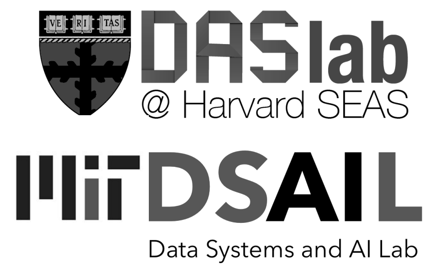
# **LEARNED** AND **SELF-DESIGNING** DATA STRUCTURES



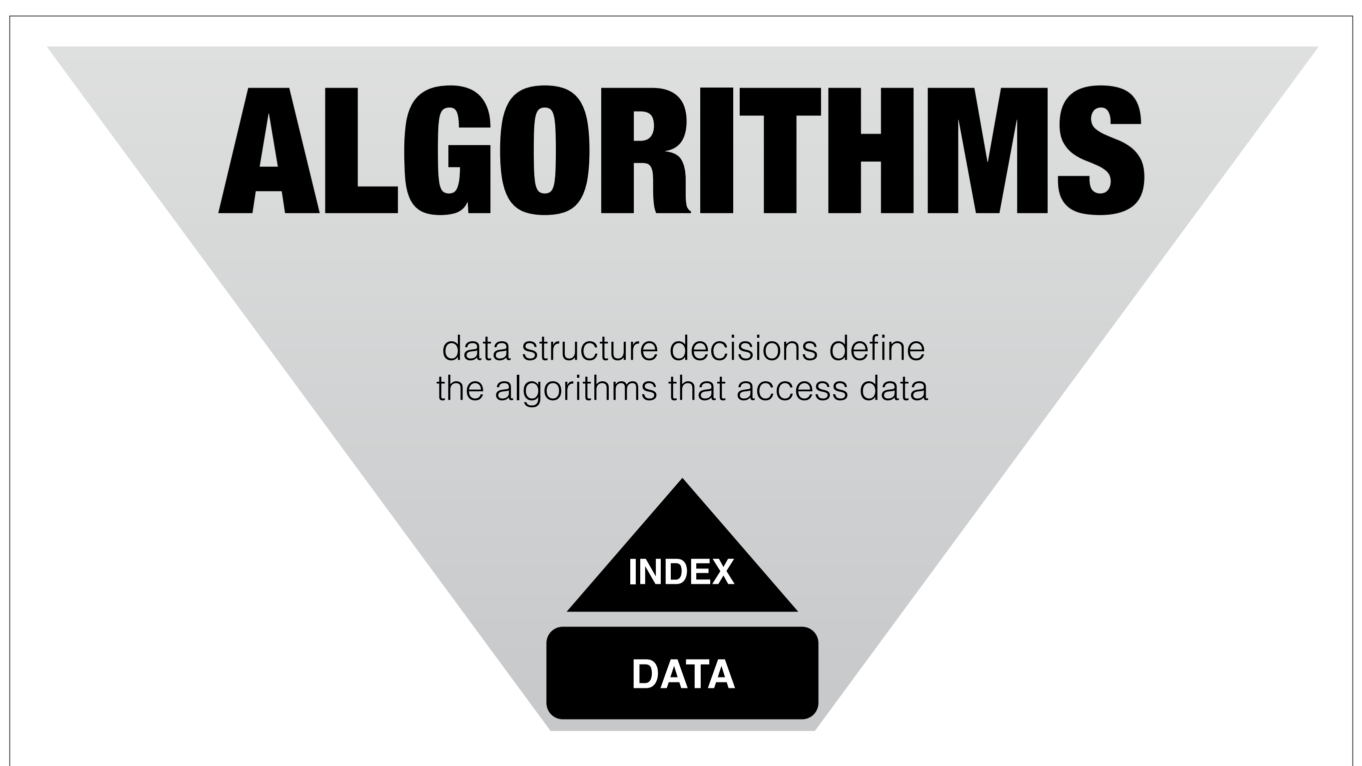

Stratos Idreos & Tim Kraska

# **LEARNED** AND **SELF-DESIGNING** DATA STRUCTURES

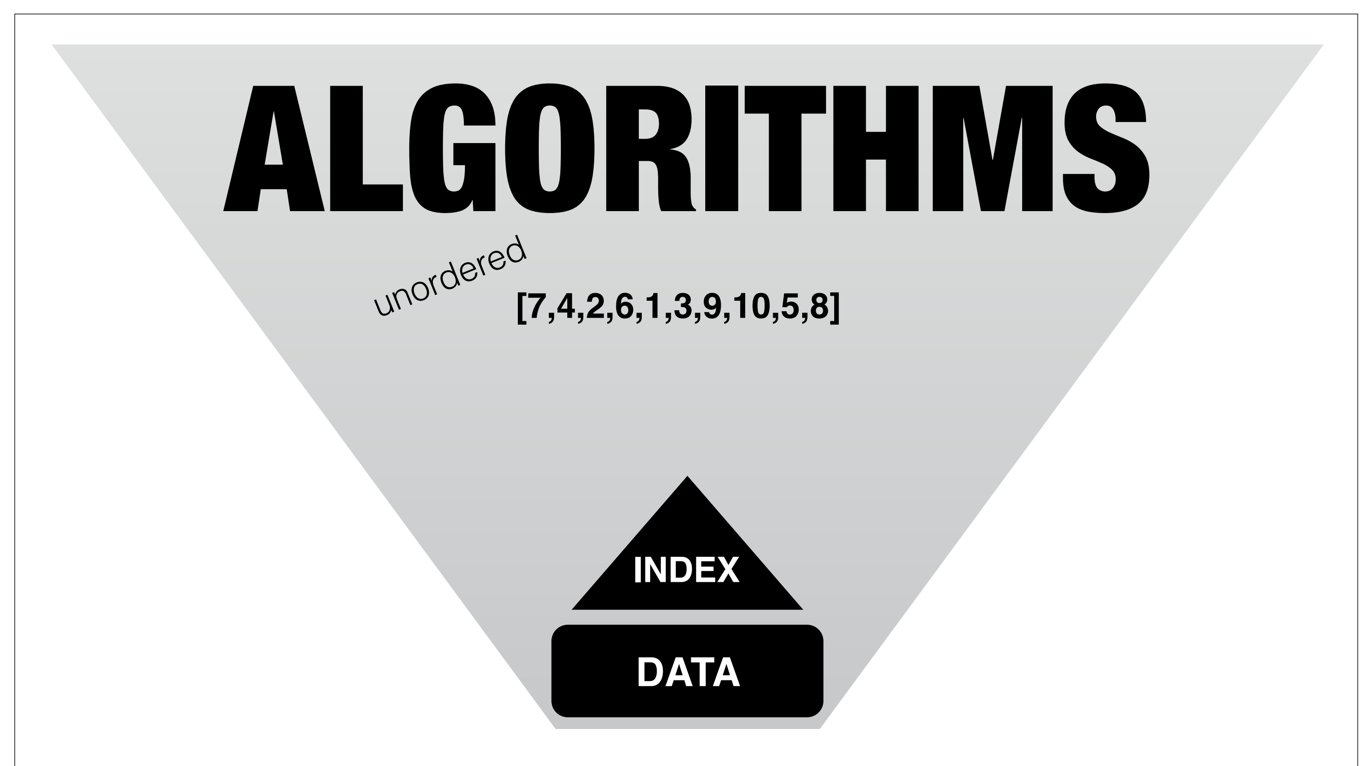
Stratos Idreos & Tim Kraska

it is time to rethink data system architecture
as a **synthesis** and **learning** problem

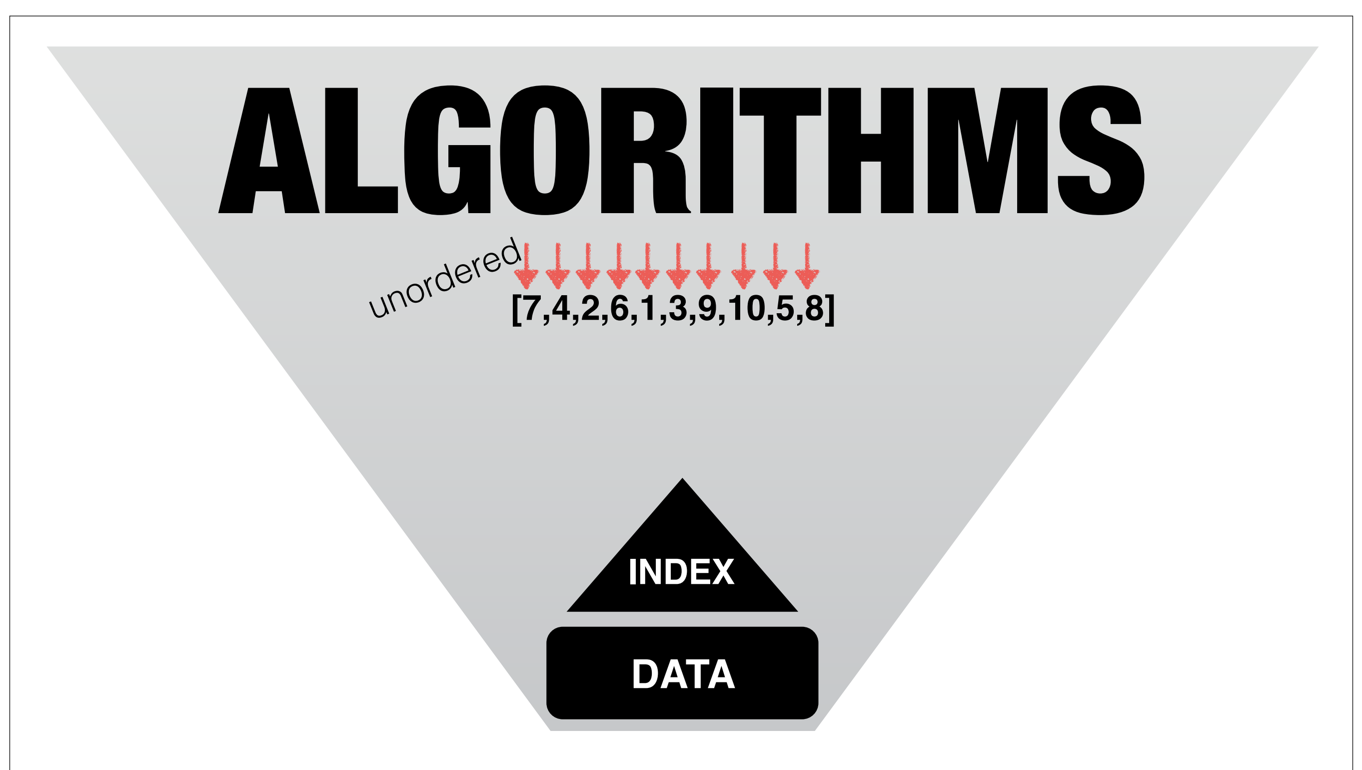
INDEX
DATA
HOW
TO STORE
DATA



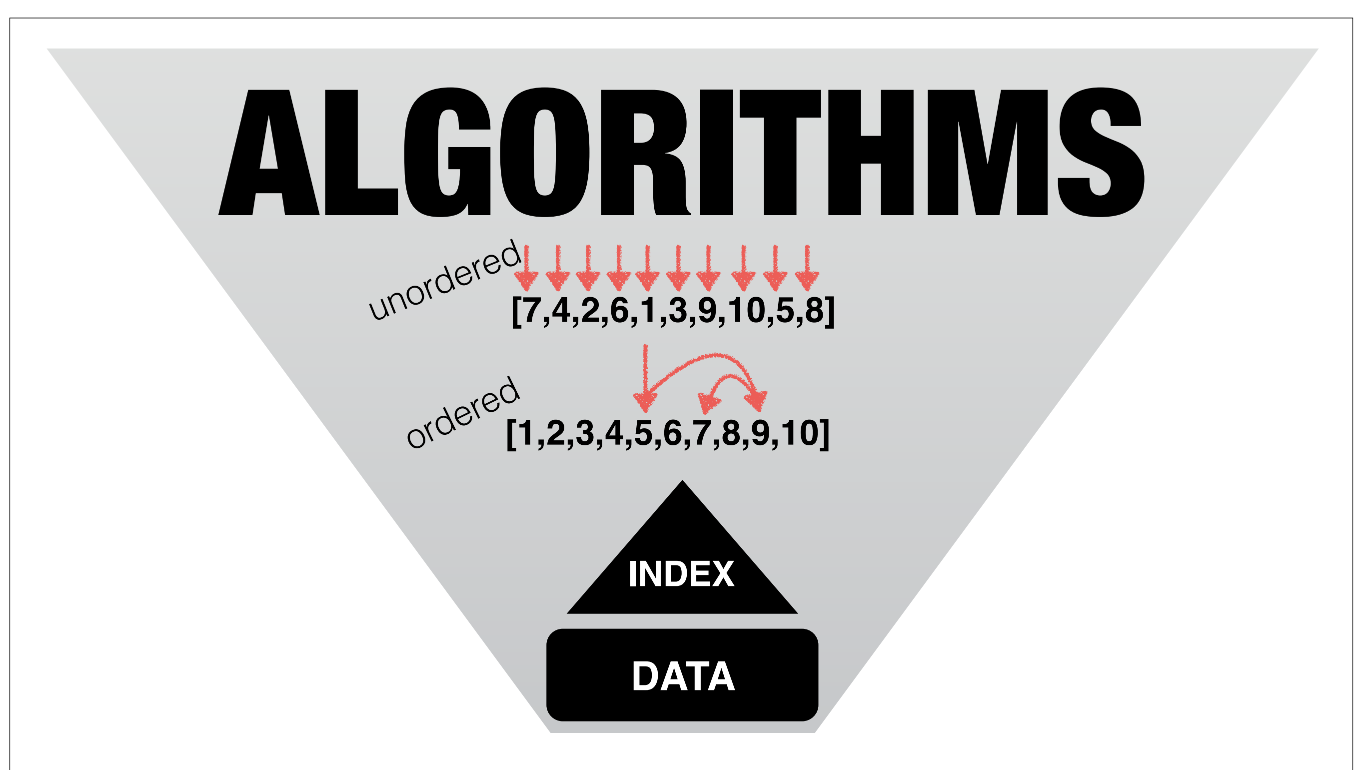

Data structures are at the core of any data driven algorithm. In fact for any given problem, the design of the data structure defines the range of algorithms that may be applied.



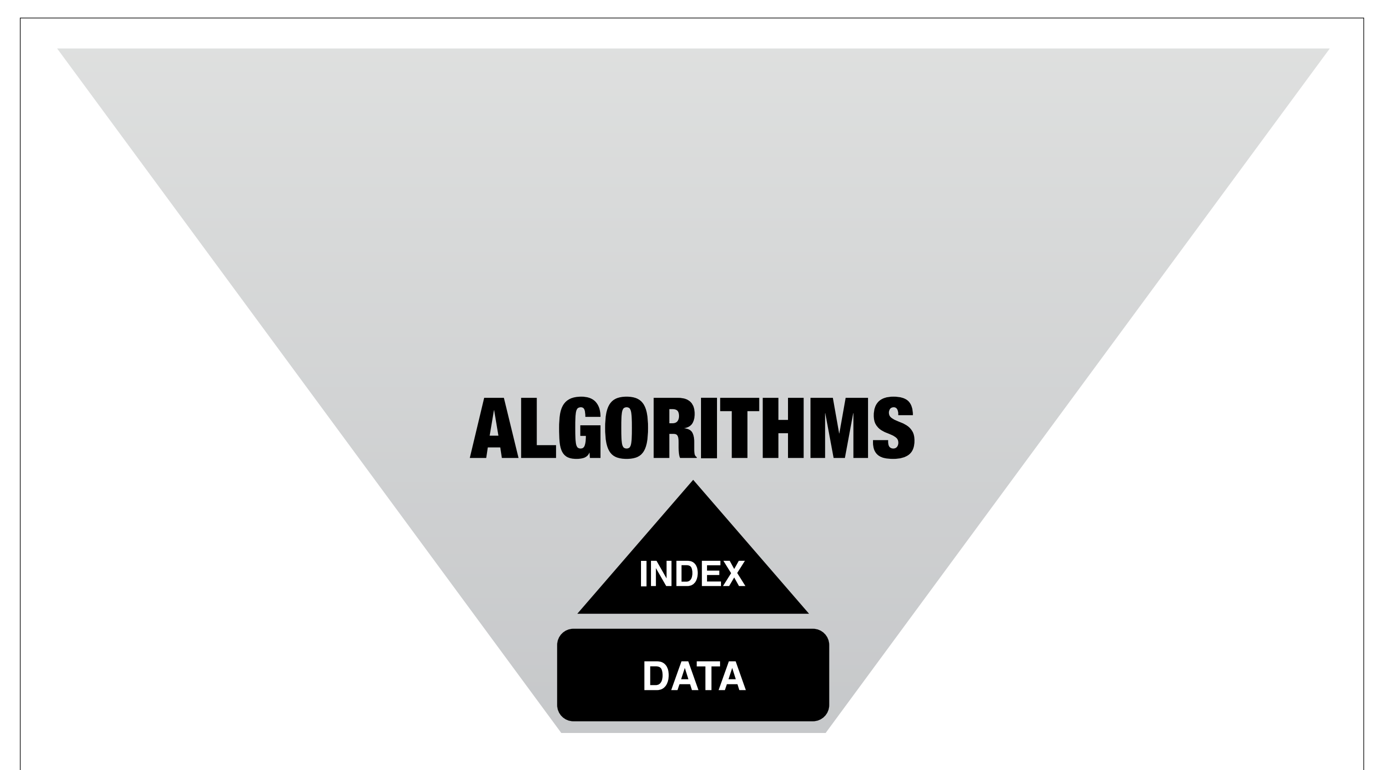

Data structures are at the core of any data driven algorithm. In fact for any given problem, the design of the data structure defines the range of algorithms that may be applied.



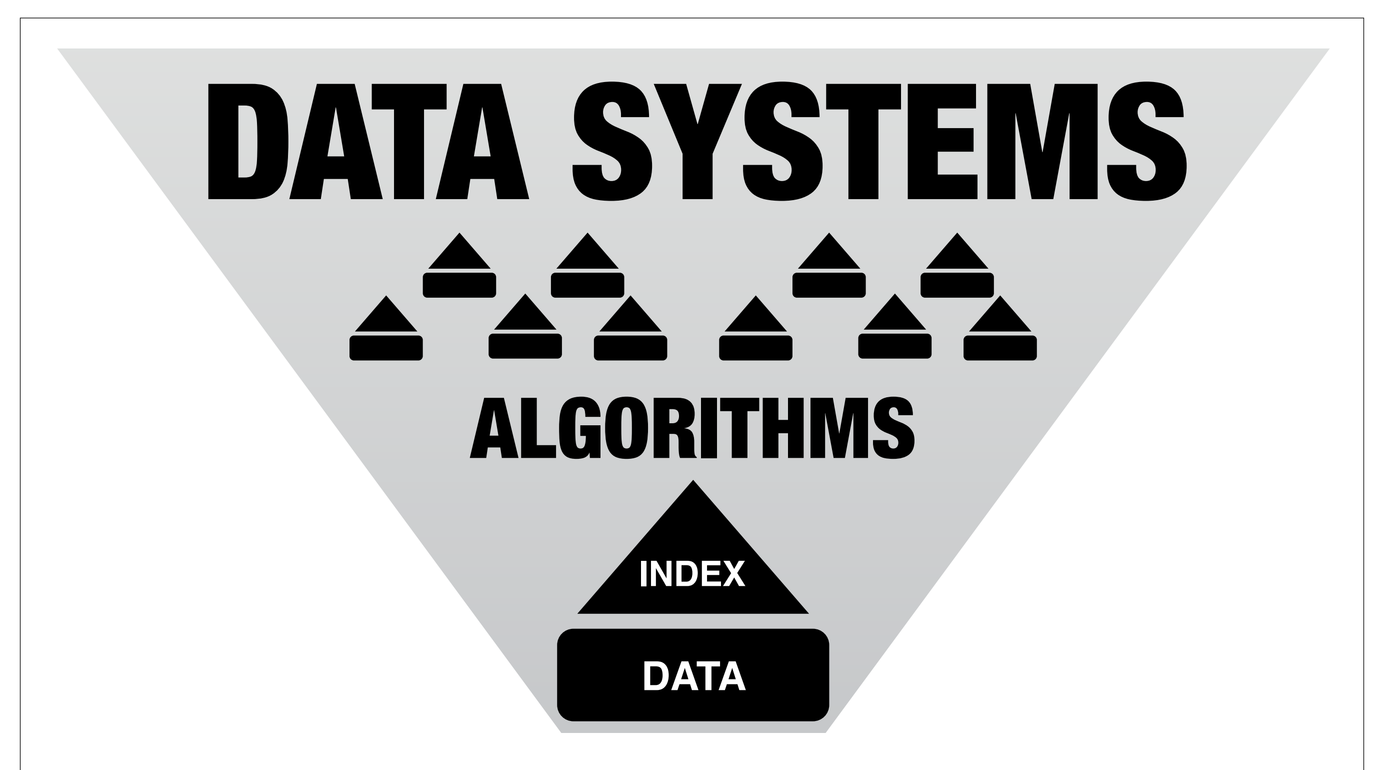

Data structures are at the core of any data driven algorithm. In fact for any given problem, the design of the data structure defines the range of algorithms that may be applied.



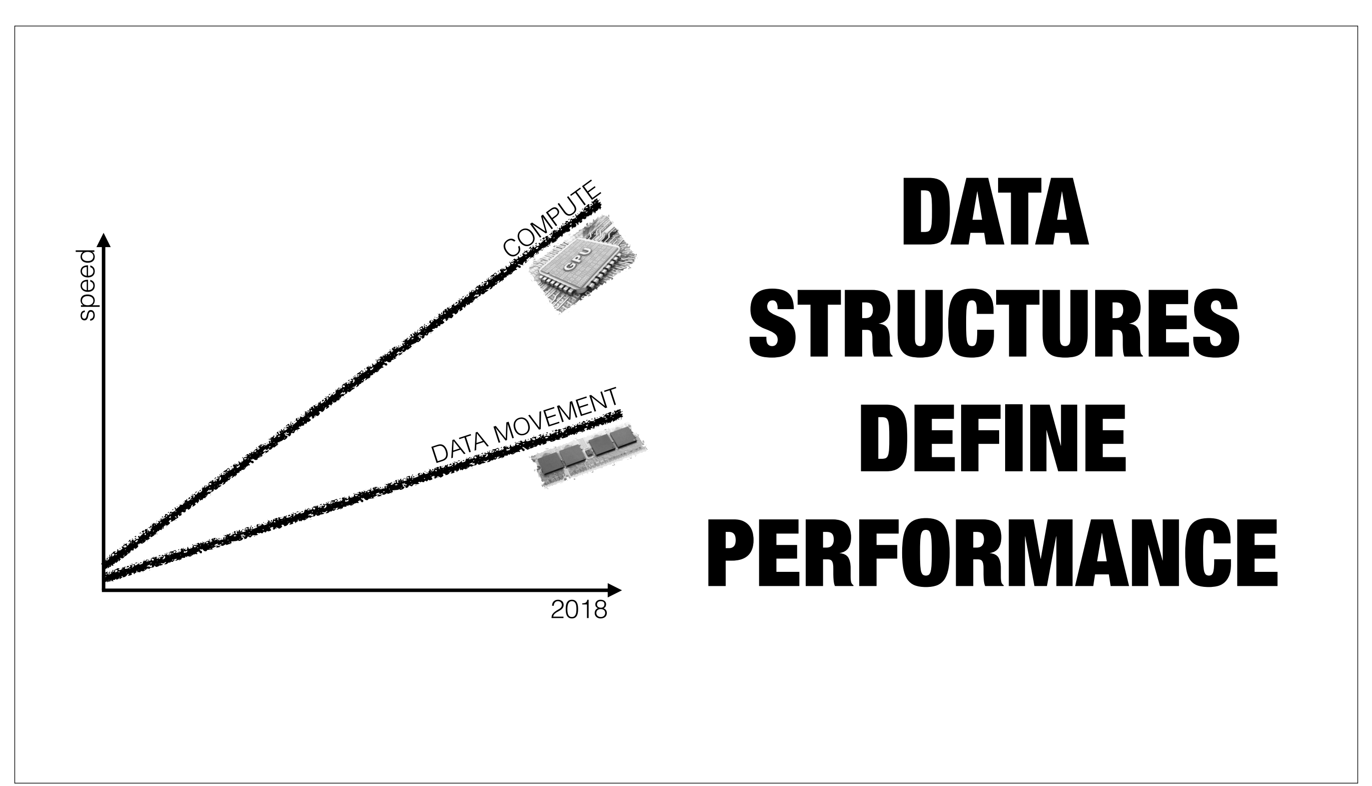

Data structures are at the core of any data driven algorithm. In fact for any given problem, the design of the data structure defines the range of algorithms that may be applied.



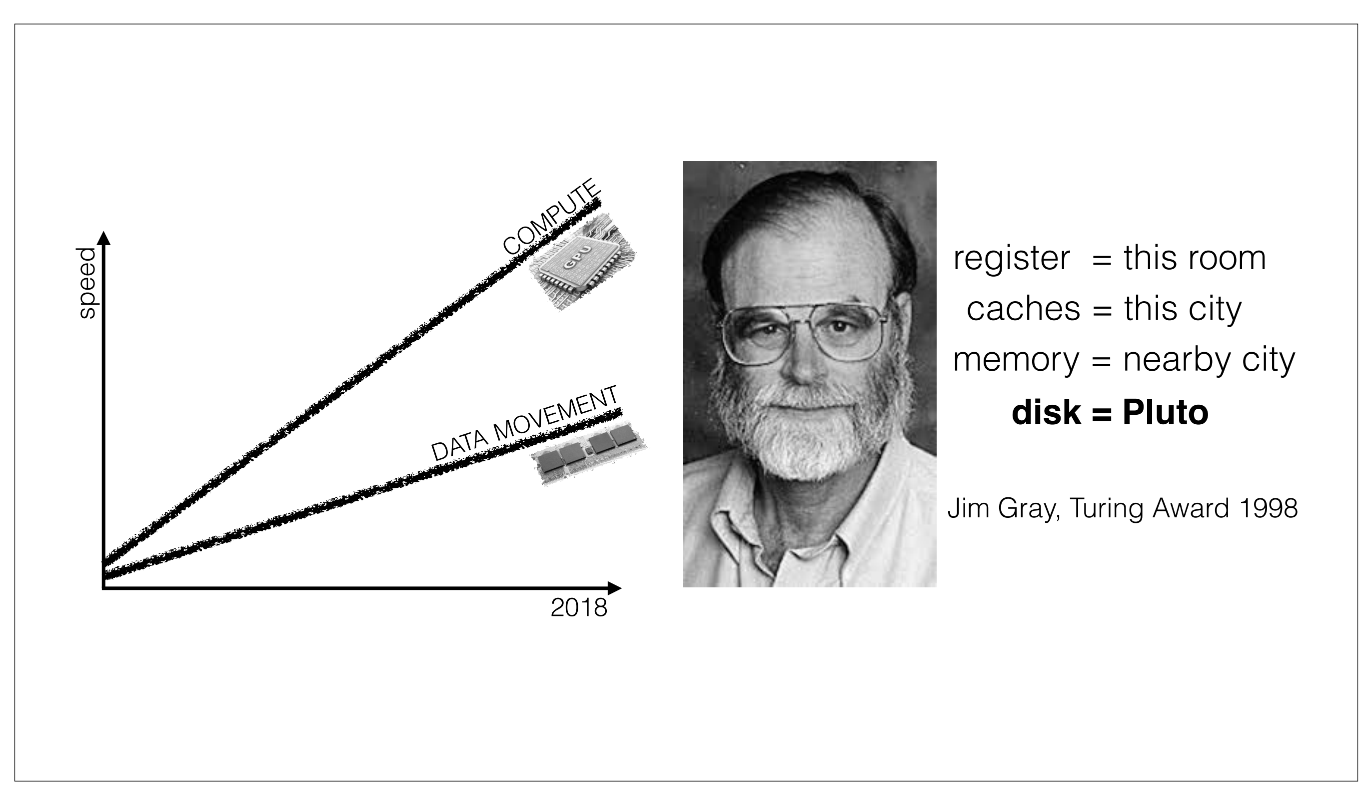

Systems can be seen as a collection of many data structures and algorithms.



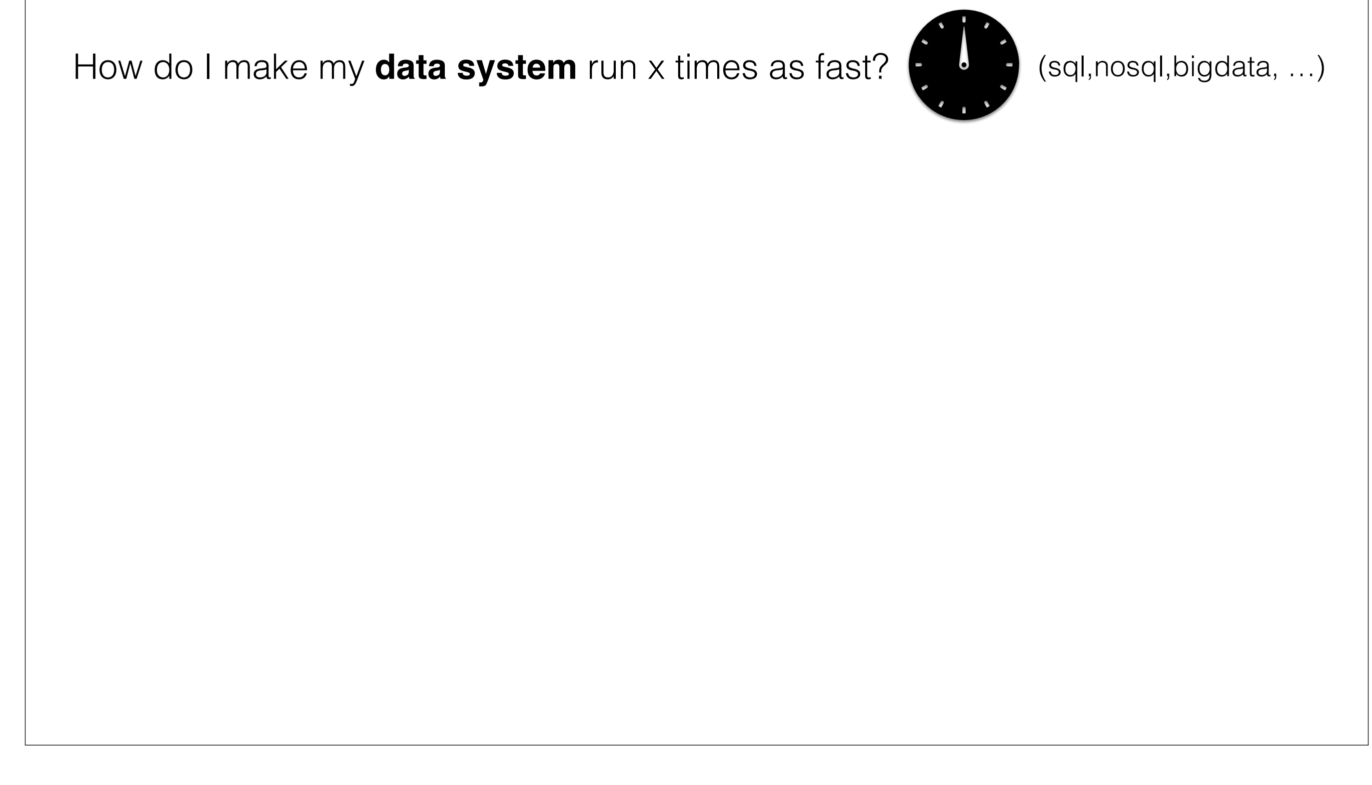

Systems can be seen as a collection of many data structures and algorithms.



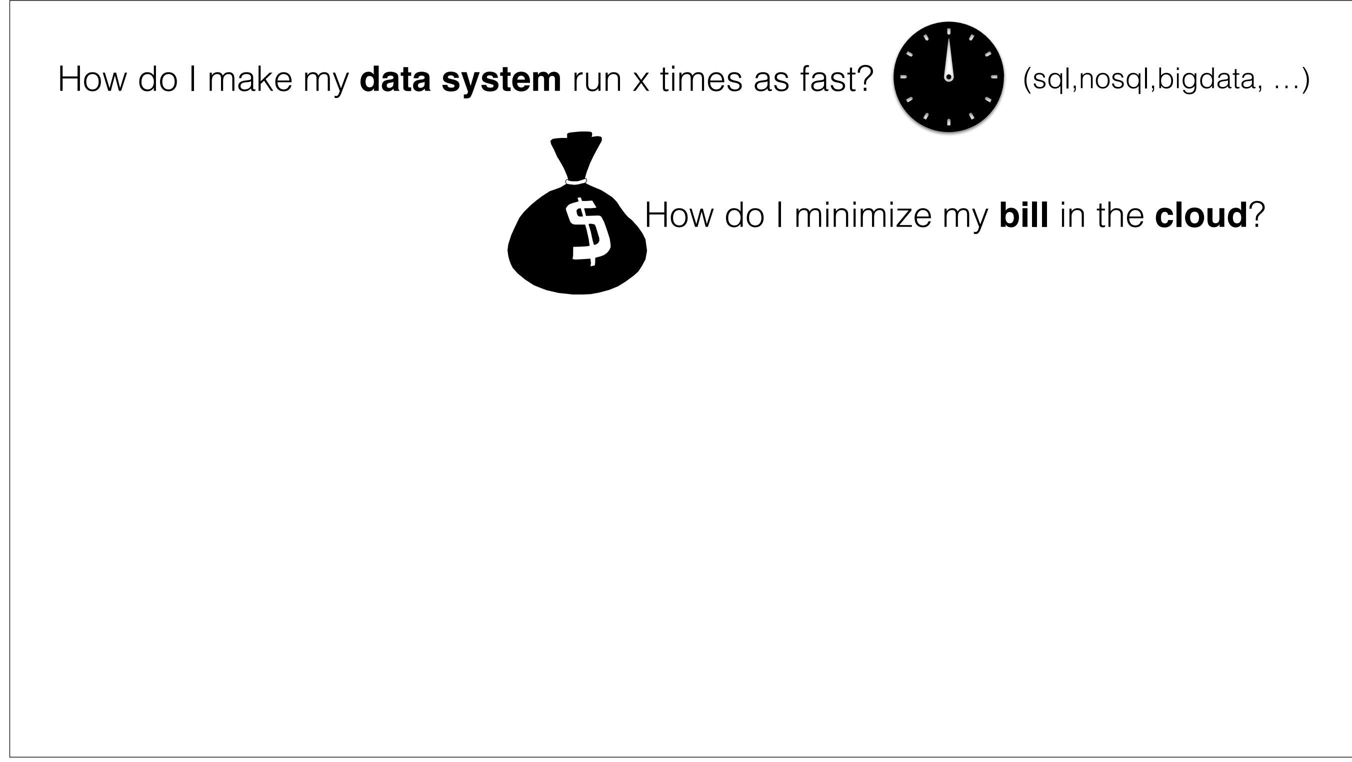

As time goes by, data structures become ever more critical for data driven applications.



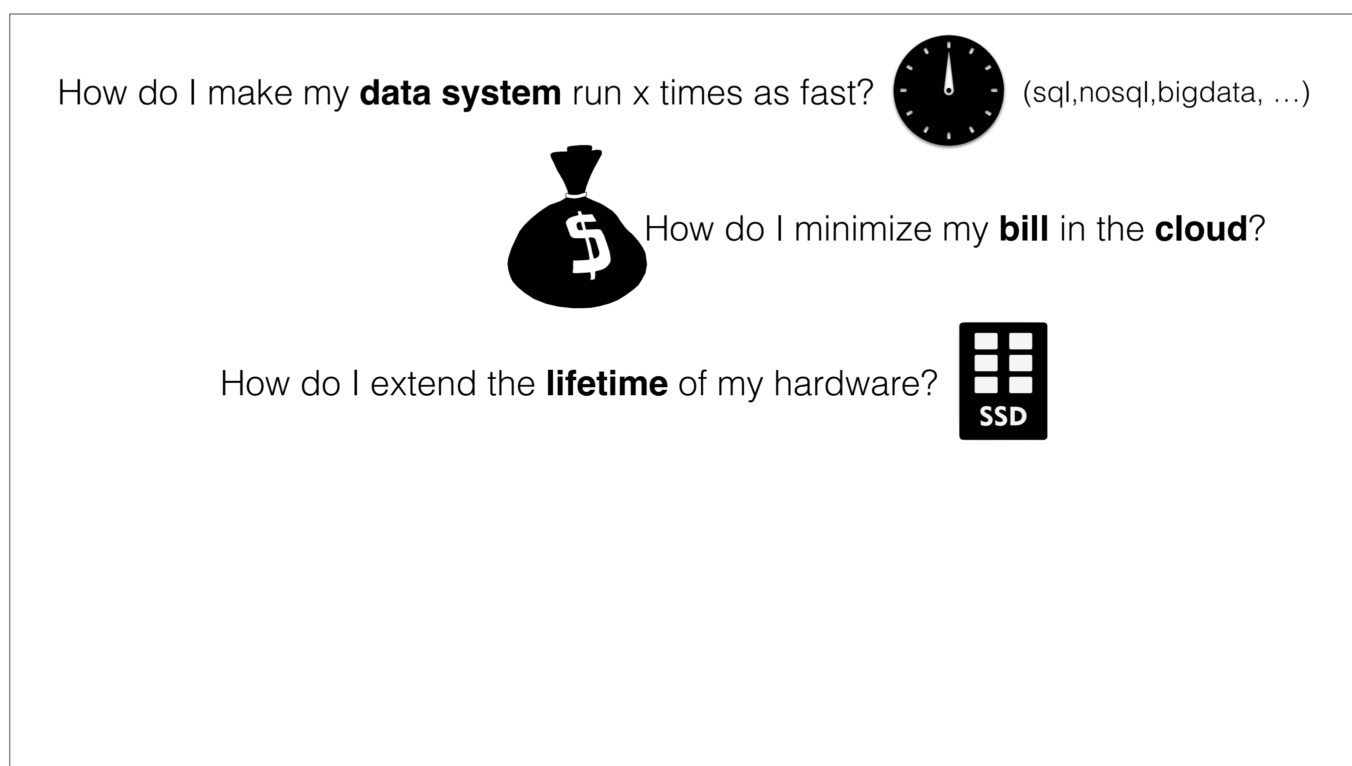

As time goes by, data structures become ever more critical for data driven applications.



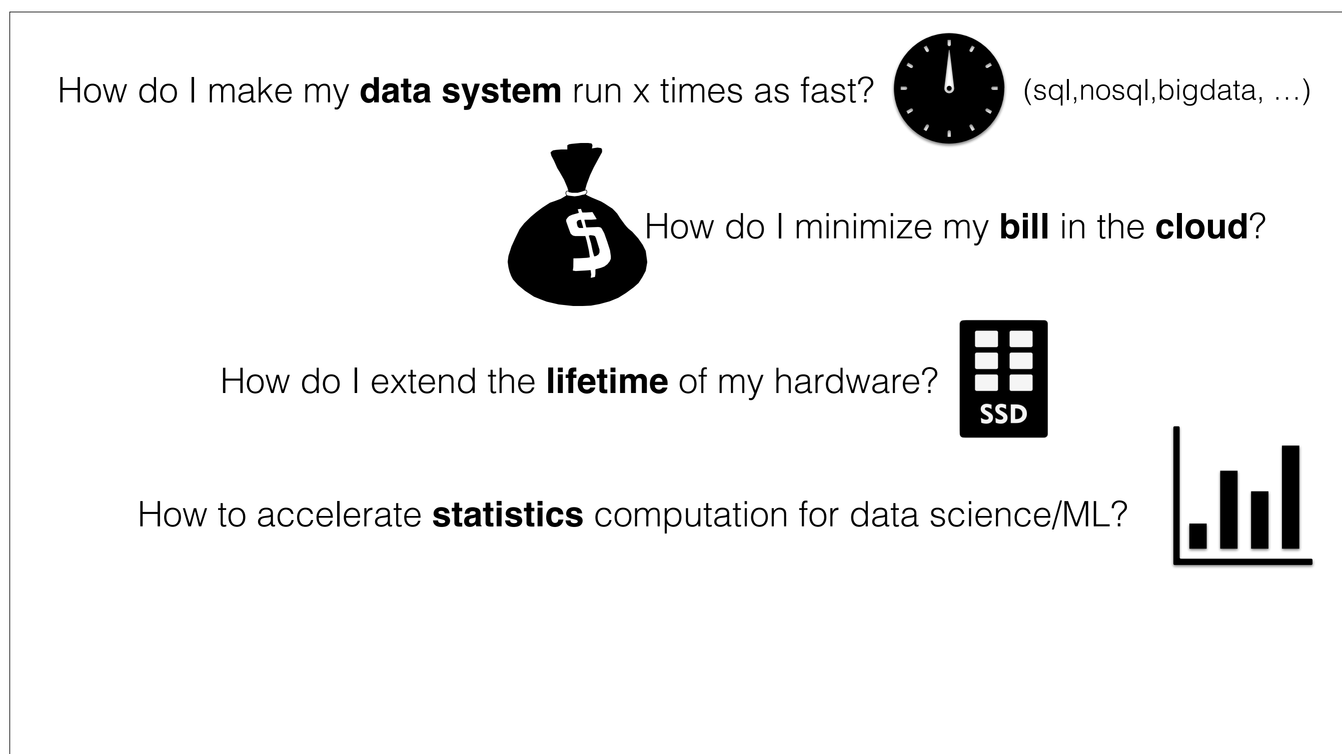

Data structures are prevalent across many applications. Many data driven problems can in fact be seen as a data structure problem.



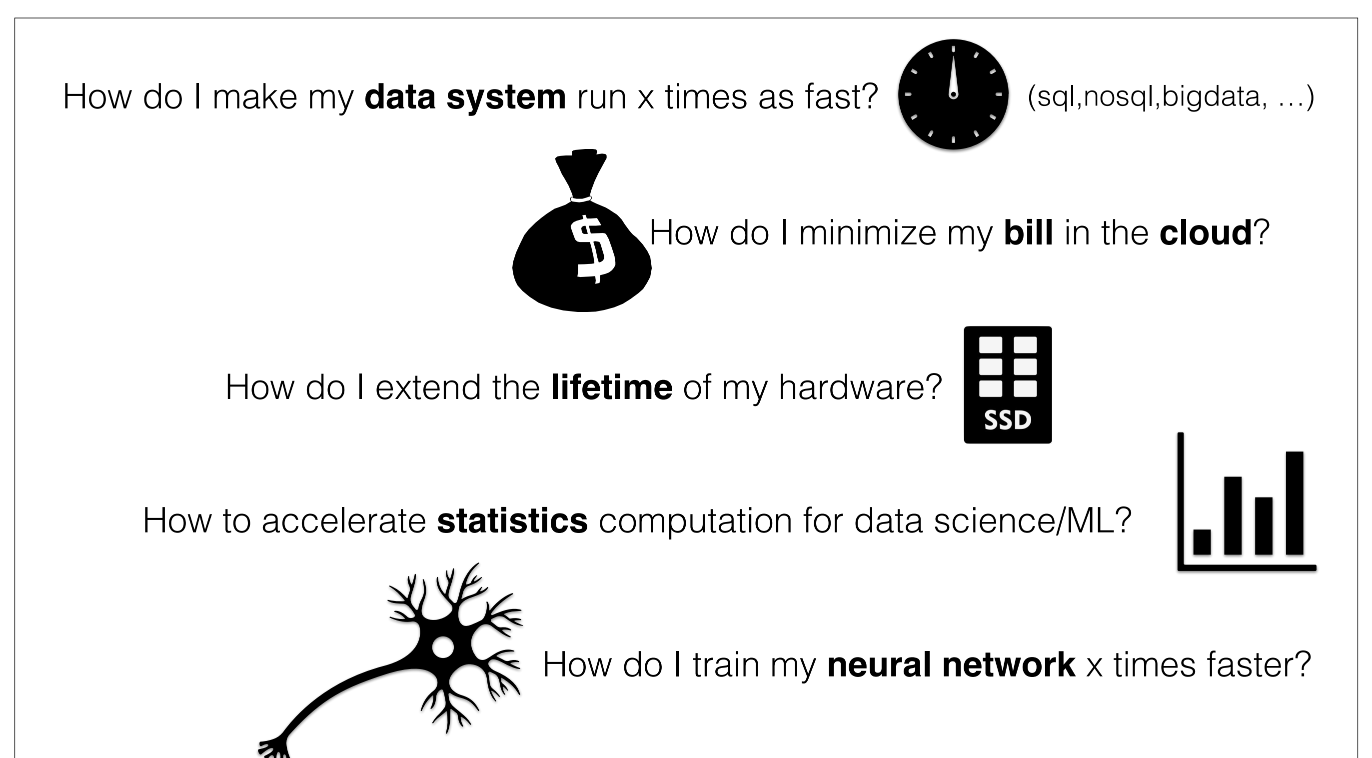

Data structures are prevalent across many applications. Many data driven problems can in fact be seen as a data structure problem.



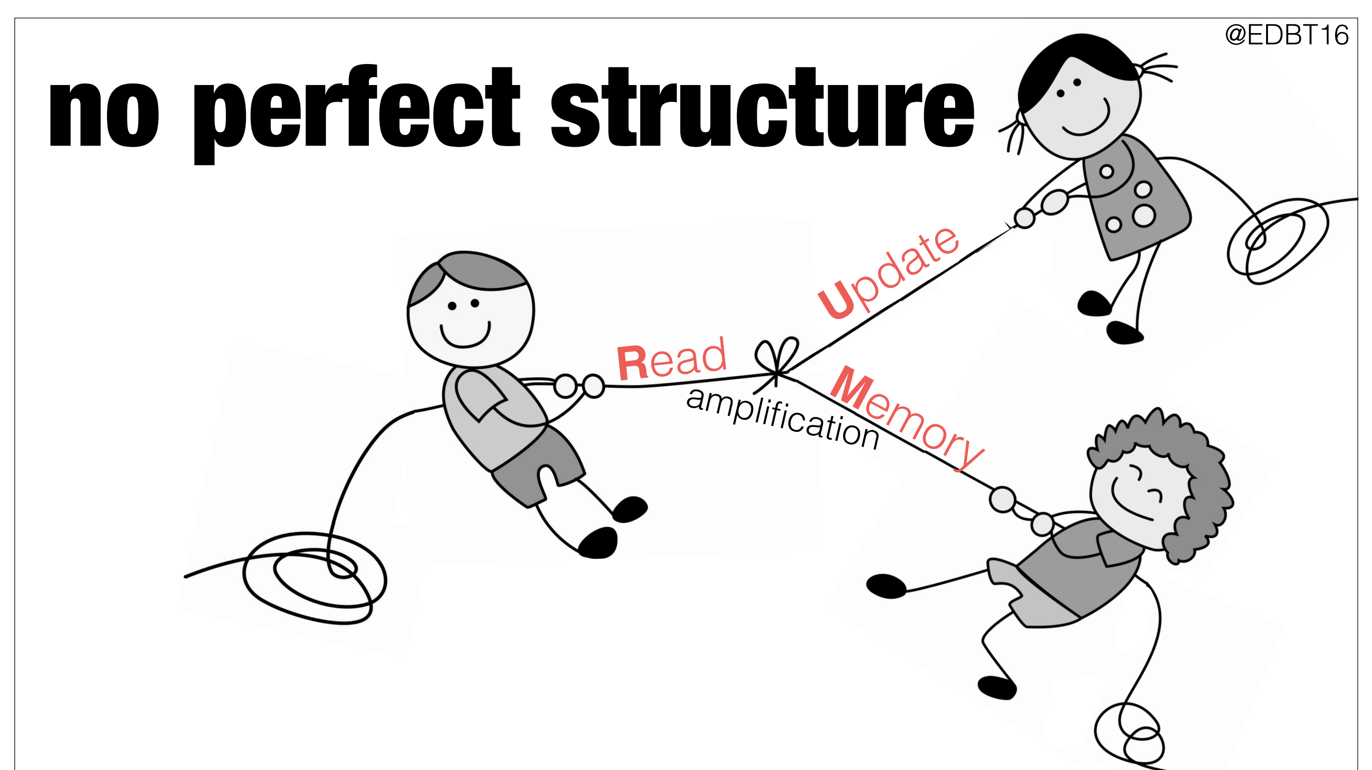

Data structures are prevalent across many applications. Many data driven problems can in fact be seen as a data structure problem.



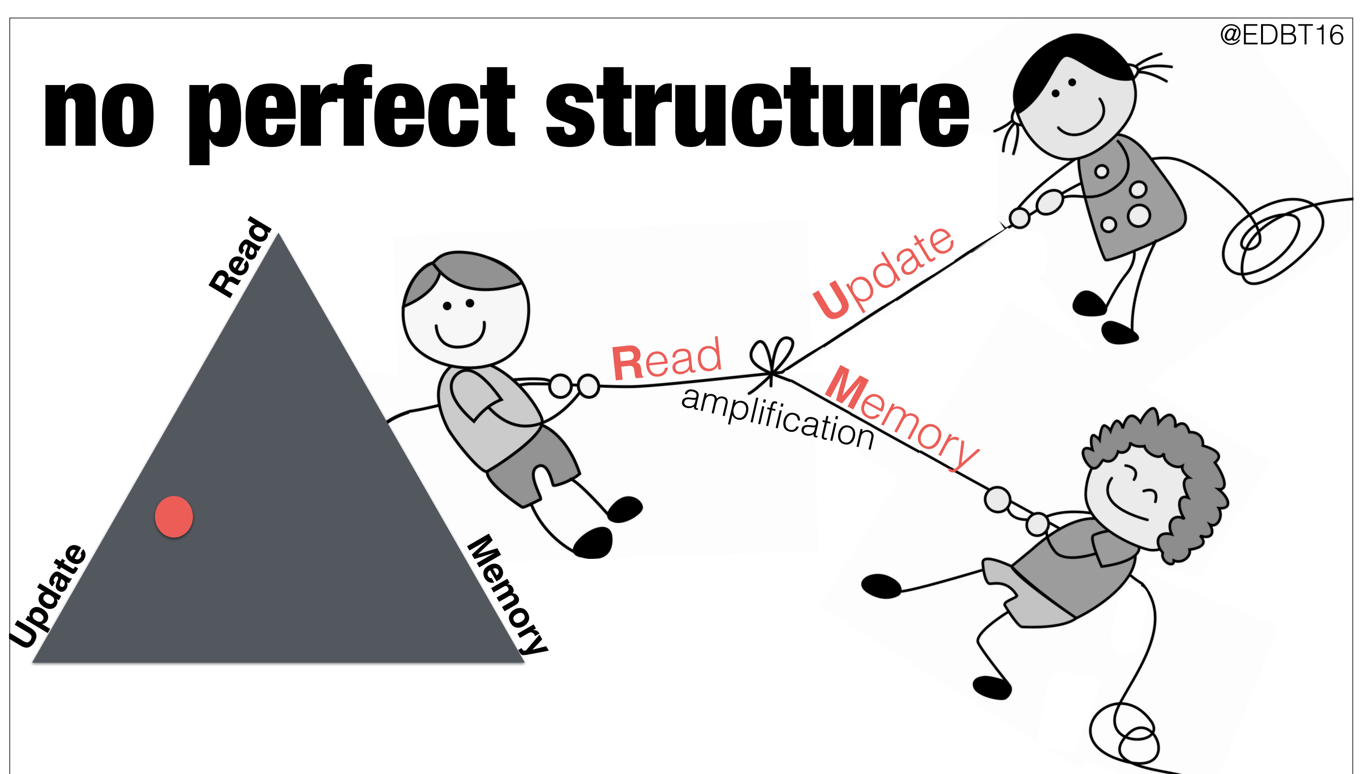

Data structures are prevalent across many applications. Many data driven problems can in fact be seen as a data structure problem.



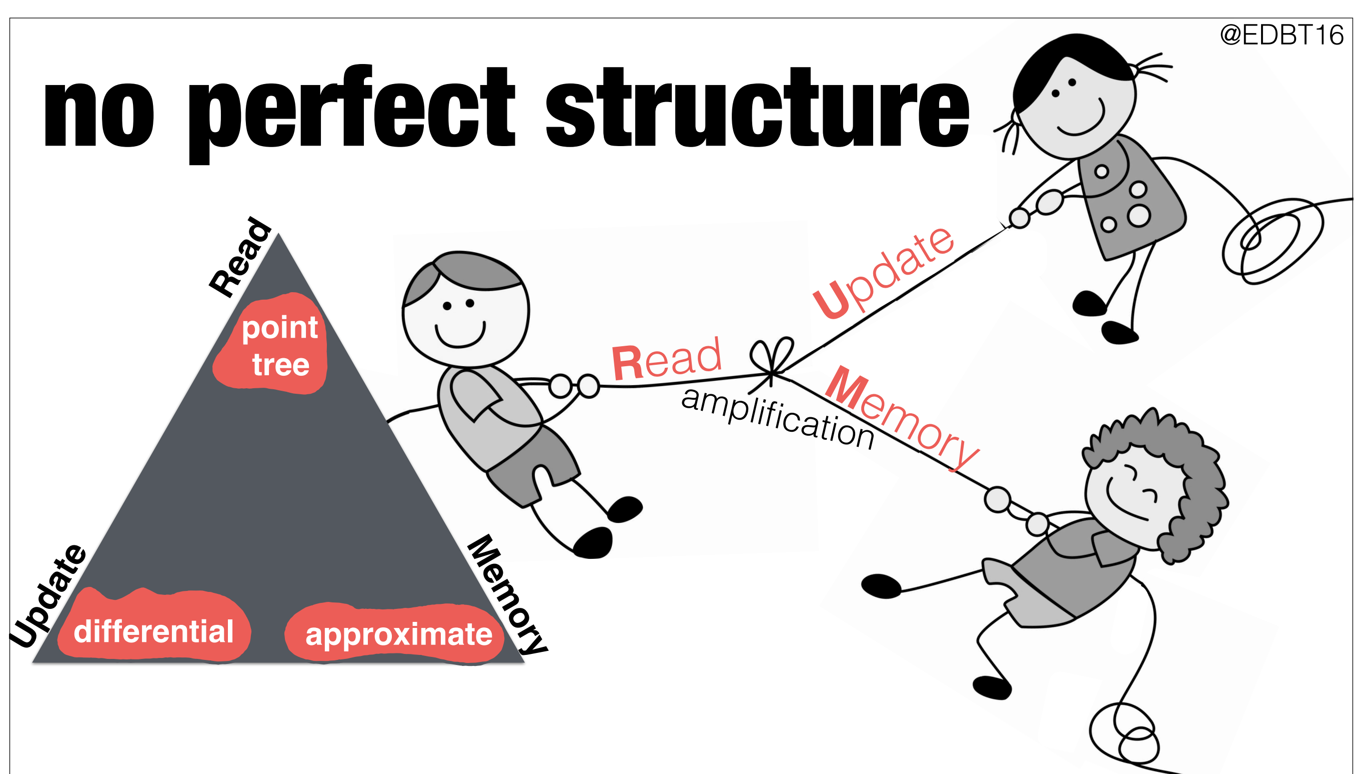

Data structures are prevalent across many applications. Many data driven problems can in fact be seen as a data structure problem.



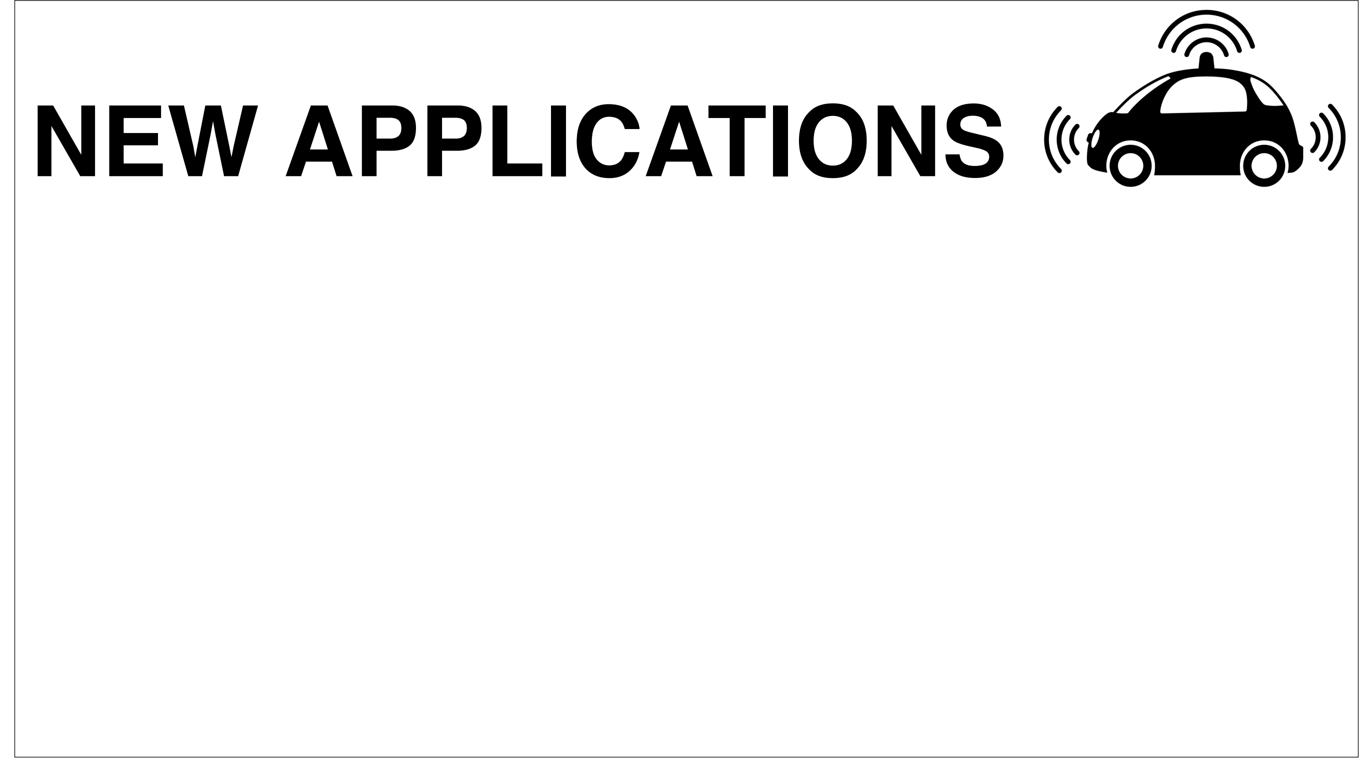

Every data structure design is simply a point in the design space of possible solutions. There is no perfect design. Every design balances the fundamental tradeoffs of Read, Update, and Memory amplification. For example, Read amplification is defined as the excess data an algorithm needs to read on top of the data it wants to read. Typically a data structure would have some kind of metadata or navigation data that help locate the actual data, e.g., the internal nodes of a B-tree. Reading this navigation data is an excess cost, adding to read amplification. Creating a data structure without any navigation data would suffer update or even more read amplification. For example, we could choose to not have any structure in the data at all. Then every query would have to touch all the data. The other extreme would be to sort all data which effectively provides an implicit structure. But then updates get expensive. Overall, there is no perfect design.



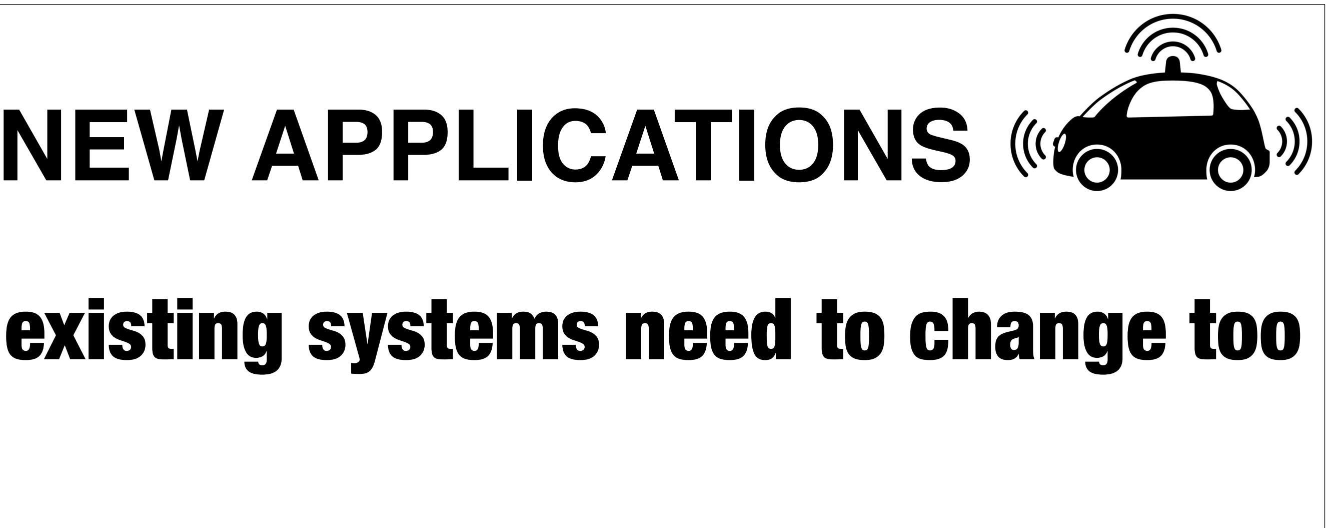

Every data structure design is simply a point in the design space of possible solutions. There is no perfect design. Every design balances the fundamental tradeoffs of Read, Update, and Memory amplification. For example, Read amplification is defined as the excess data an algorithm needs to read on top of the data it wants to read. Typically a data structure would have some kind of metadata or navigation data that help locate the actual data, e.g., the internal nodes of a B-tree. Reading this navigation data is an excess cost, adding to read amplification. Creating a data structure without any navigation data would suffer update or even more read amplification. For example, we could choose to not have any structure in the data at all. Then every query would have to touch all the data. The other extreme would be to sort all data which effectively provides an implicit structure. But then updates get expensive. Overall, there is no perfect design.



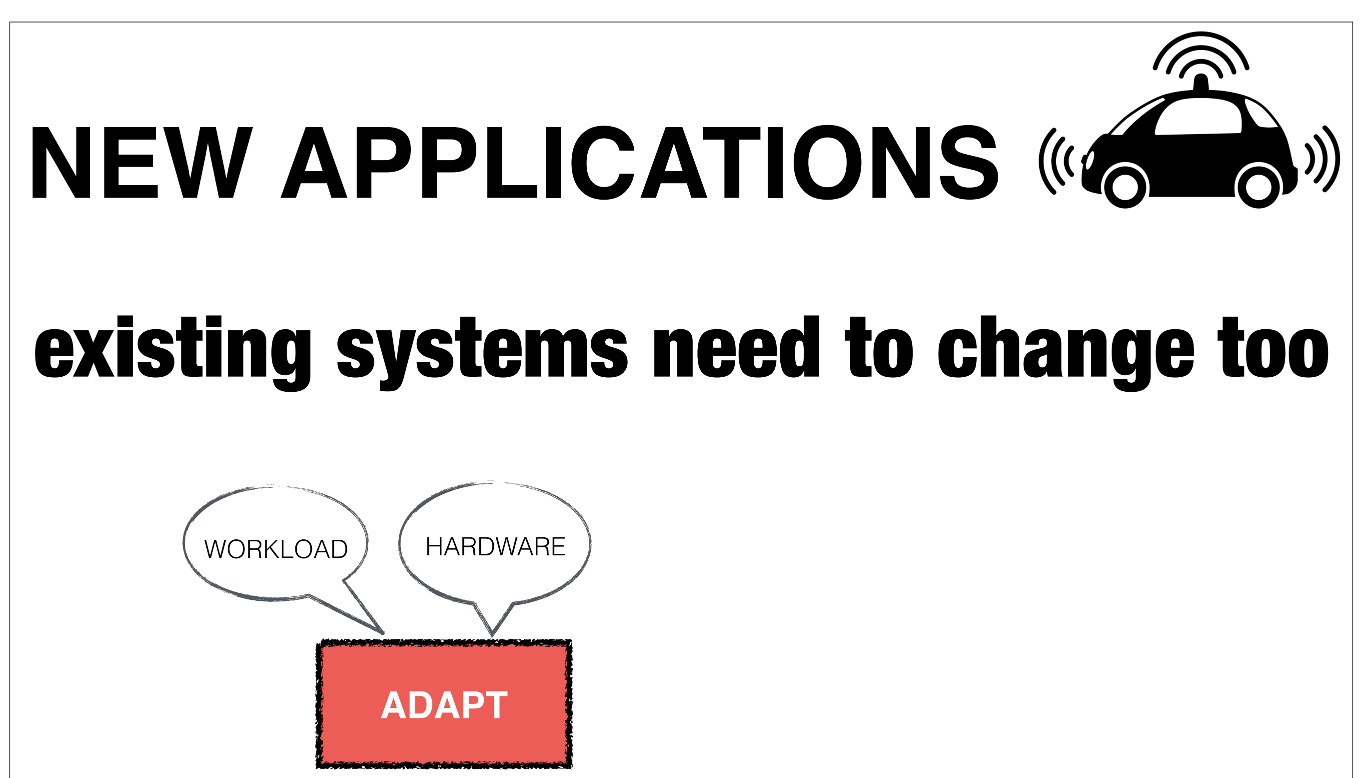

Every data structure design is simply a point in the design space of possible solutions. There is no perfect design. Every design balances the fundamental tradeoffs of Read, Update, and Memory amplification. For example, Read amplification is defined as the excess data an algorithm needs to read on top of the data it wants to read. Typically a data structure would have some kind of metadata or navigation data that help locate the actual data, e.g., the internal nodes of a B-tree. Reading this navigation data is an excess cost, adding to read amplification. Creating a data structure without any navigation data would suffer update or even more read amplification. For example, we could choose to not have any structure in the data at all. Then every query would have to touch all the data. The other extreme would be to sort all data which effectively provides an implicit structure. But then updates get expensive. Overall, there is no perfect design.



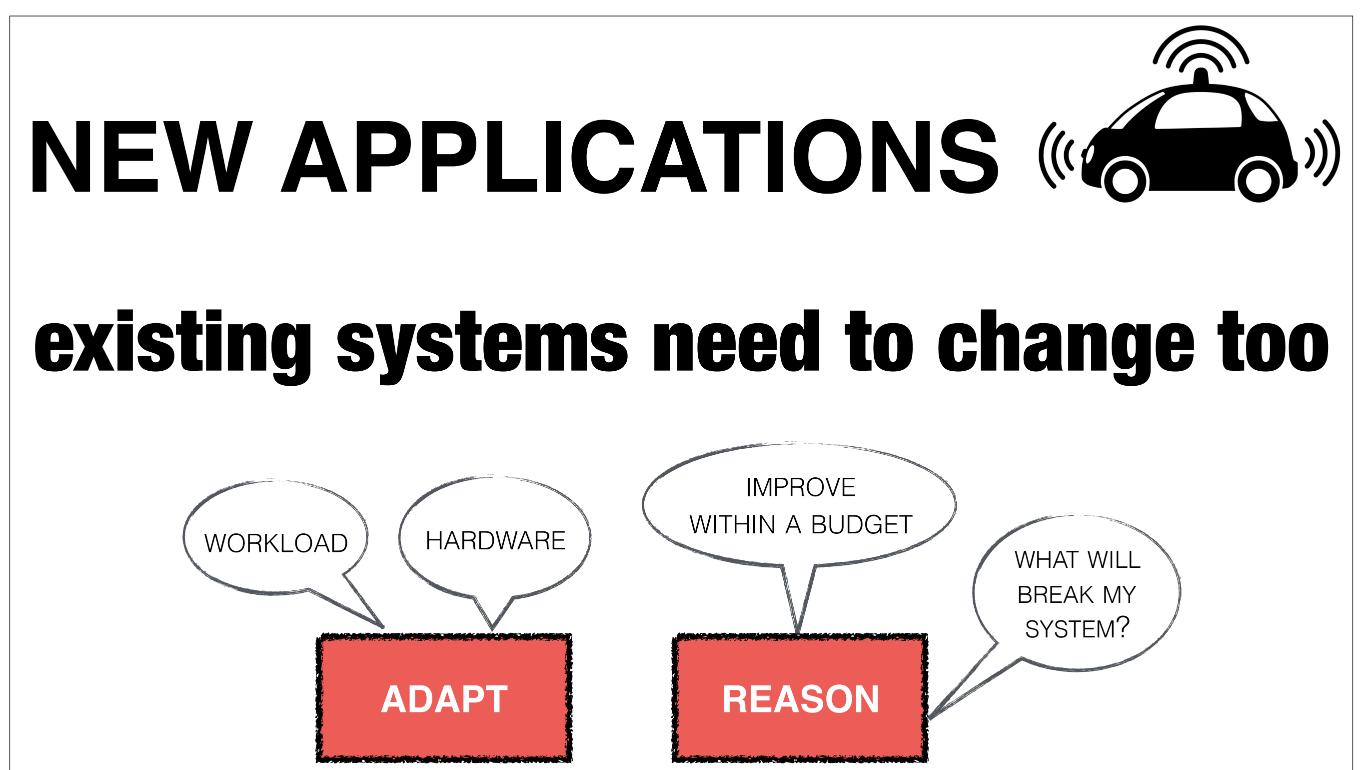

We increasingly need to think of new data structure designs, because applications and data change rapidly and because for data driven applications great performance comes only after rethinking the storage layer as well.



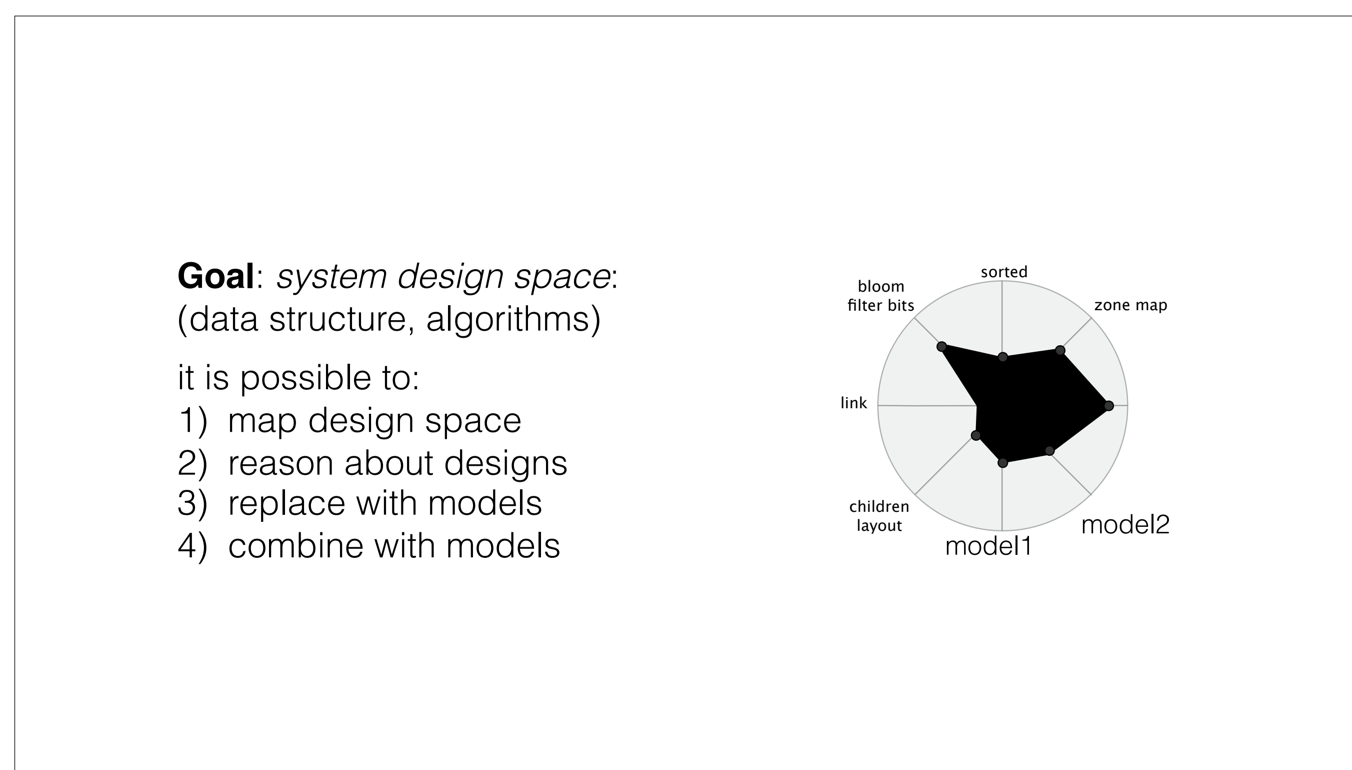

We increasingly need to think of new data structure designs, because applications and data change rapidly and because for data driven applications great performance comes only after rethinking the storage layer as well.




We increasingly need to think of new data structure designs, because applications and data change rapidly and because for data driven applications great performance comes only after rethinking the storage layer as well.




We increasingly need to think of new data structure designs, because applications and data change rapidly and because for data driven applications great performance comes only after rethinking the storage layer as well.

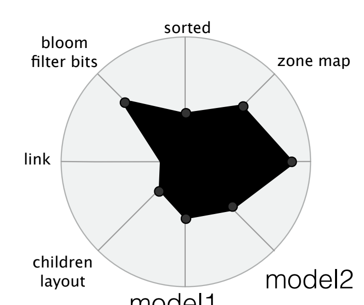


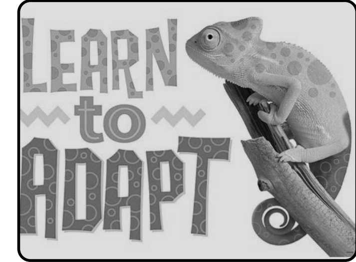

The goal of the tutorial is to provide evidence that the four items on this slide are possible. And then to argue that given these advancements the time is ripe to think of systems that can automatically adapt to their environment via “synthesis and learning”. By itself adaptation has always been a goal and many great efforts of the past have made big steps in this direction: people have used different names for such systems” adaptive, auto-tuned, just-in-time”. The new angle here is “synthesis and learning”. These represent fundamentally new paradigms that we argue can push the limits of such adaptive systems to the next level also by utilizing past techniques.

**Goal**: *system design space*:
(data structure, algorithms)

it is possible to:

1) map design space
2) reason about designs
3) replace with models
4) combine with models

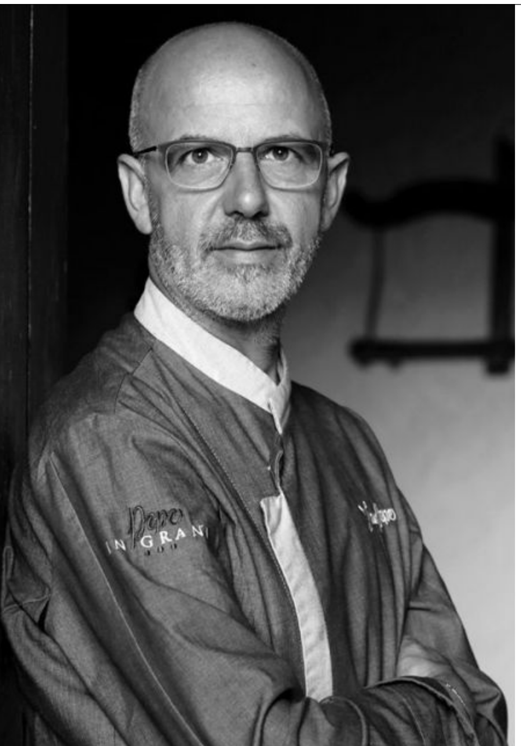



Synthesis and Learning are key to
"adaptive/auto-tuned/just-in-time/…" systems

The goal of the tutorial is to provide evidence that the four items on this slide are possible. And then to argue that given these advancements the time is ripe to think of systems that can automatically adapt to their environment via "synthesis and learning". By itself adaptation has always been a goal and many great efforts of the past have made big steps in this direction: people have used different names for such systems" adaptive, auto-tuned, just-in-time". The new angle here is "synthesis and learning". These represent fundamentally new paradigms that we argue can push the limits of such adaptive systems to the next level also by utilizing past techniques.

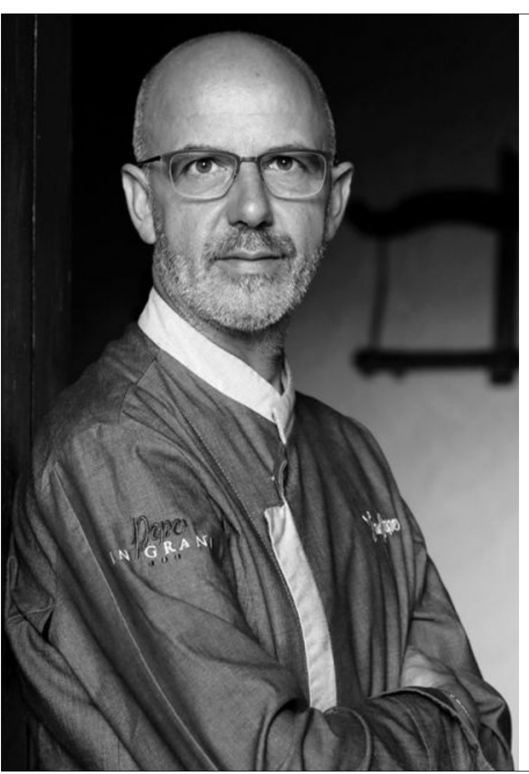
always (close to) optimal data system by
adapting to data, queries, and hardware!!



**Franco Pepe**

“there is no pizza recipe
he who thinks there is a pizza recipe is no pizza maker”

At first it may seem counter intuitive that it is possible to synthesize or automate part of the design of a system. In fact, the more expert one is, the more likely it is to think that this may be impossible. We will show many encouraging examples that this direction is an exciting research path.

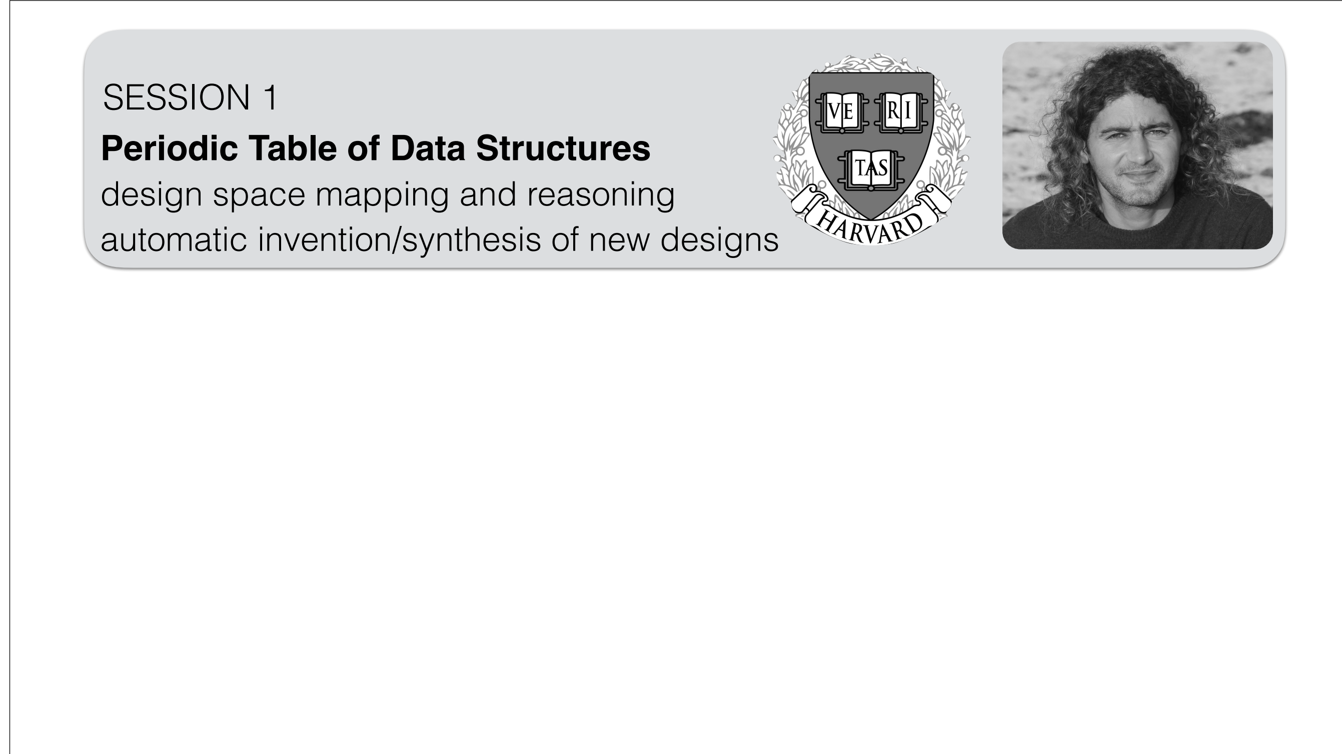


**Franco Pepe**

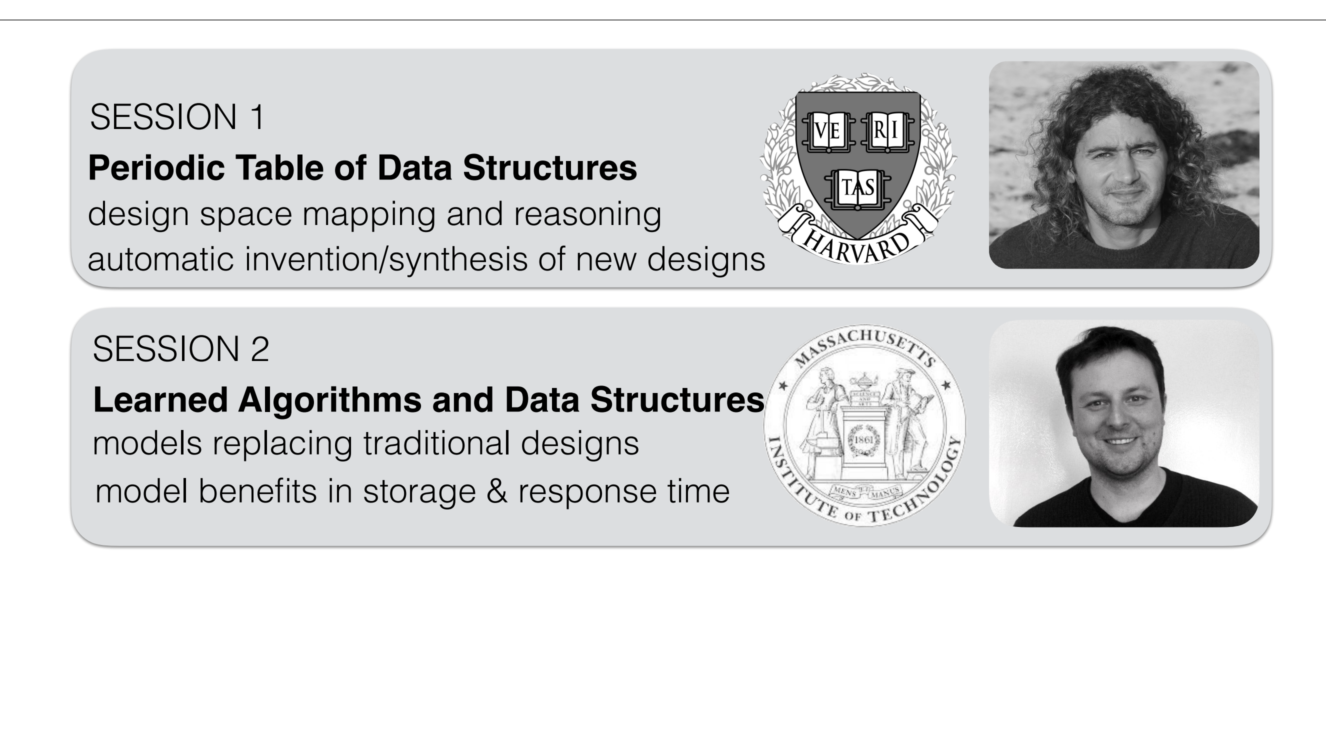
"there is no pizza recipe
he who thinks there is a pizza recipe is no pizza maker"

many encouraging results,
it is possible in subdomains already,
tons of opportunity and exciting research
any step forward is big improvement

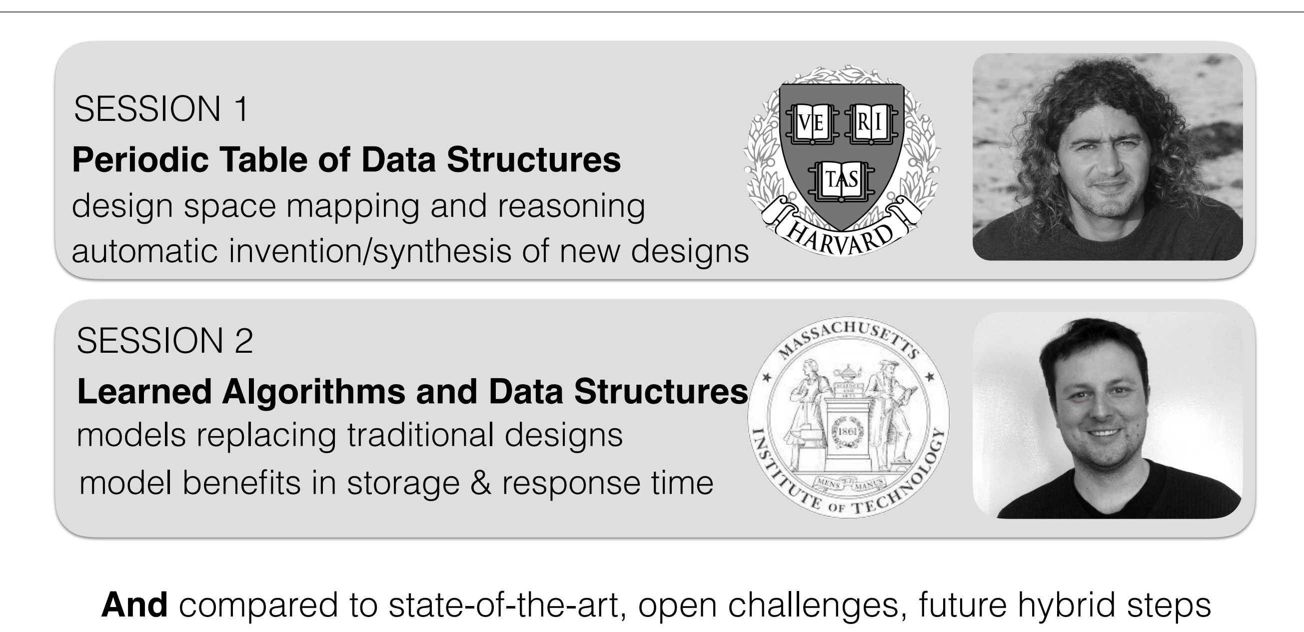
At first it may seem counter intuitive that it is possible to synthesize or automate part of the design of a system. In fact, the more expert one is, the more likely it is to think that this may be impossible. We will show many encouraging examples that this direction is an exciting research path.




The tutorial draws from our recent work on the Data Calculator, Periodic Table of Data Structures, and Learned Indexes. We present the basic principles of these works, their similarities, and possible synergies.




The tutorial draws from our recent work on the Data Calculator, Periodic Table of Data Structures, and Learned Indexes. We present the basic principles of these works, their similarities, and possible synergies.




The tutorial draws from our recent work on the Data Calculator, Periodic Table of Data Structures, and Learned Indexes. We present the basic principles of these works, their similarities, and possible synergies.

SESSION 2

**Learned Algorithms and Data Structures**

models replacing traditional designs

model benefits in storage & response time

**And** compared to state-of-the-art, open challenges, future hybrid steps

**Audience:** basic cs/data structures/algorithms

**We welcome questions any time!**

The tutorial draws from our recent work on the Data Calculator, Periodic Table of Data Structures, and Learned Indexes. We present the basic principles of these works, their similarities, and possible synergies.

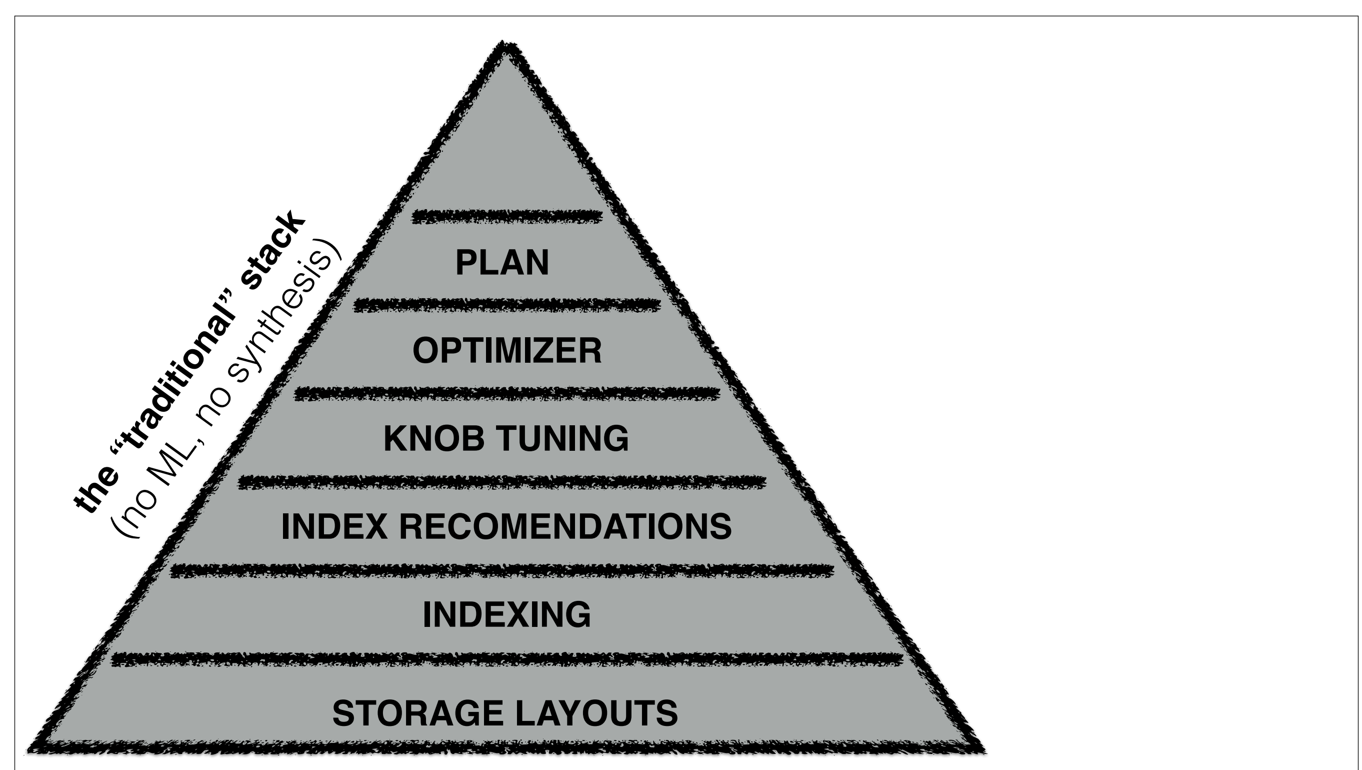
SESSION 1

**Periodic Table of Data Structures**

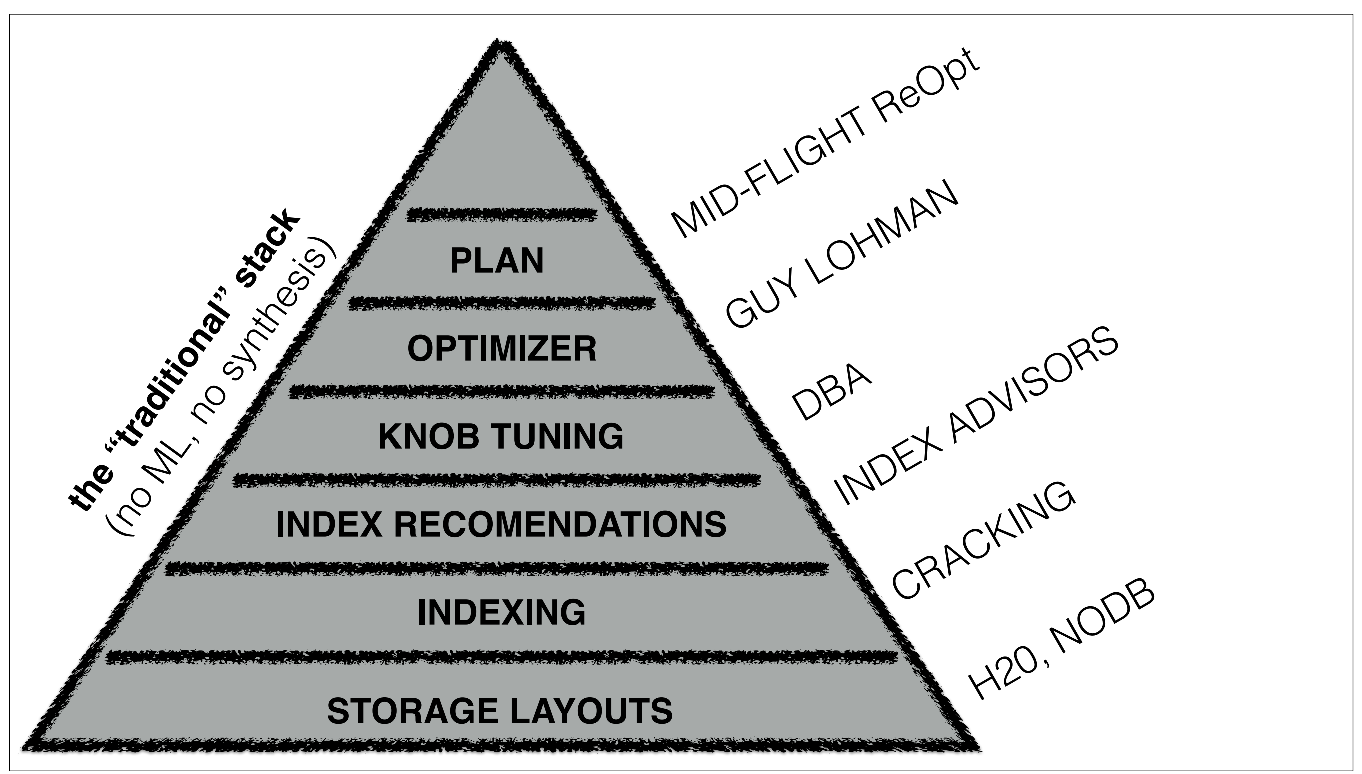
design space mapping and reasoning

automatic invention/synthesis of new designs

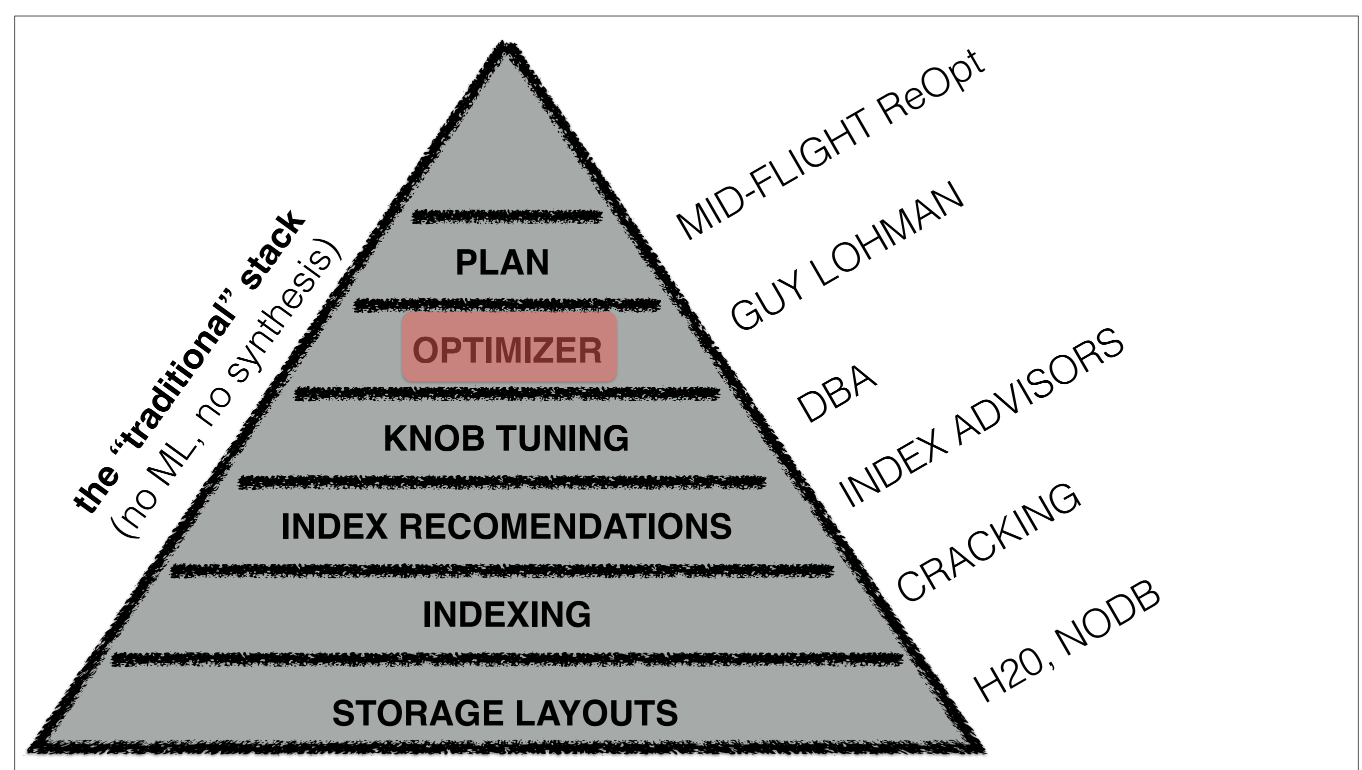
# Part 1



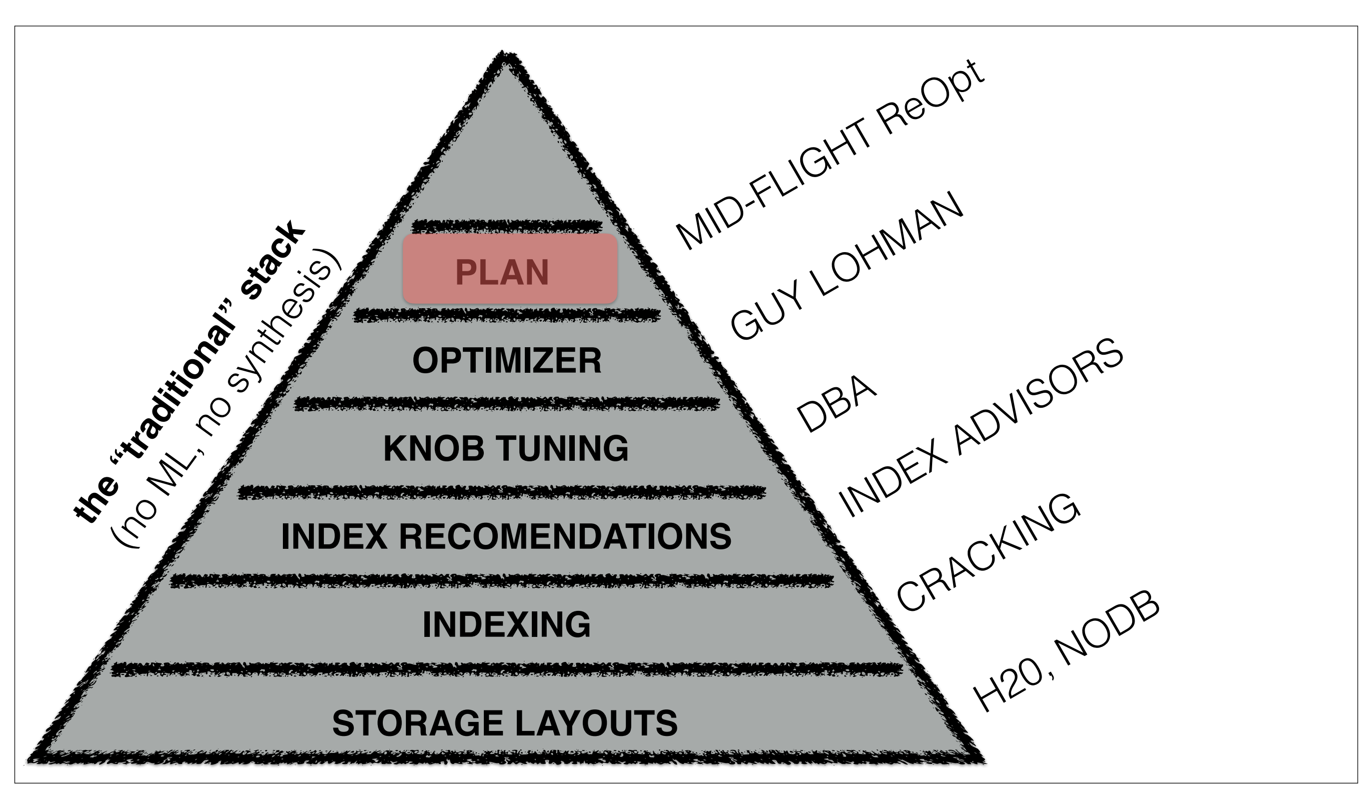

Many efforts in the field have been motivated by the vision of generating tailored systems for a specific scenario. In fact, even traditional databases are architected with this vision in mind. A generic database system can optimize a plan on the fly to match the query needs, it can choose from different storage and indexing options, etc. This is how generic database systems can be used in a wealth of applications! And then recent research has tried to push the boundaries of tailored designs be rethinking parts of the stack of a database system.




