POCS: Technical Writing

Prof. George Candea
School of Computer & Communication Sciences

"This is what writing a paper with a first-year PhD student is like"

There is too much stuff to read

Feed the reader reasons to continue reading

Good tech writing is rare

Good tech writing can be learned

Master the language

http://users.auth.gr/ksiop/phd_funny/research_in_progress__funny/pic10.gif

Good Writing

… so I wait for you like a lonely house
till you will see me again and live in me.
Till then my windows ache.

(Pablo Neruda)

The dopamine signaling in the nucleus accumbens of my basal forebrain is lower than normal due to your physical absence.

The performance of our cache becomes tremendously small when the data is accessed in a very adversarial manner.

(1st year PhD student)

The hit rate of the CPU cache drops by up to 95% if programs consistently write to the least-recently read memory address.

Lyrical writing

Technical writing

What ?   To whom ?

How to transfer efficiently?

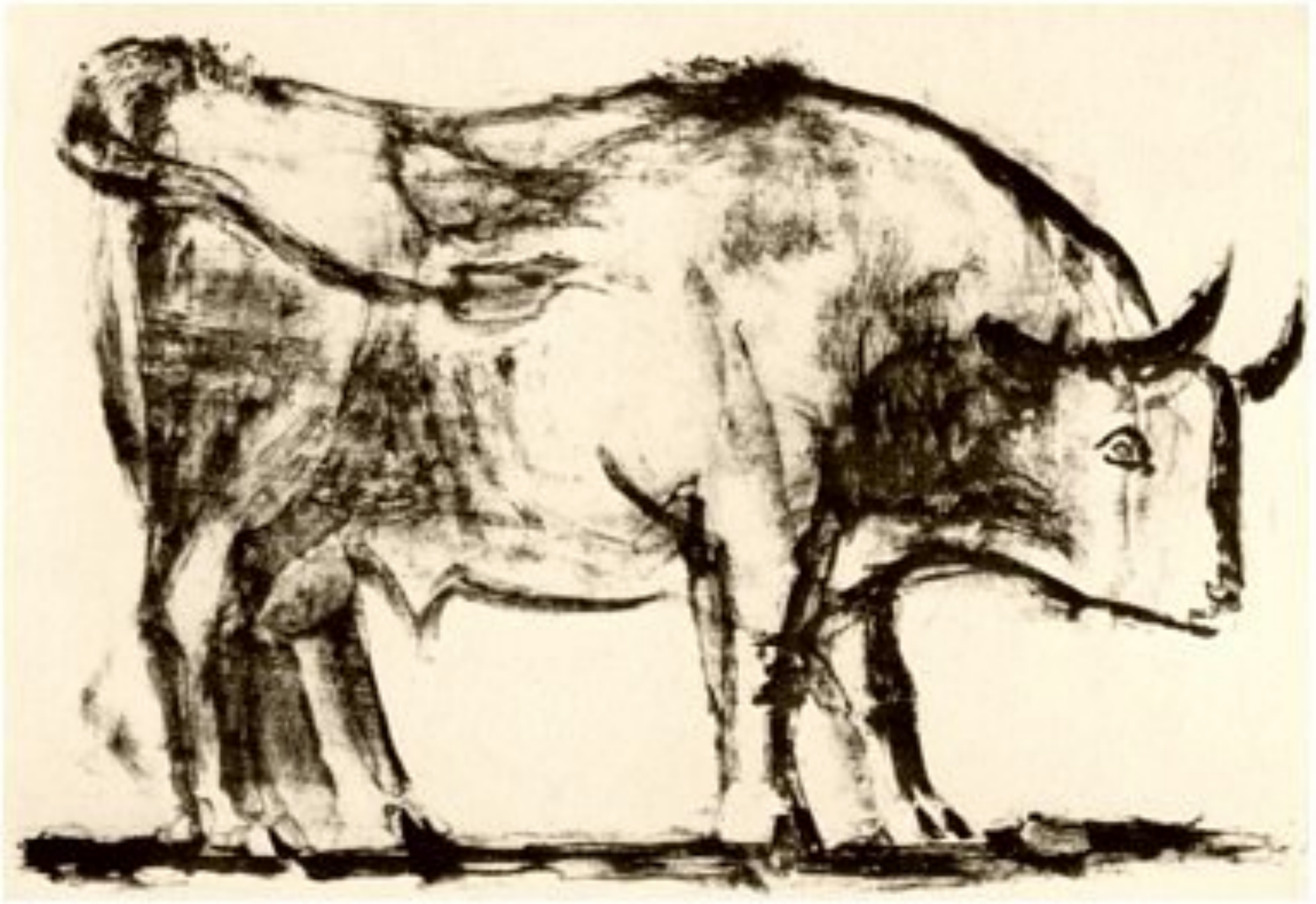
Perfection must be reached by degrees;
she requires the slow hand of time.

(attributed to Voltaire)

The Writing Process

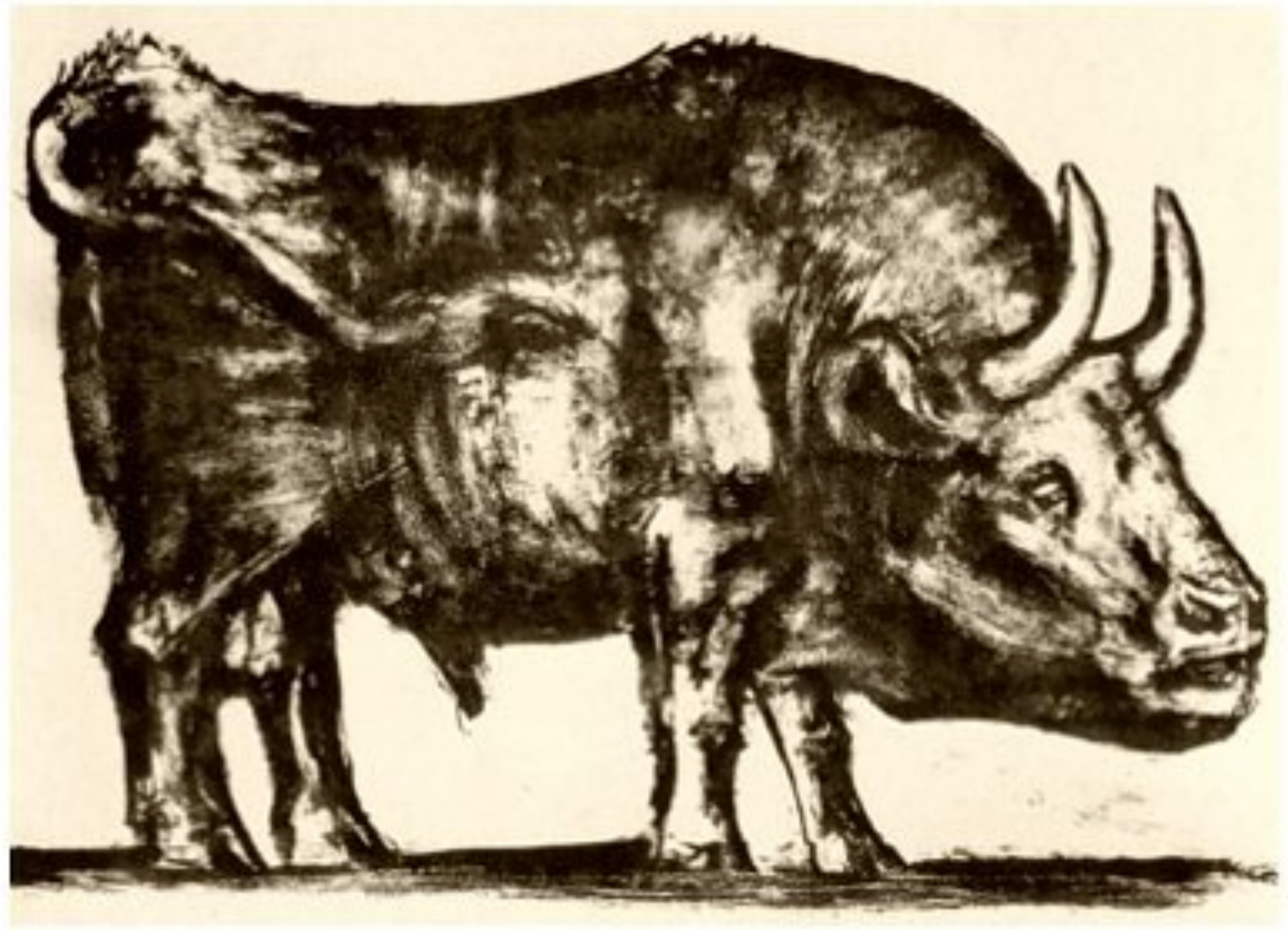
Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.

(Antoine de Saint-Exupéry, "L'Avion", Ch. III)



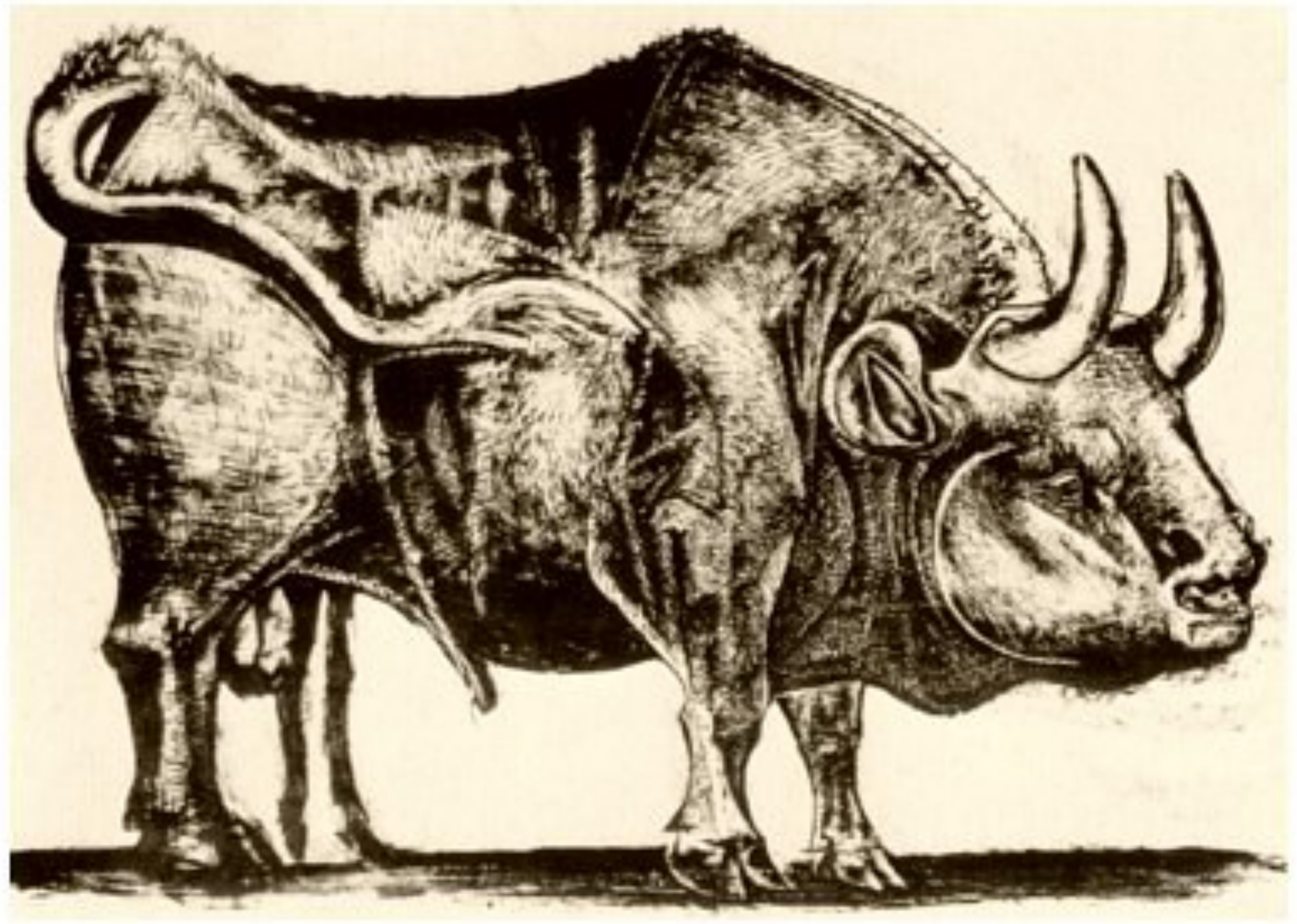
December 5, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



