) operator generalizes the following constructs:

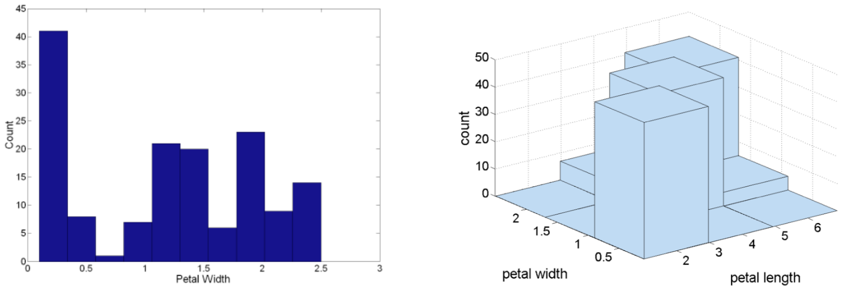
- histogram
- cross tabulation
- roll-up
- drill-down

6.4.1 Histogram

A histogram is a bar chart representing a frequency distribution; heights of the bars represent observed frequencies.

Histogram

In figure 14, two sample histograms can be seen for the Iris data set. The first is a 1D histogram (based on the petal width), the second is a 2D histogram (based on petal width and petal length). Petal length and width have not been discretized here (to “low”, “medium” and “high”) as they were previously. Instead, they were discretized into numerical ranges.



(a) 1D histogram (b) 2D histogram

Figure 14: Sample histograms for the Iris data set.

(Figures courtesy of [25].)

6.4.2 Cross tabulation

A cross tabulation *(“cross tab”)* displays the joint distribution of two or more variables, along with the marginal totals. In the case of two variables, these are the row and sum totals.
Note: a cross-tabulation over exactly two dimensions is also called a *pivot*.

Cross tabulation

Cross tabulation are slices, with added marginal totals. Table 6 is the cross-tabulation for the slice in table 2, as is table 7 the cross tabulation for table 3 and table 8 the cross tabulation for the slice in table 4.

Setosa		petal width			
		high	medium	low	*total*
petal length	high	0	0	0	0
	medium	0	0	2	2
	low	0	2	46	48
	total	0	2	48	50

Table 6: Cross tabulation of the slice where the species "Setosa" has been selected.

Versicolour		petal width			
		high	medium	low	*total*
petal length	high	2	2	0	4
	medium	3	43	0	46
	low	0	0	0	0
	total	5	45	0	50

Table 7: Cross tabulation of the slice where the species "Versicolour" has been selected.

Virginica		petal width			
		high	medium	low	*total*
petal length	high	44	3	0	47
	medium	3	0	0	3
	low	0	0	0	0
	total	47	3	0	50

Table 8: Cross tabulation of the slice where the species "Virginica" has been selected.

6.4.3 Roll-up

A roll-up is the aggregation of values *within* a dimension—not across an entire dimension!
Note: this requires that the attribute that is being rolled up can be considered hierarchical in some sense, i.e., that it can be viewed with different levels of granularity.

Roll-up

Since the Iris data set does not contain any hierarchical data, we cannot apply roll-up to it. So, another example is presented.
For example, given sales data with entries for each date, we can *roll up* (aggregate) the data across all dates in a month, resulting in monthly sales totals. This is aggregation within a dimension; aggregation across a dimension would have given us the total of all sales ever recorded.

6.4.4 Drill-down

A drill-down can be considered the inverse of a roll-up: instead of viewing the data "at a higher level", the data will be viewed with more granularity—"at a lower level".
Note: this requires that the attribute that is being rolled up can be considered hierarchical in some sense, i.e., that it can be viewed with different levels of granularity.

Drill-down

Since the Iris data set does not contain any hierarchical data, we cannot apply drill-down to it. So, another example is presented.
Continuing on the example for roll-up, a drill-down would for example split monthly sales totals into daily sales totals. For such drill-downs to be possible, it is of course a necessity that the underlying data is sufficiently granular.

6.4.5 Generalization explained

The generalization of the aforementioned constructs may appear obvious. It is simply another 'level' of aggregation. Schematically, it could be described as follows:

aggregate ($\equiv$ 0D data cube)
$\subset$
`GROUP BY` ($\equiv$ 1D data cube)
$\subset$
cross tab ($\equiv$ 2D data cube)
$\subset$
3D data cube
$\vdots$
nD data cube

To ensure that you understand this, the following illustration makes it very clear in a graphical manner:

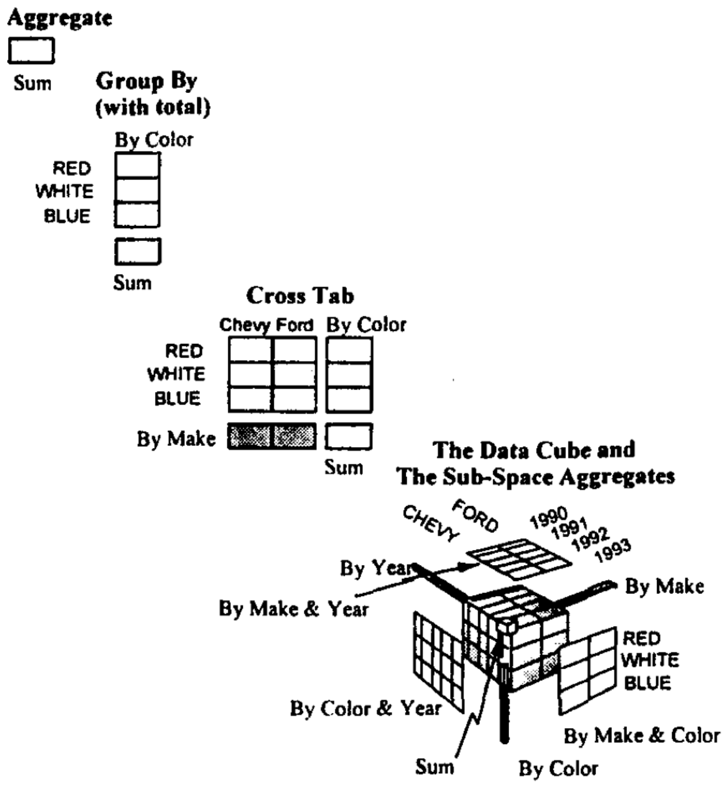


Figure 15: The date cube is the n-dimensional generalization of more simple aggregation functions.

(Figure courtesy of [65].)

6.5 The Data Cube Operator

Typically, data is stored in an RDBMS. To calculate the above constructs, the `GROUP BY` operator is necessary. This operator partitions the relation into disjoint tuple sets (based on one or more attributes that are common amongst the tuples in each tuple set) and then aggregates over each set. In other words, using the `GROUP BY` construct allows a table to be created of many aggregate values, *indexed by a set of attributes.*

However, there are some problems with using the `GROUP BY` operator [65].

Histogram

The standard SQL `GROUP BY` operator does not allow for *easy* construction of histograms (aggregation over computed categories), because it does not allow functions to be used in the `GROUP BY` clause[9].

But that doesn't mean it can't be expressed at all: SQL is Turing complete and therefore it can be expressed. It just can't be expressed very *elegantly* or succinctly. A SQL statement of the type `GROUP BY F()` is disallowed, but one can still achieve a group by on a function by applying the function in a subquery and performing the group by over the result.

For example, it is desirable to be able to write:

```
SELECT      avgPetalLength, SpeciesType
FROM        Iris
GROUP BY    AVG(PetalLength) AS avgPetalLength,
            SpeciesType
```

But instead, we're forced to use a subquery, which is less concise:

```
SELECT      avgPetalLength, SpeciesType
FROM        (SELECT AVG(PetalLength)
                            AS avgPetalLength,
                        Speciestype
            FROM Iris) AS sub
GROUP BY    avgPetalLength,
            SpeciesType
```

---

[9] Not in SQL-92, which was available at the time of writing [65] (SQL3 was in development at the time of writing [65] and was to later become the SQL:1999 standard) and still not in SQL:2008 [66], which is the latest SQL standard at the time of writing this text.

Roll-up & drill-down

To calculate a roll-up over n dimensions requires n unions: n group by SQL statements need to be unioned together—1 per dimension that is being rolled up.