Many efforts in the field have been motivated by the vision of generating tailored systems for a specific scenario. In fact, even traditional databases are architected with this vision in mind. A generic database system can optimize a plan on the fly to match the query needs, it can choose from different storage and indexing options, etc. This is how generic database systems can be used in a wealth of applications! And then recent research has tried to push the boundaries of tailored designs be rethinking parts of the stack of a database system.




Many efforts in the field have been motivated by the vision of generating tailored systems for a specific scenario. In fact, even traditional databases are architected with this vision in mind. A generic database system can optimize a plan on the fly to match the query needs, it can choose from different storage and indexing options, etc. This is how generic database systems can be used in a wealth of applications! And then recent research has tried to push the boundaries of tailored designs be rethinking parts of the stack of a database system.


the “traditional” stack
(no ML, no synthesis)
PLAN
OPTIMIZER
KNOB TUNING
INDEX RECOMENDATIONS
INDEXING
STORAGE LAYOUTS
MID-FLIGHT ReOpt
GUY LOHMAN
DBA
INDEX ADVISORS
CRACKING
H20, NODB

**ReOptimization (SIGMOD 1998, SIGMOD 05, TKDE09)**

Optimize but then reconsider if plan deviates

Utilize if possible previous results

For example, ideas that allow an optimizer to evaluate on the fly its own decisions and adapt to the running results make a data system more adaptable to dynamic workloads. Similarly ideas such as Smooth Scan which make plans less dependent on a priori knowledge data and queries allow a system to adapt on-the-fly to the data and queries while providing robust behavior.

**ReOptimization (SIGMOD 1998, SIGMOD 05, TKDE09)**

Optimize but then reconsider if plan deviates

Utilize if possible previous results

**Smooth Scan (ICDE 2015)**

Select operator: Do not plan. Instead, start probing the index and if needed smoothly switch to scan

For example, ideas that allow an optimizer to evaluate on the fly its own decisions and adapt to the running results make a data system more adaptable to dynamic workloads. Similarly ideas such as Smooth Scan which make plans less dependent on a priori knowledge data and queries allow a system to adapt on-the-fly to the data and queries while providing robust behavior.

**ReOptimization (SIGMOD 1998, SIGMOD 05, TKDE09)**

Optimize but then reconsider if plan deviates

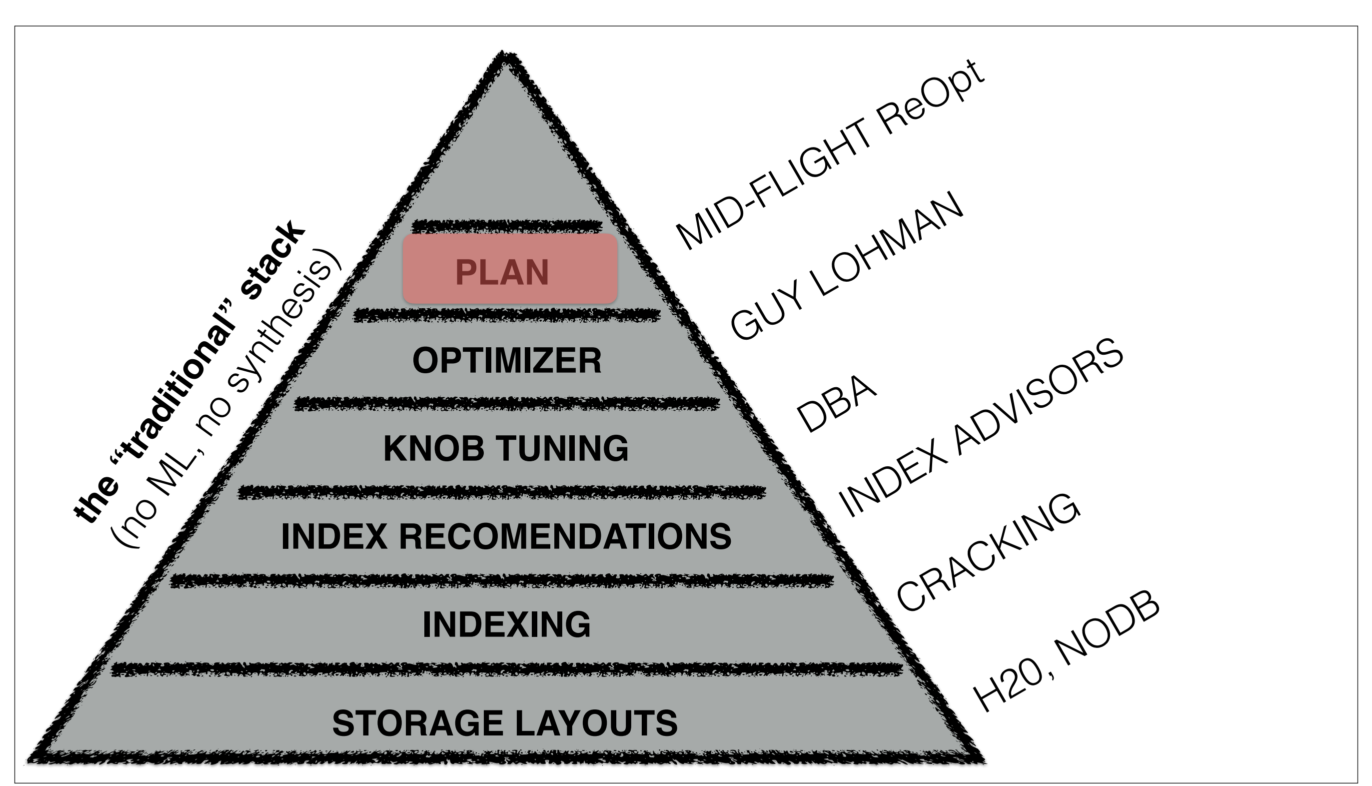
Utilize if possible previous results

**Smooth Scan (ICDE 2015)**

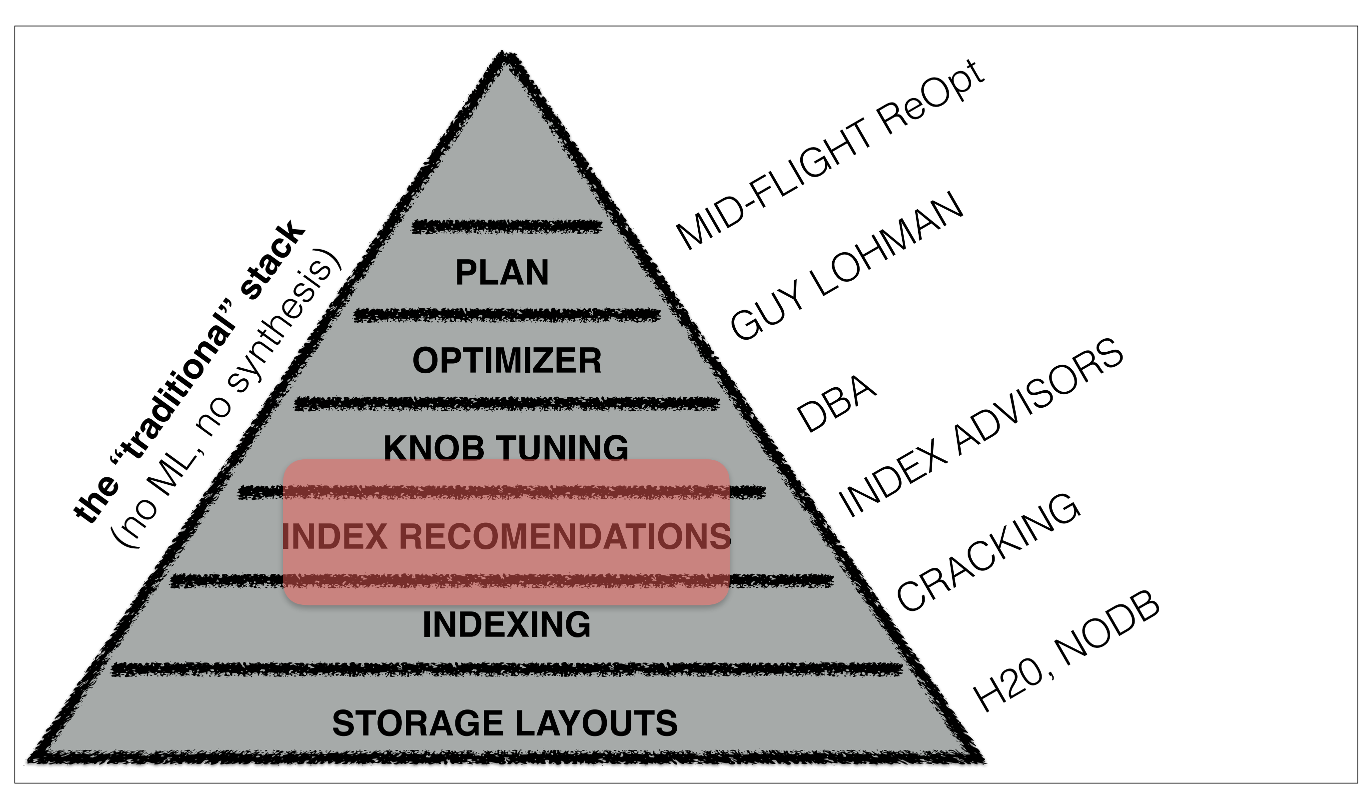
Select operator: Do not plan. Instead, start probing the index and if needed smoothly switch to scan

The database plans automatically adapt to the workload online

For example, ideas that allow an optimizer to evaluate on the fly its own decisions and adapt to the running results make a data system more adaptable to dynamic workloads. Similarly ideas such as Smooth Scan which make plans less dependent on a priori knowledge data and queries allow a system to adapt on-the-fly to the data and queries while providing robust behavior.



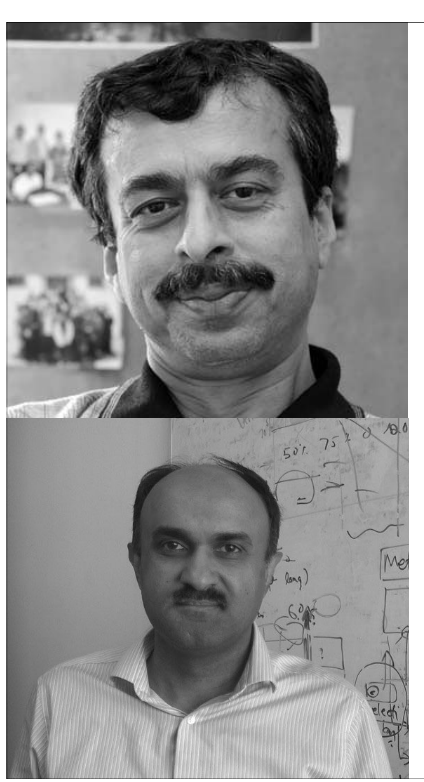
the "traditional" stack
(no ML, no synthesis)
PLAN
OPTIMIZER
KNOB TUNING
INDEX RECOMENDATIONS
INDEXING
STORAGE LAYOUTS
MID-FLIGHT ReOpt
GUY LOHMAN
DBA
INDEX ADVISORS
CRACKING
H20, NODB


the "traditional" stack
(no ML, no synthesis)
PLAN
OPTIMIZER
KNOB TUNING
INDEX RECOMENDATIONS
INDEXING
STORAGE LAYOUTS
MID-FLIGHT ReOpt
GUY LOHMAN
DBA
INDEX ADVISORS
CRACKING
H20, NODB

**Index Advisors (VLDB 1997)**

Given a workload sample:
Use the optimizer cost models to do what-if analysis

Index advisors is perhaps the best example of how existing database systems tailor their design to particular applications. By making it possible to support many different indexing schemes and by offering tools that make it possible to choose the most appropriate indexing scheme using knowledge of the workload, systems can adapt to the application. These ideas have been utilized widely for both offline and online adaptation and the latest research ideas are about experimenting with the utilization of ML for making the actual decisions.

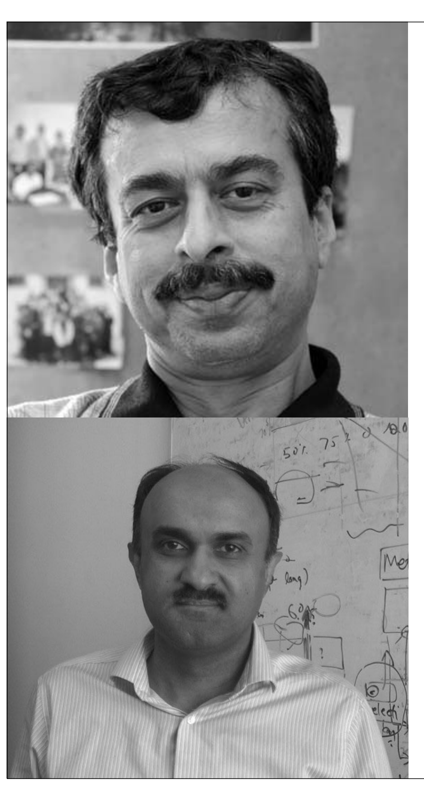


**Index Advisors (VLDB 1997)**

Given a workload sample:
Use the optimizer cost models to do what-if analysis

The database indexing semi-automatically
adapts to the workload (offline and online)

Index advisors is perhaps the best example of how existing database systems tailor their design to particular applications. By making it possible to support many different indexing schemes and by offering tools that make it possible to choose the most appropriate indexing scheme using knowledge of the workload, systems can adapt to the application. These ideas have been utilized widely for both offline and online adaptation and the latest research ideas are about experimenting with the utilization of ML for making the actual decisions.



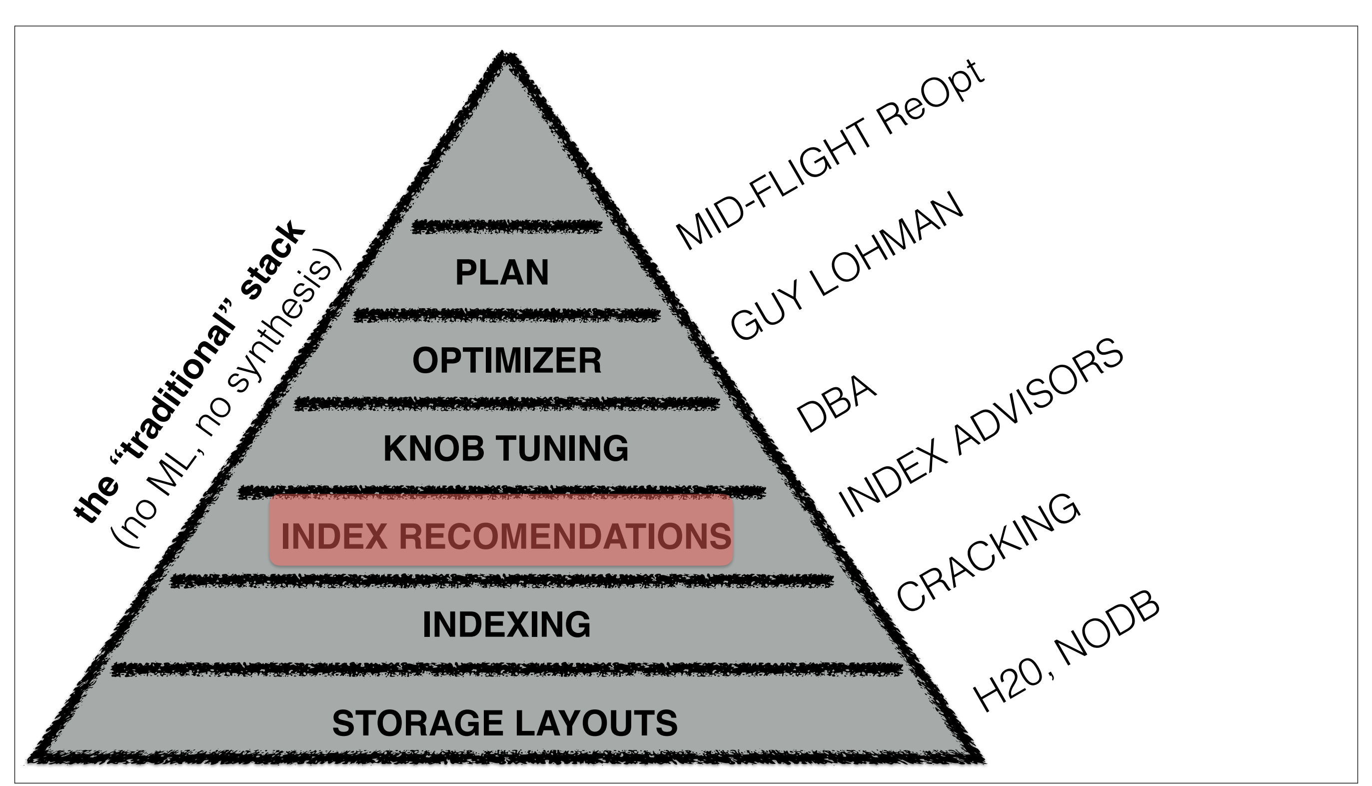
**Index Advisors (VLDB 1997)**

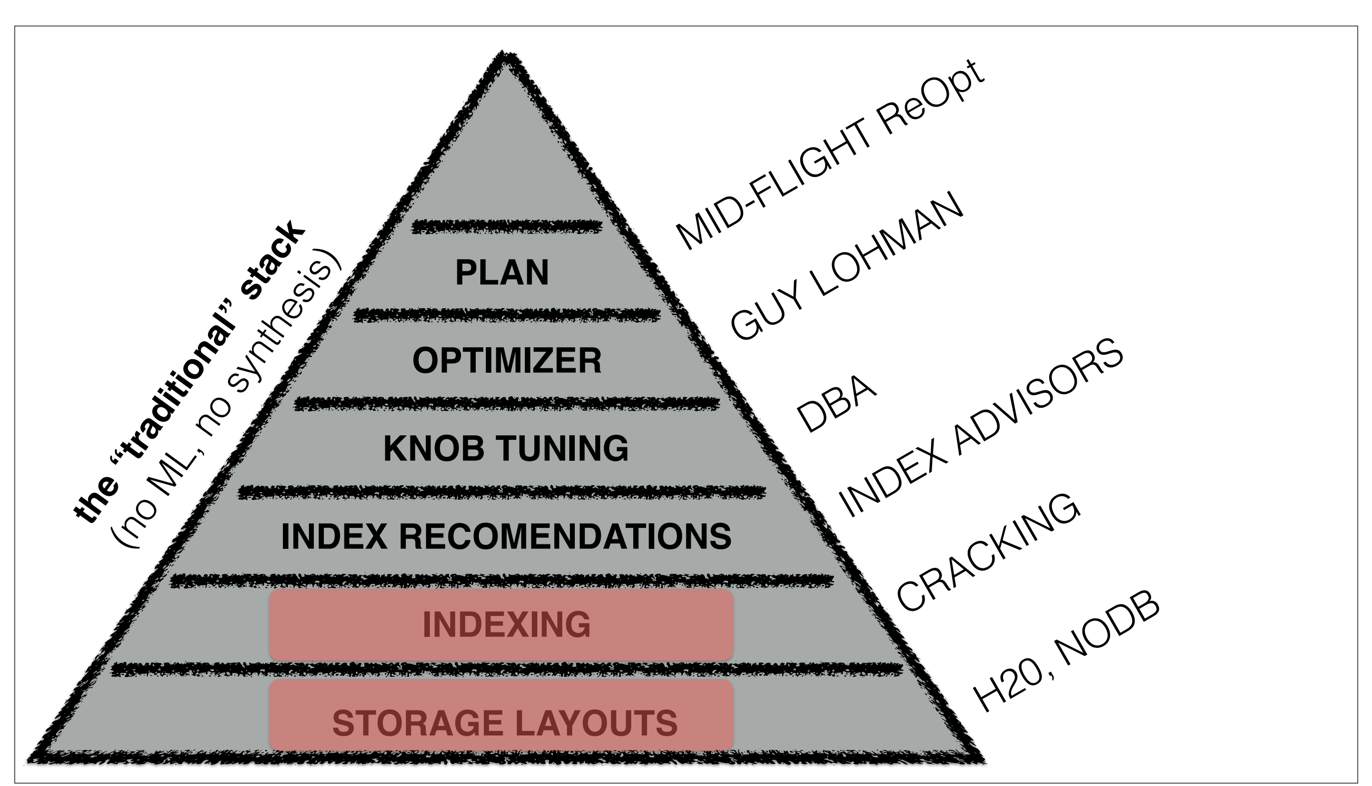
Given a workload sample:
Use the optimizer cost models to do what-if analysis

The database indexing semi-automatically
adapts to the workload (offline and online)

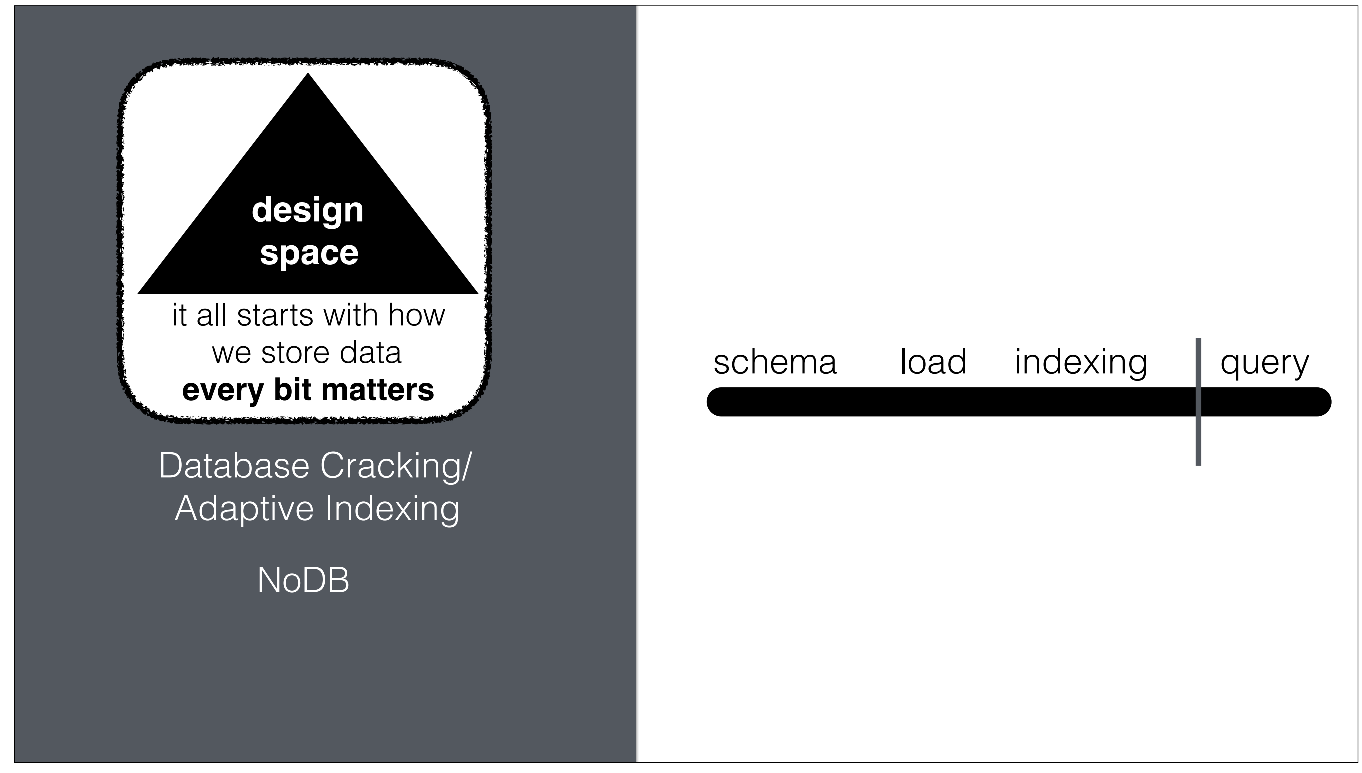
**SIGMOD 2019**: Using ML and past runs
to recommend indexes

Index advisors is perhaps the best example of how existing database systems tailor their design to particular applications. By making it possible to support many different indexing schemes and by offering tools that make it possible to choose the most appropriate indexing scheme using knowledge of the workload, systems can adapt to the application. These ideas have been utilized widely for both offline and online adaptation and the latest research ideas are about experimenting with the utilization of ML for making the actual decisions.

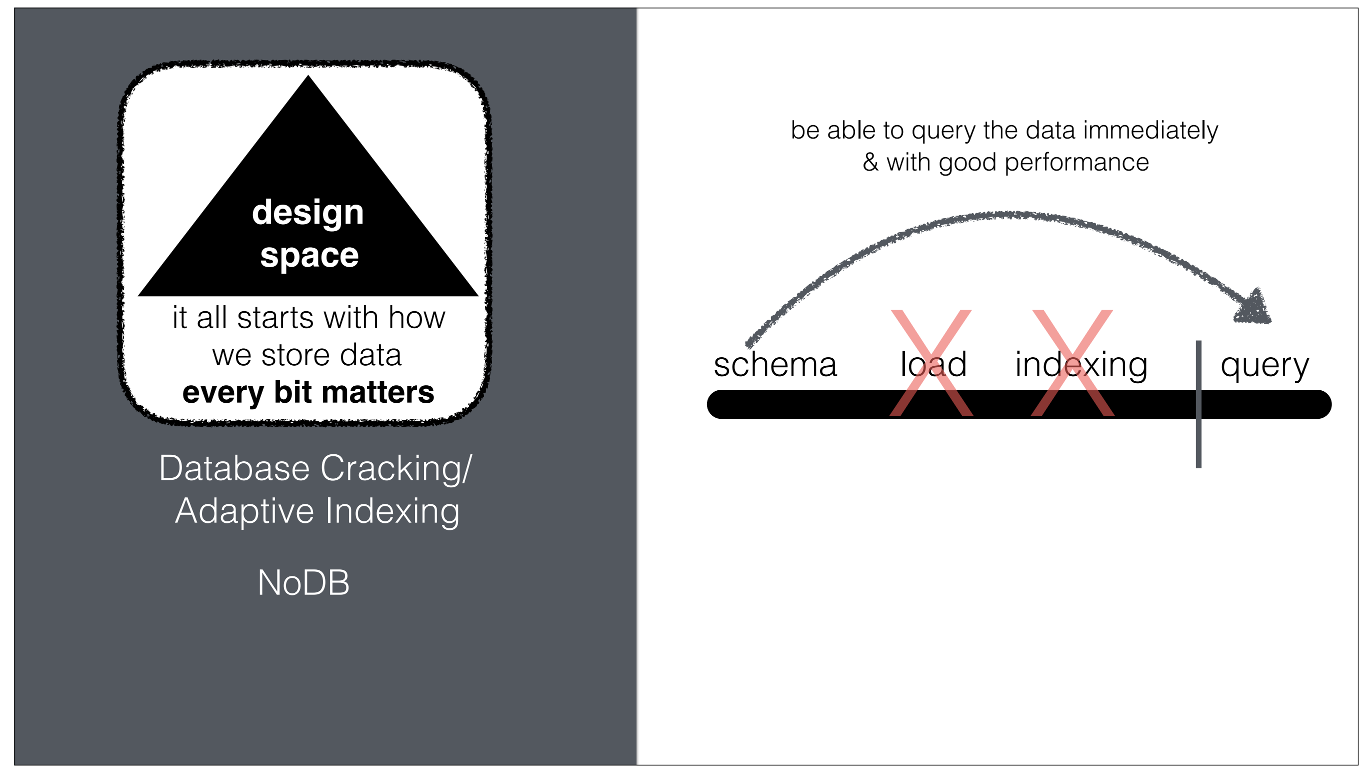

the "traditional" stack
(no ML, no synthesis)
PLAN
OPTIMIZER
KNOB TUNING
INDEX RECOMENDATIONS
INDEXING
STORAGE LAYOUTS
MID-FLIGHT ReOpt
GUY LOHMAN
DBA
INDEX ADVISORS
CRACKING
H20, NODB


the "traditional" stack
(no ML, no synthesis)
PLAN
OPTIMIZER
KNOB TUNING
INDEX RECOMENDATIONS
INDEXING
STORAGE LAYOUTS
MID-FLIGHT ReOpt
GUY LOHMAN
DBA
INDEX ADVISORS
CRACKING
H20, NODB

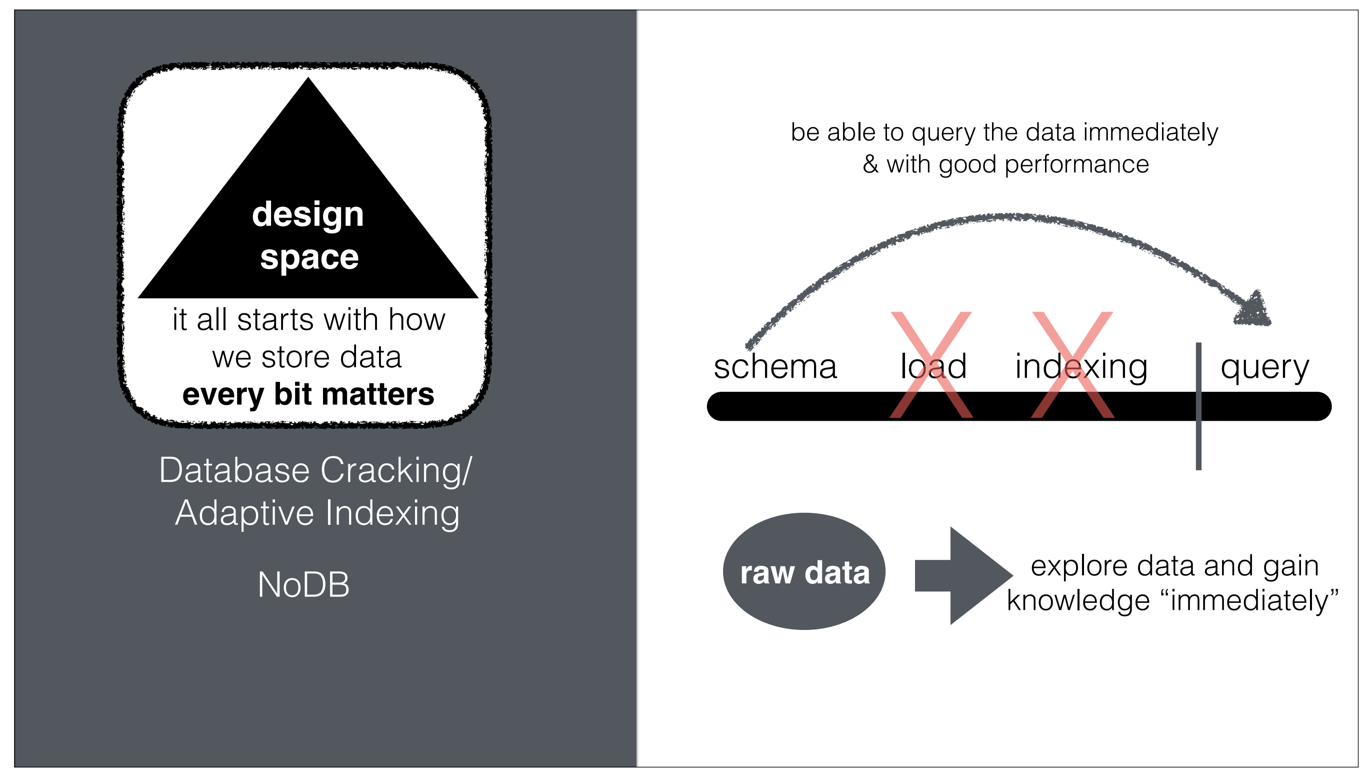
Even lower at the database system stack ideas such as db cracking, and NoDB, show how we can remove some of the static and critical decisions which typically happen offline by human DBAs (assisted by auto-tuning tools), and instead build systems that automate fully these decisions. DB cracking shows how we can build a system that automatically creates the right set of indexes without any human supervision, while NoDB shows how we can build a system that automatically loads only the needed part of the data into the system. The point here is not only the decision itself (e.g., the right index or data to load), but also to make sure that the system runs smoothly with minimum overhead compared to a fully tuned system and that we can start using the system for queries immediately without having to wait for any tuning or preparation steps.



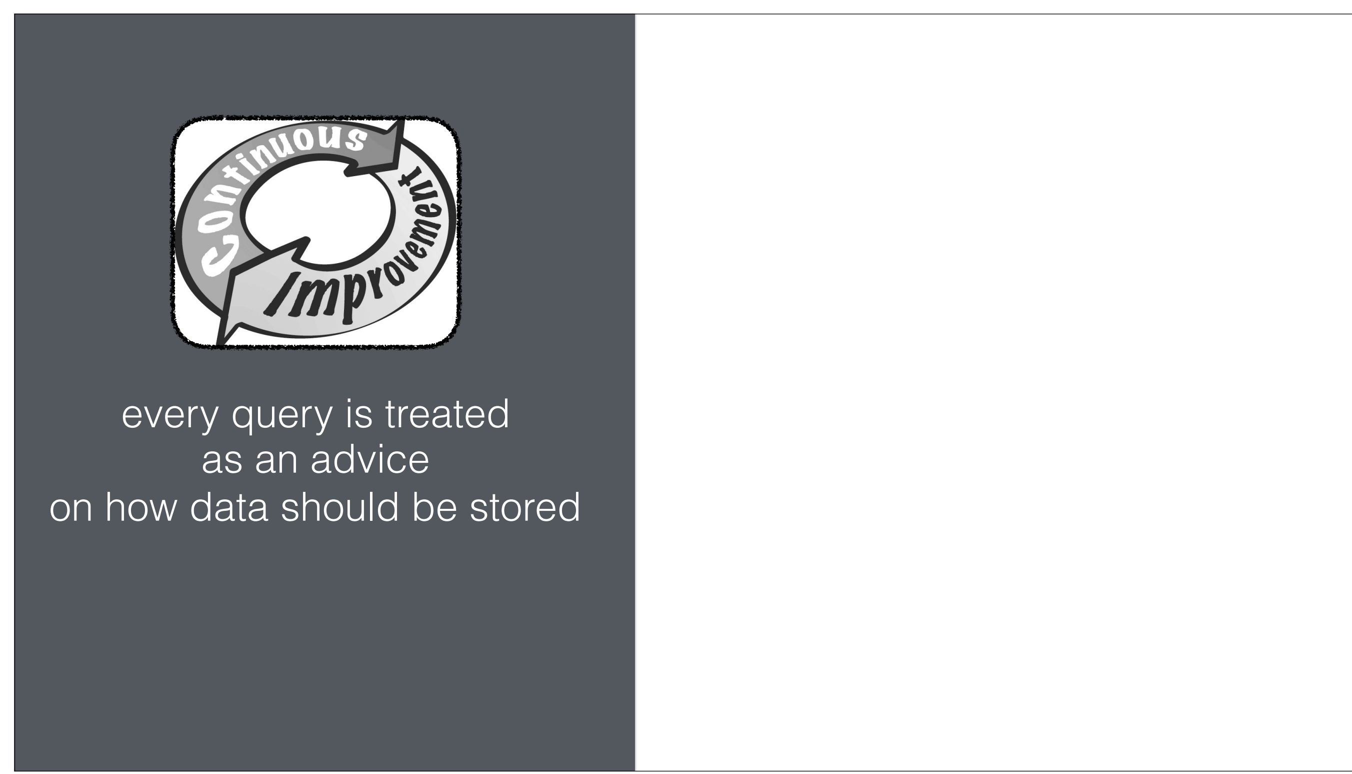

Even lower at the database system stack ideas such as db cracking, and NoDB, show how we can remove some of the static and critical decisions which typically happen offline by human DBAs (assisted by auto-tuning tools), and instead build systems that automate fully these decisions. DB cracking shows how we can build a system that automatically creates the right set of indexes without any human supervision, while NoDB shows how we can build a system that automatically loads only the needed part of the data into the system. The point here is not only the decision itself (e.g., the right index or data to load), but also to make sure that the system runs smoothly with minimum overhead compared to a fully tuned system and that we can start using the system for queries immediately without having to wait for any tuning or preparation steps.




Even lower at the database system stack ideas such as db cracking, and NoDB, show how we can remove some of the static and critical decisions which typically happen offline by human DBAs (assisted by auto-tuning tools), and instead build systems that automate fully these decisions. DB cracking shows how we can build a system that automatically creates the right set of indexes without any human supervision, while NoDB shows how we can build a system that automatically loads only the needed part of the data into the system. The point here is not only the decision itself (e.g., the right index or data to load), but also to make sure that the system runs smoothly with minimum overhead compared to a fully tuned system and that we can start using the system for queries immediately without having to wait for any tuning or preparation steps.

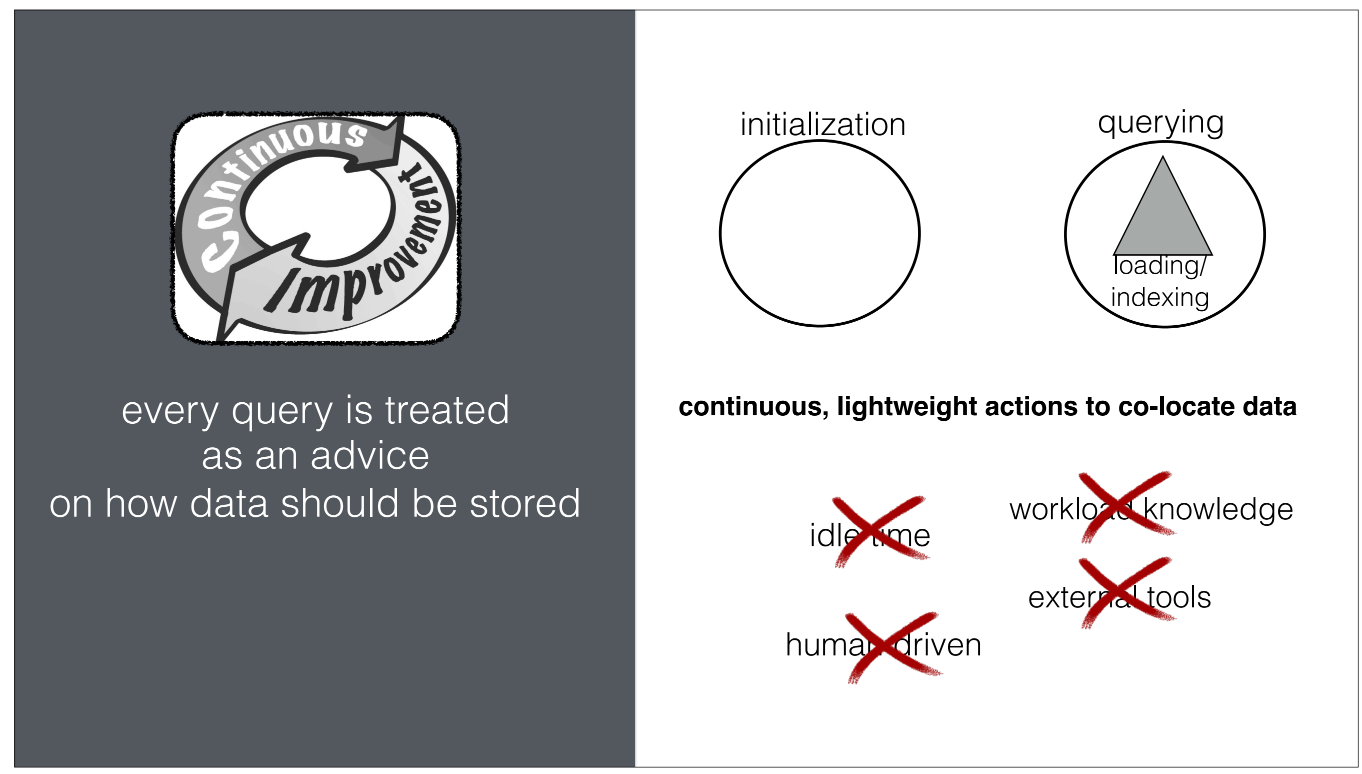



Even lower at the database system stack ideas such as db cracking, and NoDB, show how we can remove some of the static and critical decisions which typically happen offline by human DBAs (assisted by auto-tuning tools), and instead build systems that automate fully these decisions. DB cracking shows how we can build a system that automatically creates the right set of indexes without any human supervision, while NoDB shows how we can build a system that automatically loads only the needed part of the data into the system. The point here is not only the decision itself (e.g., the right index or data to load), but also to make sure that the system runs smoothly with minimum overhead compared to a fully tuned system and that we can start using the system for queries immediately without having to wait for any tuning or preparation steps.

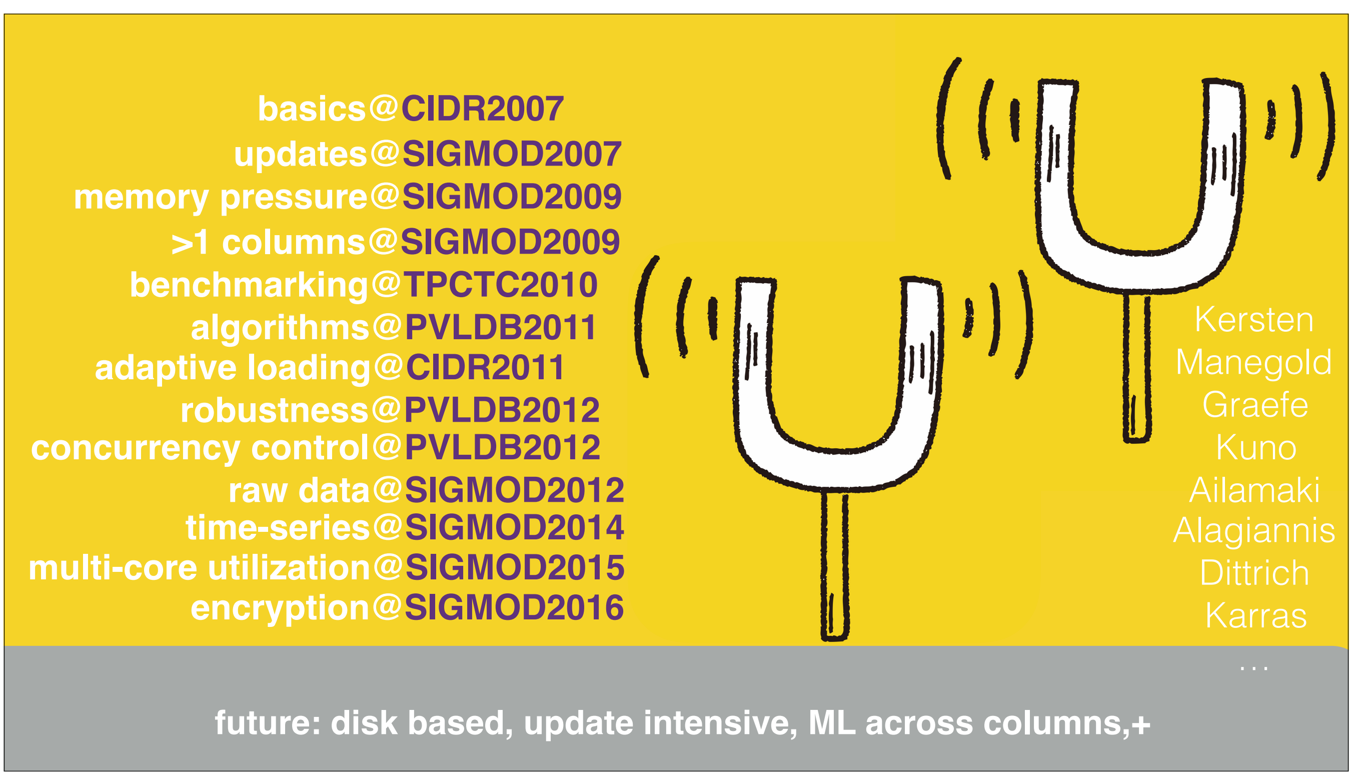



The primary trick among these lines of work is that there is zero action taken during initialization time. This means that we can start querying a system without doing any data loading and any indexing. And then as we use the system, it incrementally becomes better and better.

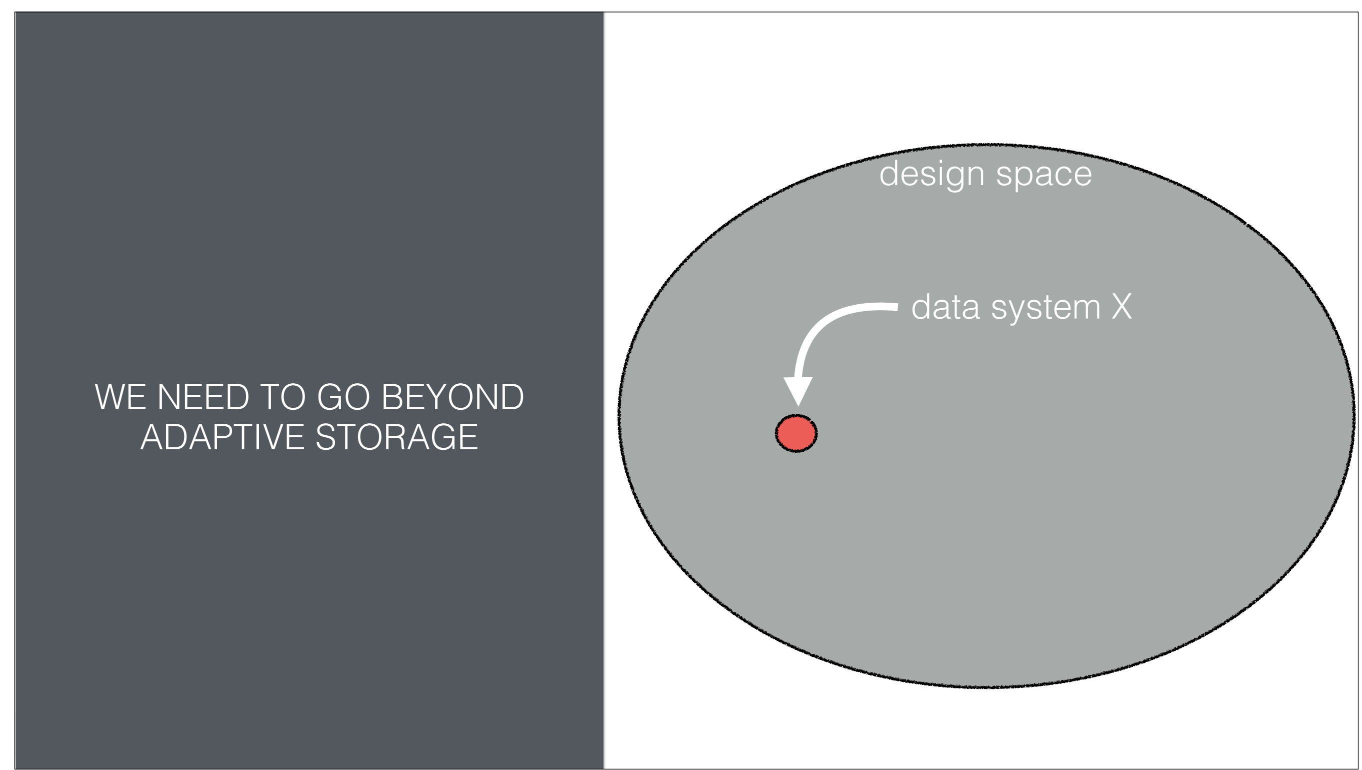
The primary trick among these lines of work is that there is zero action taken during initialization time. This means that we can start querying a system without doing any data loading and any indexing. And then as we use the system, it incrementally becomes better and better.



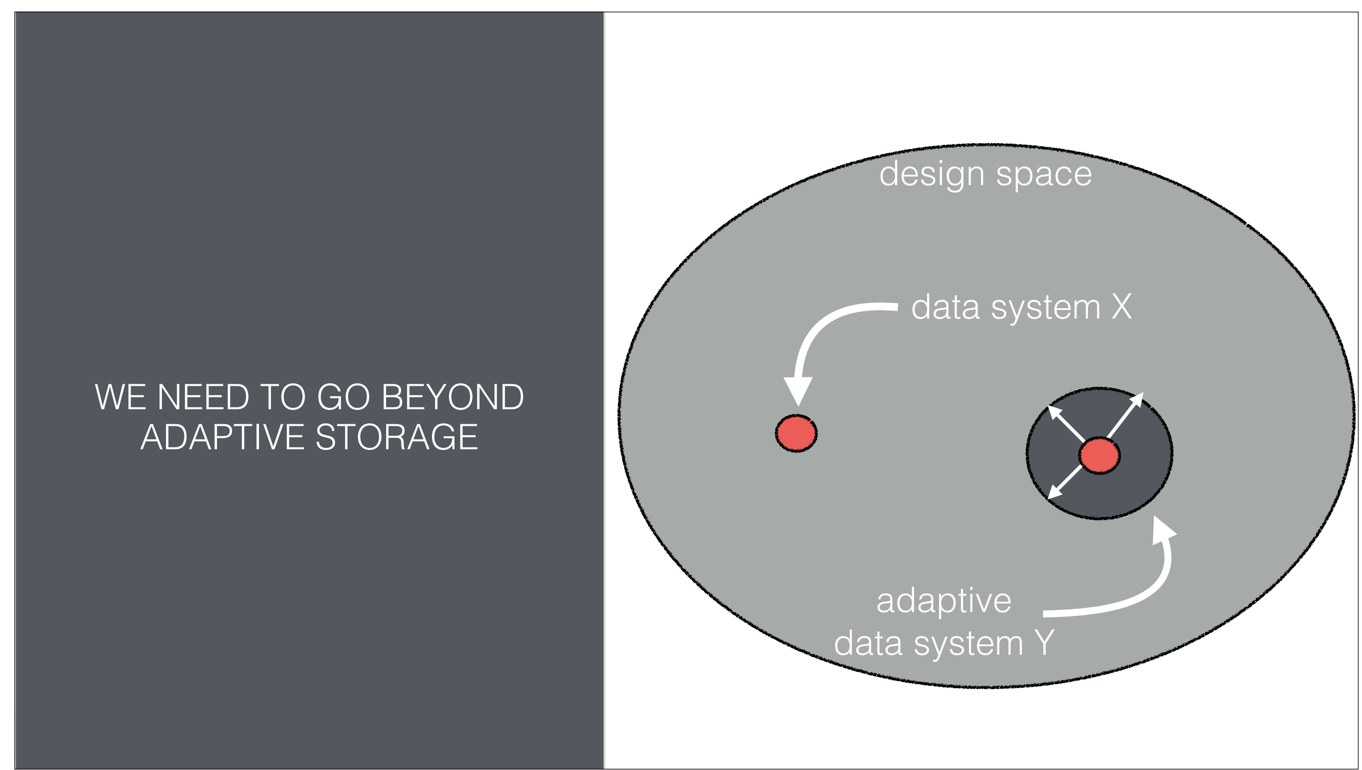

The primary trick among these lines of work is that there is zero action taken during initialization time. This means that we can start querying a system without doing any data loading and any indexing. And then as we use the system, it incrementally becomes better and better.



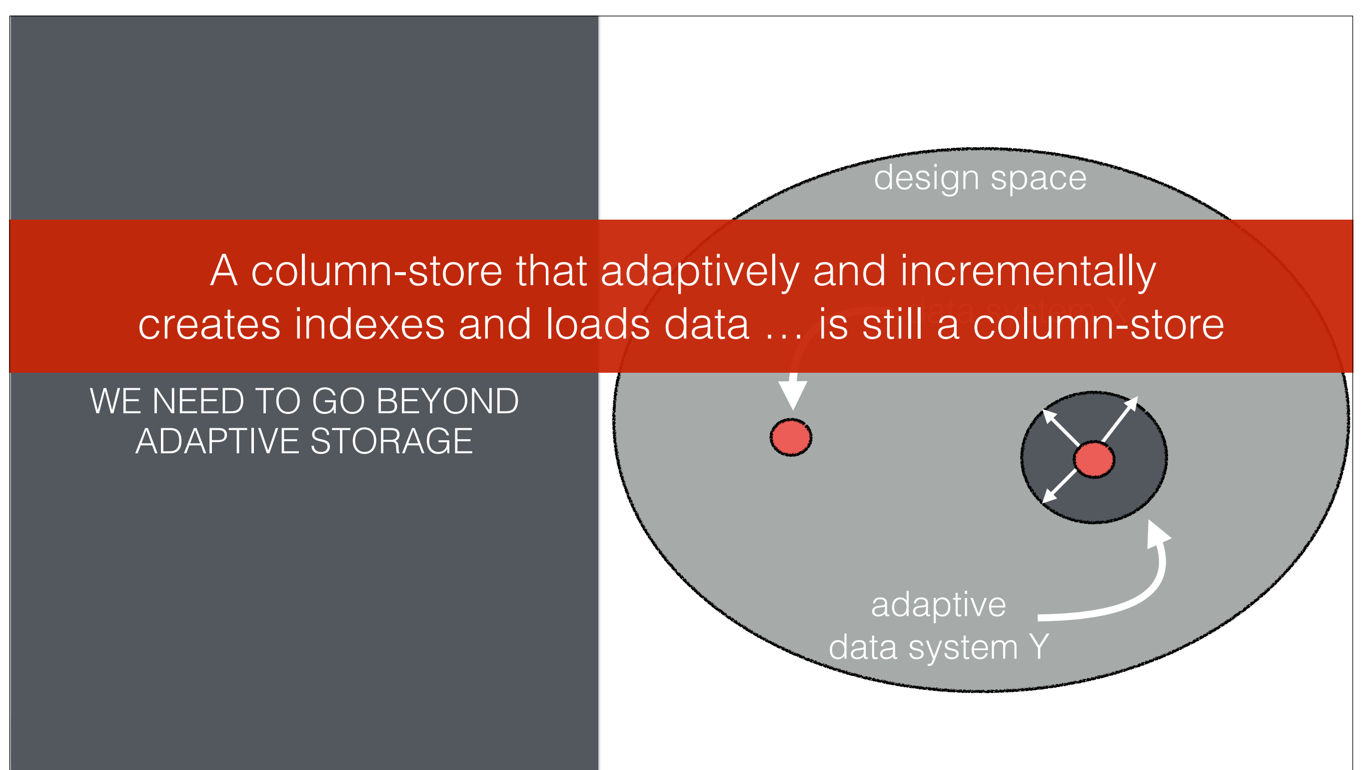

From a technical point of view the main idea is that data is continuously physically reorganized to match the ideal layout that incoming queries dictate. This means that every read query is turned into a write query which implies several side-effects for system design. There is a plethora of works over the past decade that addresses many of those issues. For example, how to reorganize data in an efficient way, how to support updates, and concurrent operations, etc.



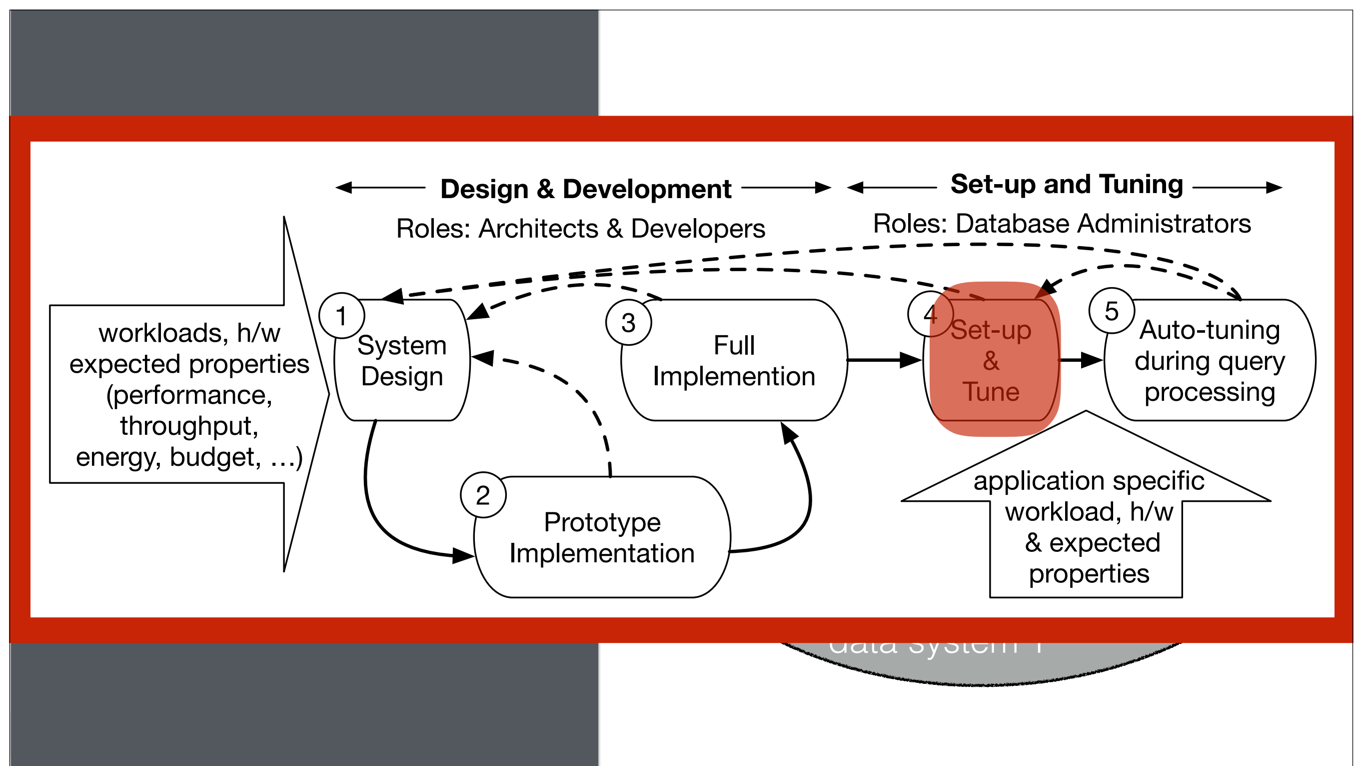

Even with all existing ideas, a data system is locked into a design “class”. For example even with adaptive loading and adaptive indexing enabled, a column-store would still be a column-store and so it would have specific performance behavior. The ultimate vision is to be able to affect the design of the system itself at its core. An adaptive system is indeed much more flexible but at the end of the day it covers a small part of the possible design space.



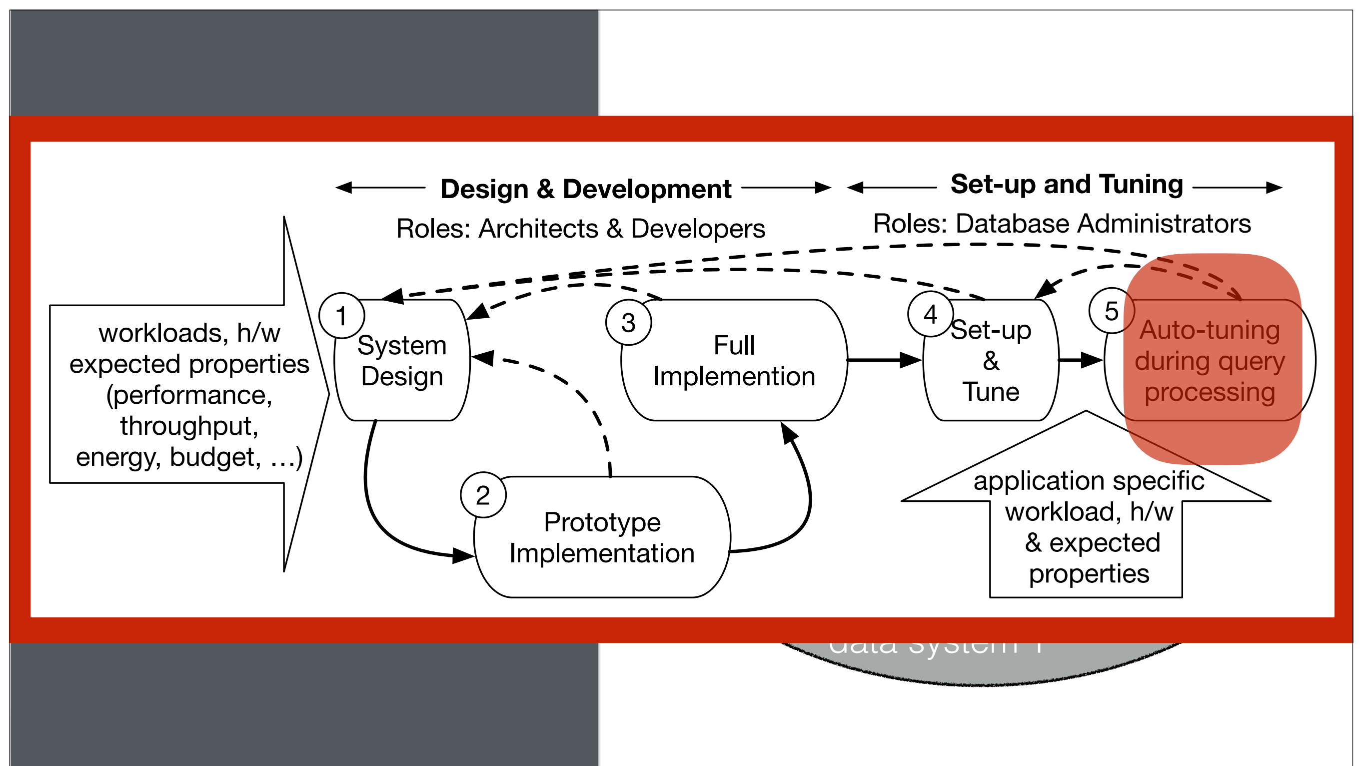

Even with all existing ideas, a data system is locked into a design “class”. For example even with adaptive loading and adaptive indexing enabled, a column-store would still be a column-store and so it would have specific performance behavior. The ultimate vision is to be able to affect the design of the system itself at its core. An adaptive system is indeed much more flexible but at the end of the day it covers a small part of the possible design space.



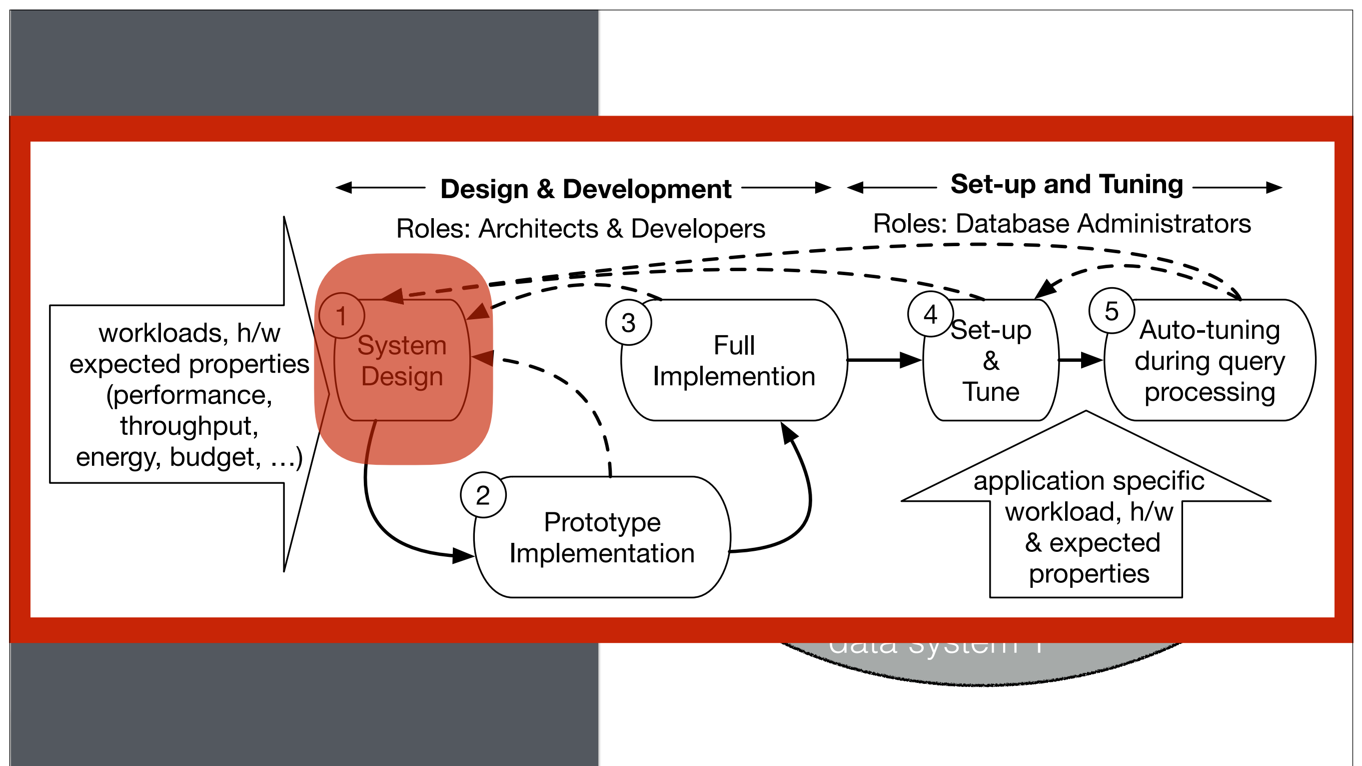

Even with all existing ideas, a data system is locked into a design “class”. For example even with adaptive loading and adaptive indexing enabled, a column-store would still be a column-store and so it would have specific performance behavior. The ultimate vision is to be able to affect the design of the system itself at its core. An adaptive system is indeed much more flexible but at the end of the day it covers a small part of the possible design space.



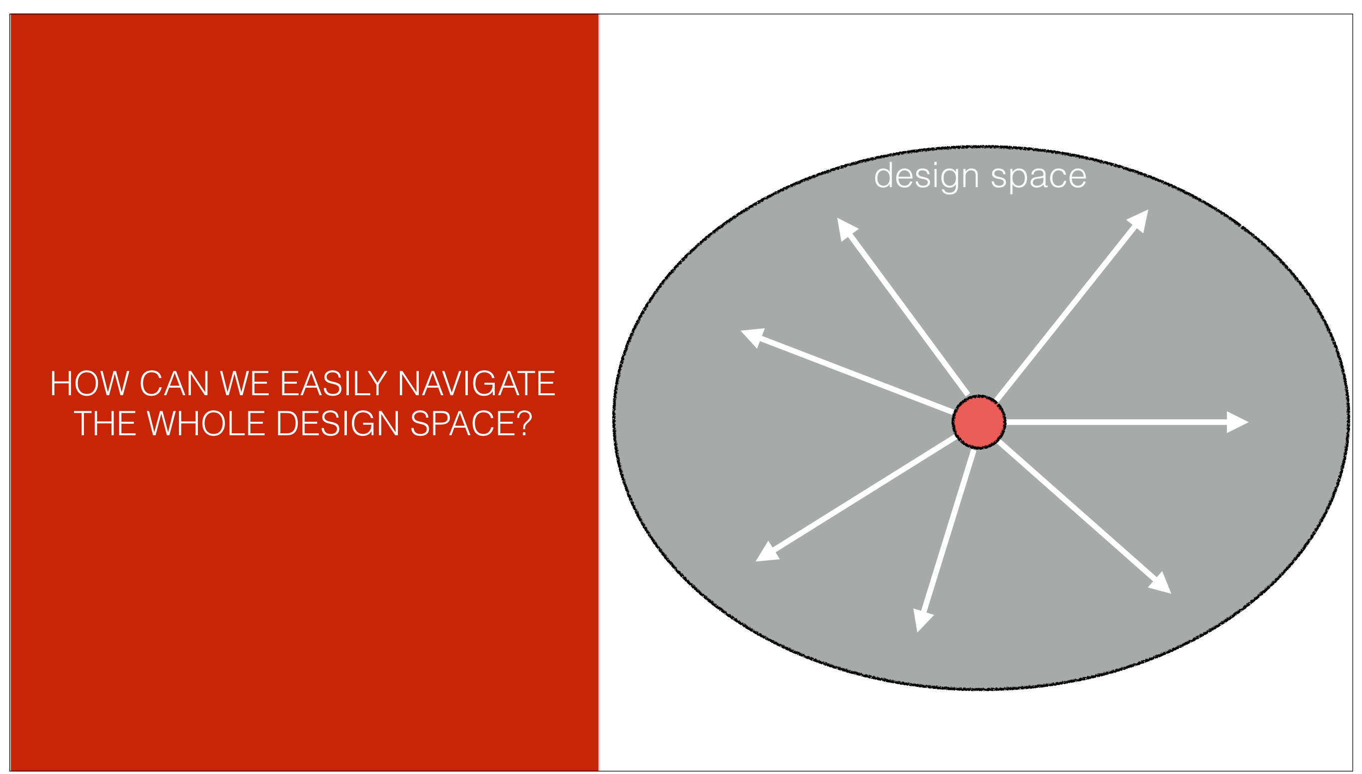

Even with all existing ideas, a data system is locked into a design “class”. For example even with adaptive loading and adaptive indexing enabled, a column-store would still be a column-store and so it would have specific performance behavior. The ultimate vision is to be able to affect the design of the system itself at its core. An adaptive system is indeed much more flexible but at the end of the day it covers a small part of the possible design space.



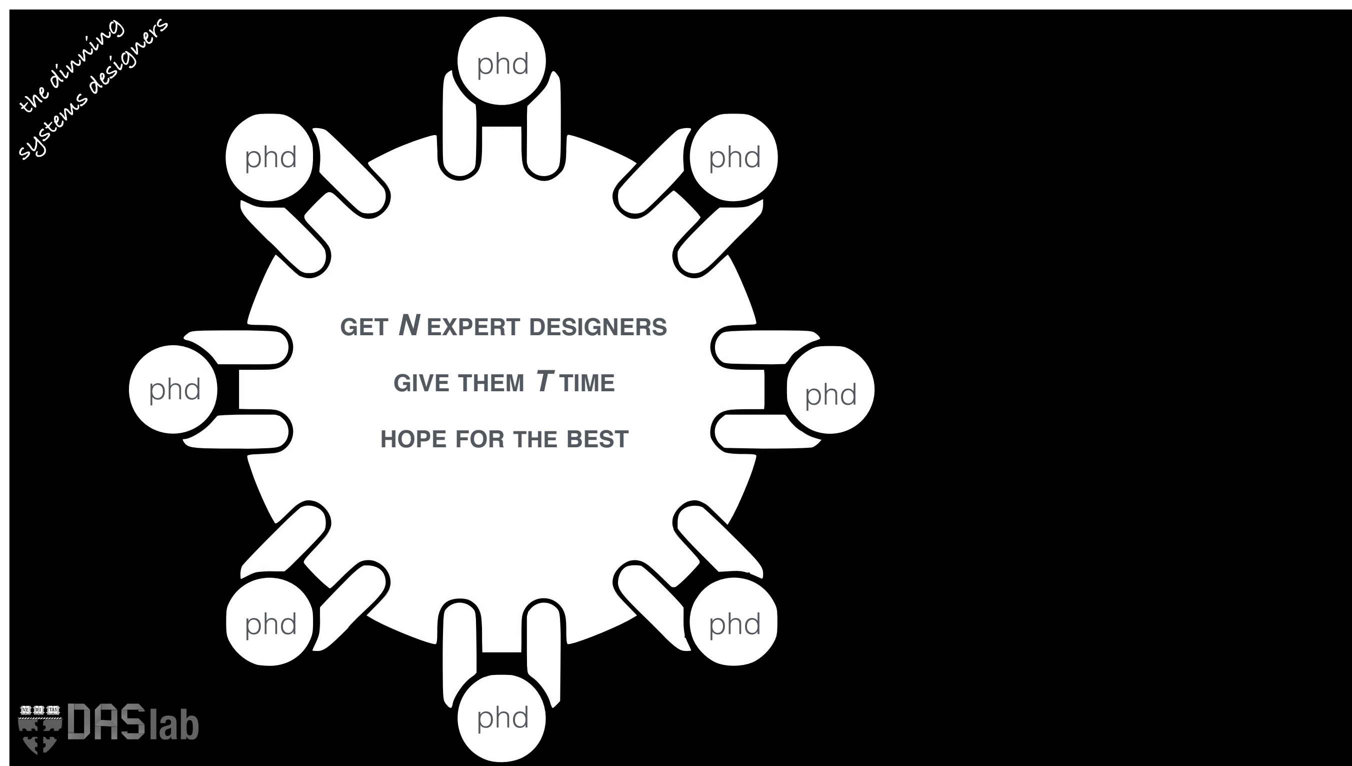

Even with all existing ideas, a data system is locked into a design "class". For example even with adaptive loading and adaptive indexing enabled, a column-store would still be a column-store and so it would have specific performance behavior. The ultimate vision is to be able to affect the design of the system itself at its core. An adaptive system is indeed much more flexible but at the end of the day it covers a small part of the possible design space.



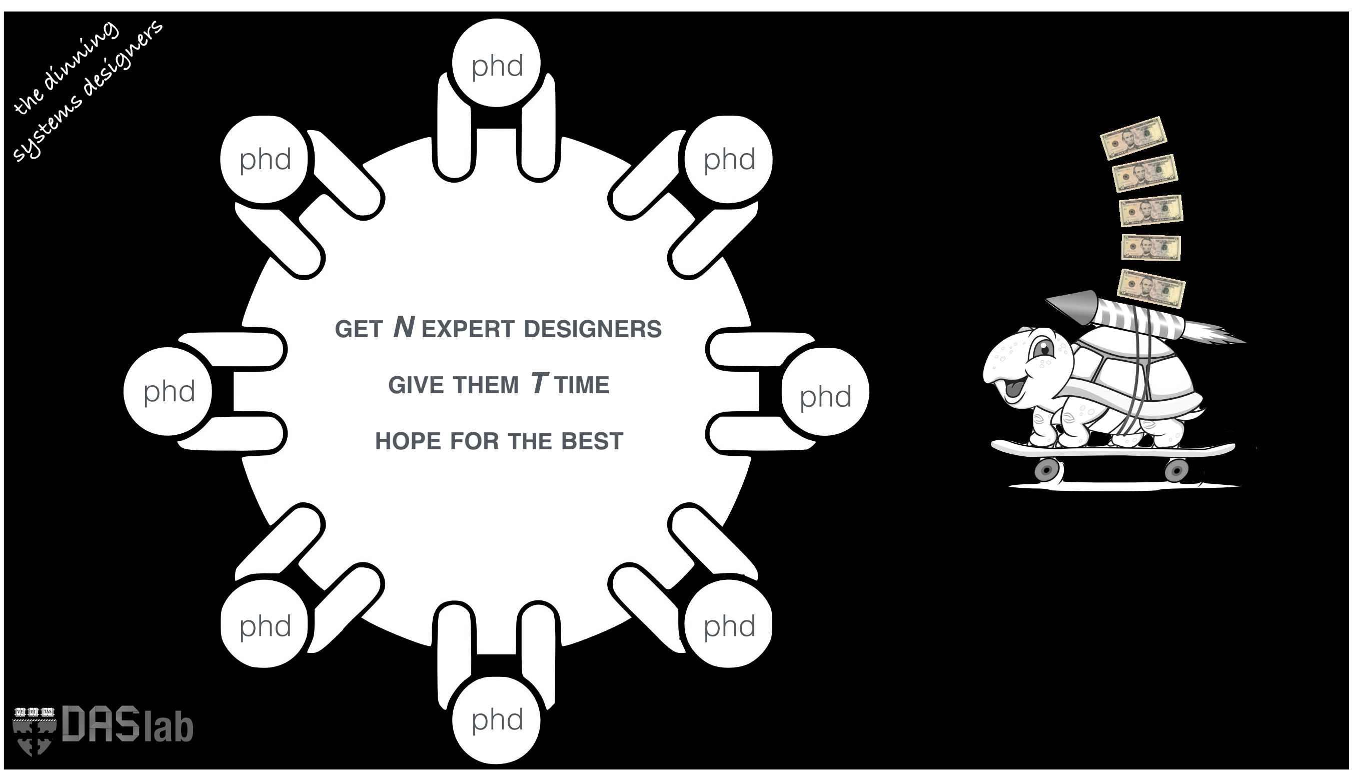

Even with all existing ideas, a data system is locked into a design “class”. For example even with adaptive loading and adaptive indexing enabled, a column-store would still be a column-store and so it would have specific performance behavior. The ultimate vision is to be able to affect the design of the system itself at its core. An adaptive system is indeed much more flexible but at the end of the day it covers a small part of the possible design space.




Ideally we would be able to cover the whole design space and create systems that can adapt to any scenario.

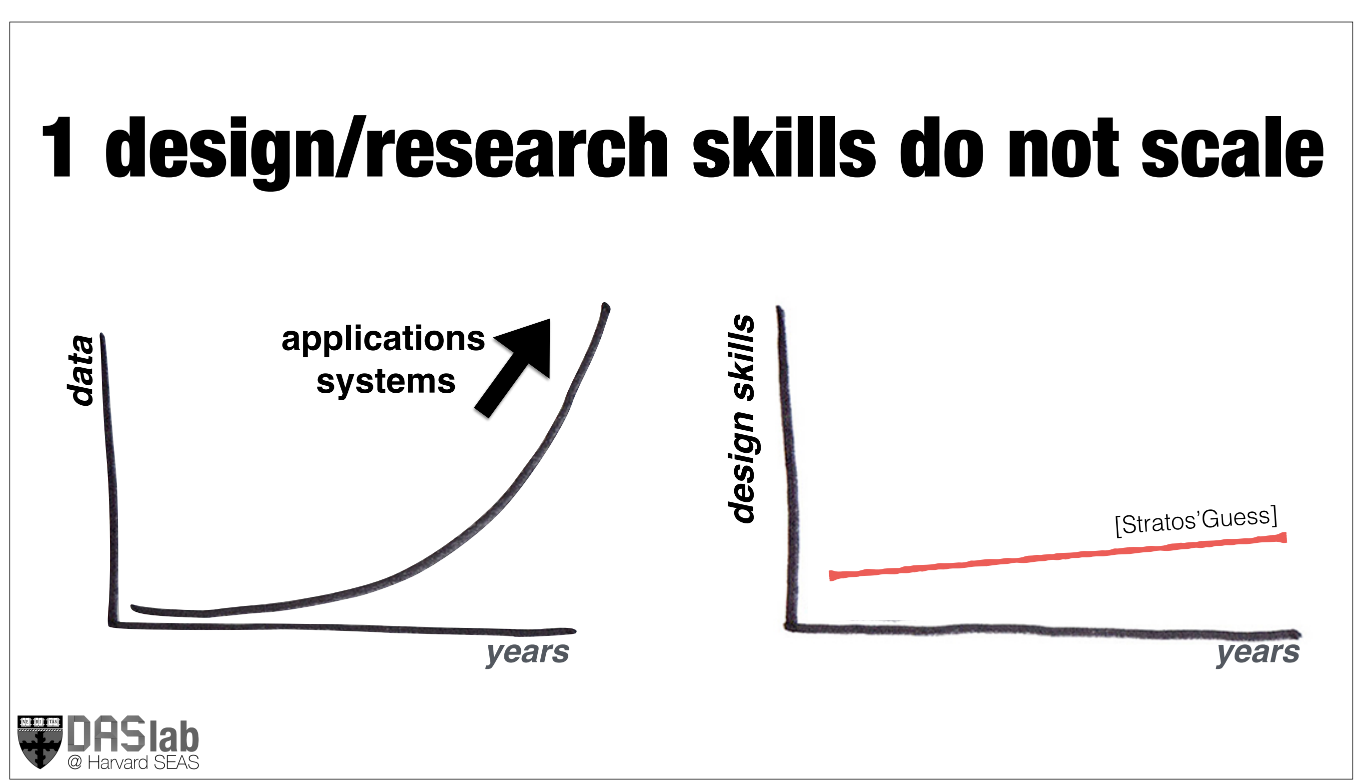



How do we do it now? We design systems manually. It takes many years. It is slow and expensive.

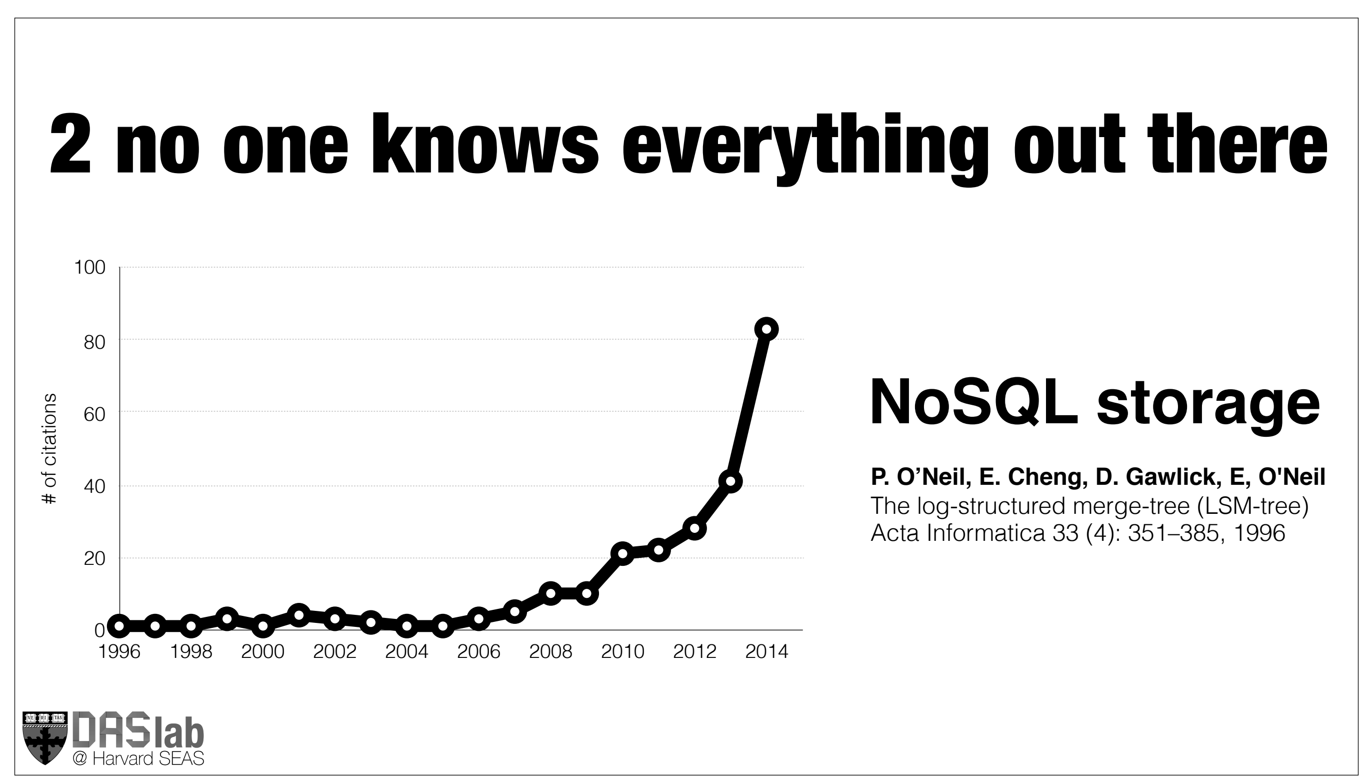



How do we do it now? We design systems manually. It takes many years. It is slow and expensive.

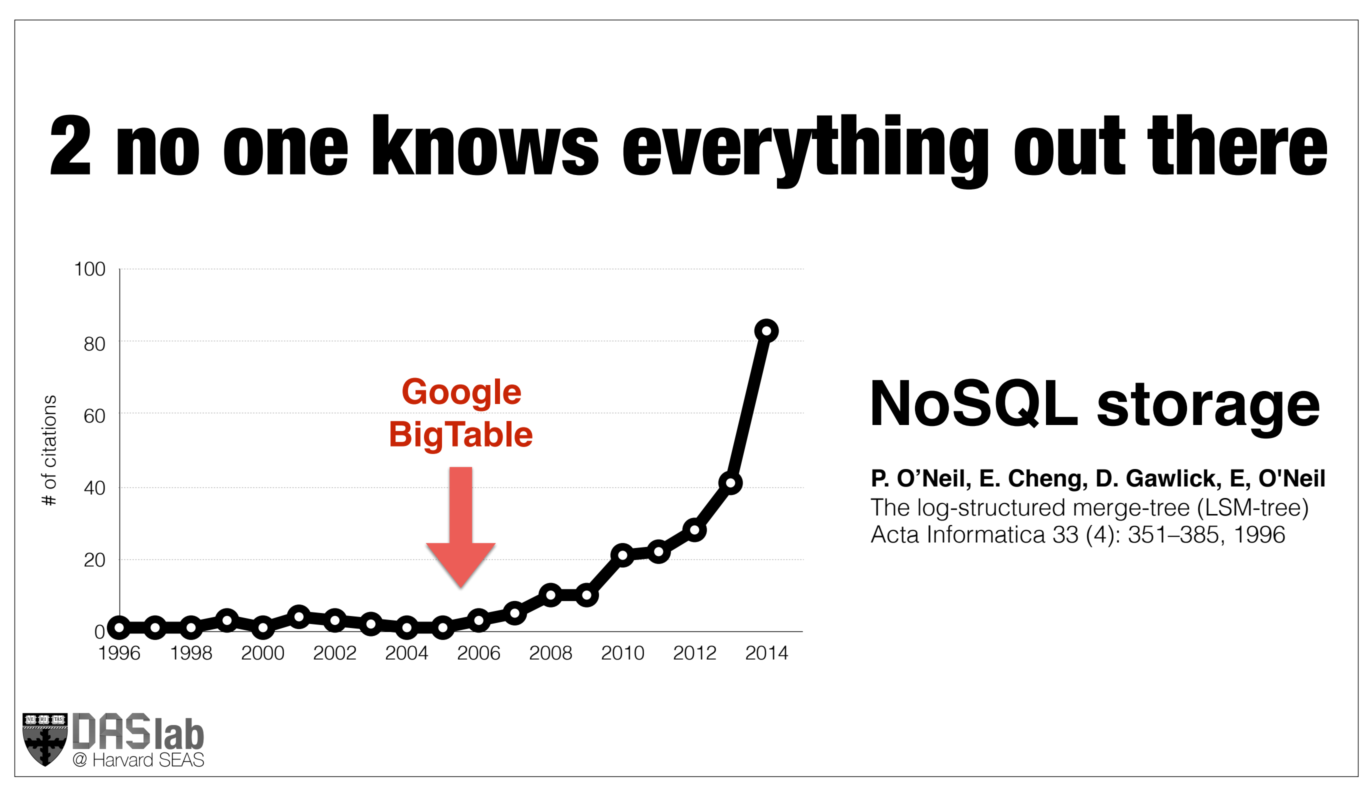
How do we do it now? We design systems manually. It takes many years. It is slow and expensive.




Manual design is not sustainable long term to keep up with the needs of new applications.




There are many critical issues with manual design. One of the biggest issues is that even for experts on a particular field it is very hard to really know everything that has been invented, everything that is possible. This problem gets increasingly harder as time goes by and more research is developed.

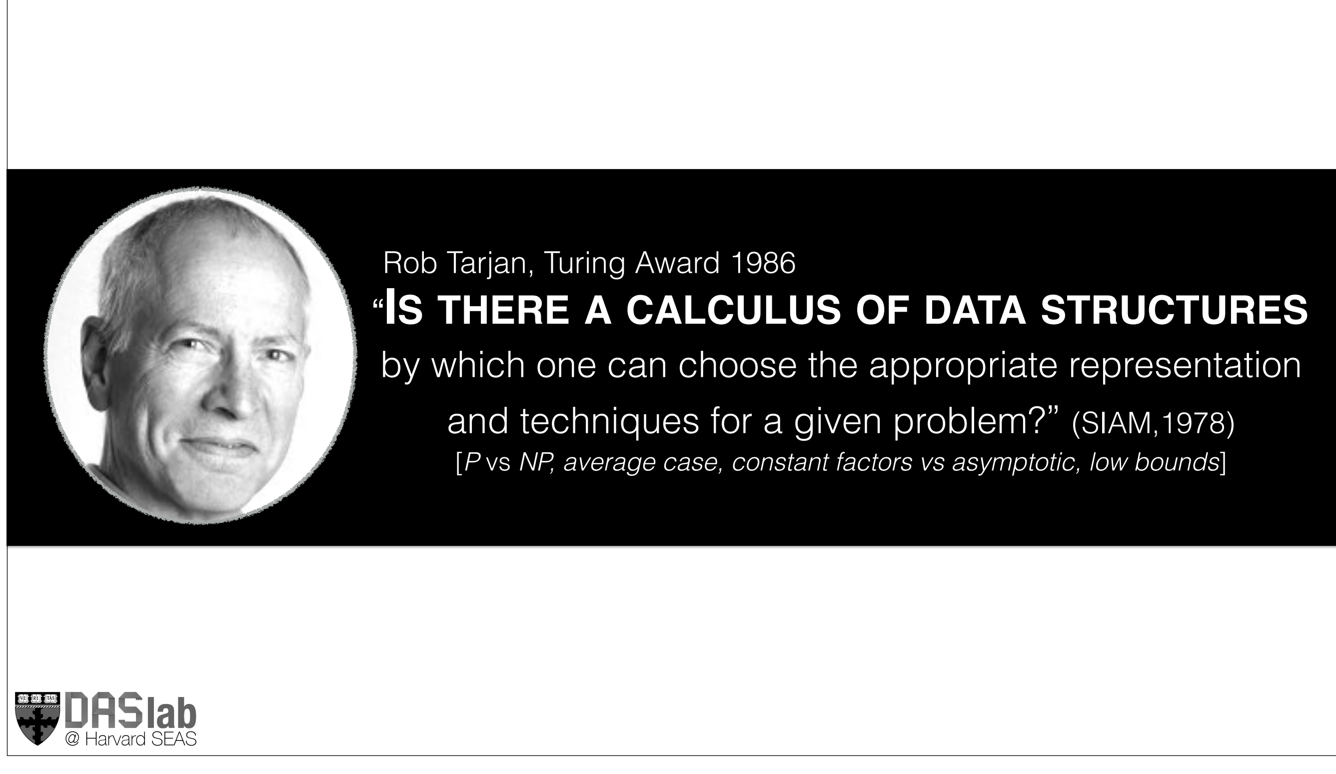



There are many critical issues with manual design. One of the biggest issues is that even for experts on a particular field it is very hard to really know everything that has been invented, everything that is possible. This problem gets increasingly harder as time goes by and more research is developed.

# AUTO DESIGN

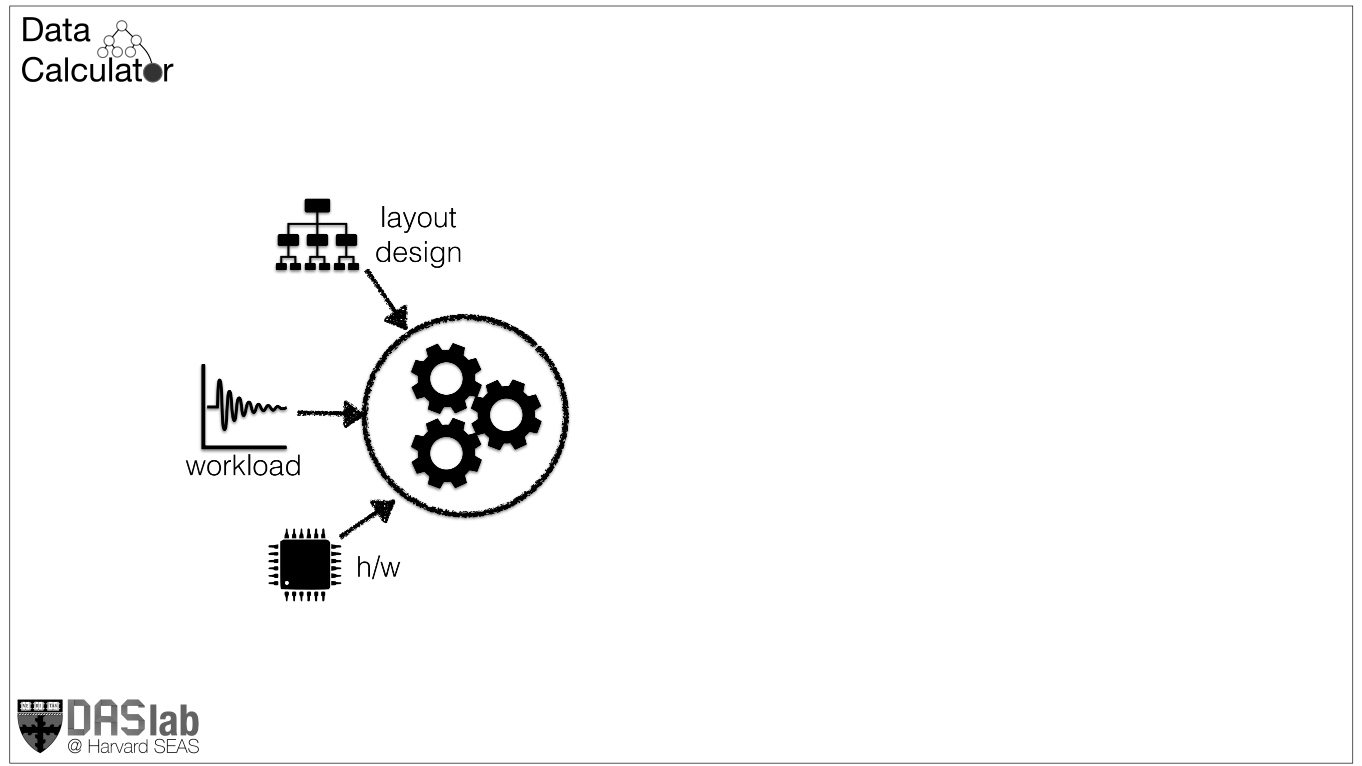
**optimal designs, fast design process, minimize cost of the design process**

Ideally we would be able to design data structures automatically for the problem at hand. And this would make it easier to design tailored systems as well. Well, this is an old vision. The most elegant description of this goal is by Rob Tarjan, who argued for a “Calculus of Data Structures” as one of the five big challenges for computer science.

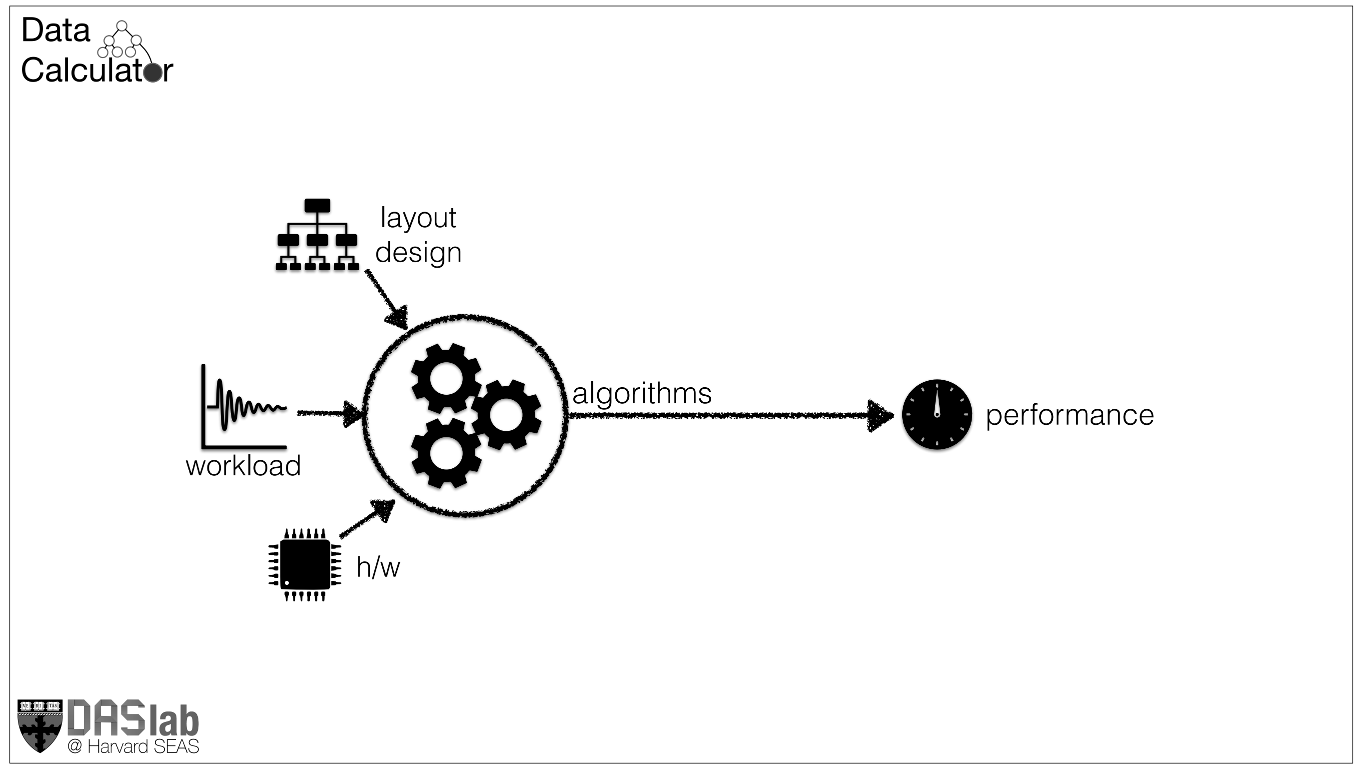



Ideally we would be able to design data structures automatically for the problem at hand. And this would make it easier to design tailored systems as well. Well, this is an old vision. The most elegant description of this goal is by Rob Tarjan, who argued for a “Calculus of Data Structures” as one of the five big challenges for computer science.