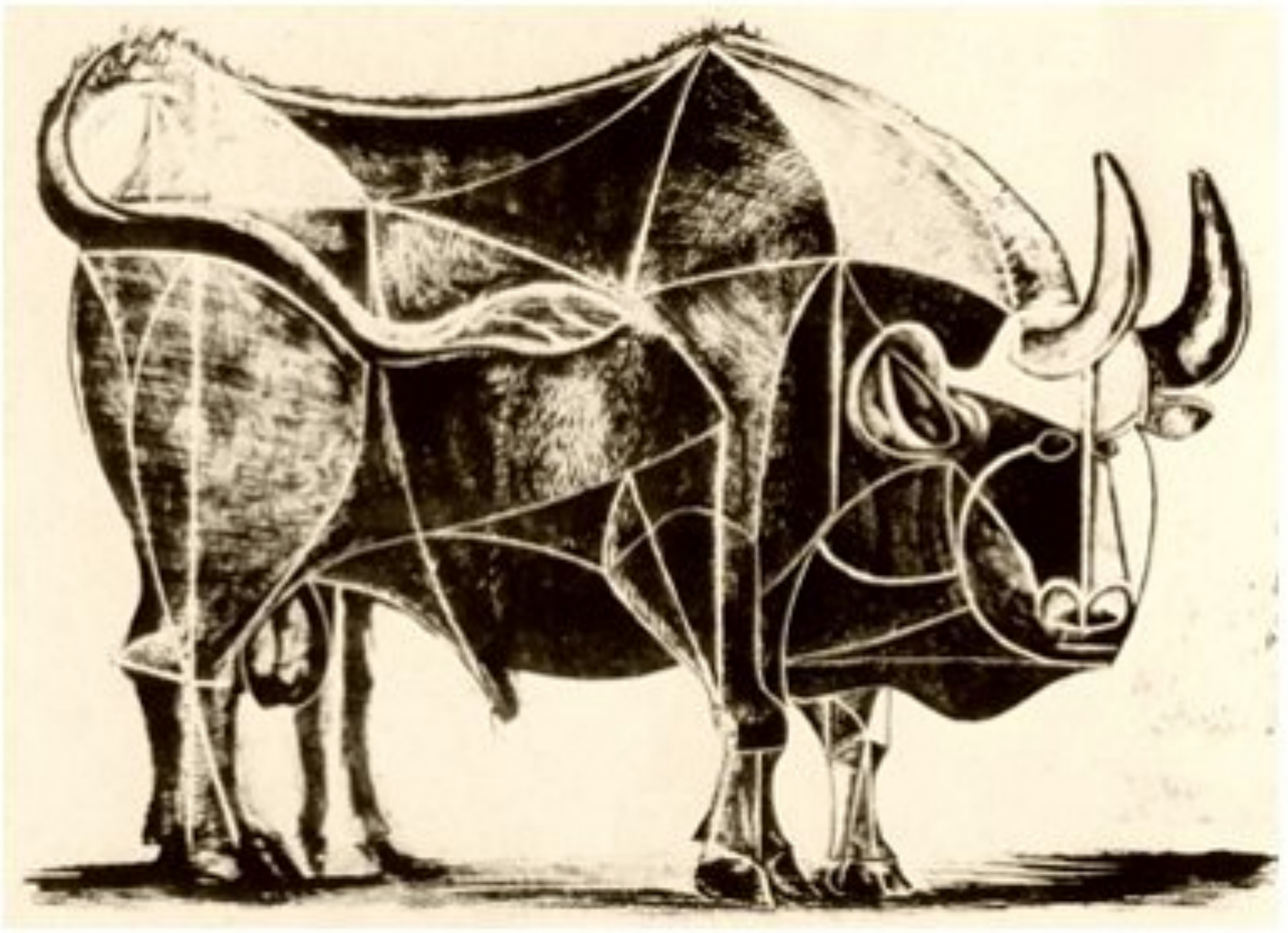
December 12, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



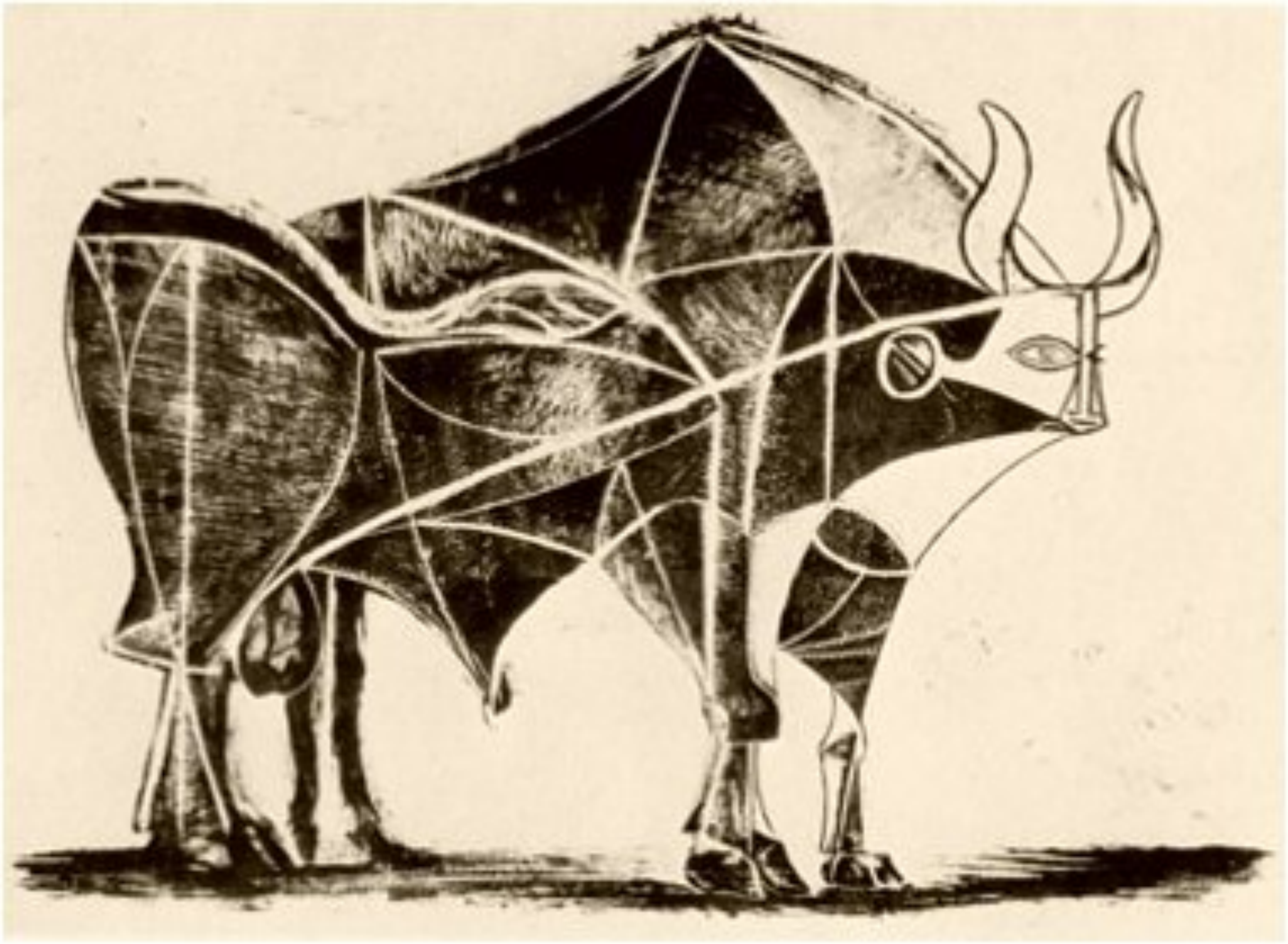
December 18, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



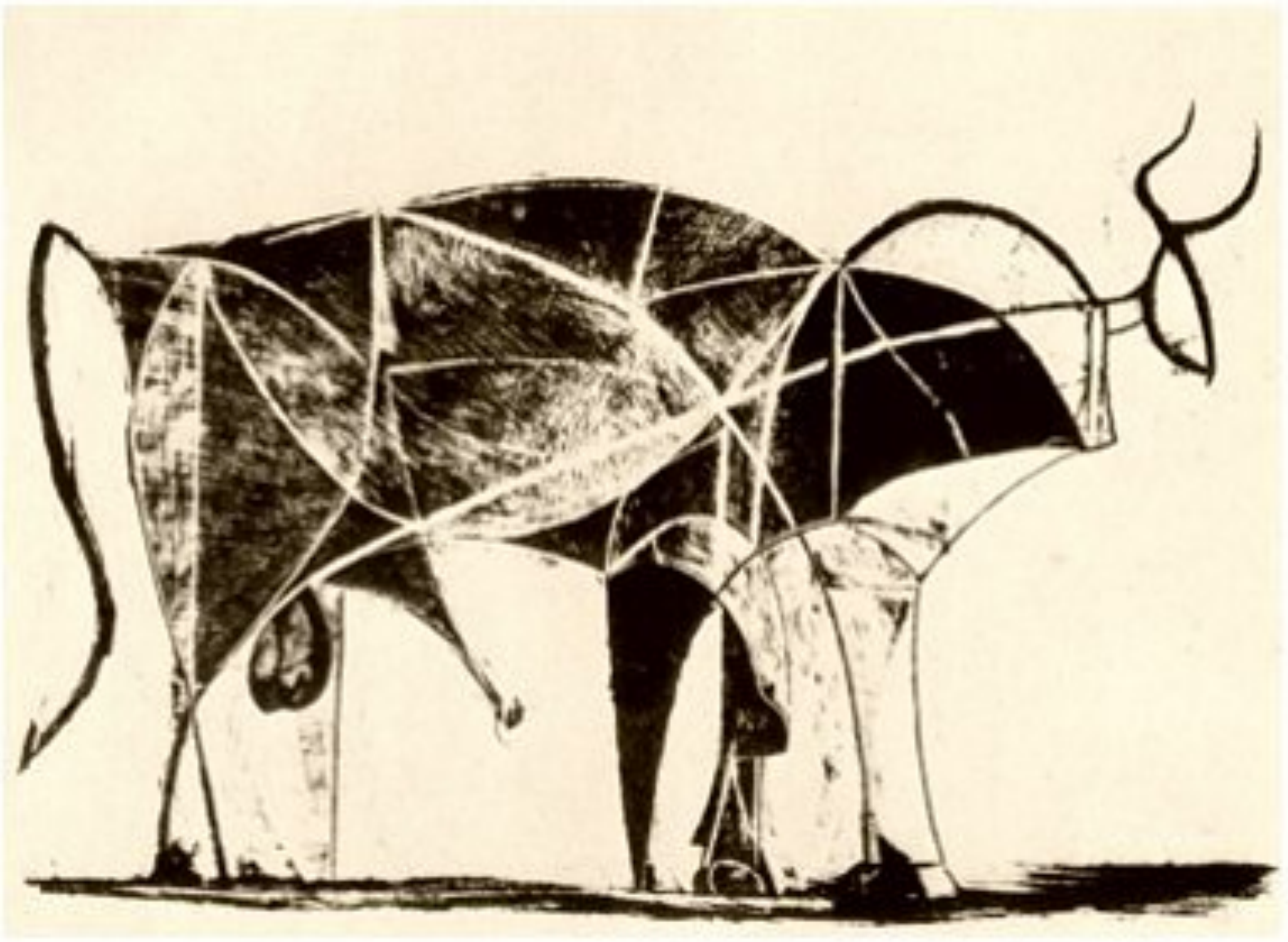
December 22, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



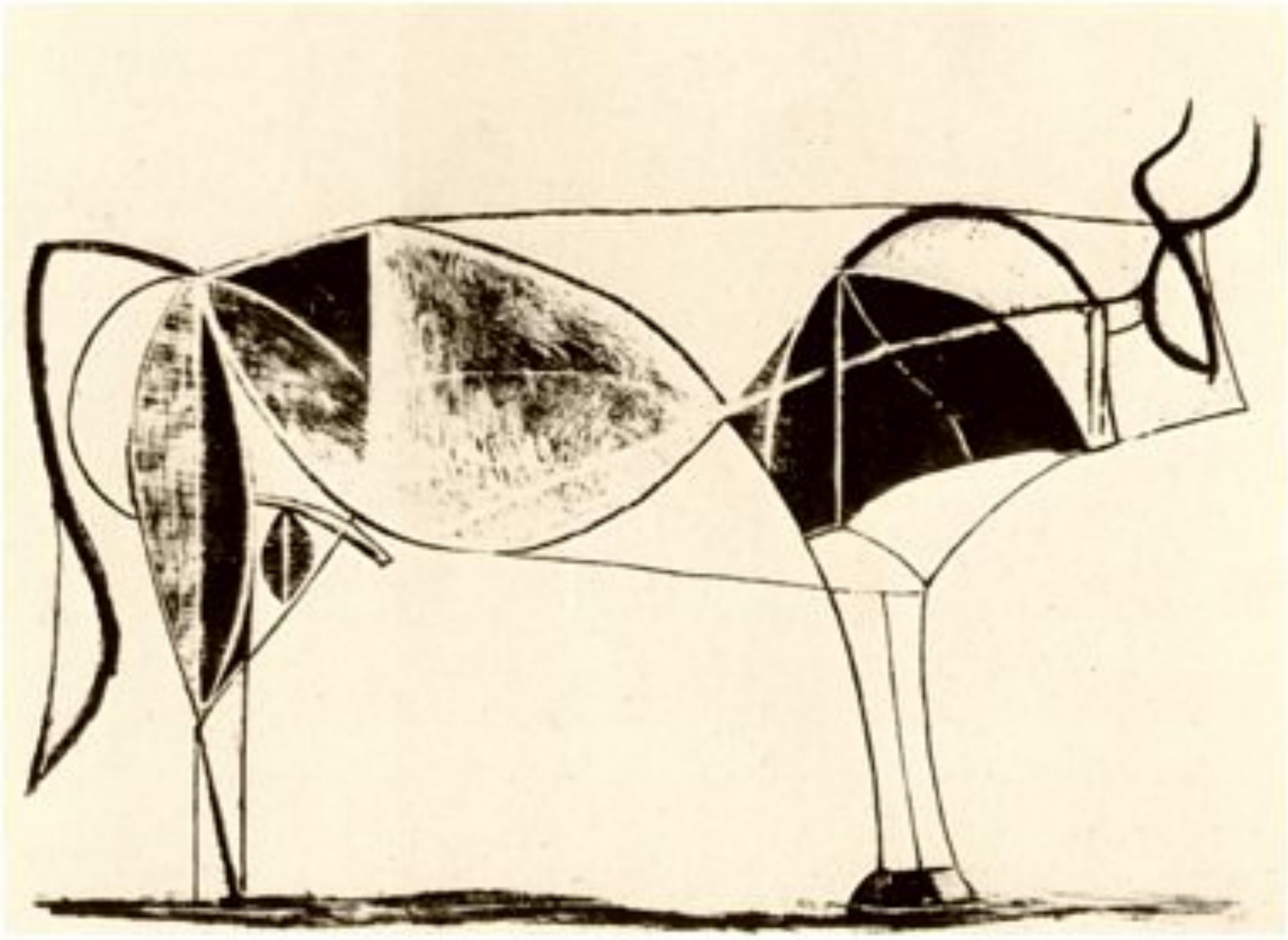
December 24, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