The drill-down case is analogous to that for roll-up.

Since the Iris data set does not contain any hierarchical data, we cannot apply drill-down to it. So, another example is presented.

Suppose car sales data is being collected in a `Car(Model, Year, Color, Sales)` table. Then it is likely that one would like to create a roll up of `Sales` by `Model` by `Year` by `Color`, i.e. ascending up the "`Model-Year-Color`" hierarchy, to decrease granularity at each step.
This would require the union of a group by on `Model`, then a group by on `Model`, `Year` and finally a group by on `Model`, `Year`, `Color`. We now have a roll-up over 3 dimensions, which required the union of 3 group by statements.

The end result looks like this:

Model	*Year*	*Color*	*Sales*
Chevy	1994	white	40
Chevy	1994	black	50
Chevy	1995	white	115
Chevy	1995	black	85
Chevy	1994	`ALL`	90
Chevy	1995	`ALL`	200
Chevy	`ALL`	`ALL`	290

Table 9: Sample roll-up result. Granularity decreases from top to bottom, as we roll up by more attributes in the hierarchy.

Cross tabulation

Roll-ups are asymmetric, cross tabulations are symmetric and require even more unions: 2^n unions!

This example continues on the roll-up example above.

Suppose we wanted to show the cross tabulation of the sales for a specific model, with the range of values for `Year` as columns and the range of values for `Color` as the rows. Then we can reuse the results for the roll-up example. But for roll-up, we didn't aggregate sales by color—this is why roll-up is called assymetric.

So we lack the rows that aggregate sales by `Color`. These rows are generated by one additional unioned in group by statement, and contain:

Model	*Year*	*Color*	*Sales*
Chevy	`ALL`	white	155
Chevy	`ALL`	black	135

Table 10: Rows generated by aggregating by `Color`.

Combined, we now have a symmetric aggregation result, which required $4 = 2^2$ unioned group by statements (3 from the roll-up example plus 1 additional group by statement from this example), while we're building a 2D cross-tabulation (on `Year` and `Color`). Molding the data into a cleaner representation like previous cross tabulations gives us:

Chevy		Year		
		1994	1995	*total* (`ALL`)
Color	black	50	85	135
	white	10	75	85
	total (`ALL`)	60	160	220

Table 11: Cross tabulation of `Model` by `Year` and `Color`.

Solution to daunting SQL: the data cube operator

As should be clear by now, the data cube operator was not absolutely necessary in the strictest sense: anything that can be achieved with the data cube operator can be achieved without it. But thanks to the data cube operator, it

is no longer necessary to apply the same patterns repeatingly: the necessary SQL becomes much less daunting (e.g. a 6D cross tabulation would require a $2^6 = 64$ way union).
And because the exact logic behind it is now abstracted away in the SQL language syntax, it paved the way for possible future optimizations.

To support the data cube operator, SQL's `SELECT-GROUP-BY-HAVING` syntax was extended to support histograms, decorations[10] and the `CUBE` operator (as well as the `ROLLUP` operator[11]).
Microsoft pioneered this in their SQL Server RDBMS product [65].

6.6 Elaborate data cube example

Continuing with the Iris data set (see table 1), a sample query that utilizes the newly introduced data cube operator is listed below:

```
SELECT PetalLength, PetalWidth, SpeciesType, COUNT(*) AS CCount
FROM Iris
GROUP BY PetalLength,
         PetalWidth,
         SpeciesType
    WITH CUBE;
```

For the semantics behind this query, see the definition in section 6.3.1.

It might be helpful to give you a deeper understanding of `ALL` values (again, see the definition), in the context of this example.
Each `ALL` value is in fact an alias for a set: the set over which the aggregate is computed. In this example, these respective sets are:

- `ALL(speciesType) = {'Setosa', 'Versicolour', 'Virginica'}`
- `ALL(petalLength) = {'low', 'medium', 'high'}`
- `ALL(petalWidth) = {'low', 'medium', 'high'}`

Thinking of the `ALL` value as an alias of these sets, makes it easier to understand and is how it operates internally. The `ALL` string really is just for display.

---

[10] Decorations are columns that do not appear in the GROUP BY list—and that are therefore not allowed to be projected (be in the `SELECT` list) in traditional SQL—but that are functionally dependent on the grouping columns. See [65] for more details.

[11] Modern RDBMSes such as MySQL 5.0 support this [67].

Data cube of 3D data

In this section, we consider all three categorical attribues of table 1: Petal Length, Petal Width and Species Type. Three categorical attributes implies 3D data and therefore we will need $1 + (2^3 - 1) = 1 + 7 = 8$ `UNION`ed queries. This is the case:

```
(
    -- Standard GROUP BY.
    SELECT PetalLength, PetalWidth, SpeciesType, COUNT(*)
    FROM Iris
    GROUP BY PetalLength, PetalWidth, SpeciesType
)
UNION
(
    -- Super-aggregate of SpeciesType.
    SELECT PetalLength, PetalWidth, ALL, COUNT(*)
    FROM Iris
    GROUP BY PetalLength, PetalWidth
)
UNION
(
    -- Super-aggregate of PetalWidth.
    SELECT PetalLength, ALL, SpeciesType, COUNT(*)
    FROM Iris
    GROUP BY PetalLength, SpeciesType
)
UNION
(
    -- Super-aggregate on PetalLength.
    SELECT ALL, PetalWidth, SpeciesType, COUNT(*)
    FROM Iris
    GROUP BY PetalWidth, SpeciesType
)
UNION
(
    -- Super-aggregate of PetalWidth and Speciestype.
    SELECT PetalLength, ALL, ALL COUNT(*)
    FROM Iris
    GROUP BY PetalLength
)
UNION
(
```

```
        -- Super-aggregate of PetalLength and PetalWidth.
        SELECT ALL, ALL, SpeciesType, COUNT(*)
        FROM Iris
        GROUP BY SpeciesType
    )
    UNION
    (
        -- Super-aggregate of PetalLength and Speciestype.
        SELECT ALL, PetalWidth, ALL, COUNT(*)
        FROM Iris
        GROUP BY PetalWidth
    )
    UNION
    (
        -- Super-aggregate of PetalLength, PetalWidth and Speciestype.
        SELECT ALL, ALL, ALL COUNT(*)
        FROM Iris
    )
```

Moreover, all 3 categorical attributes may assume 3 different values (“low”, “medium” and “high” for Petal Length and Petal Width, “Setosa”, “Versicolour” and “Virginica” for Species Type), thus $C_1 = C_2 = C_3 = 3$. This implies that the cardinality of the resulting data cube should be $(C_1 + 1) \times (C_2 + 1) + (C_3 + 1) = 4 \times 4 \times 4 = 64$.

This can also be checked by examining the table below (in which the results of the data cube operator are listed): there are $27 + (3 \times 9) + (3 \times 3) + 1 = 64$ rows, therefore its cardinality is 64.

Petal Length	*Petal Width*	*Species Type*	*Count*
The input data: no aggregation (27)			
low	low	Setosa	46
low	medium	Setosa	2
low	high	Setosa	0
medium	low	Setosa	2
medium	medium	Setosa	0
medium	high	Setosa	0
high	low	Setosa	0
high	medium	Setosa	0
high	high	Setosa	0

Petal Length	*Petal Width*	*Species Type*	*Count*
low	low	Versicolour	0
low	medium	Versicolour	0
low	high	Versicolour	0
medium	low	Versicolour	0
medium	medium	Versicolour	43
medium	high	Versicolour	3
high	low	Versicolour	0
high	medium	Versicolour	2
high	high	Versicolour	2
low	low	Virginica	0
low	medium	Virginica	0
low	high	Virginica	0
medium	low	Virginica	0
medium	medium	Virginica	0
medium	high	Virginica	3
high	low	Virginica	0
high	medium	Virginica	3
high	high	Virginica	44

By Petal Length and Petal Width (9)

low	low	ALL	46
low	medium	ALL	2
low	high	ALL	0
medium	low	ALL	2
medium	medium	ALL	43
medium	high	ALL	6
high	low	ALL	0
high	medium	ALL	5
high	high	ALL	46