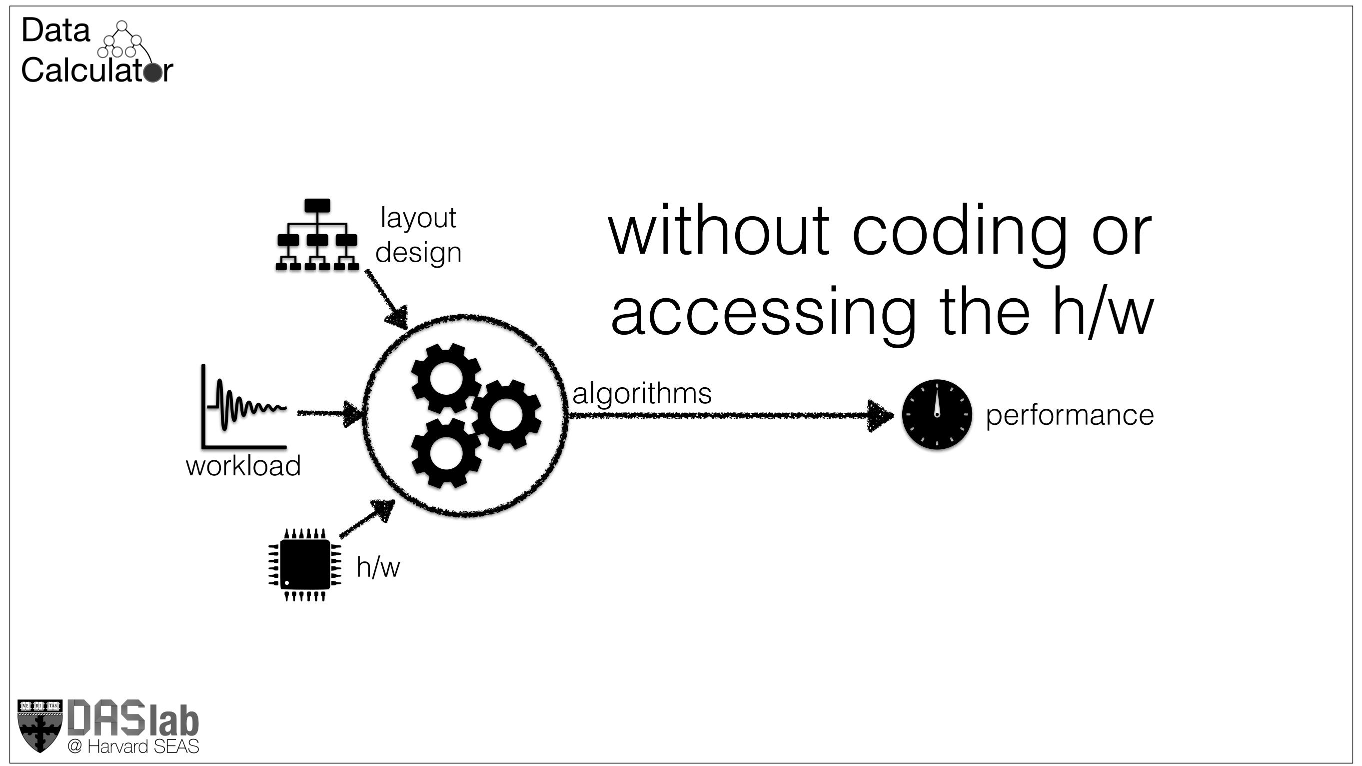
At Harvard we have been working for the last 6 years we have been focusing on exactly this problem with an end goal being a new generation of systems that can transition their data layout in between fundamentally different data structure designs to match the data, queries, and hardware. We call this self-designing systems because the kind of changes allowed are changes that normally happen at design time manually.



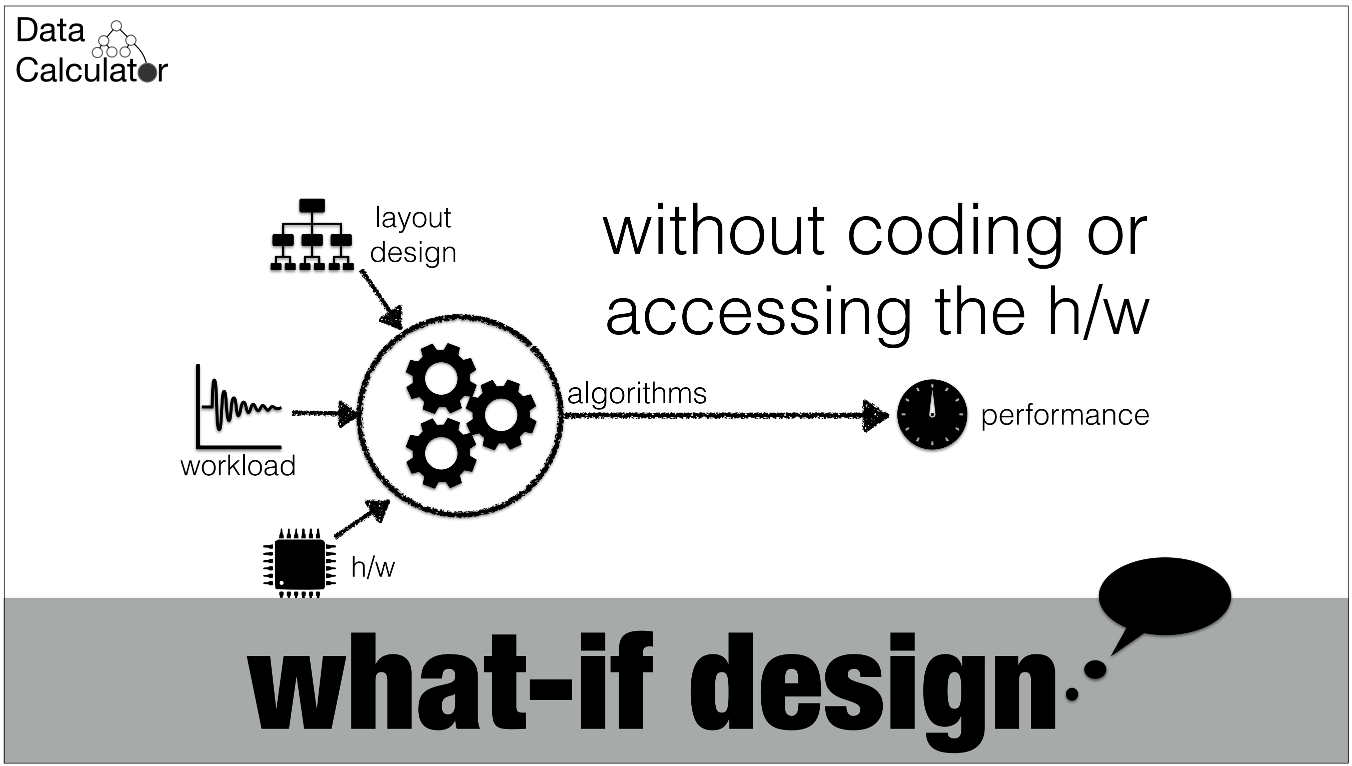

As a first step in this direction, we built an engine, which we call the Data Calculator and which takes as input the hardware, workload and layout of a data structure. It then computes automatically the algorithms that this data structure design needs to optimally process the workload on this hardware and it also computes the performance. That is, the response time that an actual implementation of this design would need to run this workload on this hardware. However, all this happens without the user having to implement anything and without even needing access to the actual hardware. Given this engine we show that we can start thinking about game-changing paradigms for system designs such as interactive design, self-designing systems, and fully automatic design for instance optimal systems.



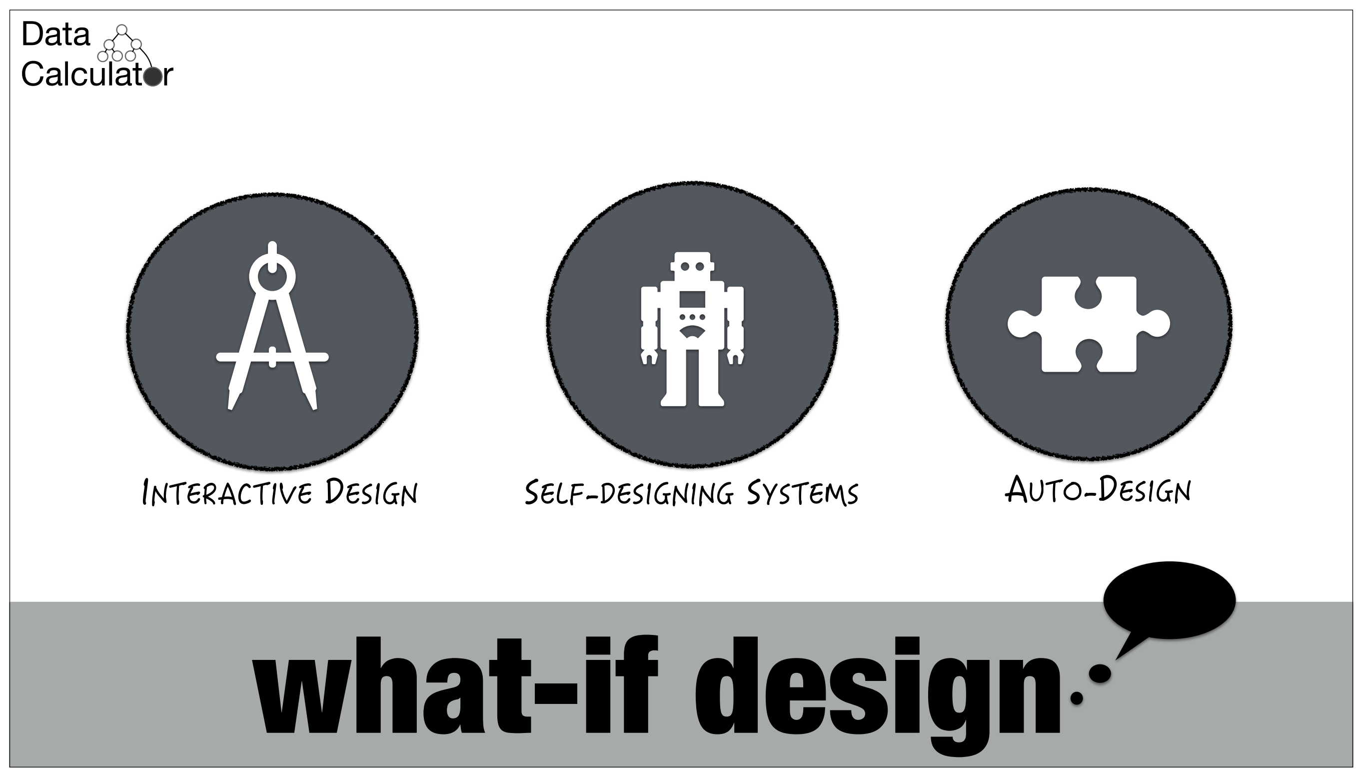

As a first step in this direction, we built an engine, which we call the Data Calculator and which takes as input the hardware, workload and layout of a data structure. It then computes automatically the algorithms that this data structure design needs to optimally process the workload on this hardware and it also computes the performance. That is, the response time that an actual implementation of this design would need to run this workload on this hardware. However, all this happens without the user having to implement anything and without even needing access to the actual hardware. Given this engine we show that we can start thinking about game-changing paradigms for system designs such as interactive design, self-designing systems, and fully automatic design for instance optimal systems.



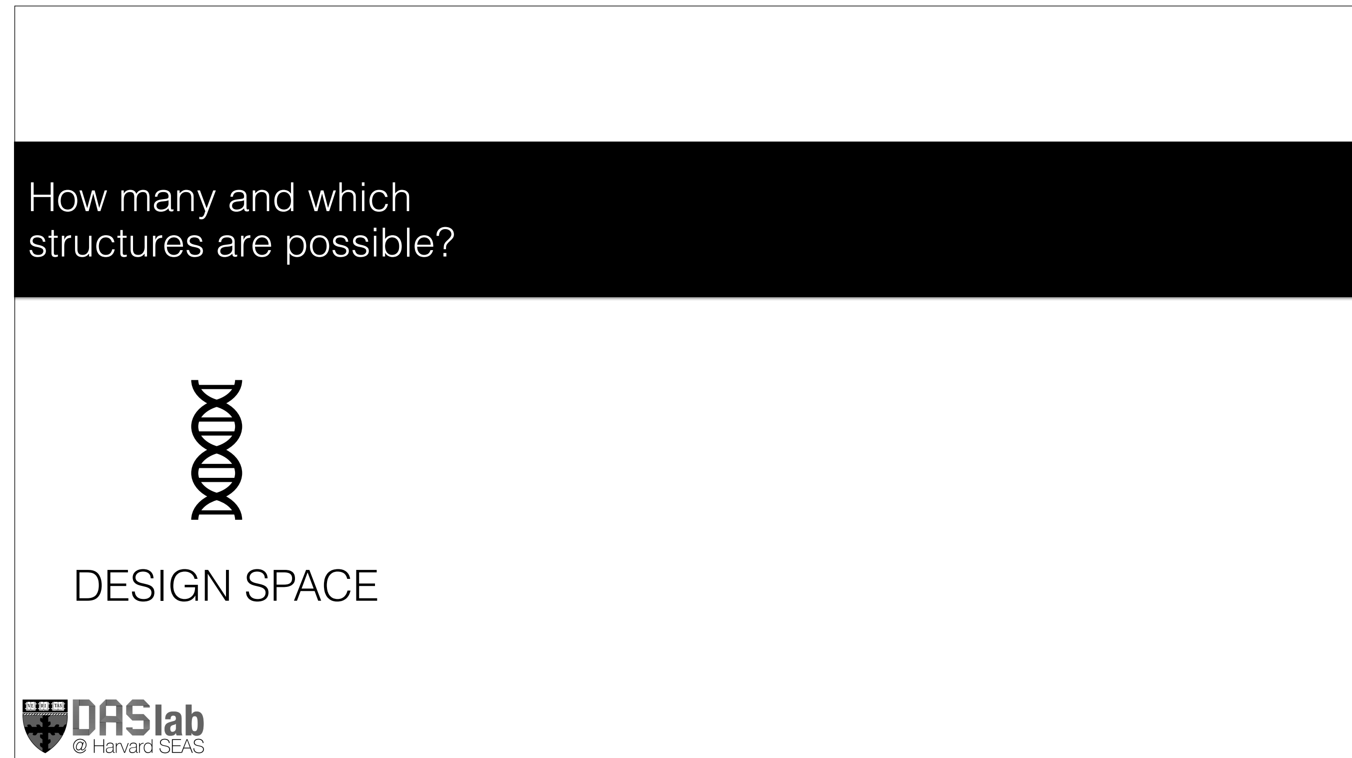

As a first step in this direction, we built an engine, which we call the Data Calculator and which takes as input the hardware, workload and layout of a data structure. It then computes automatically the algorithms that this data structure design needs to optimally process the workload on this hardware and it also computes the performance. That is, the response time that an actual implementation of this design would need to run this workload on this hardware. However, all this happens without the user having to implement anything and without even needing access to the actual hardware. Given this engine we show that we can start thinking about game-changing paradigms for system designs such as interactive design, self-designing systems, and fully automatic design for instance optimal systems.



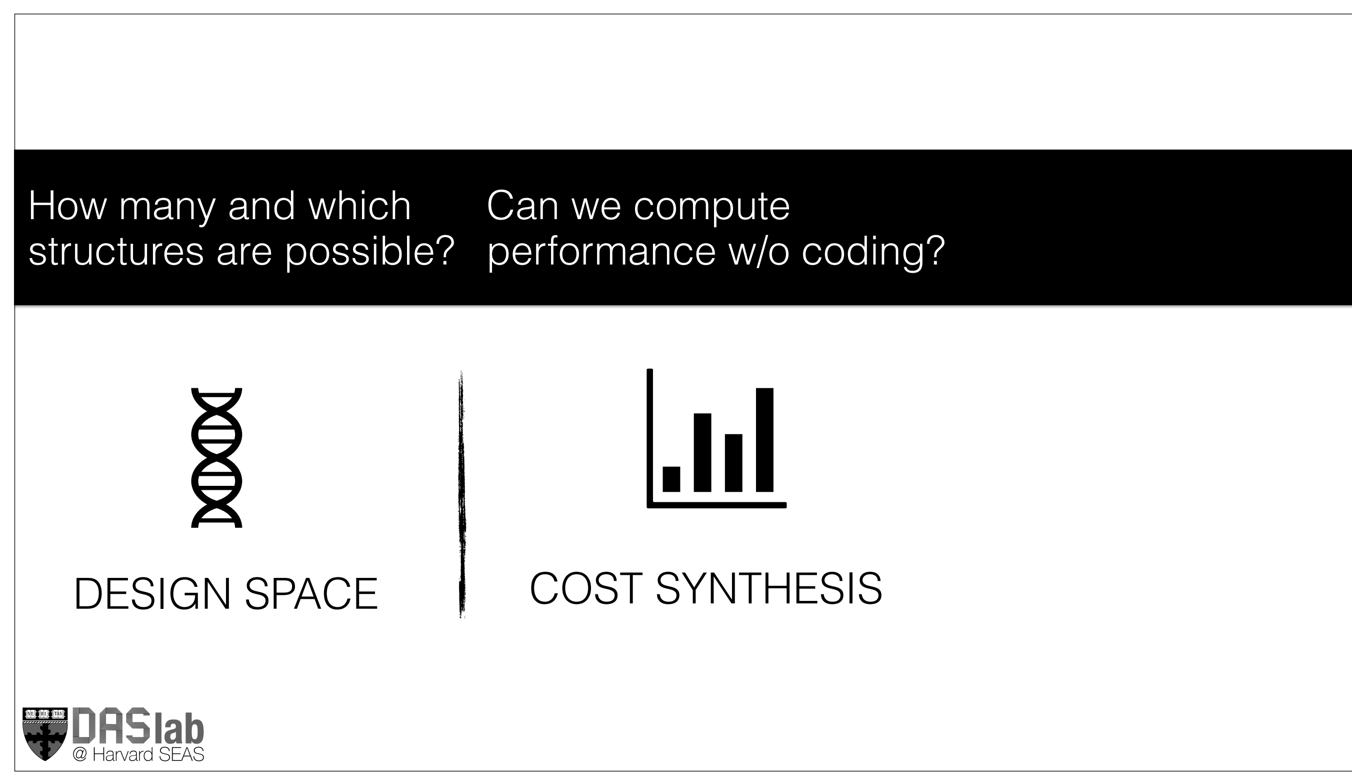

As a first step in this direction, we built an engine, which we call the Data Calculator and which takes as input the hardware, workload and layout of a data structure. It then computes automatically the algorithms that this data structure design needs to optimally process the workload on this hardware and it also computes the performance. That is, the response time that an actual implementation of this design would need to run this workload on this hardware. However, all this happens without the user having to implement anything and without even needing access to the actual hardware. Given this engine we show that we can start thinking about game-changing paradigms for system designs such as interactive design, self-designing systems, and fully automatic design for instance optimal systems.



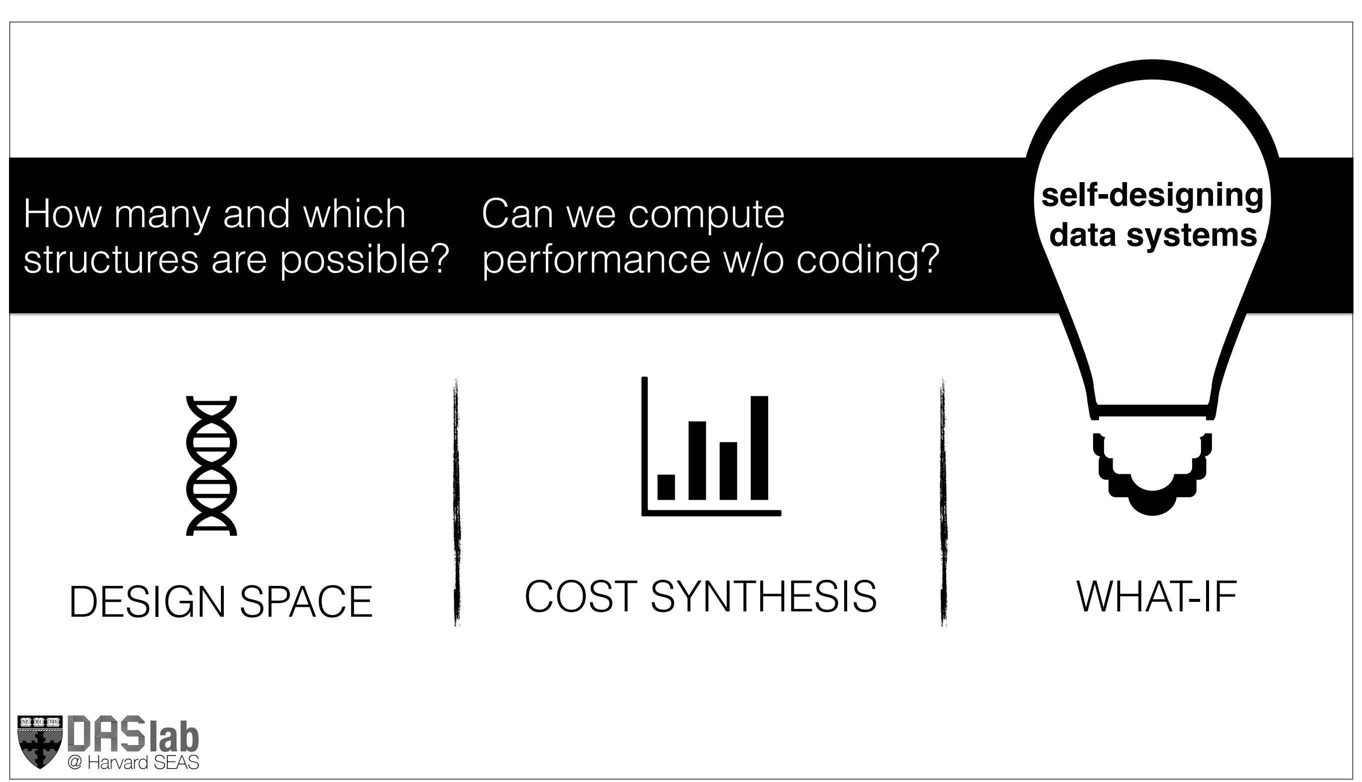

As a first step in this direction, we built an engine, which we call the Data Calculator and which takes as input the hardware, workload and layout of a data structure. It then computes automatically the algorithms that this data structure design needs to optimally process the workload on this hardware and it also computes the performance. That is, the response time that an actual implementation of this design would need to run this workload on this hardware. However, all this happens without the user having to implement anything and without even needing access to the actual hardware. Given this engine we show that we can start thinking about game-changing paradigms for system designs such as interactive design, self-designing systems, and fully automatic design for instance optimal systems.



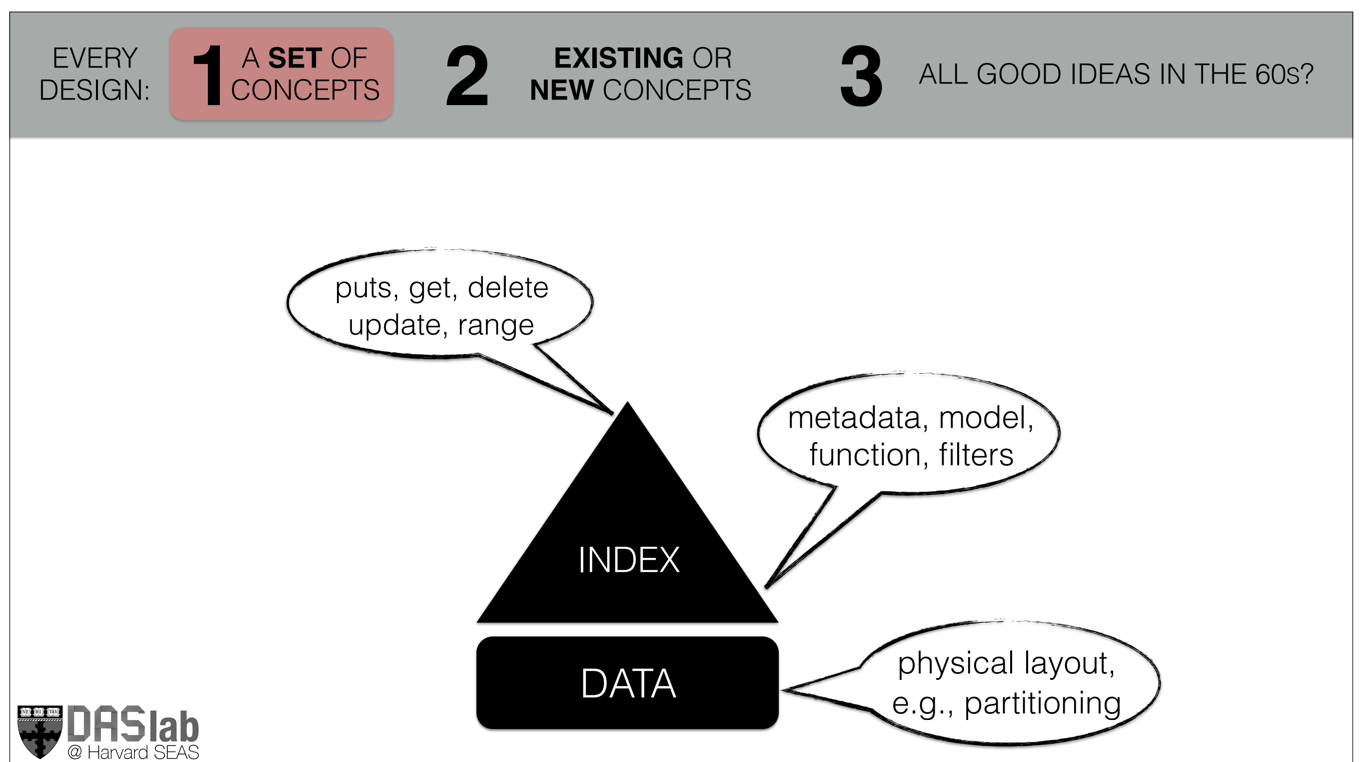

The primary observation is that all this boils down in being able to answer two questions.



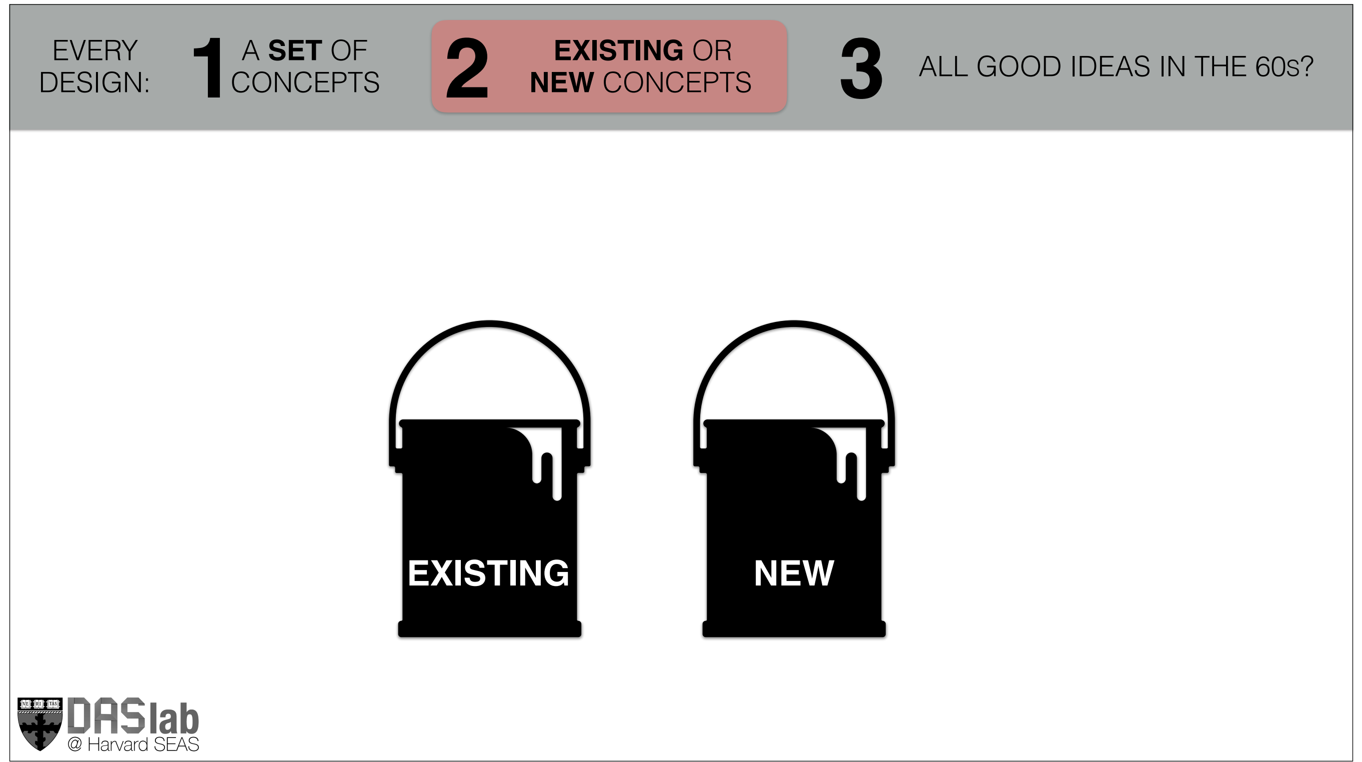

The primary observation is that all this boils down in being able to answer two questions.



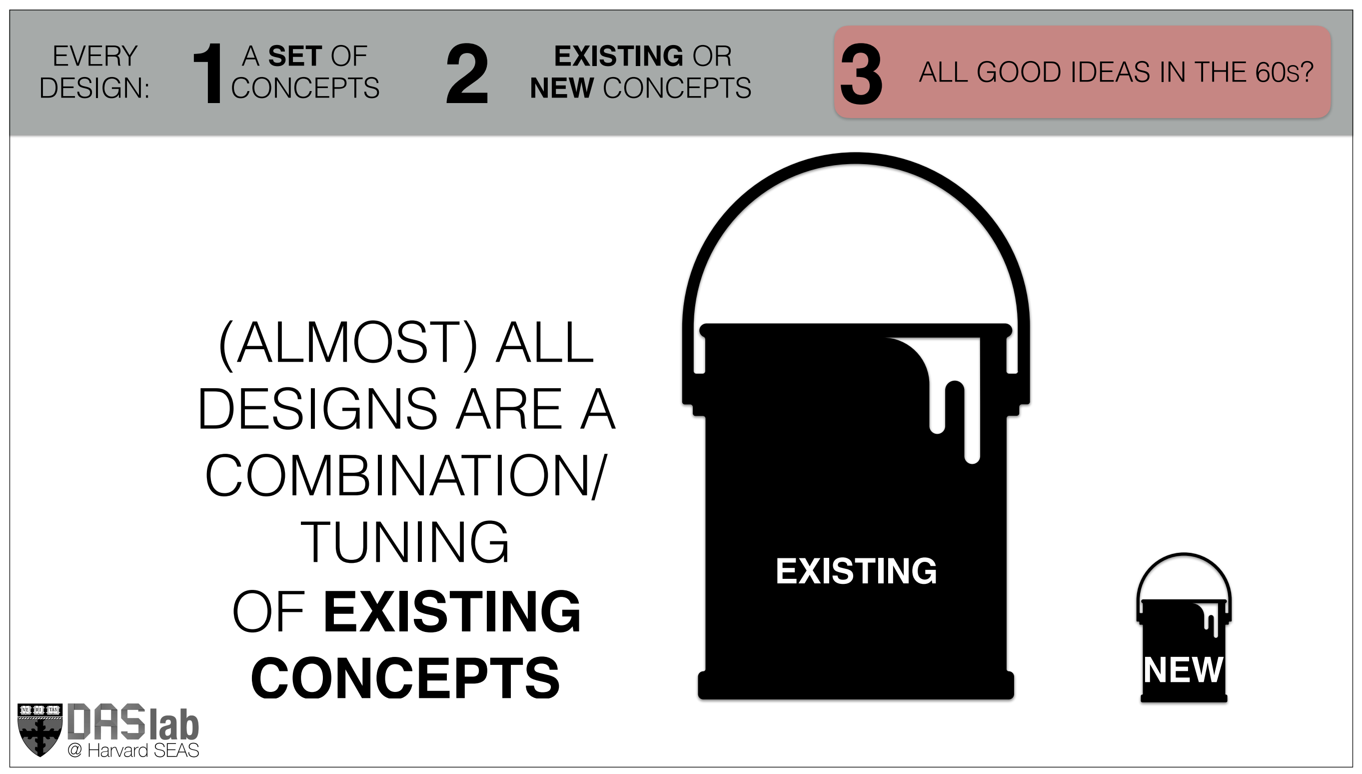

The primary observation is that all this boils down in being able to answer two questions.



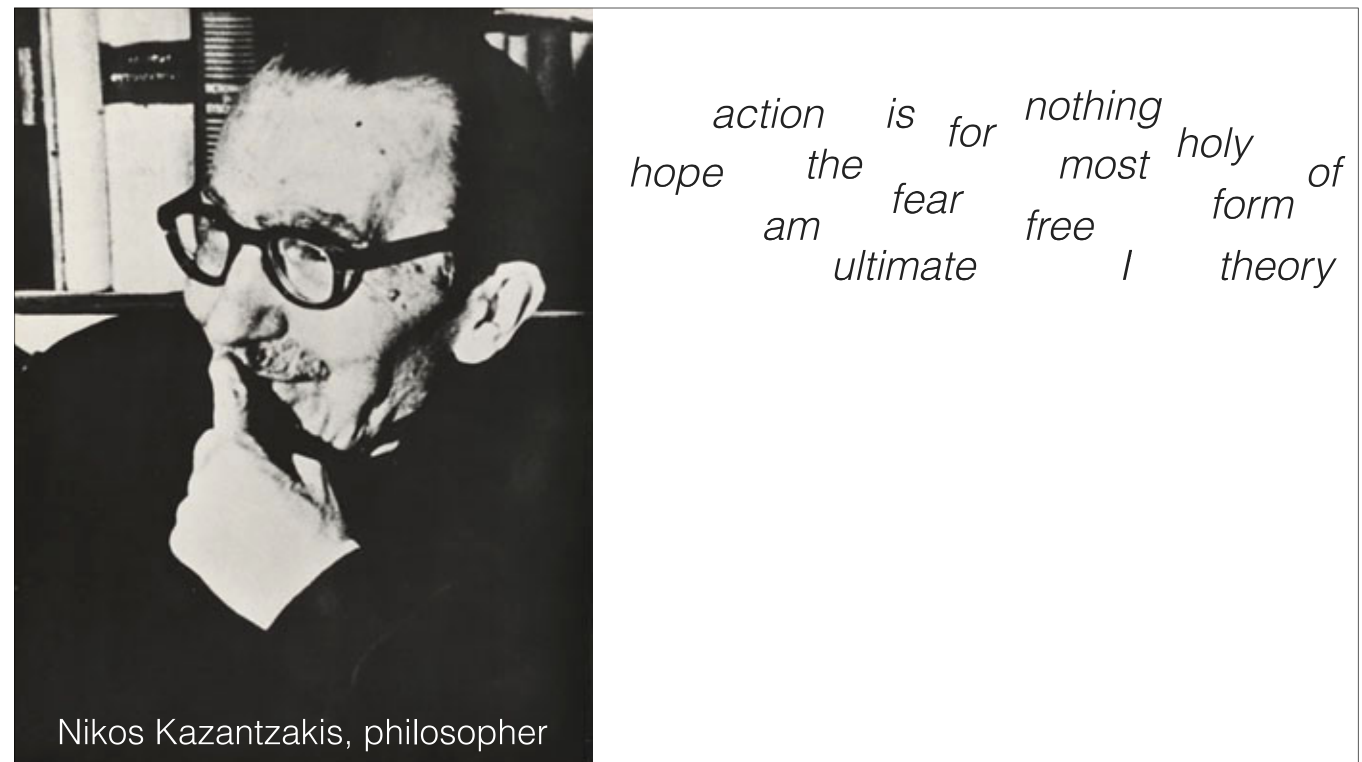

We now describe the reason why these questions are the key. First we observe that we can describe each data structure as a set of design concepts. That is all the fundamental design decisions that describe their core design.



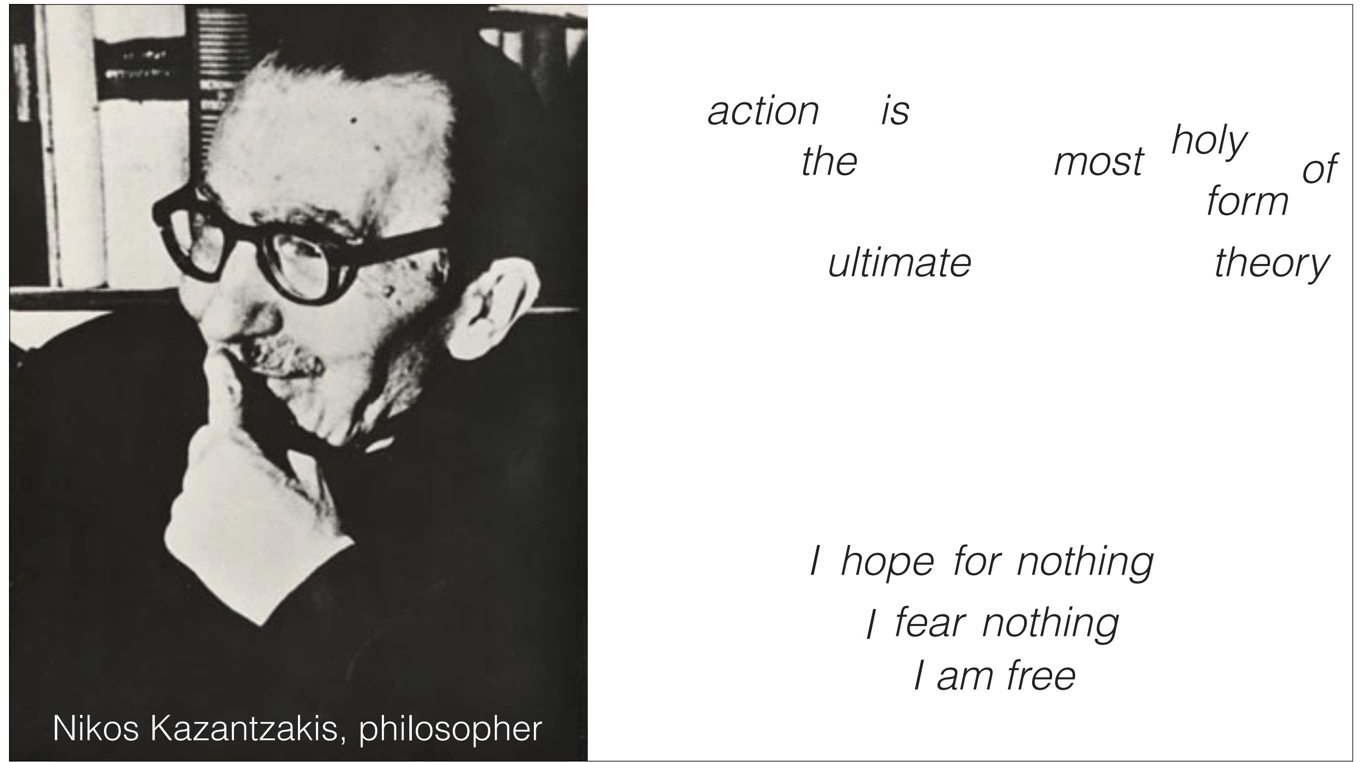

Given that observation, it follows that every new data structure design it falls in one of two buckets. It either consists fully of existing design concepts, or it contains at least one new design concept.




After 50-60 years of computer science research the point is that most of what we design it is actually a combination of existing design concepts. Hardly anything contains  an actual new design concept.

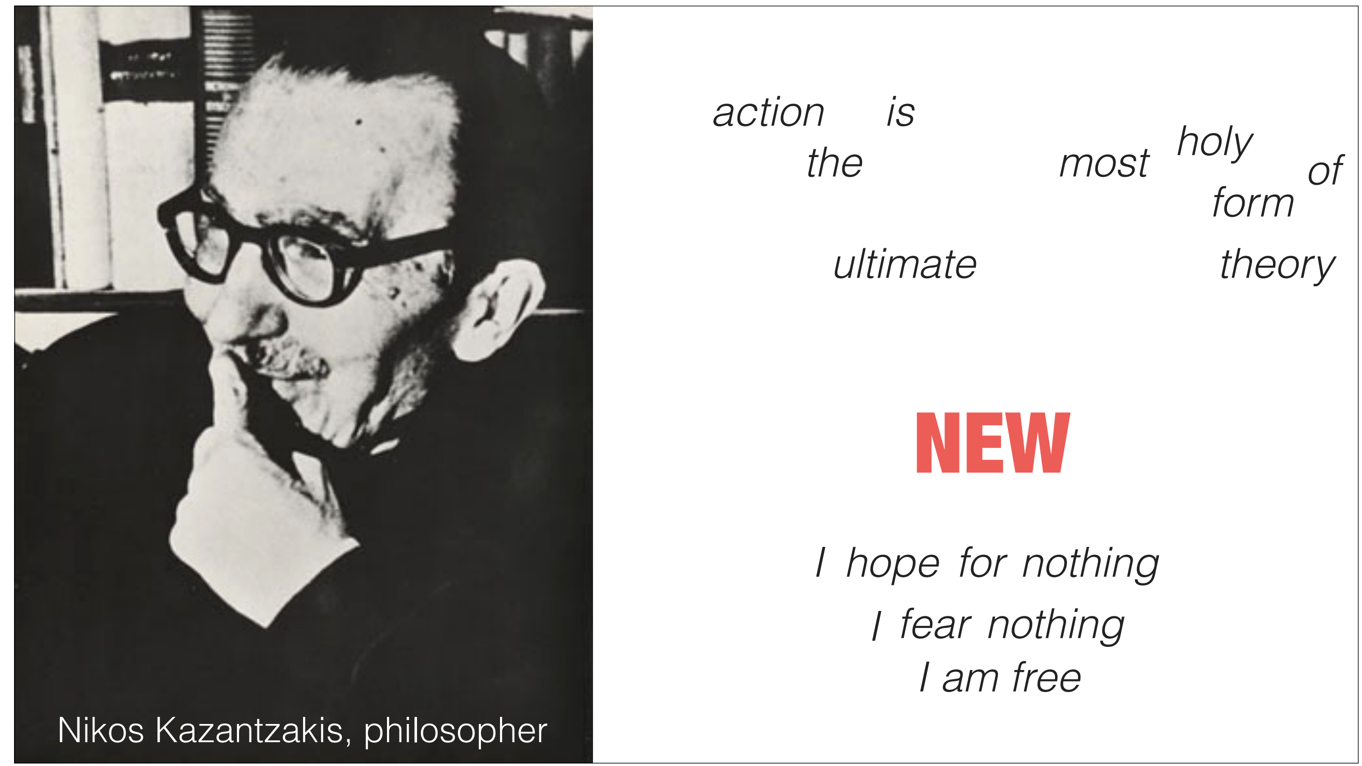



For most people the previous statement is either obvious or hard to accept. To better understand why the previous statement holds consider an example from literature. The words on the slides are known to all of us. We all know how to use them, what they mean, and we likely use them every day.



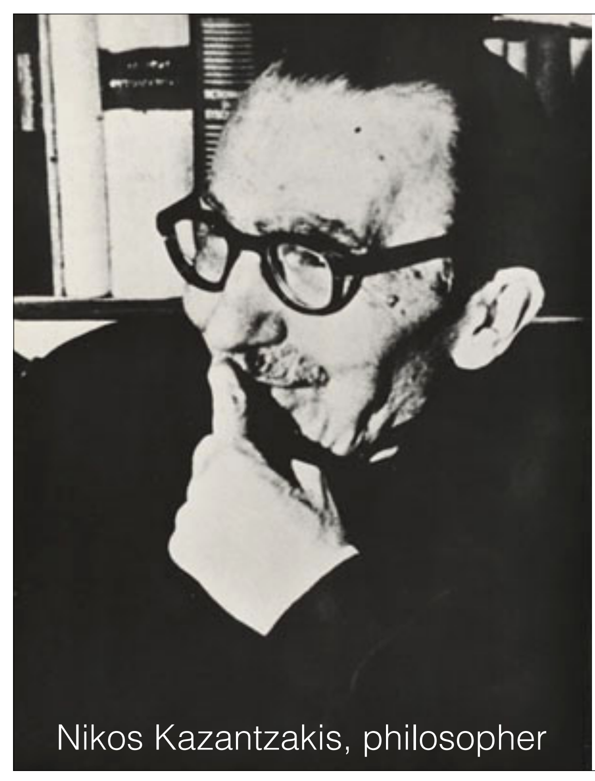
Nikos Kazantzakis, philosopher

Yet if we take a subset of these words, and put them in a particular order, we can create surprisingly new and unique sentences. However, even if this comes out of existing components it does not mean that it is easy to do. Given the massive “design space” it requires brilliance and talent to see these patterns. It is the same with data structure design and systems. Saying that most or even all designs come out of existing design concepts, does not mean that it is easy to come by. In fact it is extremely hard to do so as the design space is massive which is why many say that data structure and system design is kind of an art. The question is: can we accelerate this process somehow?



Nikos Kazantzakis, philosopher

Yet if we take a subset of these words, and put them in a particular order, we can create surprisingly new and unique sentences. However, even if this comes out of existing components it does not mean that it is easy to do. Given the massive "design space" it requires brilliance and talent to see these patterns. It is the same with data structure design and systems. Saying that most or even all designs come out of existing design concepts, does not mean that it is easy to come by. In fact it is extremely hard to do so as the design space is massive which is why many say that data structure and system design is kind of an art. The question is: can we accelerate this process somehow?

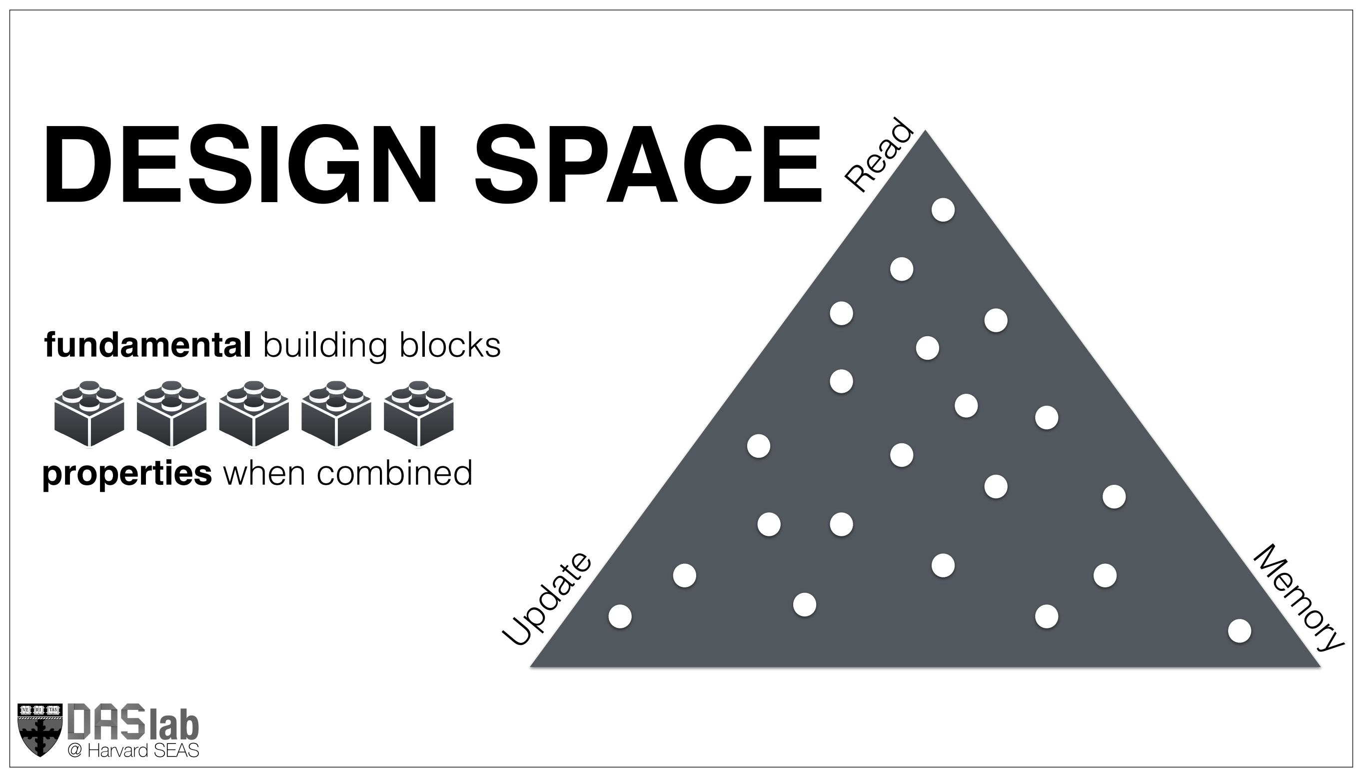


Nikos Kazantzakis, philosopher

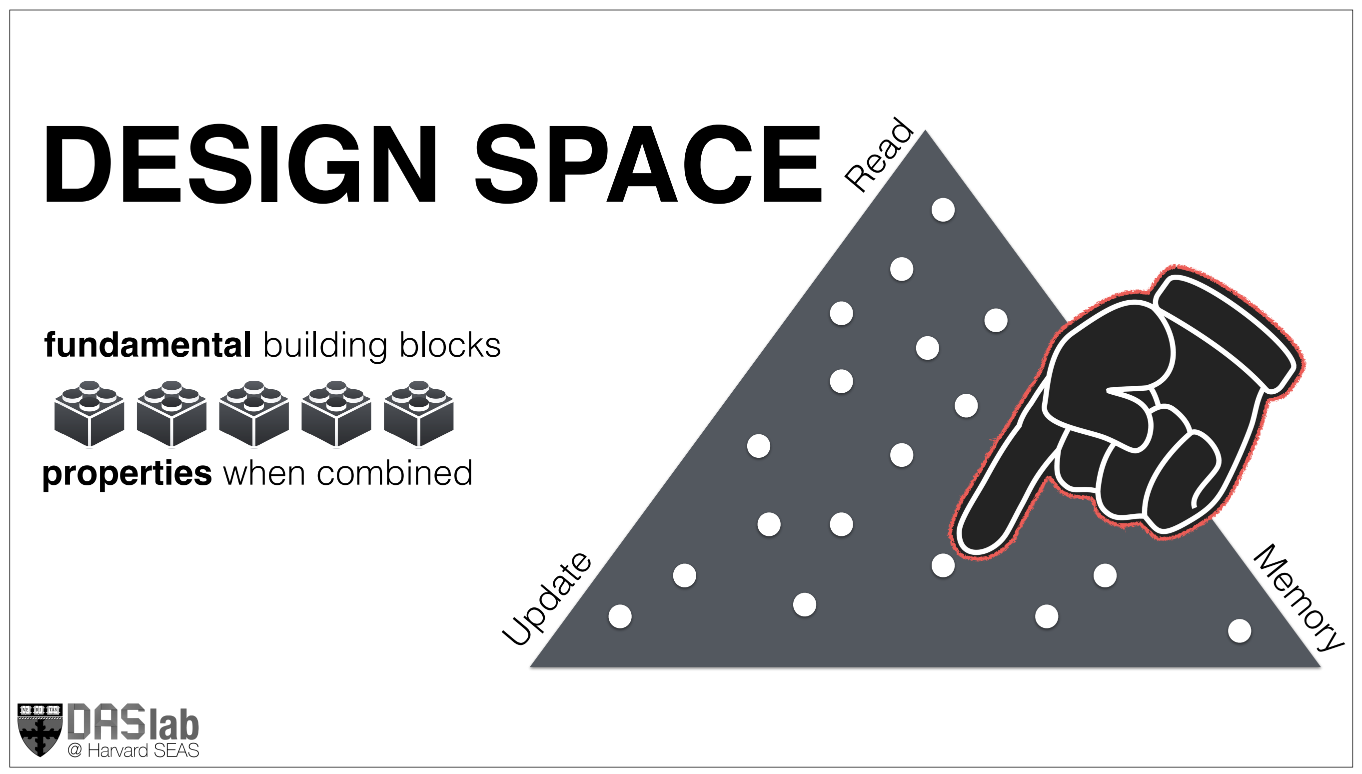
*action is holy*
*the most of*
*form*
*ultimate theory*

## NEW & BRILLIANT

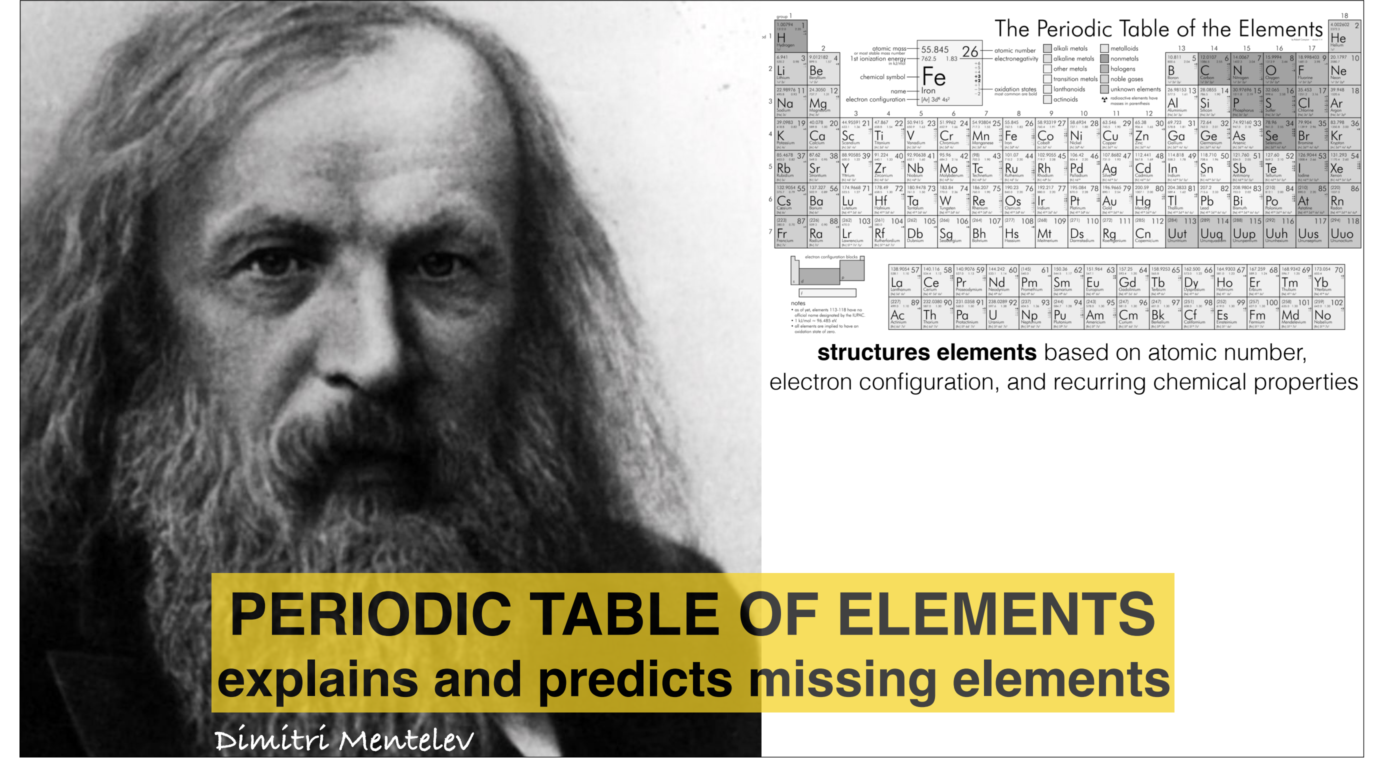
*I hope for nothing*
*I fear nothing*
*I am free*



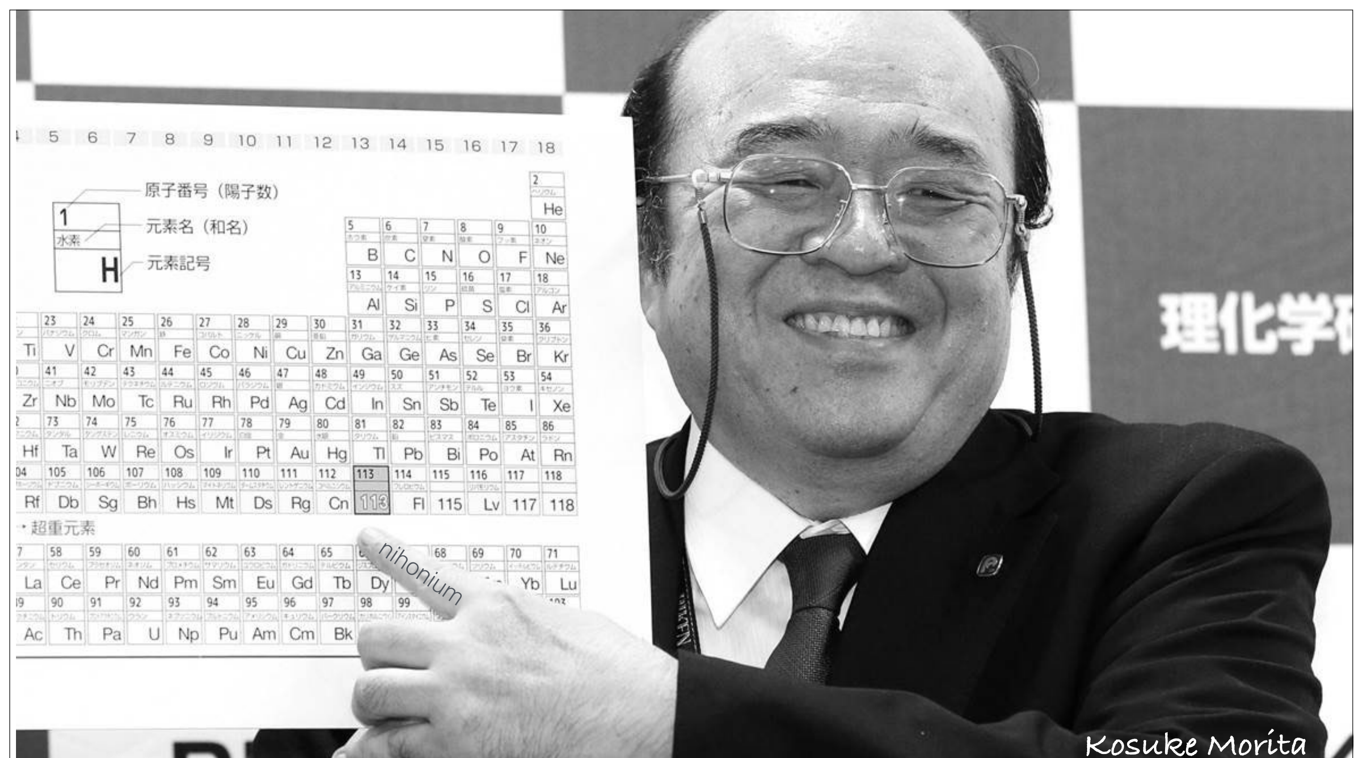

This is exactly what we do by trying to find the fundamental first principles of design, figure out how they combine, how we can synthesize them to new designs, what are their properties, and eventually to be able to automatically choose among the possible combinations.



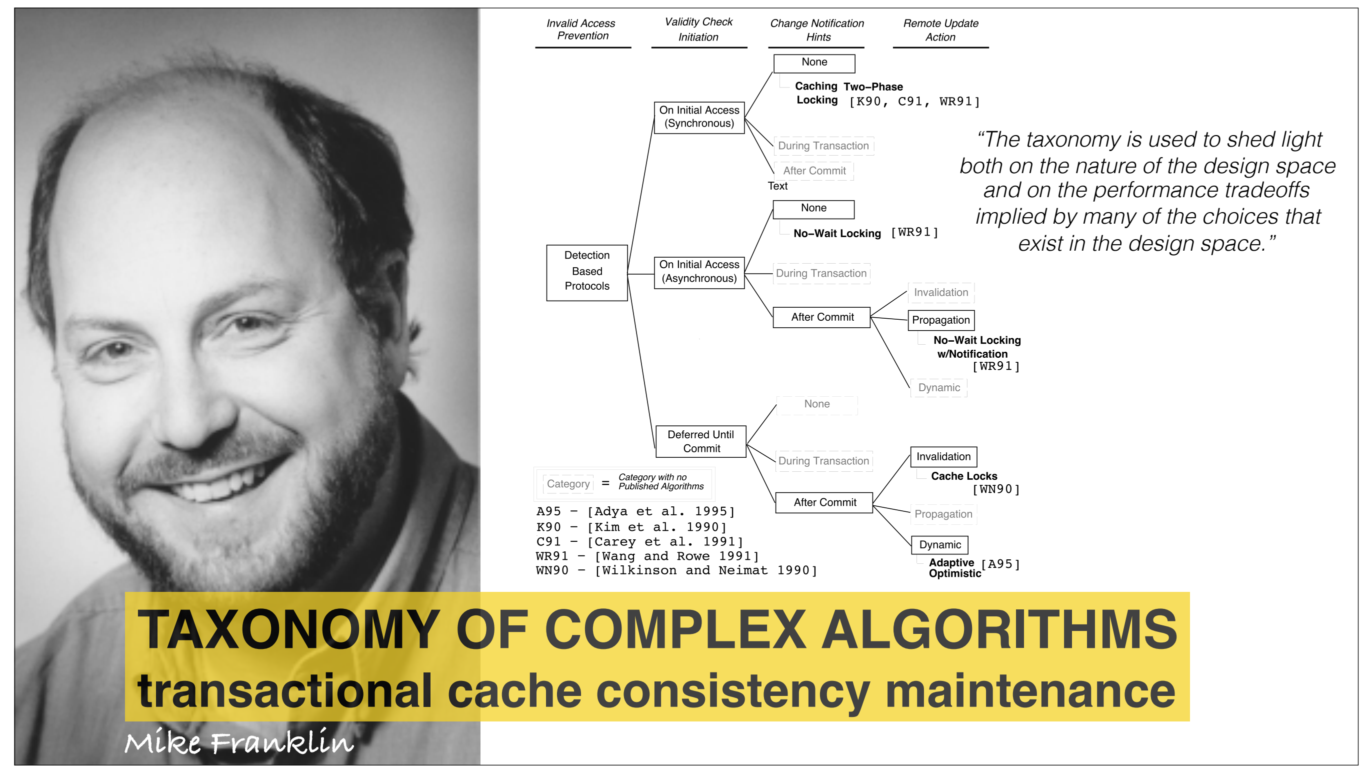

This is exactly what we do by trying to find the fundamental first principles of design, figure out how they combine, how we can synthesize them to new designs, what are their properties, and eventually to be able to automatically choose among the possible combinations.



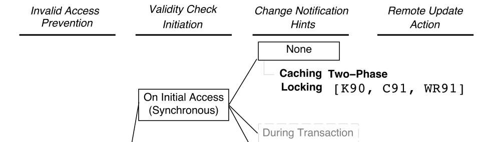

A useful analogy is the periodic table of elements in chemistry. It created the design space of chemical elements. It brought structure and allowed researchers to organize the knowledge of the time. But most importantly the structure of the design space itself allowed future researchers to guide their efforts! Researchers were doing research for more than one hundred years driven by the gaps in the periodic table.



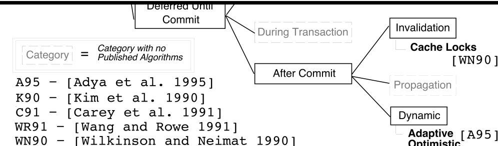

This is one of the last elements that was discovered driven purely by the structure of the 100 year old periodic table.




Similar research style has appeared in several fields of computer science including databases. Mike Franklin's PhD is an elegant example of creating a design space, organizing existing work and showing several open research opportunities through the design space.



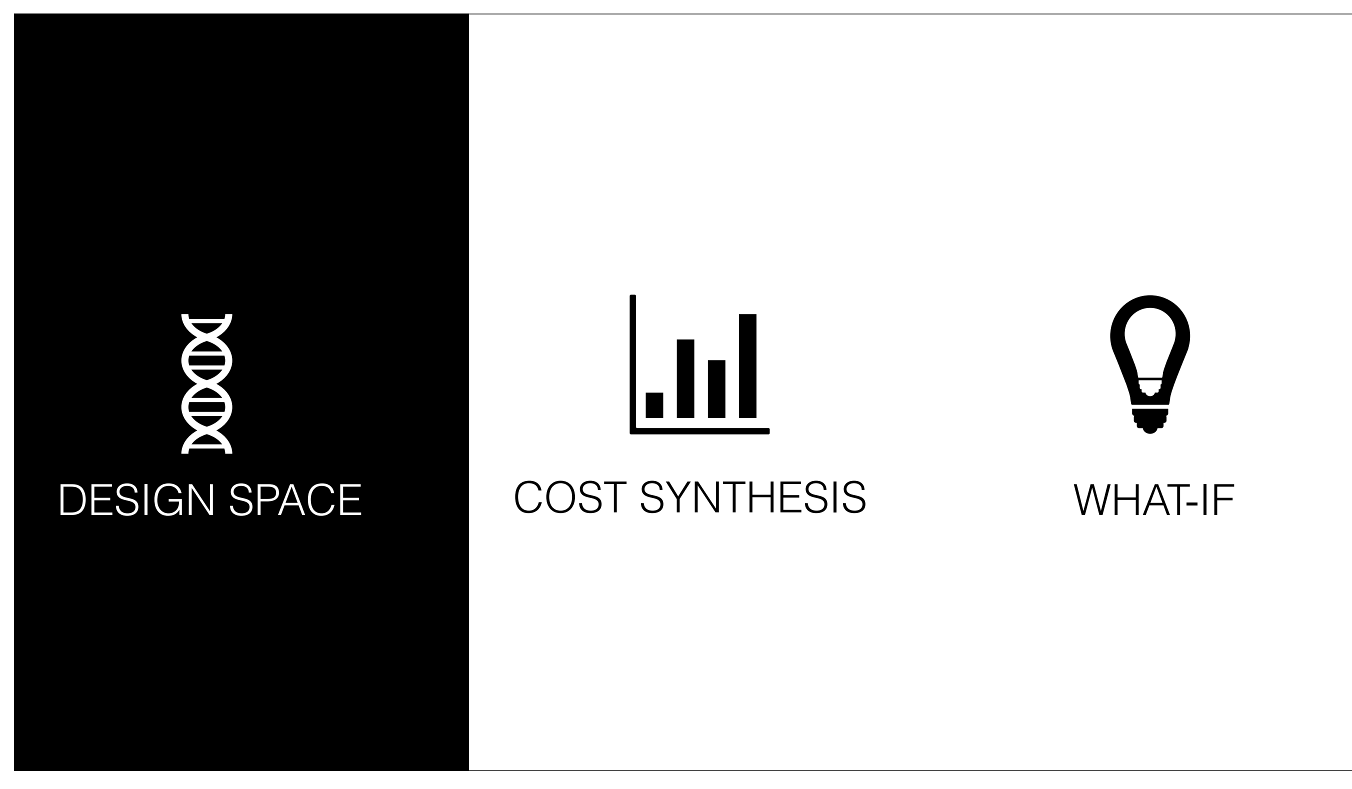

“The taxonomy is used to shed light both on the nature of the design space and on the performance tradeoffs implied by many of the choices that exist in the design space.”

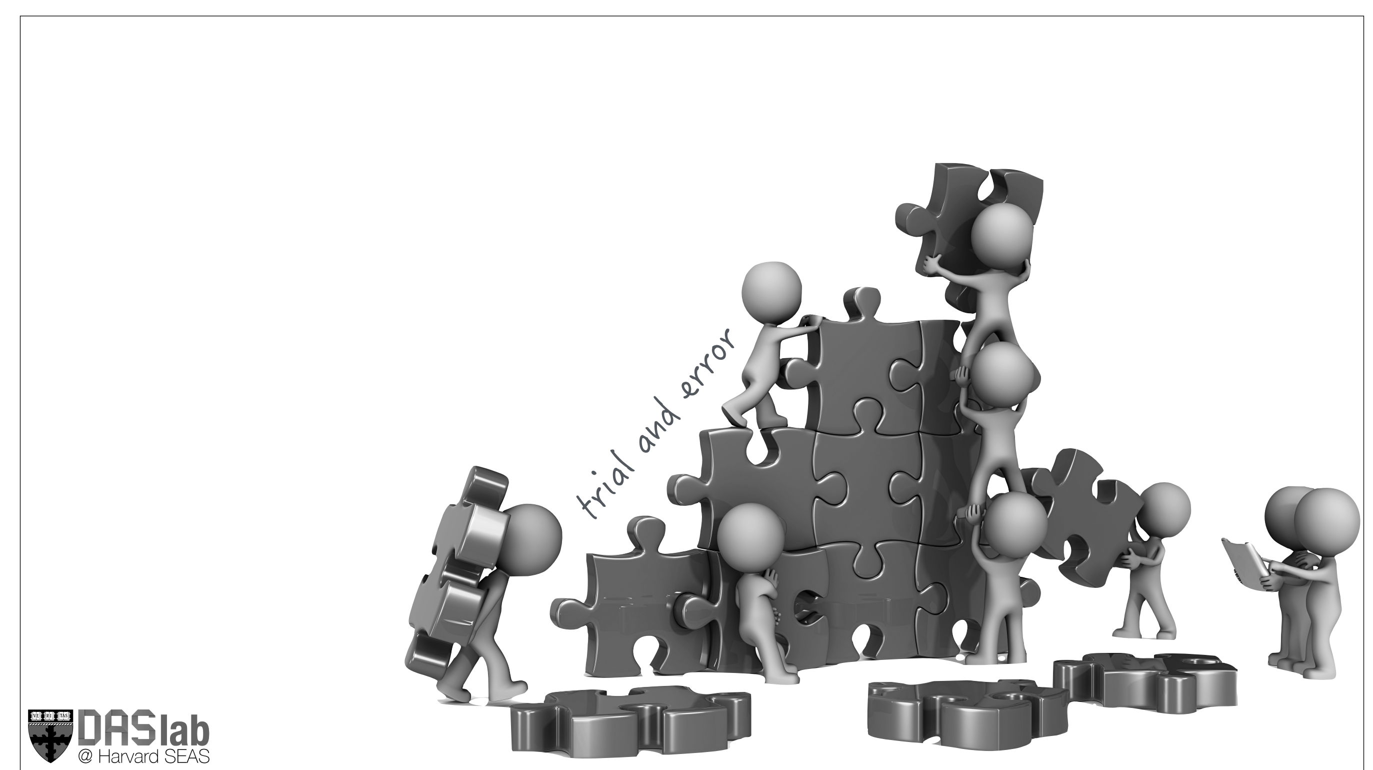



# TAXONOMY OF COMPLEX ALGORITHMS

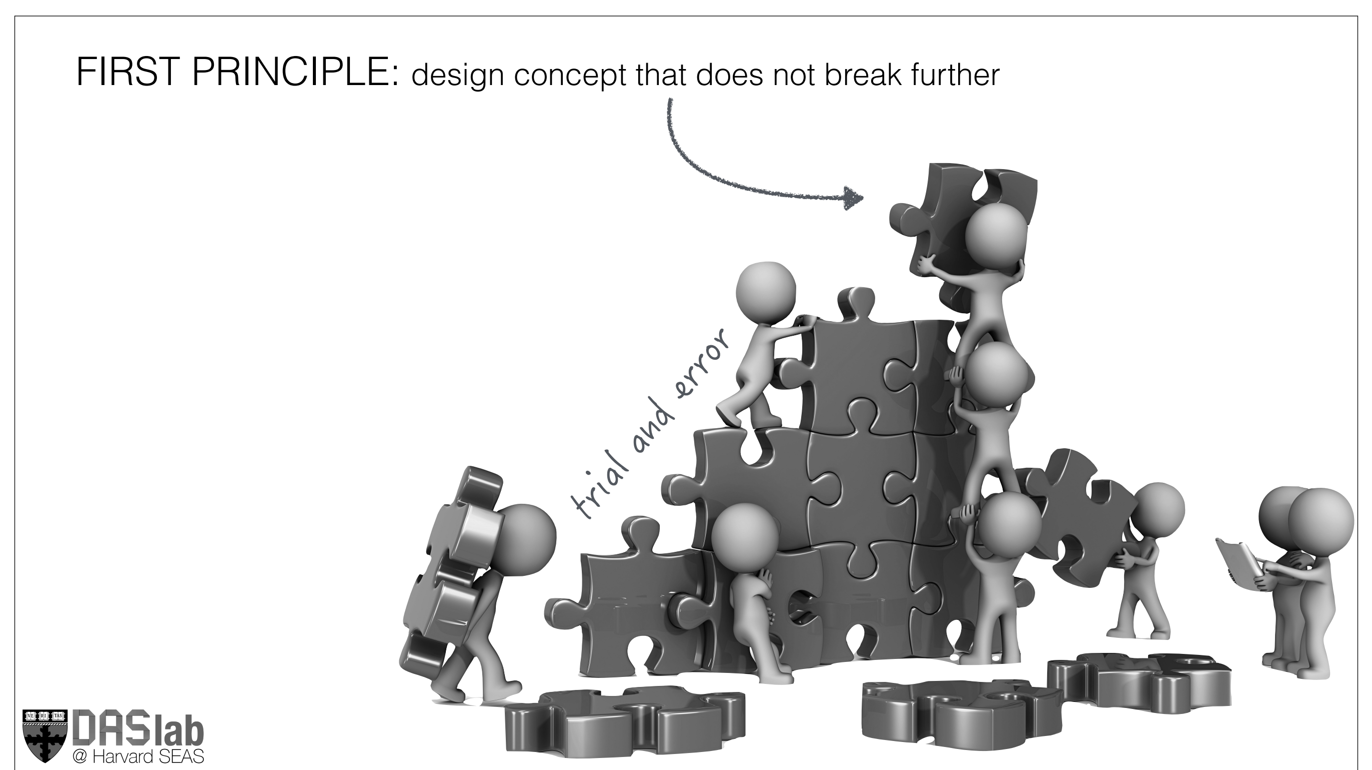
## transactional cache consistency maintenance

Mike Franklin

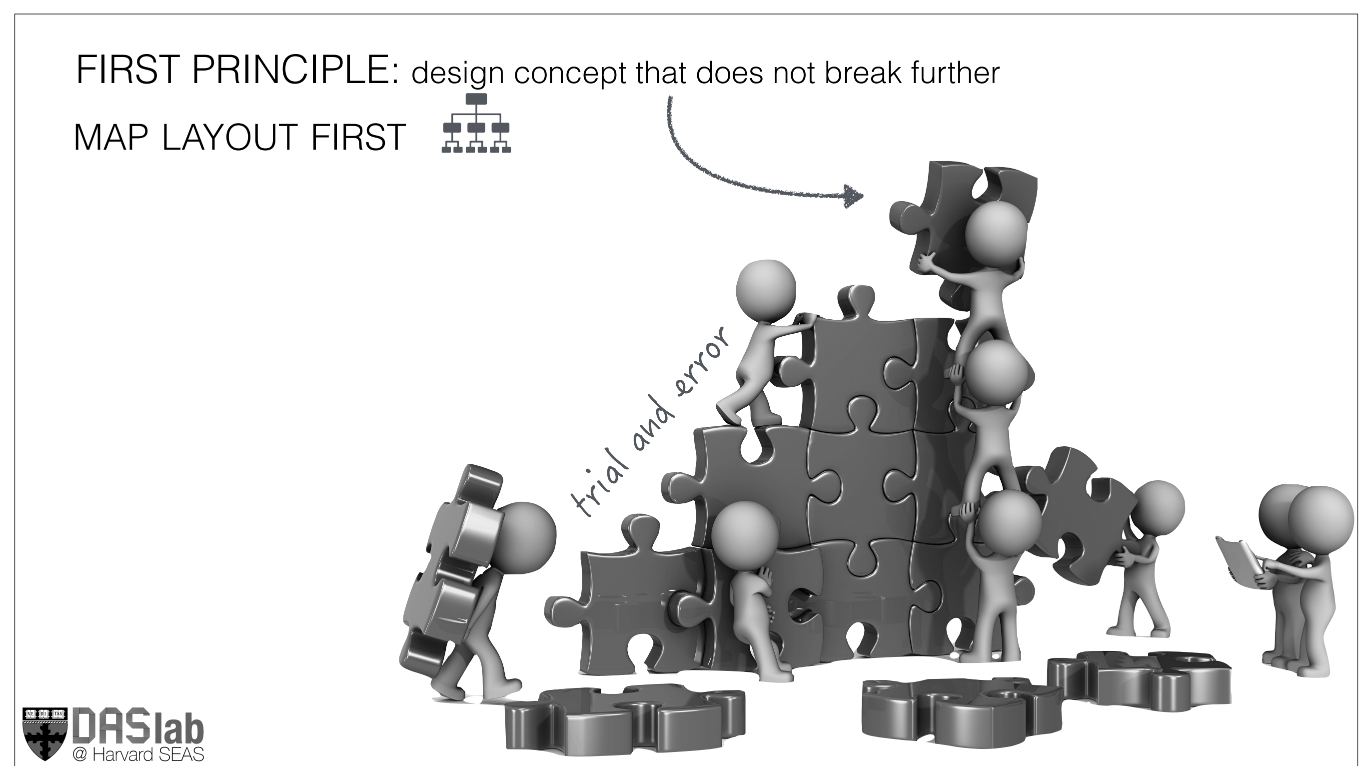

DESIGN SPACE
COST SYNTHESIS
WHAT-IF



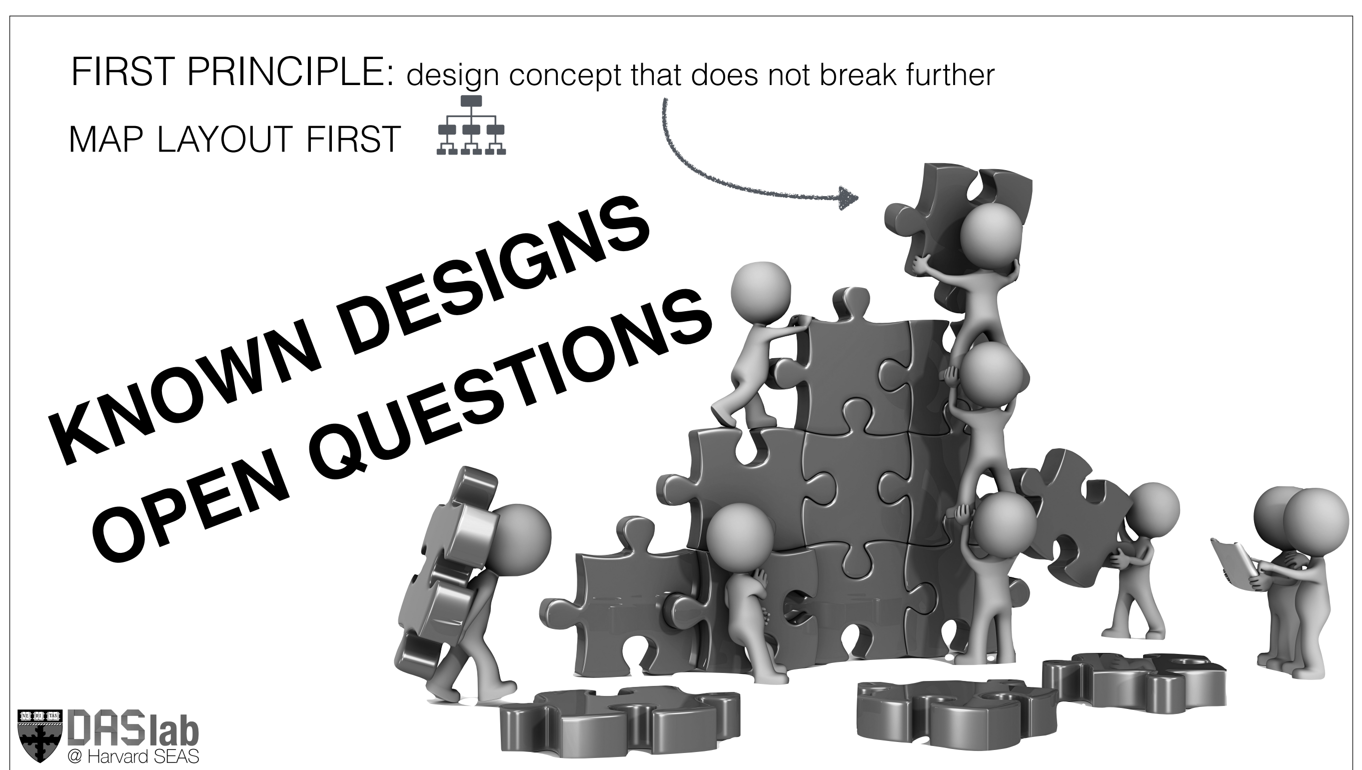

The quest to find the first principles of data structure design is an iterative process between breaking existing designs to their fundamental components and trying to answer open questions in the field through design space organization as some times this shows us that certain design principles that were thought to be fundamental they are not (for example they can be optimized out).



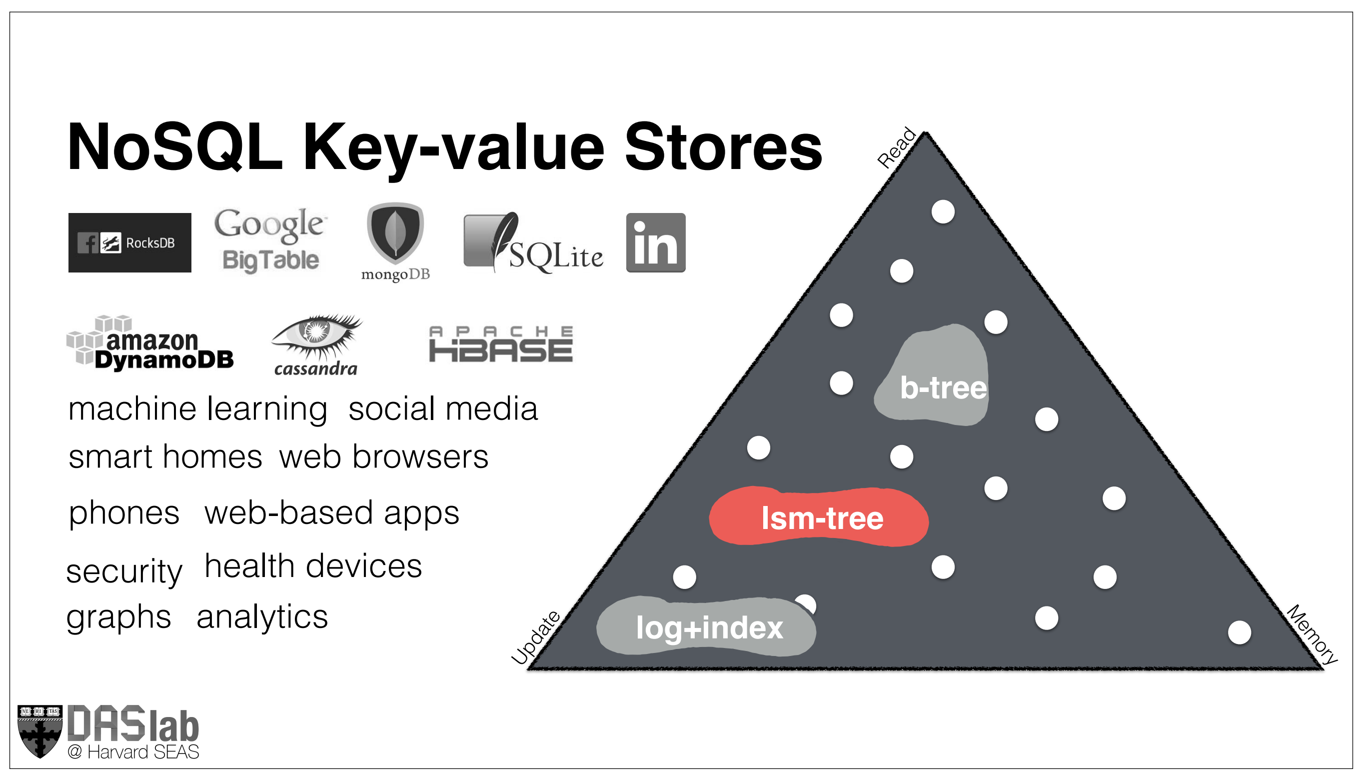

The quest to find the first principles of data structure design is an iterative process between breaking existing designs to their fundamental components and trying to answer open questions in the field through design space organization as some times this shows us that certain design principles that were thought to be fundamental they are not (for example they can be optimized out).



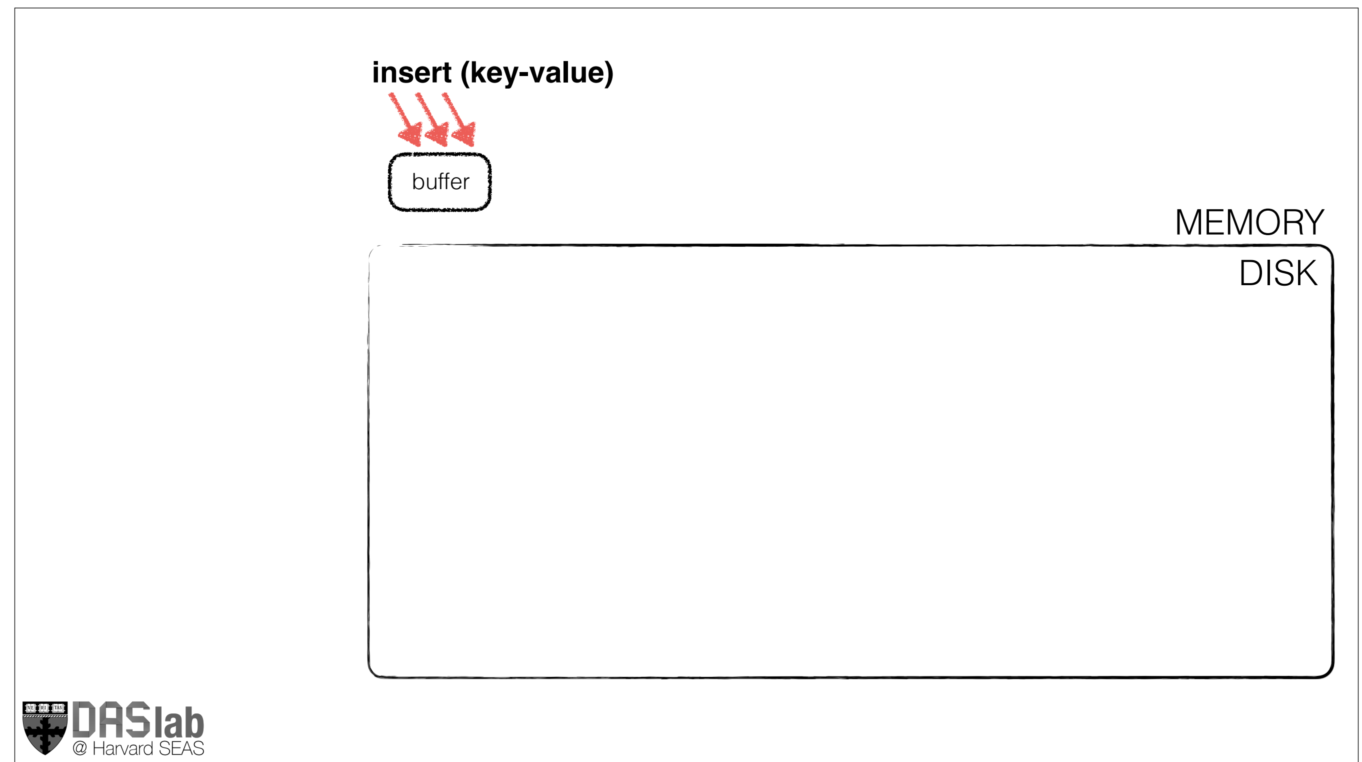

The quest to find the first principles of data structure design is an iterative process between breaking existing designs to their fundamental components and trying to answer open questions in the field through design space organization as some times this shows us that certain design principles that were thought to be fundamental they are not (for example they can be optimized out).




The quest to find the first principles of data structure design is an iterative process between breaking existing designs to their fundamental components and trying to answer open questions in the field through design space organization as some times this shows us that certain design principles that were thought to be fundamental they are not (for example they can be optimized out).




We will see a few examples of decomposing designσ to their principles. In particular we will study LSM-tree based designs.

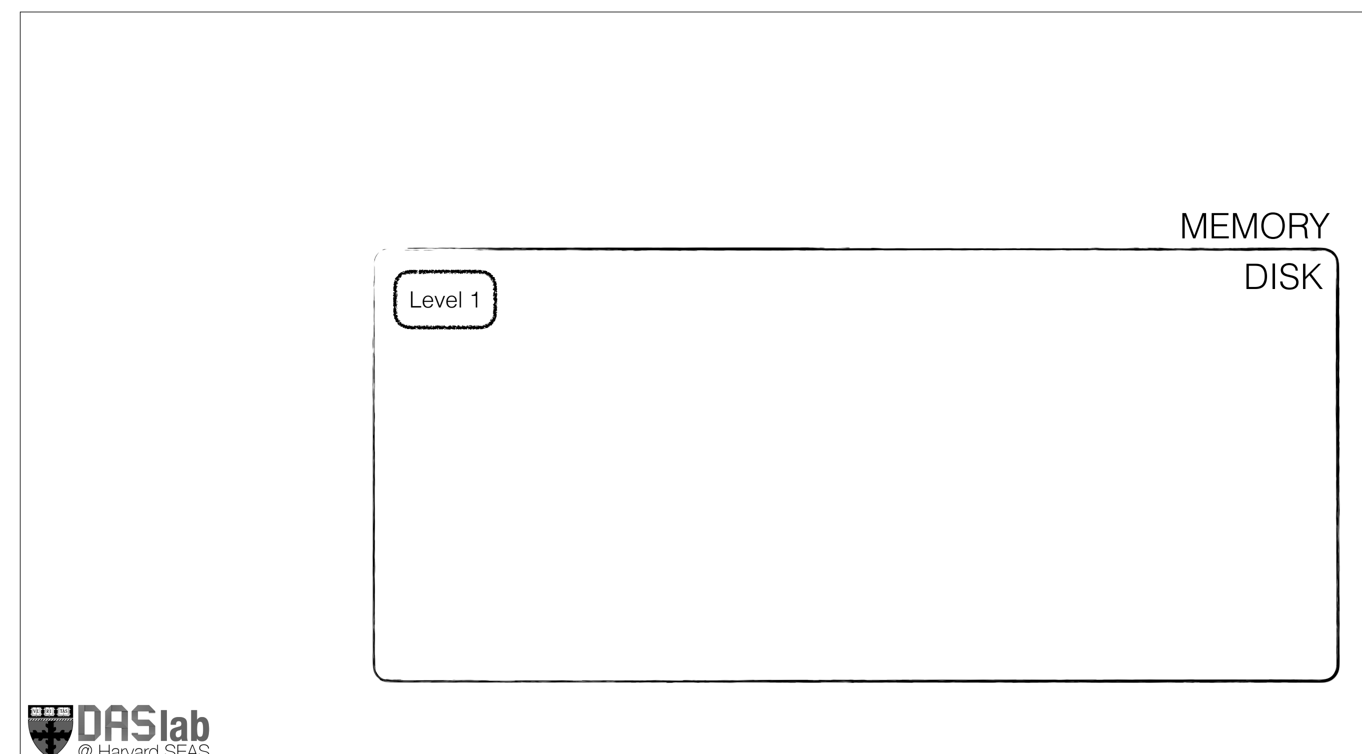


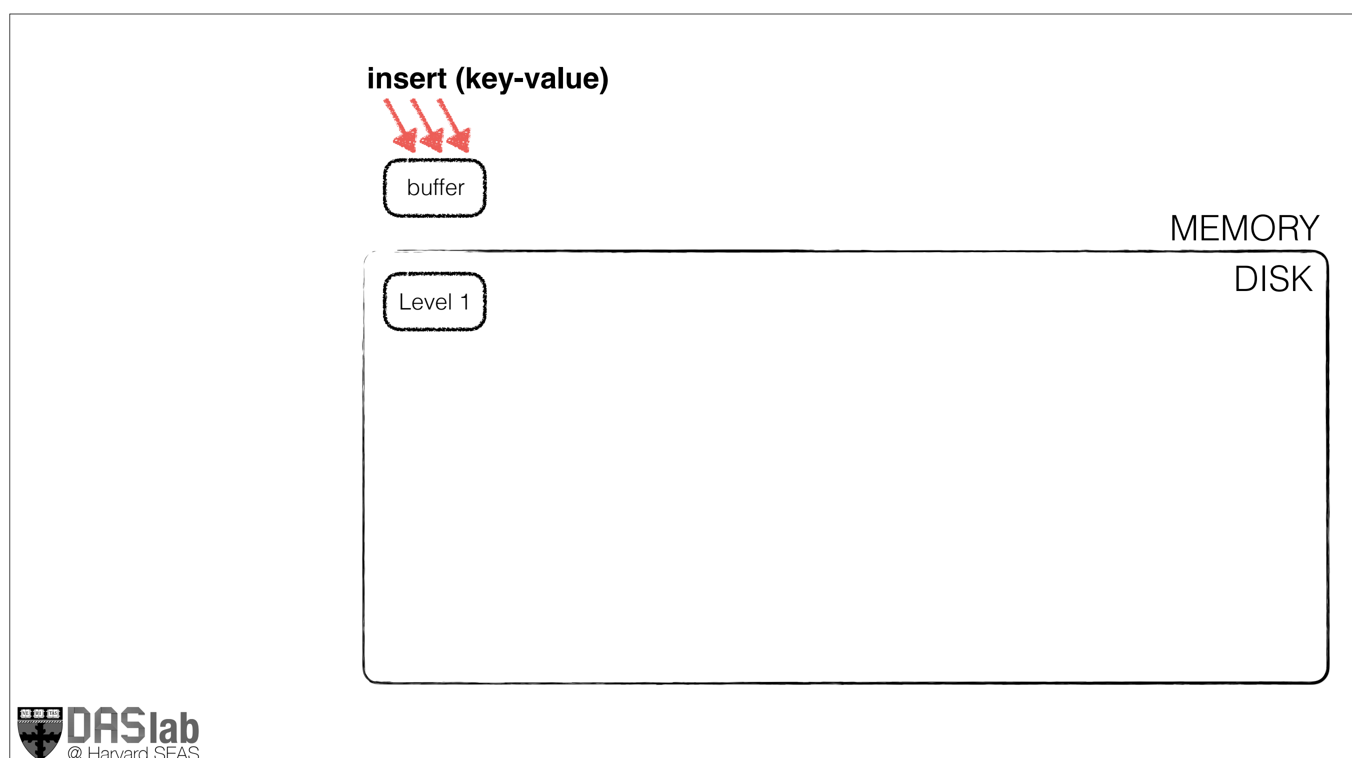

LSM-tree designs work by buffering incoming data at an in-memory buffer and then merging to increasingly larger levels to disk.