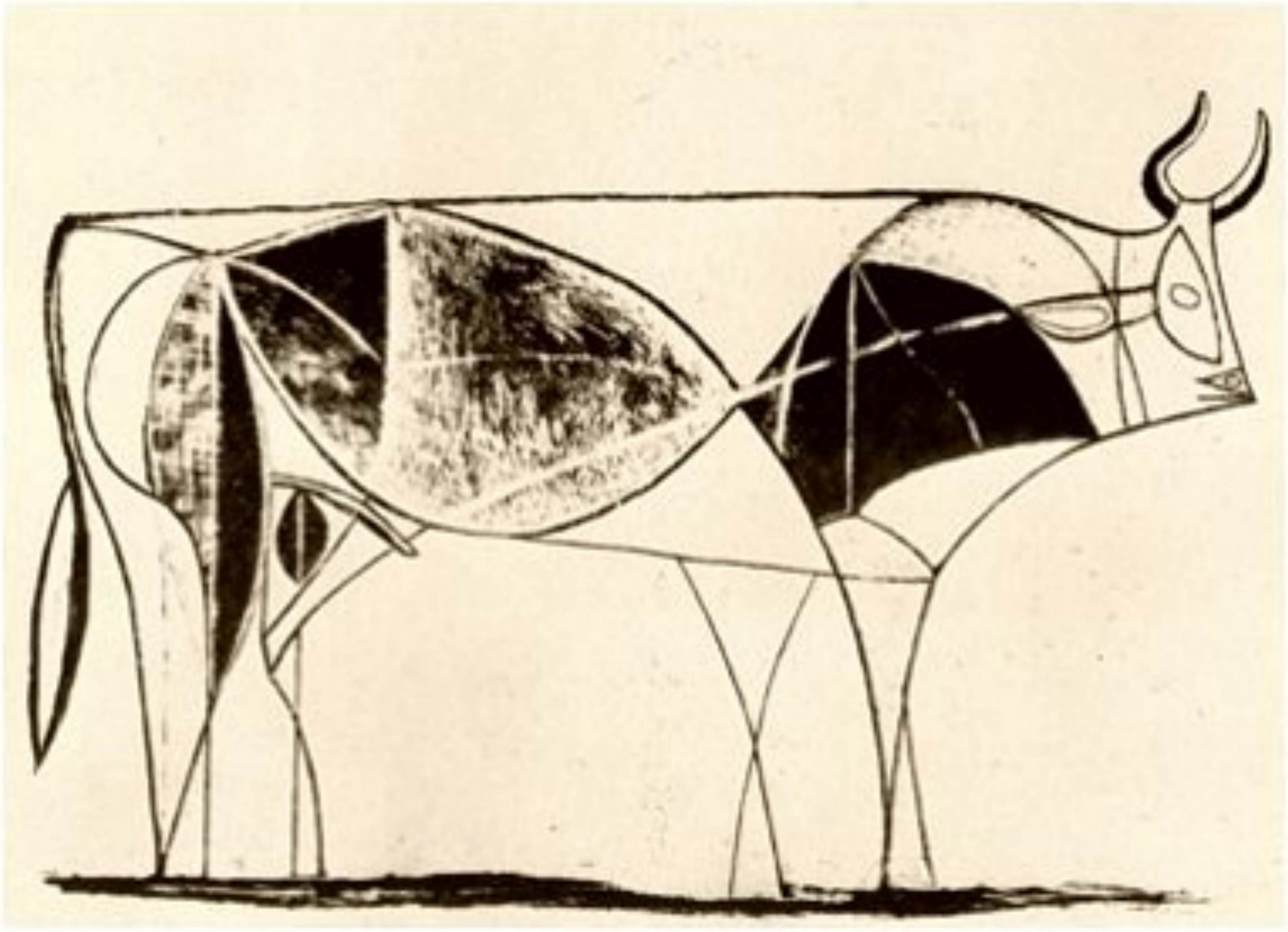
December 26, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



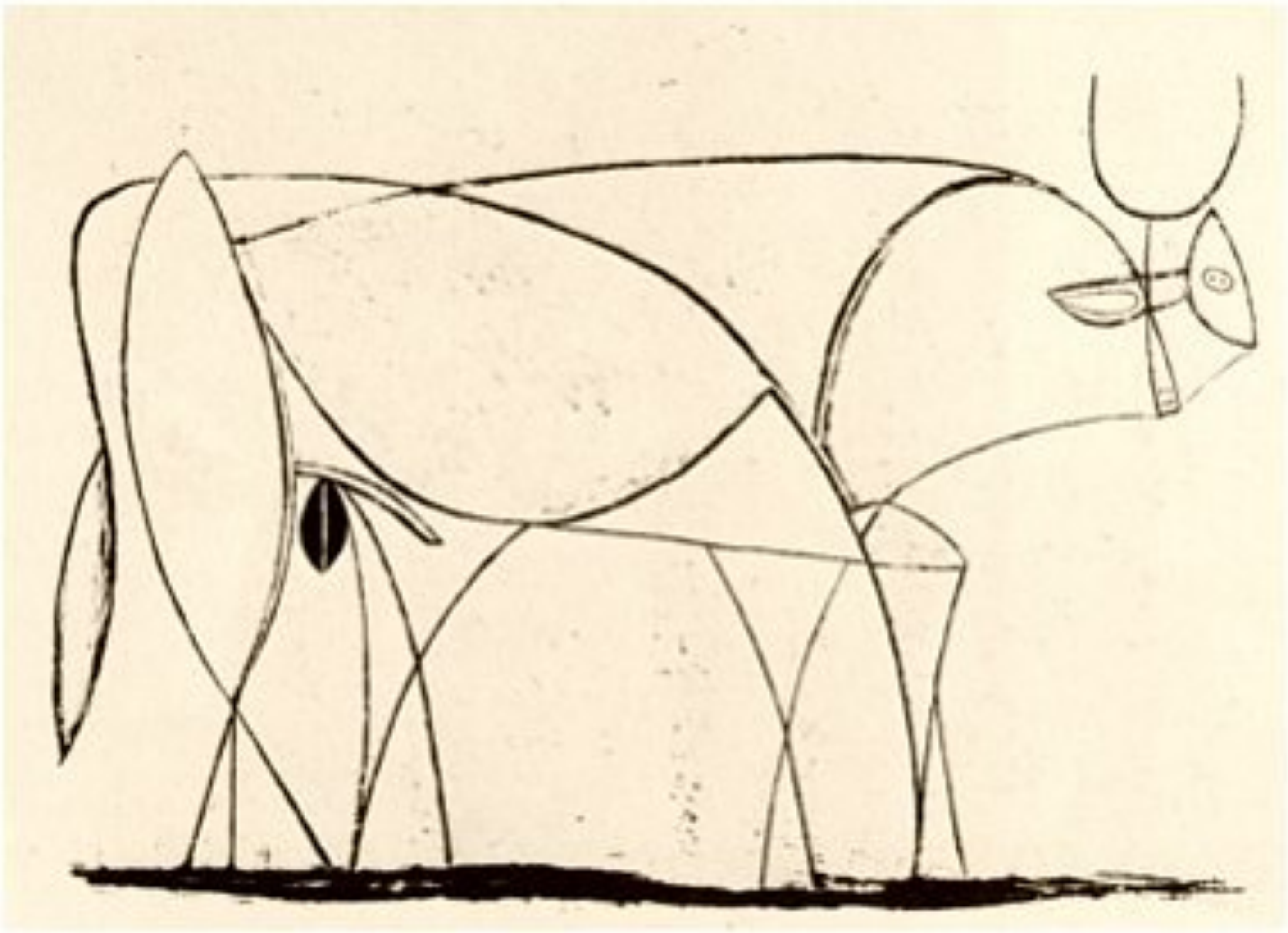
December 28, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



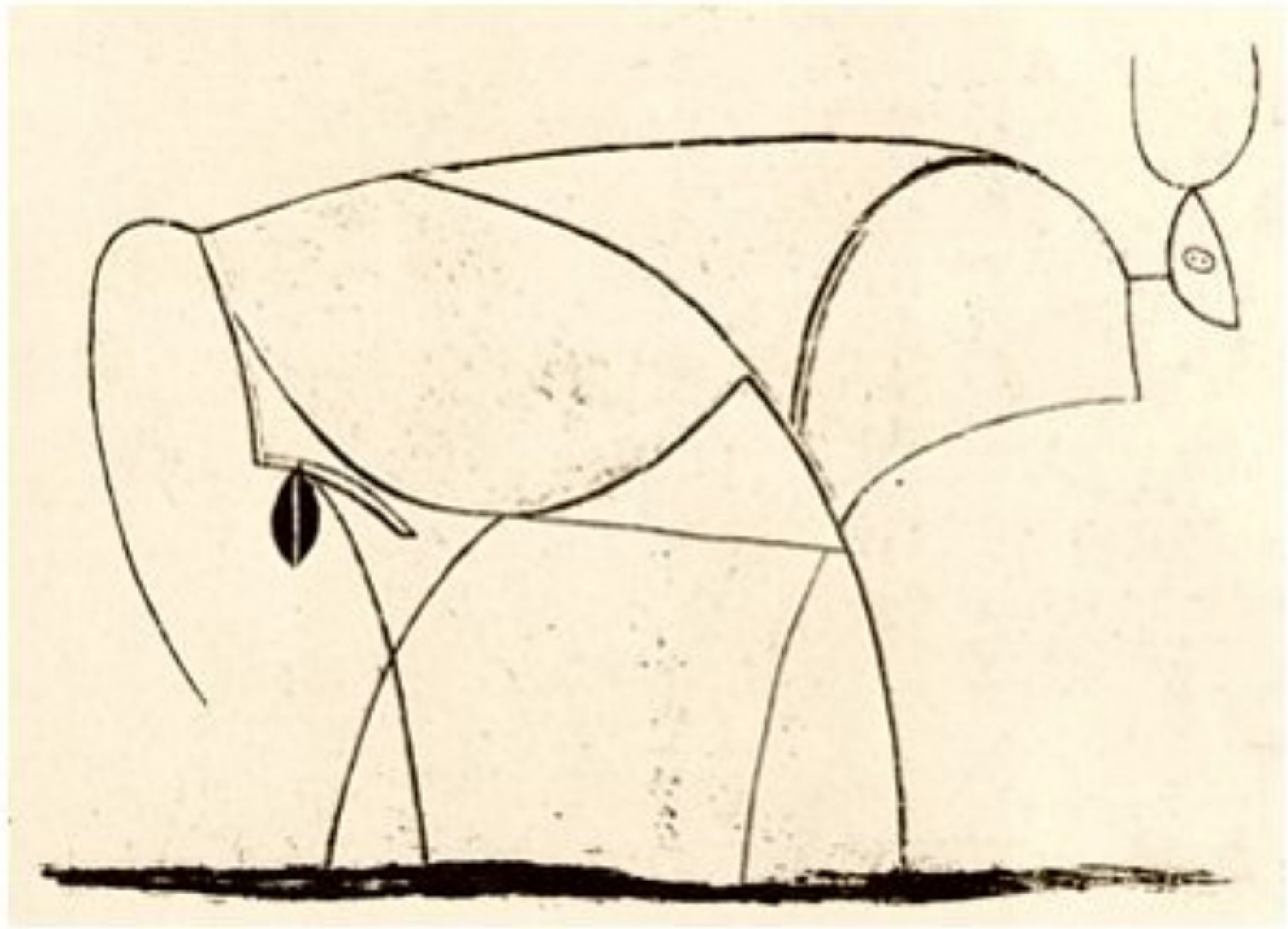
January 2, 1946

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



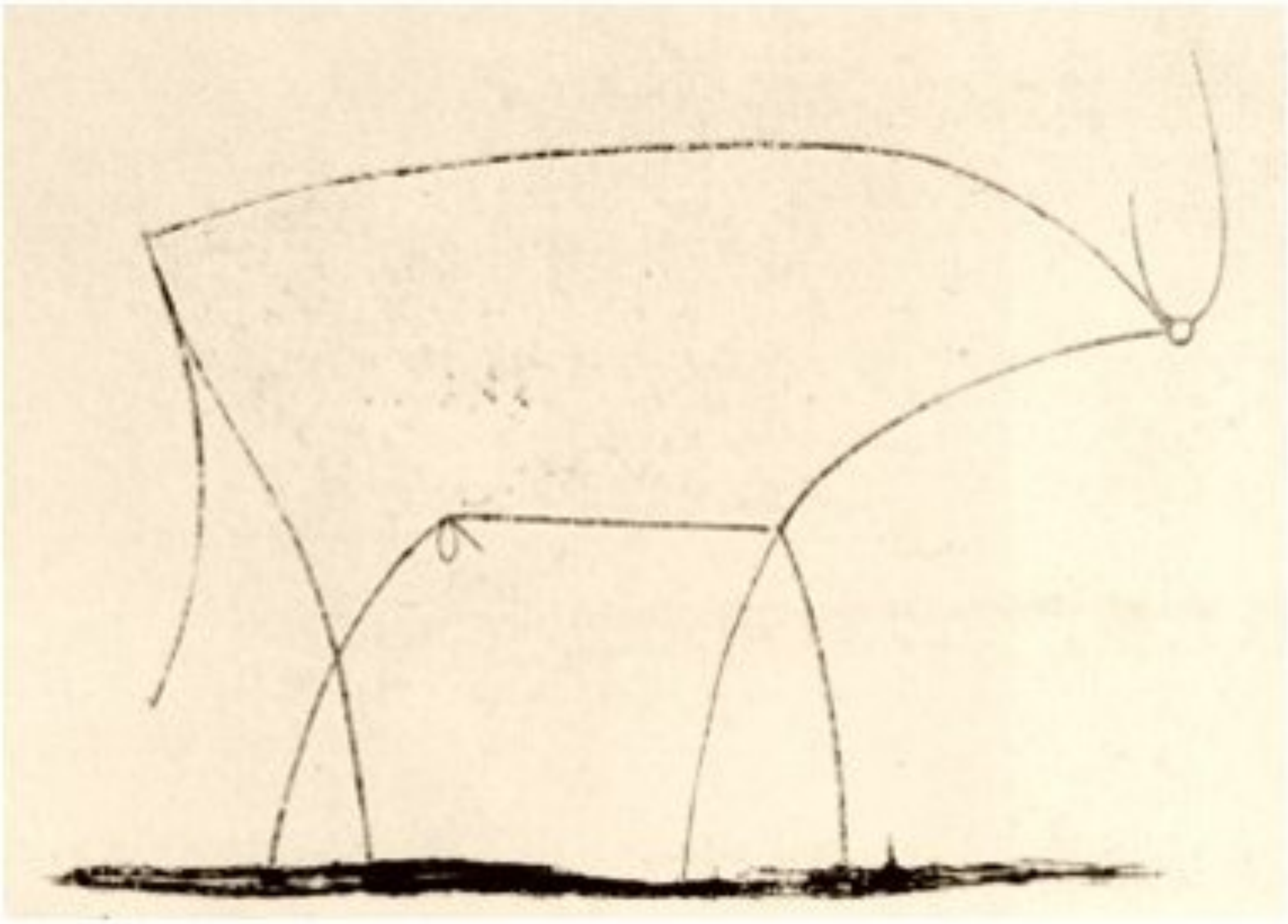
January 5, 1946

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



January 10, 1946

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



January 17, 1946

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm

It takes a lot of hard work to make something simple, to truly understand the underlying challenges and come up with elegant solutions. You have to deeply understand the essence of a product in order to be able to get rid of the parts that are not essential.