By Petal Length and Species Type (9)

low	ALL	Setosa	48
medium	ALL	Setosa	2
high	ALL	Setosa	0
low	ALL	Versicolour	0
medium	ALL	Versicolour	46
high	ALL	Versicolour	4
low	ALL	Virginica	0

Petal Length	*Petal Width*	*Species Type*	*Count*
medium	ALL	Virginica	3
high	ALL	Virginica	47

By Petal Width and Species Type (9)

ALL	low	Setosa	48
ALL	medium	Setosa	2
ALL	high	Setosa	0
ALL	low	Versicolour	0
ALL	medium	Versicolour	45
ALL	high	Versicolour	5
ALL	low	Virginica	0
ALL	medium	Virginica	3
ALL	high	Virginica	47

By Petal Length (3)

low	ALL	ALL	48
medium	ALL	ALL	51
high	ALL	ALL	51

By Petal Width (3)

ALL	low	ALL	48
ALL	medium	ALL	50
ALL	high	ALL	52

By Species Type (3)

ALL	ALL	Setosa	50
ALL	ALL	Versicolour	50
ALL	ALL	Virginica	50

Total (1)

ALL	ALL	ALL	150

6.7 Performance

One key demand of OLAP applications is that queries be answered quickly. This is of course not a demand that is unique to OLAP: it is very rare that it is a requirement for a database or any other software to respond slowly. But OLAP's requirements are fairly stringent.
Fortunately, the multidimensional data model of OLAP is structured enough to allow this key demand to be approached.

If there is one key property to OLAP or multidimensional data analysis, then it is the ability to simultaneously aggregate across many dimensions. As we have discussed before (see section 6.5) and observed in full detail (see section 6.6), this translates to many simultaneous `GROUP BY` statements in SQL, which can result in a performance bottleneck.

More efficient schemes to perform these calculations have been researched by the University of Wisconsin-Madison [68], amongst others. Initially, they have focused on efficient algorithms to compute the cube operator, using the standard RDMBS techniques of sorting and hashing. As always, precomputing frequently used data can be used to speed up computer programs. In terms of multidimensional data analysis, aggregates on some subsets of dimensions can be precomputed. However, it is impossible to precompute everything, and we may end up precomputing unneeded aggregates. And because of the hierarchical nature (i.e. one subset of dimensions may be a subset of another subset), it is possible that the increase in required storage space may be unreasonable.

6.7.1 Efficient Cubing

The key to efficient cubing of relational tables is understanding how the cuboids[12] are related to each other. Then, one can exploit these relationships to minimize the number of calculations, and, more importantly (as virtually always for database systems): less I/O. [68] suggests an approach based on a hierarchical structure. They explore a class of sorting-based methods that attempt to minimize the number sorting steps by overlapping the computations of the various cuboids (and hence minimize the number of disk I/Os). This approach *always* performs significantly better than the prototype method referenced in section 6.5, which simply computes all required `GROUP BY` statements in sequence.

---

[12] Each combination of aggregates is called a *cuboid*, and all these cuboids together form the cube.

6.7.2 Precomputing for Speed: Storage Explosion

The more aggregates that are precomputed, the faster queries can be answered. However, it is difficult to say in advance how much space (storage) will be required for a certain amount of precomputation. There are different methods (discussed in [68]) to estimate this:

1. It is assumed that the data is *uniformly distributed.* This assumption allows for a mathematical approximation of the number of tuples that will appear in the result of the cube computation. This is simple statistics:

   > If r elements are chosen uniformly and at random from a set of n elements, the expected number of distinct elements obtained is $n - n(1 - 1/n)^r$.
   > — Feller in [69], page 241

   This can then be used to calculate the upper bound on the size of the cube. n is the product of the distinct number of values of all attributes on which is being grouped (i.e. the number of all possible different combinations of values) and r the number of tuples in the relation.

2. The second method uses a simple *sampling-based* algorithm: take a random subset of the table, compute the cube on that subset. Then estimate the size of the actual cube by linearly scaling the size of the cube of the sample by the $\frac{\textit{data size}}{\textit{sample size}}$ ratio. Clearly, if the random sample is biased, then our estimate will be skewed.
   The potential advantage over the first method (based on the uniform distribution assumption) is that this method examines a statistical subset, instead of just relying on cardinalities.

3. While the first two methods are simple applications of well-known statistics methods, the third tries to exploit the nature of the process that is being applied—essentially, data is being grouped according to the distinct values within the dimensions. This method therefore estimates the number of tuples in each grouping by estimating the number of distinct values in each particular grouping.
   A suitable *probabilistic* algorithm is [70]: it counts the number of distinct values in a multi-set, and makes the estimate after a single pass through the database, using only a fixed amount of memory. Hence this algorithm is a good starting point (single pass and fixed amount of memory are very desirable properties).

When comparing these three methods, the first method only works well when the data is approximately uniformly distributed (unsurprisingly), the sampling-based method is strongly dependent on the number of duplicates, and the probabilistic method performs very well under various degrees of skew. Hence the latter provides the most reliable, accurate and predictable estimate of the three considered algorithms.

6.7.3 The Impact of the Data Structure

While OLAP is the 'container term', there are actually many variants; including ROLAP (relational OLAP) and MOLAP (multidimensional OLAP). MOLAP stores the data in an optimized multidimensional array, whereas ROLAP stores the data in a relational database. Both have their advantages and disadvantages

A noteworthy remark: in [68], they found that it was surprisingly efficient to take the data set from a table in a relational database, convert this into a multidimensional array, *cube* the array and store it back in a database—this has been found to be more efficient than cubing the table directly!

6.7.4 Conclusion

Clearly, there is much more to the cube operator than meets the eye: a straightforward implementation is likely unable to attain the desired performance; optimizations on multiple levels are necessary. Precomputing parts seems an obvious optimization, but may require too much storage; estimating how much storage this will require is also not trivial. The data structures used should be carefully selected, since the performance impact can be tremendous. And, while complex, attempts to minimize overlapping computations can also help significantly.

6.8 Performance for range-sum queries and updates

For many applications (businesses), batch updates that are executed overnight are sufficient. However, in many cases, it is a necessity to have more frequent updates:

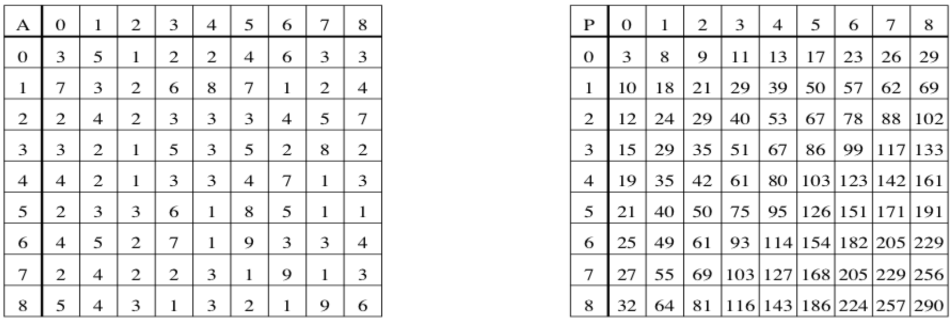
- For decision support and stock trading applications, instantaneous updates are crucial.
- OLAP implies *interactive* data analysis. Interactivity requires fast updates.
- Batch updates may have a low *average* cost per update, but performing the complete batch may take a considerable amount of time. For companies that can shut down every night, this might not be a problem, but for multinational companies, this poses a problem: at all times, access to the data is required somewhere around the world.

So, the ability to perform more frequent updates would enable other types of applications. As a side-effect, applications that don't really need it automatically get greater flexibility and 24 hour availability.

Discussed techniques

In the remainder of this section, three techniques are discussed:

1. Prefix Sum: this is an example of a technique that allows for fast range-sum queries that unfortunately can have very slow updates. It is very trivial, anybody with basic math skills could come up with it.
2. Relative Prefix Sum: this method is essentially the same as Prefix Sum, but stores its data in a smarter manner, to speed up updates.
3. Dynamic Data Cube: the third and last method is slightly inspired by (Relative) Prefix Sum but has as goal to have sub-linear performance, both for queries and updates! It is also far more efficient storage-wise: empty regions simply are *not* stored at all, whereas they would need to be created for the Prefix Sum and Relative Prefix Sum methods.
   It achieves all this by using a hierarchical (tree) structure, with each deeper level accessing more granular data.