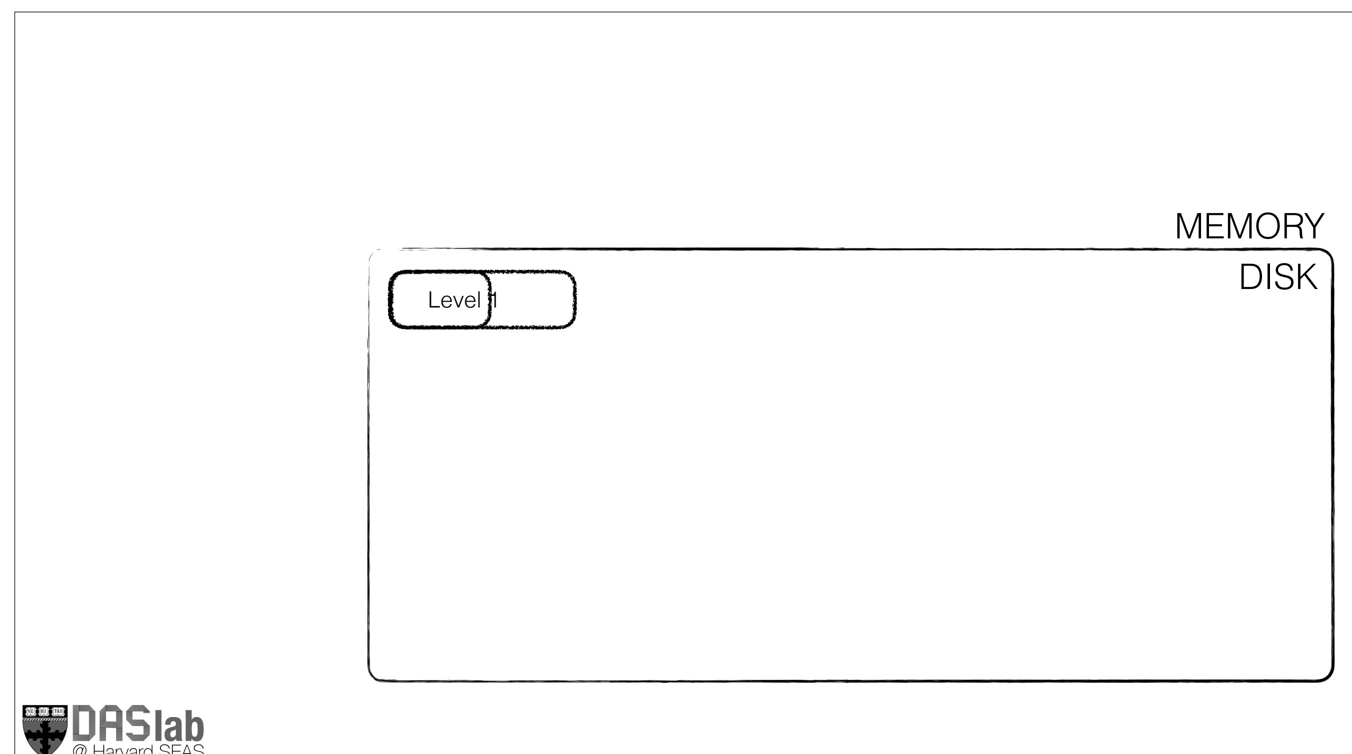
MEMORY

DISK

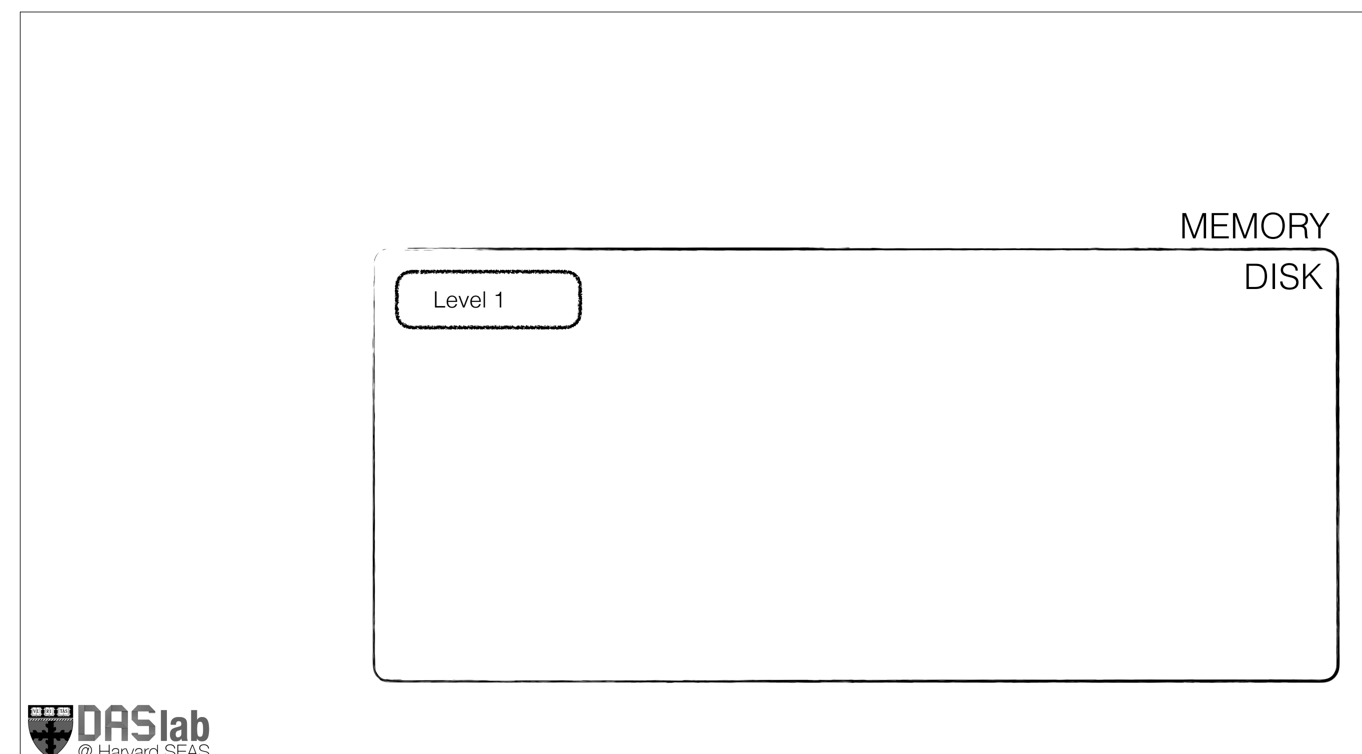
DASlab
@ Harvard SEAS

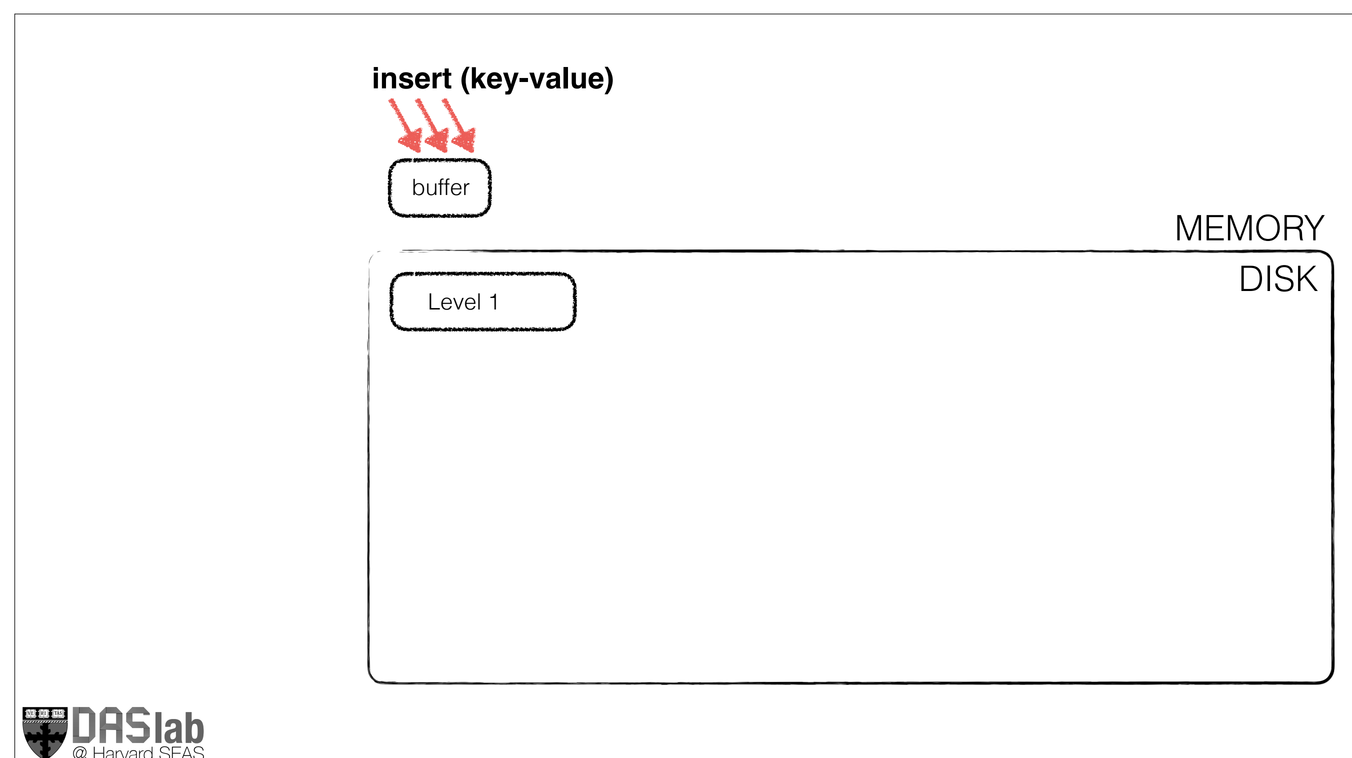

MEMORY
DISK
Level 1
DASlab
@ Harvard SEAS

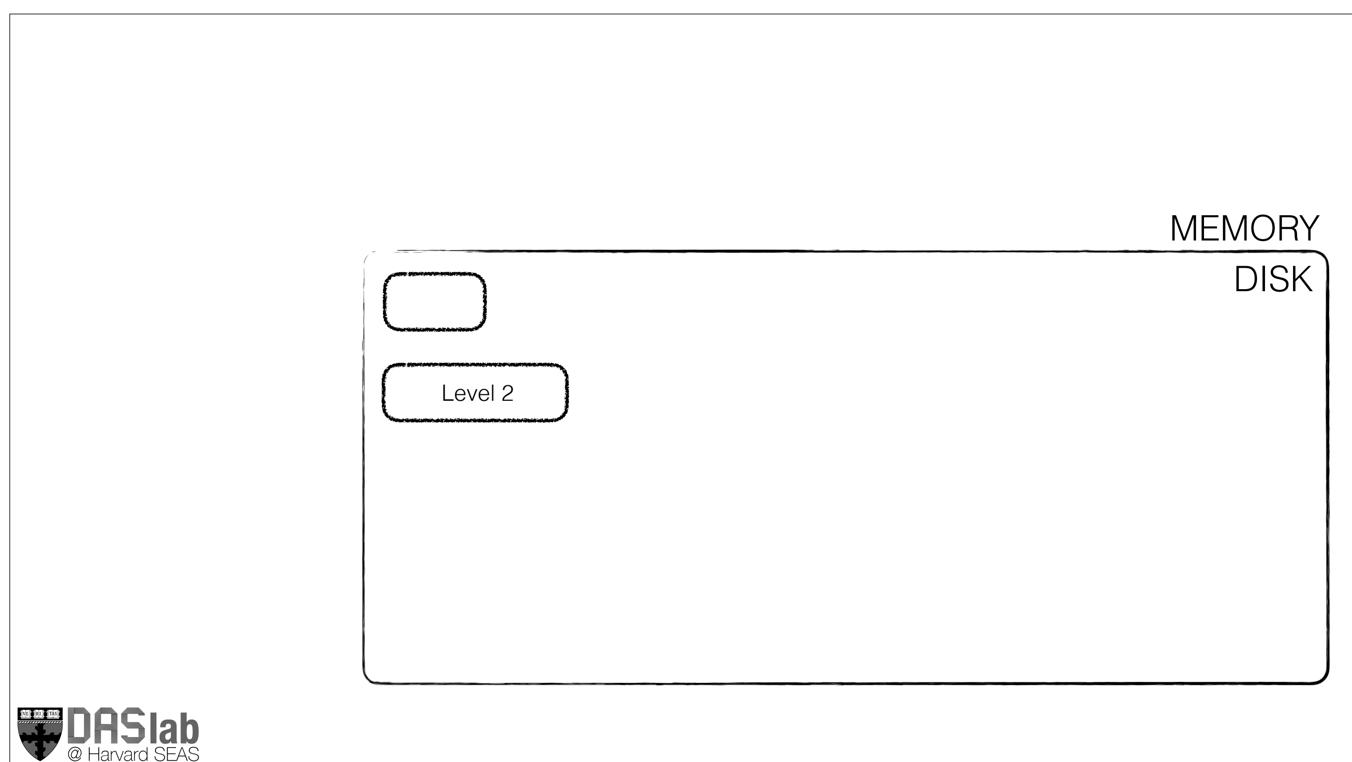

insert (key-value)
buffer
MEMORY
DISK
Level 1
DASlab
@ Harvard SEAS

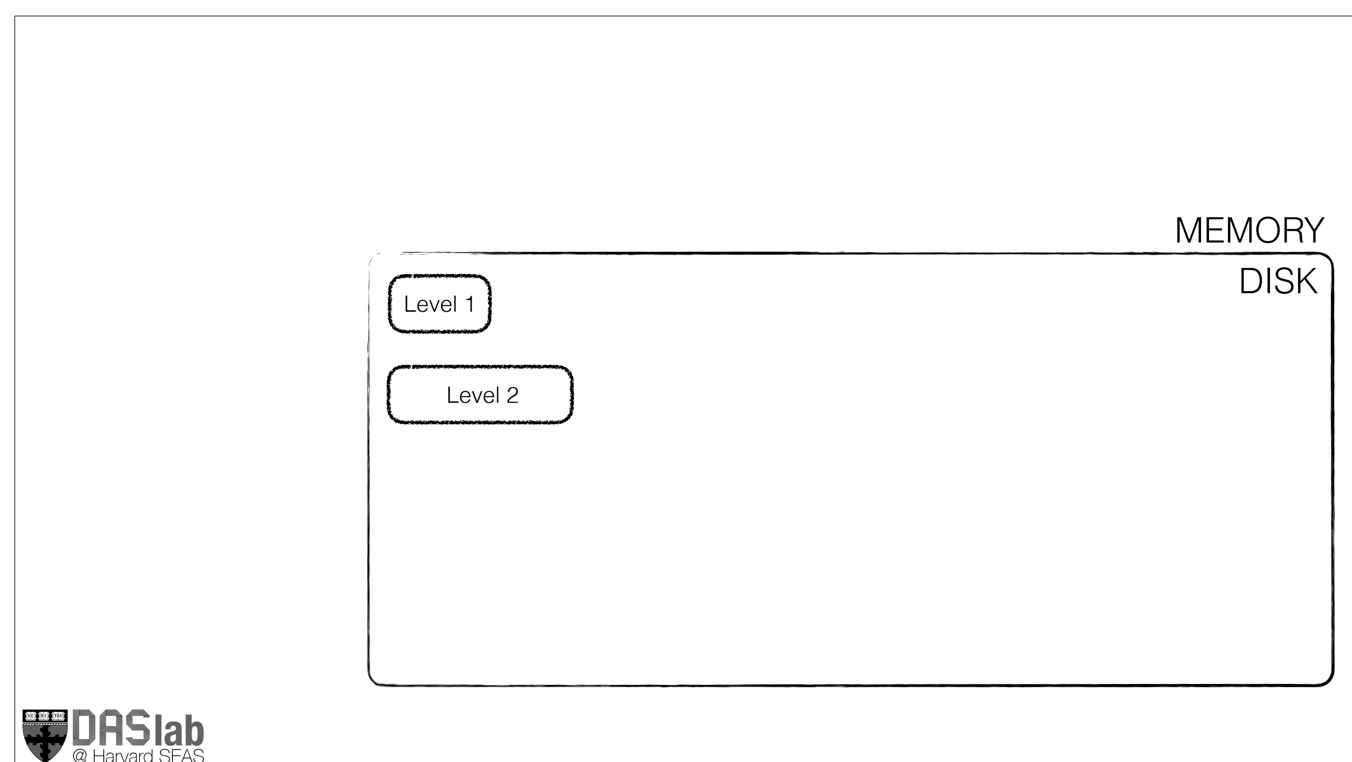


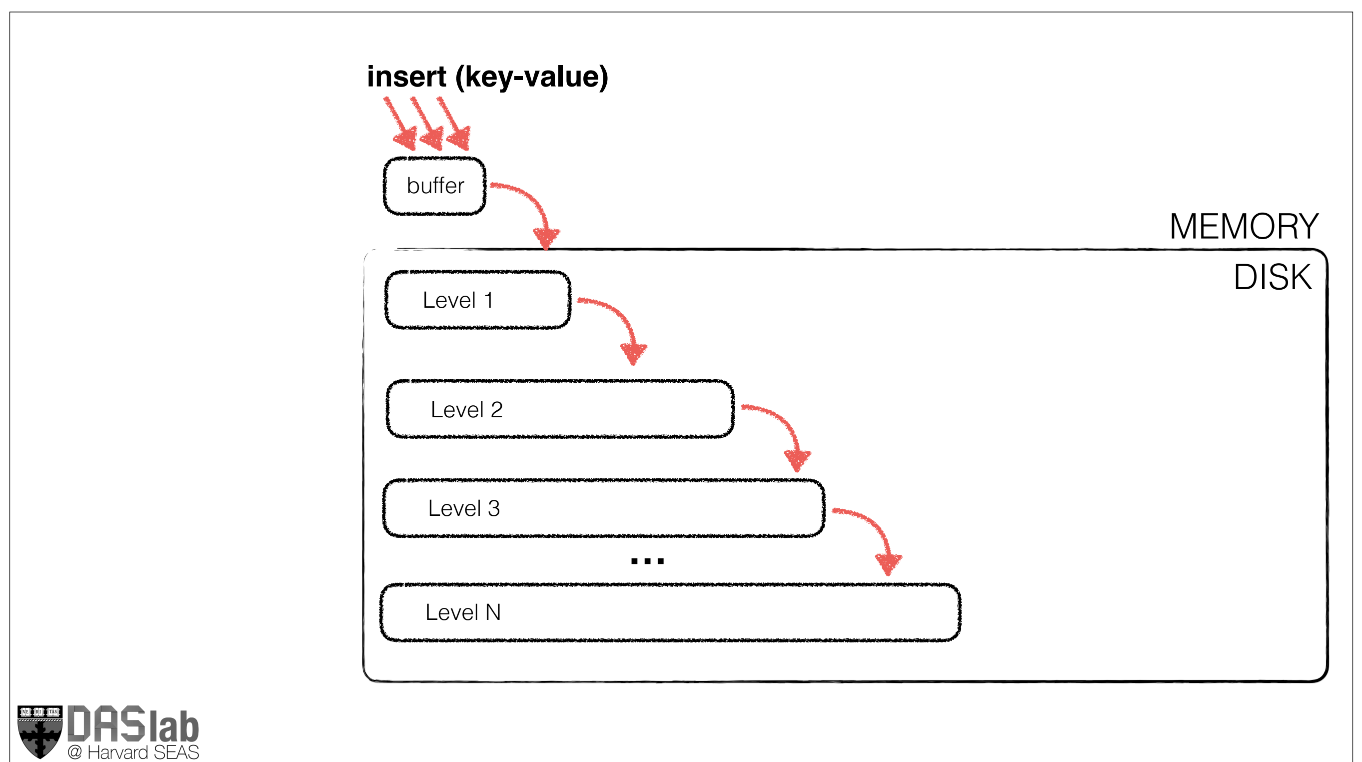
MEMORY
DISK
Level 1
DASlab
@ Harvard SEAS


MEMORY
DISK
Level 1
DASlab
@ Harvard SEAS

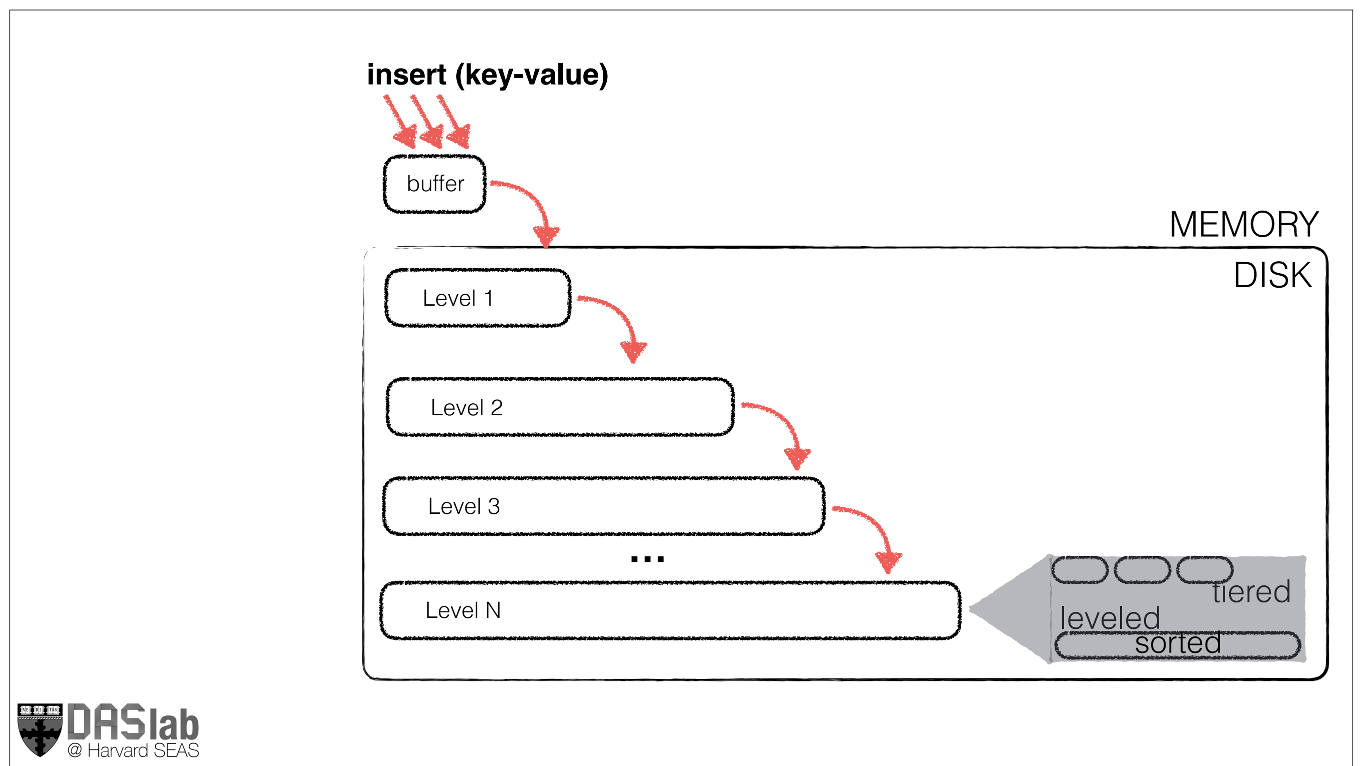

insert (key-value)
buffer
MEMORY
DISK
Level 1
DASlab
@ Harvard SEAS


MEMORY
DISK
Level 2
DASlab
@ Harvard SEAS

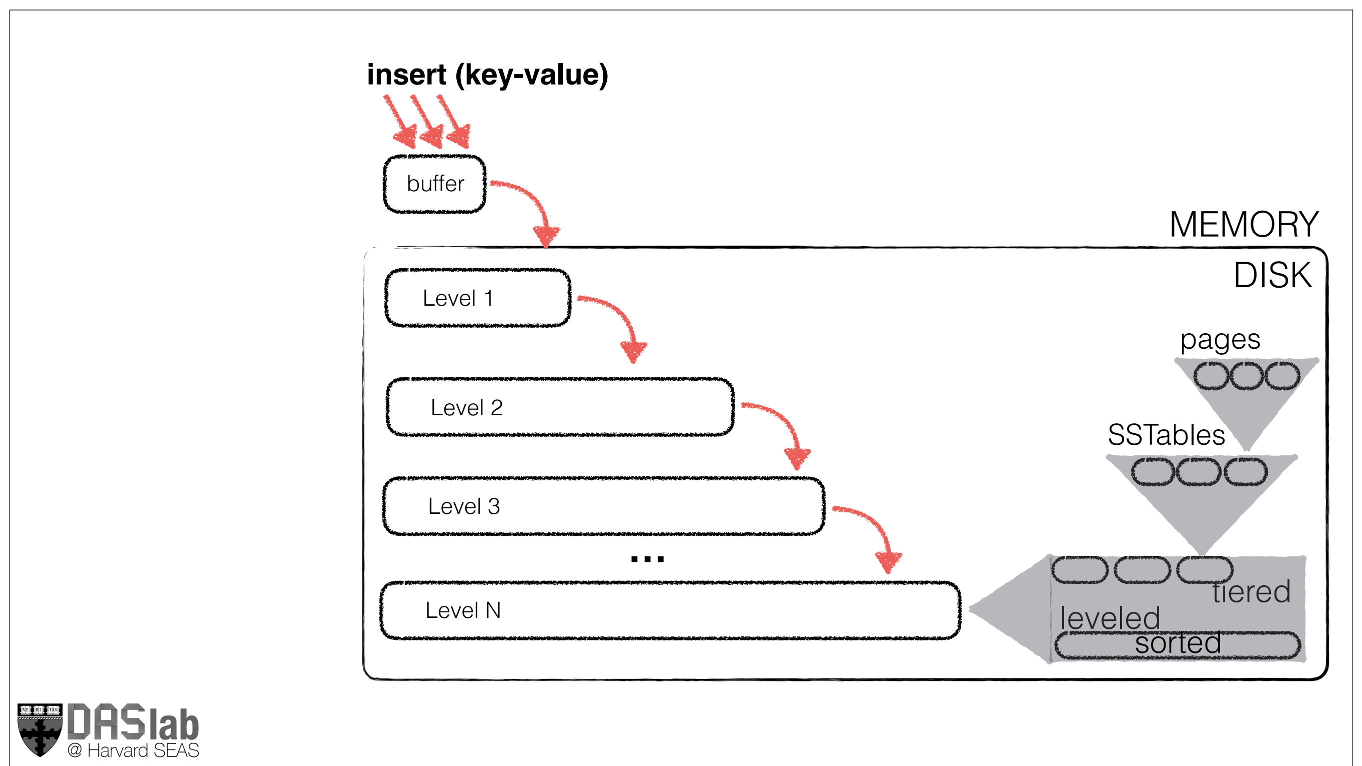

MEMORY
DISK
Level 1
Level 2
DASlab
@ Harvard SEAS



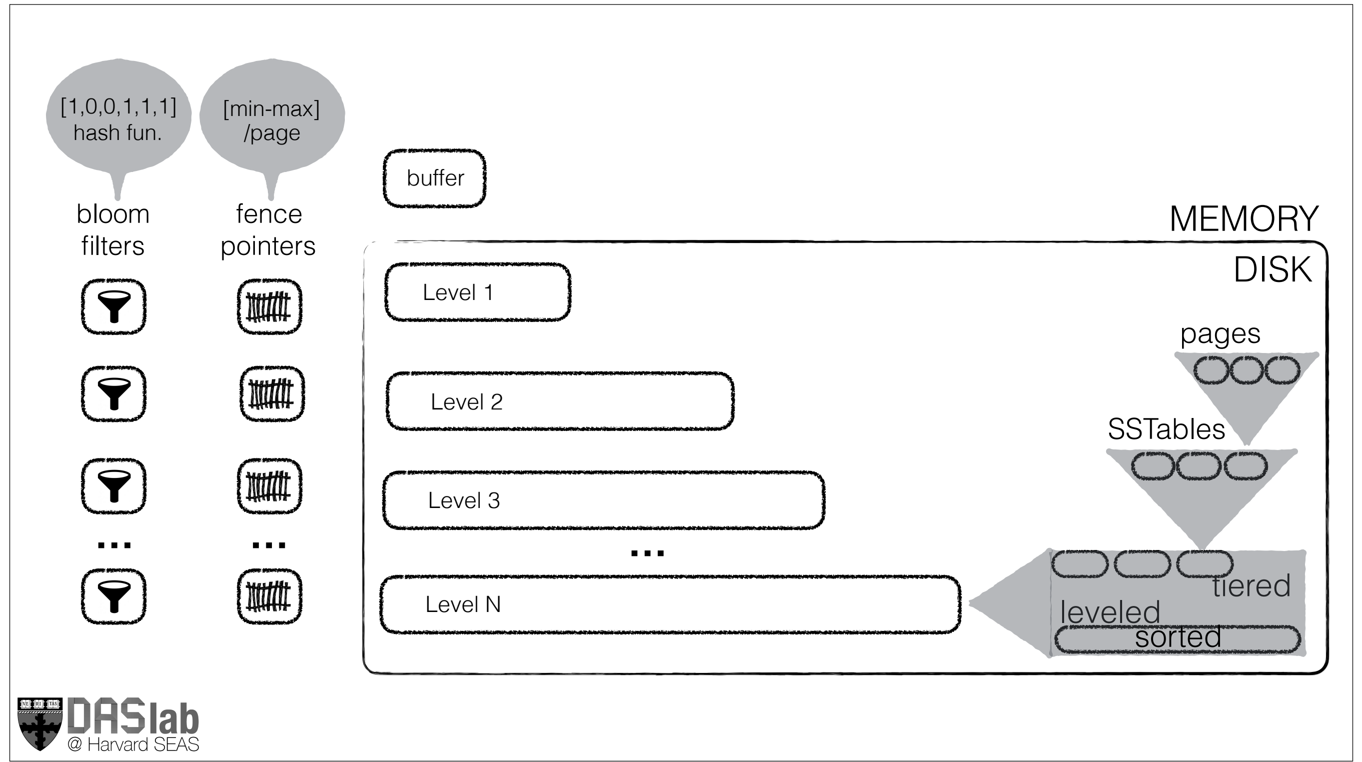

Overall we end up with a set of levels where data is temporally partitioned across the levels. In memory structures help reduce I/O at the cost of CPU cost and memory amplification. Bloom filters help us know if a data item may be present in a level (run) and fence pointers then tell us which page we should read. This means that a query will do at most one I/O per run to locate an item. We can reduce all LSM-tree designs to a small set of design principles that can be parameterized to give all existing designs.



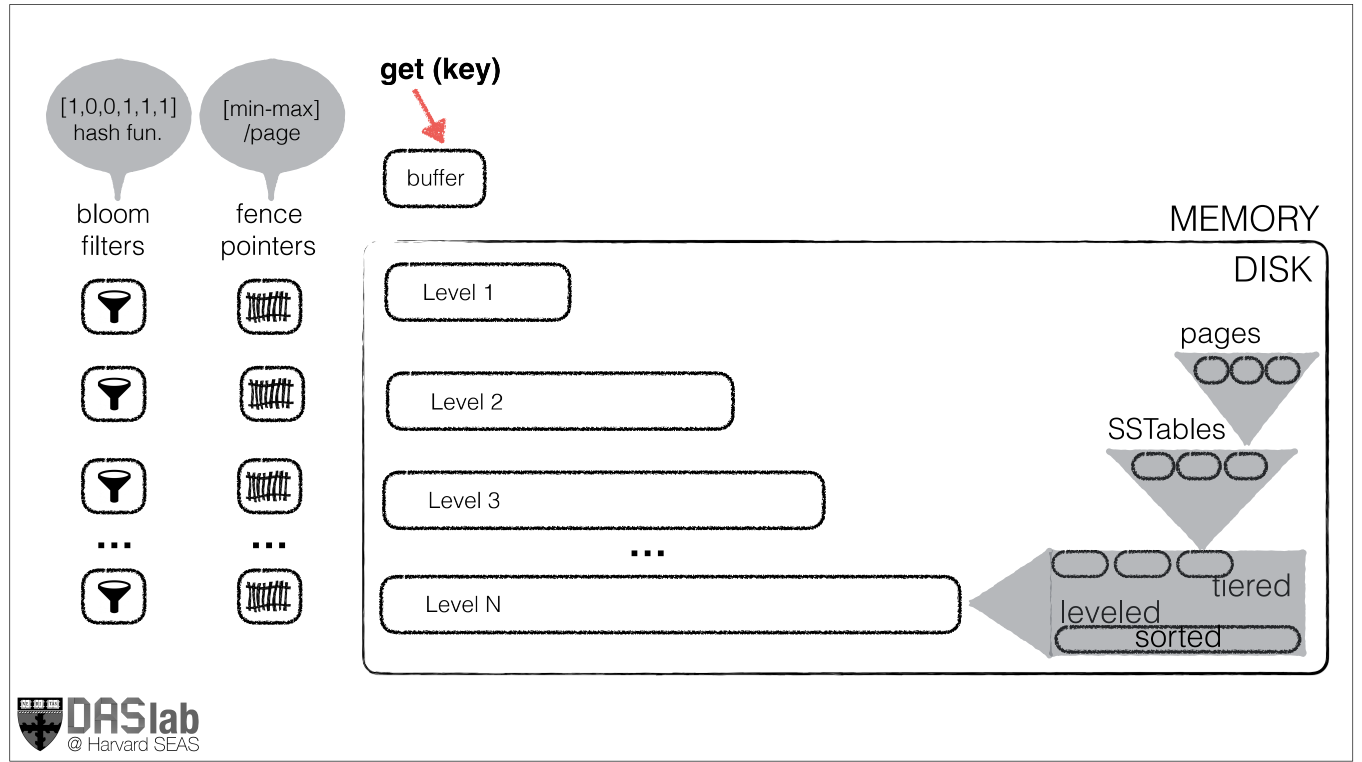

Overall we end up with a set of levels where data is temporally partitioned across the levels. In memory structures help reduce I/O at the cost of CPU cost and memory amplification. Bloom filters help us know if a data item may be present in a level (run) and fence pointers then tell us which page we should read. This means that a query will do at most one I/O per run to locate an item. We can reduce all LSM-tree designs to a small set of design principles that can be parameterized to give all existing designs.



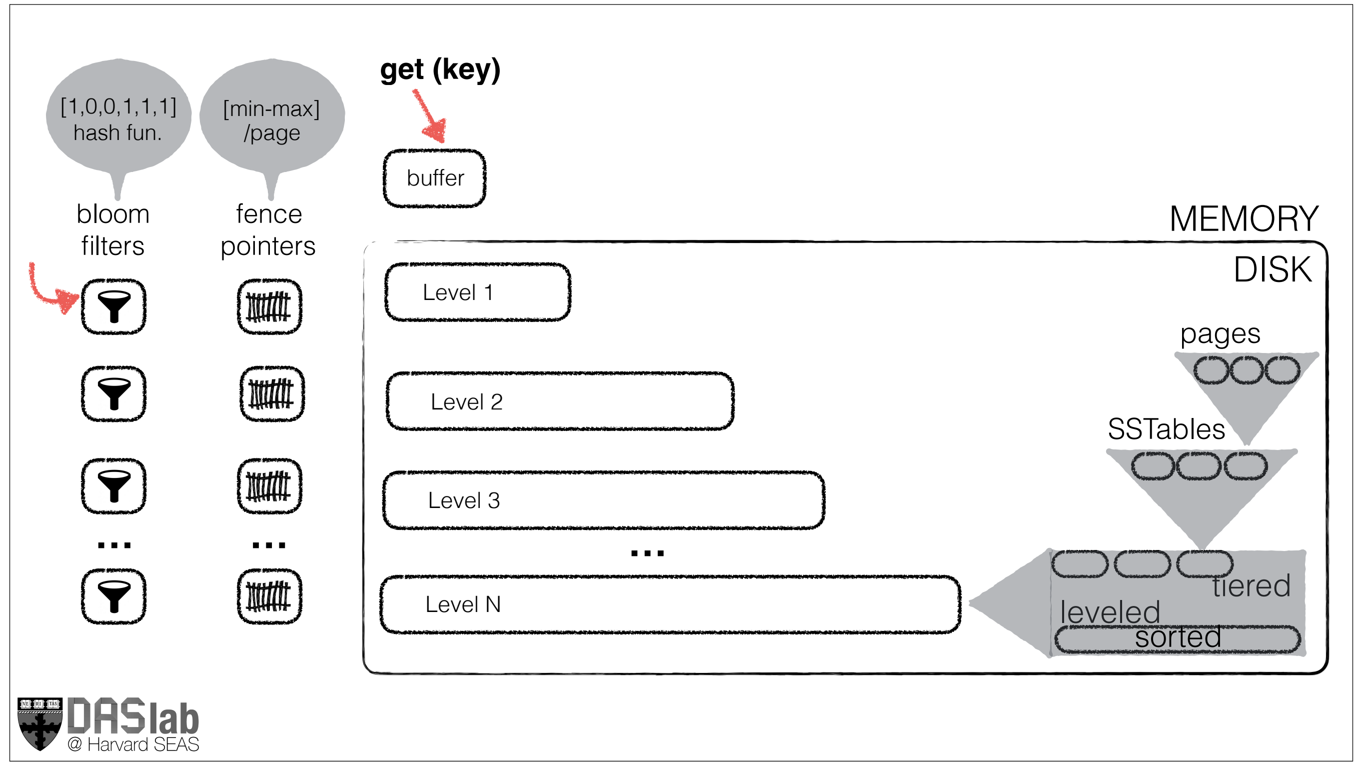

Overall we end up with a set of levels where data is temporally partitioned across the levels. In memory structures help reduce I/O at the cost of CPU cost and memory amplification. Bloom filters help us know if a data item may be present in a level (run) and fence pointers then tell us which page we should read. This means that a query will do at most one I/O per run to locate an item. We can reduce all LSM-tree designs to a small set of design principles that can be parameterized to give all existing designs.



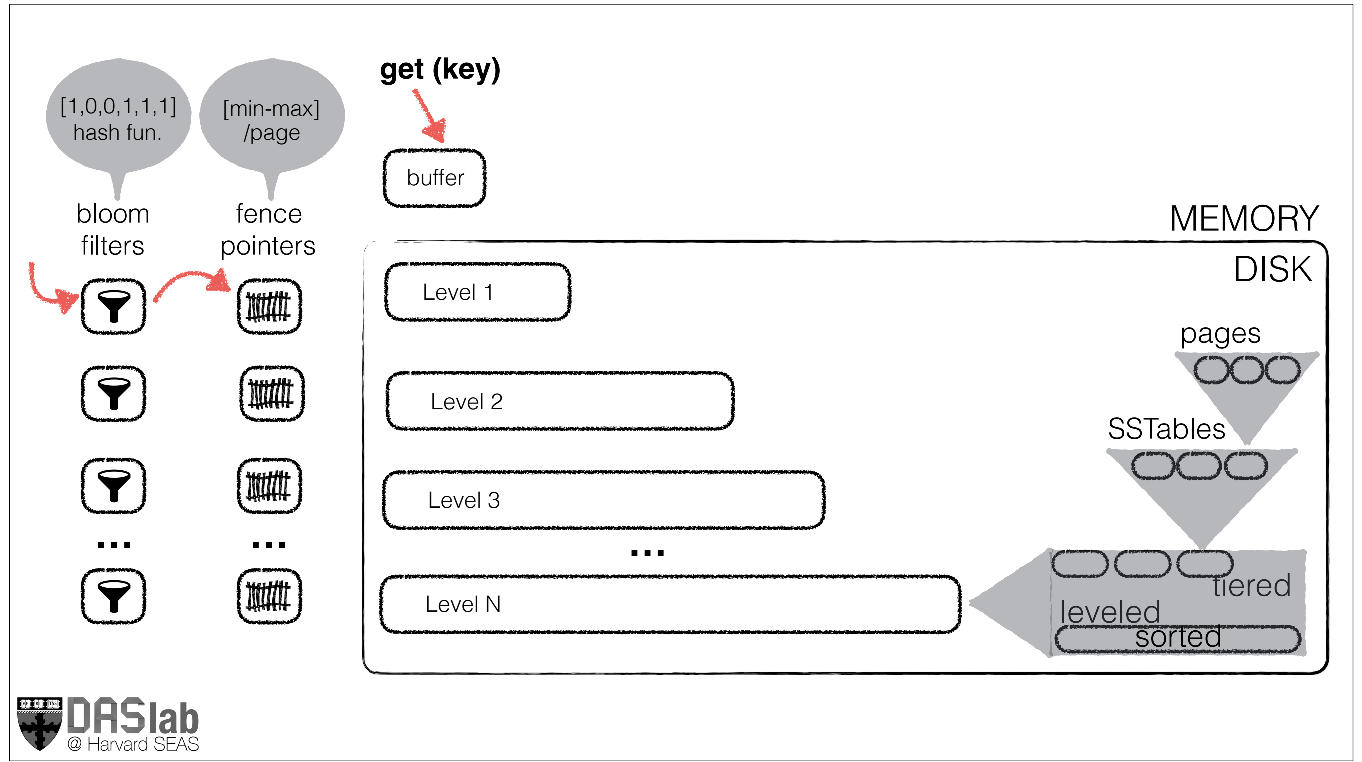

Overall we end up with a set of levels where data is temporally partitioned across the levels. In memory structures help reduce I/O at the cost of CPU cost and memory amplification. Bloom filters help us know if a data item may be present in a level (run) and fence pointers then tell us which page we should read. This means that a query will do at most one I/O per run to locate an item. We can reduce all LSM-tree designs to a small set of design principles that can be parameterized to give all existing designs.



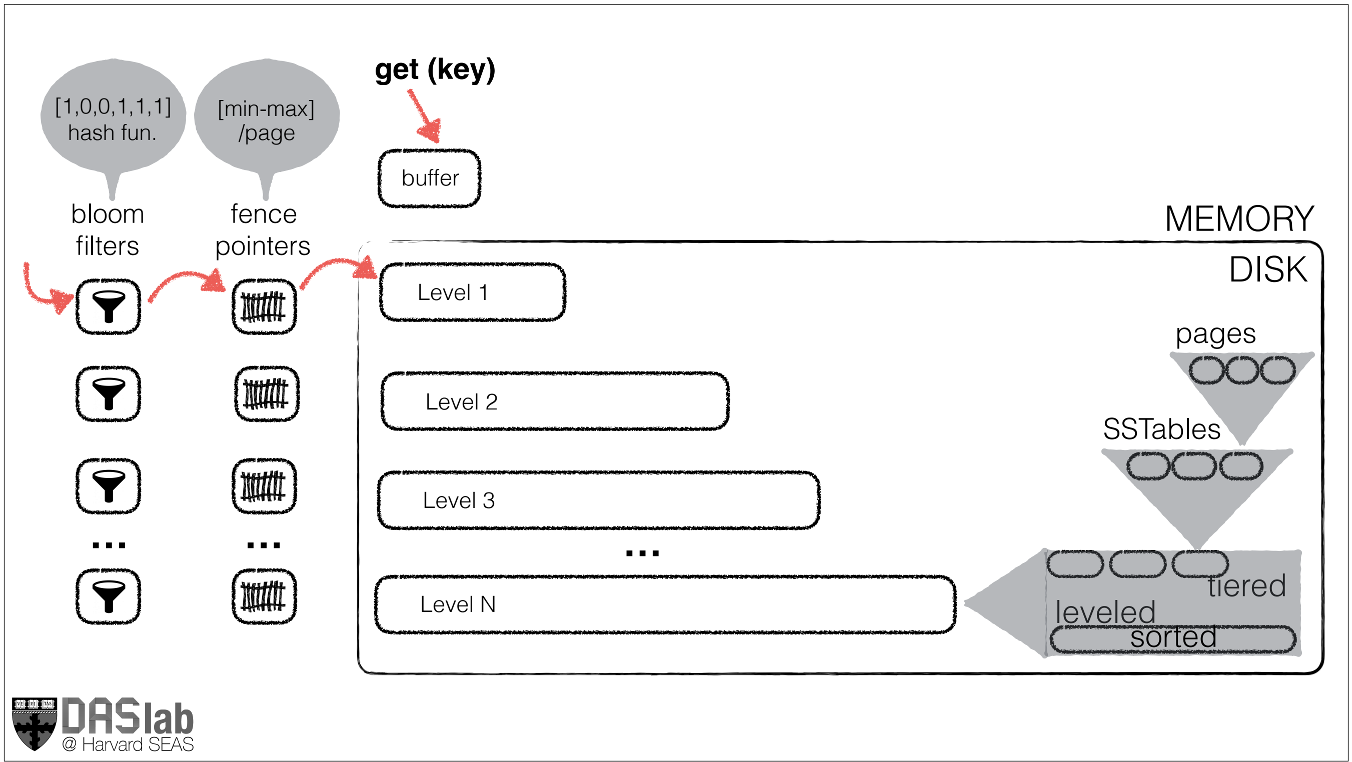

Overall we end up with a set of levels where data is temporally partitioned across the levels. In memory structures help reduce I/O at the cost of CPU cost and memory amplification. Bloom filters help us know if a data item may be present in a level (run) and fence pointers then tell us which page we should read. This means that a query will do at most one I/O per run to locate an item. We can reduce all LSM-tree designs to a small set of design principles that can be parameterized to give all existing designs.



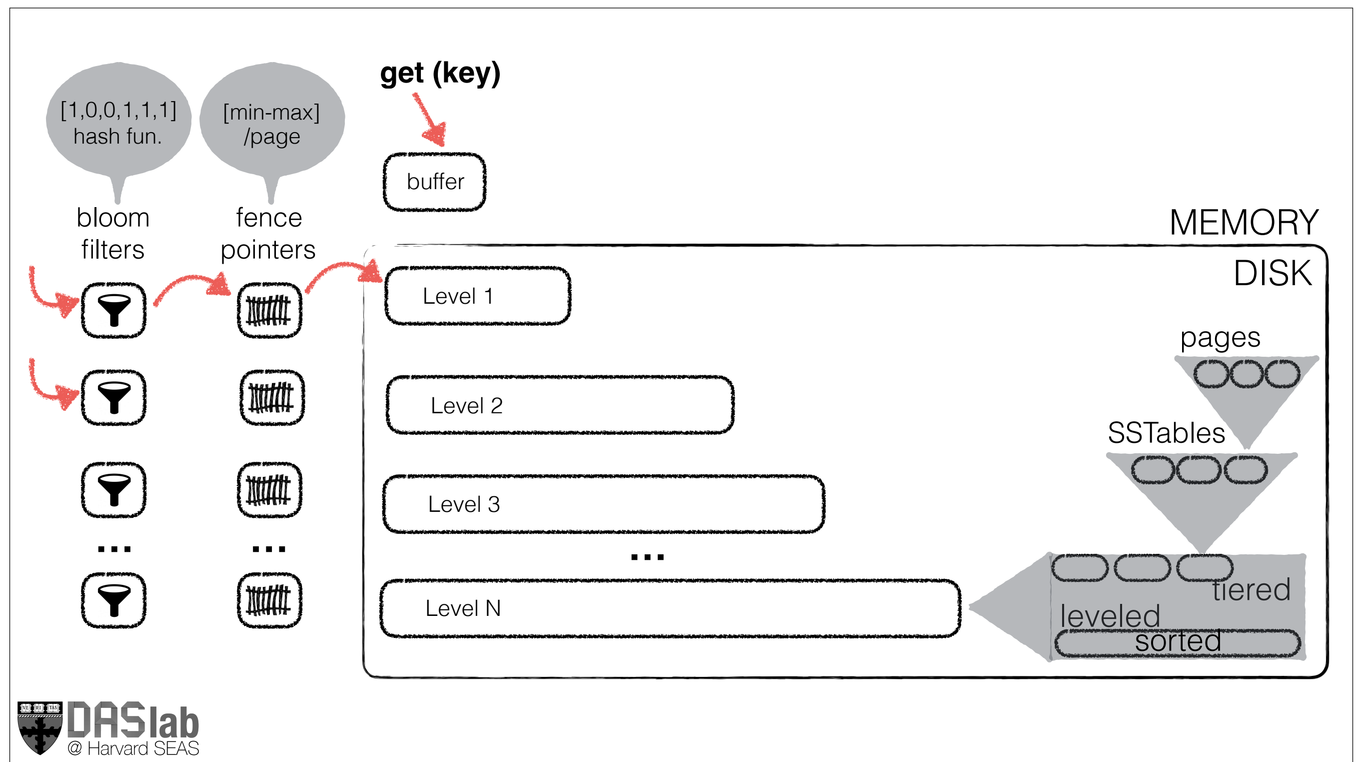

Overall we end up with a set of levels where data is temporally partitioned across the levels. In memory structures help reduce I/O at the cost of CPU cost and memory amplification. Bloom filters help us know if a data item may be present in a level (run) and fence pointers then tell us which page we should read. This means that a query will do at most one I/O per run to locate an item. We can reduce all LSM-tree designs to a small set of design principles that can be parameterized to give all existing designs.



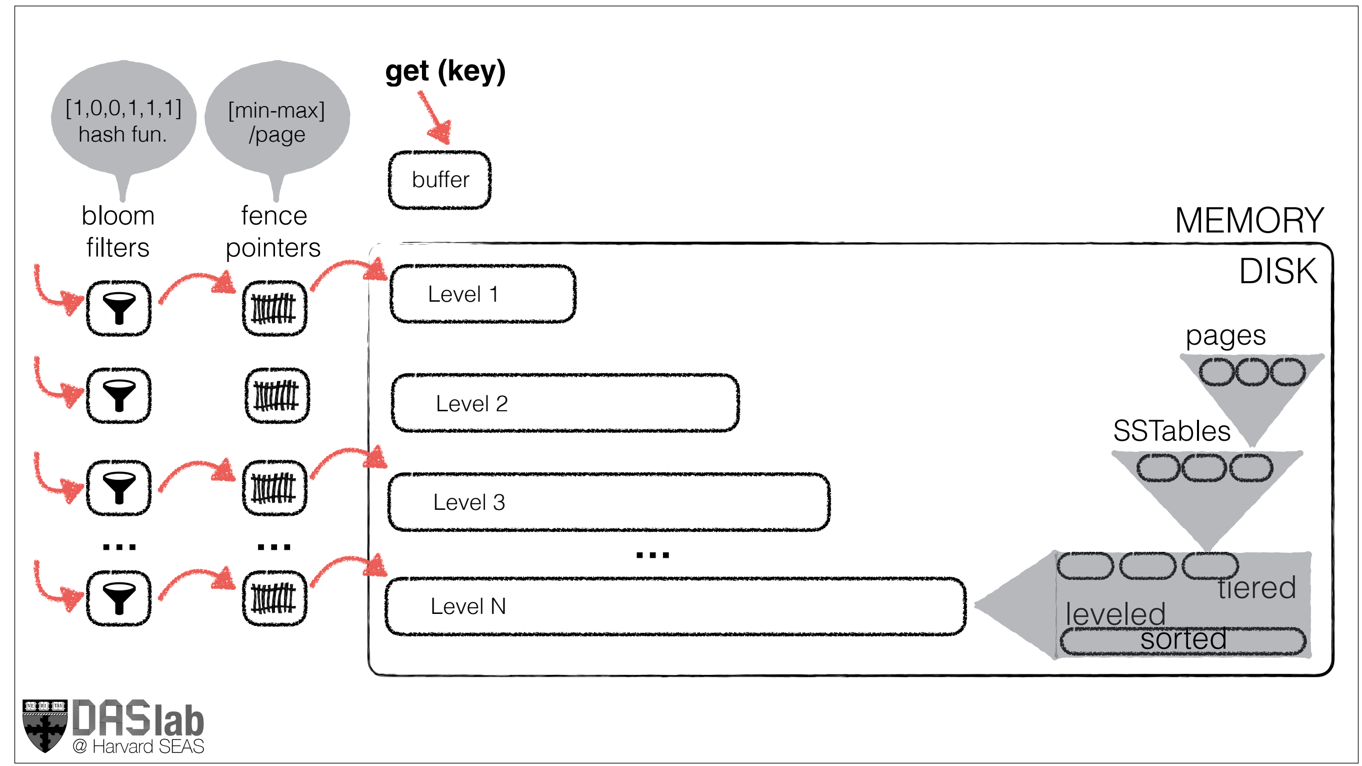

Overall we end up with a set of levels where data is temporally partitioned across the levels. In memory structures help reduce I/O at the cost of CPU cost and memory amplification. Bloom filters help us know if a data item may be present in a level (run) and fence pointers then tell us which page we should read. This means that a query will do at most one I/O per run to locate an item. We can reduce all LSM-tree designs to a small set of design principles that can be parameterized to give all existing designs.



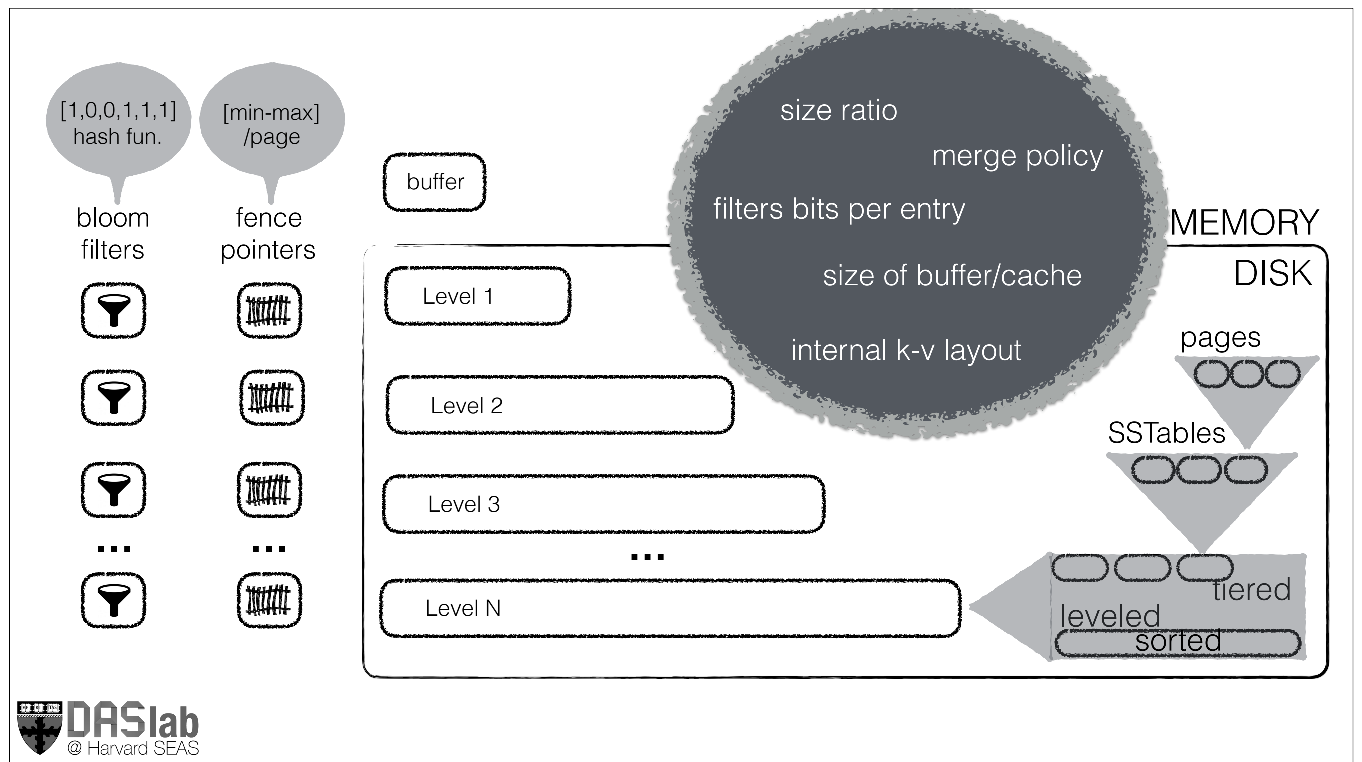

Overall we end up with a set of levels where data is temporally partitioned across the levels. In memory structures help reduce I/O at the cost of CPU cost and memory amplification. Bloom filters help us know if a data item may be present in a level (run) and fence pointers then tell us which page we should read. This means that a query will do at most one I/O per run to locate an item. We can reduce all LSM-tree designs to a small set of design principles that can be parameterized to give all existing designs.




Overall we end up with a set of levels where data is temporally partitioned across the levels. In memory structures help reduce I/O at the cost of CPU cost and memory amplification. Bloom filters help us know if a data item may be present in a level (run) and fence pointers then tell us which page we should read. This means that a query will do at most one I/O per run to locate an item. We can reduce all LSM-tree designs to a small set of design principles that can be parameterized to give all existing designs.

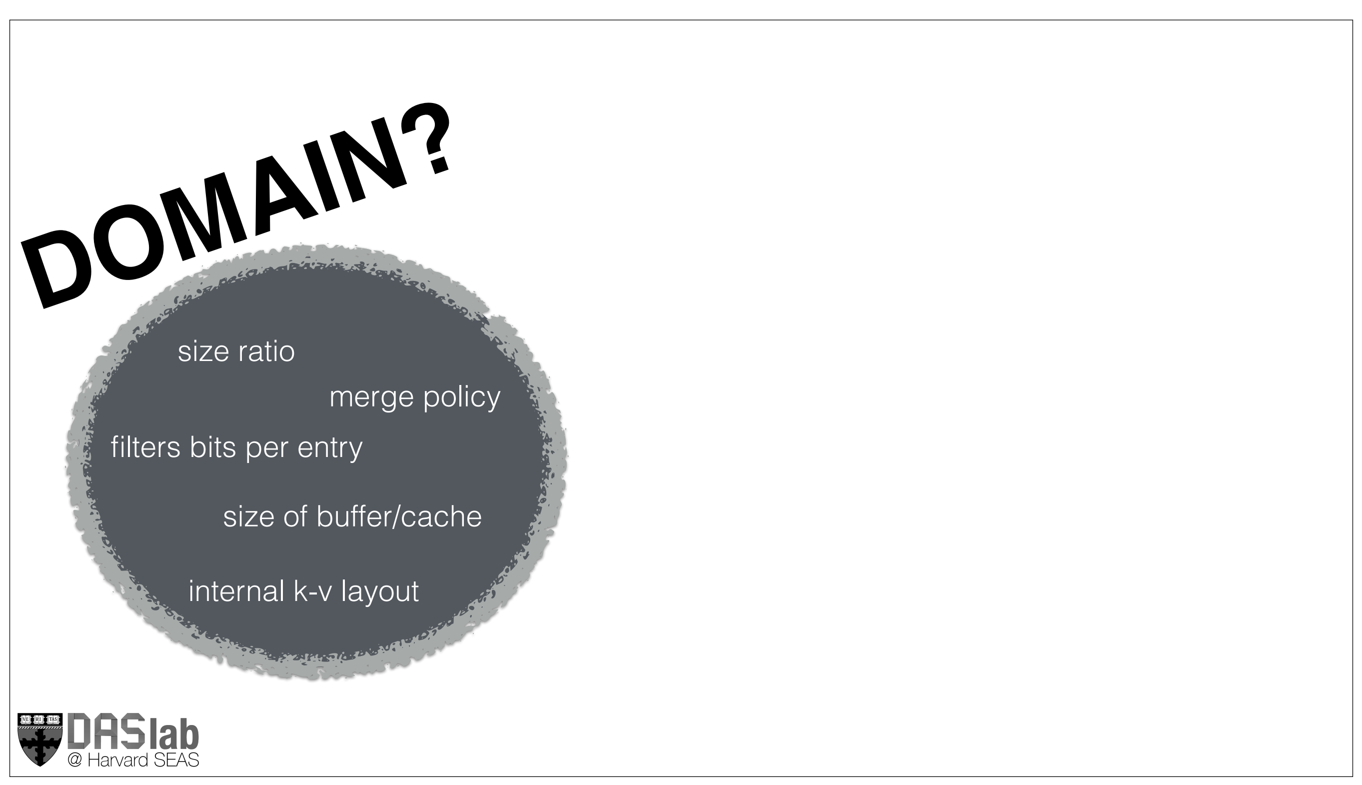



Overall we end up with a set of levels where data is temporally partitioned across the levels. In memory structures help reduce I/O at the cost of CPU cost and memory amplification. Bloom filters help us know if a data item may be present in a level (run) and fence pointers then tell us which page we should read. This means that a query will do at most one I/O per run to locate an item. We can reduce all LSM-tree designs to a small set of design principles that can be parameterized to give all existing designs.

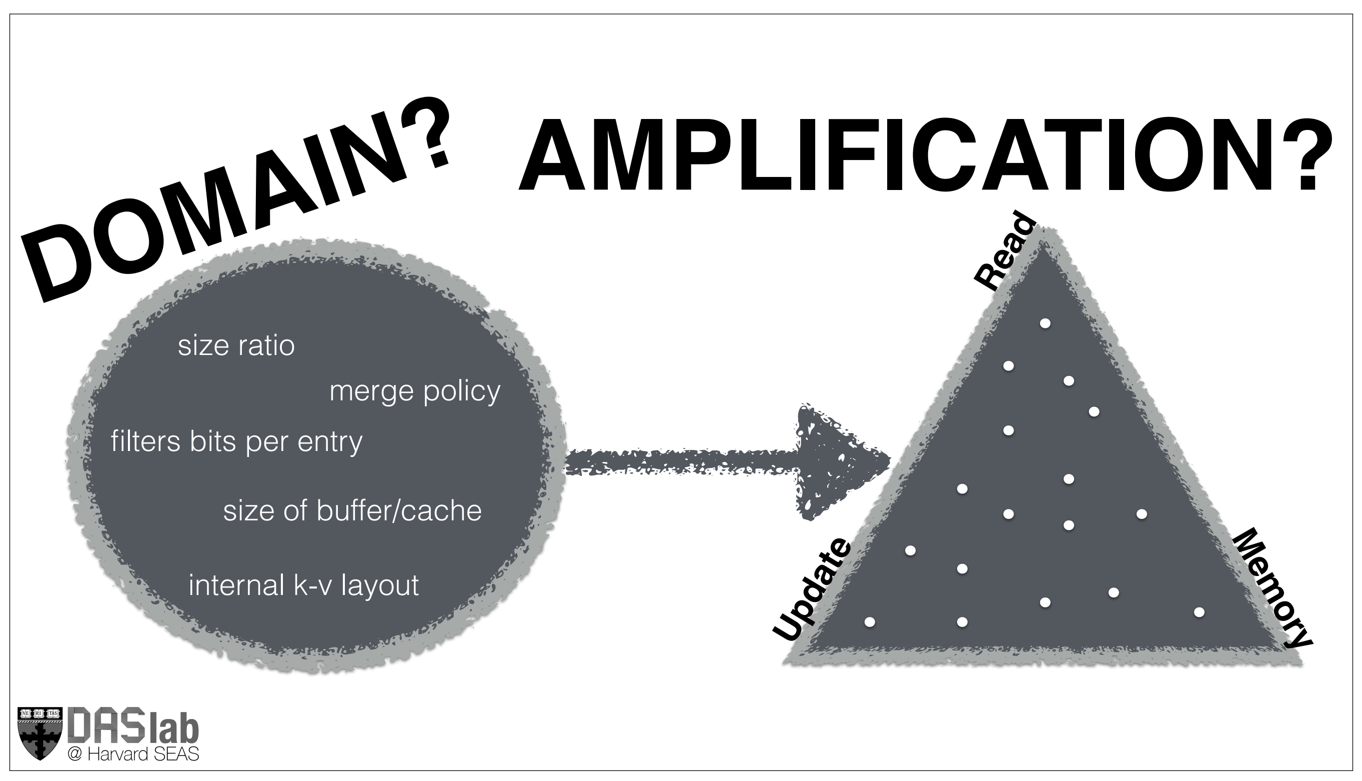



Overall we end up with a set of levels where data is temporally partitioned across the levels. In memory structures help reduce I/O at the cost of CPU cost and memory amplification. Bloom filters help us know if a data item may be present in a level (run) and fence pointers then tell us which page we should read. This means that a query will do at most one I/O per run to locate an item. We can reduce all LSM-tree designs to a small set of design principles that can be parameterized to give all existing designs.

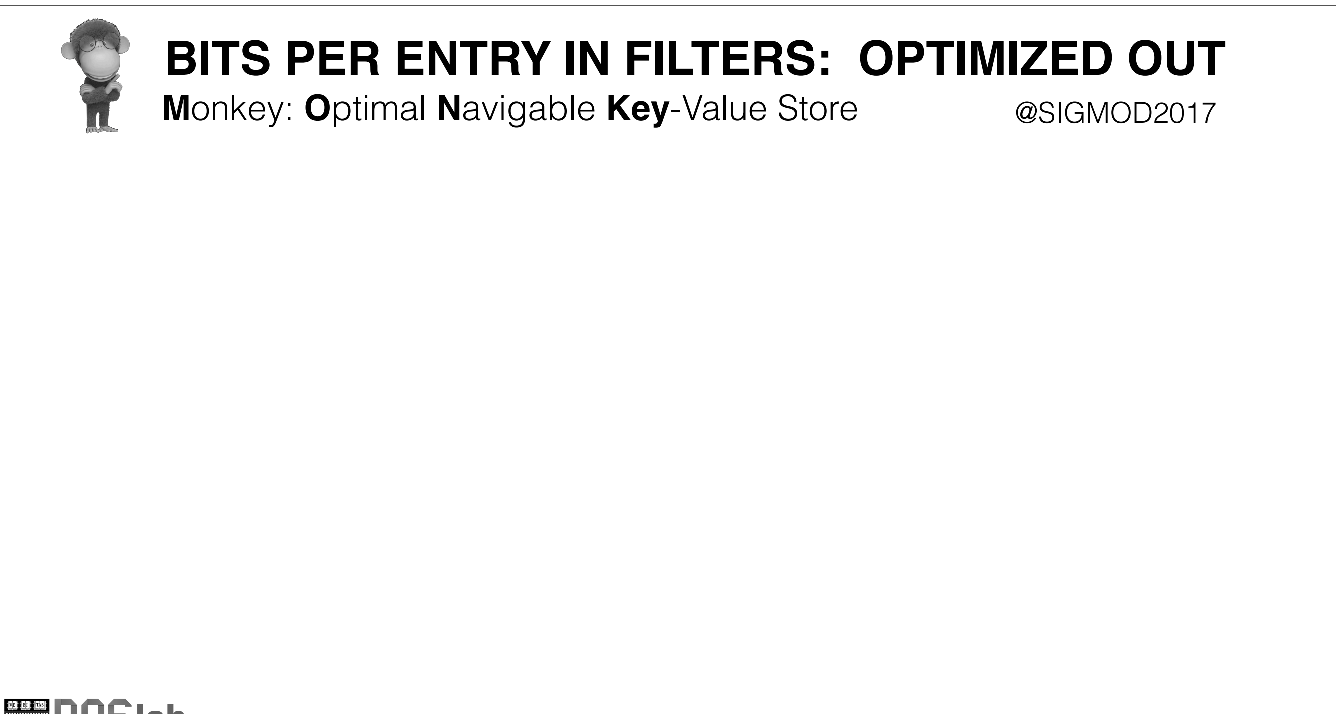
And then the goal is to understand how all combinations of those principles can be synthesized into arbitrary designs and where do these designs fall in the tradeoff space.



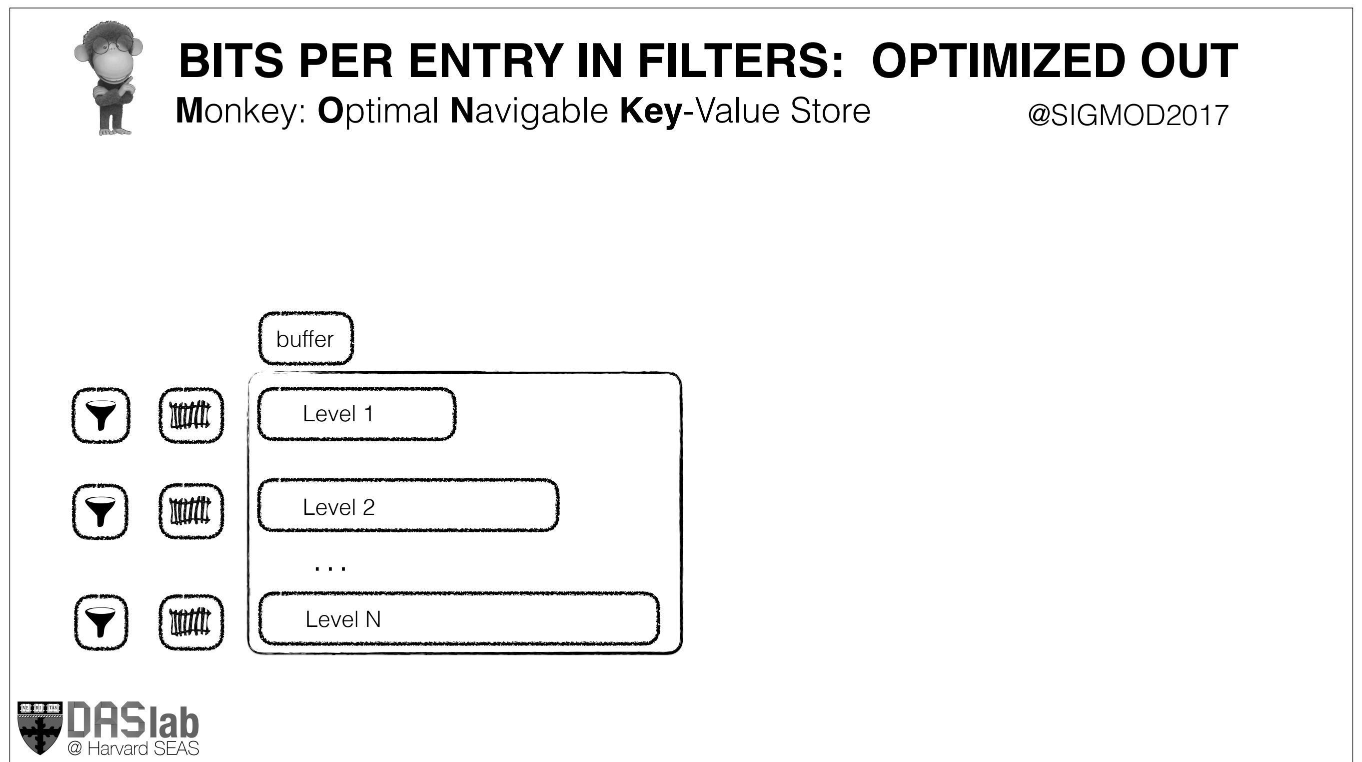

And then the goal is to understand how all combinations of those principles can be synthesized into arbitrary designs and where do these designs fall in the tradeoff space.



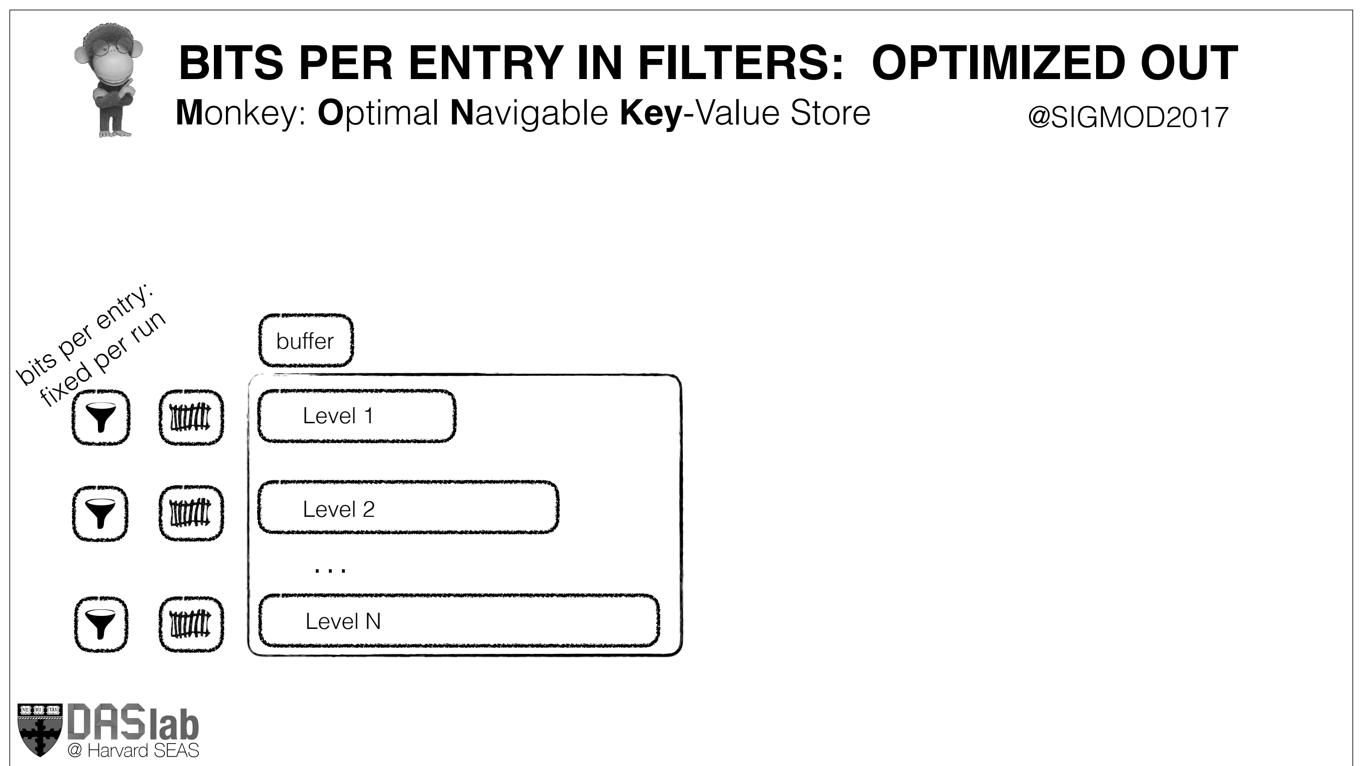

And then the goal is to understand how all combinations of those principles can be synthesized into arbitrary designs and where do these designs fall in the tradeoff space.



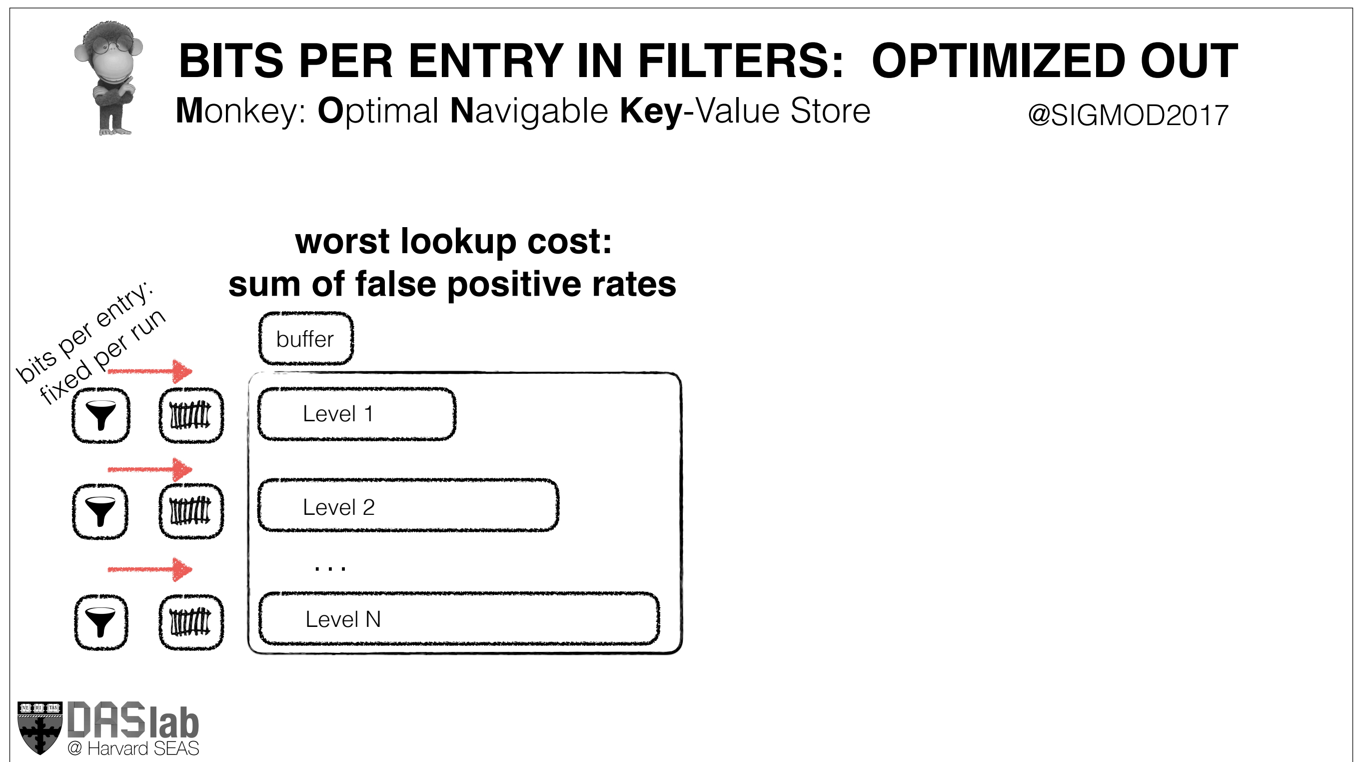

An example of a design principle that can be optimized out is the number of bits per entry in the bloom filters of an LSM-tree. We can actually write down a closed form formula that dictates what is the optimal number of bits. The trick is that it is not a fixed number for every level. In fact smaller levels should have exponentially more bits per entry compared to bigger levels. This is because every level contributes in the same way to the total cost (1 I/O per run) and it is much more beneficial to spend the memory budget at the small levels where every bit can give a bigger boost in terms of the false positive rate.



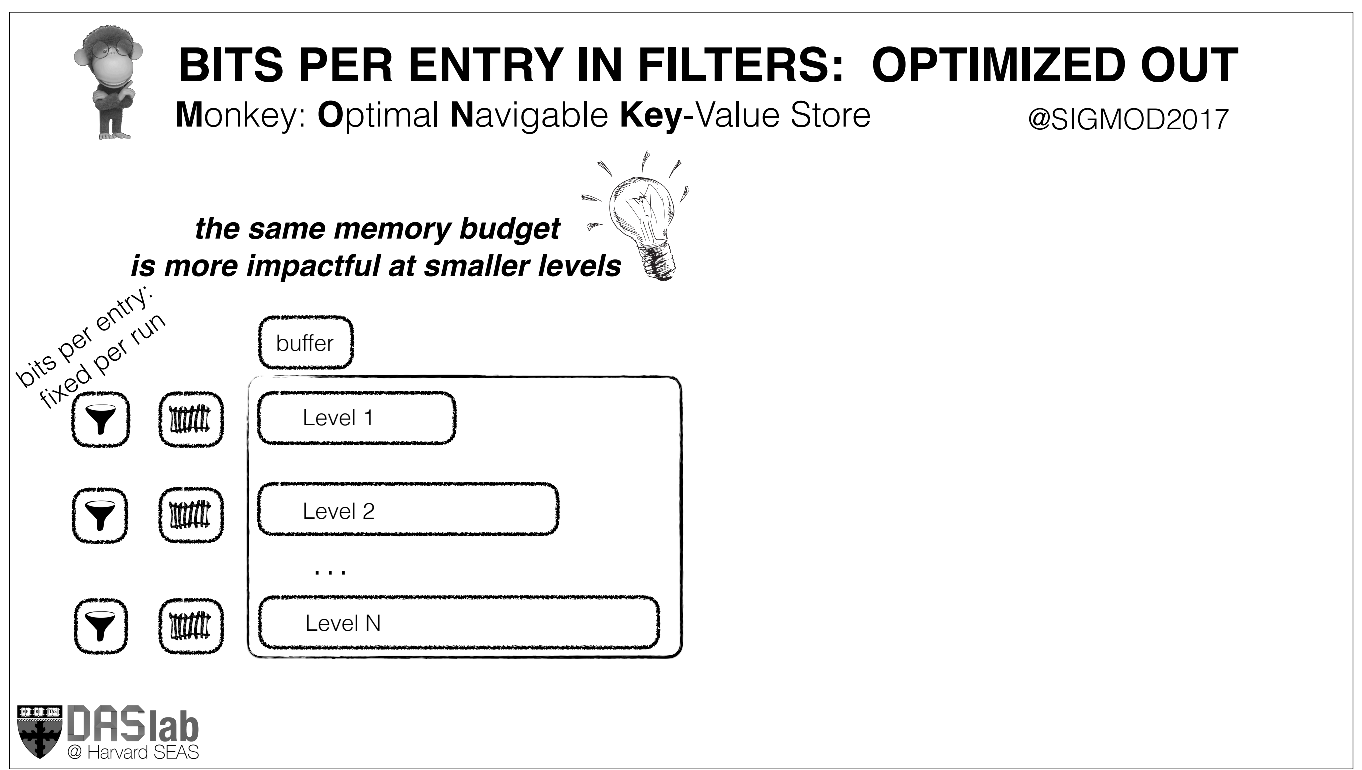

An example of a design principle that can be optimized out is the number of bits per entry in the bloom filters of an LSM-tree. We can actually write down a closed form formula that dictates what is the optimal number of bits. The trick is that it is not a fixed number for every level. In fact smaller levels should have exponentially more bits per entry compared to bigger levels. This is because every level contributes in the same way to the total cost (1 I/O per run) and it is much more beneficial to spend the memory budget at the small levels where every bit can give a bigger boost in terms of the false positive rate.



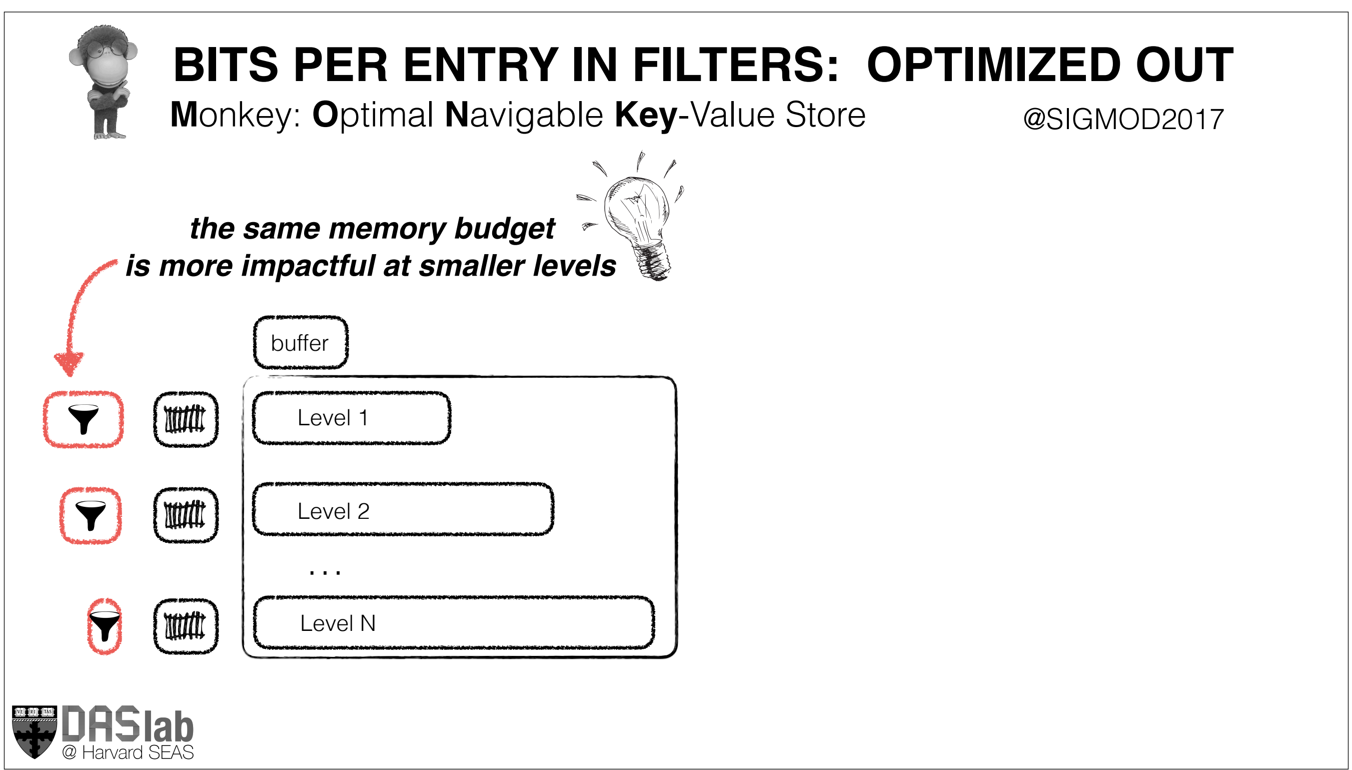

An example of a design principle that can be optimized out is the number of bits per entry in the bloom filters of an LSM-tree. We can actually write down a closed form formula that dictates what is the optimal number of bits. The trick is that it is not a fixed number for every level. In fact smaller levels should have exponentially more bits per entry compared to bigger levels. This is because every level contributes in the same way to the total cost (1 I/O per run) and it is much more beneficial to spend the memory budget at the small levels where every bit can give a bigger boost in terms of the false positive rate.



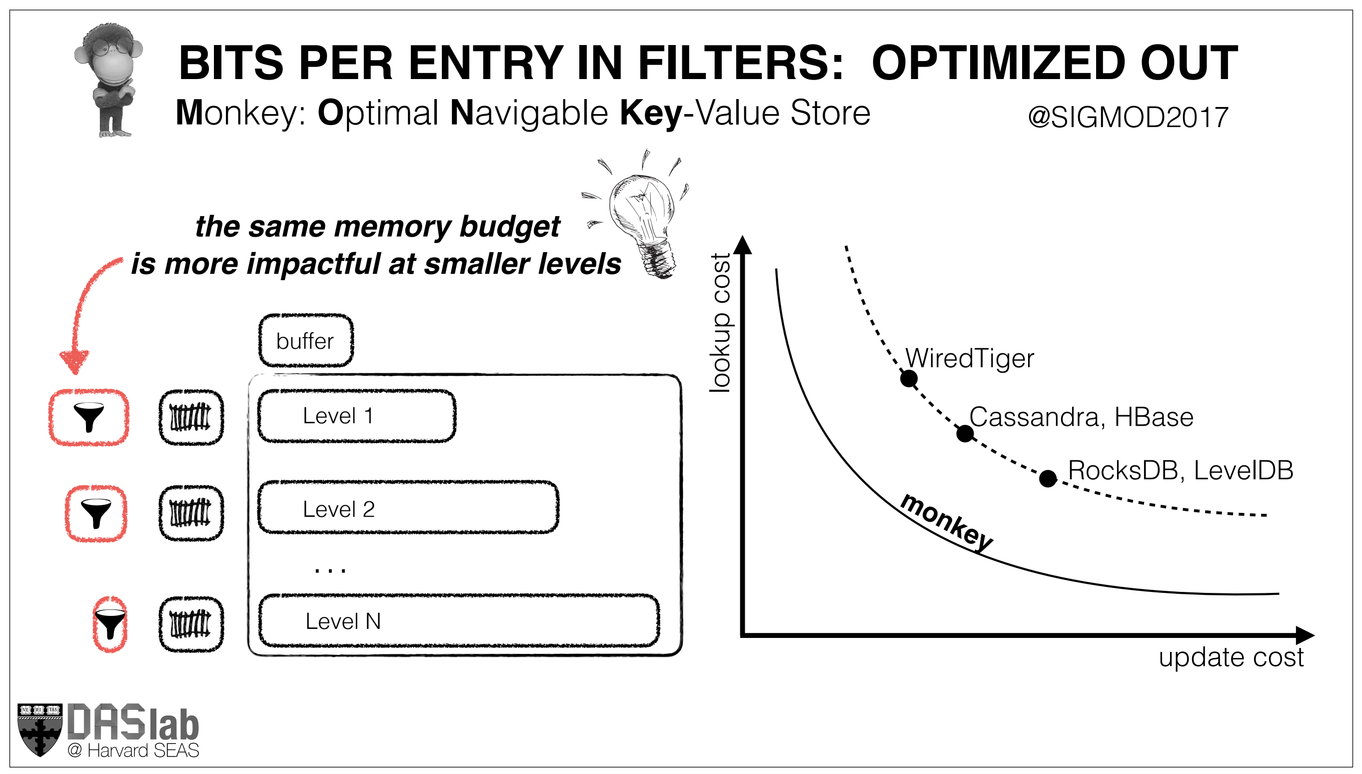

An example of a design principle that can be optimized out is the number of bits per entry in the bloom filters of an LSM-tree. We can actually write down a closed form formula that dictates what is the optimal number of bits. The trick is that it is not a fixed number for every level. In fact smaller levels should have exponentially more bits per entry compared to bigger levels. This is because every level contributes in the same way to the total cost (1 I/O per run) and it is much more beneficial to spend the memory budget at the small levels where every bit can give a bigger boost in terms of the false positive rate.



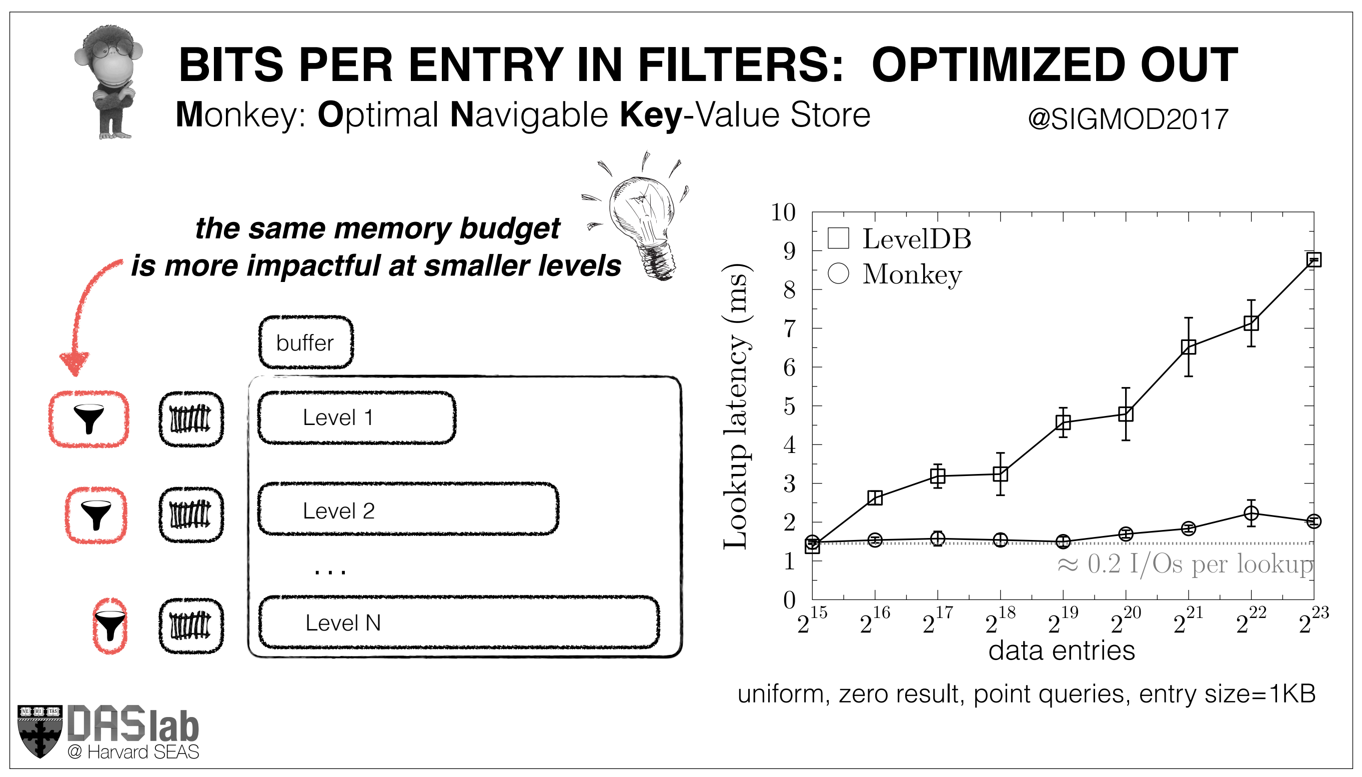

An example of a design principle that can be optimized out is the number of bits per entry in the bloom filters of an LSM-tree. We can actually write down a closed form formula that dictates what is the optimal number of bits. The trick is that it is not a fixed number for every level. In fact smaller levels should have exponentially more bits per entry compared to bigger levels. This is because every level contributes in the same way to the total cost (1 I/O per run) and it is much more beneficial to spend the memory budget at the small levels where every bit can give a bigger boost in terms of the false positive rate.




An example of a design principle that can be optimized out is the number of bits per entry in the bloom filters of an LSM-tree. We can actually write down a closed form formula that dictates what is the optimal number of bits. The trick is that it is not a fixed number for every level. In fact smaller levels should have exponentially more bits per entry compared to bigger levels. This is because every level contributes in the same way to the total cost (1 I/O per run) and it is much more beneficial to spend the memory budget at the small levels where every bit can give a bigger boost in terms of the false positive rate.




An example of a design principle that can be optimized out is the number of bits per entry in the bloom filters of an LSM-tree. We can actually write down a closed form formula that dictates what is the optimal number of bits. The trick is that it is not a fixed number for every level. In fact smaller levels should have exponentially more bits per entry compared to bigger levels. This is because every level contributes in the same way to the total cost (1 I/O per run) and it is much more beneficial to spend the memory budget at the small levels where every bit can give a bigger boost in terms of the false positive rate.




An example of a design principle that can be optimized out is the number of bits per entry in the bloom filters of an LSM-tree. We can actually write down a closed form formula that dictates what is the optimal number of bits. The trick is that it is not a fixed number for every level. In fact smaller levels should have exponentially more bits per entry compared to bigger levels. This is because every level contributes in the same way to the total cost (1 I/O per run) and it is much more beneficial to spend the memory budget at the small levels where every bit can give a bigger boost in terms of the false positive rate.

Other decisions such as the number of runs per level need to remain as critical design option and we need to tune it specifically for the problem at hand. The investigation of the design principles and the possible combinations reveals new possible designs by considering all possible ways to synthesize the design principles. In this case, we find that having an arbitrary number of runs tuned specifically for every level of the tree gives the flexibility to match the workload exactly.

# MERGE POLICY: SHOULD BE TUNED

@SIGMOD2018

**Do**stoevsky: **S**pace-**T**ime **O**ptimized **Ev**olvable **S**calable **K**ey-Value Store

buffer

Level 1

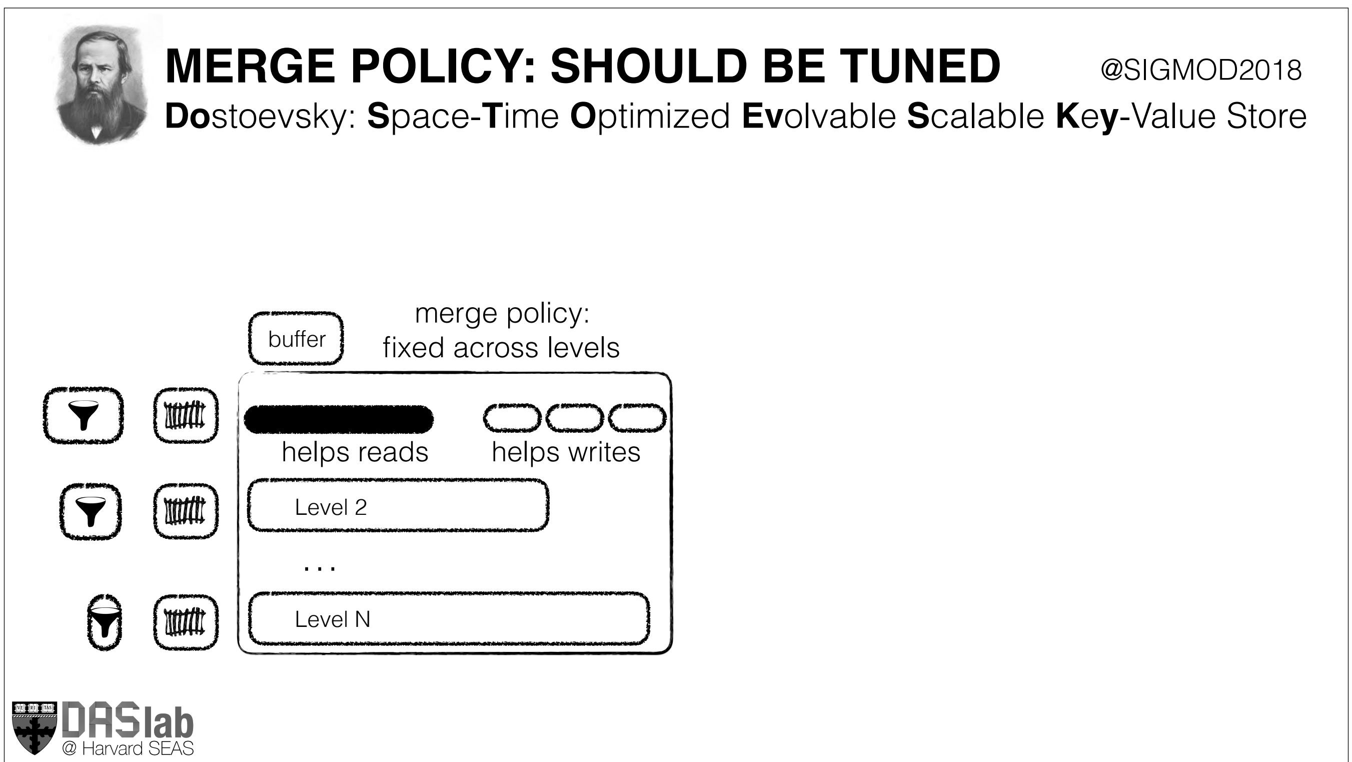
Level 2

...

Level N

DASlab @ Harvard SEAS

Other decisions such as the number of runs per level need to remain as critical design option and we need to tune it specifically for the problem at hand. The investigation of the design principles and the possible combinations reveals new possible designs by considering all possible ways to synthesize the design principles. In this case, we find that having an arbitrary number of runs tuned specifically for every level of the tree gives the flexibility to match the workload exactly.




Other decisions such as the number of runs per level need to remain as critical design option and we need to tune it specifically for the problem at hand. The investigation of the design principles and the possible combinations reveals new possible designs by considering all possible ways to synthesize the design principles. In this case, we find that having an arbitrary number of runs tuned specifically for every level of the tree gives the flexibility to match the workload exactly.

# MERGE POLICY: SHOULD BE TUNED

@SIGMOD2018

**Do**stoevsky: **S**pace-**T**ime **O**ptimized **Ev**olvable **S**calable **K**ey-Value Store

buffer

merge policy:
fixed across levels

Level 1

Level 2

. . .

Level N

DASlab @ Harvard SEAS

Other decisions such as the number of runs per level need to remain as critical design option and we need to tune it specifically for the problem at hand. The investigation of the design principles and the possible combinations reveals new possible designs by considering all possible ways to synthesize the design principles. In this case, we find that having an arbitrary number of runs tuned specifically for every level of the tree gives the flexibility to match the workload exactly.

## MERGE POLICY: SHOULD BE TUNED

@SIGMOD2018

**Do**stoevsky: **S**pace-**T**ime **O**ptimized **Ev**olvable **S**calable **K**e**y**-Value Store

***merging small levels does not help that much (point,range,space)***

buffer

merge policy: fixed across levels

Level 1

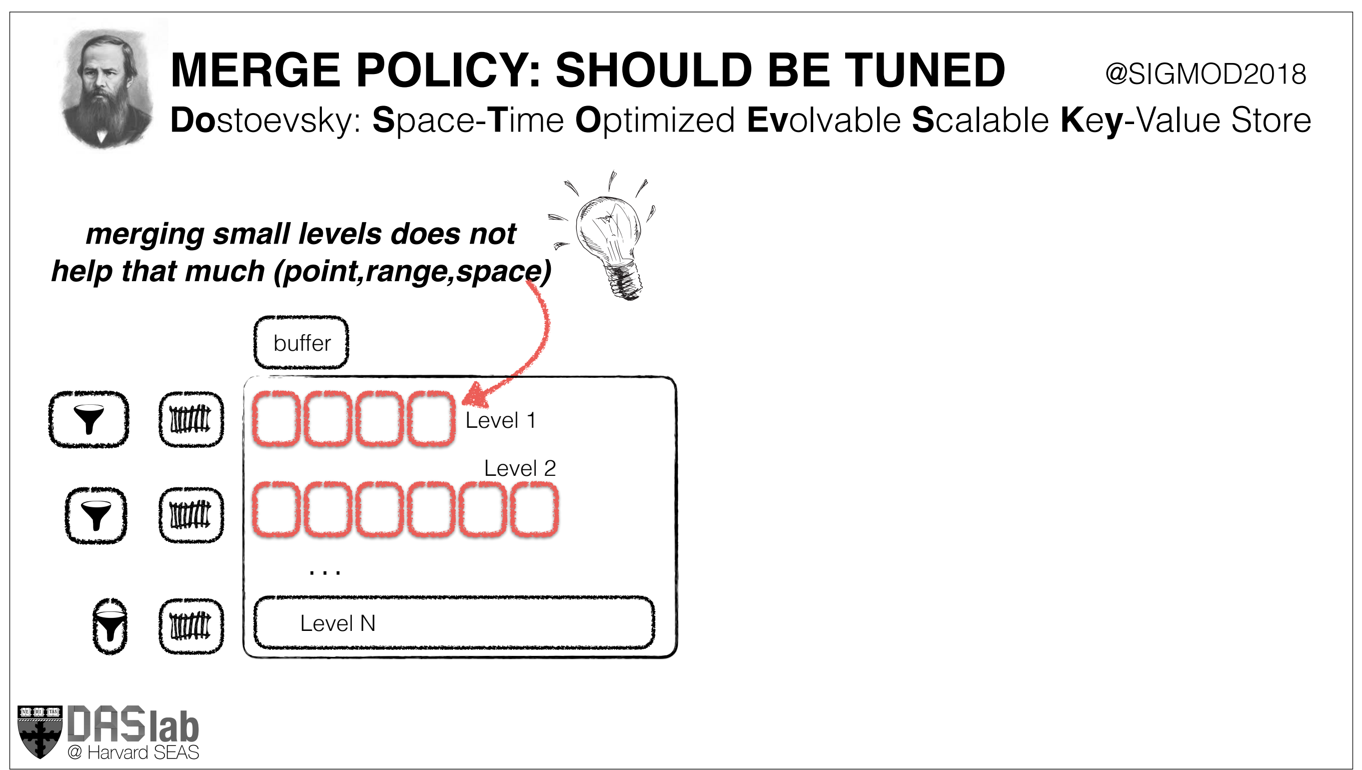
Level 2

...