Steve Jobs

vs.

vs.

The Writing Process

Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.

(Antoine de Saint-Exupéry, L'Avion, Ch. III)

Recursion in Technical Writing

Recursive Structure

Paper ***title***

Paper ***abstract***

Section ***title***

1st paragraph: section ***abstract***

Paragraph: topic sentence (***abstract***) + body

Paragraph: topic sentence (***abstract***) + body

…

Section ***conclusion***

Section ***title***

…

Paper ***conclusion***

Abstract

Deadlock immunity is a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of that and similar deadlocks. We describe a technique that enables programs to automatically gain such immunity without assistance from programmers or users. We implemented the technique for both Java and POSIX threads and evaluated it with several real systems, including MySQL, JBoss, SQLite, Apache ActiveMQ, Limewire, and Java JDK. The results demonstrate effectiveness against real deadlock bugs, while incurring modest performance overhead and scaling to 1024 threads. We therefore conclude that deadlock immunity offers programmers and users an attractive tool for coping with elusive deadlocks.

1 Introduction

Writing concurrent software is one of the most challenging endeavors faced by software engineers, because it requires careful reasoning about complex interactions between concurrently running threads. Many programmers consider concurrency bugs to be some of the most insidious and, not surprisingly, a large number of bugs are related to concurrency [16].

The simplest mechanism used for synchronizing concurrent accesses to shared data is the mutex lock. When threads do not coordinate correctly in their use of locks, deadlock can ensue—a situation whereby a group of threads cannot make forward progress, because each one is waiting to acquire a lock held by another thread in that group. Deadlock immunity helps develop resistance against such deadlocks.

Avoiding the introduction of deadlock bugs during development is challenging. Large software systems are developed by multiple teams totaling hundreds to thousands of programmers, which makes it hard to maintain the coding discipline needed to avoid deadlock bugs. Testing, although helpful, is not a panacea, because exercising all possible execution paths and thread interleavings is still infeasible in practice for all but toy programs.

Even deadlock-free code is not guaranteed to execute free of deadlocks once deployed in the field. Dependencies on deadlock-prone third party libraries or runtimes can deadlock programs that are otherwise correct. Upgrading these libraries or runtimes can introduce new executions that were not covered by prior testing. Furthermore, modern systems accommodate extensions written by third parties, which can introduce new deadlocks into the target systems (e.g., Web browser plugins, enterprise Java beans).

Debugging deadlocks is hard—merely seeing a deadlock happen does not mean the bug is easy to fix. Deadlocks often require complex sequences of low-probability events to manifest (e.g., timing or workload dependencies, presence or absence of debug code, compiler optimization options), making them hard to reproduce and diagnose. Sometimes deadlocks are too costly to fix, as they entail drastic redesign. Patches are error-prone: many concurrency bug fixes either introduce new bugs or, instead of fixing the underlying bug, merely decrease the probability of occurrence [16].

We expect the deadlock challenge to persist and likely become worse over time: On the one hand, software systems continue getting larger and more complex. On the other hand, owing to the advent of multi-core architectures and other forms of parallel hardware, new applications are written using more threads, while existing applications achieve higher degrees of runtime concurrency. There exist proposals for making concurrent programming easier, such as transactional memory [8], but issues surrounding I/O and long-running operations make it difficult to provide atomicity transparently.

In this paper, we introduce the notion of deadlock immunity—a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of similar deadlocks. We describe Dimmunix, a tool for developing deadlock immunity with no assistance from programmers or users. The first time a deadlock pattern manifests, Dimmunix automatically captures its signature and subsequently avoids entering the same pattern. Signatures can be proactively distributed to immunize users who have not yet encountered that deadlock. Dimmunix can be used by customers to defend against deadlocks while waiting for a vendor patch, and by software vendors as a safety net.

In the rest of the paper we survey related work (§2), give an overview of our system (§3-§4), present details of our technique (§5), describe three Dimmunix implementations (§6), evaluate them (§7), discuss how Dimmunix can be used in practice (§8), and conclude (§9).

Abstract

Deadlock immunity is a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of that and similar deadlocks. We describe a technique that enables programs to automatically gain such immunity without assistance from programmers or users. We implemented the technique for both Java and POSIX threads and evaluated it with several real systems, including MySQL, JBoss, SQLite, Apache ActiveMQ, Limewire, and Java JDK. The results demonstrate effectiveness against real deadlock bugs, while incurring modest performance overhead and scaling to 1024 threads. We therefore conclude that deadlock immunity offers programmers and users an attractive tool for coping with elusive deadlocks.

1 Introduction

Writing concurrent software is one of the most challenging endeavors faced by software engineers, because it requires careful reasoning about complex interactions between concurrently running threads. Many programmers consider concurrency bugs to be some of the most insidious and, not surprisingly, a large number of bugs are related to concurrency [16].

The simplest mechanism used for synchronizing concurrent accesses to shared data is the mutex lock. When threads do not coordinate correctly in their use of locks, deadlock can ensue—a situation whereby a group of threads cannot make forward progress, because each one is waiting to acquire a lock held by another thread in that group. Deadlock immunity helps develop resistance against such deadlocks.

Avoiding the introduction of deadlock bugs during development is challenging. Large software systems are developed by multiple teams totaling hundreds to thousands of programmers, which makes it hard to maintain the coding discipline needed to avoid deadlock bugs. Testing, although helpful, is not a panacea, because exercising all possible execution paths and thread interleavings is still infeasible in practice for all but toy programs.

Even deadlock-free code is not guaranteed to execute free of deadlocks once deployed in the field. Dependencies on deadlock-prone third party libraries or runtimes can deadlock programs that are otherwise correct. Upgrading these libraries or runtimes can introduce new executions that were not covered by prior testing. Furthermore, modern systems accommodate extensions written by third parties, which can introduce new deadlocks into the target systems (e.g., Web browser plugins, enterprise Java beans).

Debugging deadlocks is hard—merely seeing a deadlock happen does not mean the bug is easy to fix. Deadlocks often require complex sequences of low-probability events to manifest (e.g., timing or workload dependencies, presence or absence of debug code, compiler optimization options), making them hard to reproduce and diagnose. Sometimes deadlocks are too costly to fix, as they entail drastic redesign. Patches are error-prone: many concurrency bug fixes either introduce new bugs or, instead of fixing the underlying bug, merely decrease the probability of occurrence [16].

We expect the deadlock challenge to persist and likely become worse over time: On the one hand, software systems continue getting larger and more complex. On the other hand, owing to the advent of multi-core architectures and other forms of parallel hardware, new applications are written using more threads, while existing applications achieve higher degrees of runtime concurrency. There exist proposals for making concurrent programming easier, such as transactional memory [8], but issues surrounding I/O and long-running operations make it difficult to provide atomicity transparently.

In this paper, we introduce the notion of deadlock immunity—a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of similar deadlocks. We describe Dimmunix, a tool for developing deadlock immunity with no assistance from programmers or users. The first time a deadlock pattern manifests, Dimmunix automatically captures its signature and subsequently avoids entering the same pattern. Signatures can be proactively distributed to immunize users who have not yet encountered that deadlock. Dimmunix can be used by customers to defend against deadlocks while waiting for a vendor patch, and by software vendors as a safety net.

In the rest of the paper we survey related work (§2), give an overview of our system (§3-§4), present details of our technique (§5), describe three Dimmunix implementations (§6), evaluate them (§7), discuss how Dimmunix can be used in practice (§8), and conclude (§9).

Writing concurrent software is one of the most challenging endeavors faced by software engineers, because it requires careful reasoning about complex interactions between concurrently running threads.

The simplest mechanism used for synchronizing concurrent accesses to shared data is the mutex lock.

Avoiding the introduction of deadlock bugs during development is challenging.

Even deadlock-free code is not guaranteed to execute free of deadlocks once deployed in the field.

Debugging deadlocks is hard—merely seeing a deadlock happen does not mean the bug is easy to fix.

We expect the deadlock challenge to persist and likely become worse over time

In this paper, we introduce the notion of deadlock immunity—a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of similar deadlocks.

Abstract

Deadlock immunity is a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of that and similar deadlocks. We describe a technique that enables programs to automatically gain such immunity without assistance from programmers or users. We implemented the technique for both Java and POSIX threads and evaluated it with several real systems, including MySQL, JBoss, SQLite, Apache ActiveMQ, Limewire, and Java JDK. The results demonstrate effectiveness against real deadlock bugs, while incurring modest performance overhead and scaling to 1024 threads. We therefore conclude that deadlock immunity offers programmers and users an attractive tool for coping with elusive deadlocks.

1 Introduction

Writing concurrent software is one of the most challenging endeavors faced by software engineers, because it requires careful reasoning about complex interactions between concurrently running threads. Many programmers consider concurrency bugs to be some of the most insidious and, not surprisingly, a large number of bugs are related to concurrency [16].

The simplest mechanism used for synchronizing concurrent accesses to shared data is the mutex lock. When threads do not coordinate correctly in their use of locks, deadlock can ensue—a situation whereby a group of threads cannot make forward progress, because each one is waiting to acquire a lock held by another thread in that group. Deadlock immunity helps develop resistance against such deadlocks.

Avoiding the introduction of deadlock bugs during development is challenging. Large software systems are developed by multiple teams totaling hundreds to thousands of programmers, which makes it hard to maintain the coding discipline needed to avoid deadlock bugs. Testing, although helpful, is not a panacea, because exercising all possible execution paths and thread interleavings is still infeasible in practice for all but toy programs.

Even deadlock-free code is not guaranteed to execute free of deadlocks once deployed in the field. Dependencies on deadlock-prone third party libraries or runtimes can deadlock programs that are otherwise correct. Upgrading these libraries or runtimes can introduce new executions that were not covered by prior testing. Furthermore, modern systems accommodate extensions written by third parties, which can introduce new deadlocks into the target systems (e.g., Web browser plugins, enterprise Java beans).

Debugging deadlocks is hard—merely seeing a deadlock happen does not mean the bug is easy to fix. Deadlocks often require complex sequences of low-probability events to manifest (e.g., timing or workload dependencies, presence or absence of debug code, compiler optimization options), making them hard to reproduce and diagnose. Sometimes deadlocks are too costly to fix, as they entail drastic redesign. Patches are error-prone: many concurrency bug fixes either introduce new bugs or, instead of fixing the underlying bug, merely decrease the probability of occurrence [16].

We expect the deadlock challenge to persist and likely become worse over time: On the one hand, software systems continue getting larger and more complex. On the other hand, owing to the advent of multi-core architectures and other forms of parallel hardware, new applications are written using more threads, while existing applications achieve higher degrees of runtime concurrency. There exist proposals for making concurrent programming easier, such as transactional memory [8], but issues surrounding I/O and long-running operations make it difficult to provide atomicity transparently.

In this paper, we introduce the notion of deadlock immunity—a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of similar deadlocks. We describe Dimmunix, a tool for developing deadlock immunity with no assistance from programmers or users. The first time a deadlock pattern manifests, Dimmunix automatically captures its signature and subsequently avoids entering the same pattern. Signatures can be proactively distributed to immunize users who have not yet encountered that deadlock. Dimmunix can be used by customers to defend against deadlocks while waiting for a vendor patch, and by software vendors as a safety net.

In the rest of the paper we survey related work (§2), give an overview of our system (§3-§4), present details of our technique (§5), describe three Dimmunix implementations (§6), evaluate them (§7), discuss how Dimmunix can be used in practice (§8), and conclude (§9).

deadlock can ensue—a situation whereby a group of threads cannot make forward progress, because each one is waiting to acquire a lock held by another thread in that group. Deadlock immunity helps develop resistance against such deadlocks.

Avoiding the introduction of deadlock bugs during development is challenging. Large software systems are developed by multiple teams totaling hundreds to thousands of programmers, which makes it hard to maintain the coding discipline needed to avoid deadlock bugs. Testing, although helpful, is not a panacea, because exercising all possible execution paths and thread interleavings is still infeasible in practice for all but toy programs.

Even deadlock-free code is not guaranteed to execute free of deadlocks once deployed in the field. Dependencies on deadlock-prone third party libraries or runtimes can deadlock programs that are otherwise correct. Upgrading these libraries or runtimes can introduce

immunity—
by a given d
currences o
a tool for d
tance from
lock patter
tures its sig
same patter
to immuniz
deadlock. I
fend agains
and by soft

In the res
give an over
our techniq
tations (§6)
can be used

deadlock can ensue—a situation whereby a group of threads cannot make forward progress, because each one is waiting to acquire a lock held by another thread in that group. Deadlock immunity helps develop resistance against such deadlocks.