A	0	1	2	3	4	5	6	7	8
0	3	5	1	2	2	4	6	3	3
1	7	3	2	6	8	7	1	2	4
2	2	4	2	3	3	3	4	5	7
3	3	2	1	5	3	5	2	8	2
4	4	2	1	3	3	4	7	1	3
5	2	3	3	6	1	8	5	1	1
6	4	5	2	7	1	9	3	3	4
7	2	4	2	2	3	1	9	1	3
8	5	4	3	1	3	2	1	9	6

P	0	1	2	3	4	5	6	7	8
0	3	8	9	11	13	17	23	26	29
1	10	18	21	29	39	50	57	62	69
2	12	24	29	40	53	67	78	88	102
3	15	29	35	51	67	86	99	117	133
4	19	35	42	61	80	103	123	142	161
5	21	40	50	75	95	126	151	171	191
6	25	49	61	93	114	154	182	205	229
7	27	55	69	103	127	168	205	229	256
8	32	64	81	116	143	186	224	257	290

Figure 16: The original array (A) on the left and the cumulative array used for the Prefix Sum method (P) on the right.

(Figure courtesy of [71].)

All are applicable only to range-sum queries, which is a specific type of query, but a very common one.

Finally, all of the techniques below rely on precomputation and therefore section 6.7.2 should be kept into account as well.

6.8.1 Prefix Sum

The essential idea of the Prefix Sum method is to precompute many prefix sums of the data cube, which can then be used to answer any range-sum query in constant time. The downside is a large update cost—in the worst case, an array needs to be rebuilt that has the same size as the data cube itself.

One could describe the prefix array by the following (very simple) formula, with P the prefix array and A the original array:

$$P[i,j] = \sum_{0 \le k \le i;\, 0 \le l \le j} A[k,l]$$

Because of the nature of a prefix sum, particular updates have the potential to cause enormous cascading updates. This becomes instantly obvious when shown the data that the Prefix Sum method stores. Therefore, an example has been included: please see figure 16.
For example, when cell $A[1,3]$ would be modified, almost entire P would need recalculating.

Discussing all details would lead us to far—if interested, it is recommended to consult the original paper [72]. The worst case update cost is $O(n^d)$.

6.8.2 Relative Prefix Sum

This method provides constant time queries with reduced update complexity (when compared to the Prefix Sum technique explained in [72] on which it builds). Therefore this method is more suitable for applications where constant time queries are a necessity but updates are more frequent than the Prefix Sum method allows.

The essence of the Relative Prefix Sum approach is to limit the cascading updates that result in poor update performance. It achieves this by partitioning the array that is to be updated into fixed size regions called *overlay boxes*, these are of equal size: k in each dimension. Thus each overlay box contains k^d cells, with d the number of dimensions. The explanations below are for the 2D case, because that is easier to explain and visualize, but the same techniques can be applied to arrays of any number of dimensions.

The *anchor* cell is the "upper left" cell of each overlay box.

For each overlay box, there is an *overlay array* and a *relative-prefix array.*

Overlay array

The overlay array (OL) stores information on the sums of the "preceding" regions. By "preceding", those regions that are more to the left and to the top in a typical 2D array are meant, that is, the regions on which it depends for its range sums.

In the two-dimensional example in figure 17, the cells in the top row and leftmost column contain the sums of the values in the corresponding shaded cells: those overlay cells aggregate the corresponding shaded cells. The other, empty cells in the overlay array are not needed and would therefore not be stored in an actual implementation.

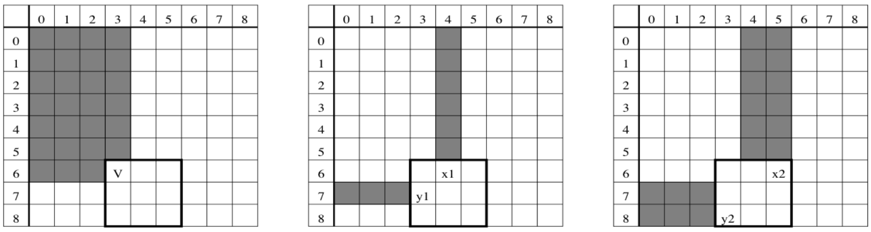


Figure 17: Calculation of overlay array values as the sum of the cells in the shaded cells.

(Figure courtesy of [71].)

More formally, the overlay array OL for the overlay box B, anchored at (i.e. with its anchor cell at) $(b_1, \ldots, b_d)$ aggregates k^d overlay cells $O = (o_1, \ldots, o_i, \ldots, o_d)$, which are those cells that satisfy for each dimension i: $b_i \leq o_i \leq b_i + k$, namely all cells in the overlay box B. Of those cells, only $k^d - (k-1)^d$ are used, namely those in the top row and the leftmost column. Then each cell in the overlay array is defined as follows:

$$\begin{aligned} OL[o_1, \ldots, o_d] &= \Big(\sum_{a_1=l_1}^{u_1} \cdots \sum_{a_d=l_d}^{u_d} A[a_1, \ldots, a_d]\Big) \\ &\quad -\Big(\sum_{a_1=m_1}^{v_1} \cdots \sum_{a_d=m_d}^{v_d} A[a_1, \ldots, a_d]\Big) \end{aligned}$$

with for all dimensions i:

if $o_i = b_i$: $\begin{cases} l_i = 0 \\ u_i = b_i \\ m_i = b_i \\ v_i = b_i \end{cases}$, and if $o_i > b_i$: $\begin{cases} l_i = b_i + 1 \\ u_i = o_i \\ m_i = b_i + 1 \\ v_i = o_i \end{cases}$

The $o_i = b_i$ case calculates the value for the anchor cell. The $o_i > b_i$ case calculates the other cells with values: those in the top row and the leftmost column.

Relative-prefix array

The relative-prefix array (RP) stores information on the relative prefix sums *within* each overlay box. Each region in RP contains prefix sums that are relative to the to the region enclosed by the box, that is, it is independent of other regions.

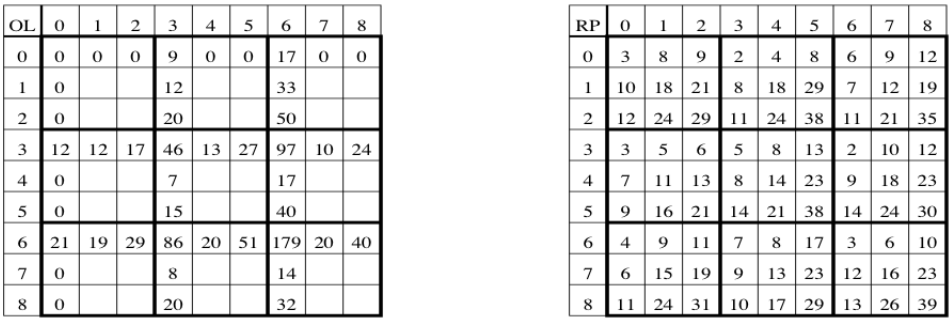
More formally, the relative-prefix array RP for the overlay box B, anchored at (i.e. with its anchor cell at) $(b_1, \ldots, b_d)$, each cell in the relative-prefix array is defined as follows:

$$RP[i_1, \ldots, i_d] = \sum_{a_1=b_1}^{b_1+k} \cdots \sum_{a_d=b_d}^{b_d+k} A[a_1, \ldots, a_d]$$

Combining the overlay array and relative-prefix arrays

By combining the information in both components (OL and RP), prefix sums can be constructed on the fly.

This too, can be made more clear through the use of figures. First look again at the right-hand side of figure 16. Then look at figure 18, which contains an example of the OL and RP components for figure 16.
It is clear that each cell in the array on the right-hand side of figure 16 can be calculated from the OL and RP components by adding the corresponding values stored in the OL and the RP.