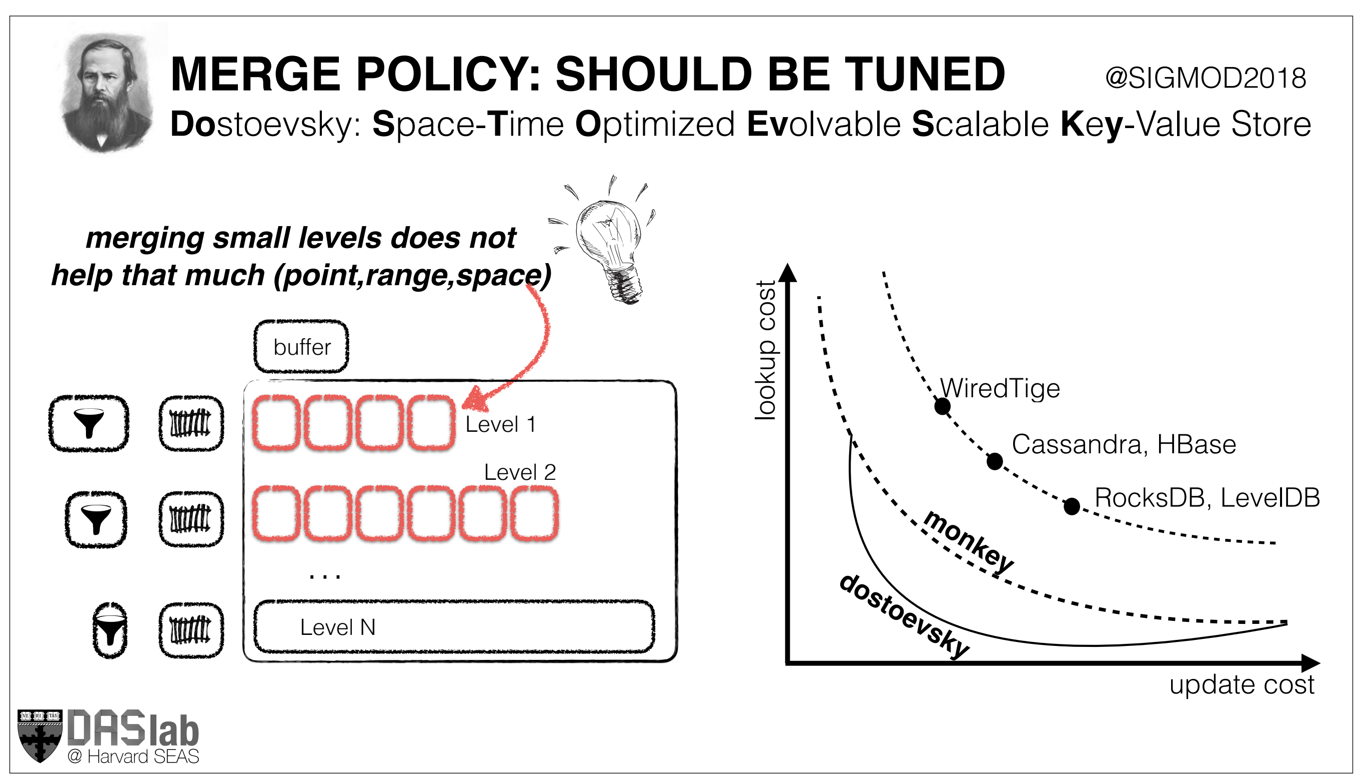
Level N

DASlab @ Harvard SEAS

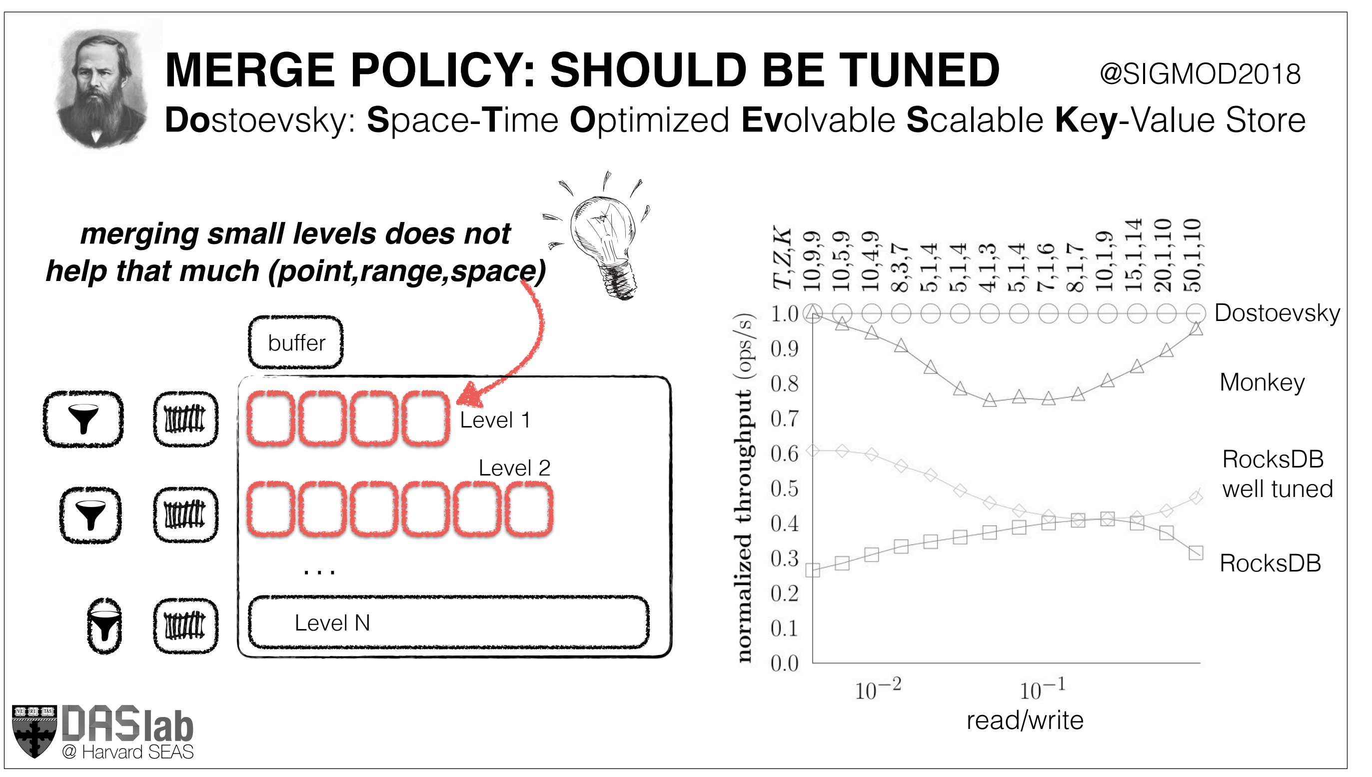
Other decisions such as the number of runs per level need to remain as critical design option and we need to tune it specifically for the problem at hand. The investigation of the design principles and the possible combinations reveals new possible designs by considering all possible ways to synthesize the design principles. In this case, we find that having an arbitrary number of runs tuned specifically for every level of the tree gives the flexibility to match the workload exactly.




Other decisions such as the number of runs per level need to remain as critical design option and we need to tune it specifically for the problem at hand. The investigation of the design principles and the possible combinations reveals new possible designs by considering all possible ways to synthesize the design principles. In this case, we find that having an arbitrary number of runs tuned specifically for every level of the tree gives the flexibility to match the workload exactly.

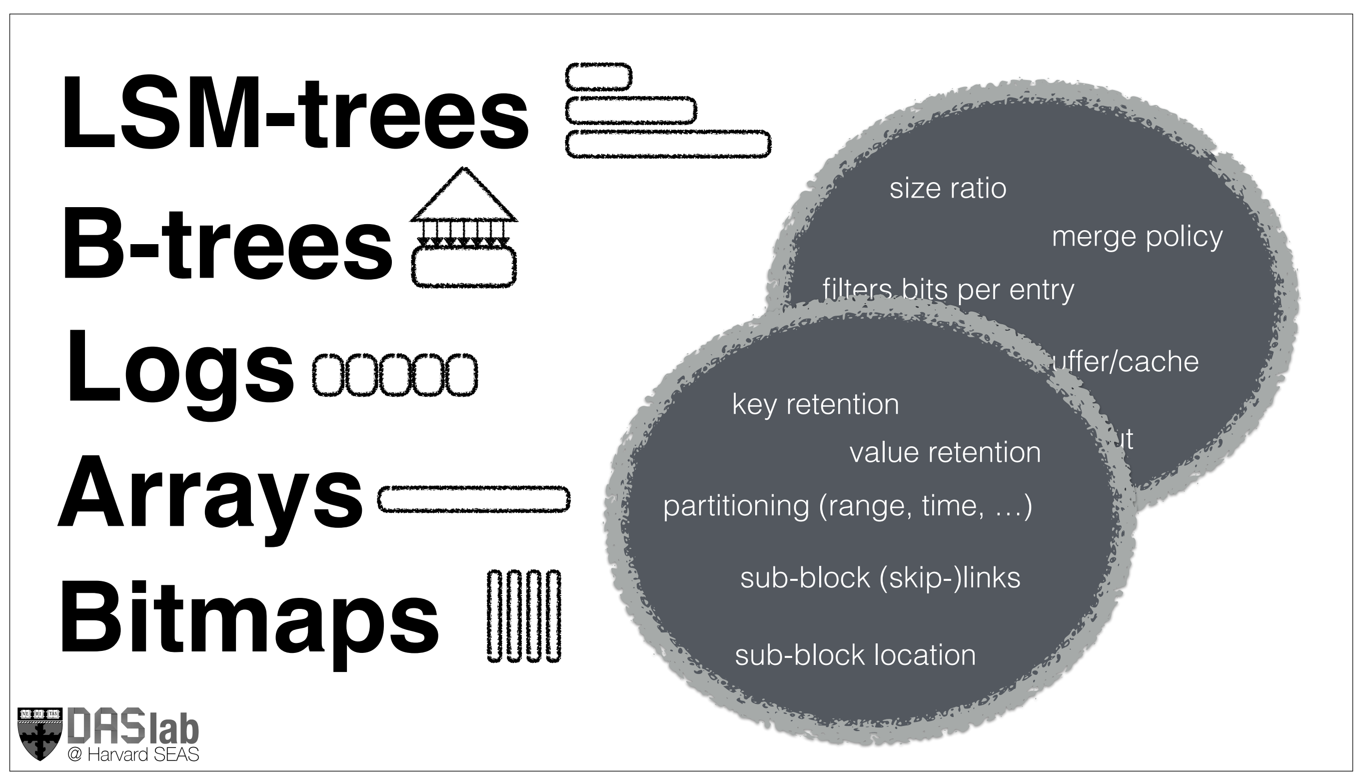



Other decisions such as the number of runs per level need to remain as critical design option and we need to tune it specifically for the problem at hand. The investigation of the design principles and the possible combinations reveals new possible designs by considering all possible ways to synthesize the design principles. In this case, we find that having an arbitrary number of runs tuned specifically for every level of the tree gives the flexibility to match the workload exactly.

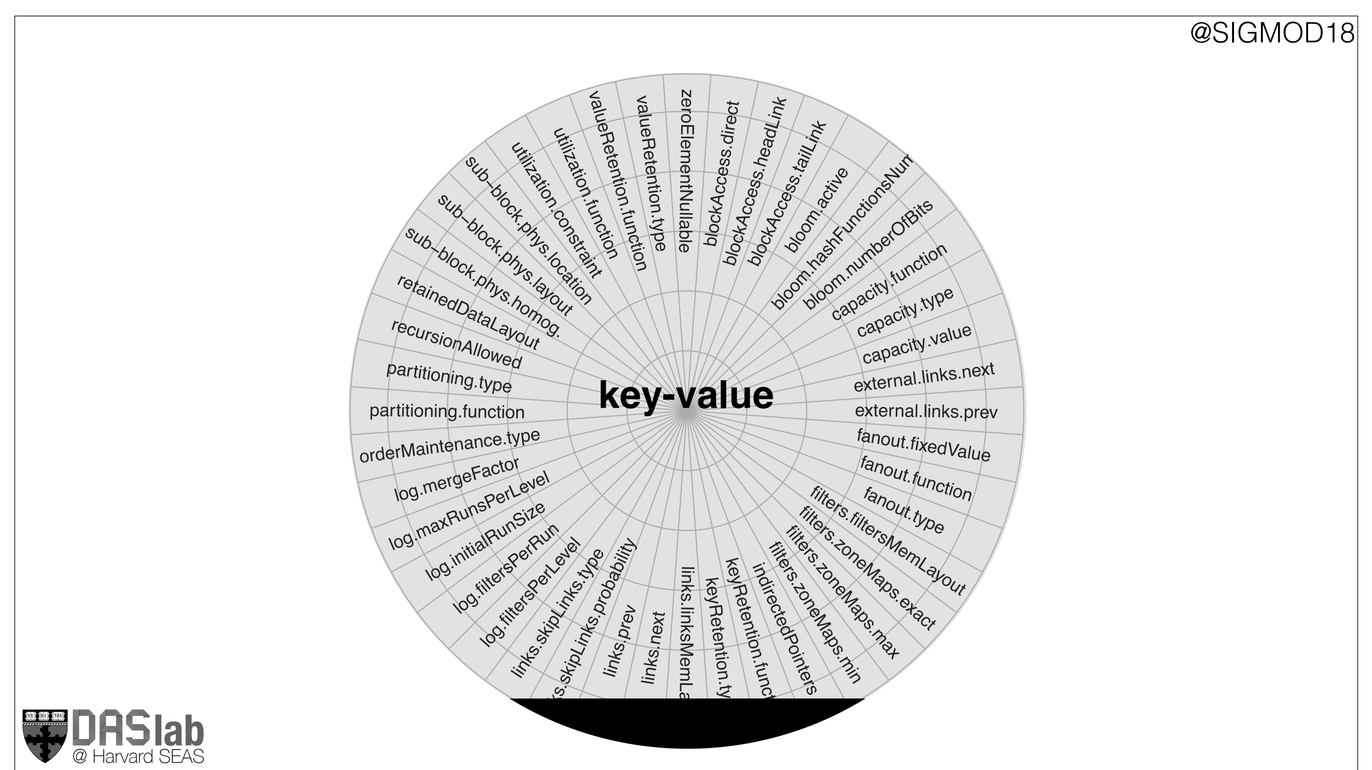



Other decisions such as the number of runs per level need to remain as critical design option and we need to tune it specifically for the problem at hand. The investigation of the design principles and the possible combinations reveals new possible designs by considering all possible ways to synthesize the design principles. In this case, we find that having an arbitrary number of runs tuned specifically for every level of the tree gives the flexibility to match the workload exactly.

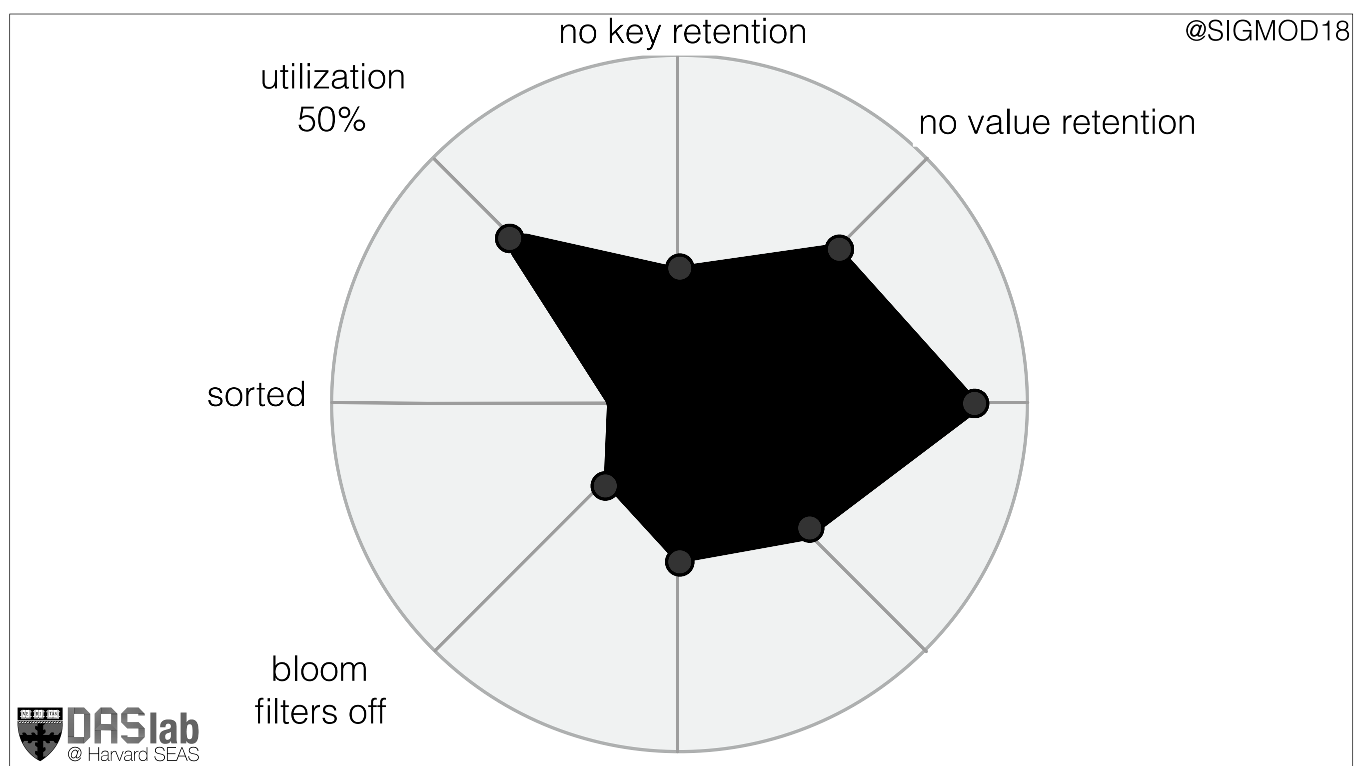
We keep doing this process of decomposing to first principles across different classes of data structures. This generates a series of design sub-spaces for every class.



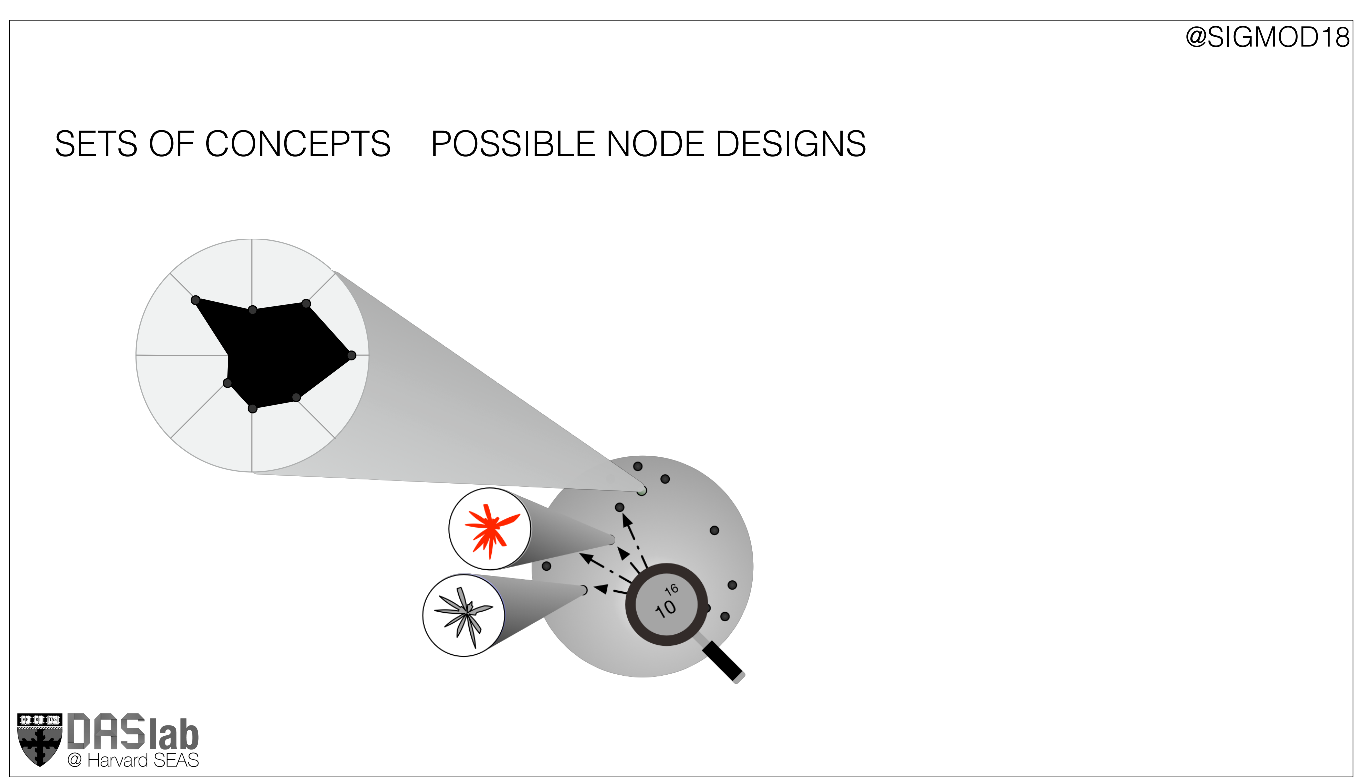

We keep doing this process of decomposing to first principles across different classes of data structures. This generates a series of design sub-spaces for every class.



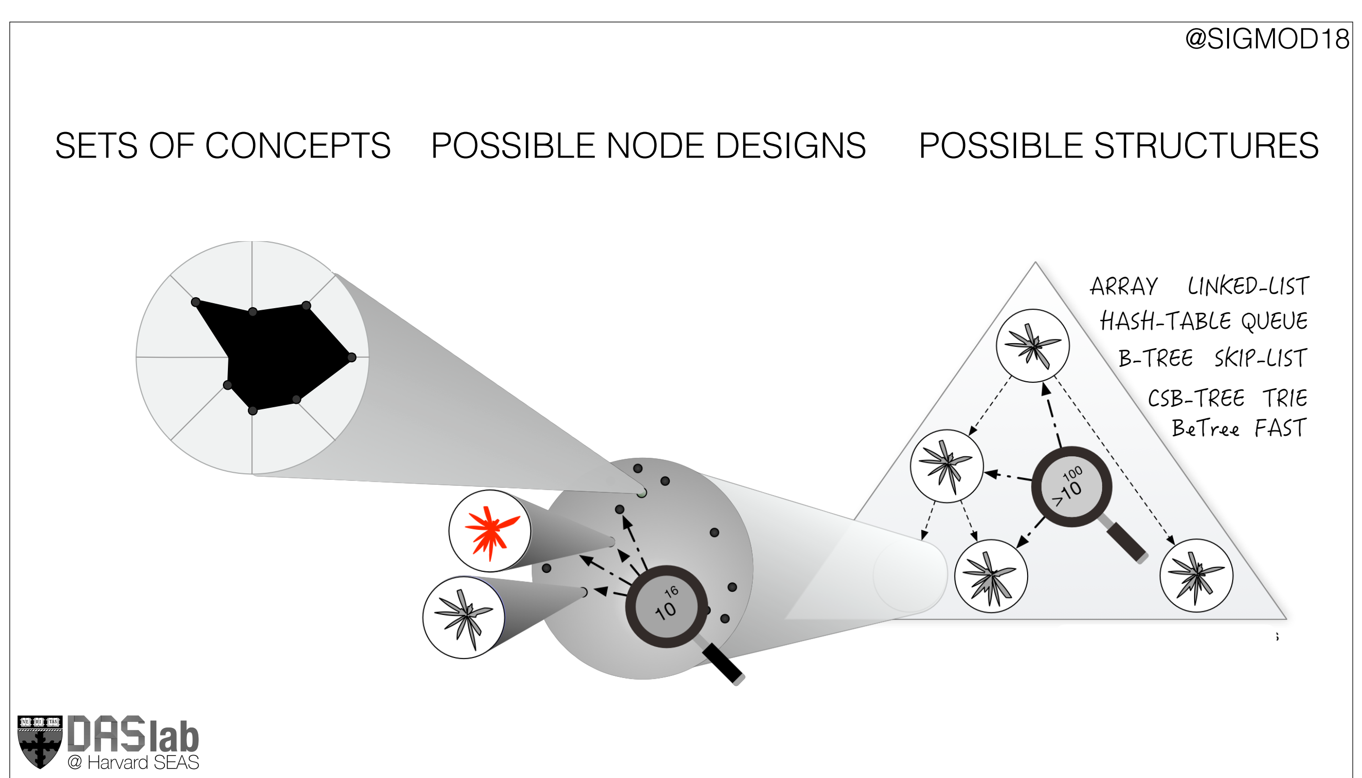

And we bring everything together to a common design space that describes all key-value data structures that we have invented in CS and many more that have not been invented yet.



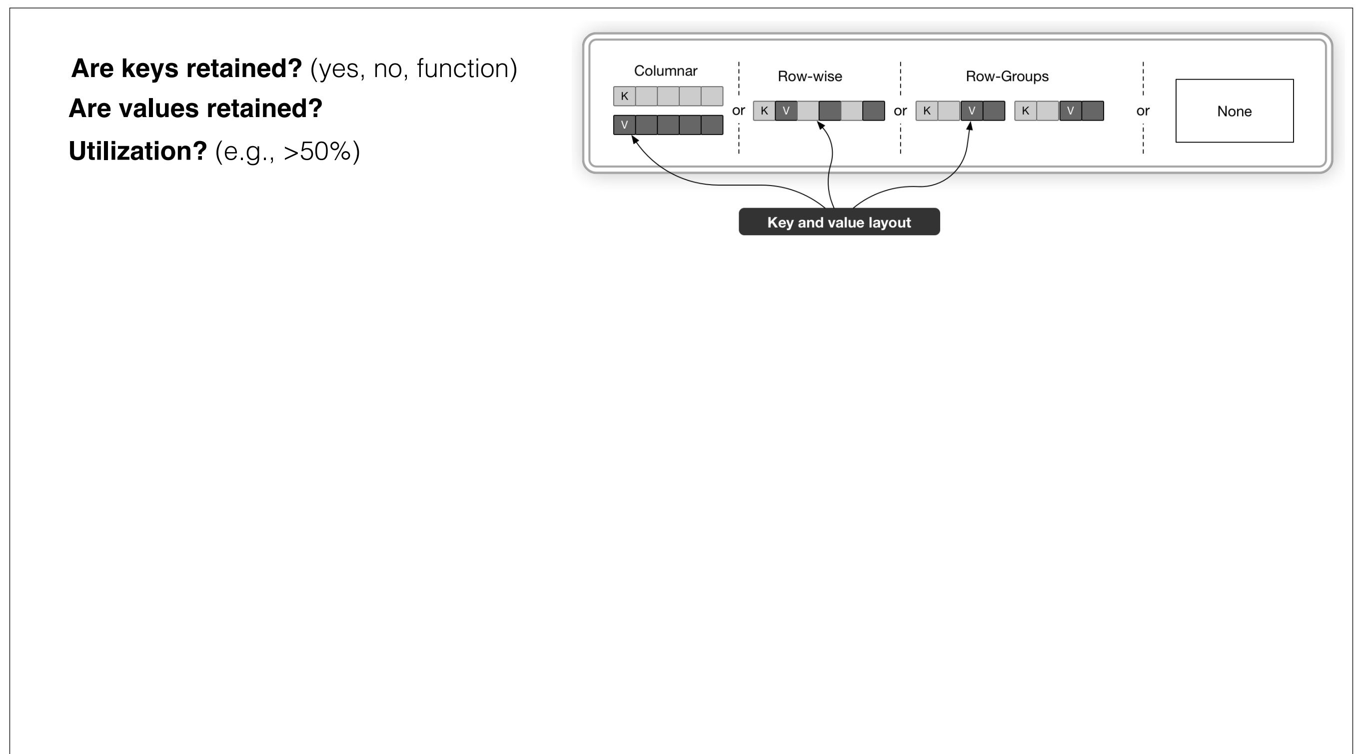

And we bring everything together to a common design space that describes all key-value data structures that we have invented in CS and many more that have not been invented yet.



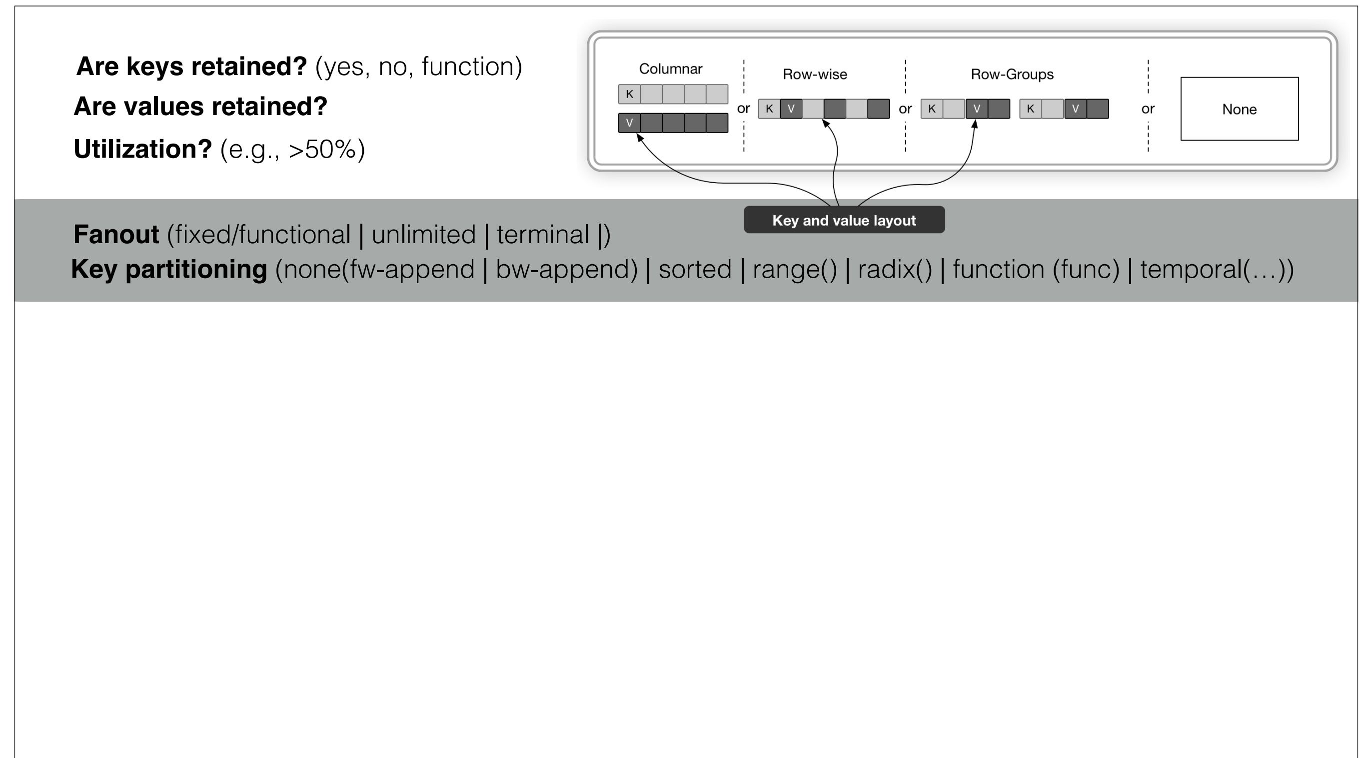

This design space works like a grammar. It allows us to describe nodes of data structures. It gives us 10 to the 16 possible node designs. And then we can use one or more types of nodes, and connect them to create complex data structures. For example, most designs we know in the literature contain two types of nodes. Usually the index node and the data node. E.g., the internal and the leaf nodes of a B-tree, or the hash map and the buckets of a hash-table. Very few data structures with three nodes have been described. Notable exceptions are the Bounded Disorder paper which combines B-tree and Hash-table elements, as well as the MassTree paper which combines B-tree and Trie elements.



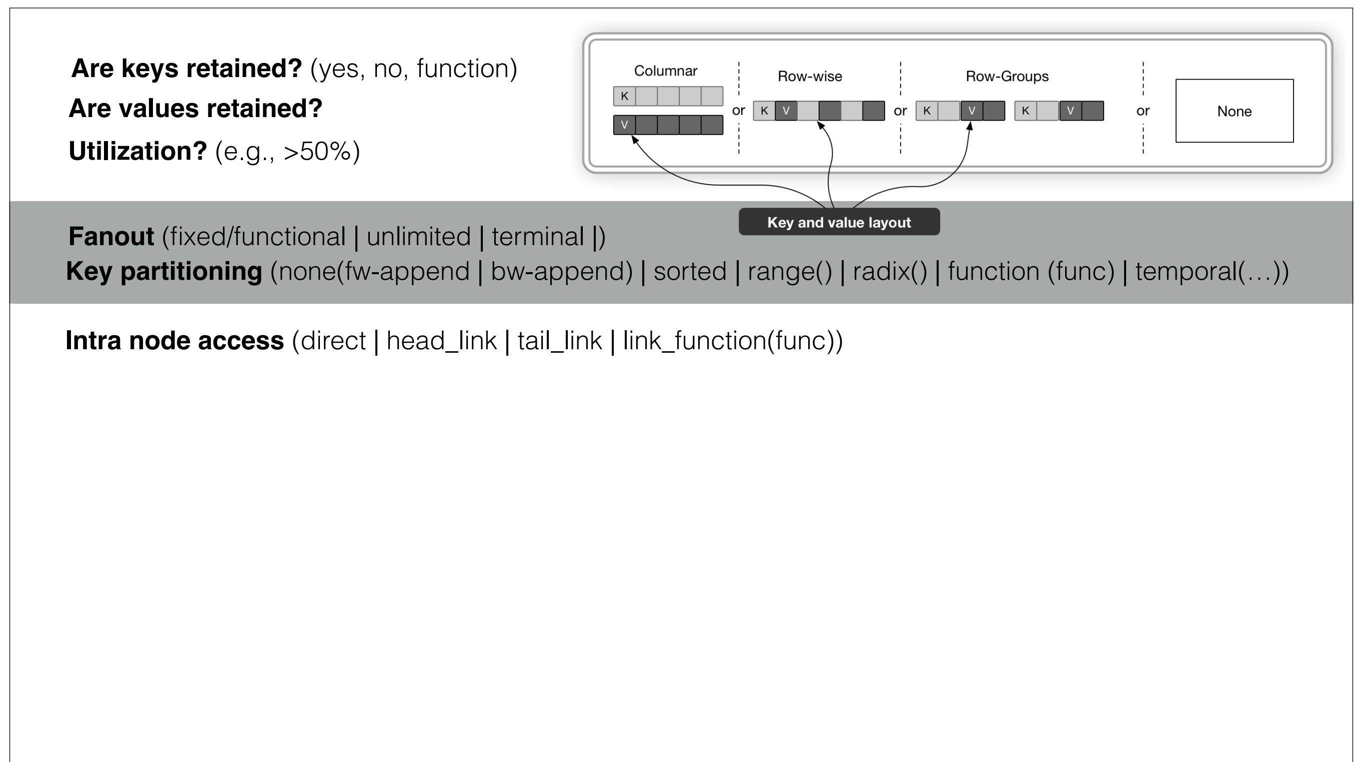

This design space works like a grammar. It allows us to describe nodes of data structures. It gives us 10 to the 16 possible node designs. And then we can use one or more types of nodes, and connect them to create complex data structures. For example, most designs we know in the literature contain two types of nodes. Usually the index node and the data node. E.g., the internal and the leaf nodes of a B-tree, or the hash map and the buckets of a hash-table. Very few data structures with three nodes have been described. Notable exceptions are the Bounded Disorder paper which combines B-tree and Hash-table elements, as well as the MassTree paper which combines B-tree and Trie elements.



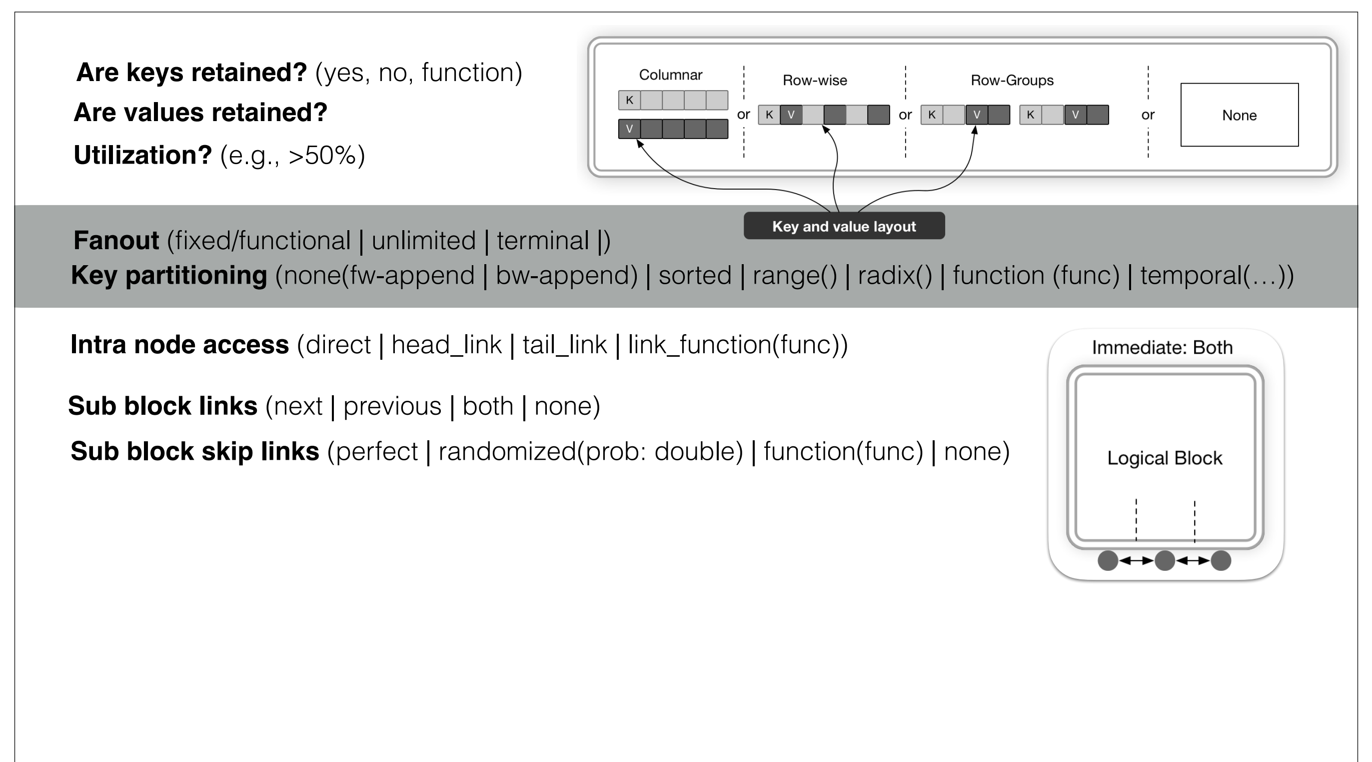

In more detail some of the design principles contained in the design space to describe data structure nodes.



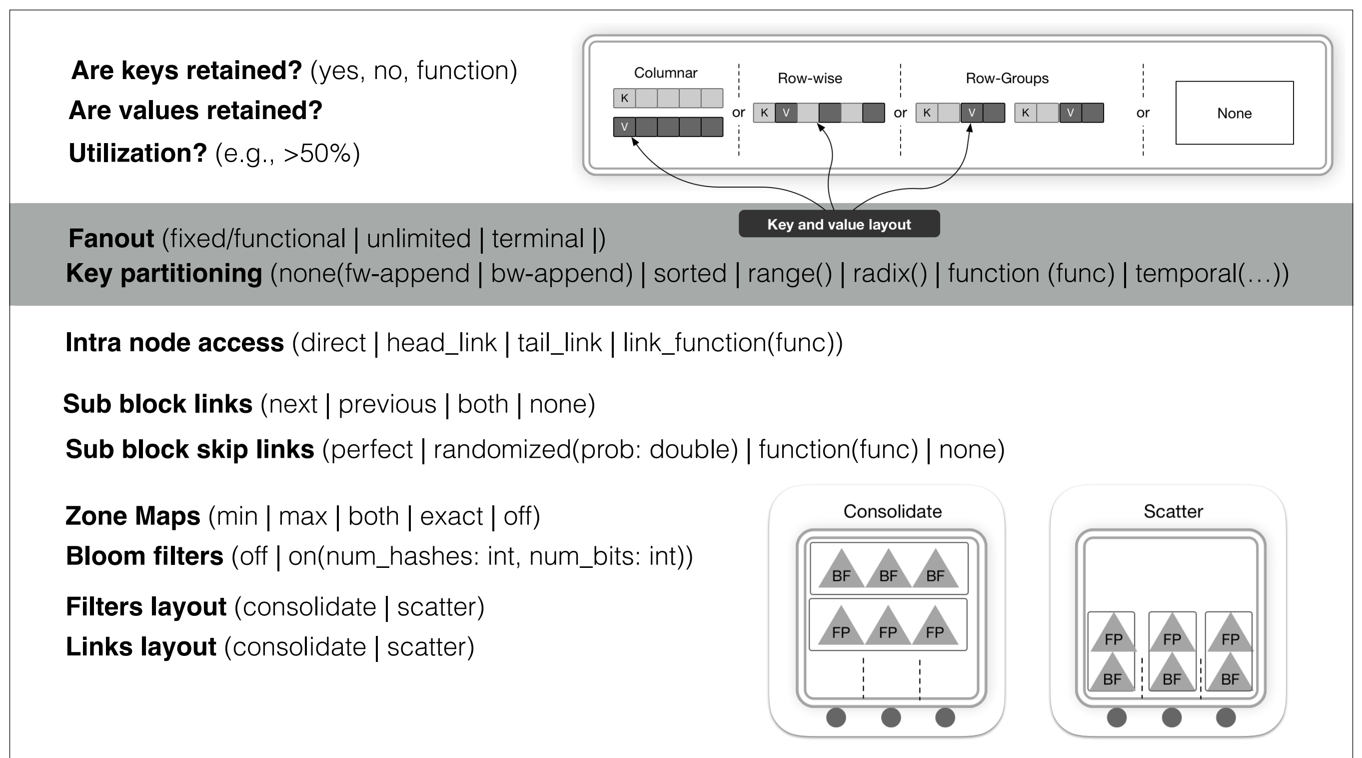

In more detail some of the design principles contained in the design space to describe data structure nodes.



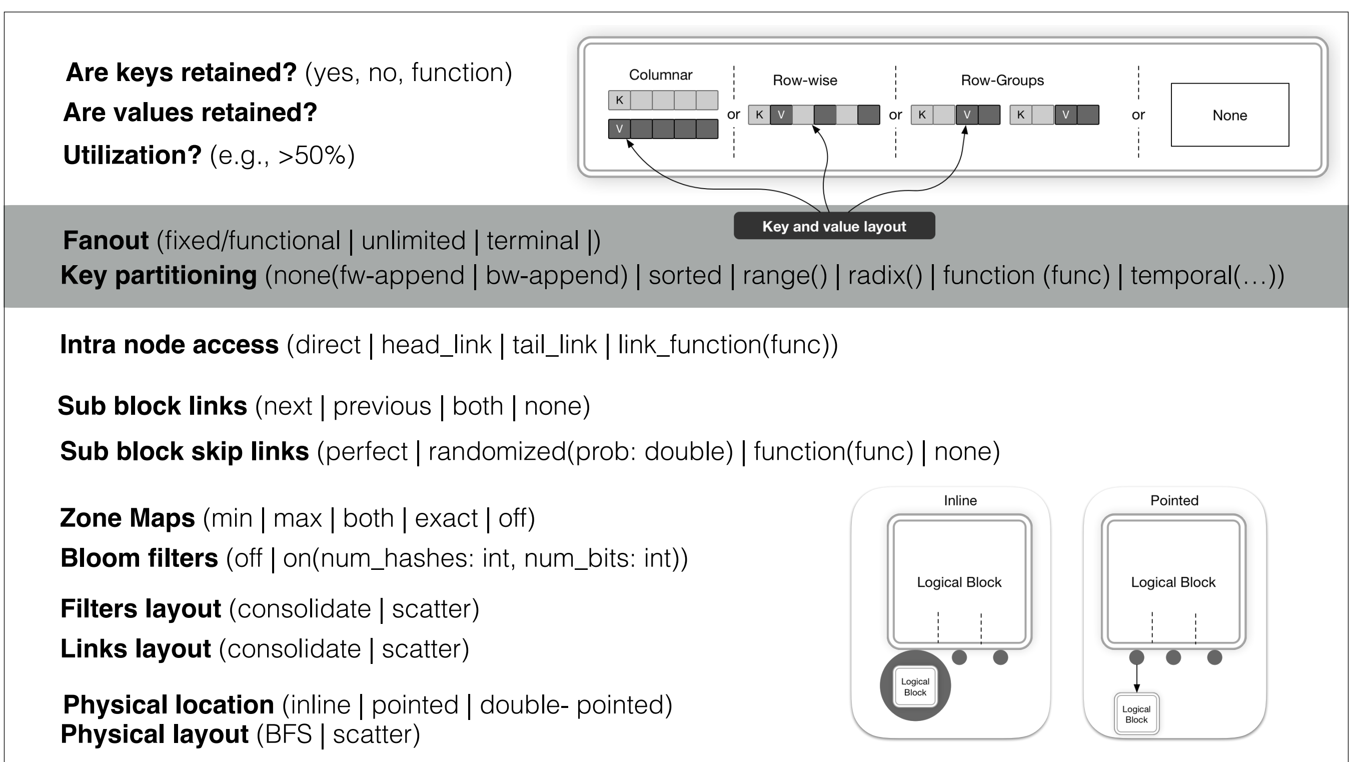

In more detail some of the design principles contained in the design space to describe data structure nodes.




In more detail some of the design principles contained in the design space to describe data structure nodes.




In more detail some of the design principles contained in the design space to describe data structure nodes.




In more detail some of the design principles contained in the design space to describe data structure nodes.

# Primitives and Instances

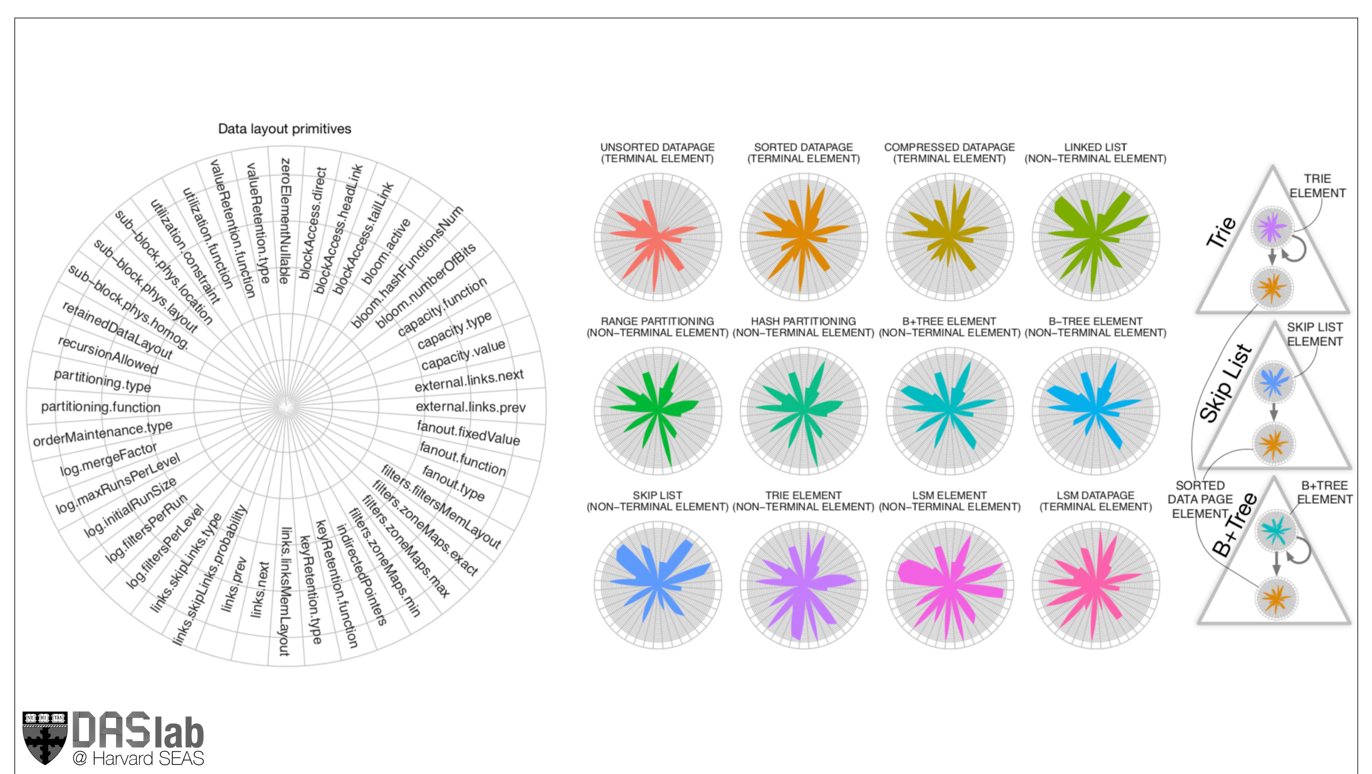
*Unless otherwise specified, we use a reduced default values domain of 100 values for integers, 10 values for doubles, and 1 value for functions.*

| Categories | | # | Primitive | Domain | size | Hash Table: H | Hash Table (LPL): LL | Hash Table (LPL): UDP | B+Tree/CSB+Tree/FAST: B+ | B+Tree/CSB+Tree/FAST: CSB+ | B+Tree/CSB+Tree/FAST: FAST | B+Tree/CSB+Tree/FAST: ODP |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Node organization | 1 | **Key retention.** *No:* node contains no real key data, e.g., intermediate nodes of b+trees and linked lists. *Yes:* contains complete key data, e.g., nodes of b-trees, and arrays. *Function:* contains only a subset of the key, i.e., as in tries. | yes \| no \| function(func) | 3 | no | no | yes | no | no | no | yes |
| | Node organization | 2 | **Value retention.** *No:* node contains no real value data, e.g., intermediate nodes of b+trees, and linked lists. *Yes:* contains complete value data, e.g., nodes of b-trees, and arrays. *Function:* contains only a subset of the values. | yes \| no \| function(func) | 3 | no | no | yes | no | no | no | yes |
| | Node organization | 3 | **Key order.** Determines the order of keys in a node or the order of fences if real keys are not retained. | none \| sorted \| k-ary (k: int) | 12 | none | none | none | sorted | sorted | 4-ary | sorted |
| | Node organization | 4 | **Key-value layout.** Determines the physical layout of key-value pairs. Rules: requires key retention != no or value retention != no. | row-wise \| columnar \| col-row-groups(size: int) | 12 | | | col. | | | | col. |
| | Node organization | 5 | **Intra-node access.** Determines how sub-blocks (one or more keys of this node) can be addressed and retrieved within a node, e.g., with direct links, a link only to the first or last block, etc. | direct \| head_link \| tail_link \| link_function(func) | 4 | direct | head | direct | direct | direct | direct | direct |
| | Node organization | 6 | **Utilization.** Utilization constraints in regards to capacity. For example, >= 50% denotes that utilization has to be greater than or equal to half the capacity. | = (X%) \| function(func) \| none *(we currently only consider X=50)* | 3 | none | none | none | >= 50% | >= 50% | >= 50% | none |
| | Node filters | 7 | **Bloom filters.** A node's sub-block can be filtered using bloom filters. Bloom filters get as parameters the number of hash functions and number of bits. | off \| on(num_hashes: int, num_bits: int) *(up to 10 num_hashes considered)* | 1001 | off | off | off | off | off | off | off |
| | Node filters | 8 | **Zone map filters.** A node's sub-block can be filitered using zone maps, e.g., they can filter based on mix/max keys in each sub-block. | min \| max \| both \| exact \| off | 5 | off | off | off | min | min | min | off |
| | Node filters | 9 | **Filters memory layout.** Filters are stored contiguously in a single area of the node or scattered across the sub-blocks. Rules: requires bloom filter != off or zone map filters != off. | consolidate \| scatter | 2 | | | | scatter | scatter | scatter | |
| | | 10 | **Fanout/Radix.** Fanout of current node in terms of sub-blocks. This can either be unlimited (i.e., no restriction on the number of sub-blocks), fixed to a number, decided by a function or the node is terminal and thus has a fixed capacity. | fixed(value: int) \| function(func) \| [un]limited \| terminal(cap: int) *(up to 10 different capacities and up to 10 fixed fanout values are considered)* | 22 | fixed(100) | unlimited | term(256) | fixed(20) | fixed(20) | fixed(16) | term(256) |

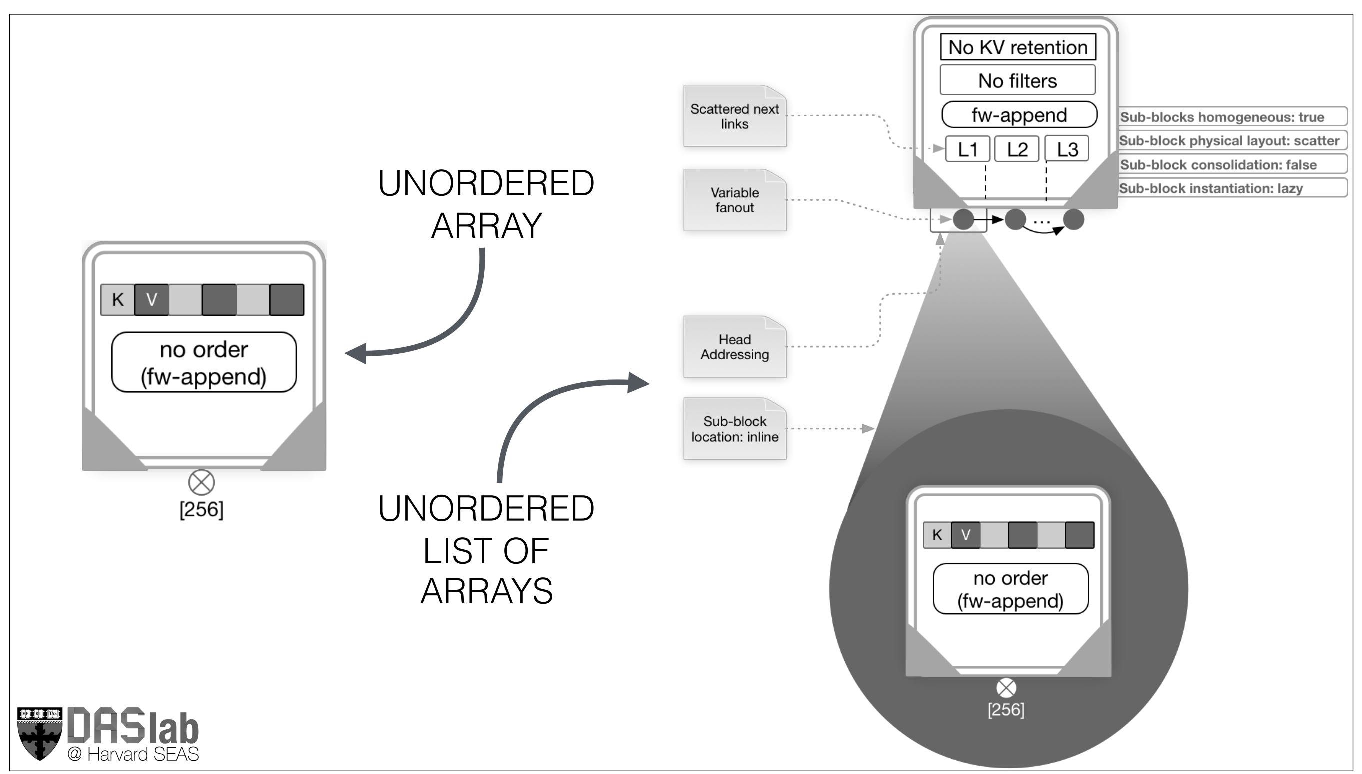
In more detail some of the design principles contained in the design space to describe data structure nodes.

| Categories | | # | Description | Domain | Size | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Partiti… | | or balanced (i.e., all sub-blocks have the same size), unrestricted or functional.<br>Rules: requires key partitioning != none. | …stricted \| function(func)<br>*(up to 10 different fixed capacity values are considered)* | 13 | unrestri… | fixed(25… | | balance… | balance… | balance… | |
| | | 13 | **Immediate node links.** Whether and how sub-blocks are connected. | next \| previous \| both \| none | 4 | none | next | none | none | none | none | none |
| | | 14 | **Skip node links.** Each sub-block can be connected to another sub-block (not only the next or previous) with skip-links. They can be perfect, randomized or custom. | perfect \| randomized(prob: double) \| function(func) \| none | 13 | none | none | none | none | none | none | none |
| | | 15 | **Area-links.** Each sub-tree can be connected with another sub-tree at the leaf level throu area links. Examples include the linked leaves of a B+Tree. | forward \| backward \| both \| none | 4 | none | none | forw. | none | none | none | none |
| | Children layout | 16 | **Sub-block physical location.** This represents the physical location of the sub-blocks. Pointed: in heap, Inline: block physically contained in parent. Double-pointed: in heap but with pointers back to the parent.<br>Rules: requires fanout/radix != terminal. | inline \| pointed \| double-pointed | 3 | pointed | inline | | pointed | pointed | pointed | |
| | | 17 | **Sub-block physical layout.** This represents the physical layout of sub-blocks. Scatter: random placement in memory. BFS: laid out in a breadth-first layout. BFS layer list: hierarchical level nesting of BFS layouts.<br>Rules: requires fanout/radix != terminal. | BFS \| BFS layer(level-grouping: int) \| scatter<br>*(up to 3 different values for layer-grouping are considered)* | 5 | scatter | scatter | | scatter | BFS | BFS-LL | |
| | | 18 | **Sub-blocks homogeneous.** Set to true if all sub-blocks are of the same type.<br>Rules: requires fanout/radix != terminal. | boolean | 2 | true | true | | true | true | true | |
| | | 19 | **Sub-block consolidation.** Single children are merged with their parents.<br>Rules: requires fanout/radix != terminal. | boolean | 2 | false | false | | false | false | false | |
| | | 20 | **Sub-block instantiation.** If it is set to eager, all sub-blocks are initialized, otherwise they are initialized only when data are available (lazy).<br>Rules: requires fanout/radix != terminal. | lazy \| eager | 2 | lazy | lazy | | lazy | lazy | lazy | |
| | | 21 | **Sub-block links layout.** If there exist links, are they all stored in a single array (consolidate) or spread at a per partition level (scatter).<br>Rules: requires immediate node links != none or skip links != none. | consolidate \| scatter | 2 | | scatter | | | | | |
| | Recursion | 22 | **Recursion allowed.** If set to yes, sub-blocks will be subsequently inserted into a node of the same type until a maximum depth (expressed as a function) is reached. Then the terminal node type of this data structure will be used.<br>Rules: requires fanout/radix != terminal. | yes(func) \| no | 3 | no | no | | yes(logn) | yes(logn) | yes(logn) | |

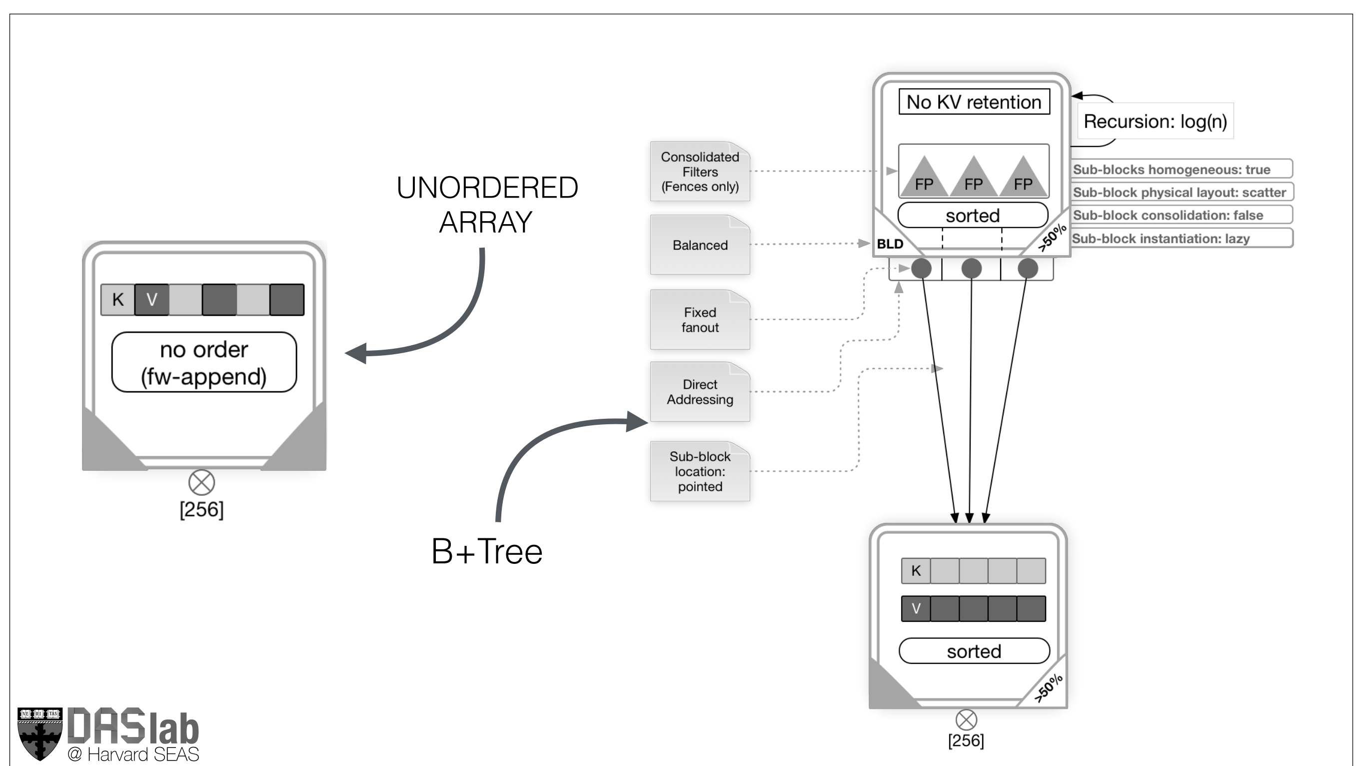
In more detail some of the design principles contained in the design space to describe data structure nodes.




We use radar visuals to describe node types.




Examples of known designs and how they can be synthesized from the design space. For example a plain array can be described as a single node design where keys and values are stored in sequence with no particular order. On top of that, we can create a linked list of arrays by adding one more node type which contains no data but serves the purpose of linking the underlying nodes. Note that the grammar does not dictate that every node type is actually materialized to a physical data structure node. In the case of linked list of arrays the top level node is a virtual node that simply dictates the connecting structure of the underlying arrays.




Similar example like before but now with B+tree.

@IEEE.EngBul18

# periodic table of data structures

| classes of primitives \ classes of designs | B-trees & Variants | Tries & Variants | LSM-Trees & Variants | Differential Files | Membership Tests | Zone maps & Variants | Bitmaps & Variants | Hashing | Base Data & Columns | RUM |
|---|---|---|---|---|---|---|---|---|---|---|
| Partitioning | DONE | DONE | DONE | | | | | DONE | DONE | ↓↑↑ |
| Logarithmic Design | DONE | DONE | DONE | | | | | | | ↓↓↑ |
| Fractional Cascading | DONE | | DONE | DONE | | | | | | ↓↑↑ |
| Log-Structured | DONE | | DONE | DONE | | | | | | ↑↓↑ |
| Buffering | DONE | | | DONE | | | DONE | | | ↓↕↑ |
| Differential Updates | DONE | | | DONE | | | | | | ↑↓↓ |
| Sparse Indexing | DONE | | | | DONE | DONE | | | | ↓↕↑ |
| Adaptivity | DONE | | | | | | | | DONE | |

DASlab @ Harvard SEAS

We have put everything together to create the periodic table of data structures. It is a zoomed out version of the design space. It classifies existing classes of data structures with respect to the design principles that have been applied. When a cell is marked as “done” this means that there is at least one research paper or system that we know of that has applied this design principles to this class of data structures. We also give guidance in terms of the RUM tradeoffs and what is expected to happen. The primary observation is that most of the design space is actually still unexplored! Like the periodic table in chemistry, the periodic table of data structures can be used to accelerate research by using the gaps as a guide.

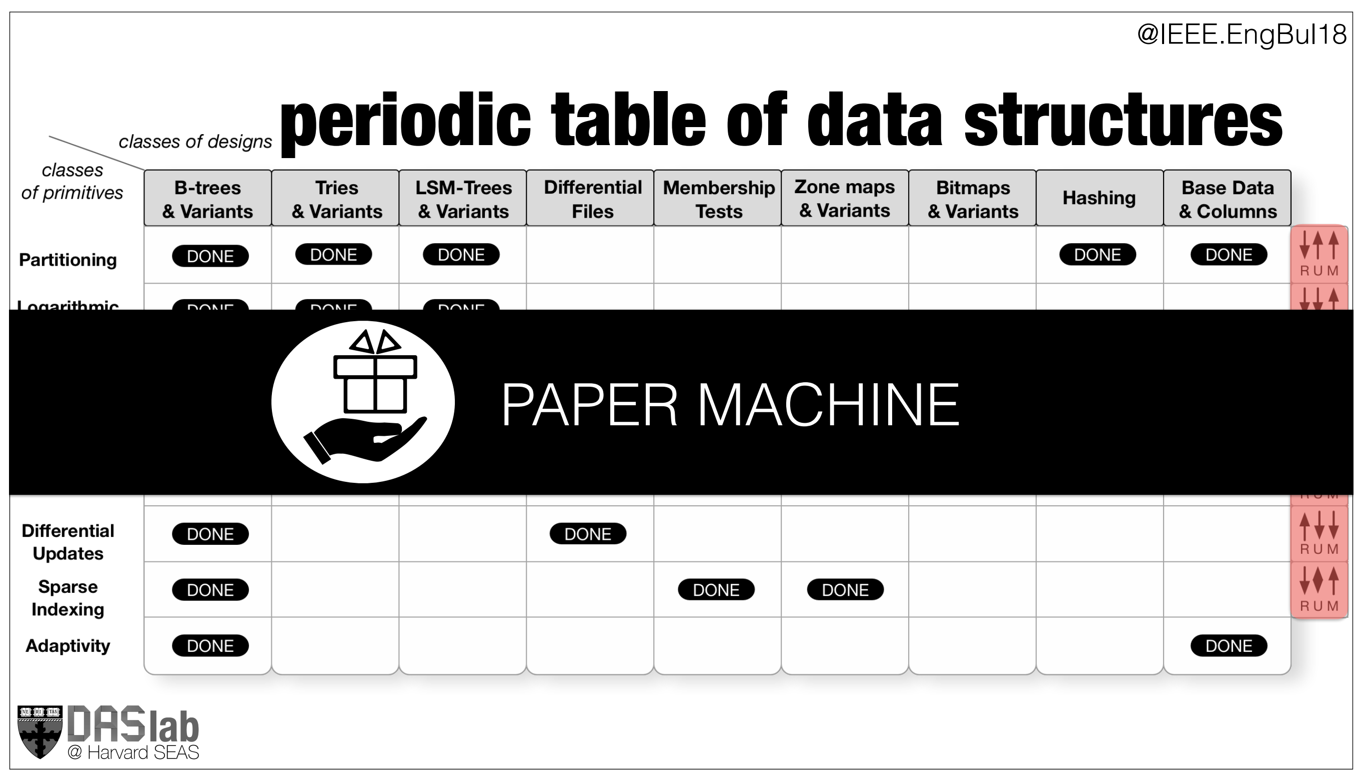
@IEEE.EngBul18

# periodic table of data structures

| classes of primitives \ classes of designs | B-trees & Variants | Tries & Variants | LSM-Trees & Variants | Differential Files | Membership Tests | Zone maps & Variants | Bitmaps & Variants | Hashing | Base Data & Columns | |
|---|---|---|---|---|---|---|---|---|---|---|
| Partitioning | DONE | DONE | DONE | | | | | DONE | DONE | R U M |
| Logarithmic Design | DONE | DONE | DONE | | | | | | | R U M |
| Fractional Cascading | DONE | | DONE | DONE | | | | | | R U M |
| Log-Structured | DONE | | DONE | DONE | | | | | | R U M |
| Buffering | DONE | | | DONE | | | DONE | | | R U M |
| Differential Updates | DONE | | | DONE | | | | | | R U M |
| Sparse Indexing | DONE | | | | DONE | DONE | | | | R U M |
| Adaptivity | DONE | | | | | | | | DONE | |

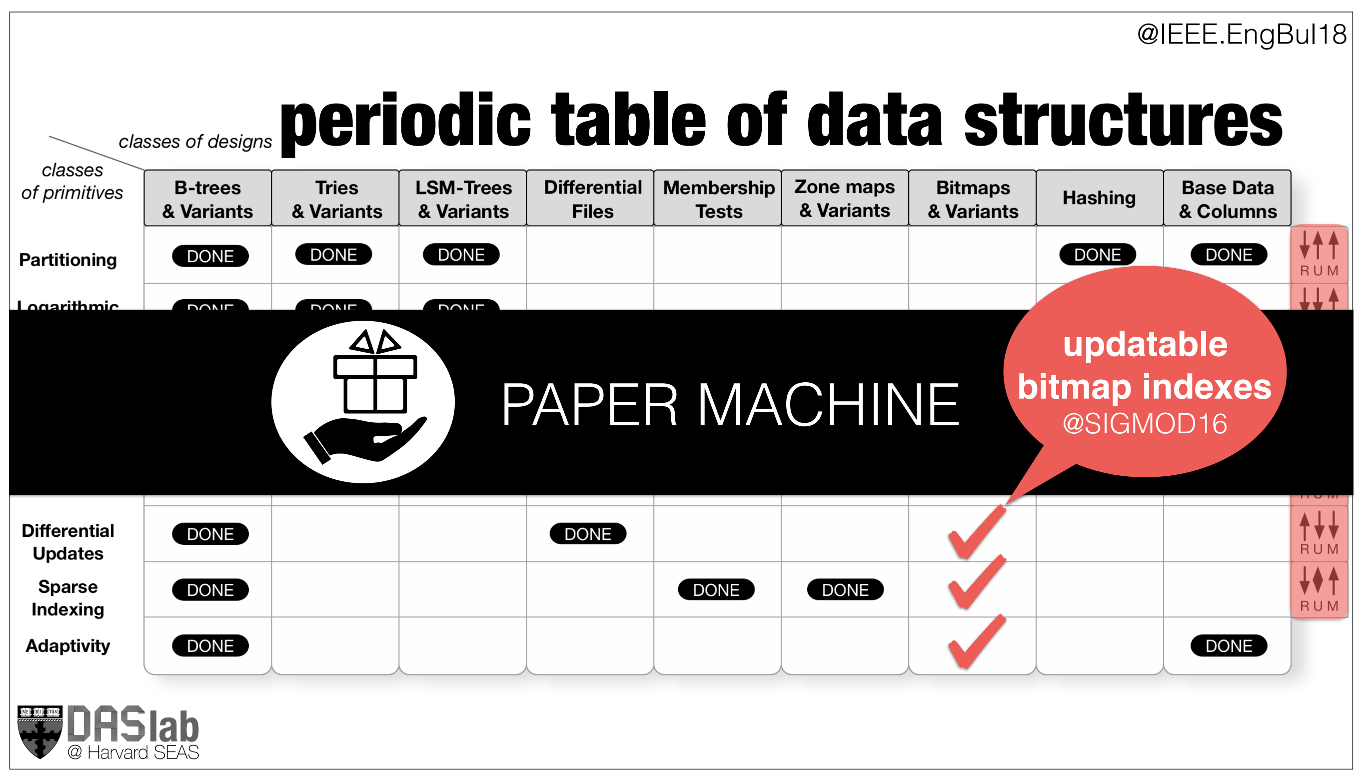
DASlab @ Harvard SEAS

We have put everything together to create the periodic table of data structures. It is a zoomed out version of the design space. It classifies existing classes of data structures with respect to the design principles that have been applied. When a cell is marked as “done” this means that there is at least one research paper or system that we know of that has applied this design principles to this class of data structures. We also give guidance in terms of the RUM tradeoffs and what is expected to happen. The primary observation is that most of the design space is actually still unexplored! Like the periodic table in chemistry, the periodic table of data structures can be used to accelerate research by using the gaps as a guide.

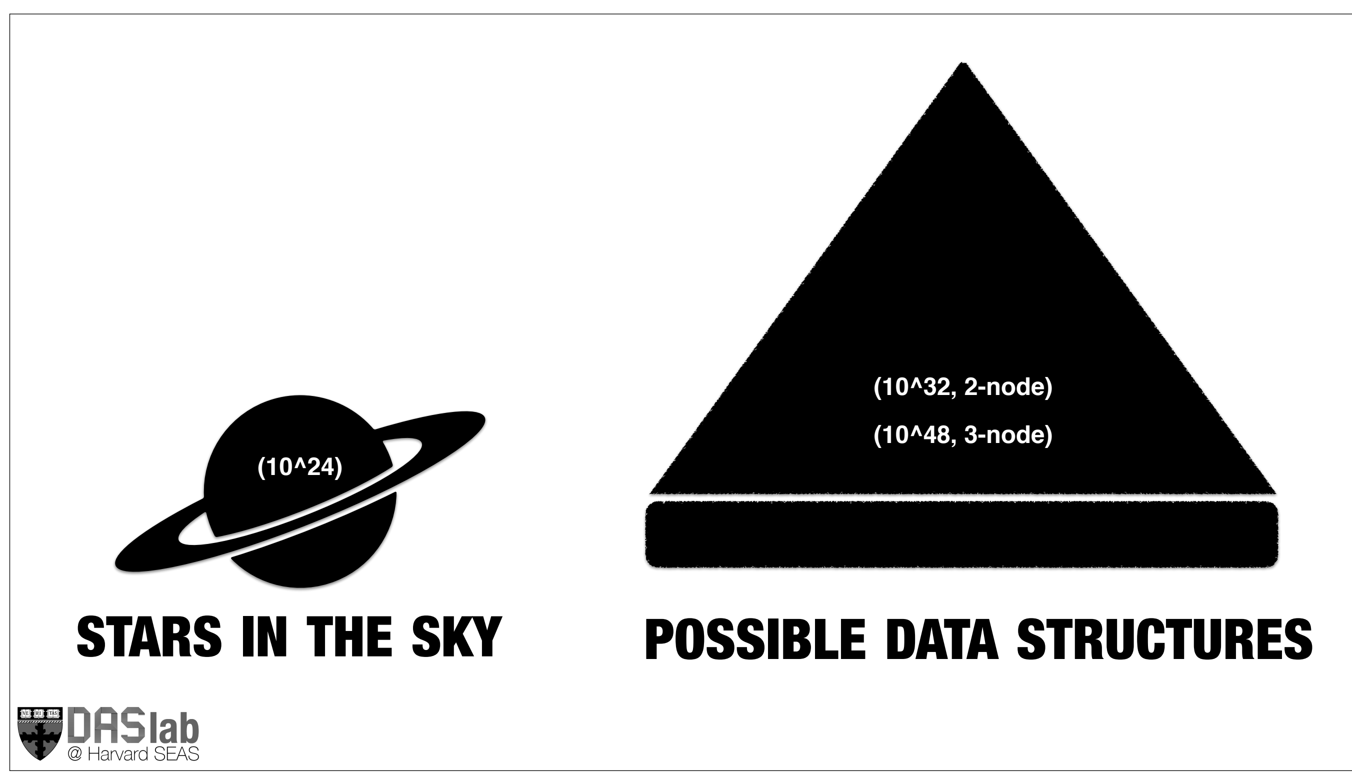



| classes of primitives | B-trees & Variants | Tries & Variants | LSM-Trees & Variants | Differential Files | Membership Tests | Zone maps & Variants | Bitmaps & Variants | Hashing | Base Data & Columns |
|---|---|---|---|---|---|---|---|---|---|
| Partitioning | DONE | DONE | DONE | | | | | DONE | DONE |
| Differential Updates | DONE | | | DONE | | | | | |
| Sparse Indexing | DONE | | | | DONE | DONE | | | |
| Adaptivity | DONE | | | | | | | | DONE |

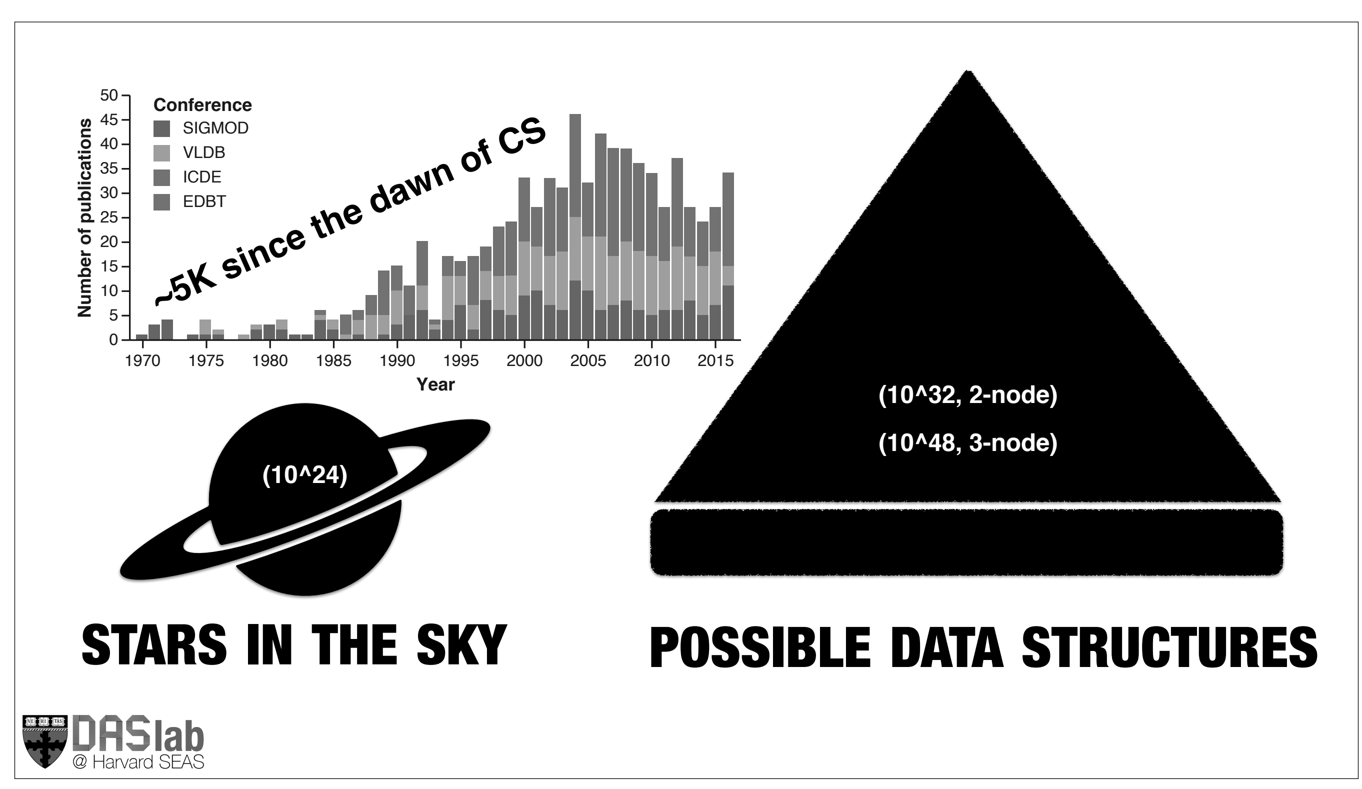
We have put everything together to create the periodic table of data structures. It is a zoomed out version of the design space. It classifies existing classes of data structures with respect to the design principles that have been applied. When a cell is marked as “done” this means that there is at least one research paper or system that we know of that has applied this design principles to this class of data structures. We also give guidance in terms of the RUM tradeoffs and what is expected to happen. The primary observation is that most of the design space is actually still unexplored! Like the periodic table in chemistry, the periodic table of data structures can be used to accelerate research by using the gaps as a guide.



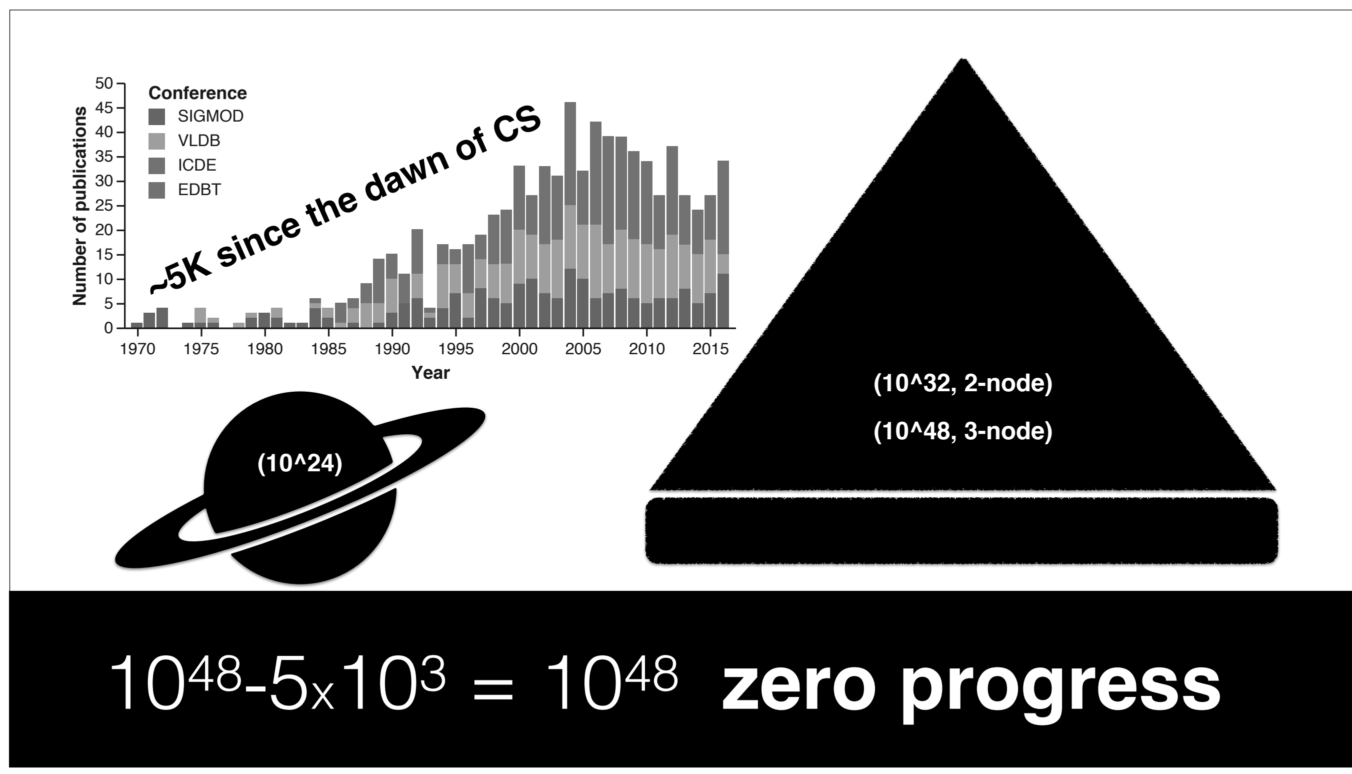

We have put everything together to create the periodic table of data structures. It is a zoomed out version of the design space. It classifies existing classes of data structures with respect to the design principles that have been applied. When a cell is marked as "done" this means that there is at least one research paper or system that we know of that has applied this design principles to this class of data structures. We also give guidance in terms of the RUM tradeoffs and what is expected to happen. The primary observation is that most of the design space is actually still unexplored! Like the periodic table in chemistry, the periodic table of data structures can be used to accelerate research by using the gaps as a guide.



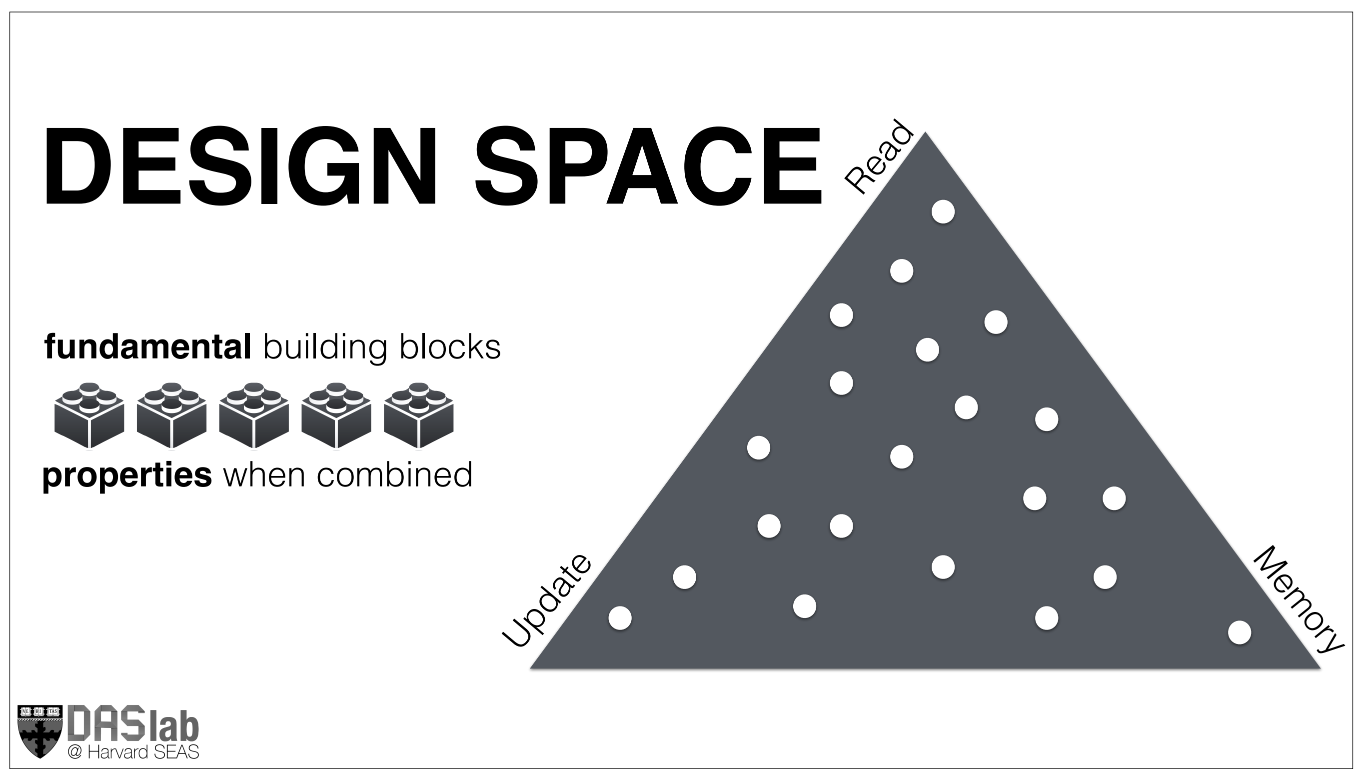

In fact we can measure the number of existing designs that have been manually invented the last 50 years using data from DBLP. It is barely around 5000 which is much much less than the possible valid designs described by the design space.



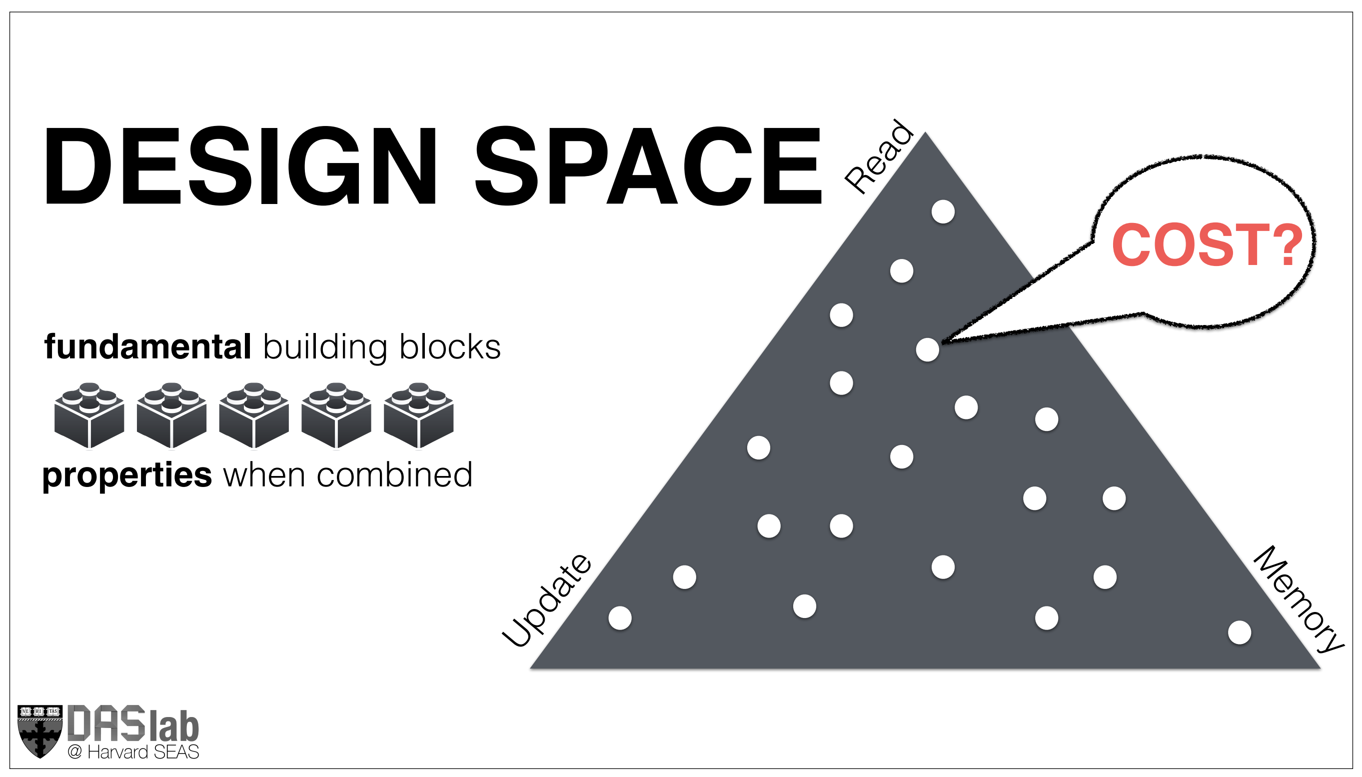

In fact we can measure the number of existing designs that have been manually invented the last 50 years using data from DBLP. It is barely around 5000 which is much much less than the possible valid designs described by the design space.



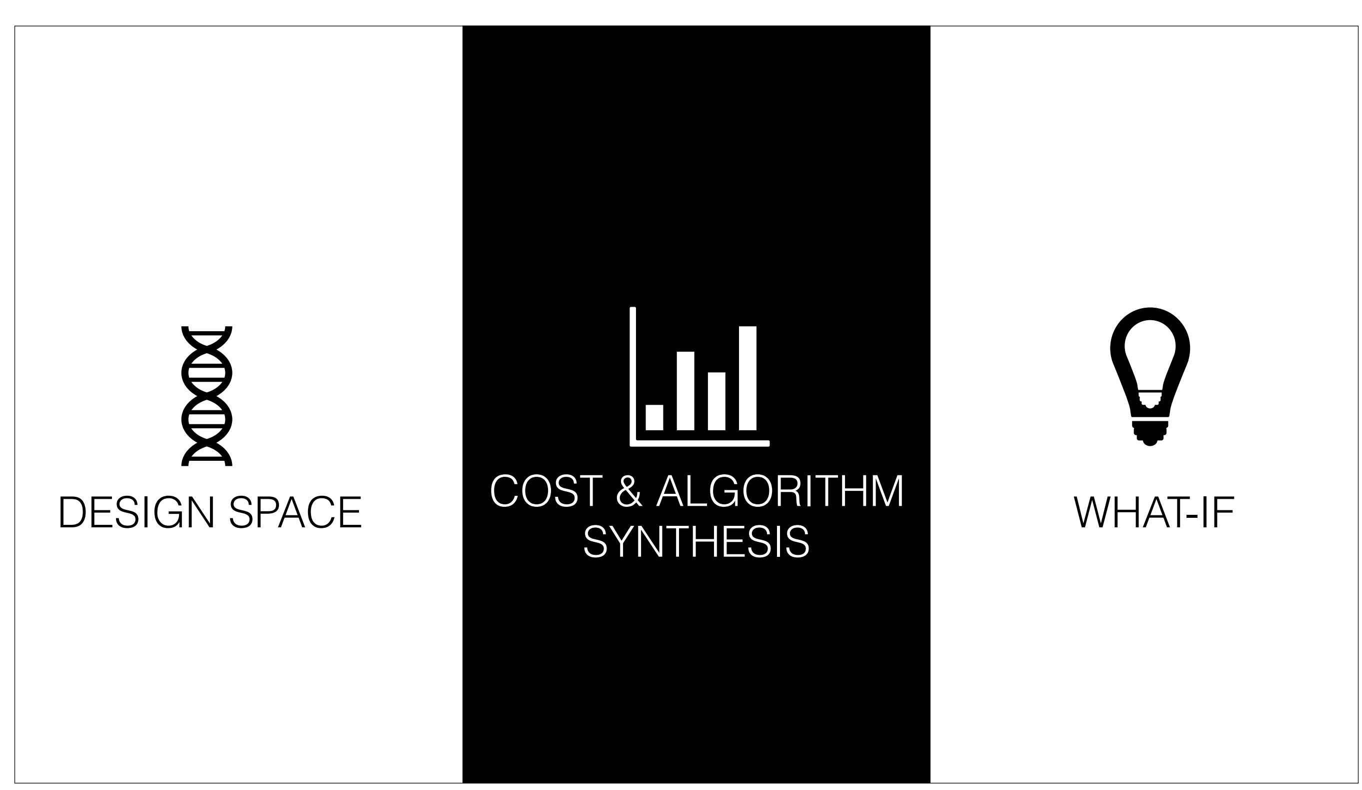

In fact we can measure the number of existing designs that have been manually invented the last 50 years using data from DBLP. It is barely around 5000 which is much much less than the possible valid designs described by the design space.

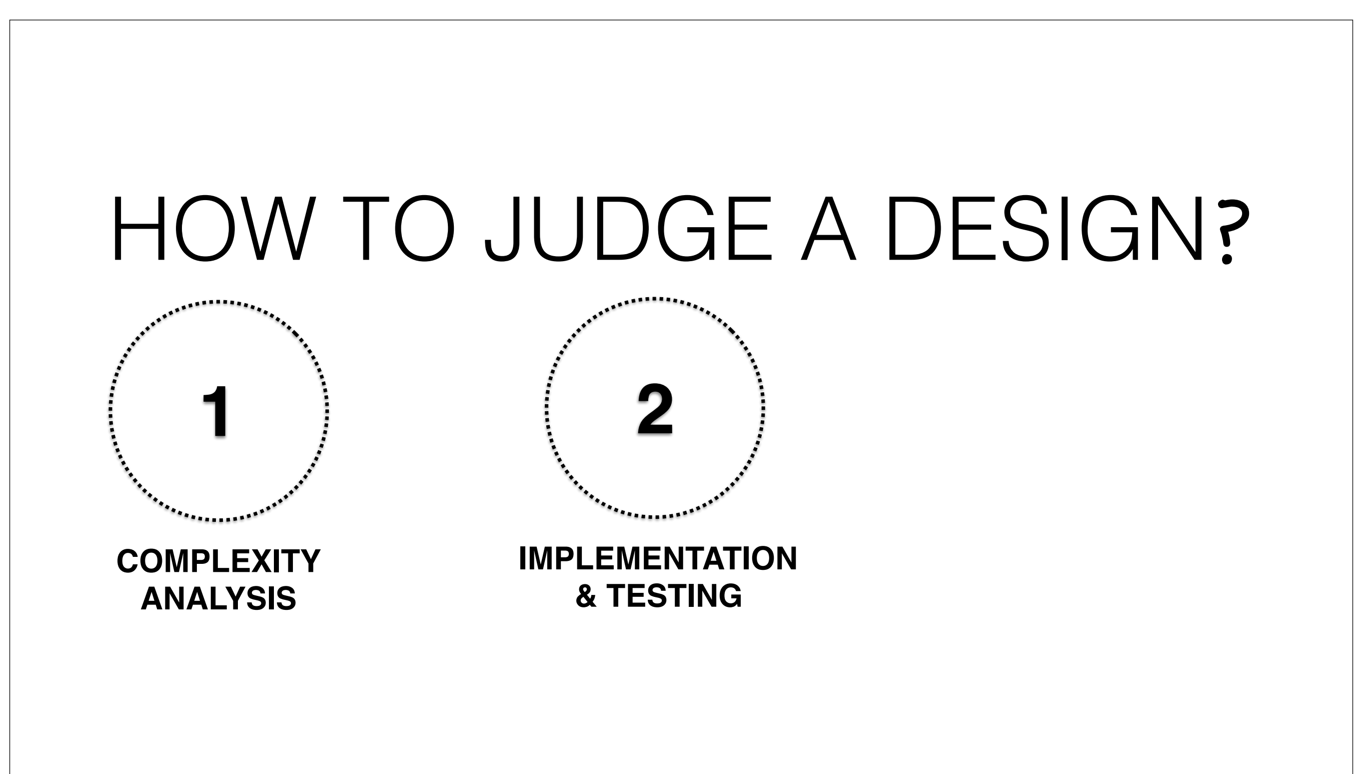



And this is why we need to answer the second question from the pair of questions we presented earlier about cost sythesis. The design space itself is not enough because it is massive. We need a way to navigate the space. And to be able to navigate and search the space the first fundamental component is to be able to cost every design, i.e., to be able to say how it would perform. If we can do that, then we can start comparing pairs of designs and we can start developing search algorithms.