Avoiding the introduction of deadlock bugs during development is challenging. Large software systems are developed by multiple teams totaling hundreds to thousands of programmers, which makes it hard to maintain the coding discipline needed to avoid deadlock bugs. Testing, although helpful, is not a panacea, because exercising all possible execution paths and thread interleavings is still infeasible in practice for all but toy programs.

Even deadlock-free code is not guaranteed to execute free of deadlocks once deployed in the field. Dependencies on deadlock-prone third party libraries or runtimes can deadlock programs that are otherwise correct. Upgrading these libraries or runtimes can introduce

immunity—
by a given d
currences o
a tool for d
tance from
lock patter
tures its sig
same patter
to immuniz
deadlock. I
fend agains
and by soft

In the res
give an over
our techniqu
tations (§6)
can be used

In the developing world 1.1 billion people still lack access to safe drinking water, 2.6 billion do not have access to adequate sanitation services, and more than 1.6 million deaths each year are traced to waterborne diseases (mostly in children under five). All too often in developing countries, water is costly or inaccessible to the poorest in society, while the wealthy have it piped into their homes. In addition, because of the infrastructure that is used to control water, whole seas are being lost, rivers are running dry, millions of people have been displaced to make room for reservoirs, groundwater aquifers are being pumped down, and disputes over water have raised tensions from local to international levels. Fresh water is a limiting resource in many parts of the world and is certain to become even more so as the 21st century unfolds.

(Wright and Boorse, Environmental Science, p. 247)

In the developing world 1.1 billion people still lack access to safe drinking water, 2.6 billion do not have access to adequate sanitation services, and more than 1.6 million deaths each year are traced to waterborne diseases (mostly in children under five). All too often in developing countries, water is costly or inaccessible to the poorest in society, while the wealthy have it piped into their homes. In addition, because of the infrastructure that is used to control water, whole seas are being lost, rivers are running dry, millions of people have been displaced to make room for reservoirs, groundwater aquifers are being pumped down, and disputes over water have raised tensions from local to international levels. Fresh water is a limiting resource in many parts of the world and is certain to become even more so as the 21st century unfolds.

(Wright and Boorse, Environmental Science, p. 247)

In colonial days, huge flocks of snowy egrets inhabited the coastal wetlands and marshes of the southeastern United States. In the 1800s, when fashion dictated fancy hats adorned with feathers, egrets and other birds were hunted for their plumage. By the late 1800s, egrets were almost extinct. In 1886, the newly formed National Audubon Society began a press campaign to shame “feather wearers” and end the practice. The campaign caught on, and gradually, attitudes changed; new laws followed. Government policies that protect animals from overharvesting are essential to keep species from the brink of extinction. Even when cultural standards change due to the efforts of individual groups (such as the National Audubon Society), laws and policy measures must follow to ensure that endangered populations remain protected. Since the 1800s, several important laws have been passed to protect a wide variety of species.

(Wright and Boorse, Environmental Science: Toward a Sustainable Future, p. 150)

In colonial days, huge flocks of snowy egrets inhabited the coastal wetlands and marshes of the southeastern United States. In the 1800s, when fashion dictated fancy hats adorned with feathers, egrets and other birds were hunted for their plumage. By the late 1800s, egrets were almost extinct. In 1886, the newly formed National Audubon Society began a press campaign to shame“feather wearers”and end the practice.The campaign caught on, and gradually, attitudes changed; new laws followed. Government policies that protect animals from overharvesting are essential to keep species from the brink of extinction. Even when cultural standards change due to the efforts of individual groups (such as the National Audubon Society), laws and policy measures must follow to ensure that endangered populations remain protected. Since the 1800s, several important laws have been passed to protect a wide variety of species.

(Wright and Boorse, Environmental Science: Toward a Sustainable Future, p. 150)

The National Cancer Institute (NCI) has taken a brute-force approach to screening species for cancer-suppressing chemicals. NCI scientists receive frozen samples of organisms from around the world, chop them up, and separate them into a number of extracts, each probably containing hundreds of components. These extracts are tested against up to 60 different types of cancer cells for their efficacy in stopping or slowing growth of the cancer. Promising extracts are then further analyzed to determine their chemical nature, and chemicals in the extract are tested singly to find the effective compound.This approach is often referred to as the “grind ’em and find ’em” strategy.

(Belk and Maier, Biology, p. 334)

The National Cancer Institute (NCI) has taken a brute-force approach to screening species for cancer-suppressing chemicals. NCI scientists receive frozen samples of organisms from around the world, chop them up, and separate them into a number of extracts, each probably containing hundreds of components. These extracts are tested against up to 60 different types of cancer cells for their efficacy in stopping or slowing growth of the cancer. Promising extracts are then further analyzed to determine their chemical nature, and chemicals in the extract are tested singly to find the effective compound. This approach is often referred to as the “grind ’em and find ’em” strategy.

(Belk and Maier, Biology, p. 334)

Writing a good topic sentence

- Structure as topic + controlling idea
  - *topic = what the paragraph is about*
  - *viewpoint = the direction the paragraph will take*

> RCU locks are a good fit for three reasons.

- Strike a balance between general and specific

> The algorithm is mostly based on prior work except for one novel detail.

- Be clear

~~Almost 90% of Americans own cell phones [18].~~

Almost 90% of Americans own cell phones [18], leading the the wide spread of SMS-based attacks.

~~This paragraph shows how to un-marshall RPC arguments.~~

Un-marshalling RPC arguments requires three steps.

Recursive Structure

Paper title

- Paper abstract
- Section title
  - 1st paragraph: section abstract
  - Paragraph: topic sentence + body
  - Paragraph: topic sentence + body
  - …
  - Section conclusion
- Section title
- …
- Paper conclusion

Writing Tips & Tricks

Keep Opinions to Yourself

Statement =

Fact ← citations

Result ← proof, measurement

Opinion ← ???

Keep Opinions to Yourself

Fact ← citations

Statement = Result ← proof, measurement

Opinion ← ???

Many researchers have considered this an important problem, but few solutions exist.

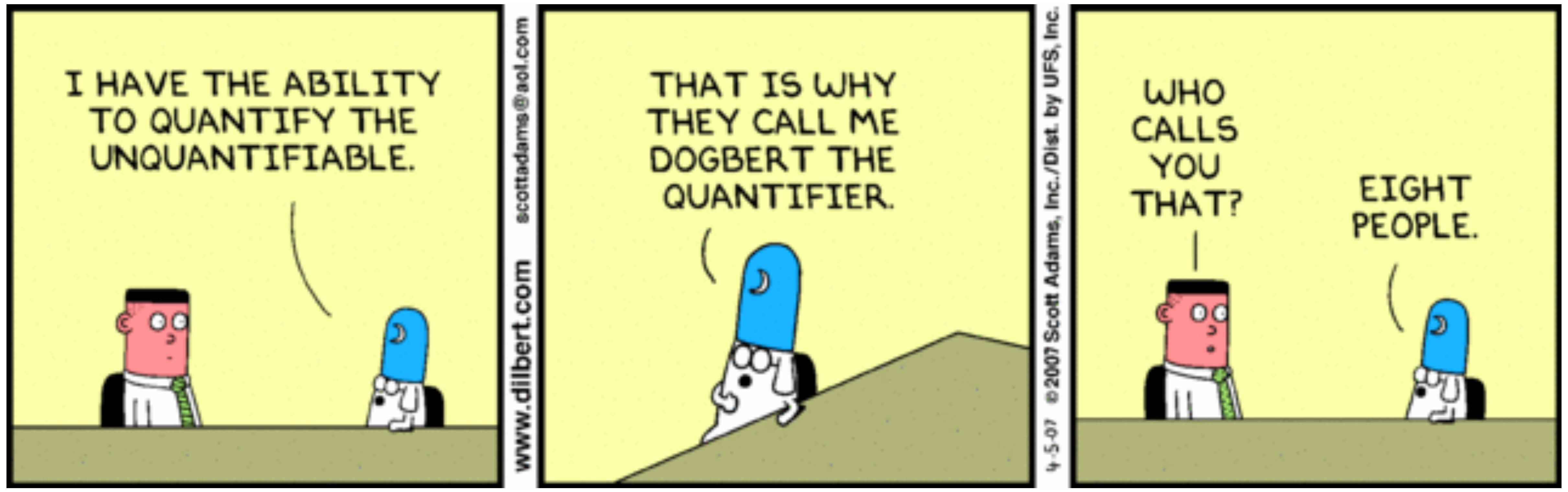
Maximum Clarity ⇔ No Vagueness

- Scientific writing instead of poetry
  - *precise, crystal clear*
  - *arguments are objective, logical, not subject to interpretation*
- Written text vs. idea in your head
- Text must withstand the scrutiny of a logician
- Consistency terminology

Clarity: Quantify

The performance of our cache becomes tremendously small when the data is accessed in a very adversarial manner.

The hit rate of the CPU cache drops by up to 95% if programs consistently write to the least-recently read memory address.



Clarity: Avoid passive voice

The items are then shown in alphabetical order.

The program then outputs the items to the console in alphabetical order.

The order of the items is alphabetical.

“A bar was walked into by the passive voice.”

Clarity: Avoid reverse anthropomorphism

Then you send the packet to the server

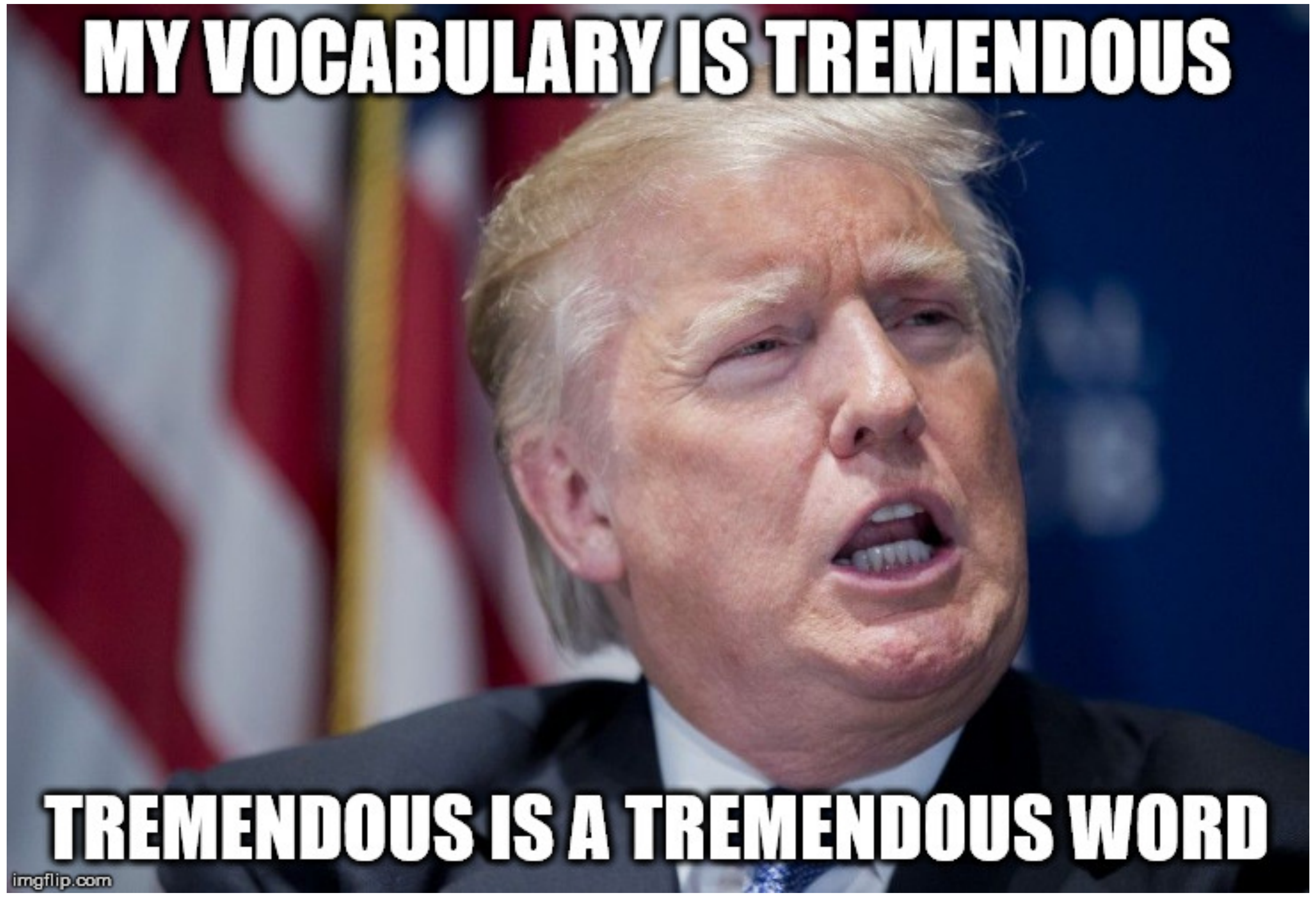
Then the client application sends the packet to the server.