OL	0	1	2	3	4	5	6	7	8
0	0	0	0	9	0	0	17	0	0
1	0			12			33		
2	0			20			50		
3	12	12	17	46	13	27	97	10	24
4	0			7			17		
5	0			15			40		
6	21	19	29	86	20	51	179	20	40
7	0			8			14		
8	0			20			32		

RP	0	1	2	3	4	5	6	7	8
0	3	8	9	2	4	8	6	9	12
1	10	18	21	8	18	29	7	12	19
2	12	24	29	11	24	38	11	21	35
3	3	5	6	5	8	13	2	10	12
4	7	11	13	8	14	23	9	18	23
5	9	16	21	14	21	38	14	24	30
6	4	9	11	7	8	17	3	6	10
7	6	15	19	9	13	23	12	16	23
8	11	24	31	10	17	29	13	26	39

Figure 18: The overlay array (OL) on the left and the relative prefix array (RP) on the right. The overlay boxes are drawn in thick lines for reference. *(Figure courtesy of [71].)*

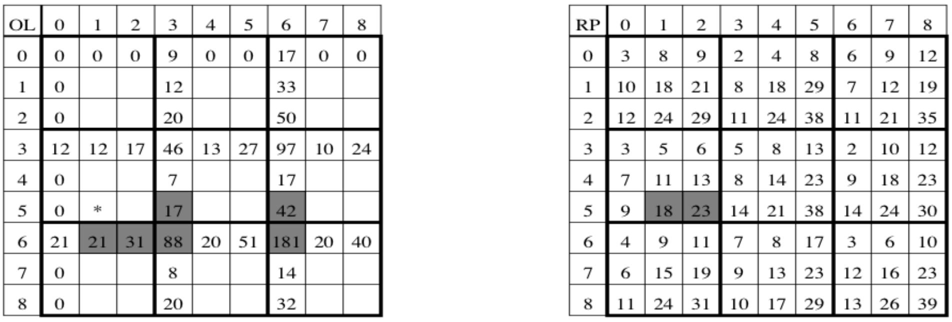
To calculate $SUM(A[0,0] : A[8,7])$, we must add $OL[6,6]$ (the anchor cell), $OL[8,6]$ (because our target cell is in column 8 and the anchor cell was in column 6, we need the value in the overlay array for column 8 as well), $OL[6,7]$ (analogously to the explanation for $OL[8,6]$) and $RP[8,7]$ (since that is our target cell). The result is $179 + 40 + 14 + 23 = 256$.

Other examples:

$$\begin{aligned} SUM(A[0,0] : A[4,0]) &= OL[3,0] + OL[4,0] + RP[4,0] \\ &= 9 + 0 + 4 = 13 \end{aligned}$$

$$\begin{aligned} SUM(A[0,0] : A[3,5]) &= OL[3,3] + OL[3,5] + RP[3,5] \\ &= 46 + 15 + 14 = 75 \end{aligned}$$

$$\begin{aligned} SUM(A[0,0] : A[6,3]) &= OL[6,3] + RP[6,3] \\ &= 97 + 2 = 99 \end{aligned}$$



OL	0	1	2	3	4	5	6	7	8
0	0	0	0	9	0	0	17	0	0
1	0			12			33		
2	0			20			50		
3	12	12	17	46	13	27	97	10	24
4	0			7			17		
5	0	*		17			42		
6	21	21	31	88	20	51	181	20	40
7	0			8			14		
8	0			20			32		

RP	0	1	2	3	4	5	6	7	8
0	3	8	9	2	4	8	6	9	12
1	10	18	21	8	18	29	7	12	19
2	12	24	29	11	24	38	11	21	35
3	3	5	6	5	8	13	2	10	12
4	7	11	13	8	14	23	9	18	23
5	9	18	23	14	21	38	14	24	30
6	4	9	11	7	8	17	3	6	10
7	6	15	19	9	13	23	12	16	23
8	11	24	31	10	17	29	13	26	39

Figure 19: Effects of an update to the cell with coordinates $(1,5)$—marked with an asterisk on the left.

(Figure courtesy of [71].)

Update performance

The goal of the Relative Prefix Sum method is to reduce the amount of update cascade. Now that has been demonstrated how this method works, it is time to look at the update performance. Suppose that $A[1,5]$ is updated from the value 3 to the value 5.

Since the OL contains the sums of the "preceding" regions, all values to the right and those in the first row below need to be updated. In the running example, these are cells $OL[3,5]$ and $OL[6,5]$ to the right and cells $OL[1,6]$, $OL[2,6]$, $OL[3.6]$ and $OL6,6]$ in the row below. This is demonstrated in figure 19. And since the RP contains relative prefix sums only for cells within the same overlay box, only cells $RP[1,5]$ and $RP[2,5]$ need to be updated (again, see figure 19).

Discussing all details would again lead us to far, for a detailed analysis it is recommended to read [71], but here it suffices to say that in the worst case, $(\frac{n}{k}+k-2)^d$ cells need to be updated (with d the dimensionality, n the number of possible attribute values and k the length of the overlay box in each dimension). The worst case update cost has been limited to $O(n^{\frac{d}{2}})$, which is significantly less than for the Prefix Sum method, since the exponent is only half as large (see section 6.8.1).

6.8.3 The Dynamic Data Cube

Like the Relative Prefix Sum method, this method also uses overlay boxes. But it uses multiple levels of overlay boxes, arranged in a hierarchy (more specifically: a tree structure). Through this particular structure (that will be explained more in-depth later on), the Dynamic Data Cube method is able to provide sub-linear performance ($O(\log^d n)$, with d again the dimensionality) for both range sum queries and updates on the data cube.

Overlay Boxes

The overlay boxes are similar to the ones used in the Relative Prefix Sum method, but they differ in the values they store, and in the number of overlay boxes.

The values that they store can best be explained through the help of a figure: see figure 20. Each box stores—just like the Relative Prefix Sum method—$k^d - (k-1)^d$ values (i.e. the leaf level stores 1 value, the level above that stores $4 - 1 = 3$ values, etc.); these values provide sums of regions *within* the overlay box. E.g., y_1 contains the sum of all the values of that row. Also, because sums of regions within the overlay box are stored, y_2 includes the value of y_1, etc. S is the cell that contains the subtotal for that overlay box.

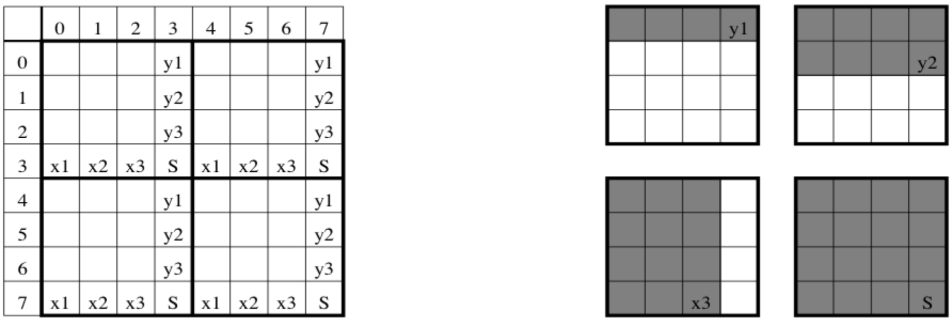
Most importantly, each overlay box is *independent* from the other ones at the same level in the hierarchy. This is different from the Relative Prefix Sum method, where each overlay box also contains the values for the "preceding" regions.

This also explains why the Dynamic Data Cube method uses the bottom row and rightmost column: it contains the subtotal for each region. Whereas the Relative Prefix Sum method uses the top row and leftmost column to store totals for the "preceding" regions in its overlay arrays and then uses the relative-prefix array to be able to calculate the other cells in that overlay box.

Construction

As stated before, overlay boxes are organized in a tree structure that recursively partitions the array. This tree structure is the reason that the number of overlay boxes differs from that of the Relative Prefix Sum method.

The root node of the tree contains the complete range of the array, in overlay



	0	1	2	3	4	5	6	7
0				y1				y1
1				y2				y2
2				y3				y3
3	x1	x2	x3	S	x1	x2	x3	S
4				y1				y1
5				y2				y2
6				y3				y3
7	x1	x2	x3	S	x1	x2	x3	S

Figure 20: Partitioning of an array into overlay boxes and calculation of overlay values..