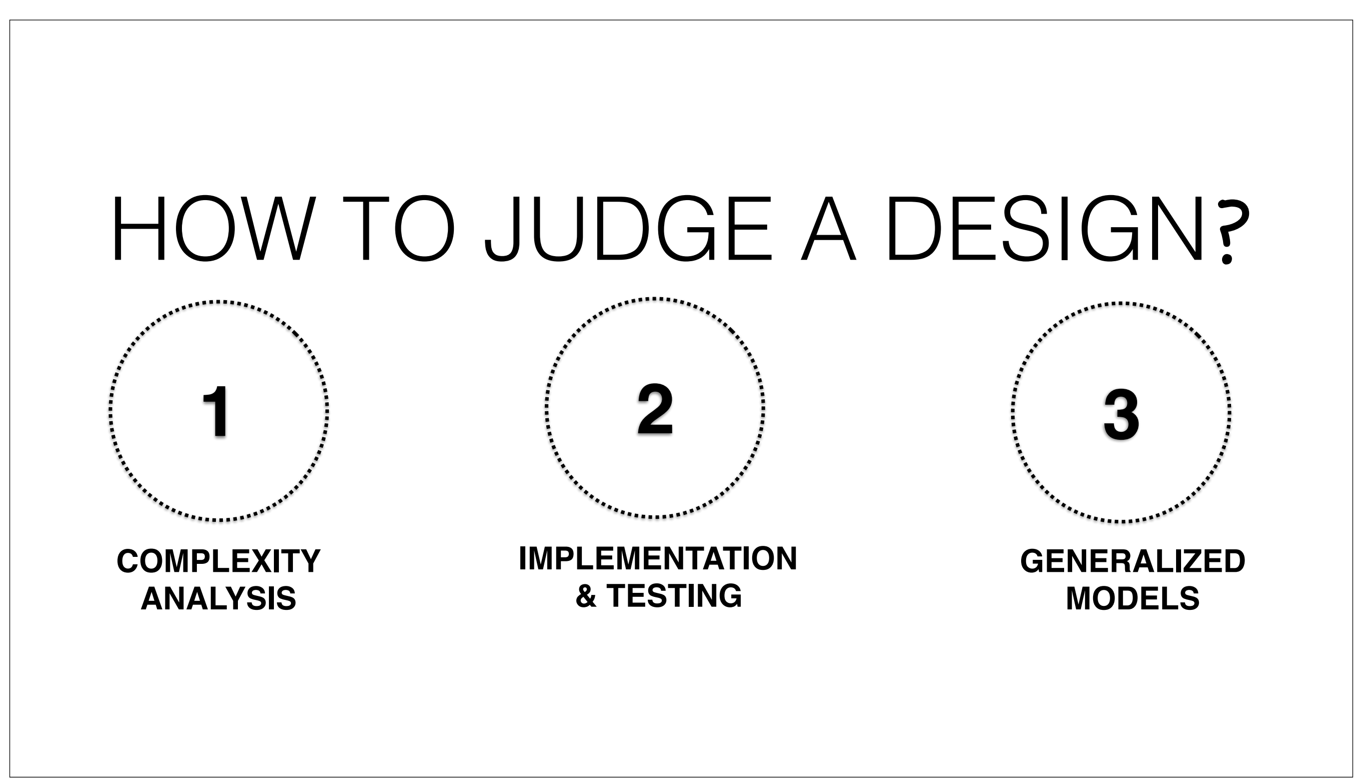



And this is why we need to answer the second question from the pair of questions we presented earlier about cost sythesis. The design space itself is not enough because it is massive. We need a way to navigate the space. And to be able to navigate and search the space the first fundamental component is to be able to cost every design, i.e., to be able to say how it would perform. If we can do that, then we can start comparing pairs of designs and we can start developing search algorithms.

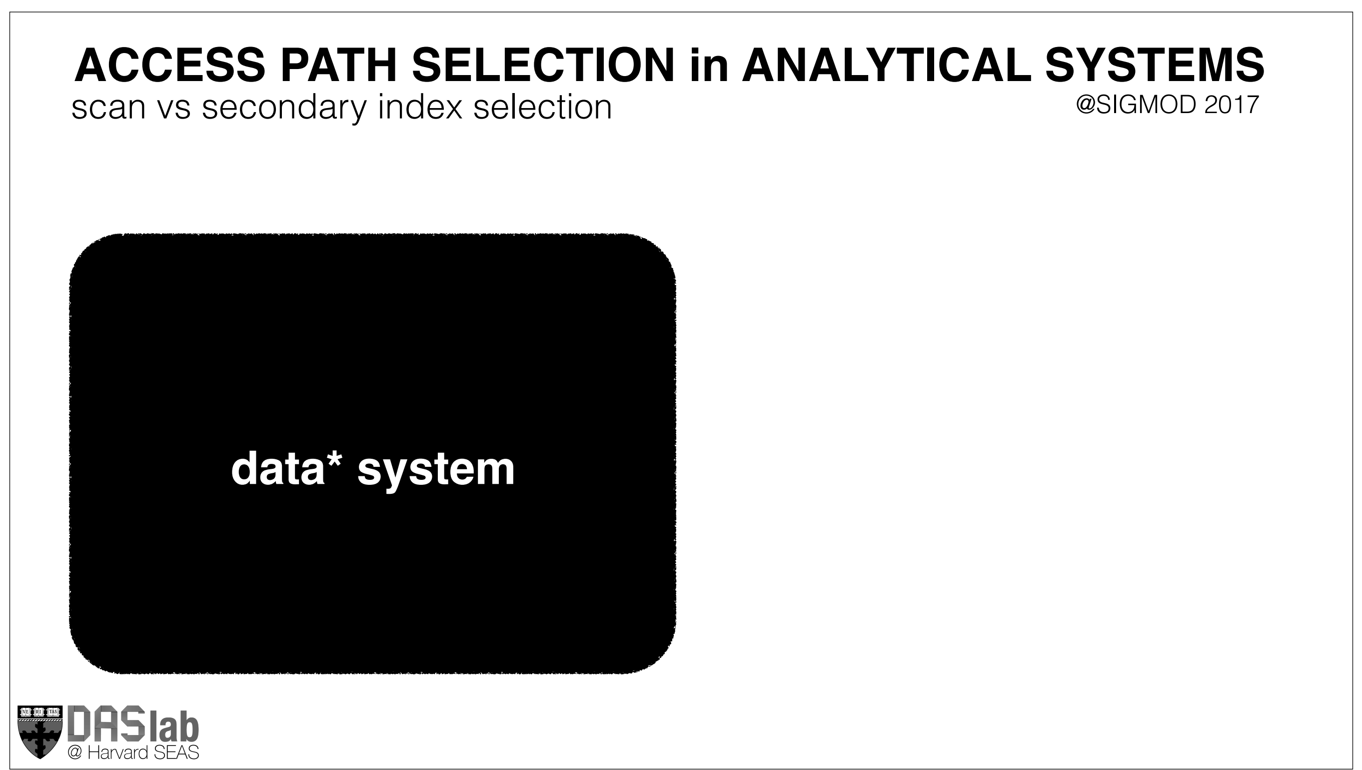

DESIGN SPACE
COST & ALGORITHM
SYNTHESIS
WHAT-IF



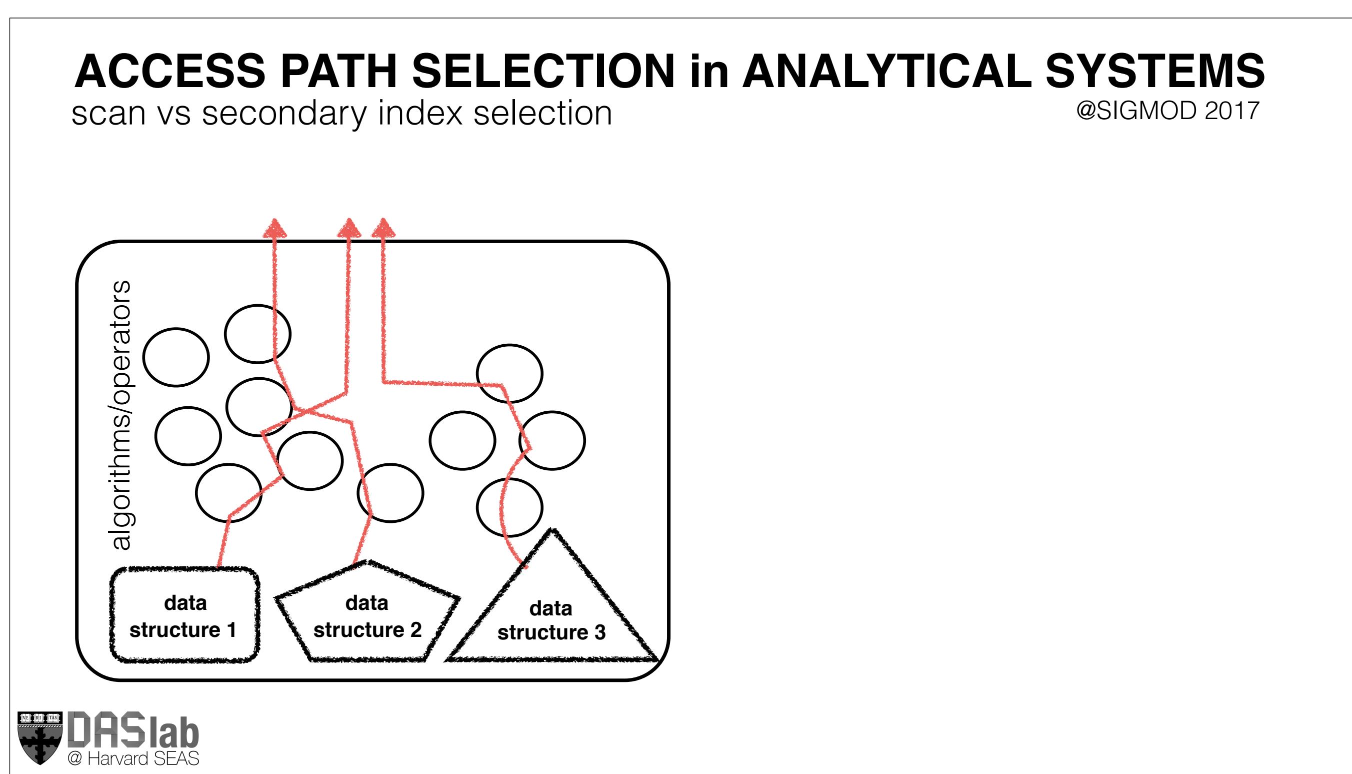

There are three primary ways to judge a data structure/algorithm design. Generalized models sounds like a very promising direction as it gives results close to what we get from the actual code, but it is a model.



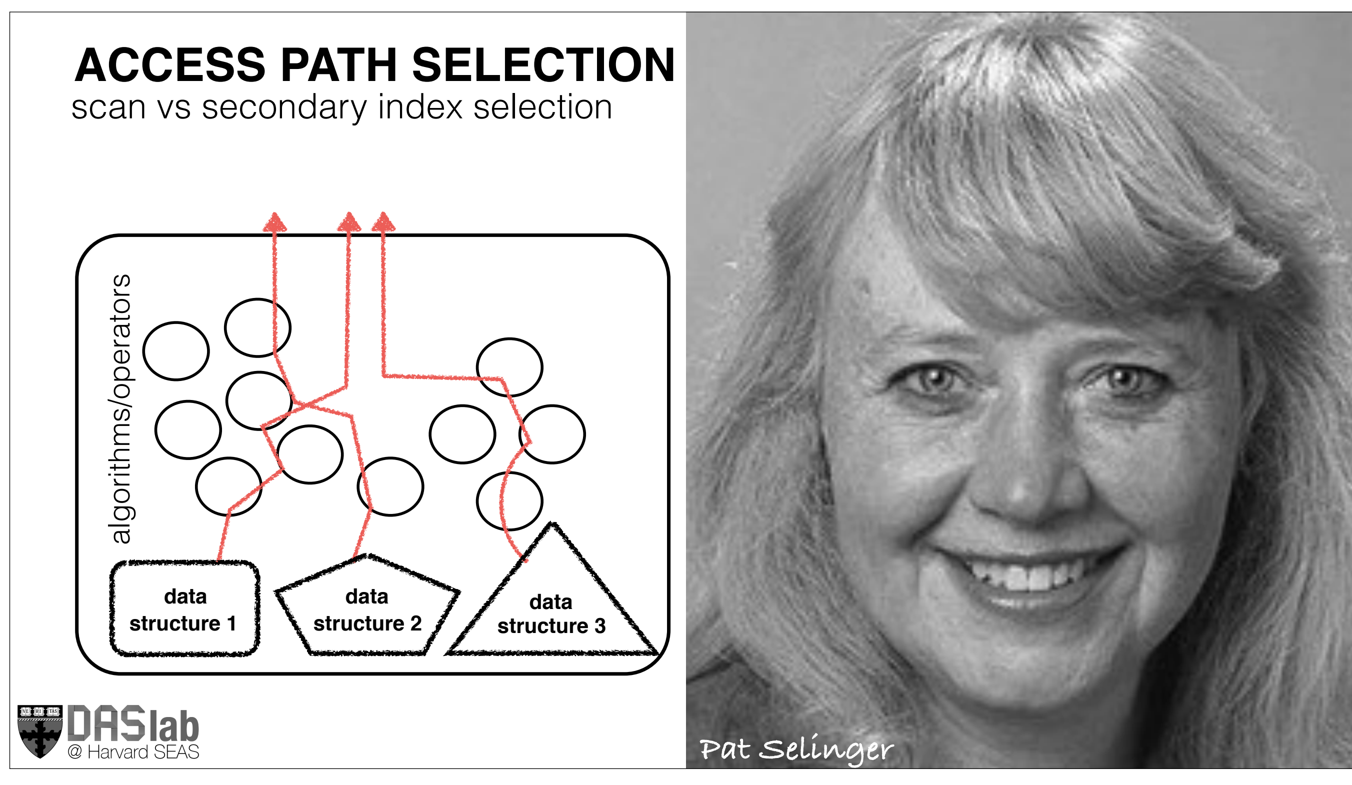

There are three primary ways to judge a data structure/algorithm design. Generalized models sounds like a very promising direction as it gives results close to what we get from the actual code, but it is a model.

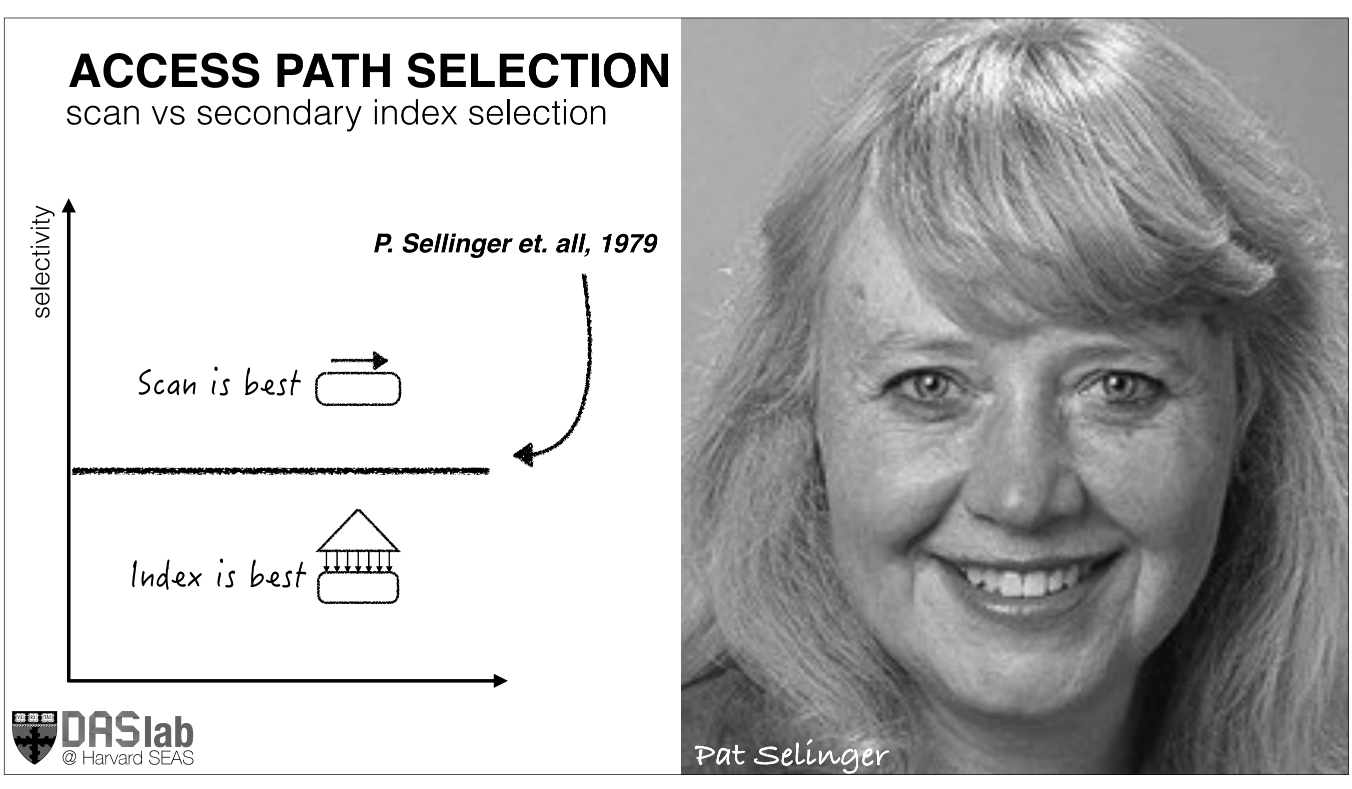


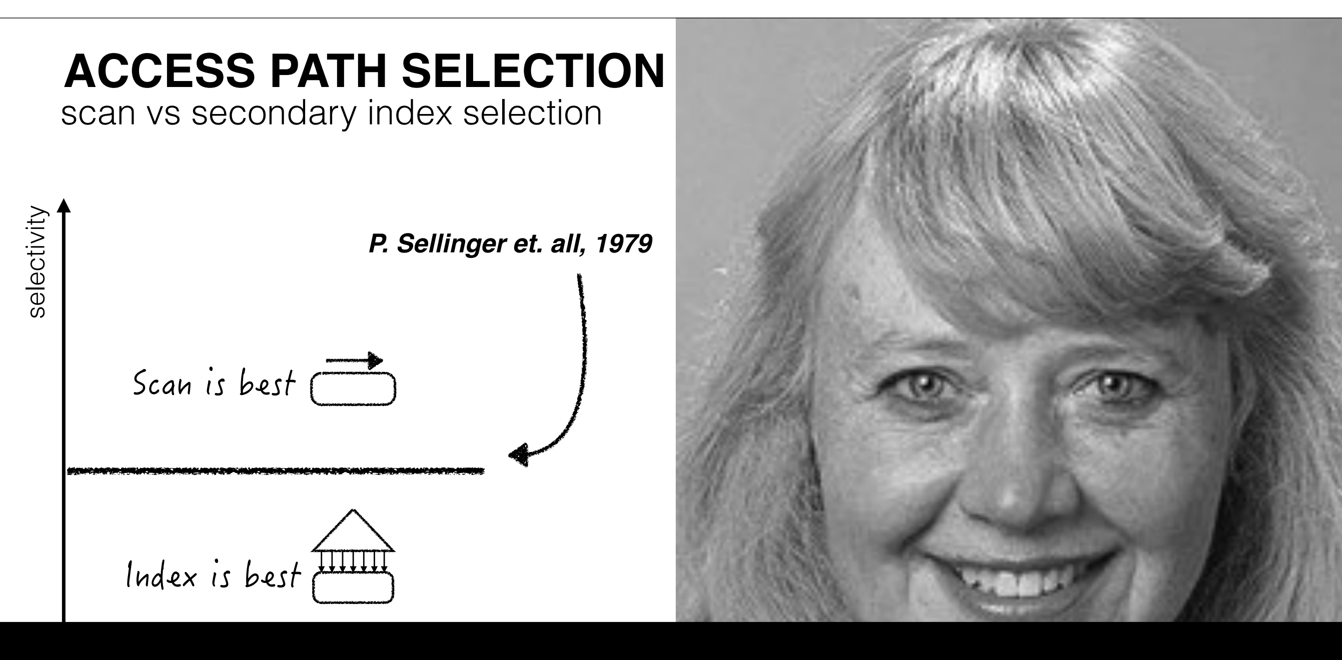

We experimented with generalized models to test the potential. We used the standard access path selection problem as a paradigm. In particular we studies access path selection in memory. Many people claim today that we do not need indexes in memory. There is quite some controversy around this issues. So using generalized models to understand this problem makes for a perfect problem setting.

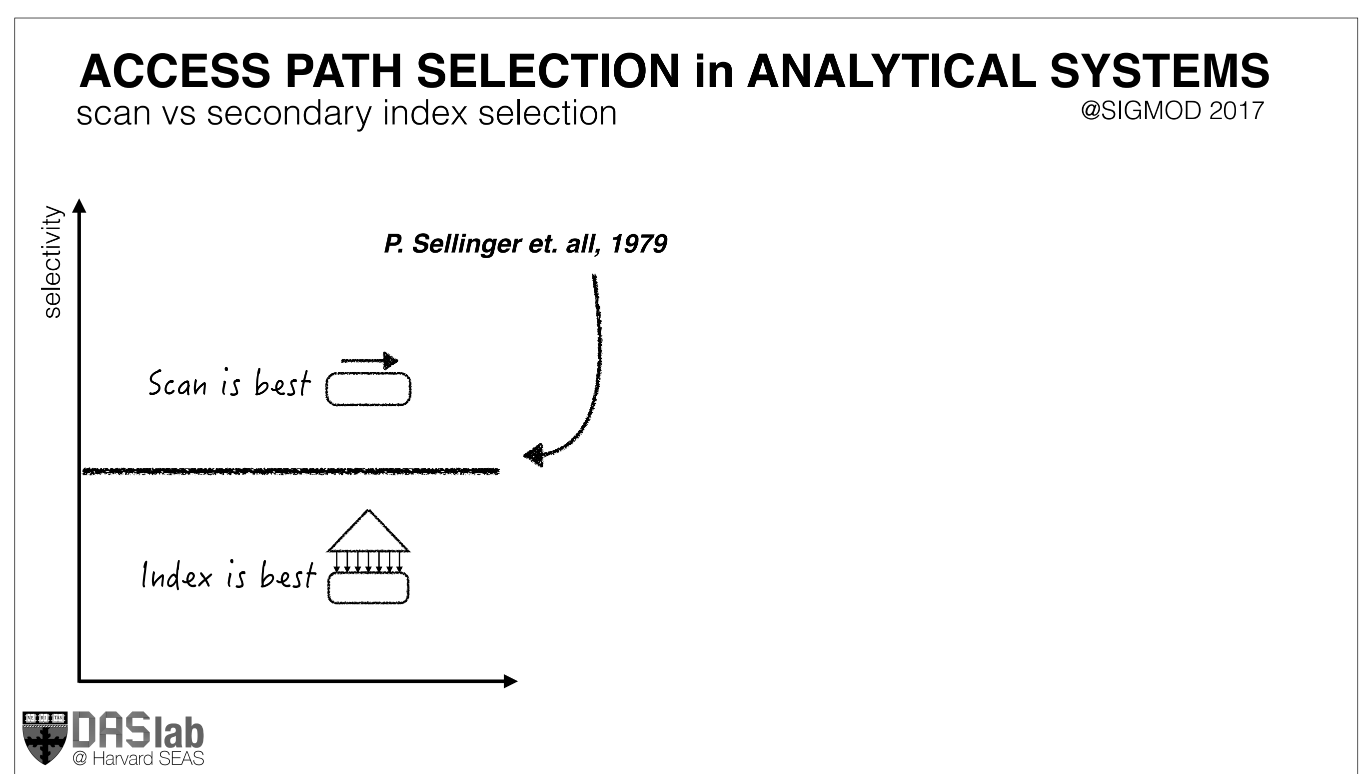



We experimented with generalized models to test the potential. We used the standard access path selection problem as a paradigm. In particular we studies access path selection in memory. Many people claim today that we do not need indexes in memory. There is quite some controversy around this issues. So using generalized models to understand this problem makes for a perfect problem setting.

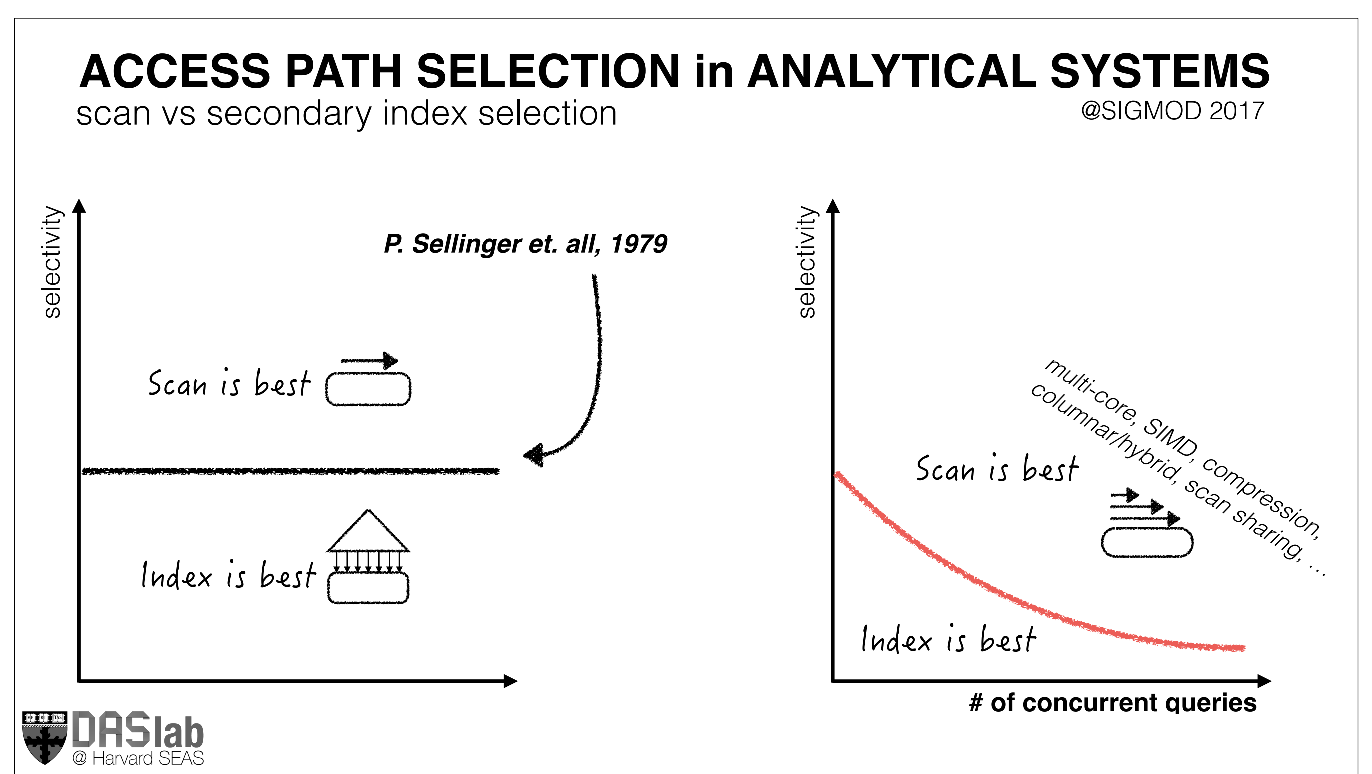

ACCESS PATH SELECTION
scan vs secondary index selection
algorithms/operators
data structure 1
data structure 2
data structure 3
DASlab
@ Harvard SEAS
Pat Selinger


ACCESS PATH SELECTION
scan vs secondary index selection
selectivity
P. Sellinger et. all, 1979
Scan is best
Index is best
DASlab
@ Harvard SEAS
Pat Selinger

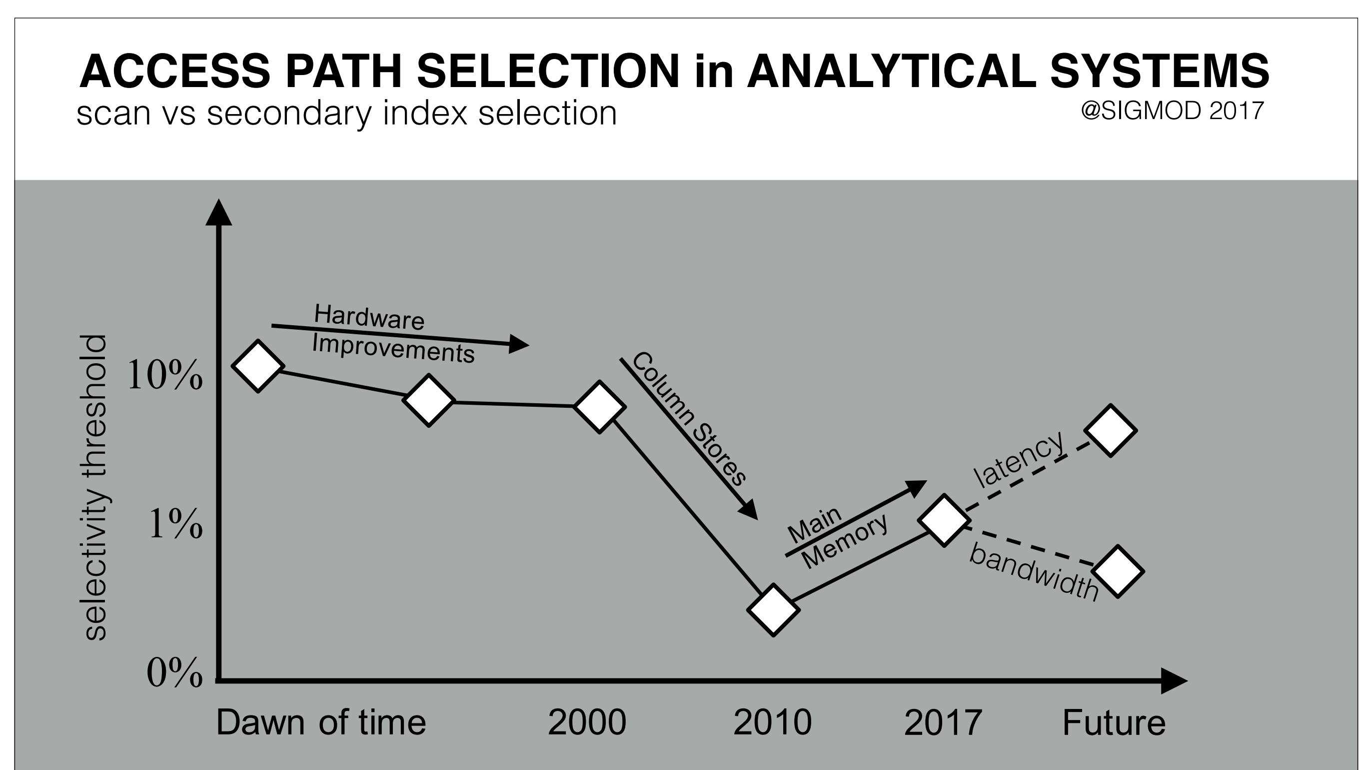

ACCESS PATH SELECTION
scan vs secondary index selection
selectivity
P. Sellinger et. all, 1979
Scan is best
Index is best
DO WE STILL NEED INDEXING? (AND IF YES HOW DO WE CHOOSE)



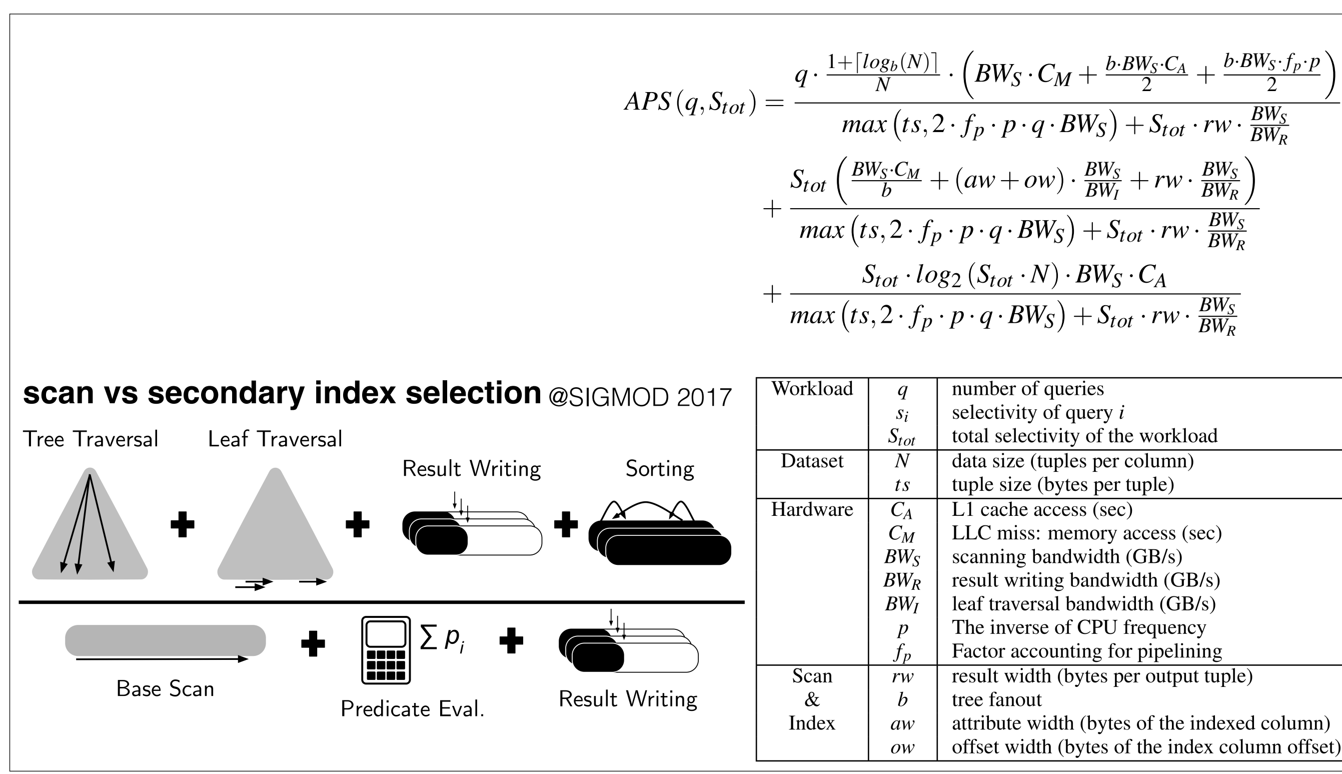

We found many interesting notes. Even in memory it can be beneficial to use indexing. The trick is that we should not have any more a fixed selectivity threshold, and instead we need to have a dynamic threshold which depends on concurrency. We also showed how we can explain the evolution of access path selection tradeoffs over the years. And also what may happen as hardware changes in the future.




We found many interesting notes. Even in memory it can be beneficial to use indexing. The trick is that we should not have any more a fixed selectivity threshold, and instead we need to have a dynamic threshold which depends on concurrency. We also showed how we can explain the evolution of access path selection tradeoffs over the years. And also what may happen as hardware changes in the future.



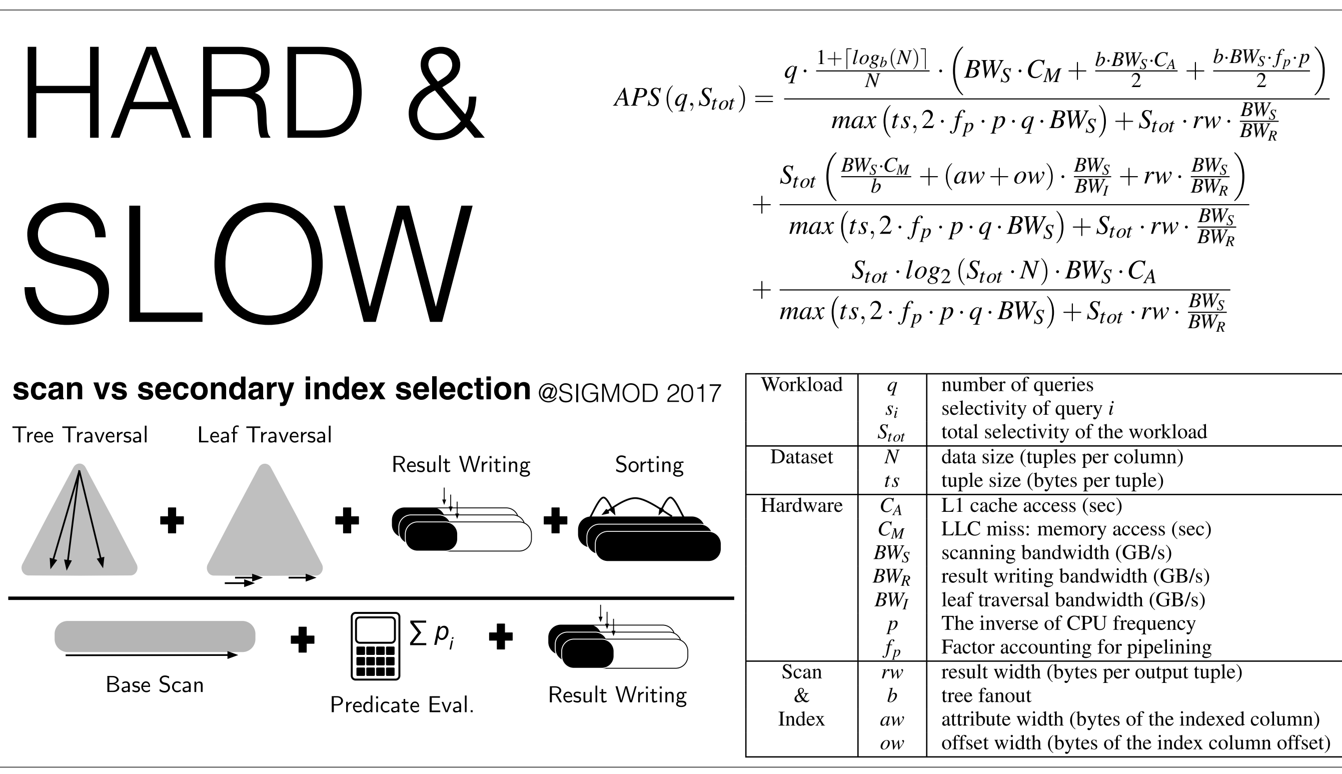

We found many interesting notes. Even in memory it can be beneficial to use indexing. The trick is that we should not have any more a fixed selectivity threshold, and instead we need to have a dynamic threshold which depends on concurrency. We also showed how we can explain the evolution of access path selection tradeoffs over the years. And also what may happen as hardware changes in the future.




$$APS(q,S_{tot}) = \frac{q \cdot \frac{1+\lceil log_b(N) \rceil}{N} \cdot \left( BW_S \cdot C_M + \frac{b \cdot BW_S \cdot C_A}{2} + \frac{b \cdot BW_S \cdot f_p \cdot p}{2} \right)}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} + \frac{S_{tot}\left( \frac{BW_S \cdot C_M}{b} + (aw+ow) \cdot \frac{BW_S}{BW_I} + rw \cdot \frac{BW_S}{BW_R} \right)}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} + \frac{S_{tot} \cdot log_2\left(S_{tot} \cdot N\right) \cdot BW_S \cdot C_A}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}}$$

| | | |
|---|---|---|
| Workload | $q$ | number of queries |
| | $s_i$ | selectivity of query $i$ |
| | $S_{tot}$ | total selectivity of the workload |
| Dataset | $N$ | data size (tuples per column) |
| | $ts$ | tuple size (bytes per tuple) |
| Hardware | $C_A$ | L1 cache access (sec) |
| | $C_M$ | LLC miss: memory access (sec) |
| | $BW_S$ | scanning bandwidth (GB/s) |
| | $BW_R$ | result writing bandwidth (GB/s) |
| | $BW_I$ | leaf traversal bandwidth (GB/s) |
| | $p$ | The inverse of CPU frequency |
| | $f_p$ | Factor accounting for pipelining |
| Scan & Index | $rw$ | result width (bytes per output tuple) |
| | $b$ | tree fanout |
| | $aw$ | attribute width (bytes of the indexed column) |
| | $ow$ | offset width (bytes of the index column offset) |

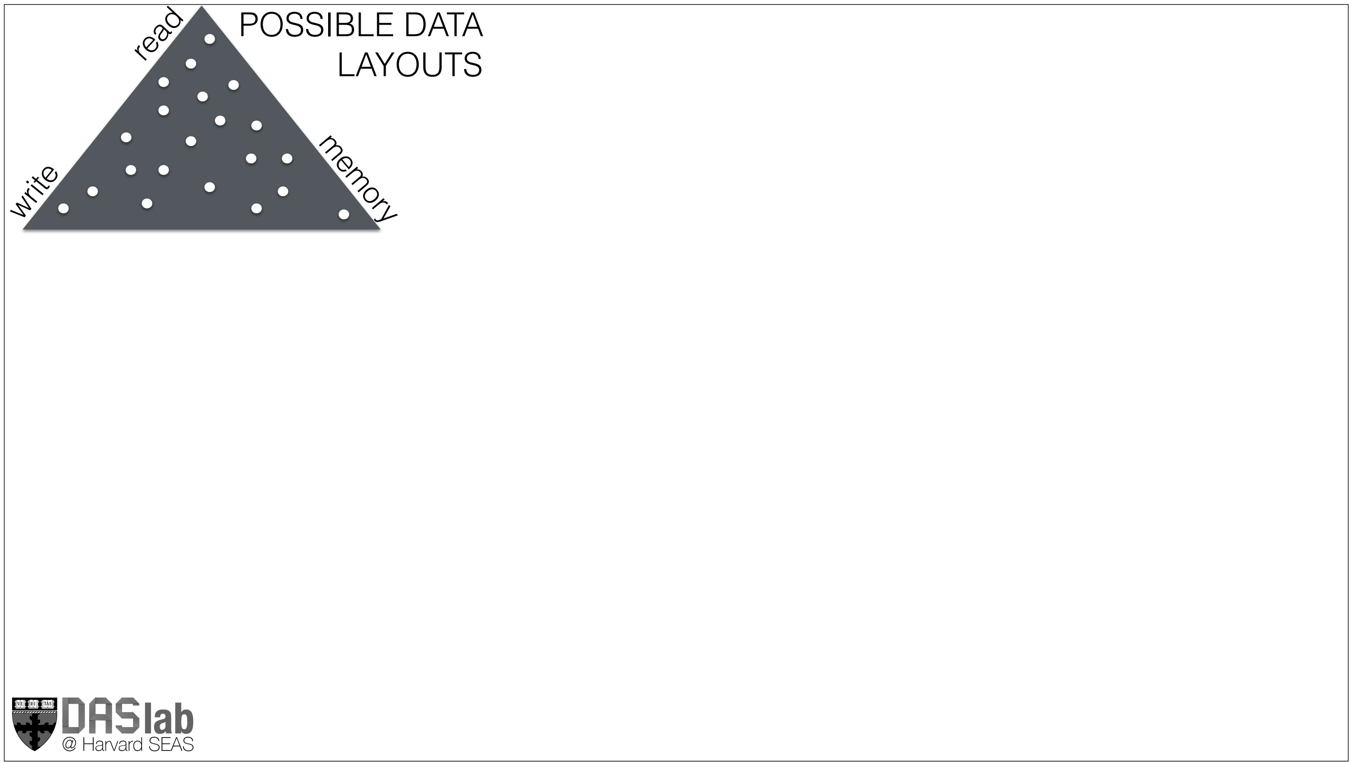
We did all that through detailed generalized modes. The problem is that this was extremely hard and time consuming to do. And this was only for two data structures, b-tree and array. This is not sustainable to do for the whole design space of data structures…



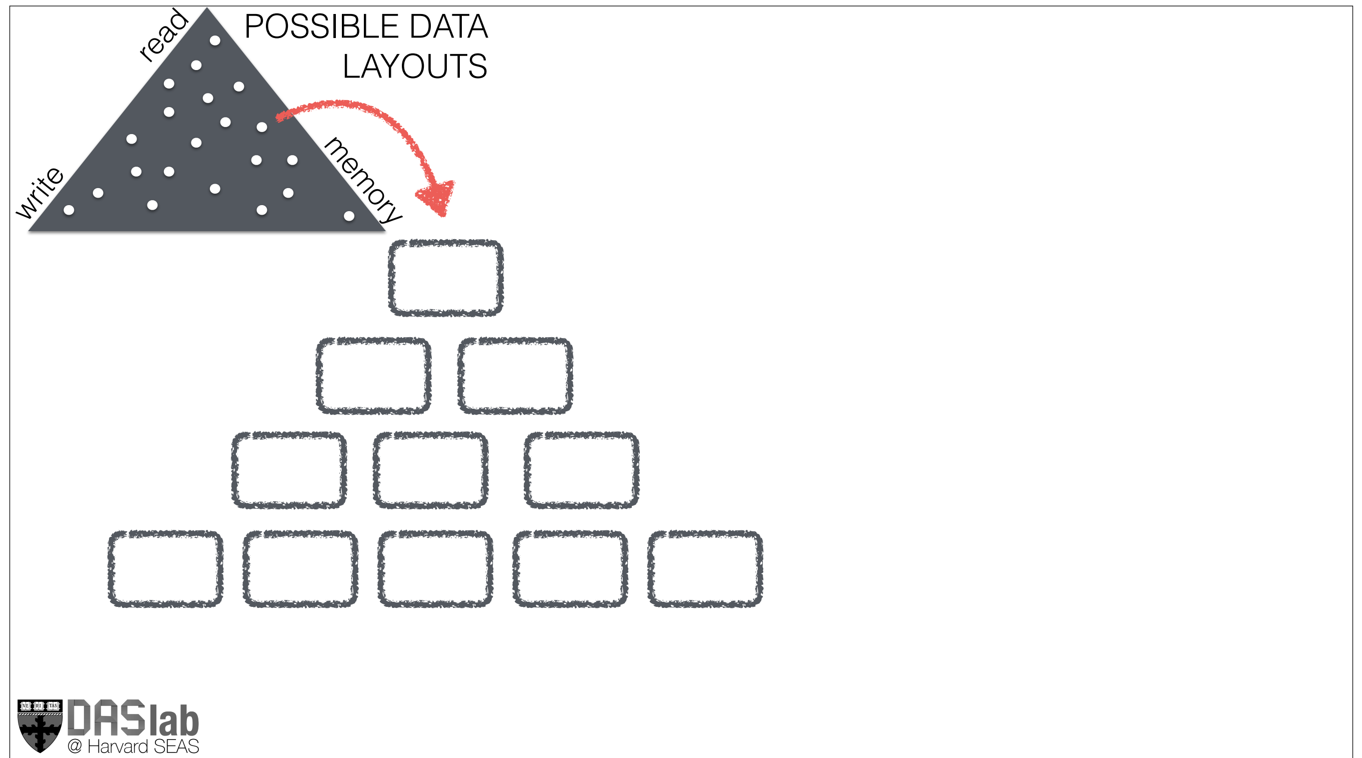

$$APS(q,S_{tot}) = \frac{q \cdot \frac{1+\lceil log_b(N) \rceil}{N} \cdot \left( BW_S \cdot C_M + \frac{b \cdot BW_S \cdot C_A}{2} + \frac{b \cdot BW_S \cdot f_p \cdot p}{2} \right)}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} + \frac{S_{tot}\left(\frac{BW_S \cdot C_M}{b} + (aw+ow) \cdot \frac{BW_S}{BW_I} + rw \cdot \frac{BW_S}{BW_R}\right)}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} + \frac{S_{tot} \cdot log_2\left(S_{tot} \cdot N\right) \cdot BW_S \cdot C_A}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}}$$

| | | |
|---|---|---|
| Workload | $q$ | number of queries |
| | $s_i$ | selectivity of query $i$ |
| | $S_{tot}$ | total selectivity of the workload |
| Dataset | $N$ | data size (tuples per column) |
| | $ts$ | tuple size (bytes per tuple) |
| Hardware | $C_A$ | L1 cache access (sec) |
| | $C_M$ | LLC miss: memory access (sec) |
| | $BW_S$ | scanning bandwidth (GB/s) |
| | $BW_R$ | result writing bandwidth (GB/s) |
| | $BW_I$ | leaf traversal bandwidth (GB/s) |
| | $p$ | The inverse of CPU frequency |
| | $f_p$ | Factor accounting for pipelining |
| Scan & Index | $rw$ | result width (bytes per output tuple) |
| | $b$ | tree fanout |
| | $aw$ | attribute width (bytes of the indexed column) |
| | $ow$ | offset width (bytes of the index column offset) |

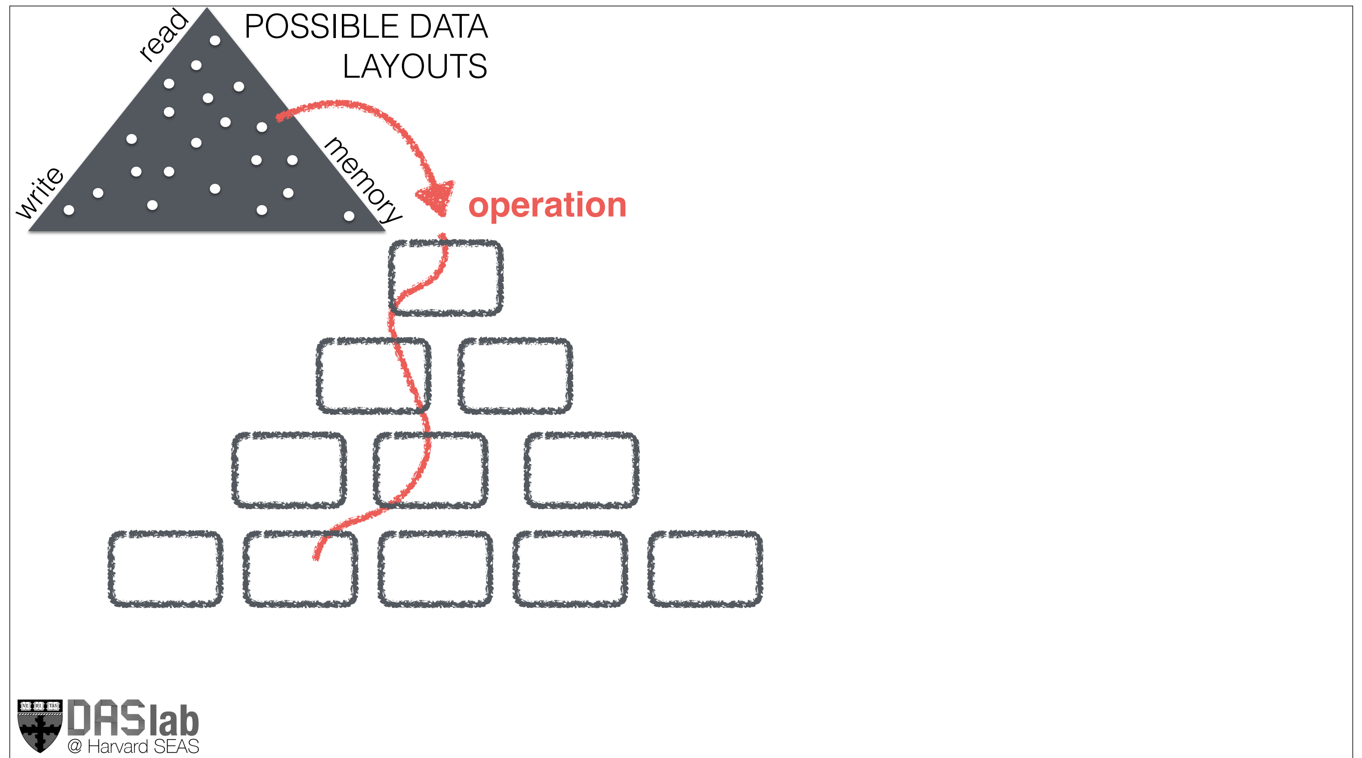
We did all that through detailed generalized modes. The problem is that this was extremely hard and time consuming to do. And this was only for two data structures, b-tree and array. This is not sustainable to do for the whole design space of data structures…



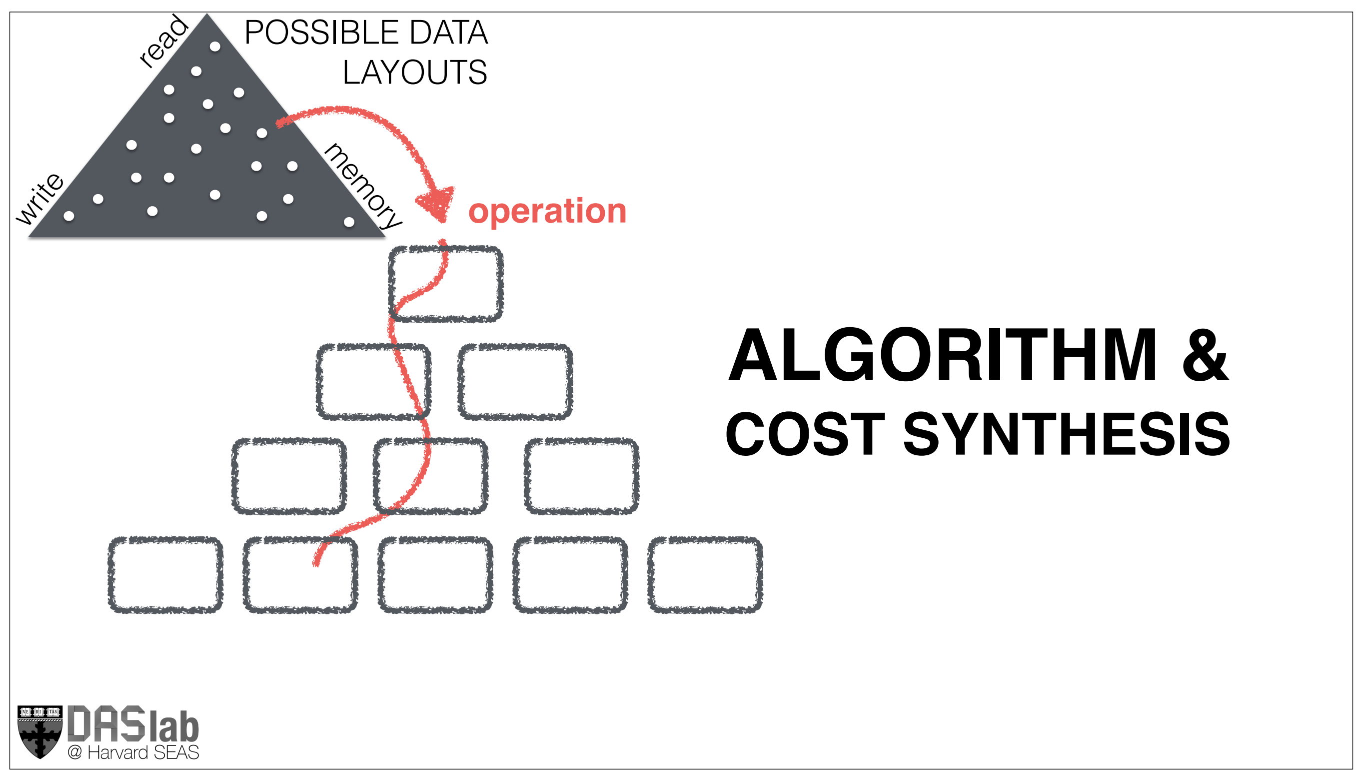

What we really want is to be able to pick an arbitrary data structure from the design space and given a target operation we would like to automatically compute the best algorithm to implement this operation and the cost of running this algorithm.



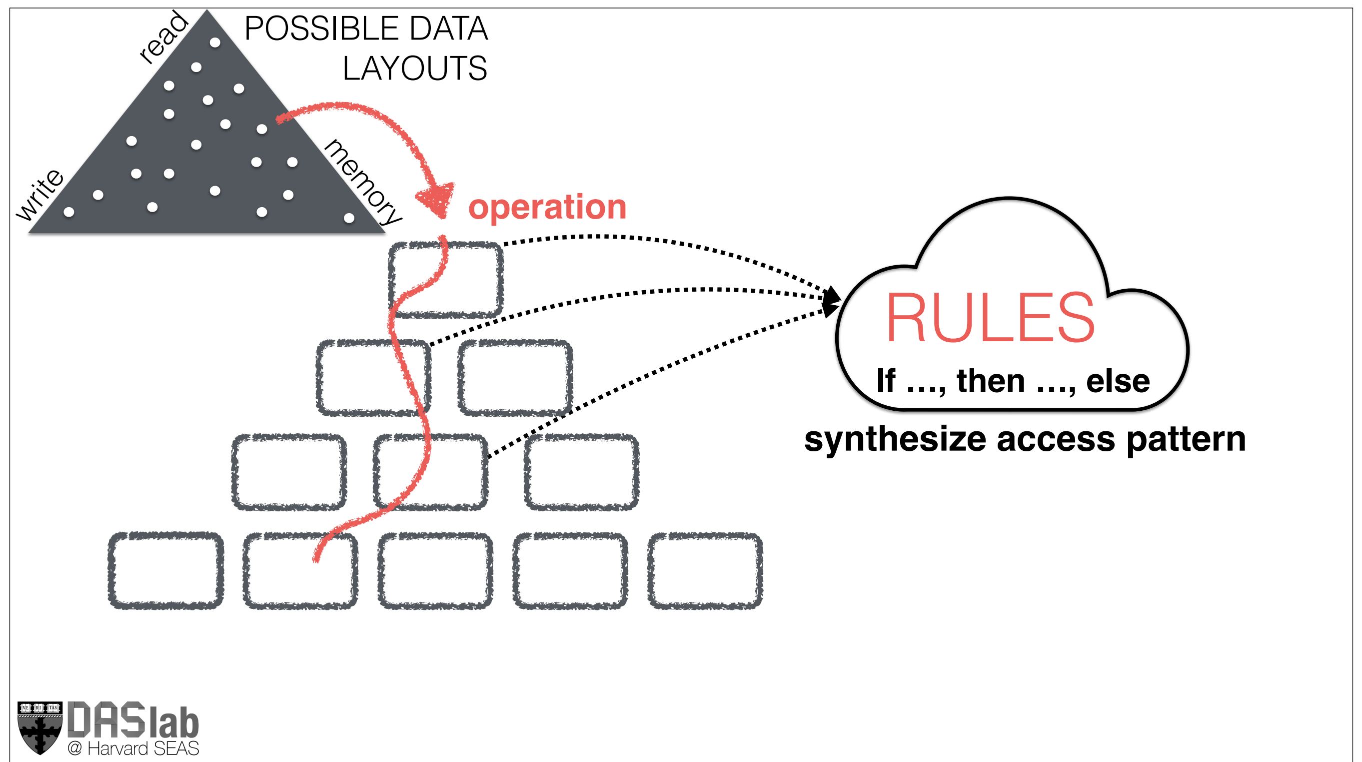

What we really want is to be able to pick an arbitrary data structure from the design space and given a target operation we would like to automatically compute the best algorithm to implement this operation and the cost of running this algorithm.




What we really want is to be able to pick an arbitrary data structure from the design space and given a target operation we would like to automatically compute the best algorithm to implement this operation and the cost of running this algorithm.

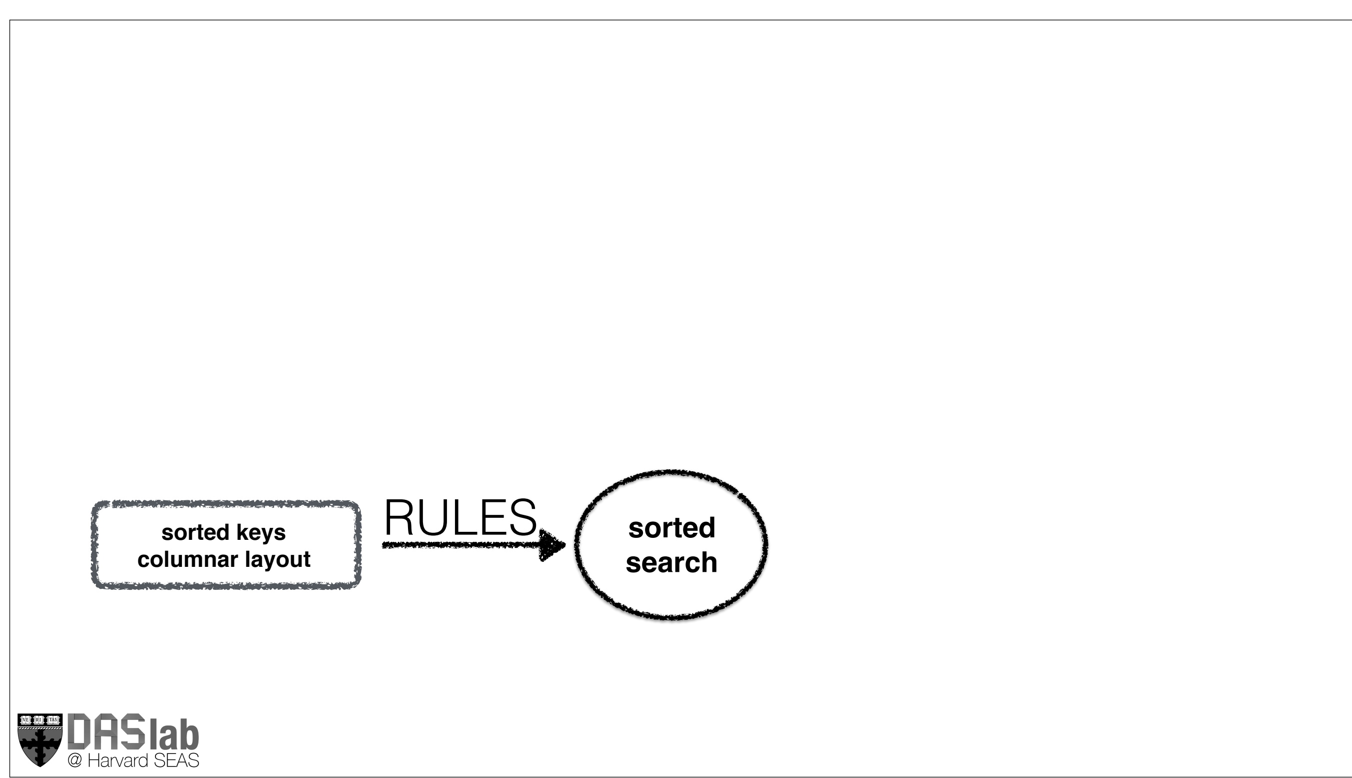



What we really want is to be able to pick an arbitrary data structure from the design space and given a target operation we would like to automatically compute the best algorithm to implement this operation and the cost of running this algorithm.

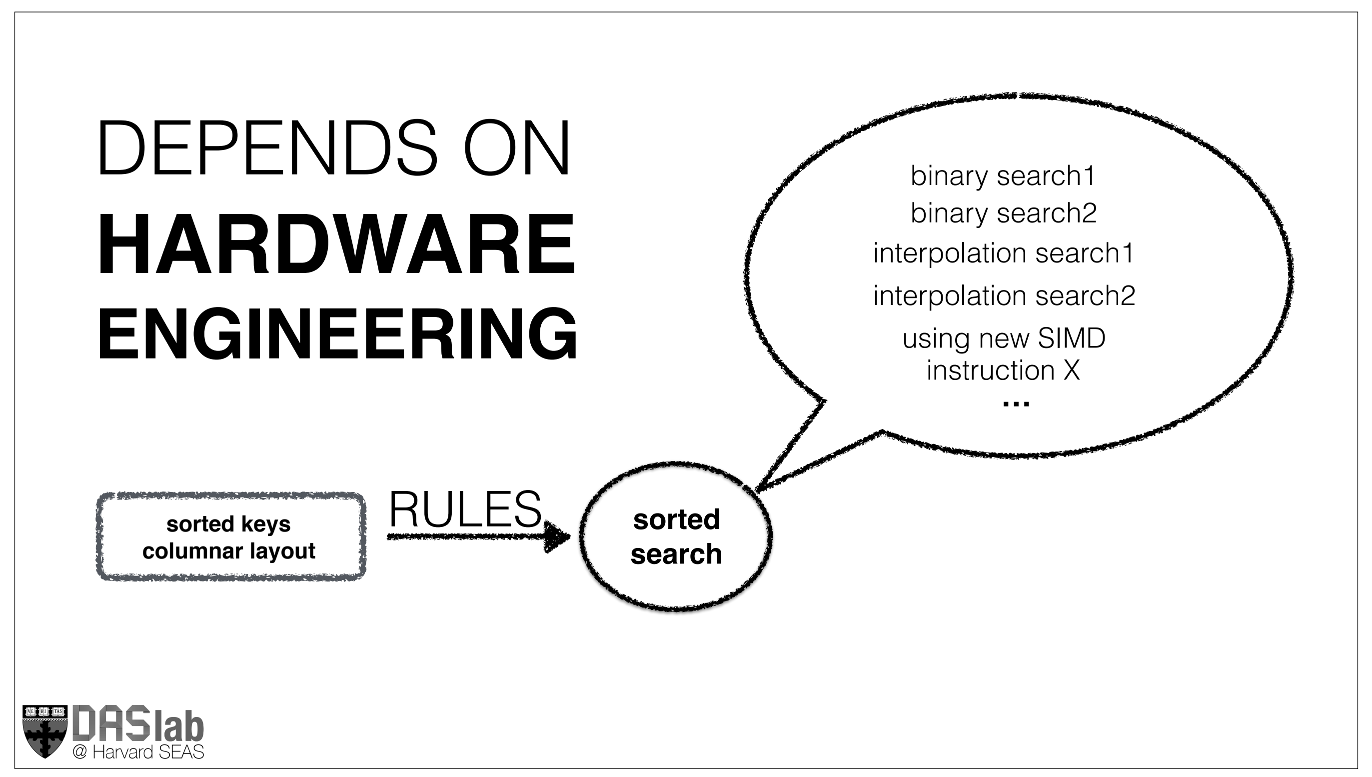



The first idea that comes to mind is creating an elaborate set of rules that given the layout map to algorithms.

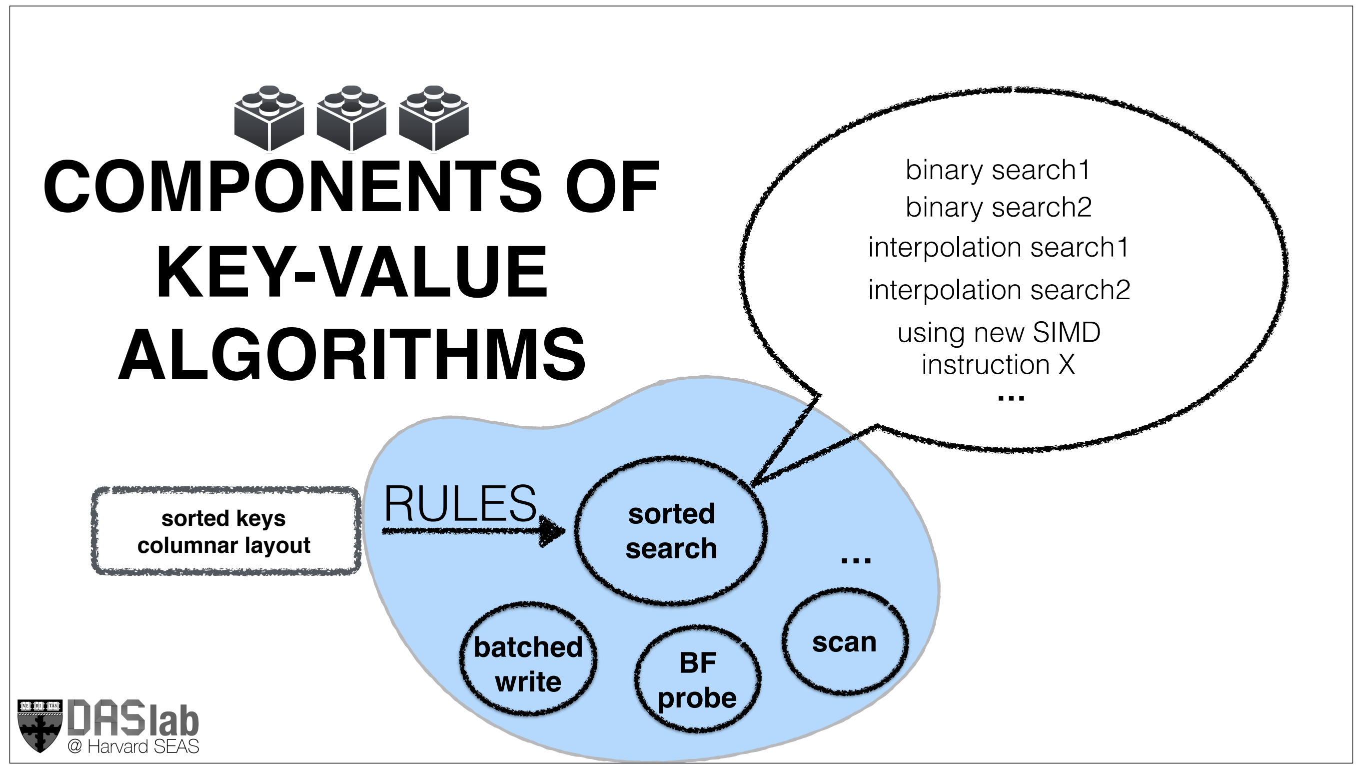
The problem is that this would be extremely complex. As past research in AI has shown maintaining complex rule based systems does not scale. What we do instead is that we rely again on synthesis from first principles. We figure out what are the fundamental building blocks for the algorithms of key-value structures, such as sorted search, scan, random access, etc. And then we develop models that describe how they behave. The trick here is that we may in fact have several options for each one of those fundamental algorithmic components. For example, sorted search may happen in many different ways and even specific implementations would be slightly different and also the hardware affects performance in critical ways. To address all those points we develop learned cost models that describe the key-value algorithms.



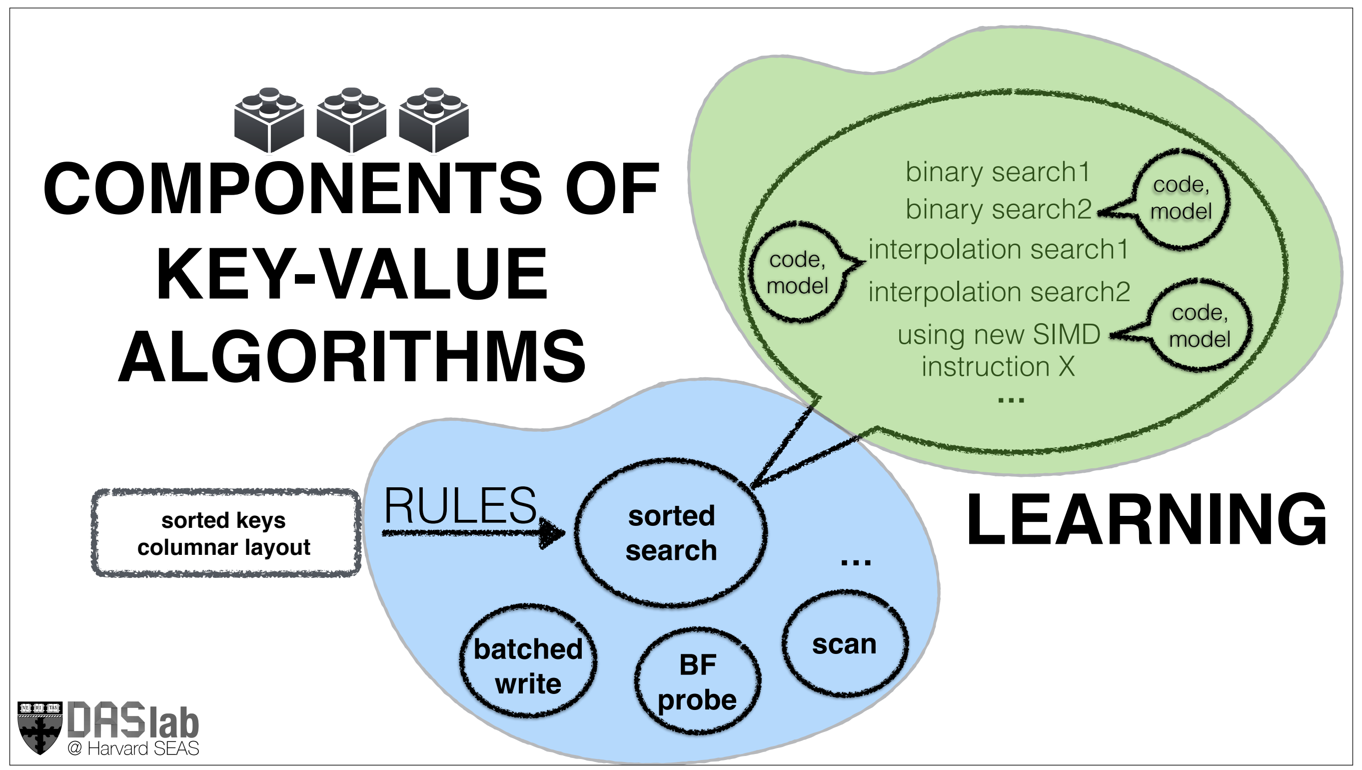

The problem is that this would be extremely complex. As past research in AI has shown maintaining complex rule based systems does not scale. What we do instead is that we rely again on synthesis from first principles. We figure out what are the fundamental building blocks for the algorithms of key-value structures, such as sorted search, scan, random access, etc. And then we develop models that describe how they behave. The trick here is that we may in fact have several options for each one of those fundamental algorithmic components. For example, sorted search may happen in many different ways and even specific implementations would be slightly different and also the hardware affects performance in critical ways. To address all those points we develop learned cost models that describe the key-value algorithms.



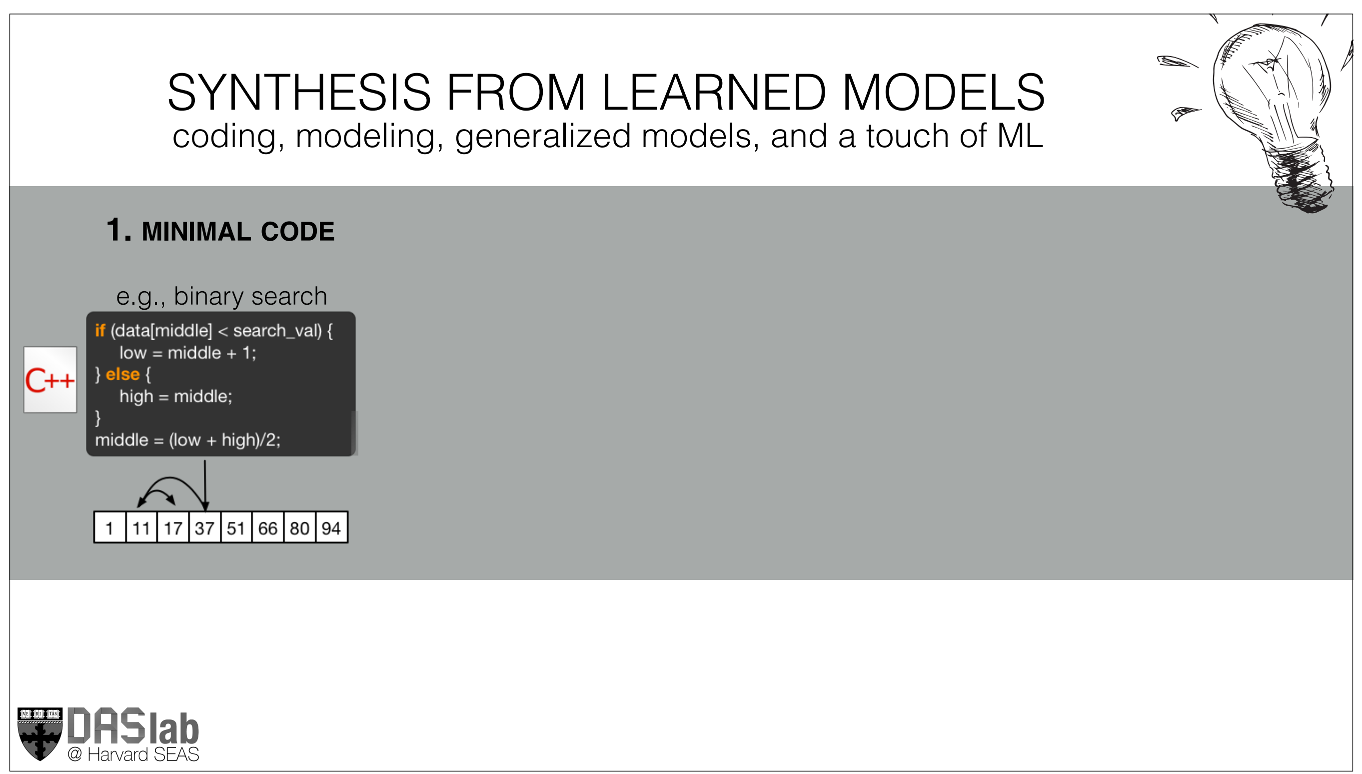

The problem is that this would be extremely complex. As past research in AI has shown maintaining complex rule based systems does not scale. What we do instead is that we rely again on synthesis from first principles. We figure out what are the fundamental building blocks for the algorithms of key-value structures, such as sorted search, scan, random access, etc. And then we develop models that describe how they behave. The trick here is that we may in fact have several options for each one of those fundamental algorithmic components. For example, sorted search may happen in many different ways and even specific implementations would be slightly different and also the hardware affects performance in critical ways. To address all those points we develop learned cost models that describe the key-value algorithms.



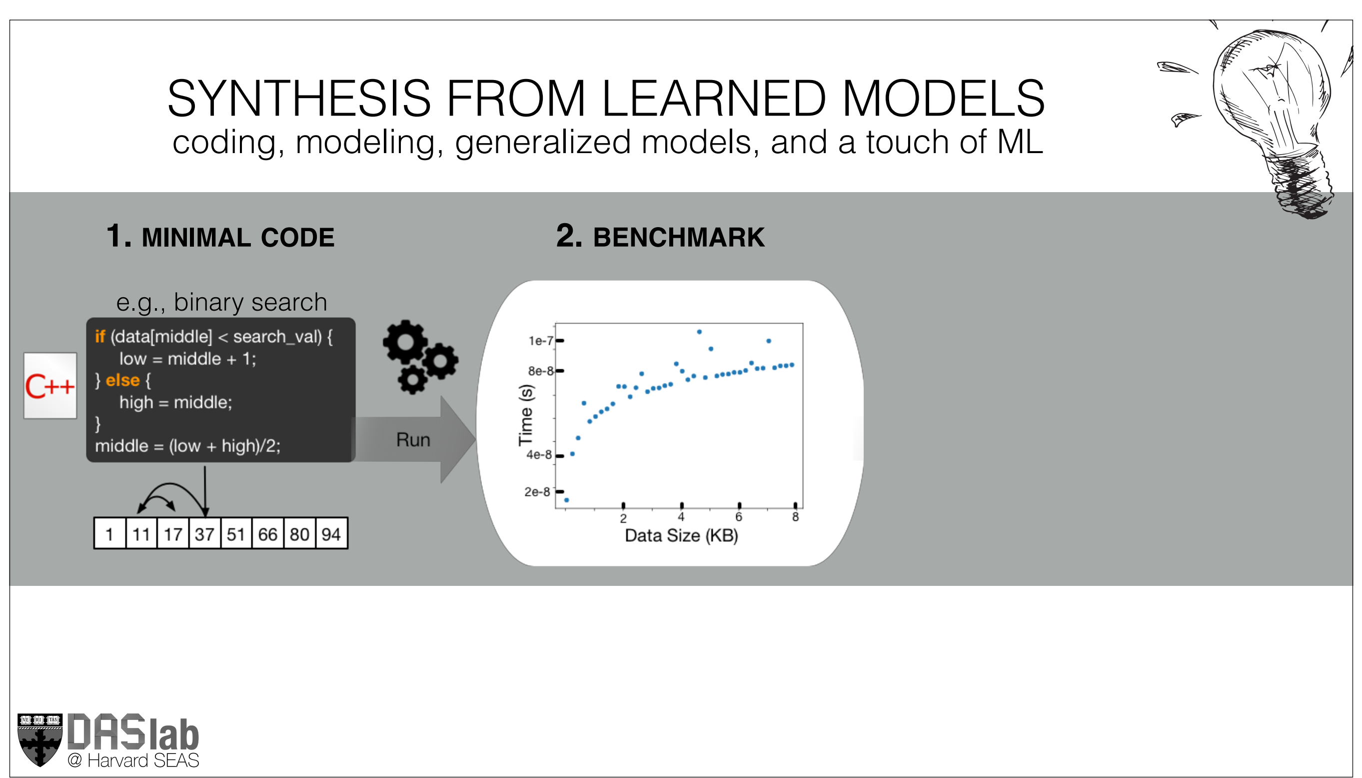

The problem is that this would be extremely complex. As past research in AI has shown maintaining complex rule based systems does not scale. What we do instead is that we rely again on synthesis from first principles. We figure out what are the fundamental building blocks for the algorithms of key-value structures, such as sorted search, scan, random access, etc. And then we develop models that describe how they behave. The trick here is that we may in fact have several options for each one of those fundamental algorithmic components. For example, sorted search may happen in many different ways and even specific implementations would be slightly different and also the hardware affects performance in critical ways. To address all those points we develop learned cost models that describe the key-value algorithms.



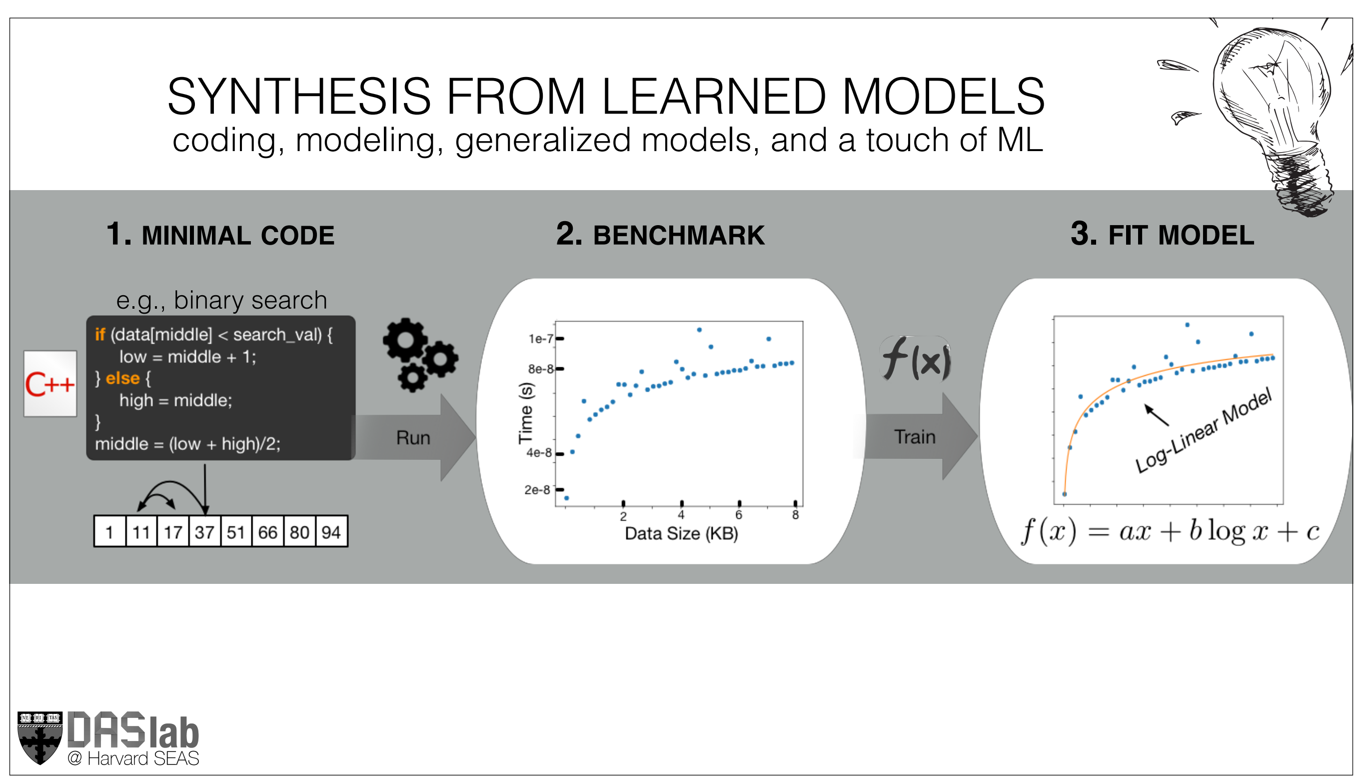

The problem is that this would be extremely complex. As past research in AI has shown maintaining complex rule based systems does not scale. What we do instead is that we rely again on synthesis from first principles. We figure out what are the fundamental building blocks for the algorithms of key-value structures, such as sorted search, scan, random access, etc. And then we develop models that describe how they behave. The trick here is that we may in fact have several options for each one of those fundamental algorithmic components. For example, sorted search may happen in many different ways and even specific implementations would be slightly different and also the hardware affects performance in critical ways. To address all those points we develop learned cost models that describe the key-value algorithms.



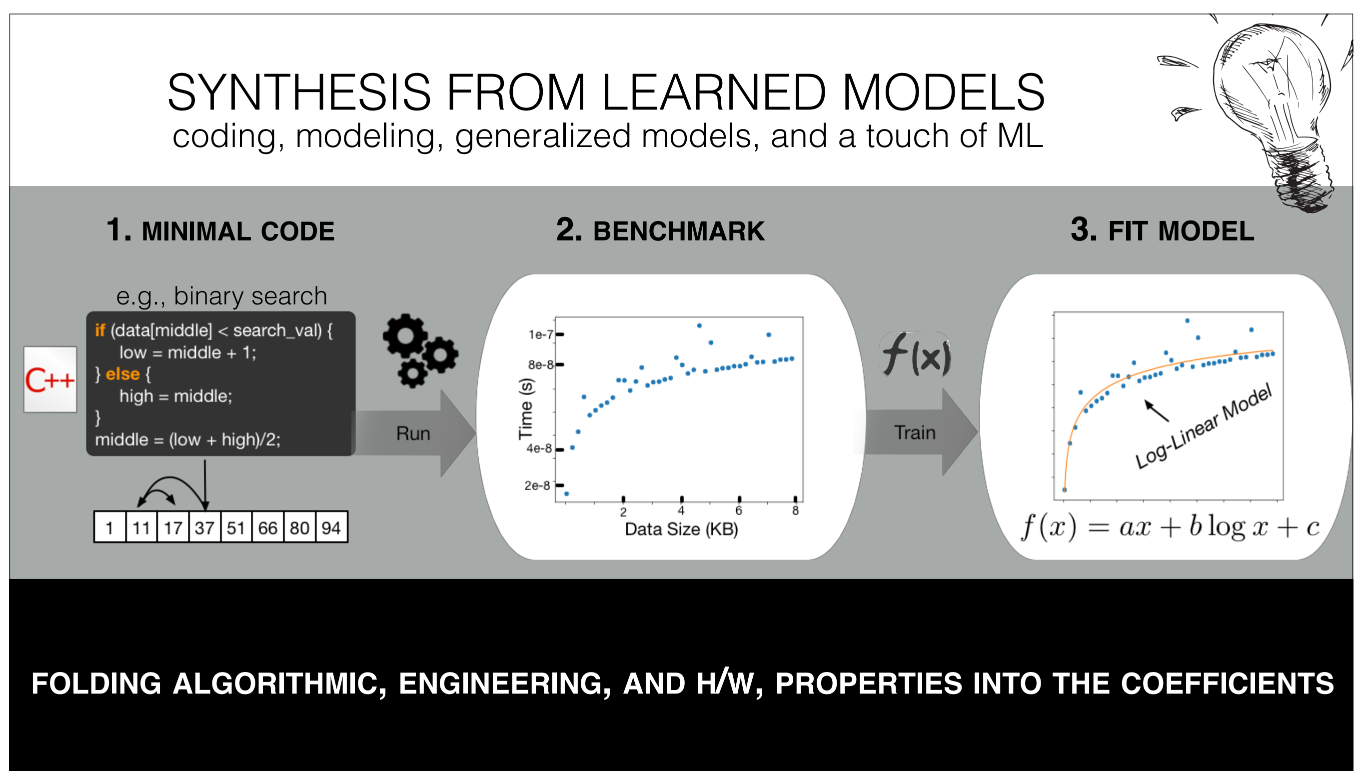

The idea is that each possible different design and implementation of a component is represented by a learned model. We need the minimum possible implementation that describes the behavior of the algorithmic component, and then we need to test it on teh given data and hardware to learn a model.



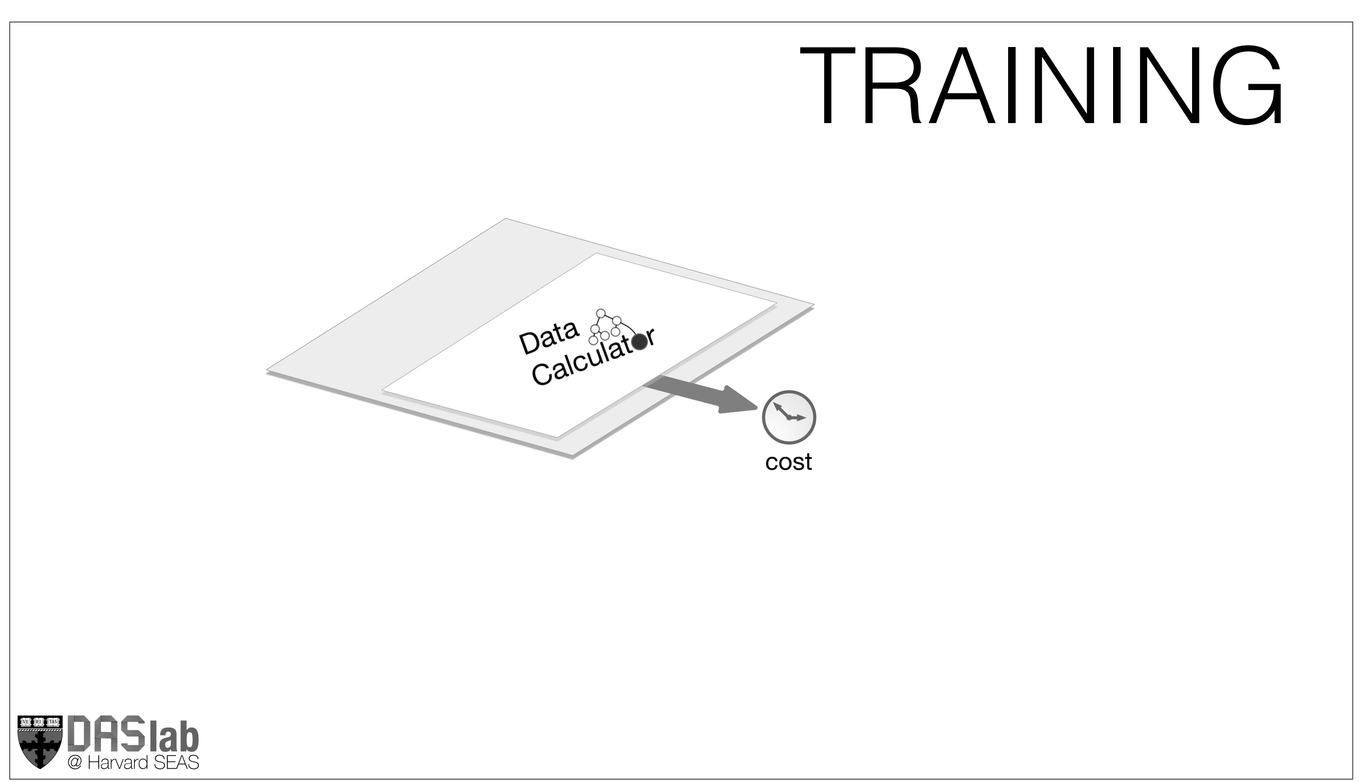

The idea is that each possible different design and implementation of a component is represented by a learned model. We need the minimum possible implementation that describes the behavior of the algorithmic component, and then we need to test it on teh given data and hardware to learn a model.



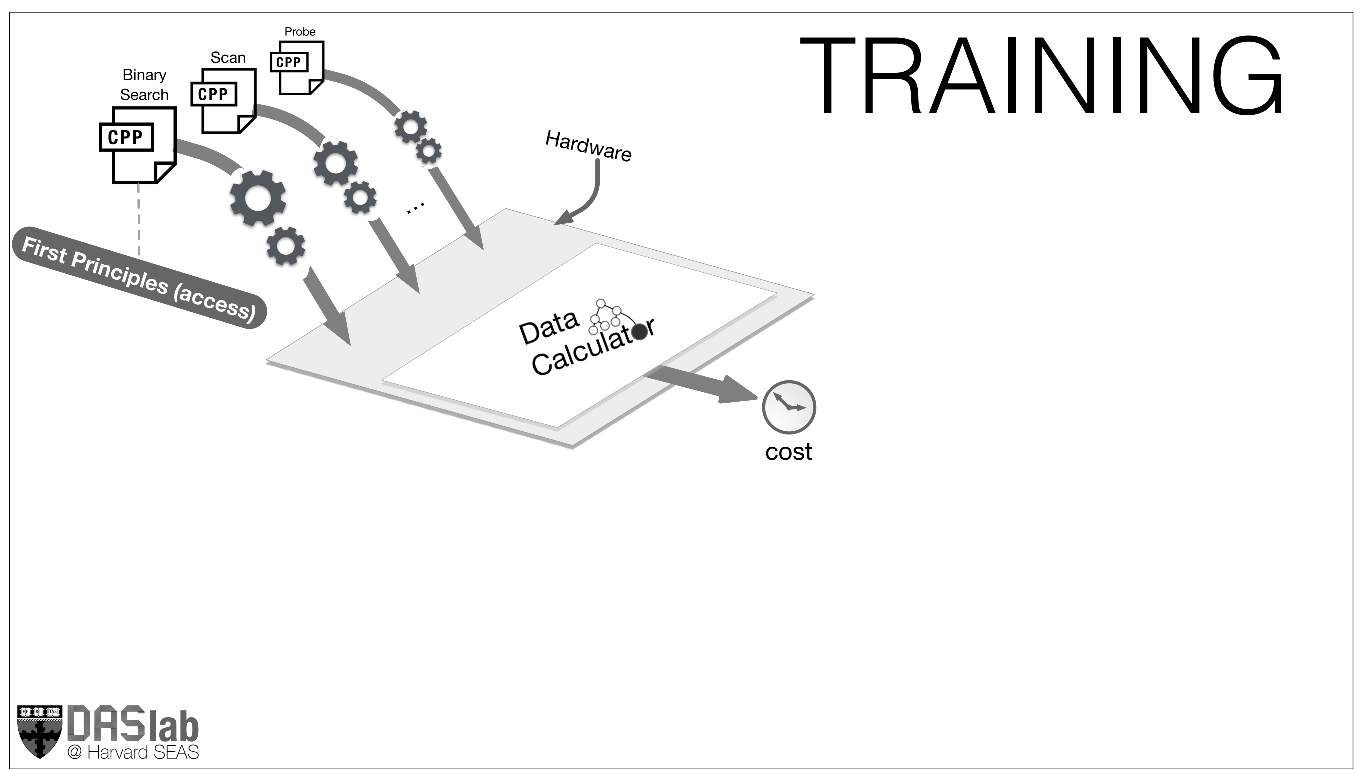

The idea is that each possible different design and implementation of a component is represented by a learned model. We need the minimum possible implementation that describes the behavior of the algorithmic component, and then we need to test it on teh given data and hardware to learn a model.



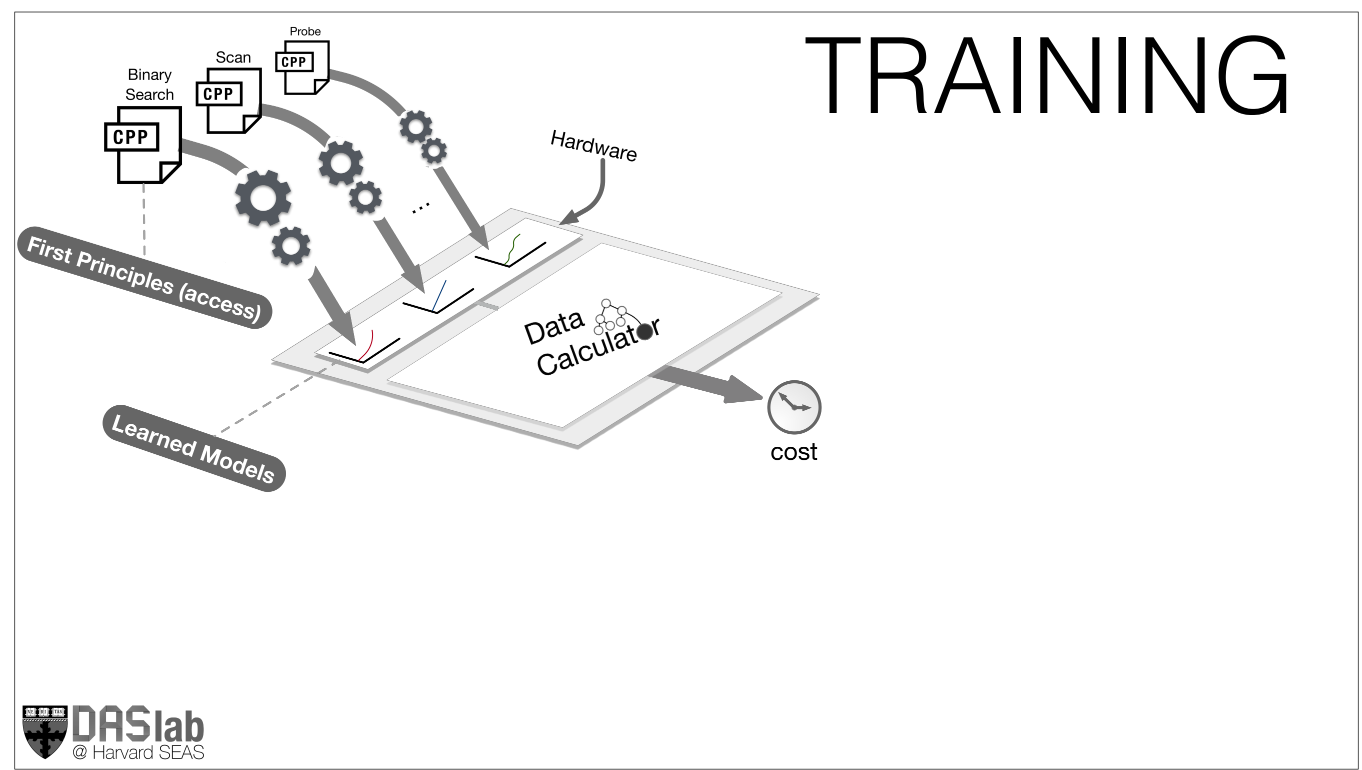

The idea is that each possible different design and implementation of a component is represented by a learned model. We need the minimum possible implementation that describes the behavior of the algorithmic component, and then we need to test it on teh given data and hardware to learn a model.