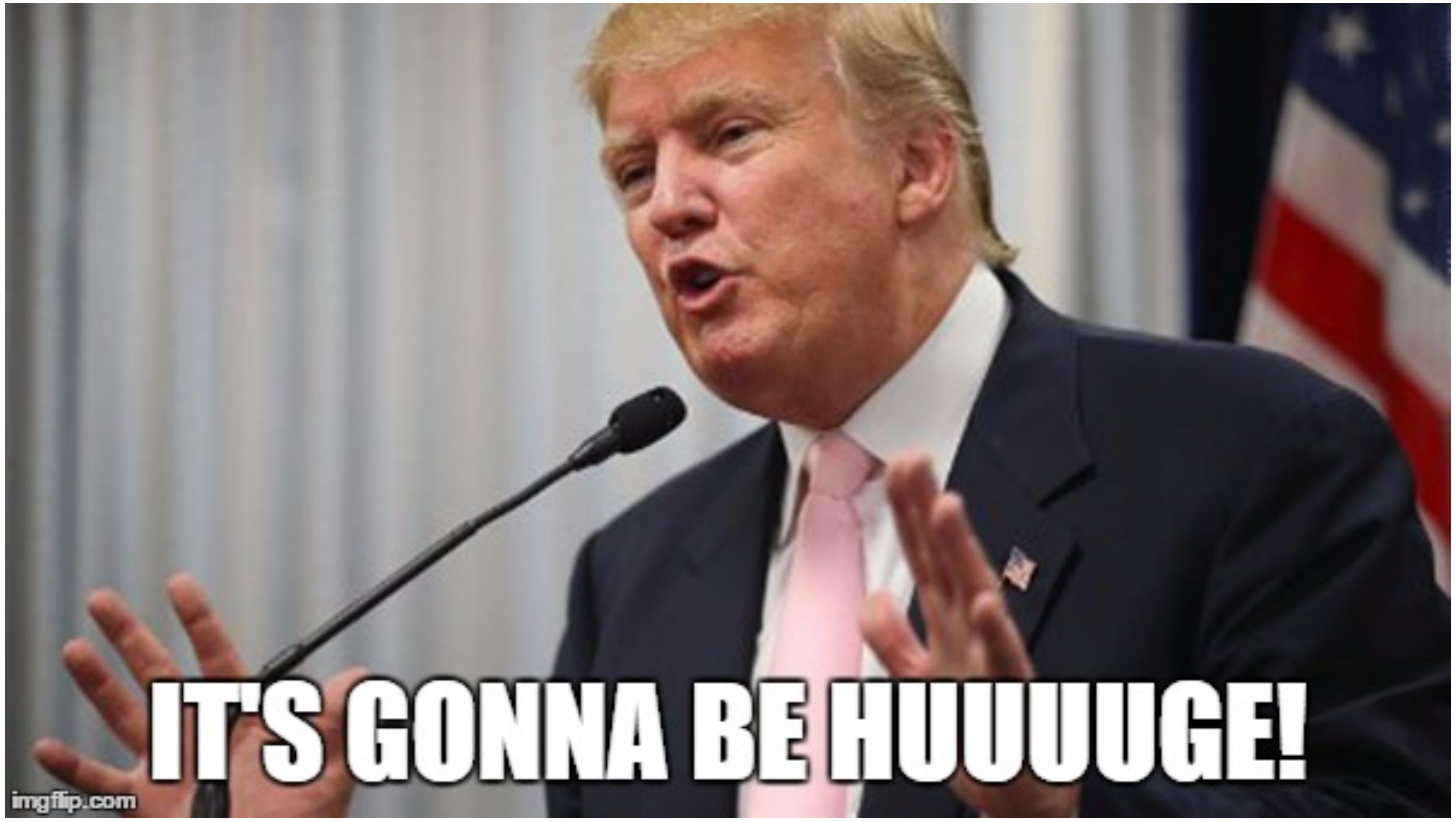
Our file system recovers …

The client's file system recovers…

Clarity: Avoid hyperbolae

We show greatly improved throughput.





Clarity: Avoid hyperbolae

We show greatly improved throughput.

We show a 26% to 77% improvement in throughput.

Abstract

Biologists are leading current research on genome characterization (sequencing, alignment, transcription), providing a huge quantity of raw data about many genome organisms. Extracting knowledge from this raw data is an important process for biologists, using usually data mining approaches. However, it is difficult to deals with these genomic information using actual bioinformatics data mining tools, because data are heterogeneous, huge in quantity and geographically distributed. In this paper, we present a new approach between data mining and virtual reality visualization, called visual data mining. Indeed Virtual Reality becomes ripe, with efficient display devices and intuitive interaction in an immersive context. Moreover, biologists use to work with 3D representation of their molecules, but in a desktop context. We present a software solution, *Genome3DExplorer*, which addresses the problem of genomic data visualization, of scene management and interaction. This solution is based on a well-adapted graphical and interaction paradigm, where local and global topological characteristics of data are easily visible, on the contrary to traditional genomic database browsers, always focused on the zoom and details level.

CR Categories: H.5.1 [Information interfaces and presentation]: Multimedia Information Systems – *Artificial, augmented, and virtual realities.* I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – *Virtual reality.* J.3 [Life and Medical Sciences]: Biology and genetics.

Keywords: Virtual Reality, Immersive Exploration, Human-Computer Interaction, Genomic Data, Graph-based Visualization.

1. Introduction

The last years witnessed a continued growth of the amount of data being stored in biologic databanks. Often the data sets are becoming so huge, that make them difficult to exploit. Extracting knowledge from this raw data is an important process for biologists, using usually data mining approaches. However, it is difficult to deals with this genomic information using actual bioinformatics data mining tools, because data becomes very huge in quantity. For example the capacity of DNA microarray data increased by thousand in several years. Even the best bioinformatics visual data mining tools on this kind of data, such as the innovative and famous hierarchical visual clustering of Eisen et al. [1998] do not achieve to deal with this size increasing. The advances in virtual reality and data visualization have thus creating increasing need for graphical tools and techniques to aid in large genomic data analysis. For example, the limit of the desktop context in the Eisen's solution, leaded Kano et al. [2002] to adapt this paradigm into an immersive context. New solutions were developed in order to deal other kind of huge data, such as huge molecule. *ADN-Viewer* [Gherbi. and Hérisson 2002] exploits the advantages of a virtual context with large display, to deals with huge nucleic molecule, and offers biologists a new representation of their huge DNA sequences, by representing its predicted 3D architecture, according to it textual sequence (list of A, C, G, T) and biophysical model. Sharma et al [2002] proposed *Atomsviewer*, a similar solution in an immersive context, in order to explore billion-atom molecules. However, there are other kinds of genomic information relating to genes or molecules, recorded in structured format within many genomic databanks. *Sequence World* [Rojdestvenski et al. 2000] proposes the first solution in an immersive context, in order to explore this kind of huge factual genomic databanks. Nevertheless, and this solution deals only with annotated gene sequence databanks such as *GenBank*, solution, and does not address the problem of heterogeneity.

As *Sequence Word*, this paper presents a visual mining approach, in an immersive context. However, our solution allows biologists to explore and manage huge and heterogeneous genomic data, not only annotated sequence databanks. Our solution is based on a well-adapted graphical and interaction paradigm for genomic data, where global topological characteristics of data are easily visible, on the contrary to traditional genomic database browsers, always focused on the zoom and details level. First, we present in how we address the problem of the format heterogeneity of this kind of databases, in order to explore them with a common visualization paradigm. We explain then how our software deals with huge genomic data, using a specific data representation, an immersive context and simple scene management. Finally, we present some results and experiments produced by *Genome3DExplorer*, leaded by biologists on various sets of biological data.

Fewer Words, More Examples

I would have written a shorter letter, but I did not have the time.

(Blaise Pascal, Provincial Letters # XVI)

Someone once asked President Woodrow Wilson how long it took him to prepare a speech.

“It depends. If I am to speak ten minutes, I need a week for preparation;
if fifteen minutes, three days;
if half an hour, two days;
if an hour, I am ready now.”

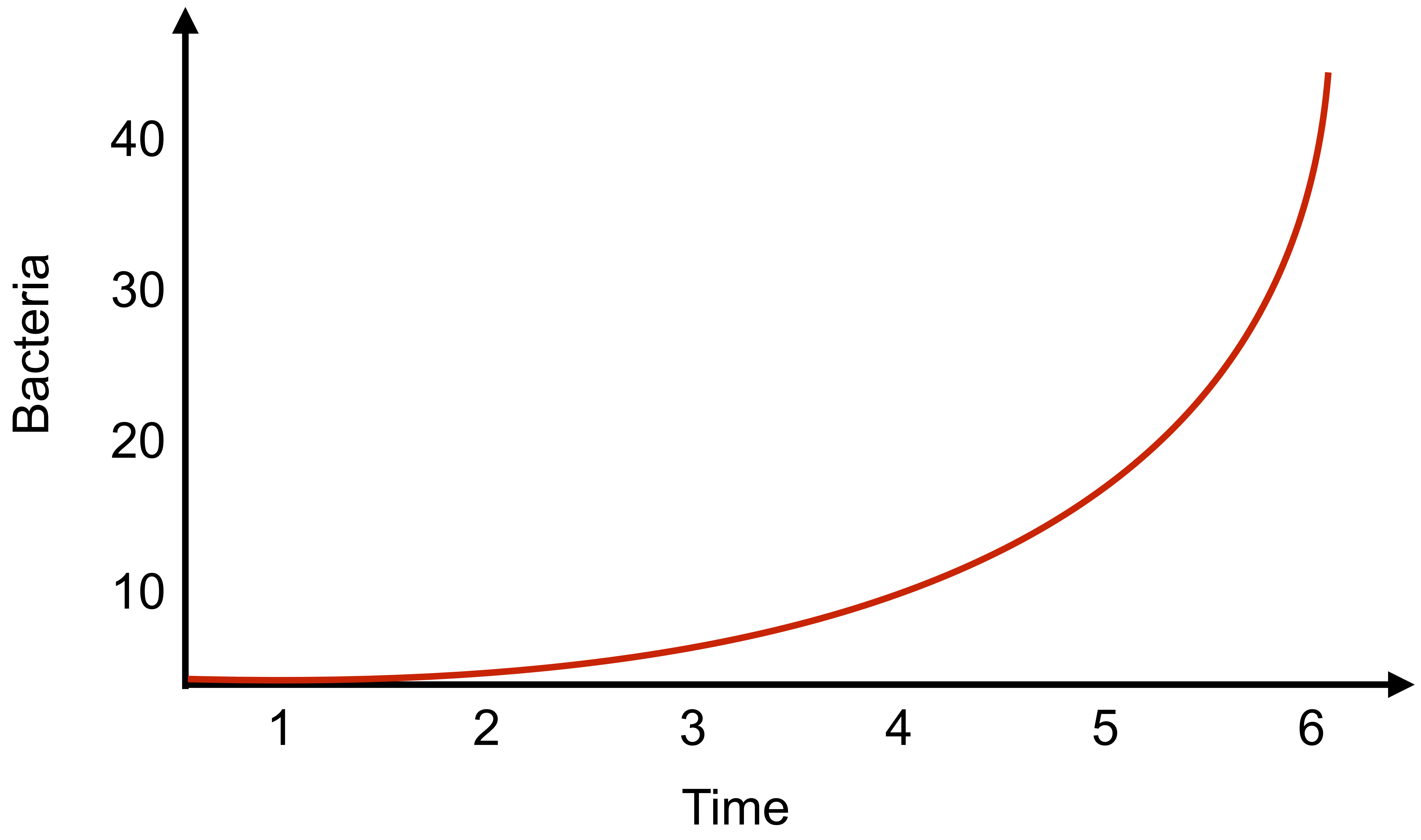
(*Josephus Daniels, The Wilson Era: Years of War and After 1917-1923*)