(Figure courtesy of [71].)

boxes of size $k = \frac{n}{2}$. Each overlay box is again divided in half (so now $k = \frac{n}{4}$), and so on. This recursive partitioning continues until the leaf level, where $k = 1$ and each overlay box contains a single cell. At that point, the leaf level contains the values stored in the original array.
For a graphical explanation, see the three different levels, from root to leaf level, as illustrated in figure 21.

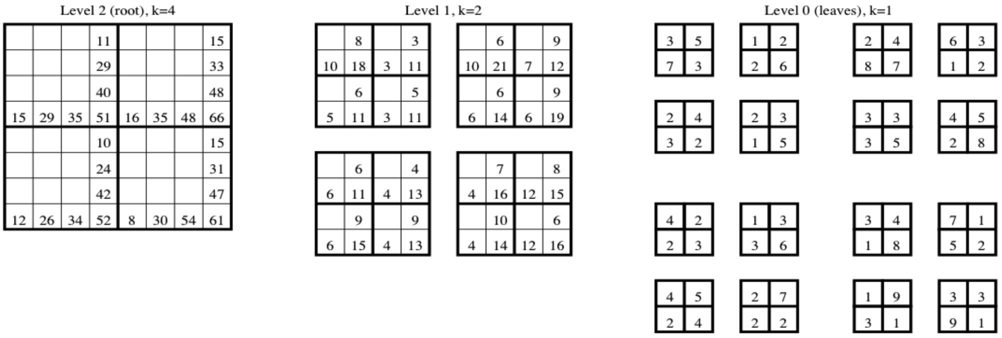
Because the overlay boxes are stored in special structures, sub-linear query and update times can be guaranteed. For two-dimensional overlays ($d = 2$), overlay boxes are not stored in arrays, but in a specialized hierarchical structure with an access and update cost of $O(\log n)$; for details on that see [73]. When the data cubes have a higher dimension ($d > 2$), the overlay box values of a d-dimensional data cube can be stored as $(d-1)$-dimensional data cubes in a recursive manner[13]—the recursion of course stops for $d = 2$.

Queries & Updates

The range sum for any query can be calculated by retrieving only overlay box values. The query begins at the root node of the tree and includes every overlay box that is “covered in every dimension” by the coordinates of the cell whose range sum we’re calculating (i.e. if that cell’s index is greater than or equal to the overlay box’ index in every dimension), i.e. the included overlay boxes contribute their subtotals to the sum. If the cell intersects the overlay box, then the box contributes the corresponding overlay value (a row

---

[13] The surfaces containing the overlay values of a d-dimensional overlay box are $(d-1)$-dimensional.



Level 2 (root), k=4

			11				15
			29				33
			40				48
15	29	35	51	16	35	48	66
			10				15
			24				31
			42				47
12	26	34	52	8	30	54	61

Level 1, k=2

	8		3		6		9
10	18	3	11	10	21	7	12
	6		5		6		9
5	11	3	11	6	14	6	19
	6		4		7		8
6	11	4	13	4	16	12	15
	9		9		10		6
6	15	4	13	4	14	12	16

Level 0 (leaves), k=1

3	5	1	2	2	4	6	3
7	3	2	6	8	7	1	2
2	4	2	3	3	3	4	5
3	2	1	5	3	5	2	8
4	2	1	3	3	4	7	1
2	3	3	6	1	8	5	2
4	5	2	7	1	9	3	3
2	4	2	2	3	1	9	1

Figure 21: Dynamic Data Cube: all levels of the tree.

(Figure courtesy of [71].)

sum value in a 2D data cube, such as y_2 in figure 20). Then, we go to a deeper level in the tree until we reach the leaf level.
Since overlay boxes at the same tree level do not intersect, at most one child will be traversed down. The same algorithm is applied again.
Thanks to this recursive nature, less values need to be retrieved, resulting in an overall query cost of $O(\log^d n)$—for details see again [73].
The same descent down the tree must be made when performing an update instead of a request, resulting in a worst case update cost that is identical to the overall query cost. Again, see [73] for details.

Dynamic Growth

Neither the Prefix Sum nor the Relative Prefix Sum methods carry optimizations to limit growth of the data cube. Instead, they assume that the size of each dimension is known a priori, or simply that size is not an issue. For some cases, it is more convenient (and space efficient) to grow the size of the data cube dynamically, just enough to suit the size of the data. For example, the number of possible values of an attribute could be large, but the number of *actual* different values that are taken is low.

The Prefix Sum and Relative Prefix Sum methods would need to grow new rows (for lack of a better term in >3D; more accurately: expansion in a specific dimension) for even a single cell in a previously non-existing area—see figure 22 for an example. The Dynamic Data Cube, on the other hand, could just grow into the required direction, affecting just one overlay box at each tree level.

This makes the Dynamic Data Cube a natural fit for data that contains large

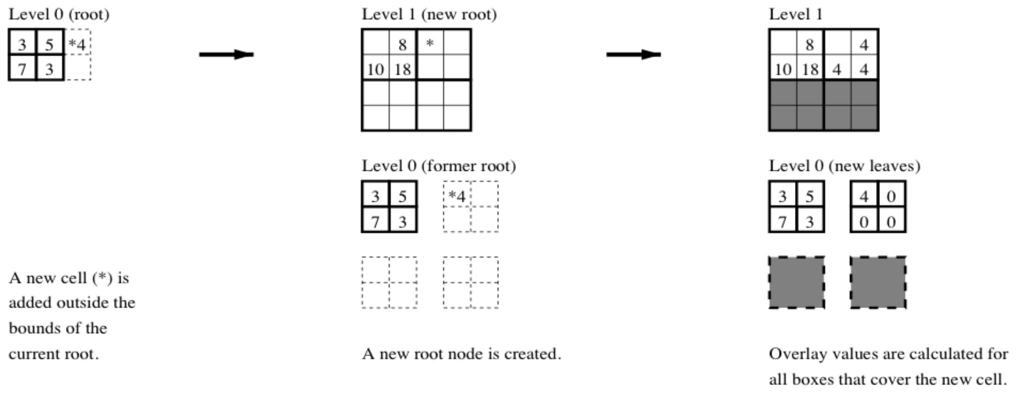


Figure 22: Growth of a Dynamic Data Cube—shaded areas don't store values.

(Figure courtesy of [71].)

non-populated regions: where there is no data, the overlay boxes will simply not be created. In other words: the Dynamic Data Cube *avoids the storage of empty regions*, unlike the Prefix Sum and Relative Prefix Sum methods.

7 Conclusion

The user begins by integrating Episodes with his web site, which will log the measured results to an Episodes log file. This log file by itself is a good set of data that can be interpreted, but it would be too time-intensive to manually analyze it. Hence the mining of meaningful associations between the context of a page view and a slow episode needs to be automated.

Episodes log mining (section 3), which is a specialization of web usage mining, has been studied from a high-level perspective: more detail would have added many implementation details, and the implementation belongs in part two of this thesis. Therefore, the necessary details will be added in part two of this thesis.

Also, because web usage mining is only designed to work with static data sets (that are updated in batches), regular data mining techniques were not sufficient for the purpose of this thesis, in which the goal is to detect problems instantaneously: we need mining of data *streams*, i.e. data sets to which data is being appended continuously.

Hence **data stream mining** (section 4) is the next subject that has been studied. We've looked at a large set of frequent item mining algorithms and two frequent item*set* mining algorithms, one of which builds upon a frequent item algorithm and the other of which builds upon a famous frequent itemset mining algorithm for static data sets, FP-growth.

However, frequent pattern mining algorithms can only find problems that persist over a certain period over time, that gradually grow and fade. We also need to be able to detect brief problems, e.g. caused by traffic spikes. That is, we also want to detect *infrequent issues.*

For this, we look into **anomaly detection** (section 5) in general and *contextual anomaly detection* in particular. We've discussed two contextual anomaly detection algorithms.