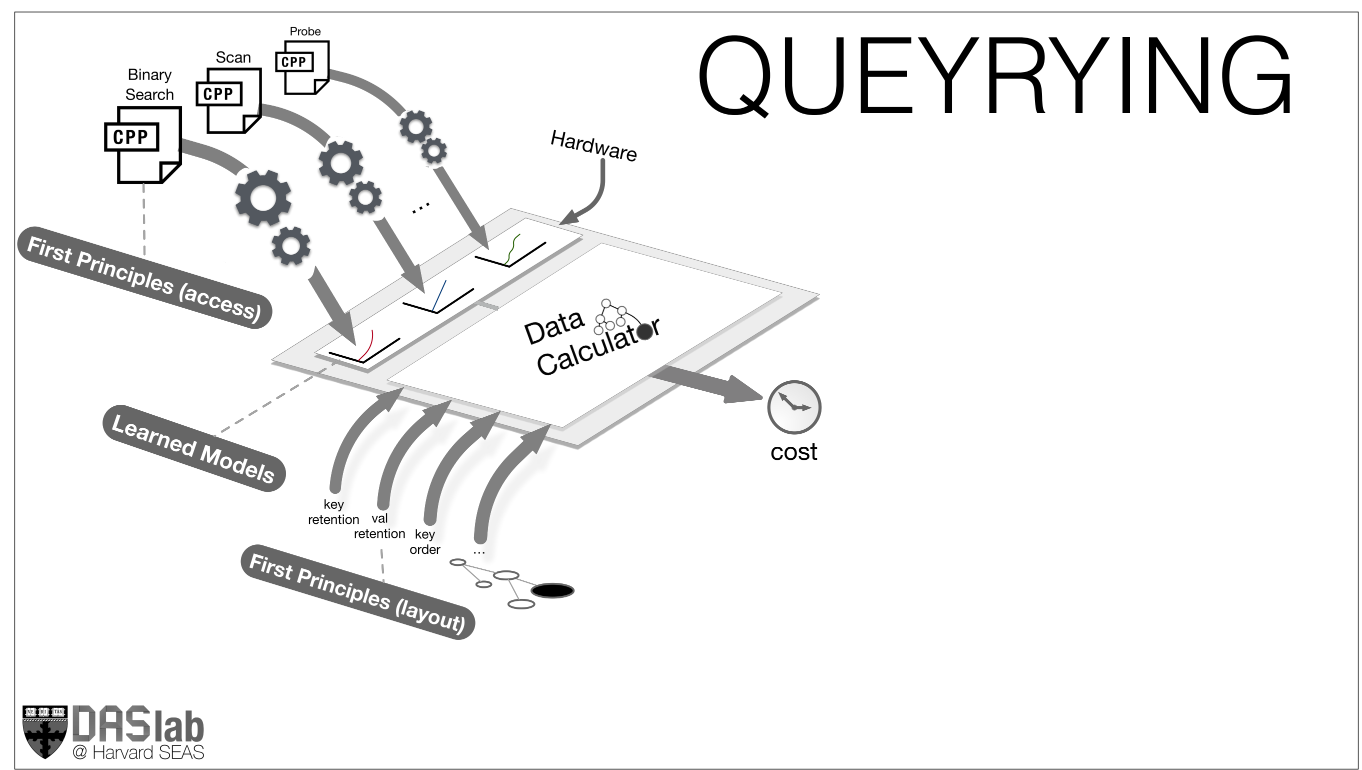

Overall the calculator first needs some time to train its models on the target hardware: this takes about 1 minute. Then it can take as input the layout of a target data structure and a workload (data and queries) and it computes the cost to run those queries over this data structure. It does this by constructing the state of the data structure at each step of the workload, computing the algorithm for each operation, and the path that the algorithm should follow over the data structure. It then assembles the cost using its models for each algorithmic component.



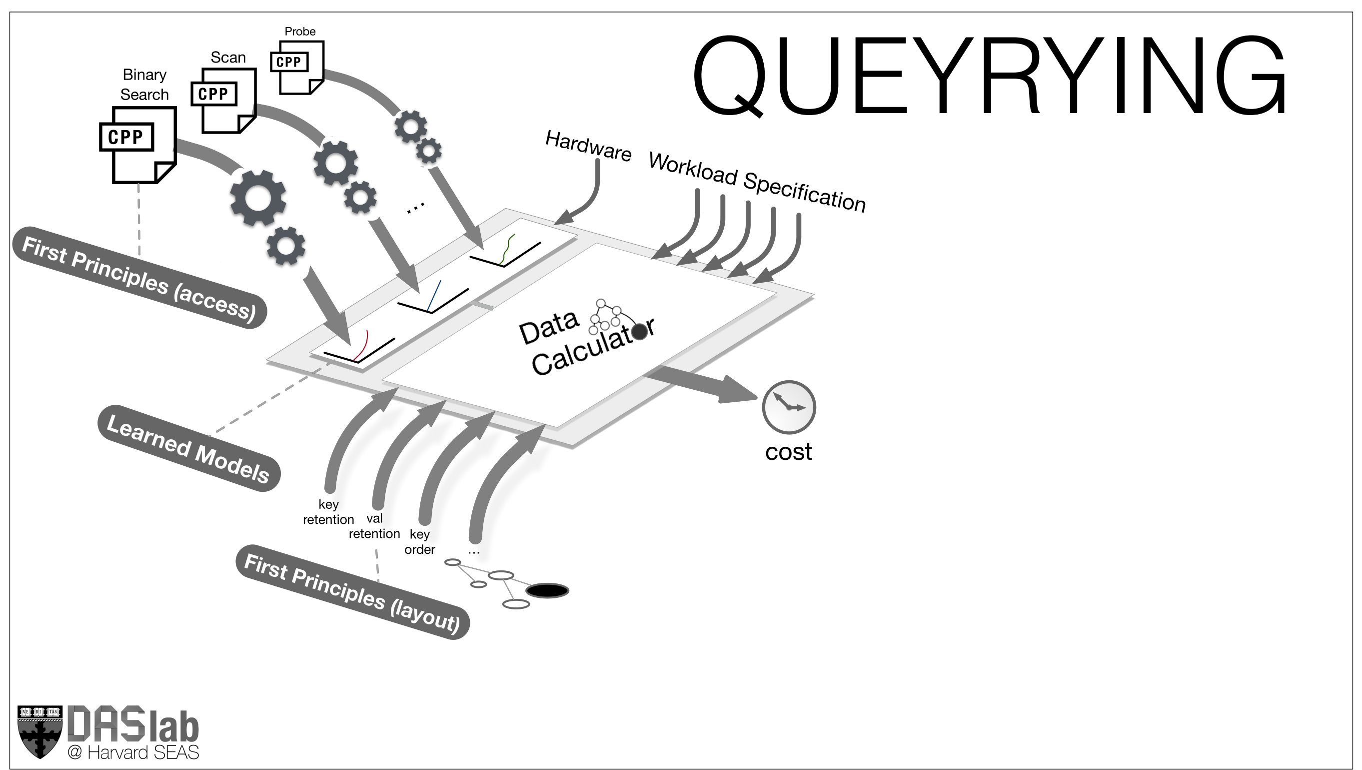

Overall the calculator first needs some time to train its models on the target hardware: this takes about 1 minute. Then it can take as input the layout of a target data structure and a workload (data and queries) and it computes the cost to run those queries over this data structure. It does this by constructing the state of the data structure at each step of the workload, computing the algorithm for each operation, and the path that the algorithm should follow over the data structure. It then assembles the cost using its models for each algorithmic component.



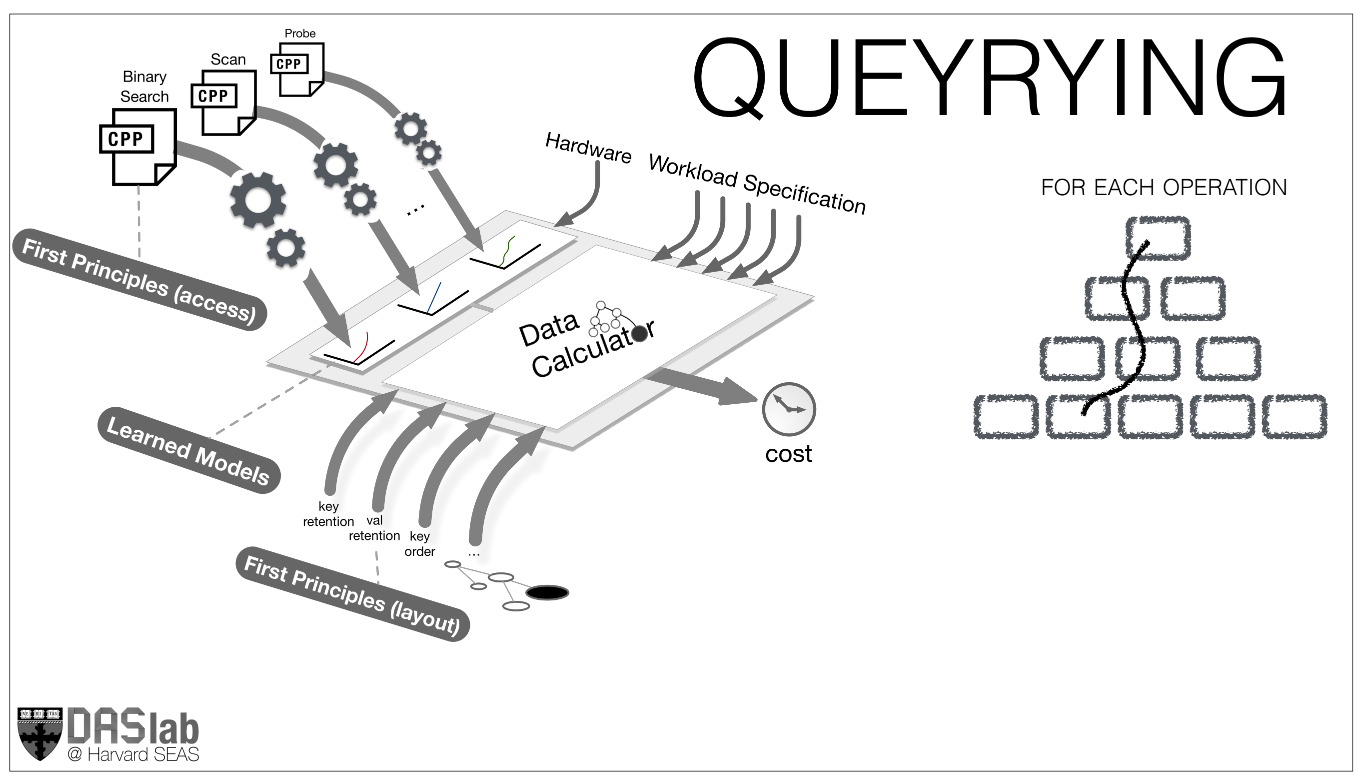

Overall the calculator first needs some time to train its models on the target hardware: this takes about 1 minute. Then it can take as input the layout of a target data structure and a workload (data and queries) and it computes the cost to run those queries over this data structure. It does this by constructing the state of the data structure at each step of the workload, computing the algorithm for each operation, and the path that the algorithm should follow over the data structure. It then assembles the cost using its models for each algorithmic component.



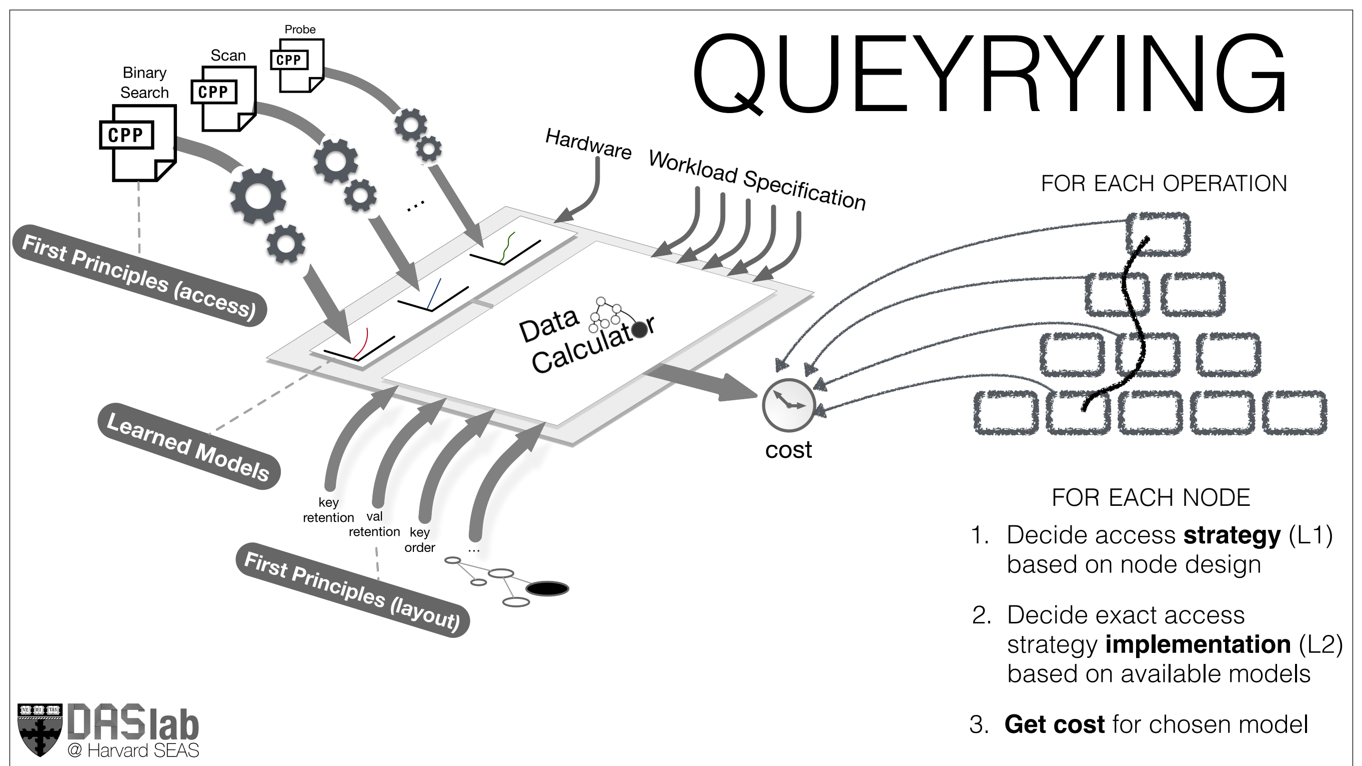

Overall the calculator first needs some time to train its models on the target hardware: this takes about 1 minute. Then it can take as input the layout of a target data structure and a workload (data and queries) and it computes the cost to run those queries over this data structure. It does this by constructing the state of the data structure at each step of the workload, computing the algorithm for each operation, and the path that the algorithm should follow over the data structure. It then assembles the cost using its models for each algorithmic component.



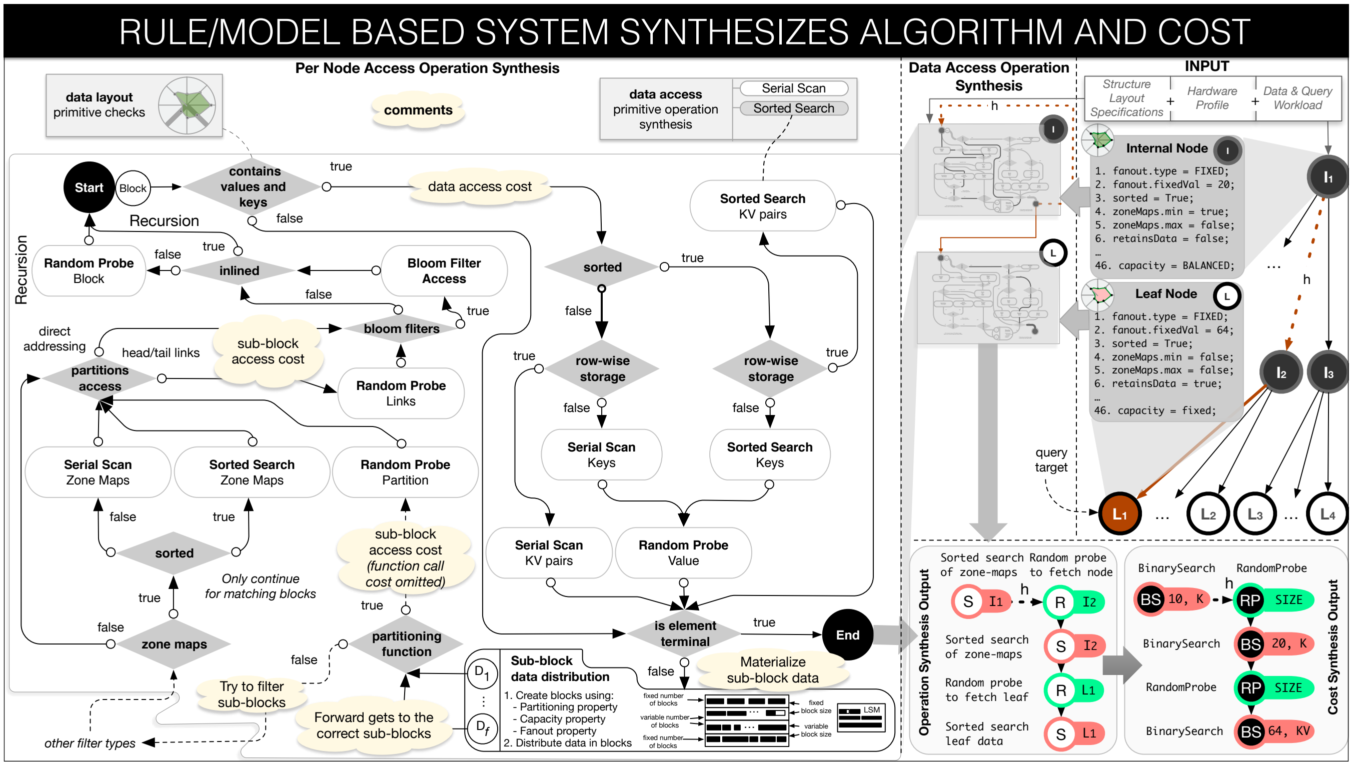

Overall the calculator first needs some time to train its models on the target hardware: this takes about 1 minute. Then it can take as input the layout of a target data structure and a workload (data and queries) and it computes the cost to run those queries over this data structure. It does this by constructing the state of the data structure at each step of the workload, computing the algorithm for each operation, and the path that the algorithm should follow over the data structure. It then assembles the cost using its models for each algorithmic component.



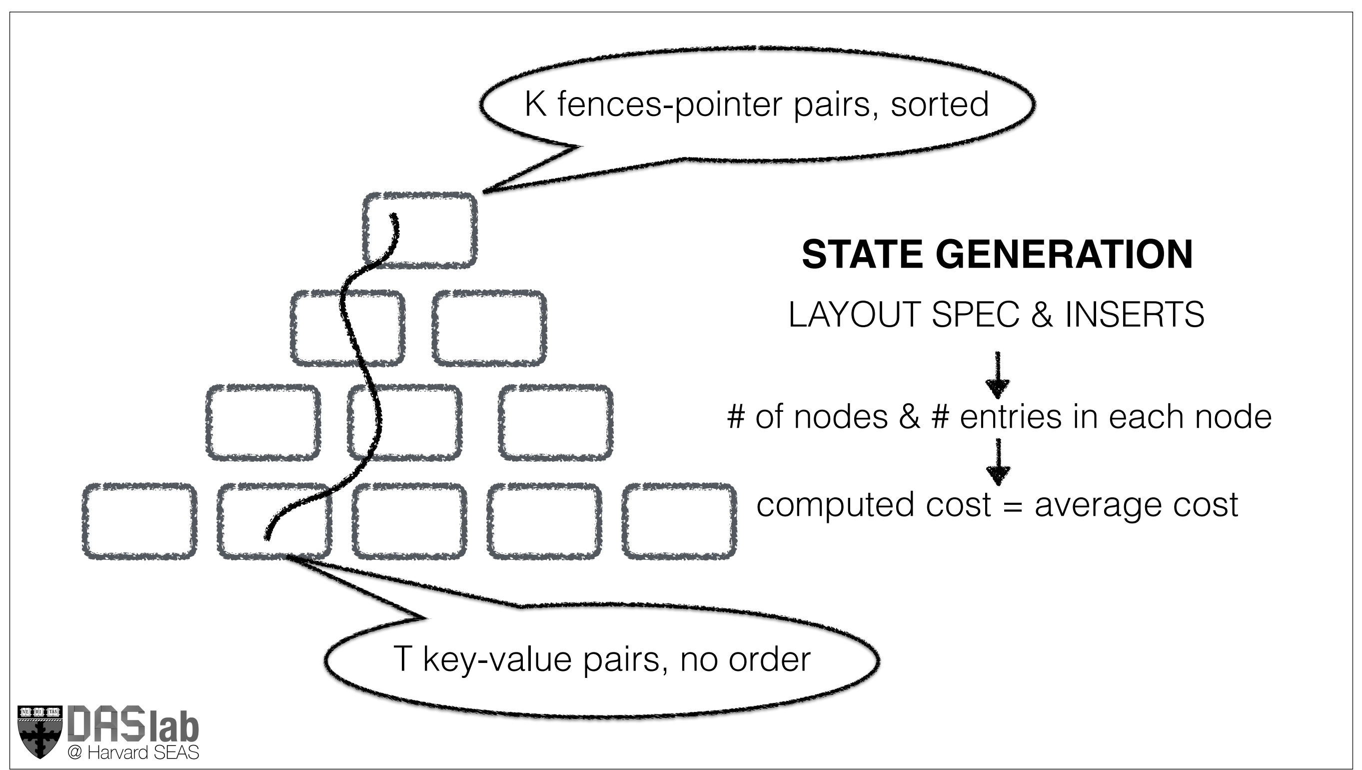

Overall the calculator first needs some time to train its models on the target hardware: this takes about 1 minute. Then it can take as input the layout of a target data structure and a workload (data and queries) and it computes the cost to run those queries over this data structure. It does this by constructing the state of the data structure at each step of the workload, computing the algorithm for each operation, and the path that the algorithm should follow over the data structure. It then assembles the cost using its models for each algorithmic component.



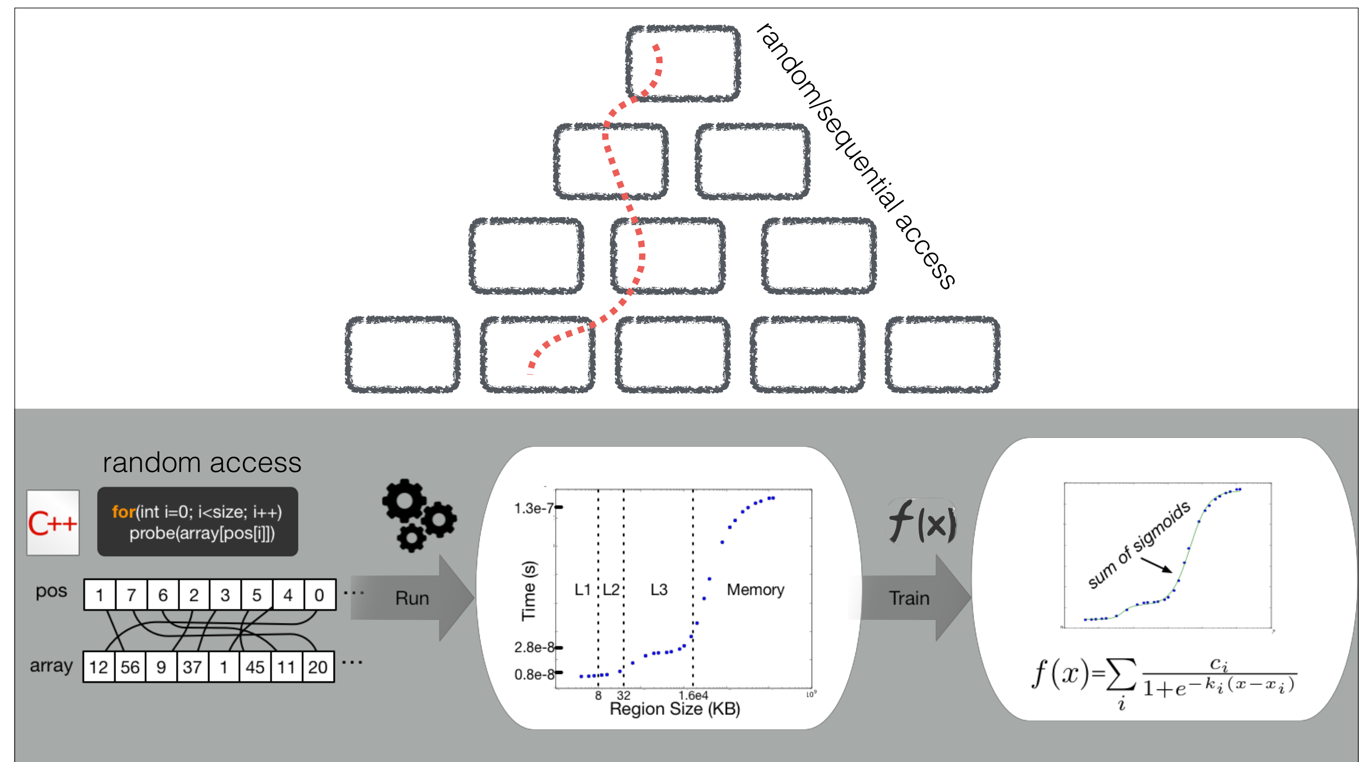

Overall the calculator first needs some time to train its models on the target hardware: this takes about 1 minute. Then it can take as input the layout of a target data structure and a workload (data and queries) and it computes the cost to run those queries over this data structure. It does this by constructing the state of the data structure at each step of the workload, computing the algorithm for each operation, and the path that the algorithm should follow over the data structure. It then assembles the cost using its models for each algorithmic component.



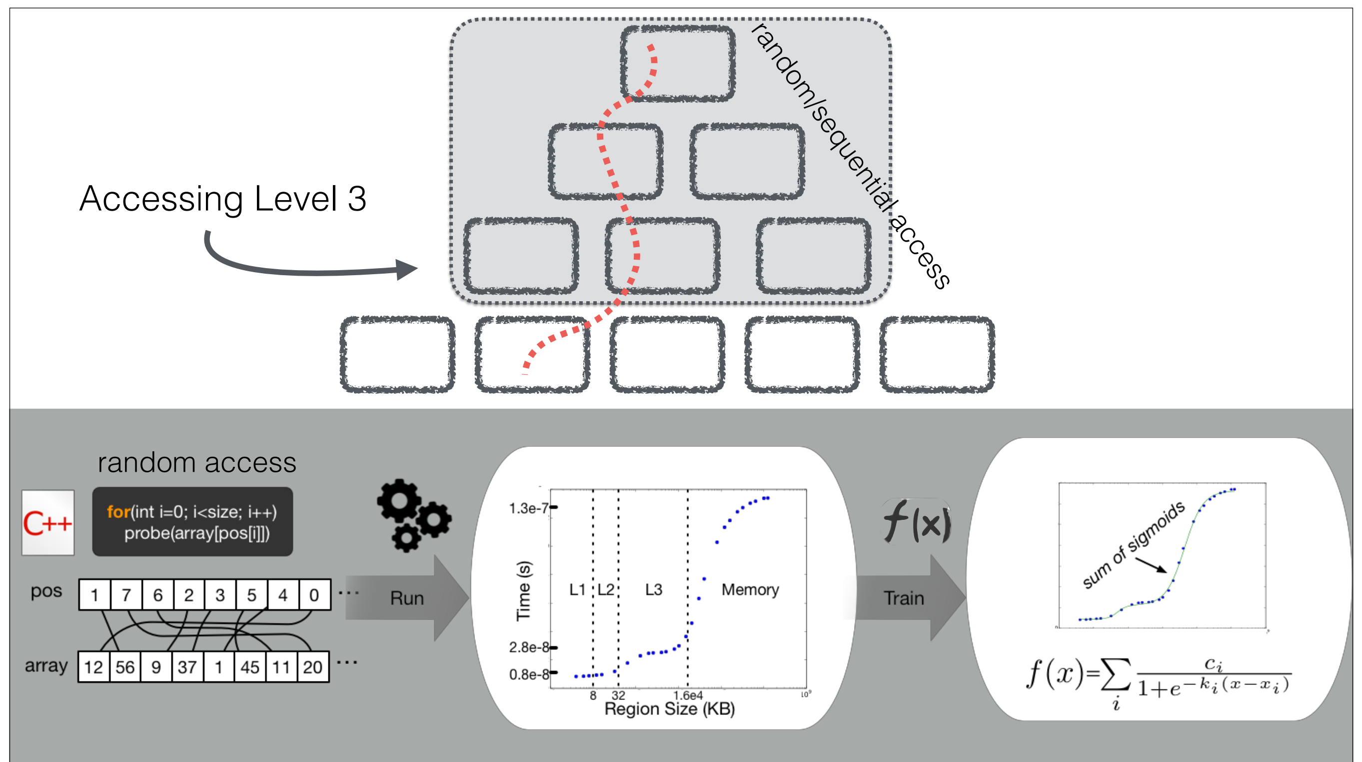

This is the high level rule-based system the Calculator uses to synthesize algorithms given a data layout. It reads from right to left. Given a high level description of the layout of a data structure, it uses the description of every node and given the path an operation needs to follow it constructs how the node should be accessed. Once it knows that, then it asks the learned models for the best algorithm and implementation to materialize the target algorithm given the exact state of the node (data size).



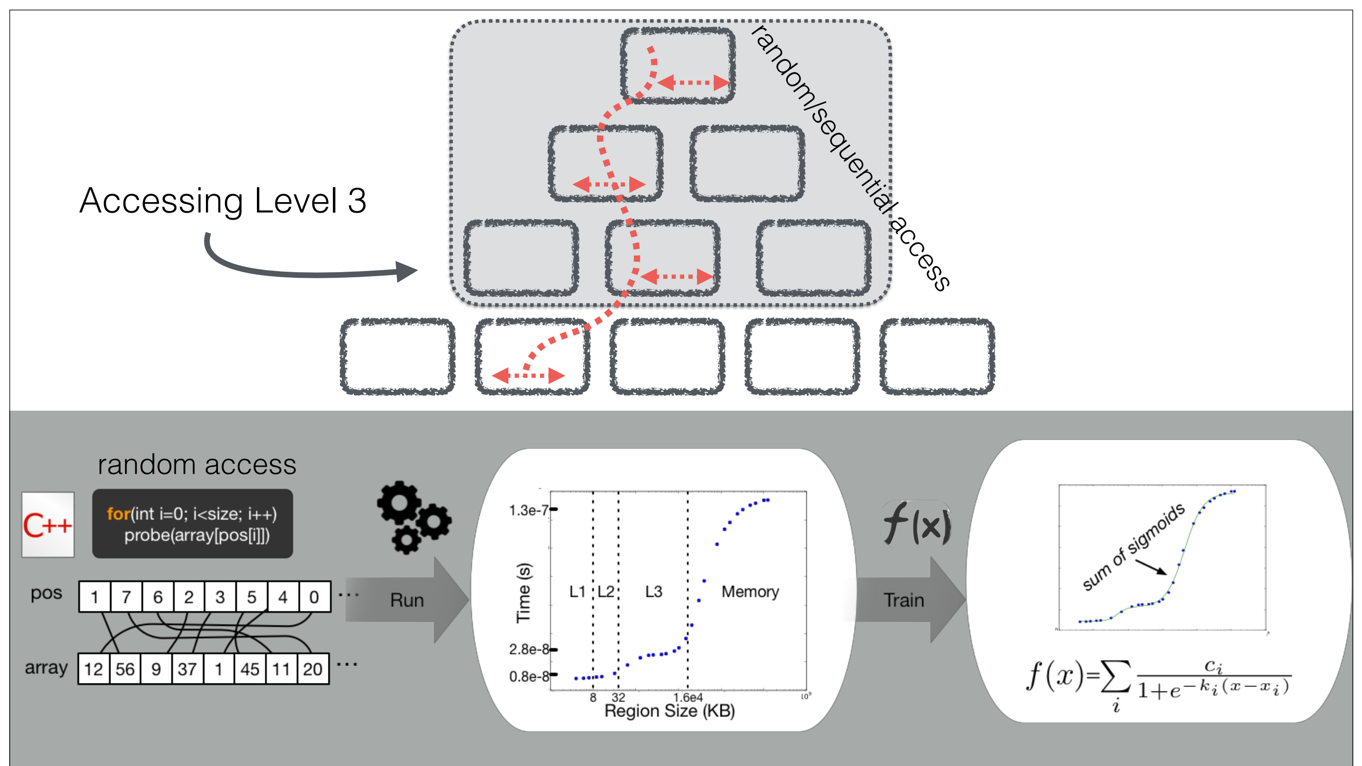

The only state needed is the number of entries in a node and the layout spec of the node.




There are several challenging issues. One example is how to properly account for caching effects. In this case we use a random access learned model which is trained for the target hardware but it is also weighted in terms of data size every time we invoke it. For example, if we are going to do a random access to go from the second level of a tree-based data structure to the third level, we can consider this as a random access within a flat array that includes all nodes of the first three levels of the tree. This implies that accessing deeper levels in the tree correctly gets more expensive (as shown in the model on the slide) as higher levels of the tree are naturally going to be cached with the higher probability.



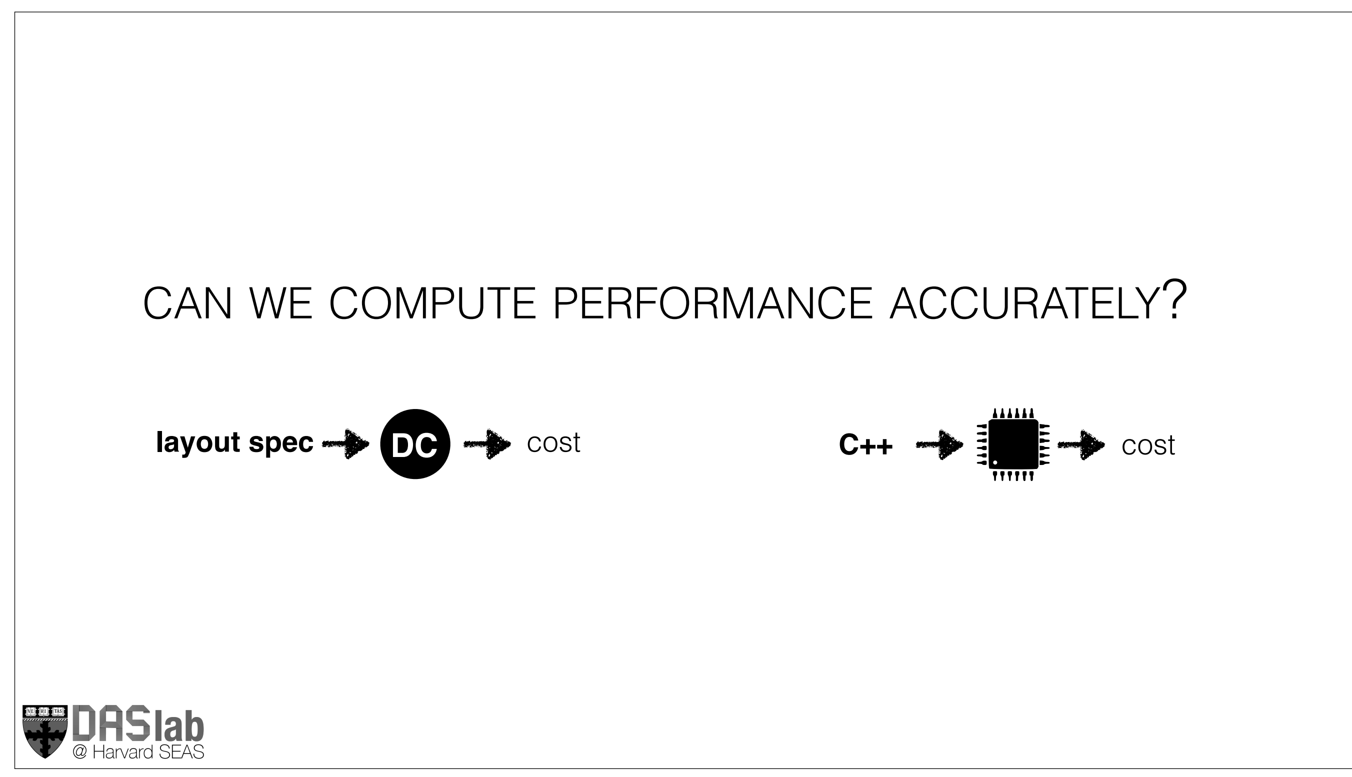

There are several challenging issues. One example is how to properly account for caching effects. In this case we use a random access learned model which is trained for the target hardware but it is also weighted in terms of data size every time we invoke it. For example, if we are going to do a random access to go from the second level of a tree-based data structure to the third level, we can consider this as a random access within a flat array that includes all nodes of the first three levels of the tree. This implies that accessing deeper levels in the tree correctly gets more expensive (as shown in the model on the slide) as higher levels of the tree are naturally going to be cached with the higher probability.




There are several challenging issues. One example is how to properly account for caching effects. In this case we use a random access learned model which is trained for the target hardware but it is also weighted in terms of data size every time we invoke it. For example, if we are going to do a random access to go from the second level of a tree-based data structure to the third level, we can consider this as a random access within a flat array that includes all nodes of the first three levels of the tree. This implies that accessing deeper levels in the tree correctly gets more expensive (as shown in the model on the slide) as higher levels of the tree are naturally going to be cached with the higher probability.

CAN WE COMPUTE PERFORMANCE ACCURATELY?

The primary metric for the calculator is if it can compute cost accurately.




The primary metric for the calculator is if it can compute cost accurately.

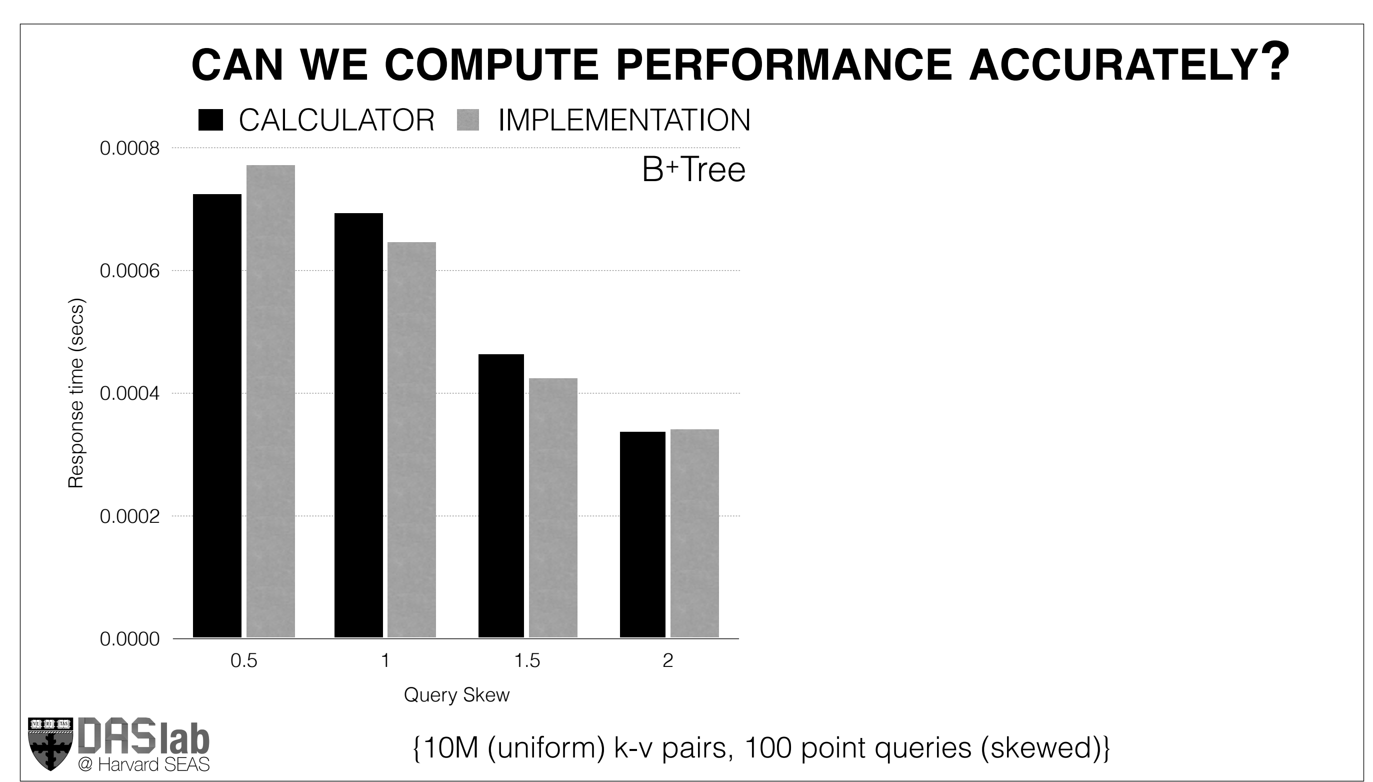
CAN WE COMPUTE PERFORMANCE ACCURATELY?

**layout spec** → DC → cost VS **C++** → [chip] → cost

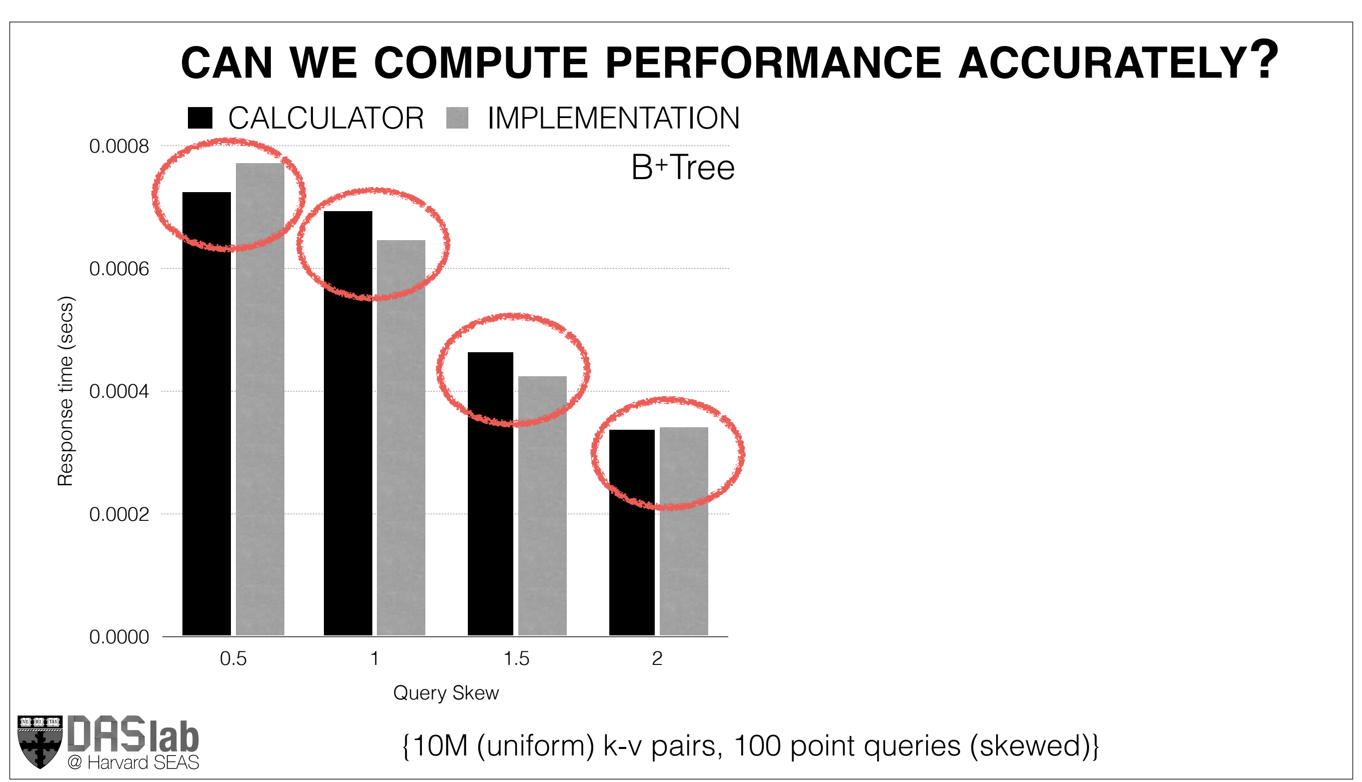
(same workload, hardware, data)

DASlab @ Harvard SEAS

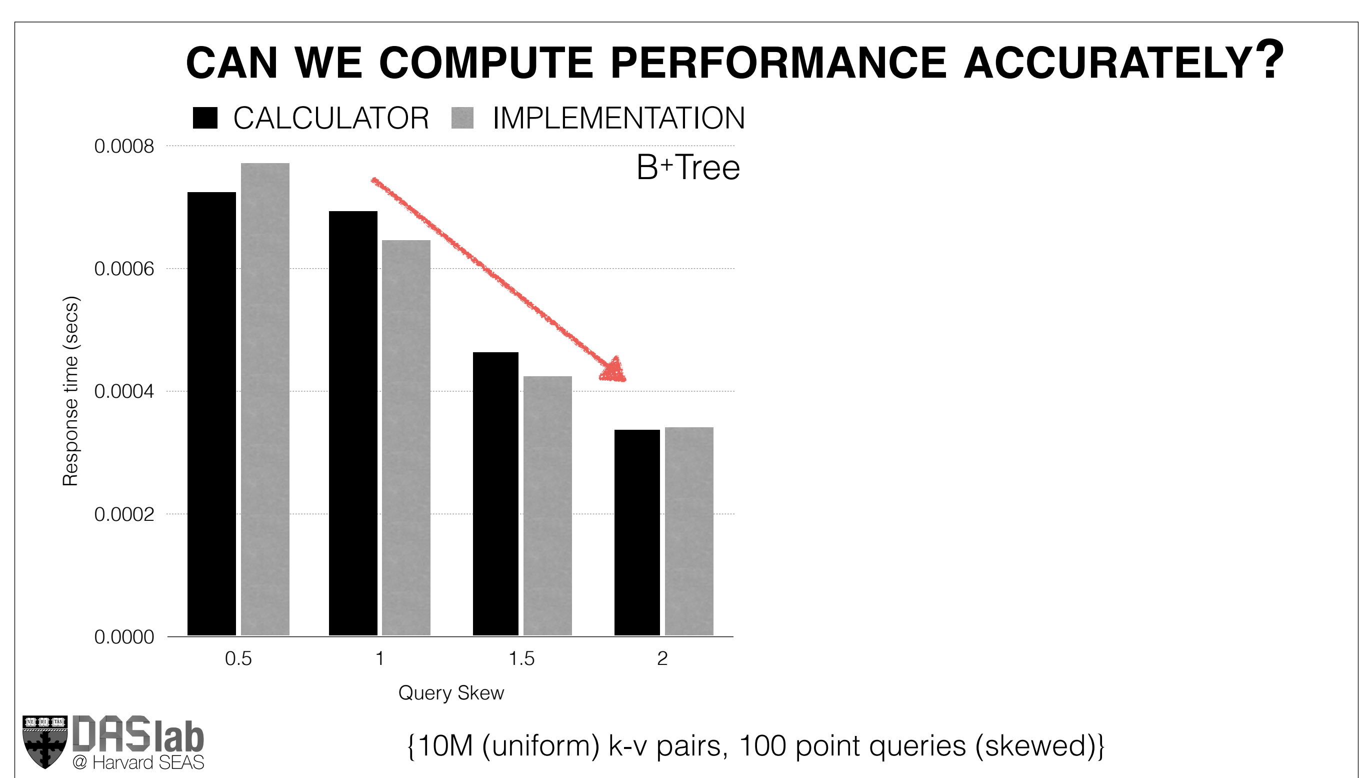
The primary metric for the calculator is if it can compute cost accurately.



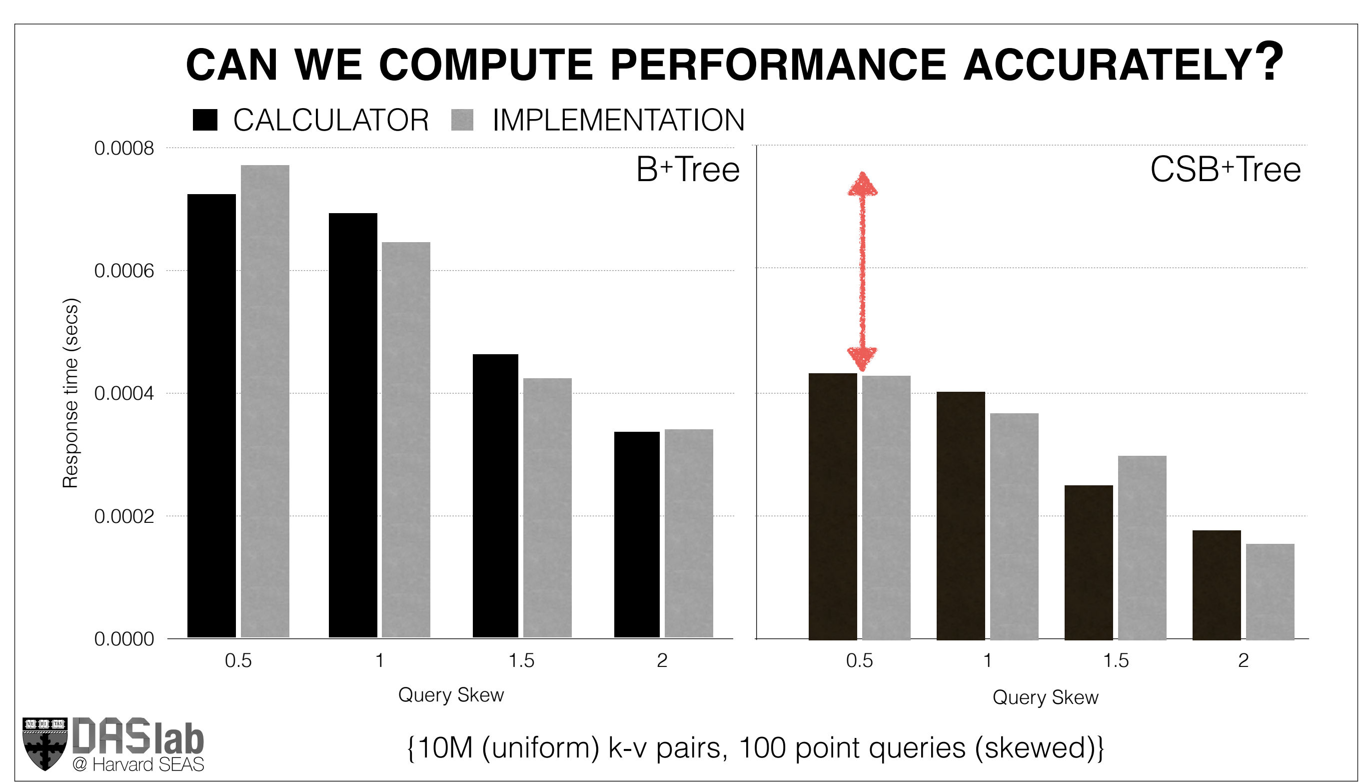

Here we compare the cost as it is computed by the Calculator against the actual cost observed when running a real implementation using the same data and queries and hardware as the input we give to the Calculator. In one case, we spend a couple of minutes describing a B-tree design in the Calculator syntax. In the other case, we spent weeks implementing the actual design. The Calculator gives accurate cost estimation even in the presence of query skew (which stresses caching effects). Similar results we observe when testing with cache conscious b-trees.



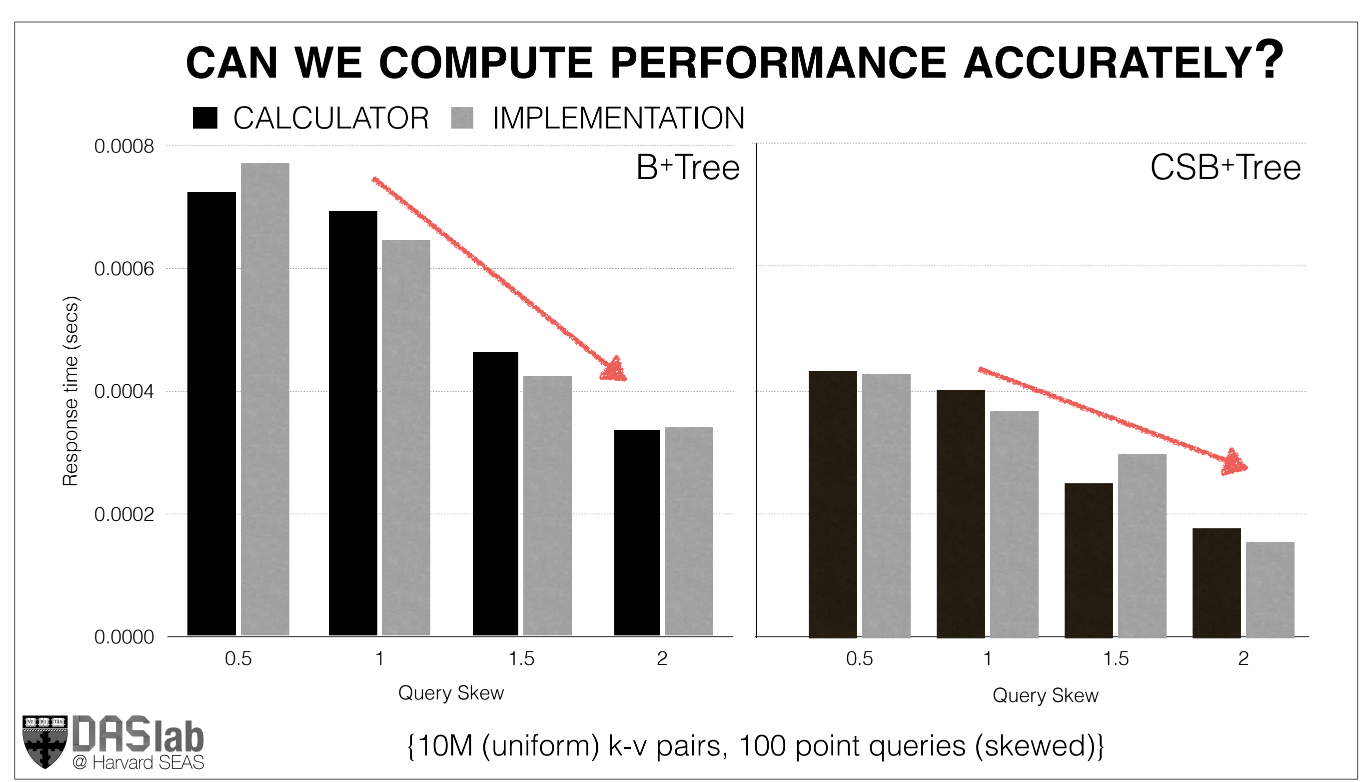

Here we compare the cost as it is computed by the Calculator against the actual cost observed when running a real implementation using the same data and queries and hardware as the input we give to the Calculator. In one case, we spend a couple of minutes describing a B-tree design in the Calculator syntax. In the other case, we spent weeks implementing the actual design. The Calculator gives accurate cost estimation even in the presence of query skew (which stresses caching effects). Similar results we observe when testing with cache conscious b-trees.



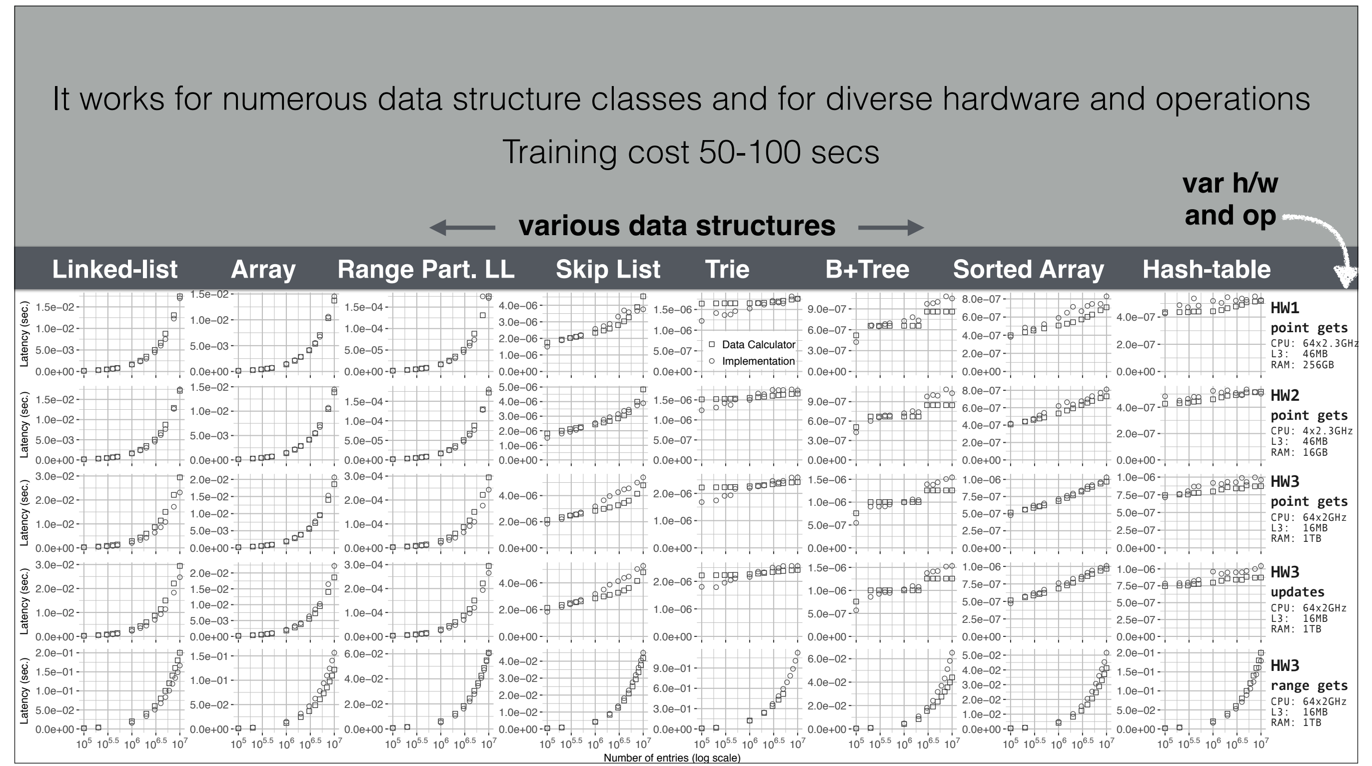

Here we compare the cost as it is computed by the Calculator against the actual cost observed when running a real implementation using the same data and queries and hardware as the input we give to the Calculator. In one case, we spend a couple of minutes describing a B-tree design in the Calculator syntax. In the other case, we spent weeks implementing the actual design. The Calculator gives accurate cost estimation even in the presence of query skew (which stresses caching effects). Similar results we observe when testing with cache conscious b-trees.



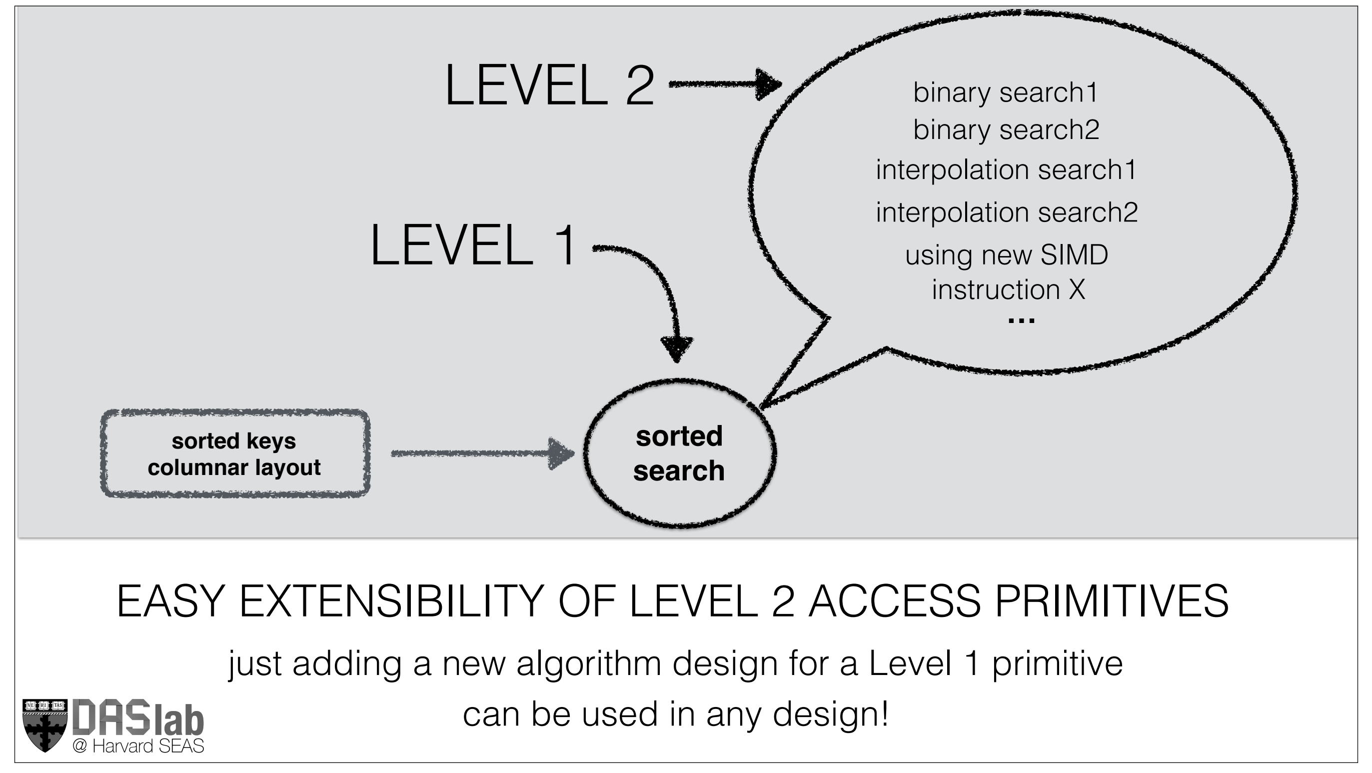

Here we compare the cost as it is computed by the Calculator against the actual cost observed when running a real implementation using the same data and queries and hardware as the input we give to the Calculator. In one case, we spend a couple of minutes describing a B-tree design in the Calculator syntax. In the other case, we spent weeks implementing the actual design. The Calculator gives accurate cost estimation even in the presence of query skew (which stresses caching effects). Similar results we observe when testing with cache conscious b-trees.




Here we compare the cost as it is computed by the Calculator against the actual cost observed when running a real implementation using the same data and queries and hardware as the input we give to the Calculator. In one case, we spend a couple of minutes describing a B-tree design in the Calculator syntax. In the other case, we spent weeks implementing the actual design. The Calculator gives accurate cost estimation even in the presence of query skew (which stresses caching effects). Similar results we observe when testing with cache conscious b-trees.

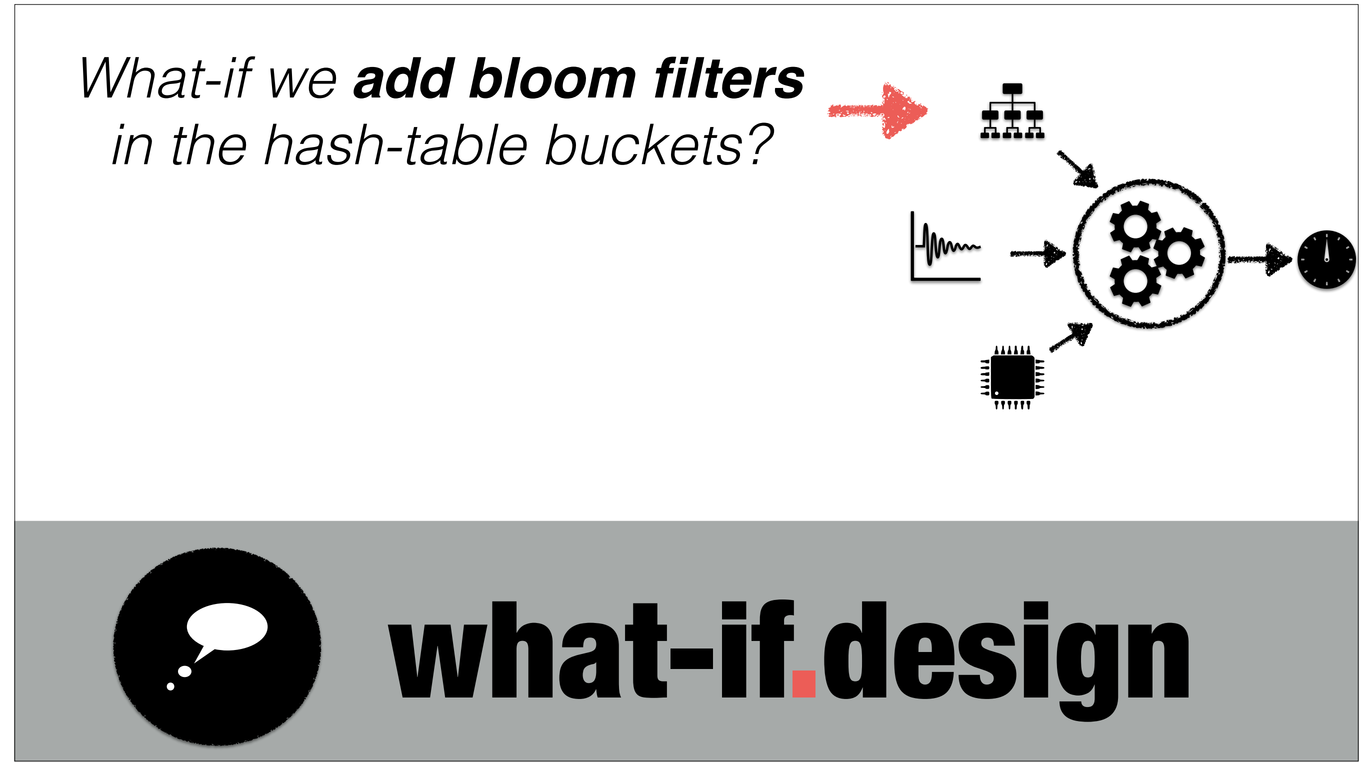



The Calculator has been tested successfully for a diverse set of designs.

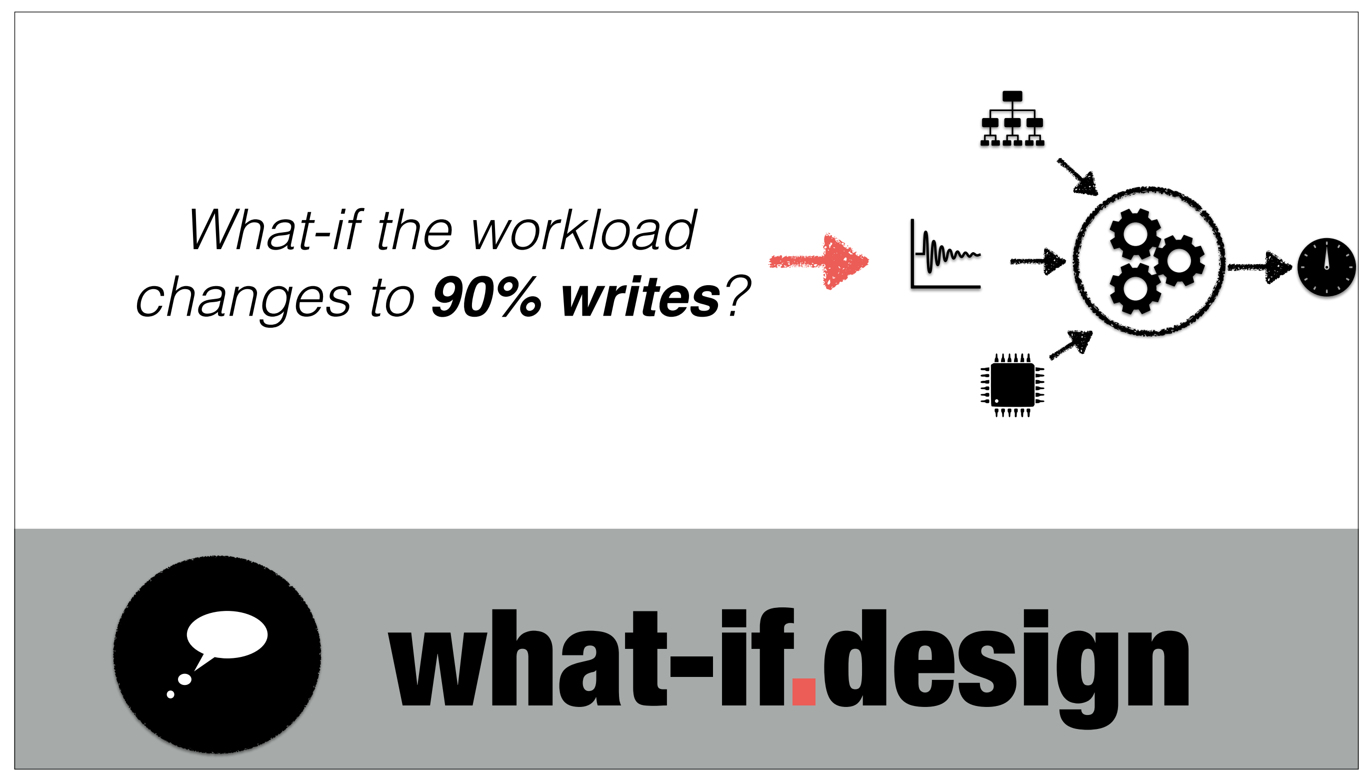



Another big advantage of synthesis is that it created natural scenarios where new ideas can cross-pollinate across many areas. For example, imagine someone inventing a new design or a new implementation for the binary search used in their B-tree -like design. Once this new implementation becomes part of the Calculator as a learned model, the Calculator will consider it for any data structure design where any node is sorted. This helps easily utilize new ideas at a low level across all possible designs.

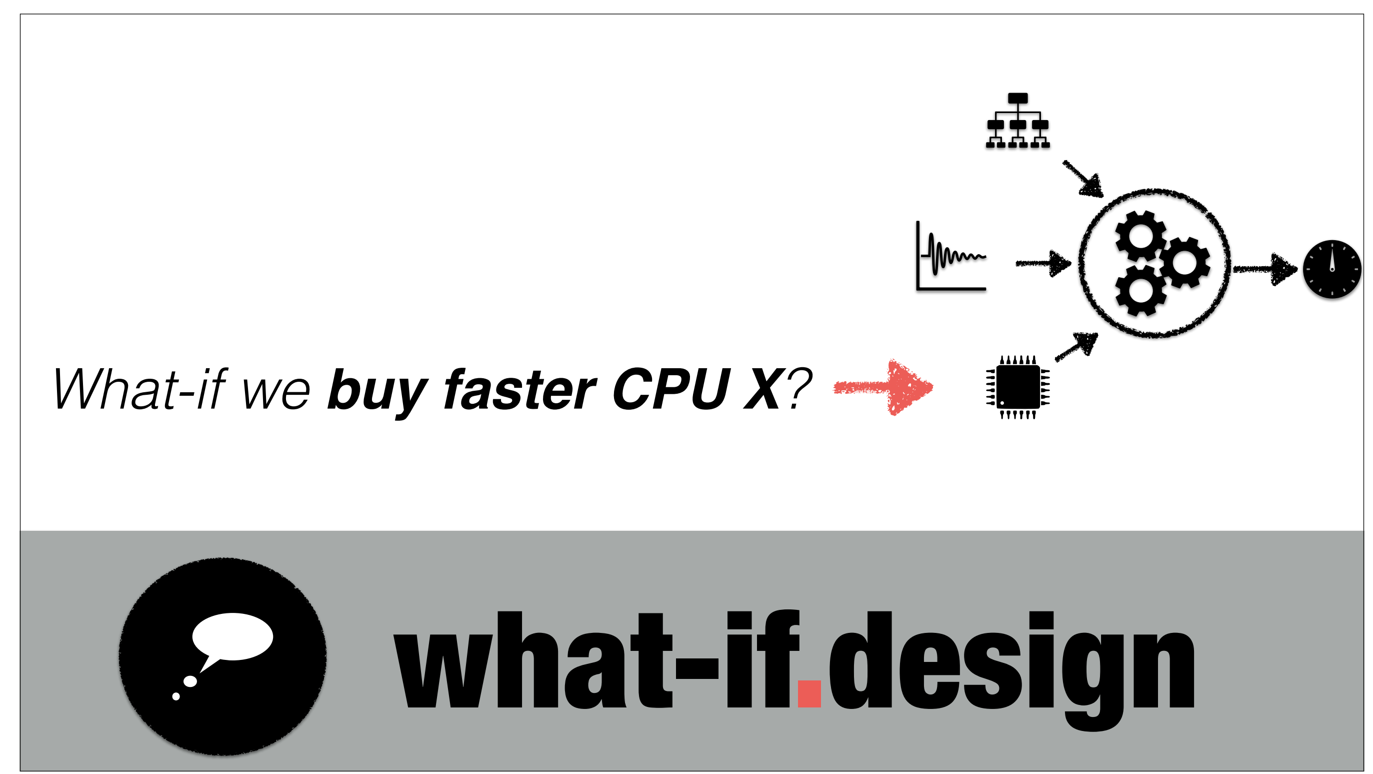
DESIGN SPACE
COST SYNTHESIS
HOW TO USE



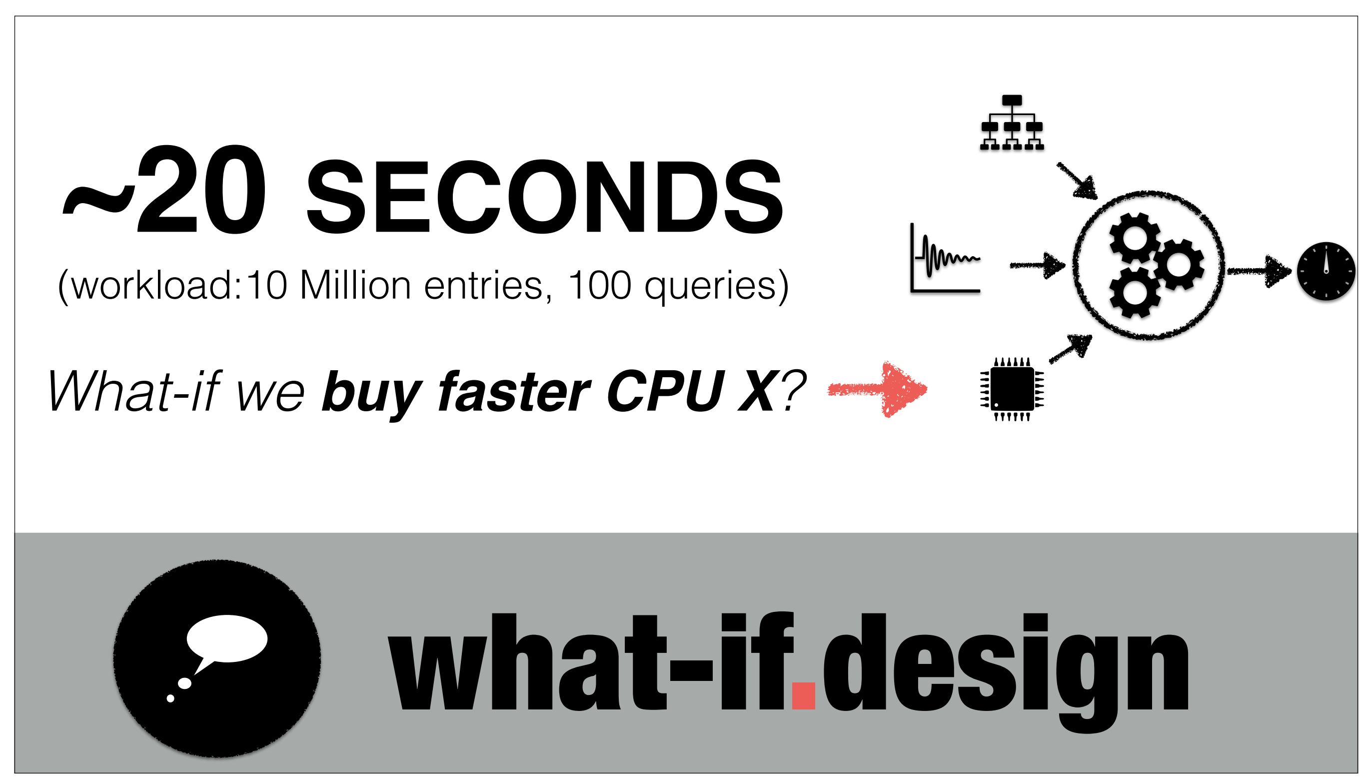

The first usage of the Calculator is as a what-if engine. One can simply query the Calculator twice by changing one of the inputs the second time. For example if we try out two different data structure layout designs while using the same workload and hardware we can see the effect of the design changes. In our current implementation what-if queries take in the order of 20 seconds for a dataset of 10 million entries. The competition is fully implementing and testing the target design for several weeks…



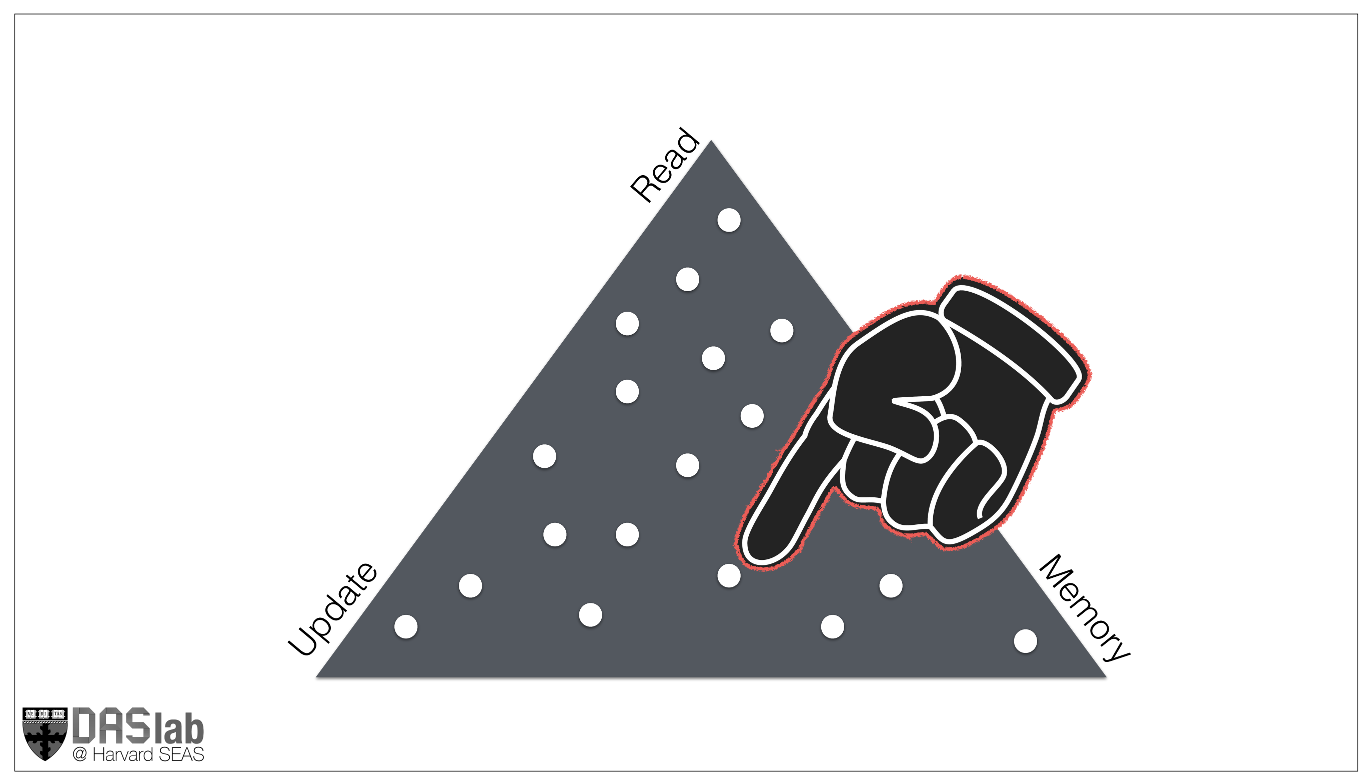

The first usage of the Calculator is as a what-if engine. One can simply query the Calculator twice by changing one of the inputs the second time. For example if we try out two different data structure layout designs while using the same workload and hardware we can see the effect of the design changes. In our current implementation what-if queries take in the order of 20 seconds for a dataset of 10 million entries. The competition is fully implementing and testing the target design for several weeks…




The first usage of the Calculator is as a what-if engine. One can simply query the Calculator twice by changing one of the inputs the second time. For example if we try out two different data structure layout designs while using the same workload and hardware we can see the effect of the design changes. In our current implementation what-if queries take in the order of 20 seconds for a dataset of 10 million entries. The competition is fully implementing and testing the target design for several weeks…




The first usage of the Calculator is as a what-if engine. One can simply query the Calculator twice by changing one of the inputs the second time. For example if we try out two different data structure layout designs while using the same workload and hardware we can see the effect of the design changes. In our current implementation what-if queries take in the order of 20 seconds for a dataset of 10 million entries. The competition is fully implementing and testing the target design for several weeks…

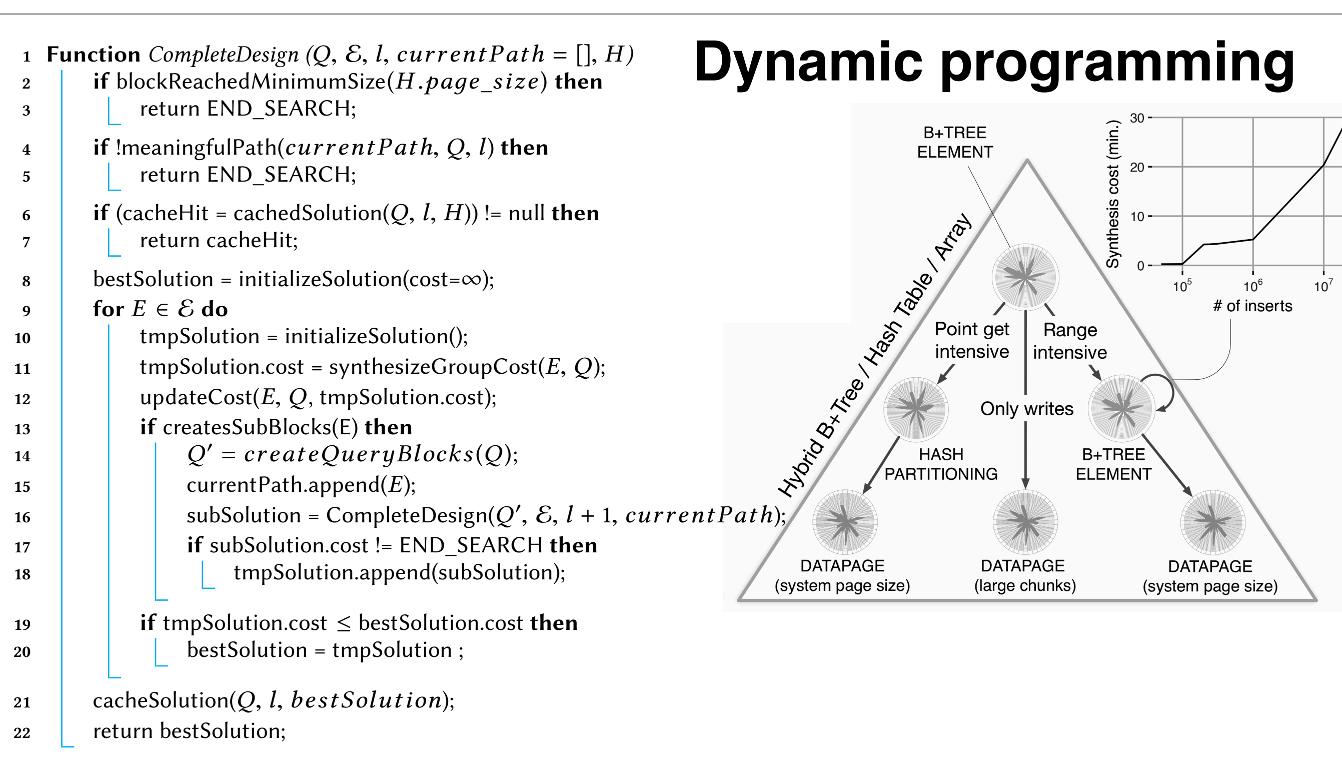



Our end goal was about automated design…

# Dynamic programming

```
1  Function CompleteDesign (Q, ℰ, l, currentPath = [], H)
2      if blockReachedMinimumSize(H.page_size) then
3          return END_SEARCH;
4      if !meaningfulPath(currentPath, Q, l) then
5          return END_SEARCH;
6      if (cacheHit = cachedSolution(Q, l, H)) != null then
7          return cacheHit;
8      bestSolution = initializeSolution(cost=∞);
9      for E ∈ ℰ do
10         tmpSolution = initializeSolution();
11         tmpSolution.cost = synthesizeGroupCost(E, Q);
12         updateCost(E, Q, tmpSolution.cost);
13         if createsSubBlocks(E) then
14             Q′ = createQueryBlocks(Q);
15             currentPath.append(E);
16             subSolution = CompleteDesign(Q′, ℰ, l + 1, currentPath);
17             if subSolution.cost != END_SEARCH then
18                 tmpSolution.append(subSolution);
19         if tmpSolution.cost ≤ bestSolution.cost then
20             bestSolution = tmpSolution ;
21     cacheSolution(Q, l, bestSolution);
22     return bestSolution;
```

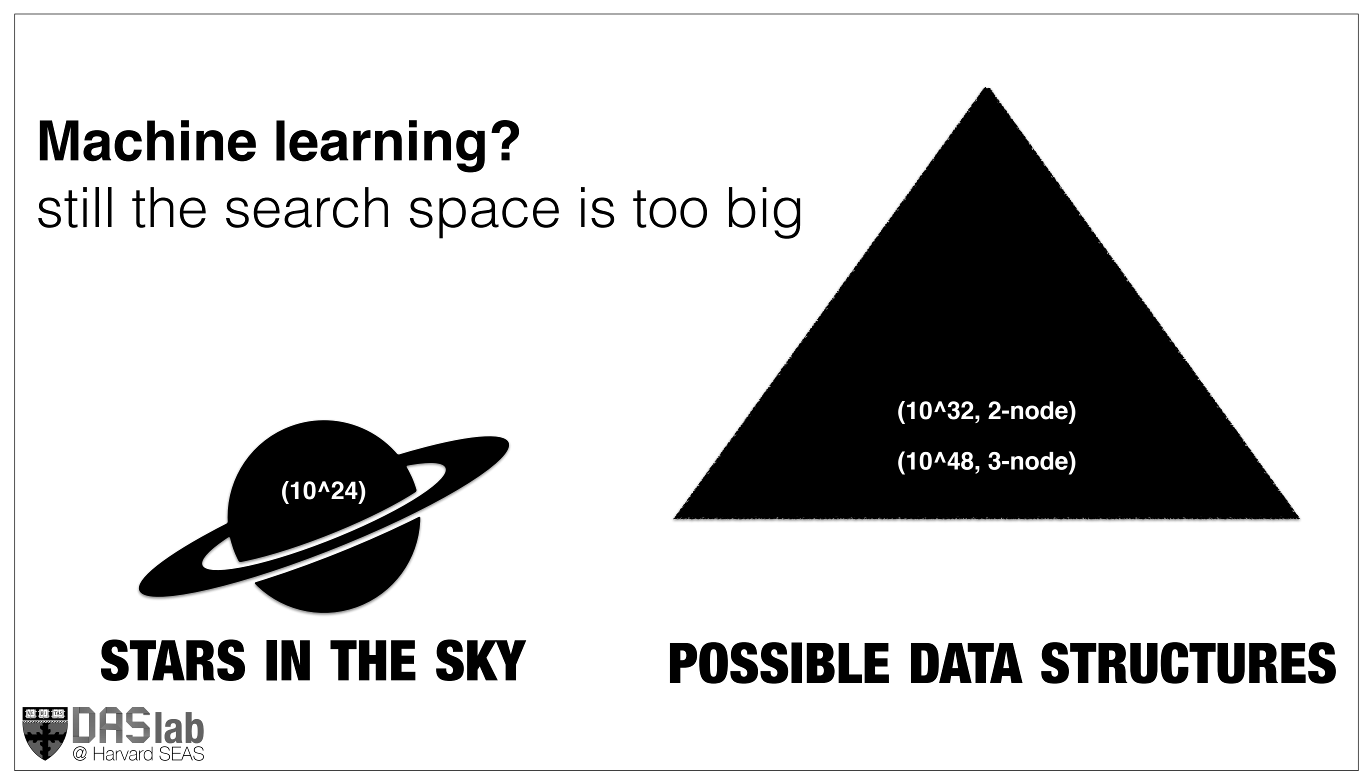
We can use the Calculator to build auto-design algorithms. E.g., algorithms that use its costing capabilities to iterate over multiple designs. In a recent example, we showed that using dynamic programming the calculator can design completely new data structures that are optimal for a given workloads and hardware within 30 minutes.



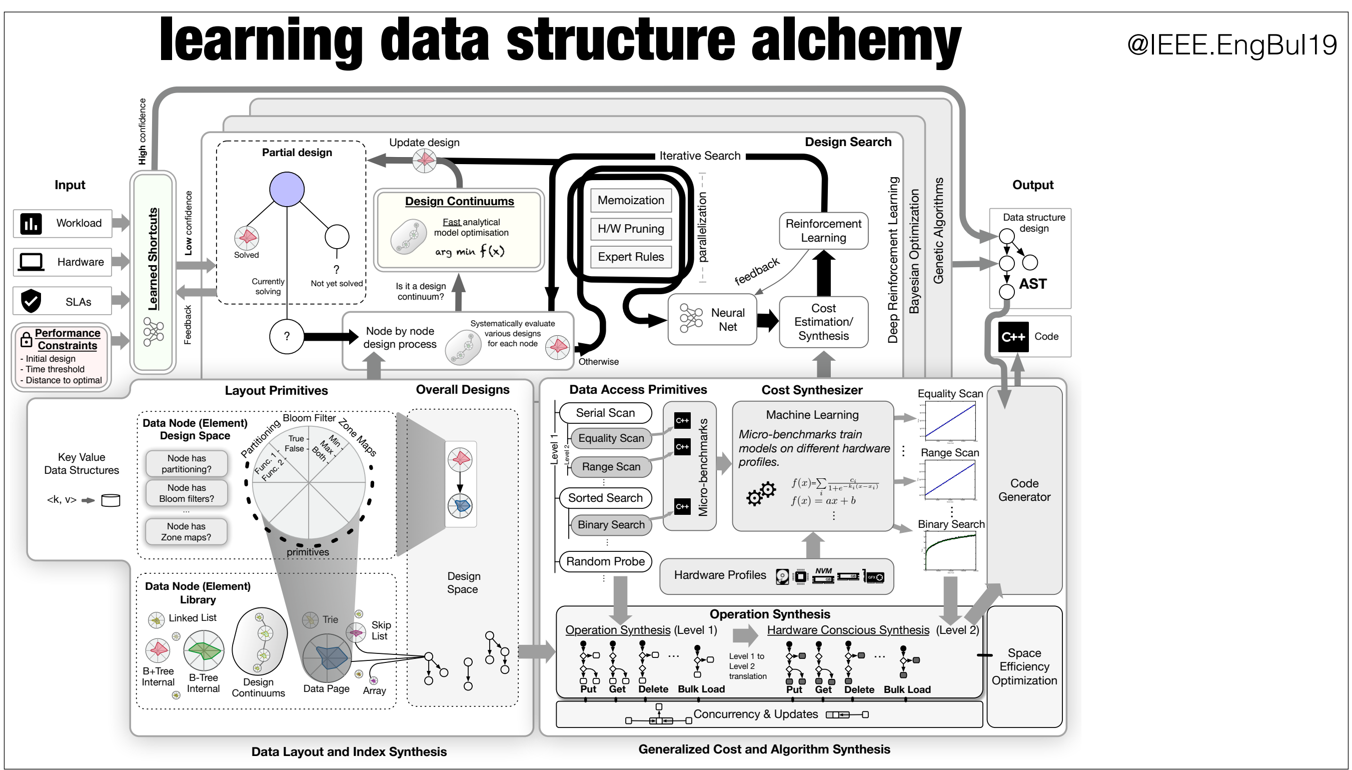

We can use the Calculator to build auto-design algorithms. E.g., algorithms that use its costing capabilities to iterate over multiple designs. In a recent example, we showed that using dynamic programming the calculator can design completely new data structures that are optimal for a given workloads and hardware within 30 minutes.

**Machine learning?**
still the search space is too big

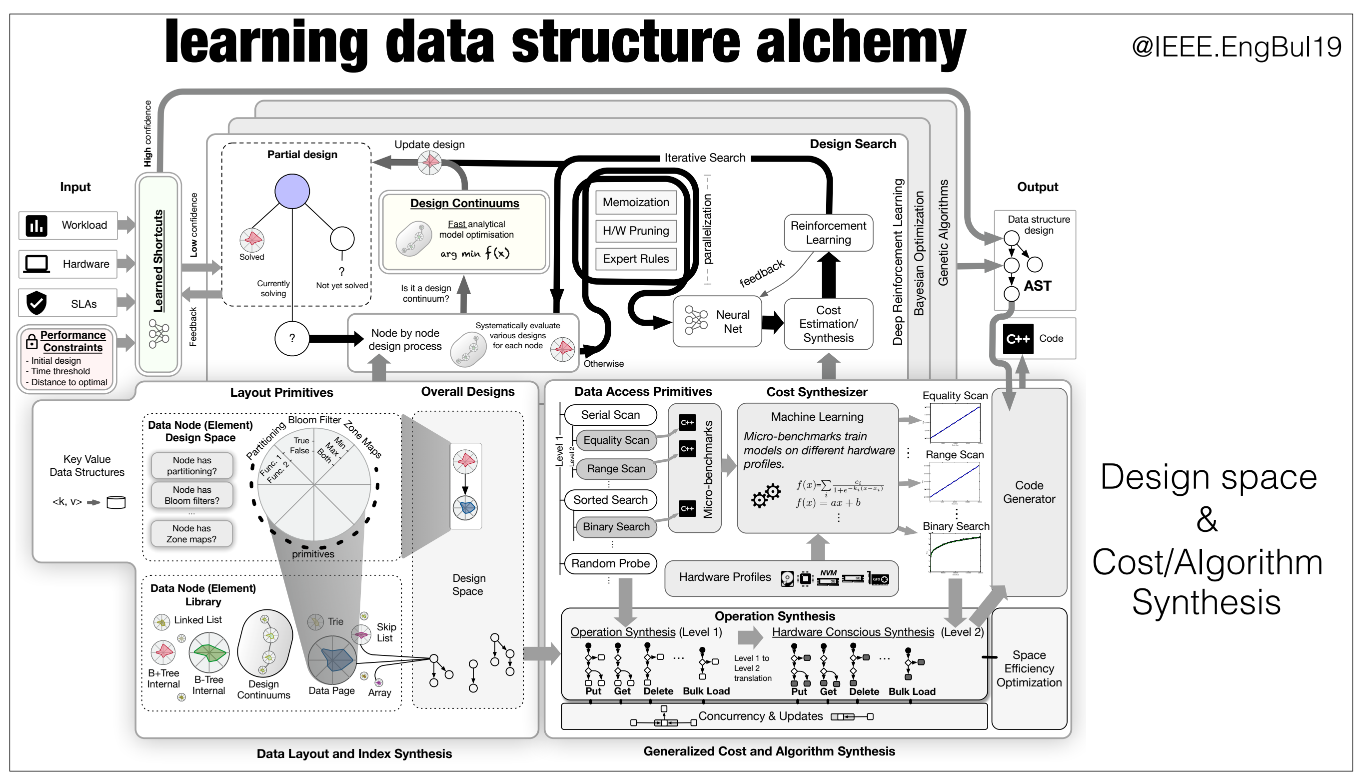
To make dynamic programming work we had to reduce the design space the algorithm searched. While this leads to very useful results, the end goal is to be able to search the whole possible design space. The next natural question is whether we can ML -inspired algorithms.



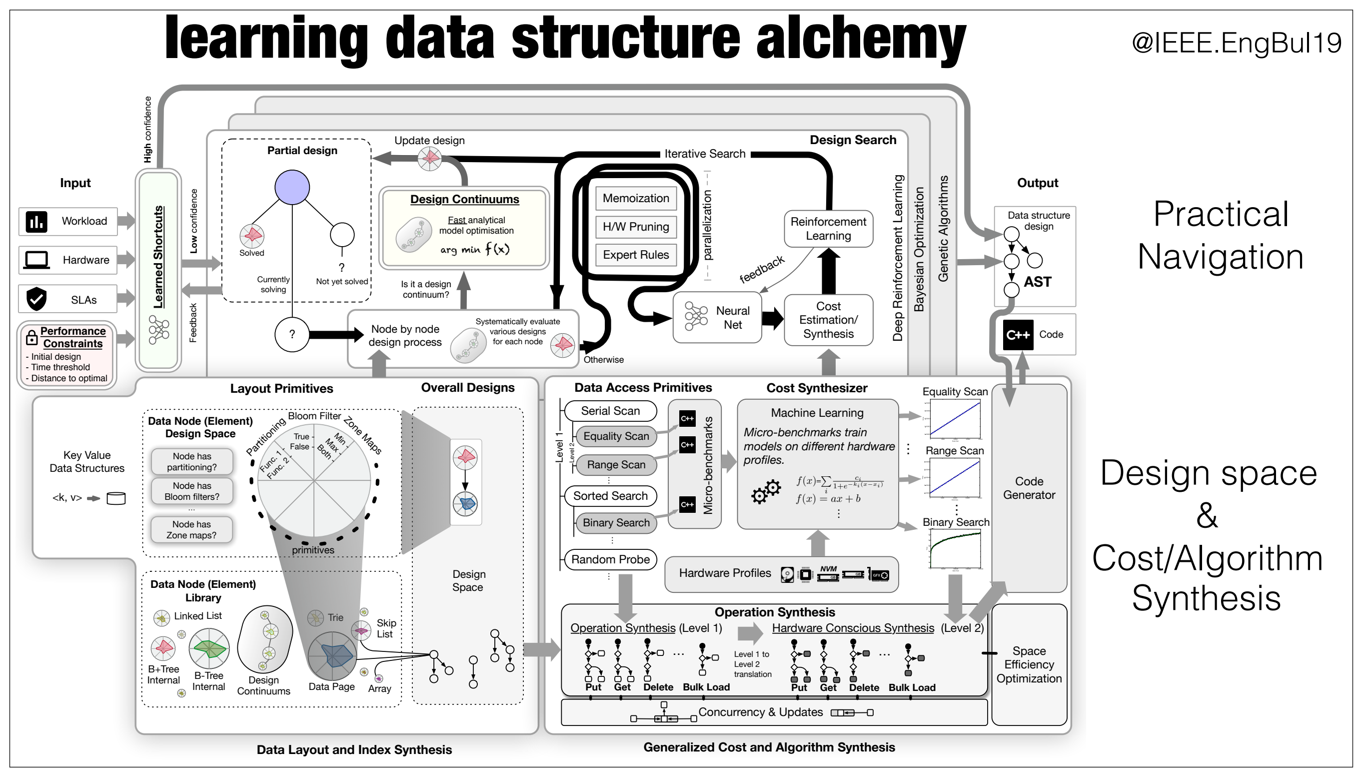

To make dynamic programming work we had to reduce the design space the algorithm searched. While this leads to very useful results, the end goal is to be able to search the whole possible design space. The next natural question is whether we can ML -inspired algorithms.