Making Clear Graphs and Tables

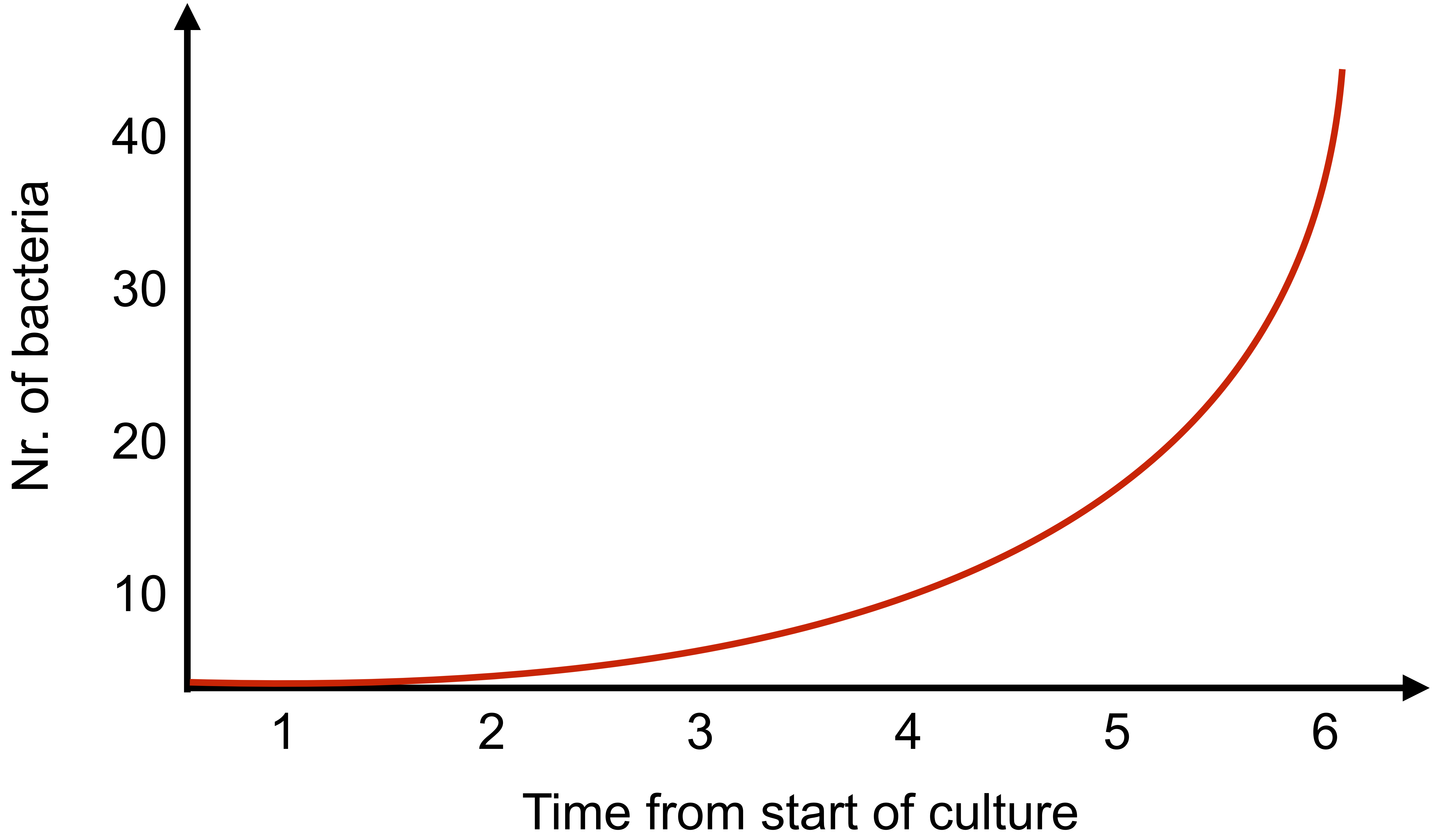
Clear Graphs/Tables

1 graph/table = 1 story

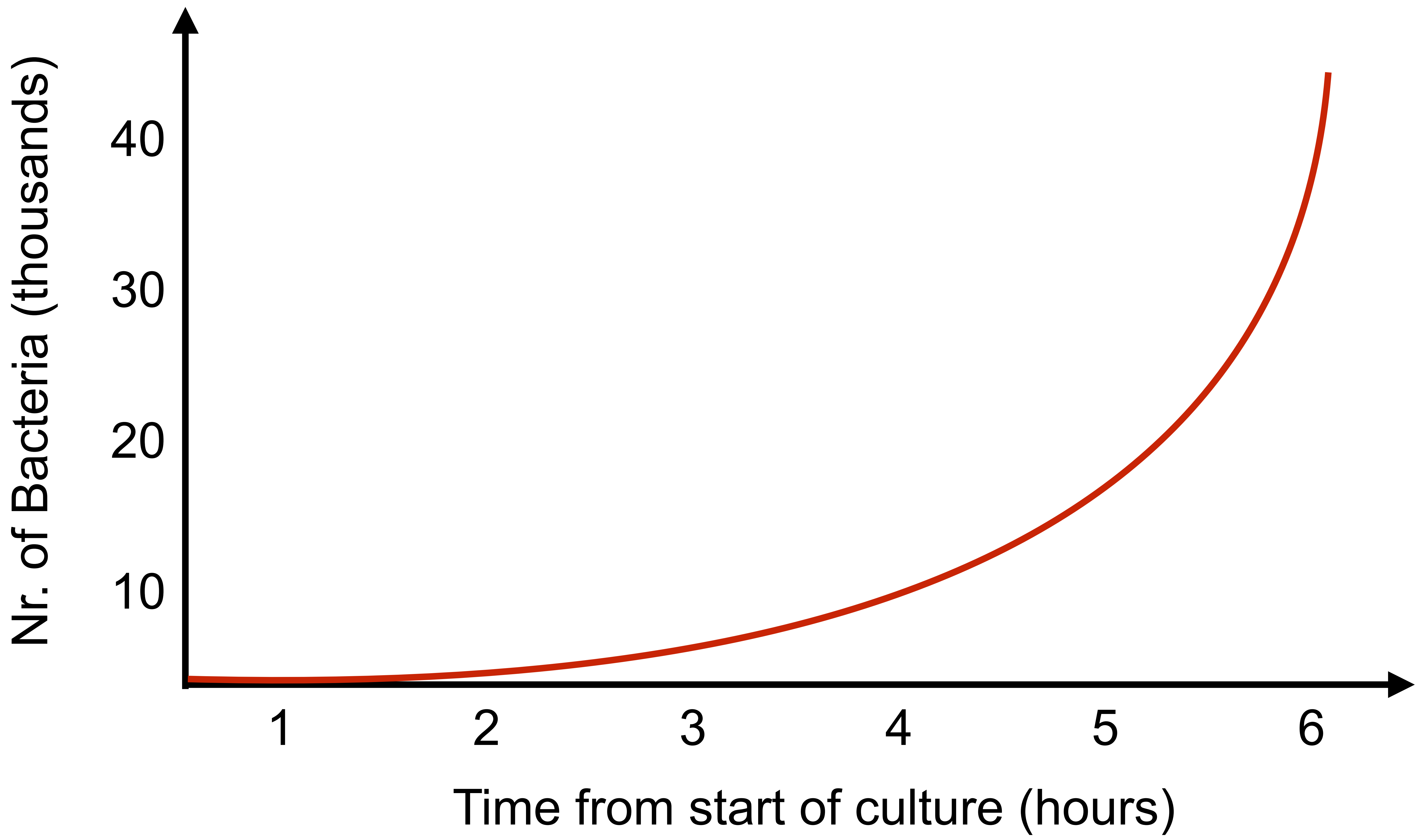
Clear Graphs



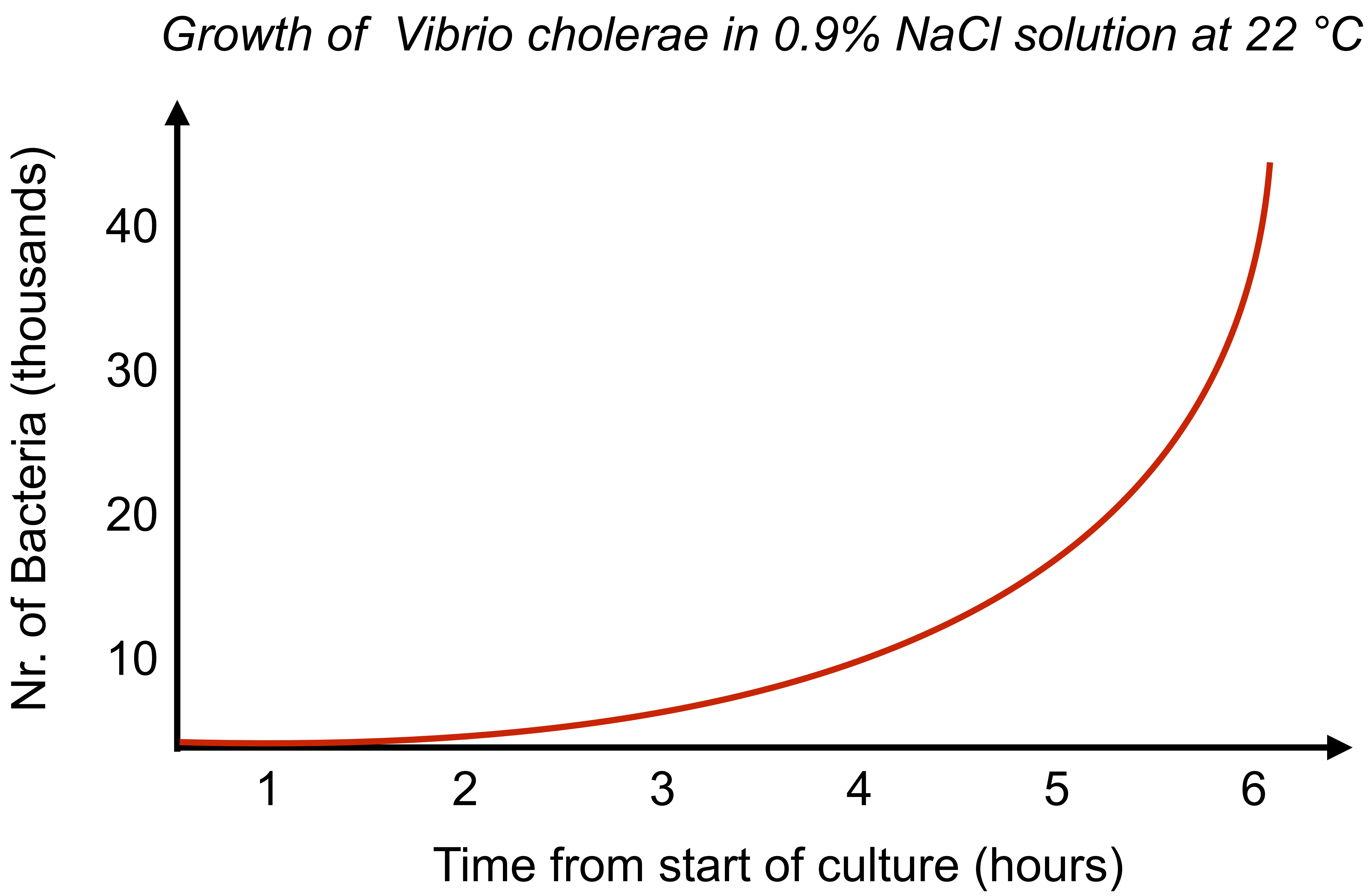
Clear Graphs



Clear Graphs

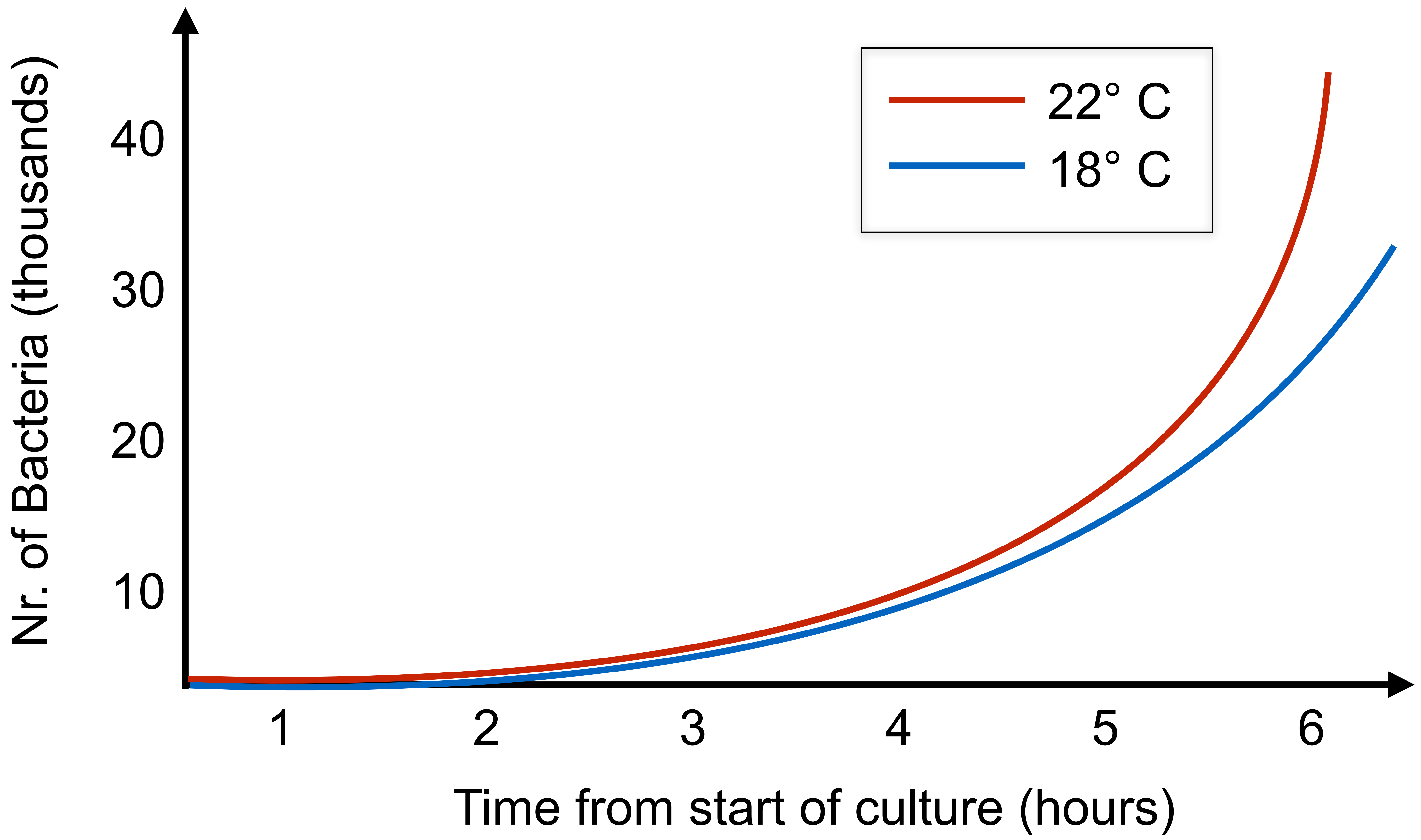


Clear Graphs



Clear Graphs

Growth of Vibrio cholerae in 0.9% NaCl solution at



Stack layer			**Lines of code**		
VigNAT	VigBr	VigLB	969	815	850
VigPol	VigFw		725	754	
libVig			1674		
KLEE-uClibc (libc)			60556		
DPDK			62380		
Ixgbe Driver			24211		
Operating system (NFOS)			1958		

Table 2. Size of each layer in the Vigor stack.

Stack layer			Lines of code		
VigNAT	VigBr	VigLB	969	815	850
VigPol	VigFw		725	754	
libVig			1,674		
KLEE-uClibc (libc)			60,556		
DPDK			62,380		
Ixgbe Driver			24,211		
Operating system (NFOS)			1,958		

Table 2. Size of each layer in the Vigor stack.

Stack layer			Lines of code		
VigNAT	VigBr	VigLB	969	815	850
VigPol	VigFw		725	754	
libVig			1,674		
KLEE-uClibc (libc)			60,556		
DPDK			62,380		
Ixgbe Driver			24,211		
Operating system (NFOS)			1,958		

Table 2. Size of each layer in the Vigor stack.

- Use font faces and sizes consistently

Explain Your Data

The purpose of computing is insight, not numbers.