Finally, automatically detecting problems and presenting them to the user is excellent, but the user may also want to inspect all data himself. He may for example want to look at charts of average page loading time in Belgium and those in the United States. Or maybe compare this month's performance with that of a year ago in Internet Explorer, because optimizations have been made particularly for that browser. In other words: the user may want to inspect the data from multiple contexts, with each context limiting one or more contextual, categorical attributes (e.g. browser, location, operating system . . .) to one or more values.

That can be achieved with **OLAP** (section 6), which is designed to be able to quickly answer queries about multidimensional data. We've explained the *data cube* in-depth and discussed several algorithms that help improve its query performance.

How to continue from this first part of the thesis in the second part next year is explained in the outlook, which is the next section.

8 Outlook

This thesis text has only partially been completed: it is only part one. In part two, there likely will need to be some additional literature study to cover the missing gaps (see 8.1). However, for the most part it will consist of implementing what I've always intended to build (see the introduction again, in section 1); a preliminary toolchain has already been devised (see 8.3). For all of this, a planning has been created (see 8.4).

8.1 Further Literature Study

- The presented contextual anomaly detection algorithms are in fact *prediction* algorithms: I've been unable to find actual *detection* algorithms so far. Surely, there must be some, and they need to be evaluated.
- The data cube has been covered extensively in this literature study. However, most of it assumes *static data sets* (i.e. only updated in batches at an infrequent pace), while it is likely that also for OLAP, explicit support and optimizations for *data streams* will be necessary. A starting point can be found in [41].

8.2 Implementation

What the implementation will exactly entail has already been indicated in the introduction (see section 1), in which the goals for this thesis have also been stated. The details of the implementation will still have to be worked out in part 2 of this thesis. However, it has not yet been explained *how* anything should be implemented, i.e. using which algorithms. That too, should be done in the second part of this thesis. Of course, it was impossible to write about the "how" part when the literature study was not yet written, and it is possible that the "what" part may change due to low feasibility of some of the desirable features.

In other words: the specification and design of the implementation have not yet been defined, but will be in the second part of this thesis. They will of course be based on the results found in the literature study.

8.3 Preliminary Toolchain

Currently, I plan to *not* implement this thesis using Google AppEngine as Steve Souders suggested (see section 1), but to use C++/Qt because it allows for far more efficient resource usage, is not reliant on a specific cloud computing platform (which is what Google AppEngine is) and because Google AppEngine might take too much time to get up to speed with. Thus, I will write a desktop application and not a web application. This implies that the Episodes log file will somehow have to be sent from the web server to the desktop application. This can be achieved through a very simple web application.

This desktop application will in fact contain an embedded web browser which will contain the actual UI (i.e. one cannot visit any web sites using it, it's merely used for its XHTML/CSS/JS/SVG capabilities). By building the actual UI using web technology, this makes the application easier to port to a web application in the future.

Note that this is *preliminary* and my view on this may still change.

- Calculations: C++/Qt.
- UI: QtWebKit to embed a browser in the C++/Qt application, XHTML/CSS/JS/Raphael for the actual UI (Raphael is a JavaScript library to create vector graphics in general and charts in specific).
- Episodes log file passthrough web application: PHP.

8.4 Planning

The planning is on a per-month basis, with goals to be completed by the end of each month:

Month	Goals
September 2010	further literature study + episodes log mining
October 2010	data stream mining
November 2010	OLAP + initial UI
December 2010	finish UI + anomaly detection

9 Glossary

binarization similar to discretization, but instead of transforming into categorical attributes, transformations into one or more binary attributes are made
— based on [25], pages 57—63

browser A web browser is an application that runs on end user computers to view web sites (which live on the World Wide Web). Examples are Firefox, Internet Explorer, Safari and Opera.

categorical attributes also known as qualitative attributes; attributes without numeric properties: they should be treated like *symbols*; subclasses of this type of attribute are nominal and ordinal attributes
— based on [25], pages 25—27

CDN A content delivery network (CDN) is a collection of web servers distributed across multiple locations to deliver content more efficiently to users. The server selected for delivering content to a specific user is typically based on a measure of network proximity.

component A component of a web page, this can be a CSS style sheet, a JavaScript file, an image, a font, a movie file, et cetera. Synonyms: resource, web resource.

DBMS a computer program that aids in controlling the creation, usage and maintenance of a database

discretization some kinds of processing data require categorical attributes; if these need to be applied on a continuous attribute, this continuous attribute may need to be transformed into a categorical attribute: this is called *discretization*. Additionally, if the resulting categorical attribute has a large number of values (categories), it may be useful to reduce the number of categories by combining some of them.
This is necessary for e.g. histograms.
— based on [25], pages 57—63

episode An episode in the page loading sequence.

Episodes The Episodes framework [5] (note the capital 'e').

page loading performance The time it takes to load a web page and all its components.

page rendering performance The time the server needs to render a web page.

PoP A Point of Presence is an access point to the internet where multiple Internet Service Providers connect with each other.

quantitative attributes also known as numeric attributes; attributes that can be represented as numbers and have most of the properties of numbers; either integer-valued or continuous; subclasses of this type of attribute are interval and ratio attributes
— based on [25], pages 25—27

RDBMS a relational DBMS that is based on the relational model, as introduced by Codd. Examples are MySQL, PostgreSQL, SQL Server, Oracle . . .

web page An (X)HTML document that potentially references components.

References

[1] *Improving Drupal's page loading performance*, Wim Leers, Universiteit Hasselt, 2009, `http://wimleers.com/blog/finished-my-bachelor-degree`

[2] *Drupal*, `http://drupal.org/`

[3] *File Conveyor*, `http://fileconveyor.org/`

[4] *High Performance Web Sites*, Steve Souders, 2007, O'Reilly, `http://stevesouders.com/hpws/`

[5] *Episodes: a Framework for Measuring Web Page Load Times*, Steve Souders, July 2008, `http://stevesouders.com/episodes/paper.php`

[6] *Episodes: a shared approach for timing web pages*, Steve Souders, 2008, `http://stevesouders.com/docs/episodes-tae-20080930.ppt`

[7] *Gomez*, `http://www.gomez.com/`

[8] *Keynote*, `http://www.keynote.com/`

[9] *WebMetrics*, `http://www.webmetrics.com/`

[10] *Pingdom*, `http://pingdom.com/`

[11] *Episodes module for Drupal*, `http://drupal.org/project/episodes`

[12] *Deep Tracing of Internet Explorer*, John Resig, Mozilla, November 17, 2009, `http://ejohn.org/blog/deep-tracing-of-internet-explorer/`

[13] *An Update for Google Chrome's Developer Tools*, Pavel Feldman, Google, November 30, 2009, `http://code.google.com/events/io/2009/sessions/MeasureMillisecondsPerformanceTipsWebToolkit.html`

[14] *Yahoo! YSlow*, `http://developer.yahoo.com/yslow/`

[15] *Google Page Speed*, `http://code.google.com/speed/page-speed/`

[16] *A 2x Faster Web*, The Chromium Blog, Mike Belshe, November 11, 2009, `http://blog.chromium.org/2009/11/2x-faster-web.html`

[17] *Making browsers faster: Resource Packages*, Alexander Limi, November 17, 2009, `http://limi.net/articles/resource-packages/`

[18] *Fewer requests through resource packages*, Steve Souders, November 18, 2009, `http://www.stevesouders.com/blog/2009/11/18/fewer-requests-through-resource-packages/`

[19] *Web Timing (Working Draft)*, Zhiheng Wang, Google Inc., September 26, 2009, `http://dev.w3.org/2006/webapi/WebTiming/`

[20] *Google: Page Speed May Become a Ranking Factor in 2010*, WebProNews, November 19, 2009, `http://www.webpronews.com/topnews/2009/11/13/google-page-speed-may-be-a-ranking-factor-in-2010`

[21] *Using site speed in web search ranking*, Google Webmaster Central Blog, April 9, 2010, `http://googlewebmastercentral.blogspot.com/2010/04/using-site-speed-in-web-search-ranking.html`

[22] *How fast is your site?*, Webmaster Central Blog, Sreeram Ramachandra & Arvind Jain, December 2, 2009, `http://googlewebmastercentral.blogspot.com/2009/12/how-fast-is-your-site.html`

[23] *Google Analytics*, `http://google.com/analytics`

[24] *Google AppEngine*, `http://code.google.com/appengine`

[25] *Introduction to Data Mining*, Pang-Ning Tan; Michael Steinbach; Vipin Kumar, Pearson-Addison Wesley, 2006

[26] *UCI Machine Learning Repository*, R.A. Fisher, 1936, `http://archive.ics.uci.edu/ml/datasets/Iris`

[27] *Web Data Mining*, Bing Liu, 2008

[28] *Web Mining Course*, Gregory Piatetsky-Shapiro, KDnuggets, 2006, `http://www.kdnuggets.com/web_mining_course/`

[29] *Log Files—Apache HTTP Server*, `http://httpd.apache.org/docs/1.3/logs.html`

[30] *Computer Networking: A Top-Down Approach (4th Edition)*, James F. Kurose; Keith W. Ross, Addison Wesley, 2007

[31] *Classless Inter-Domain Routing*, `http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing`

[32] *Mining association rules between sets of items in large databases*, R. Agrawal; T. Imielinski; A. N. Swami, *Proc. ACM SIGMOD, pages 207–216*, 1993

[33] *Mining quantitative association rules in large relational tables*, R. Srikant; R. Agrawal, *Proc. ACM SIGMOD*, 1996

[34] *Mining Generalized Association Rules*, Ramakrishnan Srikant; Rakesh Agrawal, *Proceedings of the 21th International Conference on Very Large Data Bases*, p.407-419, September 11-15, 1995

[35] *Mining Rank-Correlated Sets of Numerical Attributes*, Toon Calders (University of Antwerp); Bart Goethals (Szczecin University of Technology), *Proc. KDD'06*

[36] *Rank Correlation Methods*, M. Kendall, Oxford University Press, 1990

[37] *Measures of Association*, A.M. Liebetrau, *volume 32 of Quantitative Applications in the Social Sciences*, Sage Publications, 1983

[38] *The art and craft of postload preloads*, Stoyan Stefanov, August 2009, `http://www.phpied.com/the-art-and-craft-of-postload-preloads/`

[39] *Preload CSS/JavaScript without execution*, Stoyan Stefanov, April 2010, `http://www.phpied.com/preload-cssjavascript-without-execution/`

[40] *Same Origin Policy*, W3C, `http://www.w3.org/Security/wiki/Same_Origin_Policy`

[41] *Data Mining: Concepts and Techniques*, Jiawei Han; Micheline Kamber, *Morgan Kaufmann*, 2006

[42] *Approximate Query Processing Using Wavelets*, K. Chakrabarti; M. Garofalakis; R. Rastogi; K. Shim, Proceedings of the International Conference on Very Large Databases, 2000

[43] *The space complexity of approximating the frequency moments*, N. Alon; Y. Matias; M. Szegedy, *Proceedings of the Twenty-Eighth Annual ACM Symposium on theory of Computing*, 1996

[44] *Optimal approximations of the frequency moments of data streams*, P. Indyk; D. Woodruff, *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, 2005

[45] *Simpler algorithm for estimating frequency moments of data streams*, L. Bhuvanagiri, S. Ganguly; D. Kesh; C. Saha, *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, 2006

[46] *Models and issues in data stream systems*, B. Babcock; S. Babu; M. Datar; R. Motwani; J. Widom, *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2002

[47] *Tracking join and self-join sizes in limited storage*, N. Alon, P. Gibbons; Y. Matias; M. Szegedy, *Proc. of the 1999 ACM Symp. on Principles of Database Systems, pages 10–20*, 1999.

[48] *The space complexity of approximating the frequency moments*, N. Alon; Y. Matias; M. Szegedy, *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996

[49] *New Sampling-Based Summary Statistics for Improving Approximate Query Answers*, P. B. Gibbons; Y. Matias, *SIGMOD RECORD, 1998, VOL 27; NUMBER 2, pages 331-342*, 1998

[50] *Synopsis data structures for massive data sets*, P. B. Gibbons; Y. Matias, *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms table of contents*, 1999

[51] Run-Length Encoding, `http://en.wikipedia.org/wiki/Run-length_encoding`

[52] *An Improved Data Stream Summary: The Count-Min Sketch and Its Applications*, G. Cormode; S. Muthukrishnan, *LECTURE NOTES IN COMPUTER SCIENCE 2004, issue 2976, pages 29-38*, 2004

[53] *Approximate frequency counts over data streams*, G. S. Manku; R. Motwani, *Proceedings of the 28th international conference on Very Large Data Bases*, 2002

[54] *Random sampling with a reservoir*, J. S. Vitter, *ACM Transactions on Mathematical Software (TOMS)*, 1985

[55] *Finding Frequent Items in Data Streams*, M. Charikar; K. Chen; M. Farach-Colton, *LECTURE NOTES IN COMPUTER SCIENCE, 2002, ISSU 2380, pages 693-703*, 2002

[56] *Probabilistic Lossy Counting: An efficient algorithm for finding heavy hitters*, X. Dimitropoulos; P. Hurley; A. Kind, *ACM SIGCOMM COMPUTER COMMUNICATION REVIEW 2008, VOL 38; NUMB 1, pages 5-16*, 2008

[57] *A proof for the queueing formula: $l = \lambda w$*, J. D. C. Little, *Operations Research, 9(3):383–387*, 1961

[58] *Mining Frequent Patterns in Data Streams at Multiple Time Granularities*, C. Giannella; J. Han; J. Pei; X. Yan; P. S. Yu, *Next generation data mining*, 2003

[59] *A simple algorithm for finding frequent elements in streams and bags*, R. M. Karp; S. Shenker; C. H. Papadimitriou, *ACM TRANSACTIONS ON DATABASE SYSTEMS, 2003, VOL 28; PART 1, pages 51-55*, 2003

[60] *Fast algorithms for mining association rules*, R. Agrawal; R. Srikant, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, 1994

[61] *Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach*, J. Han; J. Pei; Y. Yin; R. Mao, *DATA MINING AND KNOWLEDGE DISCOVERY, 2004, VOL 8; NUMBER 1, pages 53-87*, 2000

[62] *Anomaly Detection: A Survey*, V. Chandola; A. Banerjee; V. Kumar, *ACM Computing Surveys (CSUR) Volume 41, Issue 3*, 2009

[63] *Learning to Predict Rare Events in Event Sequences*, G. M. Weiss; H. Hirsh, *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining*, 1998

[64] *Predicting Rare Events In Temporal Domains*, R. Vilalta; S. Ma, *Proceedings of the 2002 IEEE International Conference on Data Mining*, 2002

[65] *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals*, Jim Gray (Microsoft); Adam Bosworth (Microsoft); Andrew Layman (Microsoft); Hamid Pirahesh (IBM), 1996

[66] *ISO/IEC 9075-1:2008*, 2009, `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45498`

[67] *`GROUP BY` Modifiers*, MySQL 5.0 Reference Manual, `http://dev.mysql.com/doc/refman/5.0/en/group-by-modifiers.html`

[68] *Cubing Algorithms, Storage Estimation, and Storage and Processing Alternatives for OLAP*, Prasad M. Deshpande; Jeffrey F. Naughton; Karthikeyan Ramasamy; Amit Shukla; Kristin Tufte; Yihong Zhao, University of Wisconsin-Madison, *Bulletin of the Technical Committee on Data Engineering* Vol. 20 No. 1, 1997

[69] *An Introduction to Probability Theory and Its Applications*, W. Feller, 1957

[70] *Probabilistic Counting Algorithms for Database Applications*, P. Flajolet; G.N. Martin, Journal of Computer and System Sciences, *Journal of Computer and System Sciences* 31(2): 182-209, 1985.

[71] *Data Cubes in Dynamic Environments*, Steven P. Geffner; Mirek Riedewald; Divyakant Agrawal; Amr El Abbadi, University of California, *Bulletin of the Technical Committee on Data Engineering* Vol. 22 No. 4, 1999

[72] *Range Queries in OLAP Data Cubes*, C. Ho; R. Agrawal; N. Megiddo; R. Srikant, *Proc. ACMSIGMOD*, 1997

[73] *The dynamic data cube*, S. Geffner; D. Agrawal; A. El Abbadi, *Proc. EDBT*, 2000