(Richard Hamming, Numerical Methods for Scientists and Engineers, 1962)

Explain Your Data

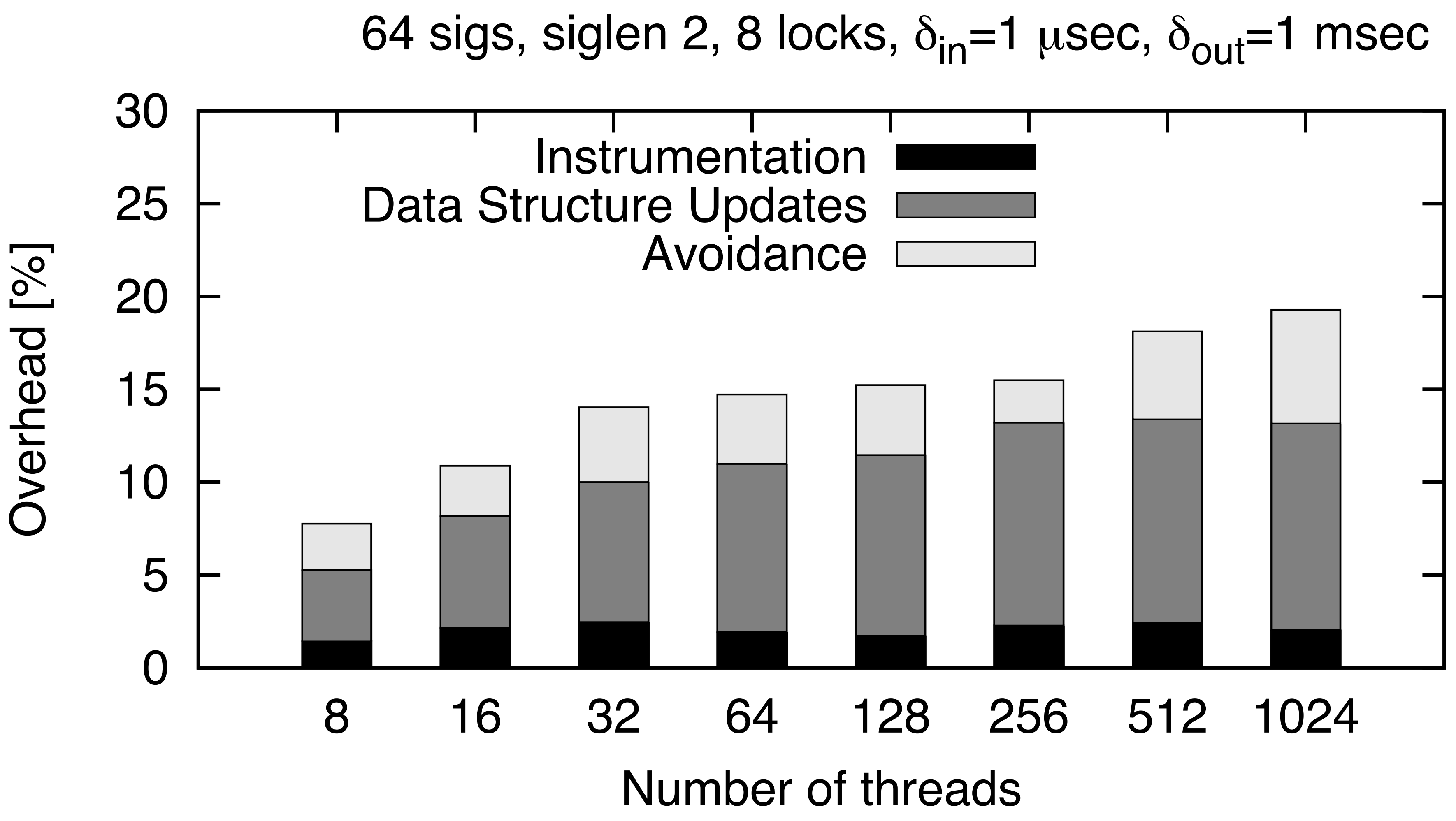


Figure 8: Breakdown of overhead for Java Dimmunix.

The results for Java are shown in Figure 8—the bulk of the overhead is introduced by the data structure lookups and updates.

Use Text to Give Context and Draw Conclusions

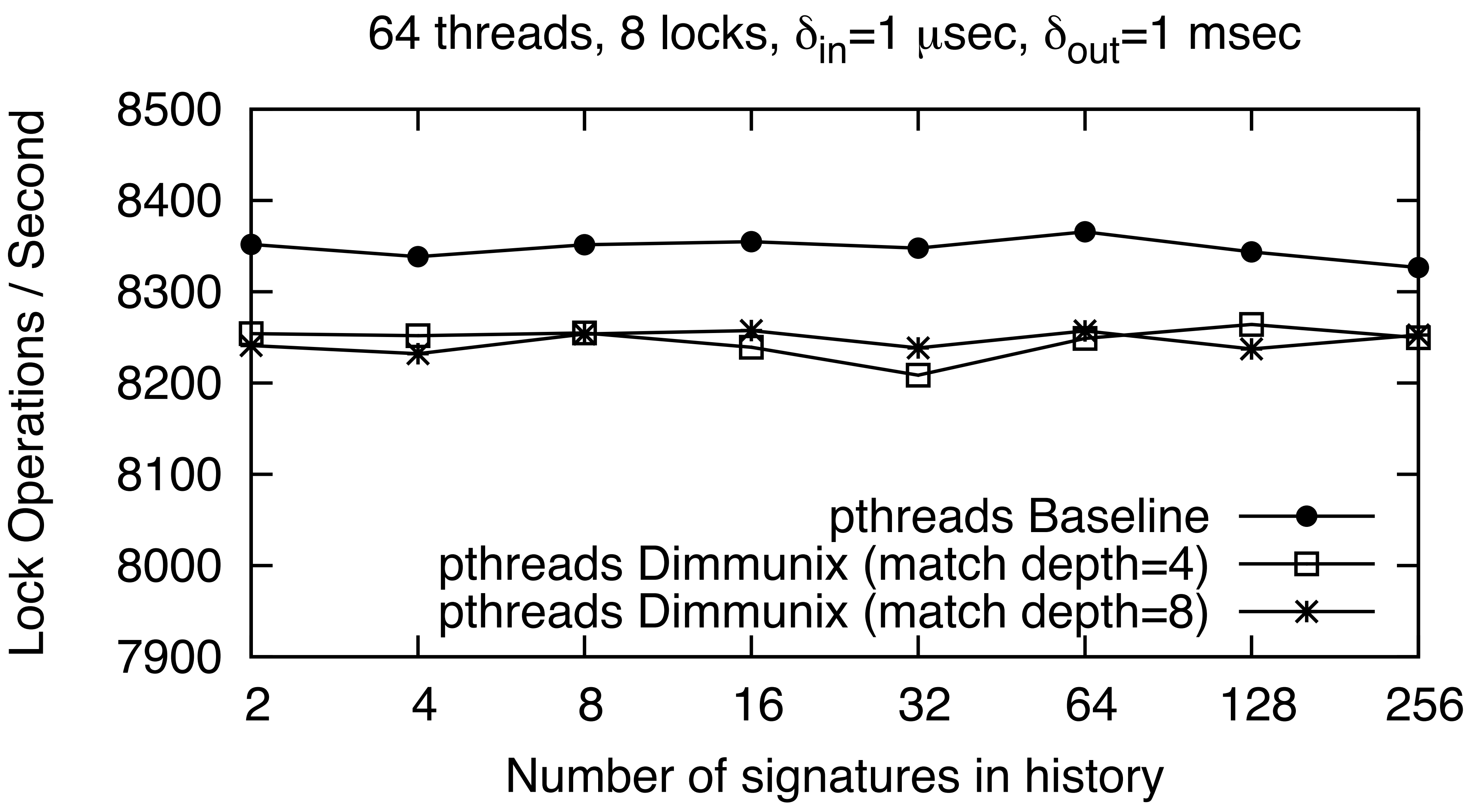


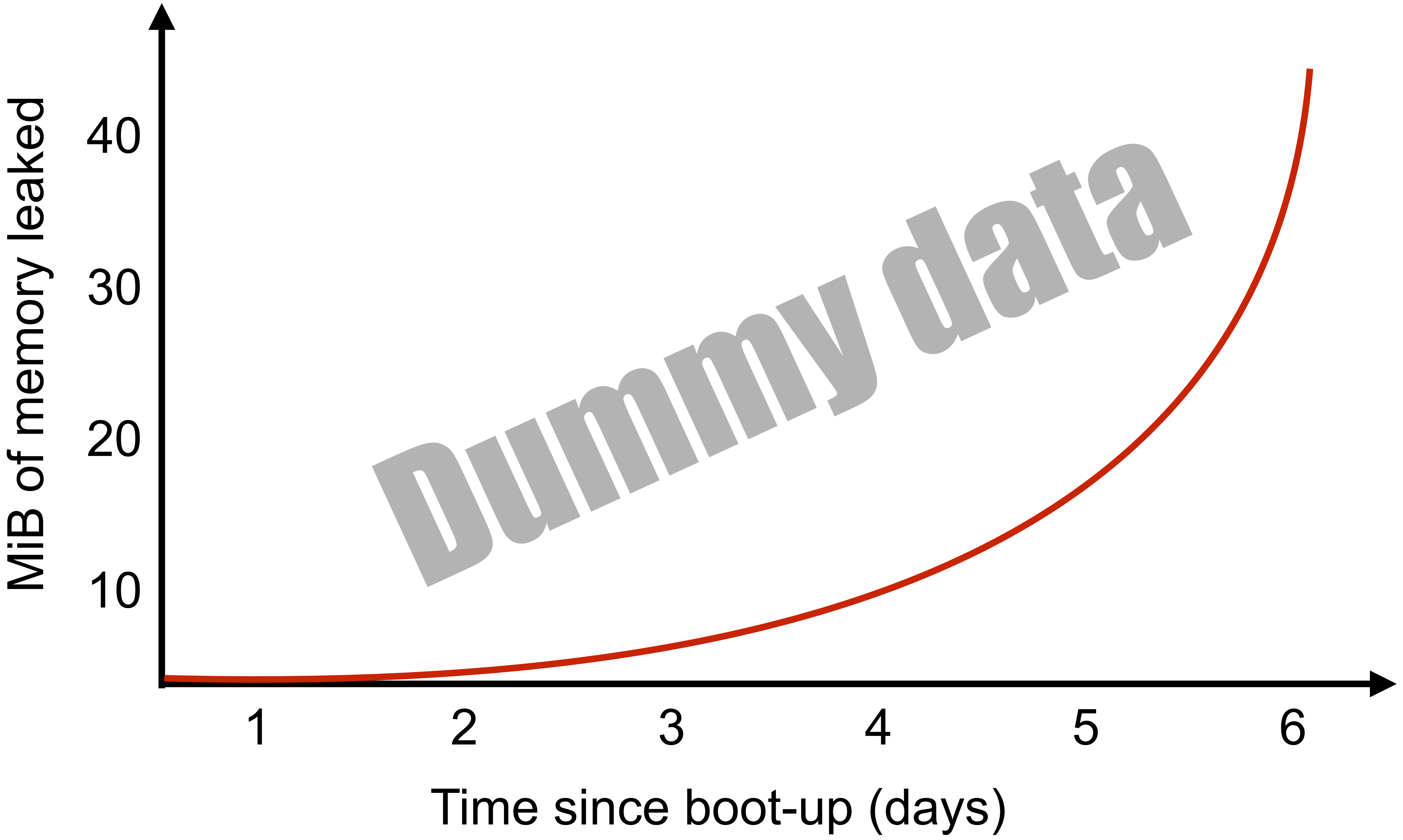
Figure 7: Lock throughput as a function of history size and matching depth for pthreads. Java results are similar.

Impact of history size and matching depth: The performance penalty incurred by matching current executions against signatures from history should increase with the size of the history (i.e., number of signatures) as well as the depth at which signatures are matched with current stacks. Average length of a signature (i.e., average number of threads involved in the captured deadlock) also influences matching time, but the vast majority of deadlocks in practice are limited to two threads [16], so variation with signature size is not that interesting.

In addition to the matching overhead, as more and more deadlocks are discovered in the program, the program must be serialized increasingly more in order to be deadlock-safe (i.e., there are more deadlocks to avoid)—our overhead measurements include both effects.

We show in Figure 7 the performance overhead introduced by varying history size from 2-256 signatures. The overhead introduced by history size and matching depth is relatively constant across this range, which means that searching through history is a negligible component of Dimmunix overhead.

Dummy Graphs



Unfortunately, there is an inherent conflict in the design goals behind these devices: as mobile systems, they should be designed to maximize battery life, but as intelligent devices, they need powerful processors, which consume more energy than those in simpler devices, thus reducing battery life. In spite of continuous advances in semiconductor and battery technologies that allow microprocessors to provide much greater computation per unit of energy and longer total battery life, the fundamental tradeoff between performance and battery life remains critically important.

Recently, significant research and development efforts have been made on *Dynamic Voltage Scaling* (DVS) [2, 4, 7, 8, 12, 19, 21, 22, 23, 24, 25, 26, 28, 30]. DVS tries to address the tradeoff between performance and battery life by taking into account two important characteristics of most current computer systems: (1) the peak computing rate needed is much higher than the average throughput that must be sustained; and (2) the processors are based on CMOS logic. The first characteristic effectively means that high performance is needed only for a small fraction of the time, while for the rest of the time, a low-performance, low-power processor would suffice. We can achieve the low performance by simply lowering the operating frequency of the processor when the full speed is not needed. DVS goes beyond this and scales the operating voltage of the processor along with the frequency. This is possible because static CMOS logic, used in the vast majority of microprocessors today, has a voltage-dependent maximum operating frequency, so when used at a reduced frequency, the processor can operate at a lower supply voltage. Since the energy dissipated per cycle with CMOS circuitry scales quadratically to the supply voltage ($E \propto V^2$) [2], DVS can potentially provide a very large net energy savings through frequency and voltage scaling.

Conclusion

- Technical writing ≠ Lyrical writing
- Write iteratively (the way Picasso drew)
- Clean, recursive structure to ease reader's load
- Avoid opinions, vagueness
- Reduce # of words, increase # of examples
- Clear graphs with explained data

OP1 (Naming)

In the current Internet, when a client wants to access some content, it first contacts DNS to obtain an IP address for a service that serves the desired content; only after this name lookup is complete can the client start communicating with the target service and accessing the target content.

Assuming we can change the Internet architecture, is it possible to remove the need for the client to do a separate name lookup in order to access the target content?

Assume you can change the Internet architecture any way you want, e.g., you can change the TCP/IP stack, the inter-domain routing protocol, the way packet switches and routers operate, etc.

Assume that a client names content using a bit string of bounded length. (For example, a DNS name or a URL is a bit string of bounded length.)