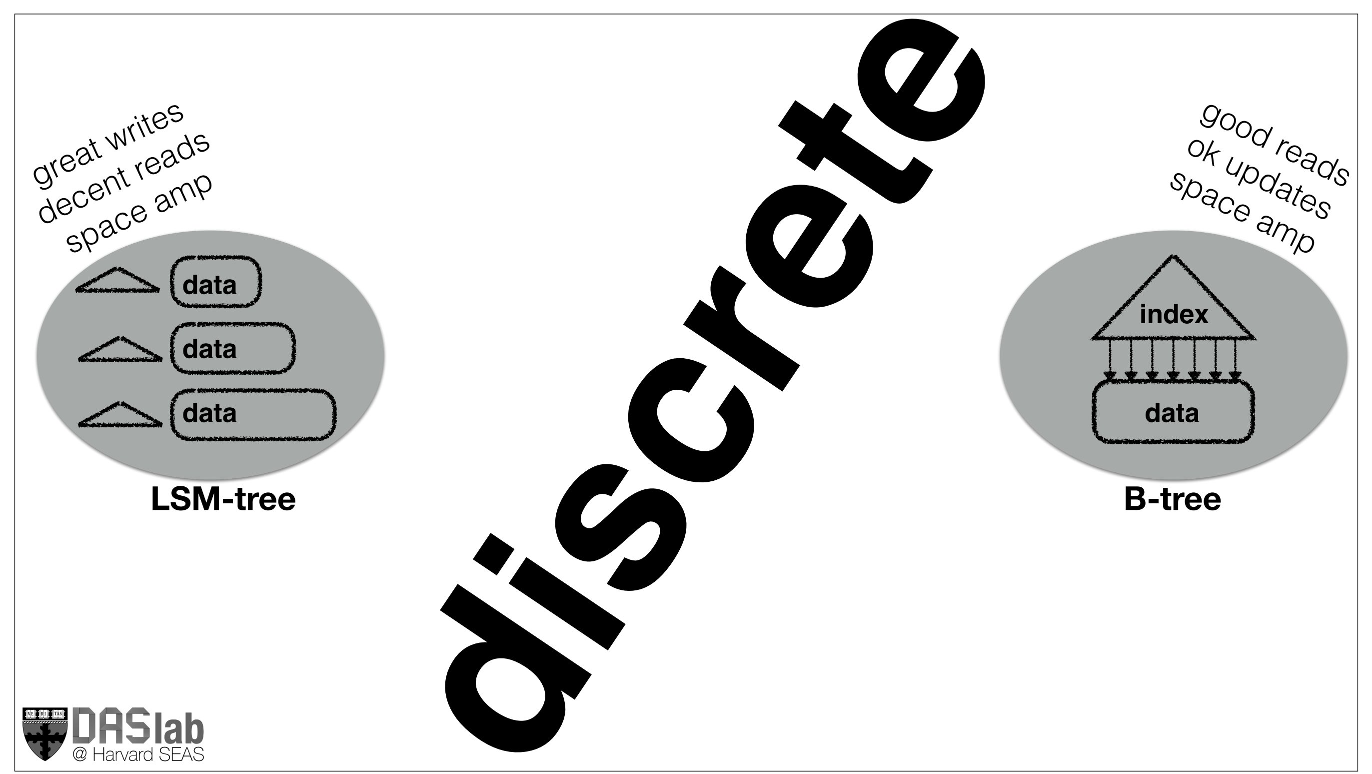

Our recent work on data structure alchemy shows the roadmap for ML search algorithms that automatically design data structures over the massive possible design space. While the mapping of the search problem itself to individual ML -based algorithms is an exciting problem, the true challenge is the ability to restrict the design space automatically so we can make it practical for the algorithms to run. Recall that the useful space is larger than 10 to the 100. The figure in this slide shows the initial architecture of a system we are building called the Data Alchemist which describes how to create shortcuts in many steps of the path towards auto-design. Examples include the usage of design continuums (see respective CIDR19 paper) which allows part of the design space to actually be described by closed form models (and thus can be computed fast). Other examples, include termination triggers that have to do with hardware restrictions, i.e., if we reach latency or throughput very close to the ideal achieved by the target hardware we can stop even if a better design exists. Other considerations regarding the ideal design include the complexity of the design or how far it is from an existing design that we would be migrating from, etc.



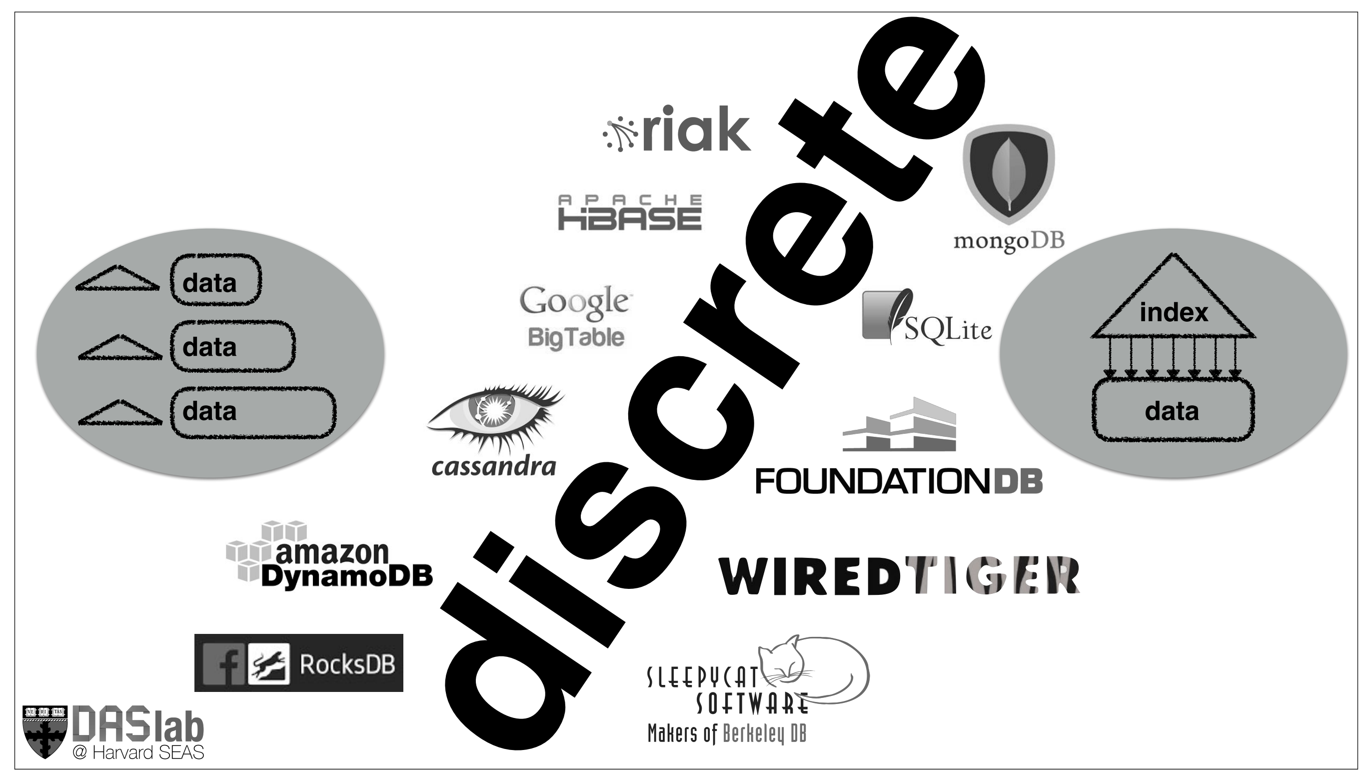

Our recent work on data structure alchemy shows the roadmap for ML search algorithms that automatically design data structures over the massive possible design space. While the mapping of the search problem itself to individual ML -based algorithms is an exciting problem, the true challenge is the ability to restrict the design space automatically so we can make it practical for the algorithms to run. Recall that the useful space is larger than 10 to the 100. The figure in this slide shows the initial architecture of a system we are building called the Data Alchemist which describes how to create shortcuts in many steps of the path towards auto-design. Examples include the usage of design continuums (see respective CIDR19 paper) which allows part of the design space to actually be described by closed form models (and thus can be computed fast). Other examples, include termination triggers that have to do with hardware restrictions, i.e., if we reach latency or throughput very close to the ideal achieved by the target hardware we can stop even if a better design exists. Other considerations regarding the ideal design include the complexity of the design or how far it is from an existing design that we would be migrating from, etc.



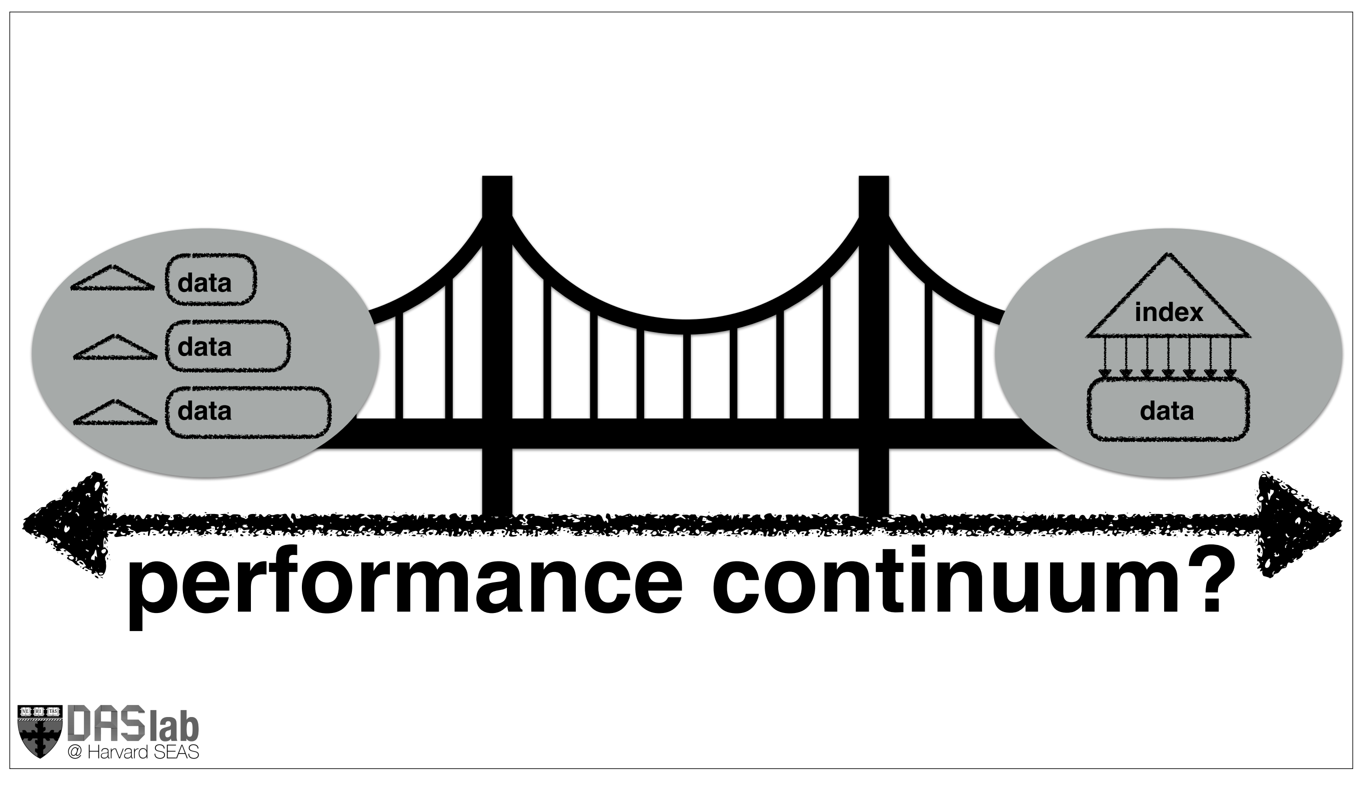

Our recent work on data structure alchemy shows the roadmap for ML search algorithms that automatically design data structures over the massive possible design space. While the mapping of the search problem itself to individual ML -based algorithms is an exciting problem, the true challenge is the ability to restrict the design space automatically so we can make it practical for the algorithms to run. Recall that the useful space is larger than 10 to the 100. The figure in this slide shows the initial architecture of a system we are building called the Data Alchemist which describes how to create shortcuts in many steps of the path towards auto-design. Examples include the usage of design continuums (see respective CIDR19 paper) which allows part of the design space to actually be described by closed form models (and thus can be computed fast). Other examples, include termination triggers that have to do with hardware restrictions, i.e., if we reach latency or throughput very close to the ideal achieved by the target hardware we can stop even if a better design exists. Other considerations regarding the ideal design include the complexity of the design or how far it is from an existing design that we would be migrating from, etc.



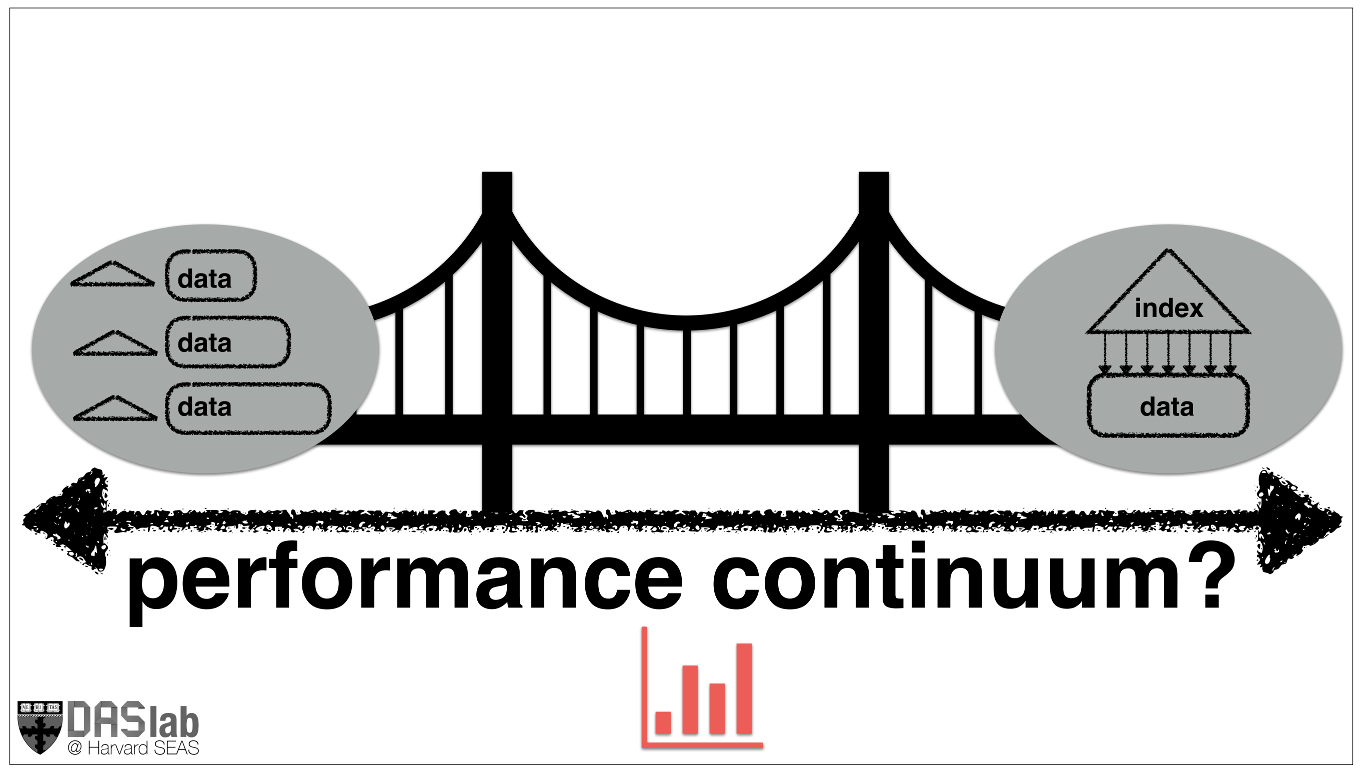

We are going to go into more detail on one of the ideas on how to quickly search the data structure design space for instance optimality. We call this “design continuums” and it draws motivation from the fact that we currently incorrectly perceive different data structures as fundamentally different. For example, take LSM-trees and B-trees: people think they are truly different and whole separate systems and industries have been built around each one of them.



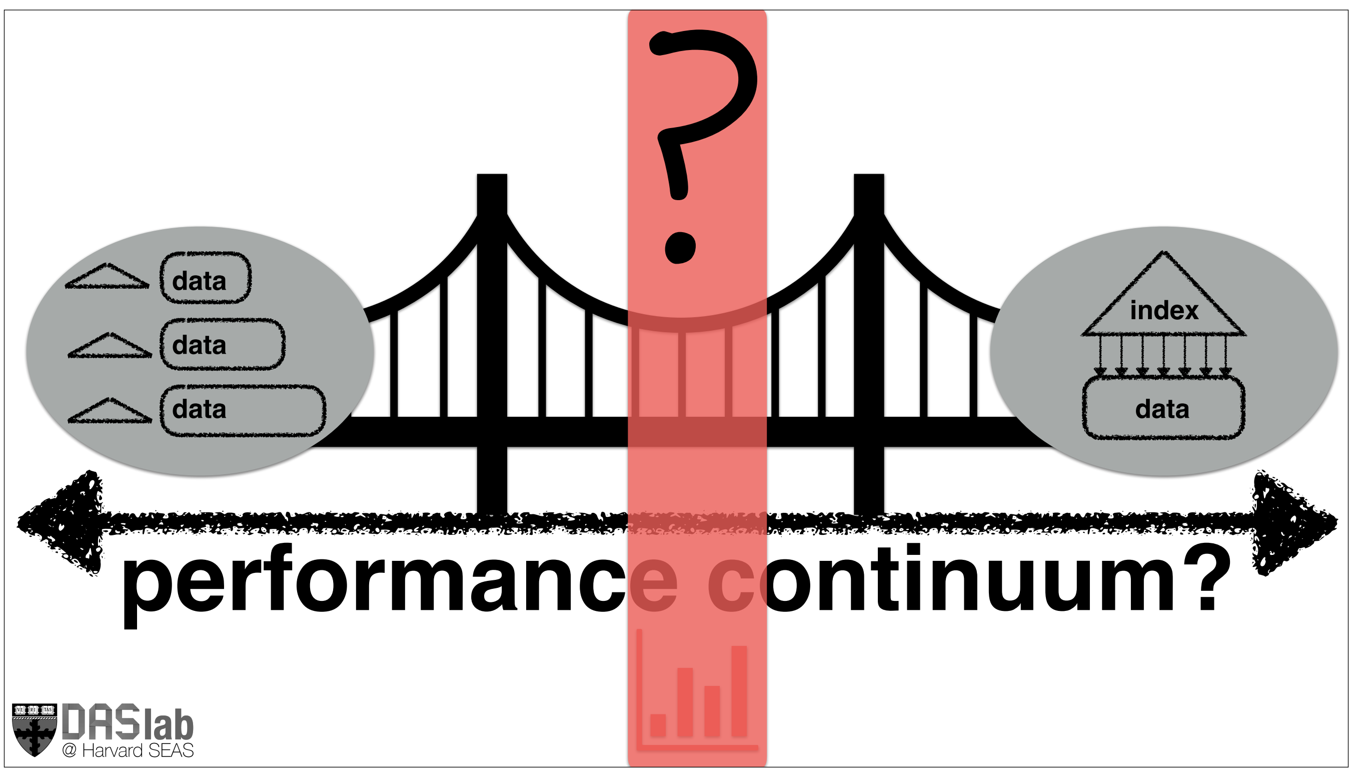

We are going to go into more detail on one of the ideas on how to quickly search the data structure design space for instance optimality. We call this “design continuums” and it draws motivation from the fact that we currently incorrectly perceive different data structures as fundamentally different. For example, take LSM-trees and B-trees: people think they are truly different and whole separate systems and industries have been built around each one of them.



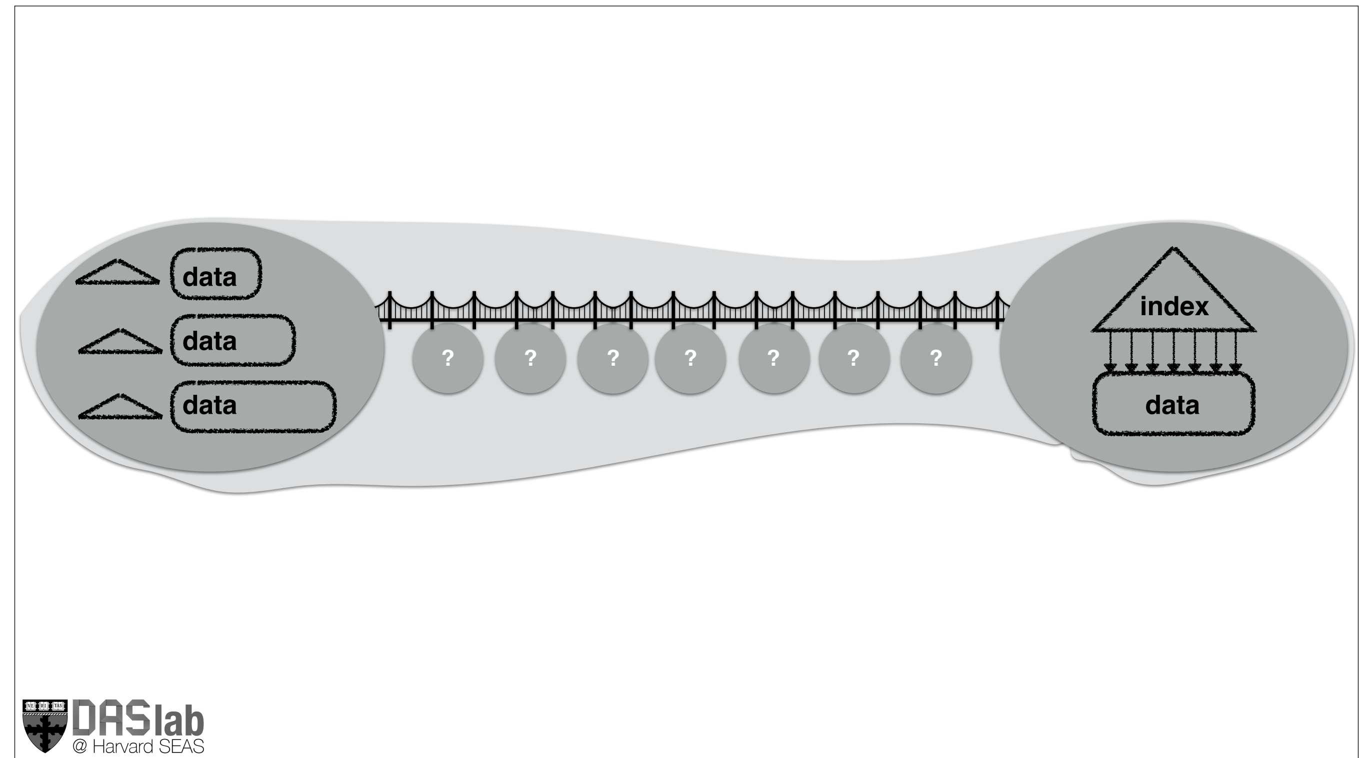

It would be great if we could somehow navigate the performance properties of both!



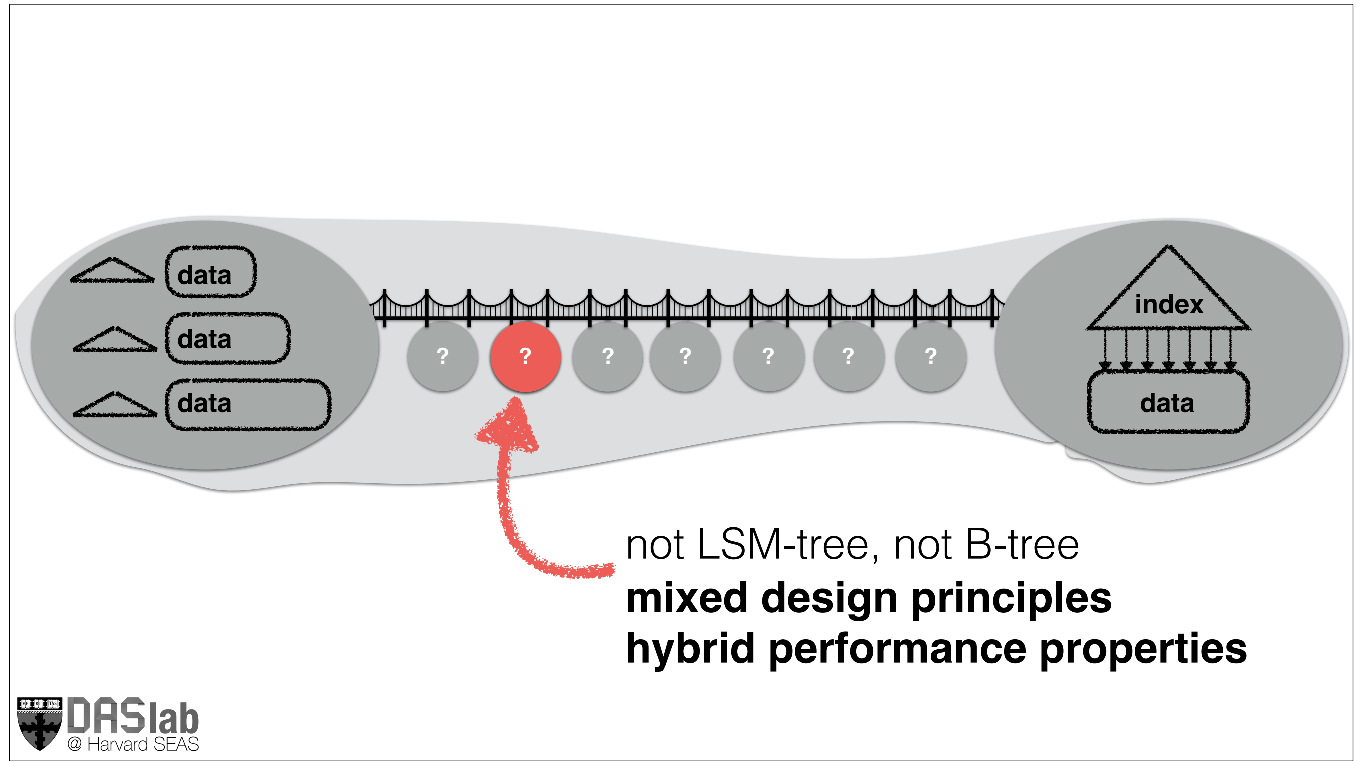

It would be great if we could somehow navigate the performance properties of both!



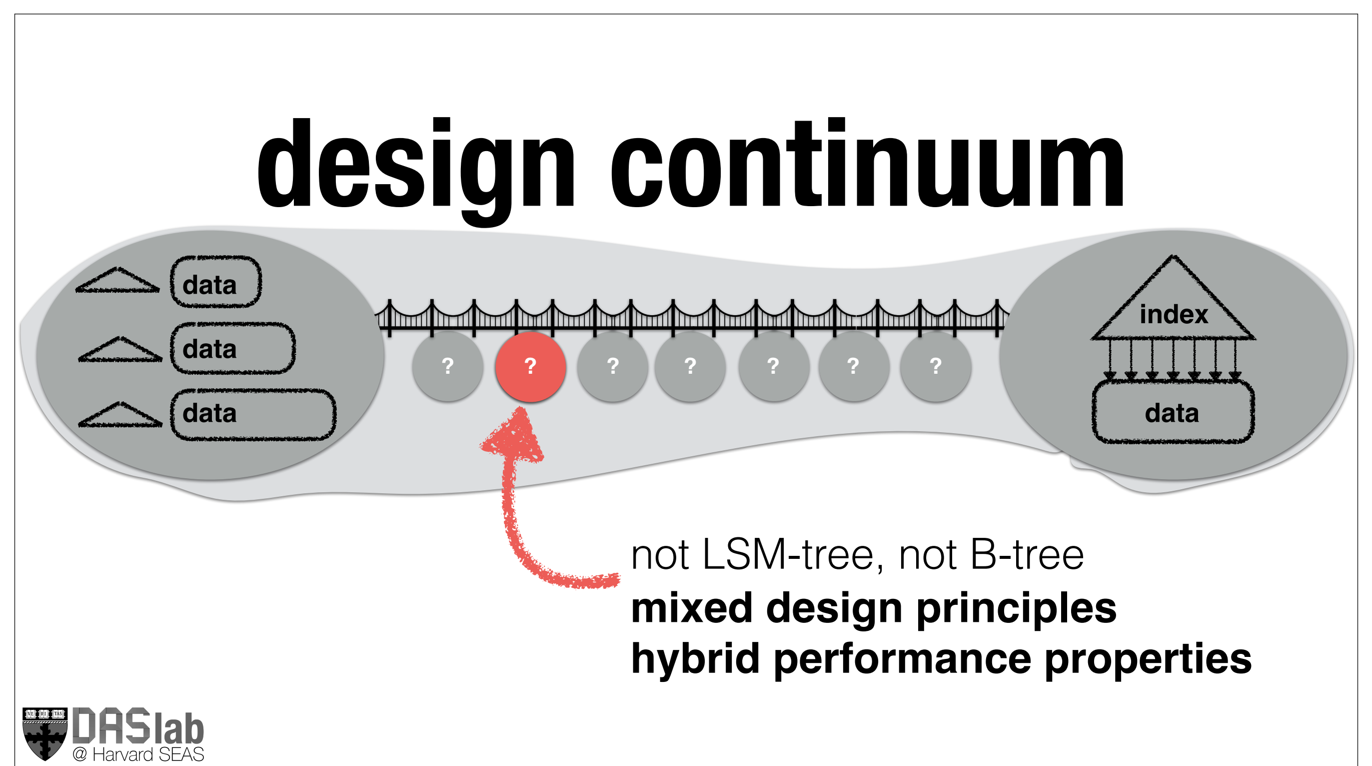

It would be great if we could somehow navigate the performance properties of both!



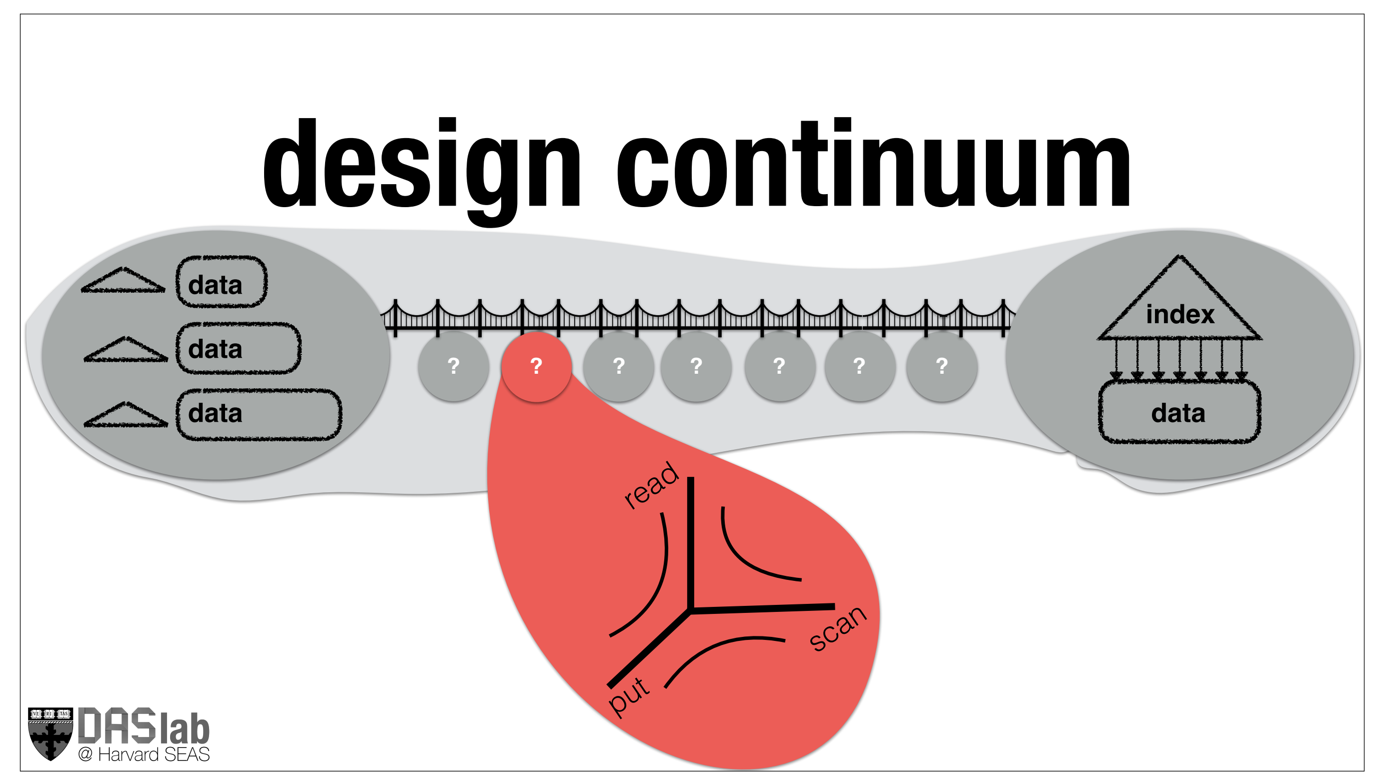

This is what design continuums allow. They allow us to “see” the design points in between different classes of data structures. And be able to navigate them.



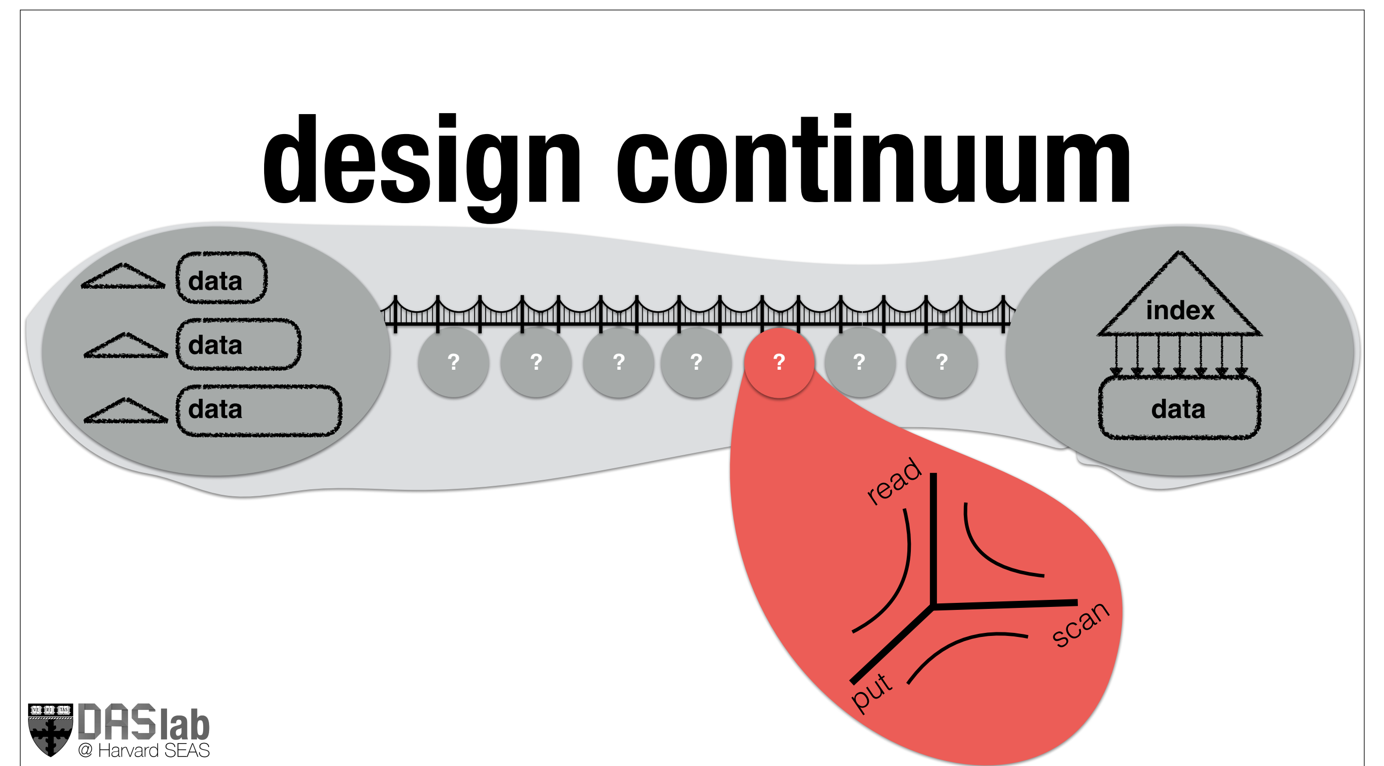

This is what design continuums allow. They allow us to “see” the design points in between different classes of data structures. And be able to navigate them.




This is what design continuums allow. They allow us to “see” the design points in between different classes of data structures. And be able to navigate them.


design continuum
data
data
data
index
data
?
?
?
?
?
?
?
read
scan
put
DASlab
@ Harvard SEAS

# design continuum



# totally different?

Level 1

Level 2

Level 3

...

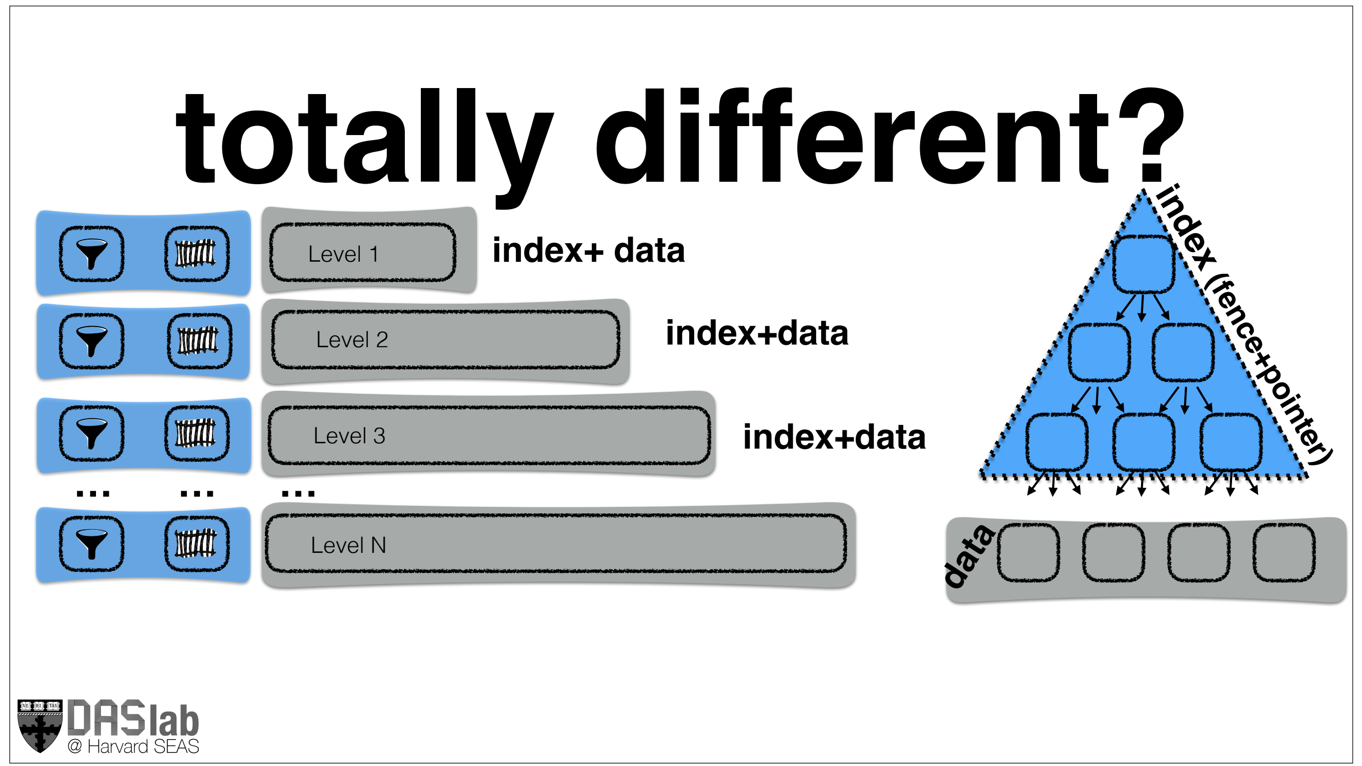
Level N

index (fence+pointer)

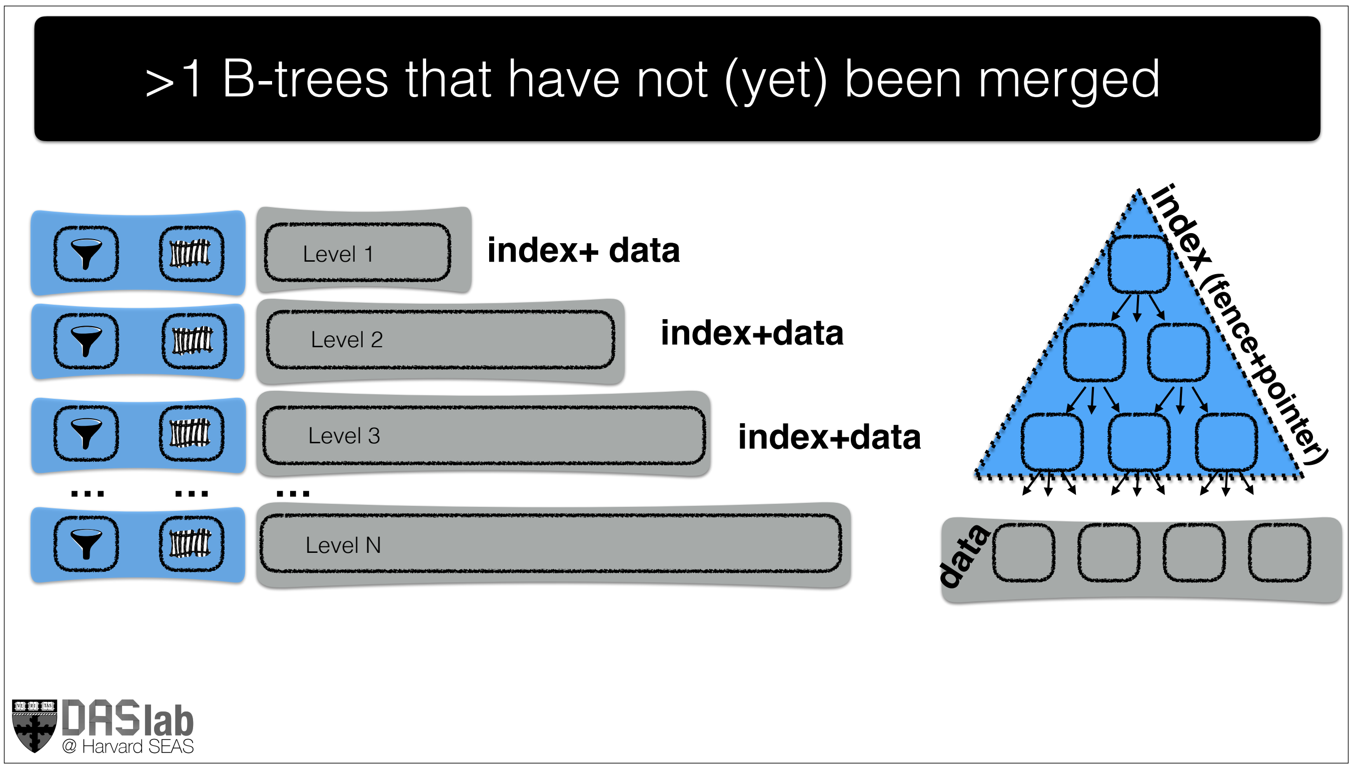
data

DASlab @ Harvard SEAS

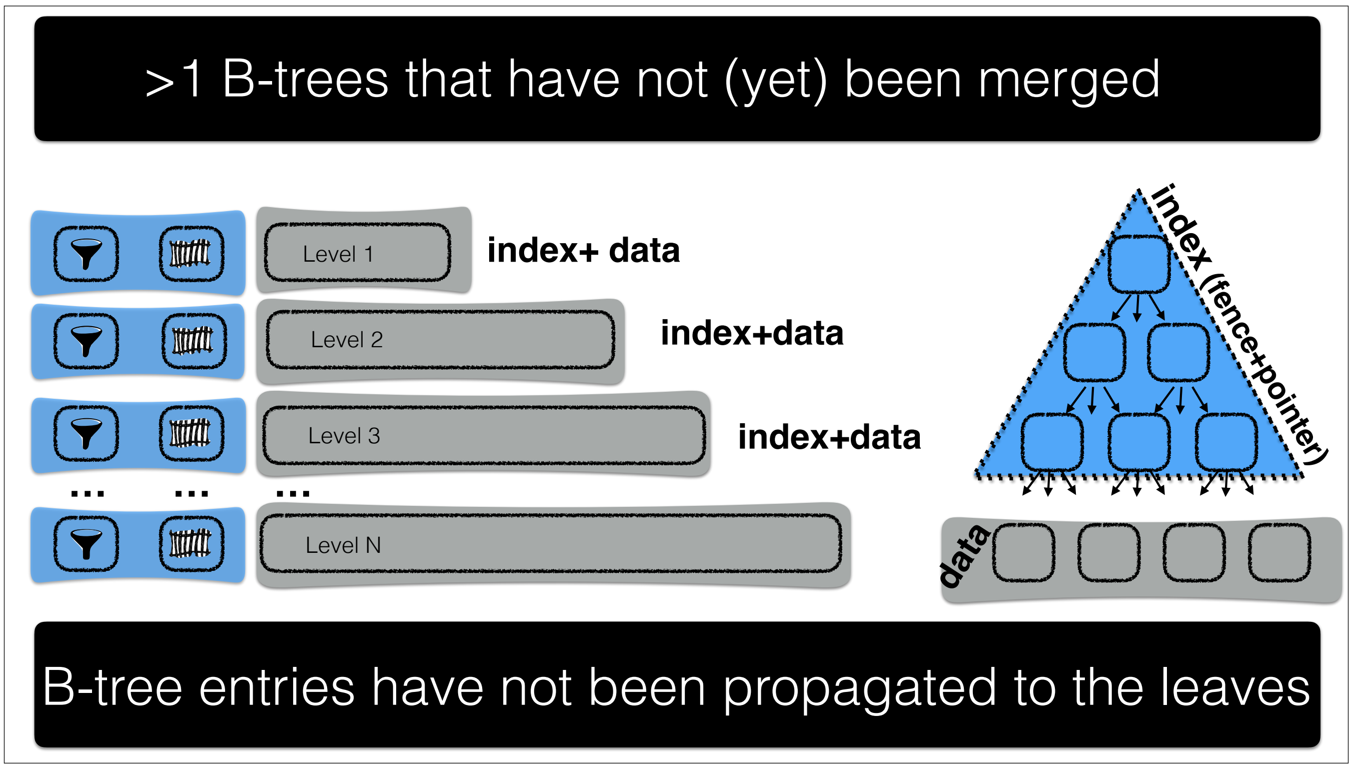
The explanation of why this is possible goes back to first principle and synthesis. Now that we know the design space of data structures we can actually observe that these designs consist of many similar components and many components that while they are different they serve the same functionality.



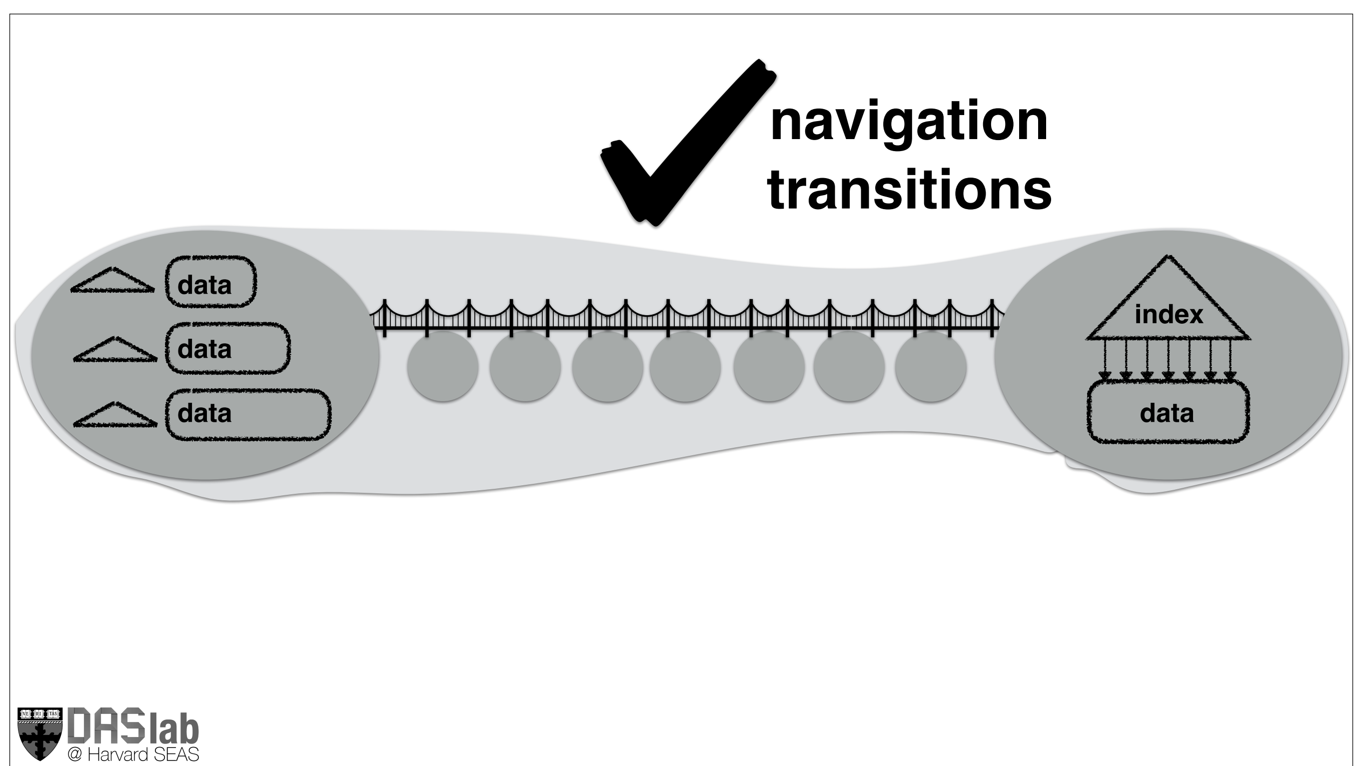

The explanation of why this is possible goes back to first principle and synthesis. Now that we know the design space of data structures we can actually observe that these designs consist of many similar components and many components that while they are different they serve the same functionality.

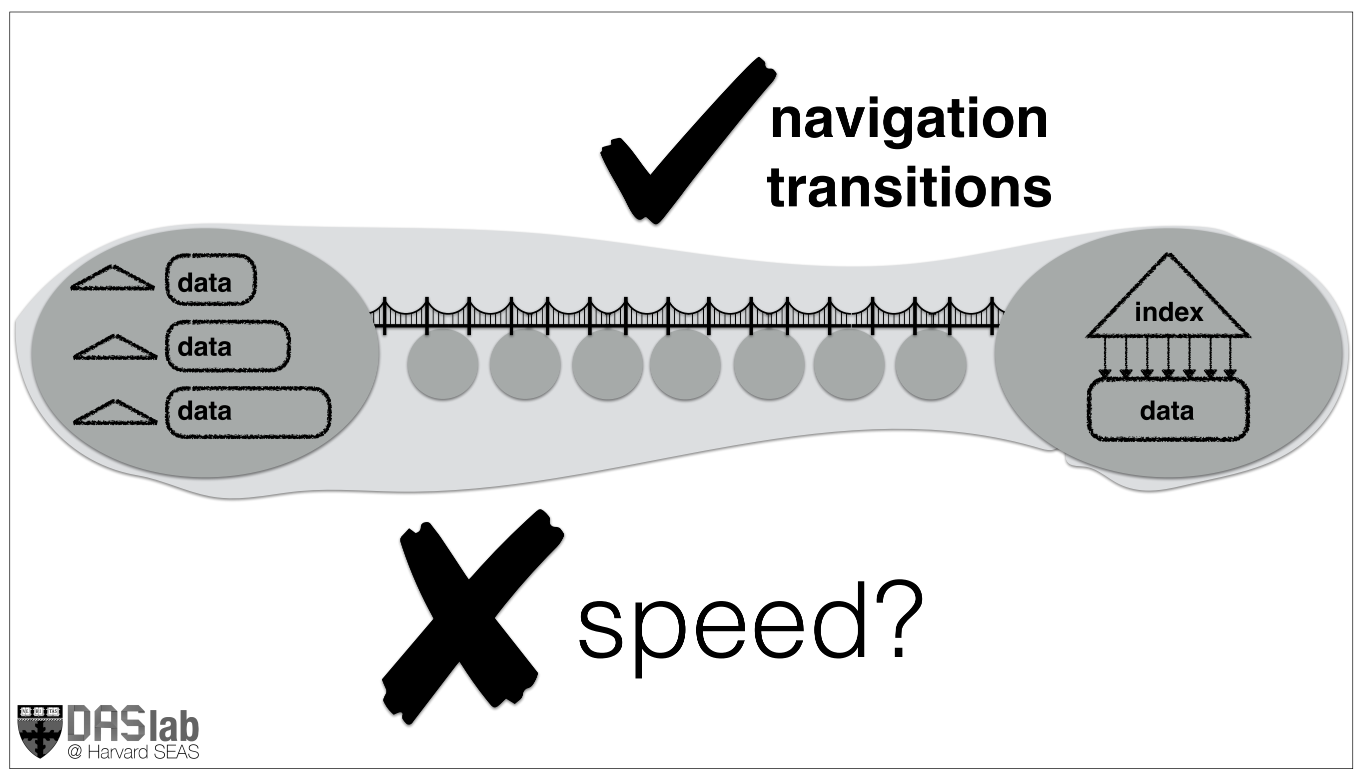


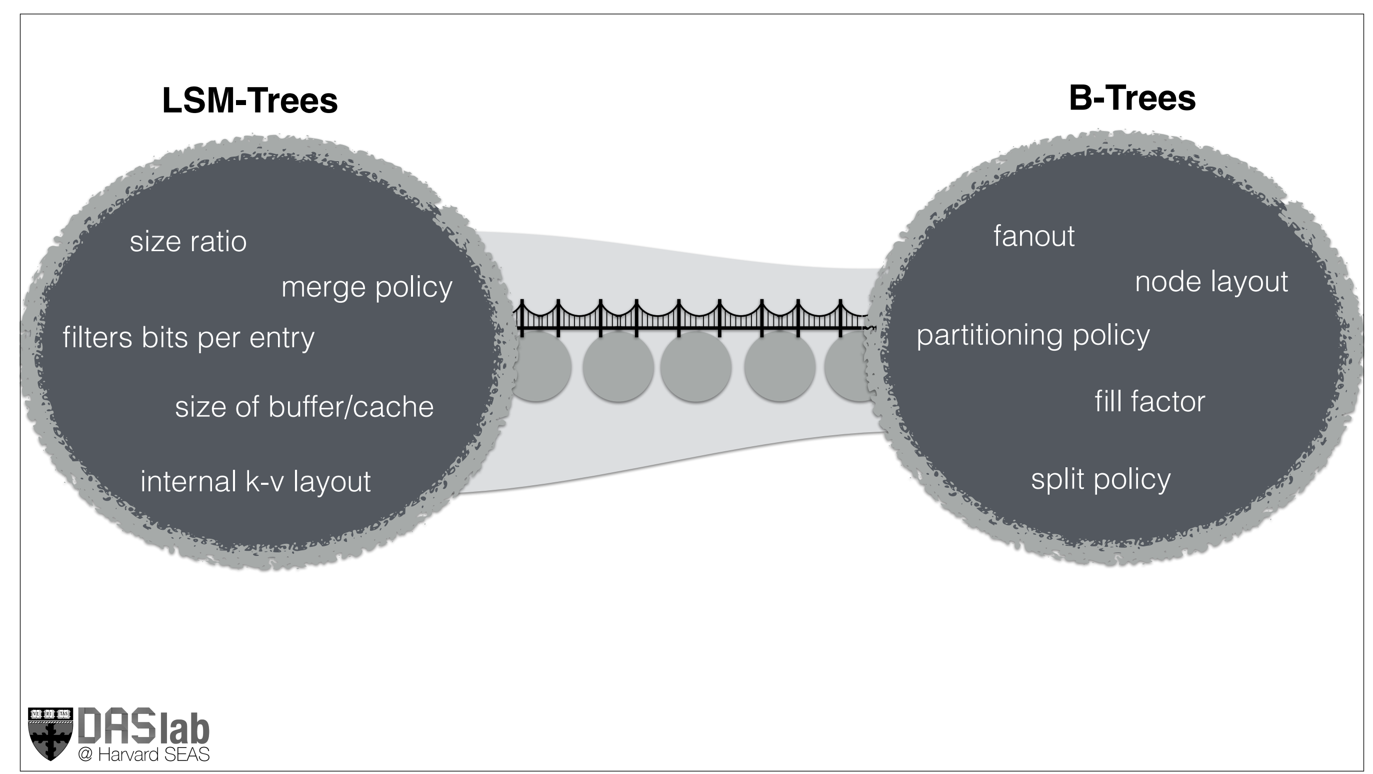

The explanation of why this is possible goes back to first principle and synthesis. Now that we know the design space of data structures we can actually observe that these designs consist of many similar components and many components that while they are different they serve the same functionality.

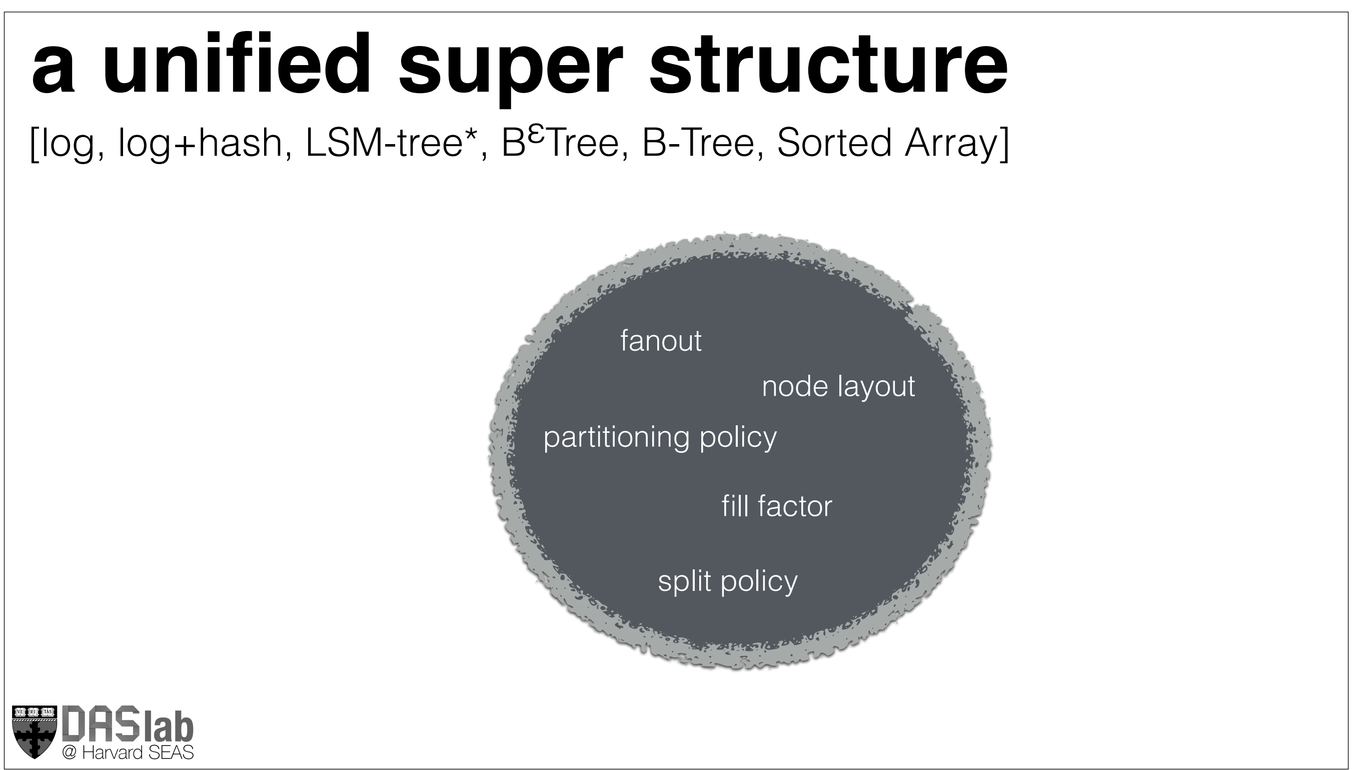



The explanation of why this is possible goes back to first principle and synthesis. Now that we know the design space of data structures we can actually observe that these designs consist of many similar components and many components that while they are different they serve the same functionality.

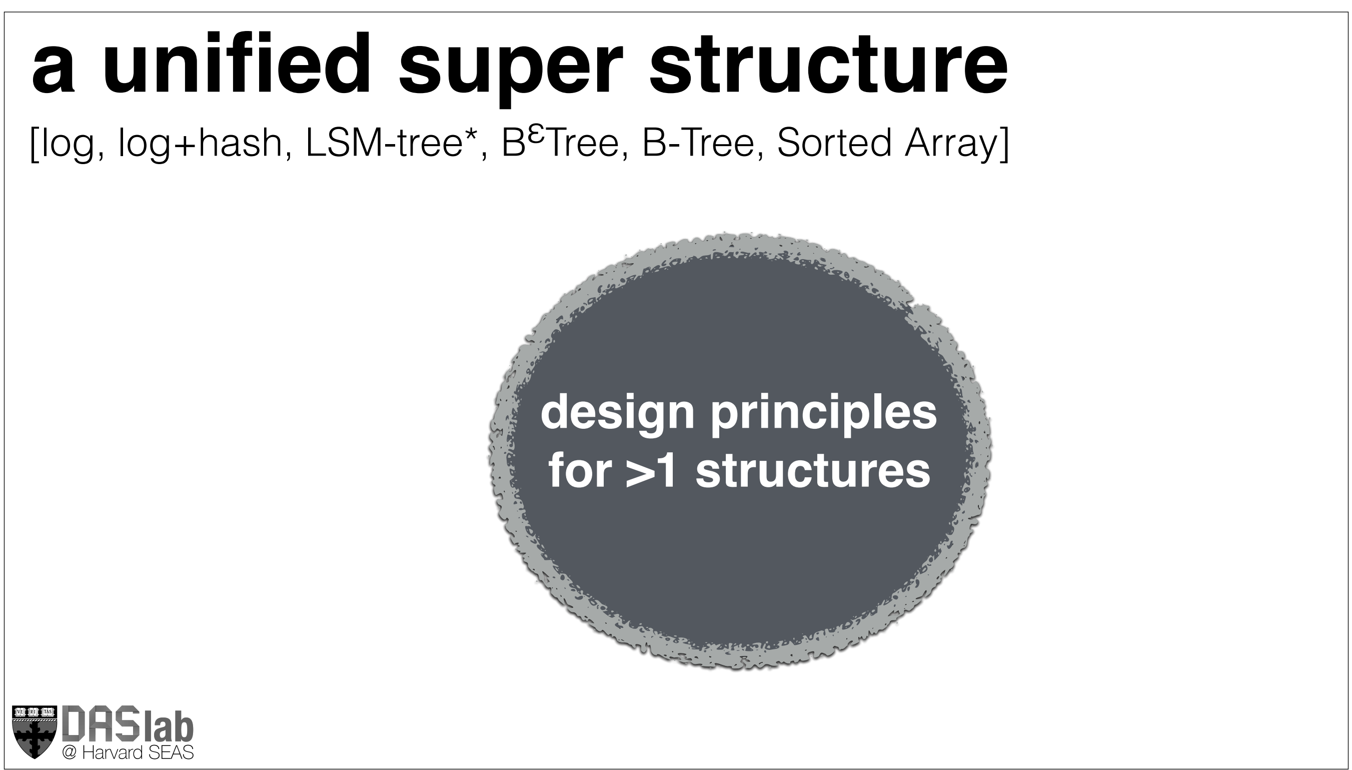

navigation
transitions
data
data
data
index
data
DASlab
@ Harvard SEAS


navigation
transitions
data
data
data
index
data
speed?
DASlab
@ Harvard SEAS

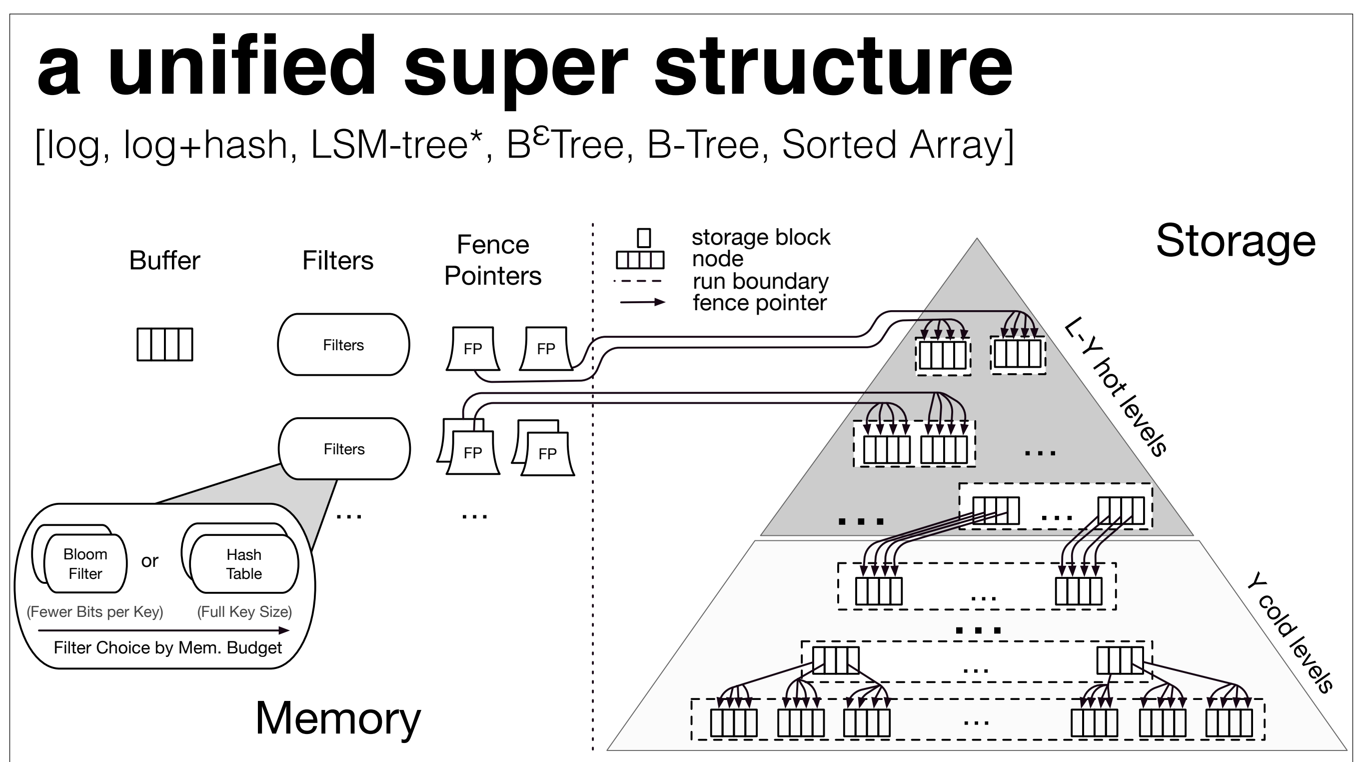

LSM-Trees
B-Trees
size ratio
merge policy
filters bits per entry
size of buffer/cache
internal k-v layout
fanout
node layout
partitioning policy
fill factor
split policy
DASlab
@ Harvard SEAS



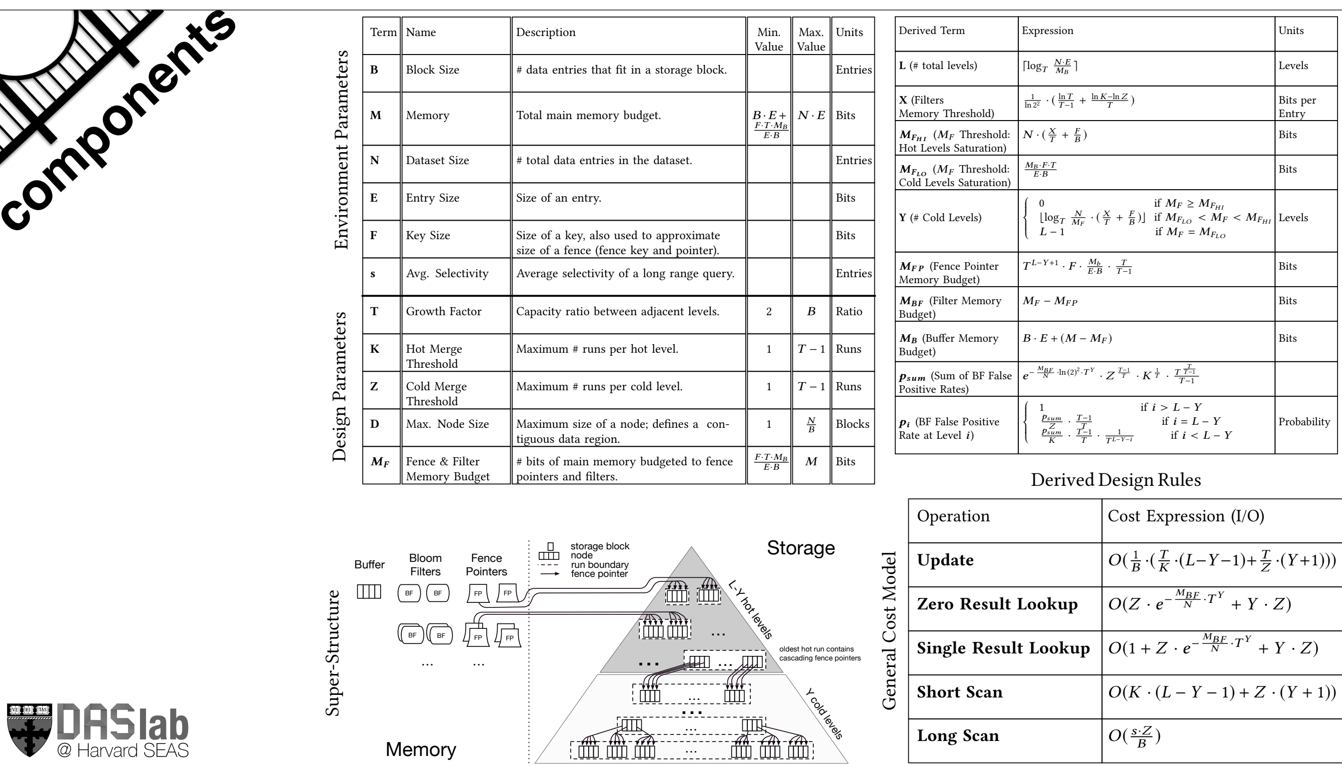

Using the design principles we have crafted the first design continuum which connects several well known data structures spanning log, LSM-based designs, B-tree based design, all the way to sorted array. The design continuum means that we can see those design as incarnations of a single “super structure”.



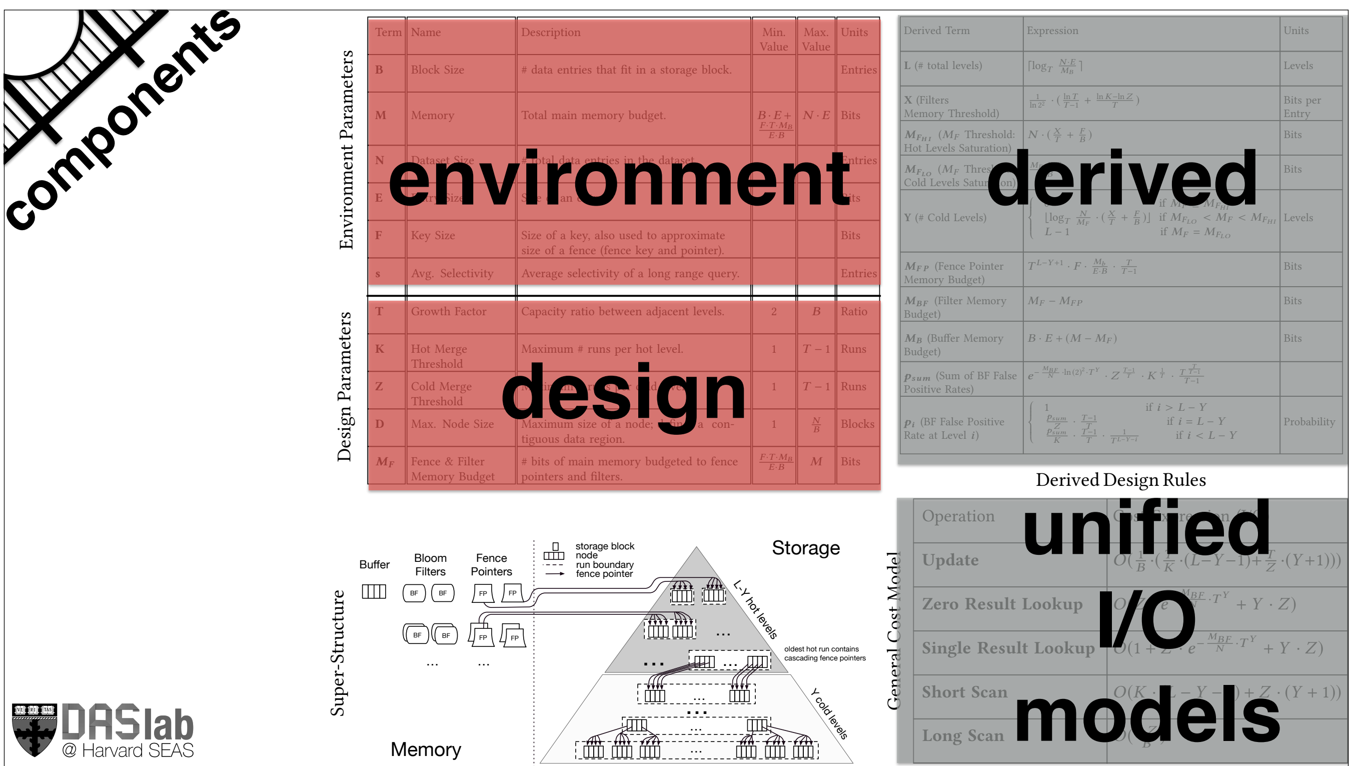

Using the design principles we have crafted the first design continuum which connects several well known data structures spanning log, LSM-based designs, B-tree based design, all the way to sorted array. The design continuum means that we can see those design as incarnations of a single “super structure”.



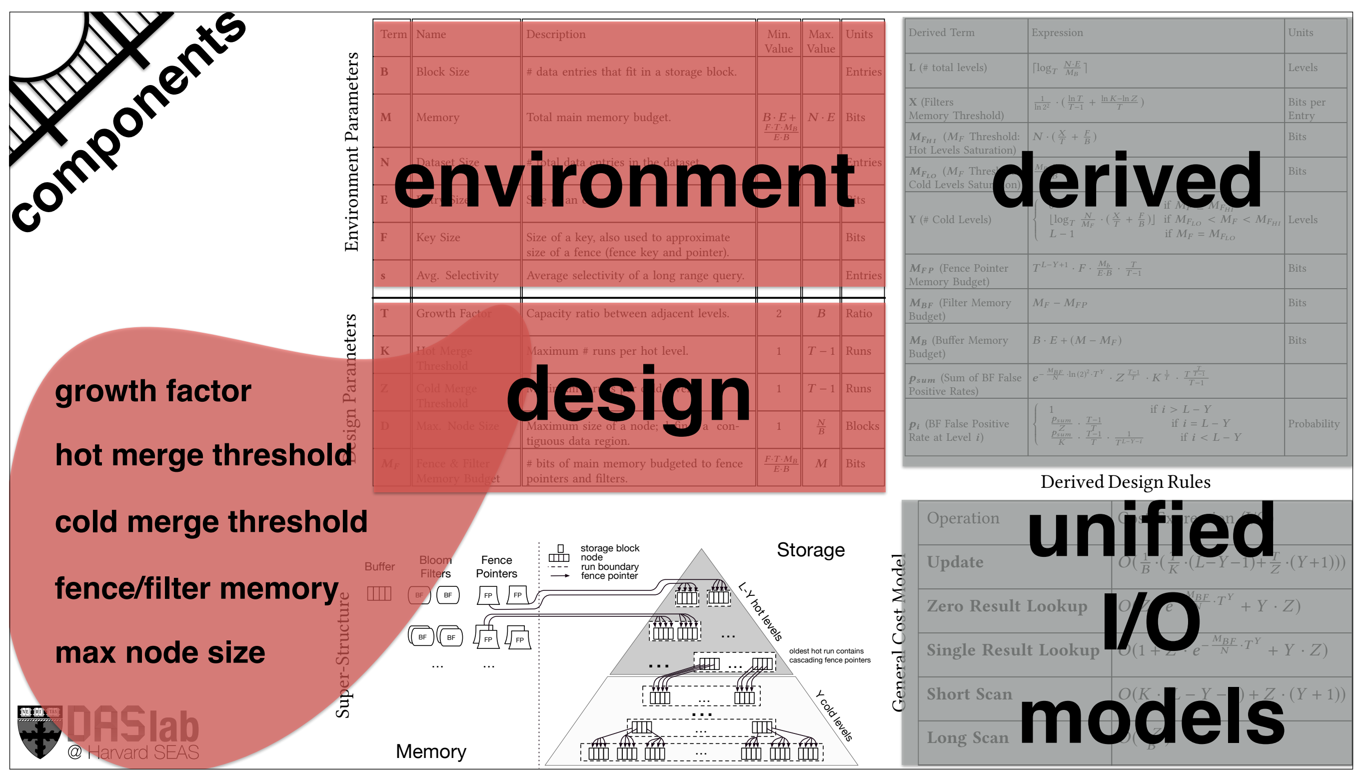

Using the design principles we have crafted the first design continuum which connects several well known data structures spanning log, LSM-based designs, B-tree based design, all the way to sorted array. The design continuum means that we can see those design as incarnations of a single “super structure”.




A design continuum contains a very small set of knobs (this first one contains 5) which can be used to instantiate any member design. The “magic” comes from a carefully crafted set of rules which dictate how the design is synthesized from the design principles and a single set of closed form I/O models that describe basic operations across *all* member designs.  While we currently do not have an automatic way to construct design continuums the exciting fact is that it is actually possible to do so as proven by the first design continuum. The CIDR paper explains in more detail how to create one and what are the desired properties.




A design continuum contains a very small set of knobs (this first one contains 5) which can be used to instantiate any member design. The “magic” comes from a carefully crafted set of rules which dictate how the design is synthesized from the design principles and a single set of closed form I/O models that describe basic operations across *all* member designs.  While we currently do not have an automatic way to construct design continuums the exciting fact is that it is actually possible to do so as proven by the first design continuum. The CIDR paper explains in more detail how to create one and what are the desired properties.




A design continuum contains a very small set of knobs (this first one contains 5) which can be used to instantiate any member design. The "magic" comes from a carefully crafted set of rules which dictate how the design is synthesized from the design principles and a single set of closed form I/O models that describe basic operations across *all* member designs.  While we currently do not have an automatic way to construct design continuums the exciting fact is that it is actually possible to do so as proven by the first design continuum. The CIDR paper explains in more detail how to create one and what are the desired properties.

**from write to read optimized**

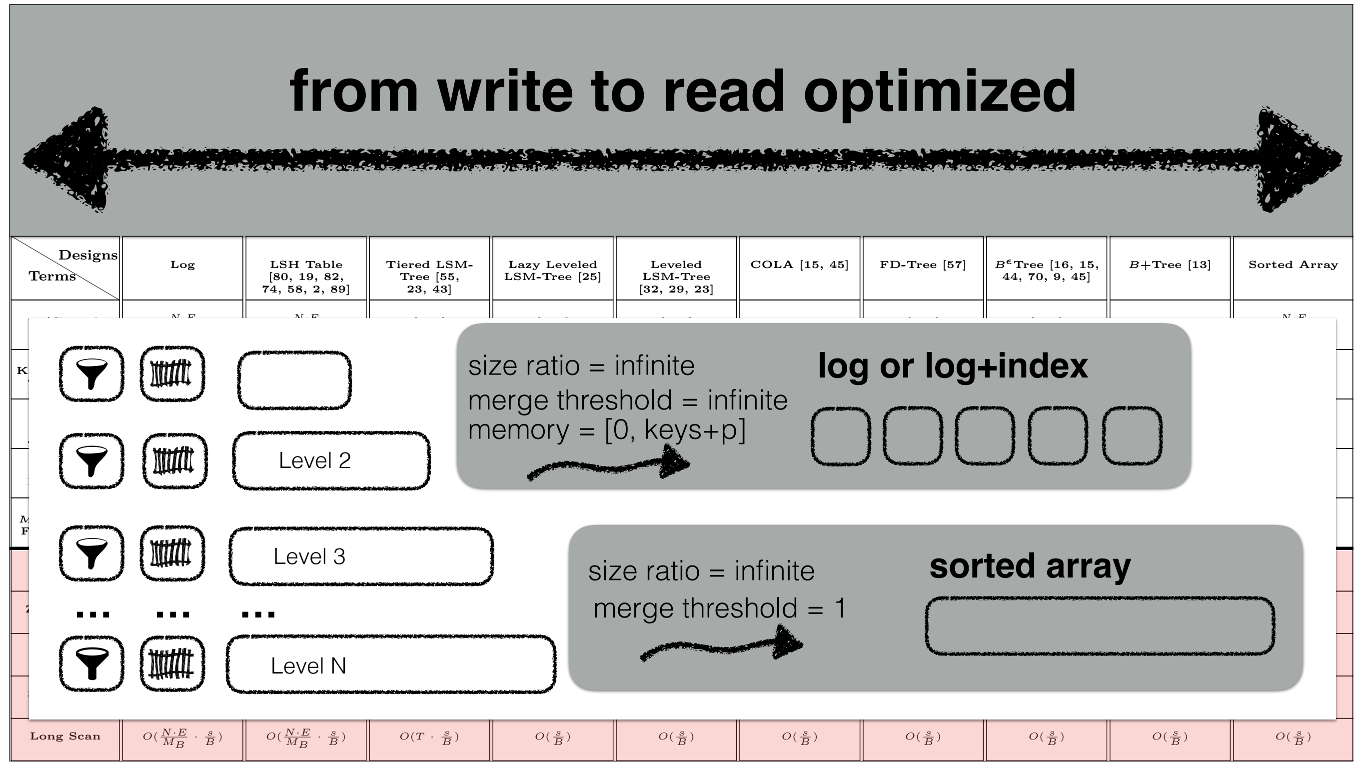
| Terms \ Designs | Log | LSH Table [80, 19, 82, 74, 58, 2, 89] | Tiered LSM-Tree [55, 23, 43] | Lazy Leveled LSM-Tree [25] | Leveled LSM-Tree [32, 29, 23] | COLA [15, 45] | FD-Tree [57] | $B^\epsilon$Tree [16, 15, 44, 70, 9, 45] | $B$+Tree [13] | Sorted Array |
|---|---|---|---|---|---|---|---|---|---|---|
| T (Growth Factor) | $\frac{N \cdot E}{M_B}$ | $\frac{N \cdot E}{M_B}$ | $[2, B]$ | $[2, B]$ | $[2, B]$ | $2$ | $[2, B]$ | $[2, B]$ | $B$ | $\frac{N \cdot E}{M_B}$ |
| K (Hot Merge Threshold) | $T - 1$ | $T - 1$ | $T - 1$ | $T - 1$ | $1$ | $1$ | $1$ | $1$ | $1$ | $1$ |
| Z (Cold Merge Threshold) | $T - 1$ | $T - 1$ | $T - 1$ | $1$ | $1$ | $1$ | $1$ | $1$ | $1$ | $1$ |
| D (Max. Node Size) | $1$ | $1$ | $[1, \frac{N}{B}]$ | $[1, \frac{N}{B}]$ | $[1, \frac{N}{B}]$ | $\frac{N}{B}$ | $\frac{N}{B}$ | $1$ | $1$ | $\frac{N}{B}$ |
| $M_F$ (Fence & Filter Mem.) | $\frac{N \cdot F}{B}$ | $N \cdot F \cdot (1 + \frac{1}{B})$ | $N \cdot (\frac{F}{B} + 10)$ | $N \cdot (\frac{F}{B} + 10)$ | $N \cdot (\frac{F}{B} + 10)$ | $\frac{F \cdot T \cdot M_B}{E \cdot B}$ | $\frac{F \cdot T \cdot M_B}{E \cdot B}$ | $\frac{F \cdot T \cdot M_B}{E \cdot B}$ | $\frac{F \cdot T \cdot M_B}{E \cdot B}$ | $\frac{N \cdot F}{B}$ |
| **Update** | $O(\frac{1}{B})$ | $O(\frac{1}{B})$ | $O(\frac{L}{B})$ | $O(\frac{1}{B} \cdot (T + L))$ | $O(\frac{T}{B} \cdot L)$ | $O(\frac{L}{B})$ | $O(\frac{T}{B} \cdot L)$ | $O(\frac{T}{B} \cdot L)$ | $O(L)$ | $O(\frac{N \cdot E}{M_B \cdot B})$ |
| **Zero Result Lookup** | $O(\frac{N \cdot E}{M_B})$ | $O(0)$ | $O(T \cdot e^{\frac{-M_B F}{N}})$ | $O(e^{-\frac{M_B F}{N}})$ | $O(e^{-\frac{M_B F}{N}})$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(1)$ |
| **Existing Lookup** | $O(\frac{N \cdot E}{M_B})$ | $O(1)$ | $O(1 + T \cdot e^{\frac{-M_B F}{N}})$ | $O(1)$ | $O(1)$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(1)$ |
| **Short Scan** | $O(\frac{N \cdot E}{M_B})$ | $O(\frac{N \cdot E}{M_B})$ | $O(L \cdot T)$ | $O(1 + T \cdot (L - 1))$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(1)$ |
| **Long Scan** | $O(\frac{N \cdot E}{M_B} \cdot \frac{s}{B})$ | $O(\frac{N \cdot E}{M_B} \cdot \frac{s}{B})$ | $O(T \cdot \frac{s}{B})$ | $O(\frac{s}{B})$ | $O(\frac{s}{B})$ | $O(\frac{s}{B})$ | $O(\frac{s}{B})$ | $O(\frac{s}{B})$ | $O(\frac{s}{B})$ | $O(\frac{s}{B})$ |

Examples of how different data structures are instantiated by the first design continuum. One of the proofs is that the closed form models actually work, i.e., they produce the expected cost for these designs.

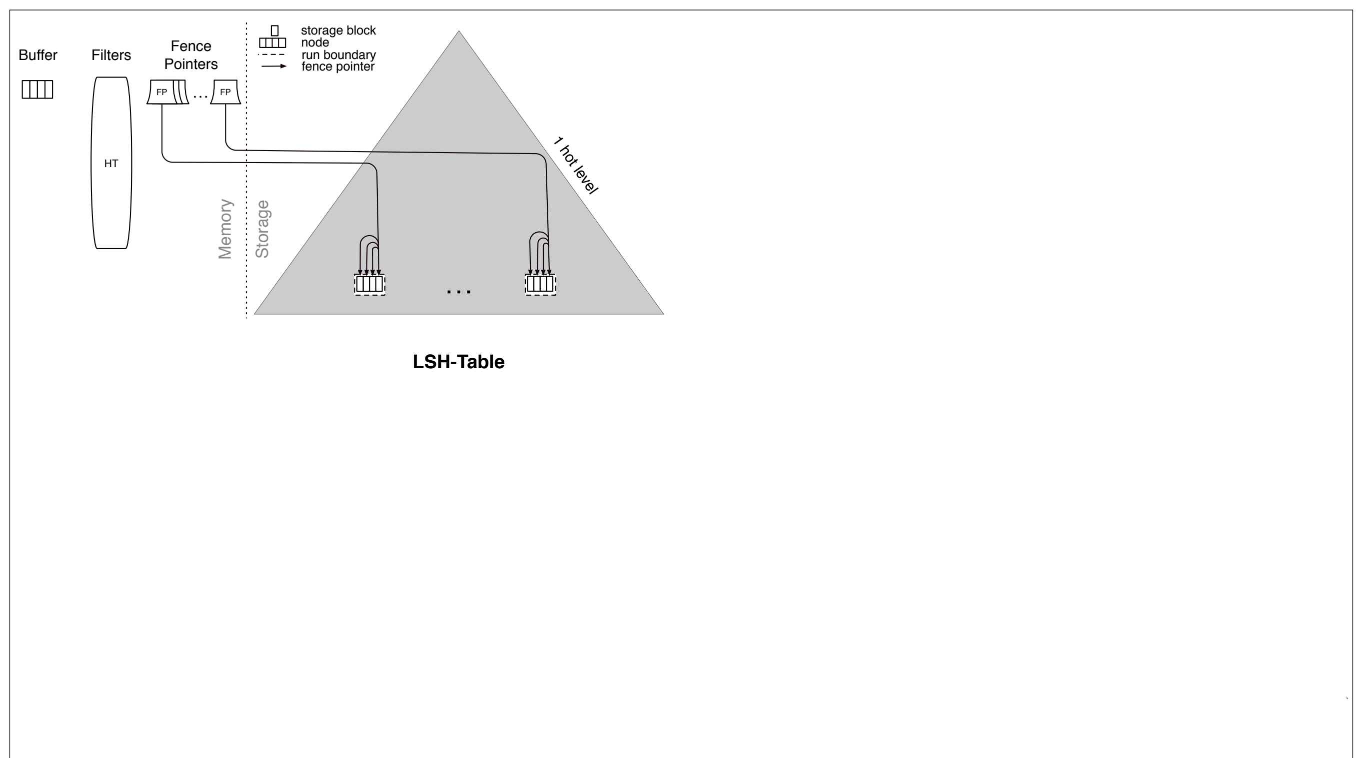
## from write to read optimized

| Terms \ Designs | Log | LSH Table [80, 19, 82, 74, 58, 2, 89] | Tiered LSM-Tree [55, 23, 43] | Lazy Leveled LSM-Tree [25] | Leveled LSM-Tree [32, 29, 23] | COLA [15, 45] | FD-Tree [57] | $B^{\epsilon}$Tree [16, 15, 44, 70, 9, 45] | $B$+Tree [13] | Sorted Array |
|---|---|---|---|---|---|---|---|---|---|---|
| T (Growth Factor) | $\frac{N \cdot E}{M_B}$ | $\frac{N \cdot E}{M_B}$ | $[2, B]$ | $[2, B]$ | $[2, B]$ | $2$ | $[2, B]$ | $[2, B]$ | $B$ | $\frac{N \cdot E}{M_B}$ |
| K (Hot Merge Threshold) | $T - 1$ | $T - 1$ | $T - 1$ | $T - 1$ | $1$ | $1$ | $1$ | $1$ | $1$ | $1$ |
| Z (Cold Merge Threshold) | $T - 1$ | $T - 1$ | $T - 1$ | $1$ | $1$ | $1$ | $1$ | $1$ | $1$ | $1$ |
| D (Max. Node Size) | $1$ | $1$ | $[1, \frac{N}{B}]$ | $[1, \frac{N}{B}]$ | $[1, \frac{N}{B}]$ | $\frac{N}{B}$ | $\frac{N}{B}$ | $1$ | $1$ | $\frac{N}{B}$ |
| $M_F$ (Fence & Filter Mem.) | $\frac{N \cdot F}{B}$ | $N \cdot F \cdot (1 + \frac{1}{B})$ | $N \cdot (\frac{F}{B} + 10)$ | $N \cdot (\frac{F}{B} + 10)$ | $N \cdot (\frac{F}{B} + 10)$ | $\frac{F \cdot T \cdot M_B}{E \cdot B}$ | $\frac{F \cdot T \cdot M_B}{E \cdot B}$ | $\frac{F \cdot T \cdot M_B}{E \cdot B}$ | $\frac{F \cdot T \cdot M_B}{E \cdot B}$ | $\frac{N \cdot F}{B}$ |
| Update | $O(\frac{1}{B})$ | $O(\frac{1}{B})$ | $O(\frac{L}{B})$ | $O(\frac{1}{B} \cdot (T + L))$ | $O(\frac{T}{B} \cdot L)$ | $O(\frac{L}{B})$ | $O(\frac{T}{B} \cdot L)$ | $O(\frac{T}{B} \cdot L)$ | $O(L)$ | $O(\frac{N \cdot E}{M_B \cdot B})$ |
| Zero Result Lookup | $O(\frac{N \cdot E}{M_B})$ | $O(0)$ | $O(T \cdot e^{\frac{-M_{BF}}{N}})$ | $O(e^{-\frac{M_{BF}}{N}})$ | $O(e^{-\frac{M_{BF}}{N}})$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(1)$ |
| Existing Lookup | $O(\frac{N \cdot E}{M_B})$ | $O(1)$ | $O(1+T \cdot e^{\frac{-M_{BF}}{N}})$ | $O(1)$ | $O(1)$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(1)$ |
| Short Scan | $O(\frac{N \cdot E}{M_B})$ | $O(\frac{N \cdot E}{M_B})$ | $O(L \cdot T)$ | $O(1 + T \cdot (L - 1))$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(L)$ | $O(1)$ |
| Long Scan | $O(\frac{N \cdot E}{M_B} \cdot \frac{s}{B})$ | $O(\frac{N \cdot E}{M_B} \cdot \frac{s}{B})$ | $O(T \cdot \frac{s}{B})$ | $O(\frac{s}{B})$ | $O(\frac{s}{B})$ | $O(\frac{s}{B})$ | $O(\frac{s}{B})$ | $O(\frac{s}{B})$ | $O(\frac{s}{B})$ | $O(\frac{s}{B})$ |

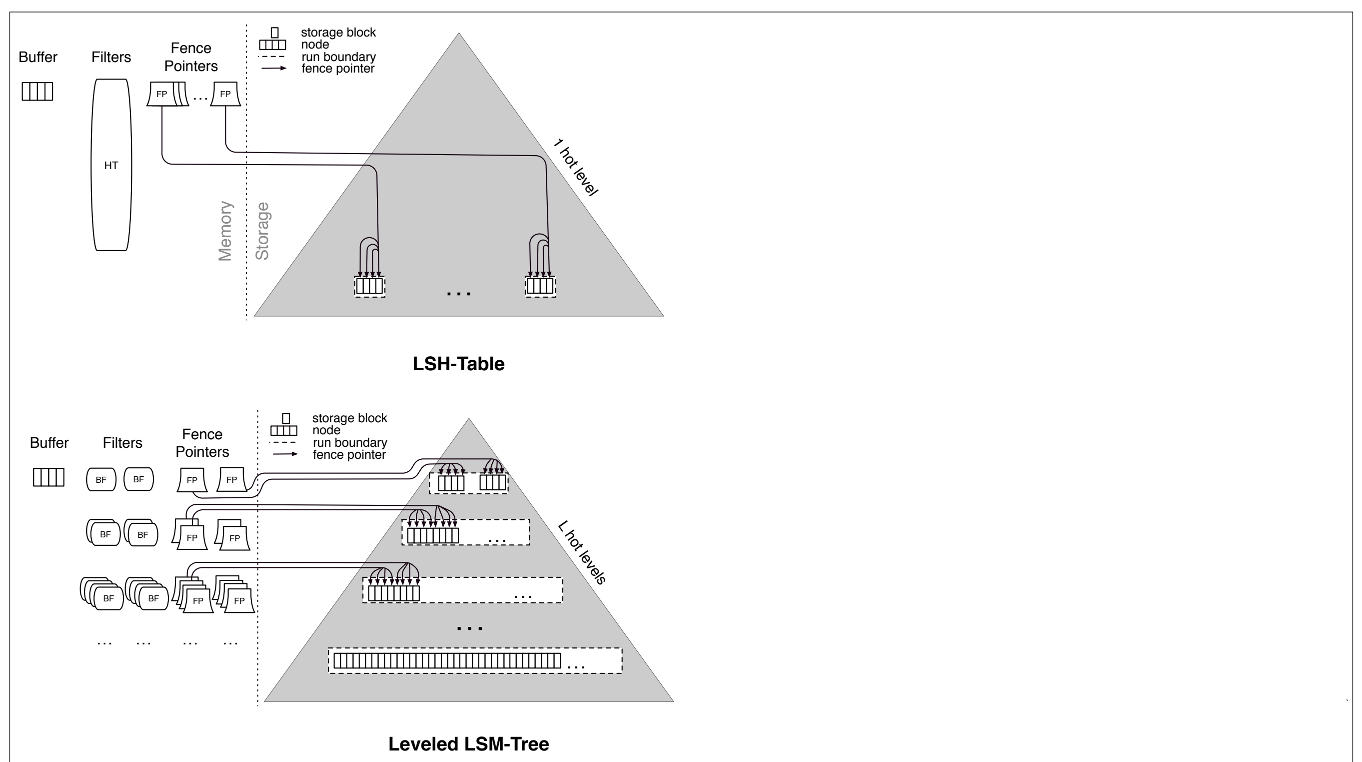
Examples of how different data structures are instantiated by the first design continuum. One of the proofs is that the closed form models actually work, i.e., they produce the expected cost for these designs.



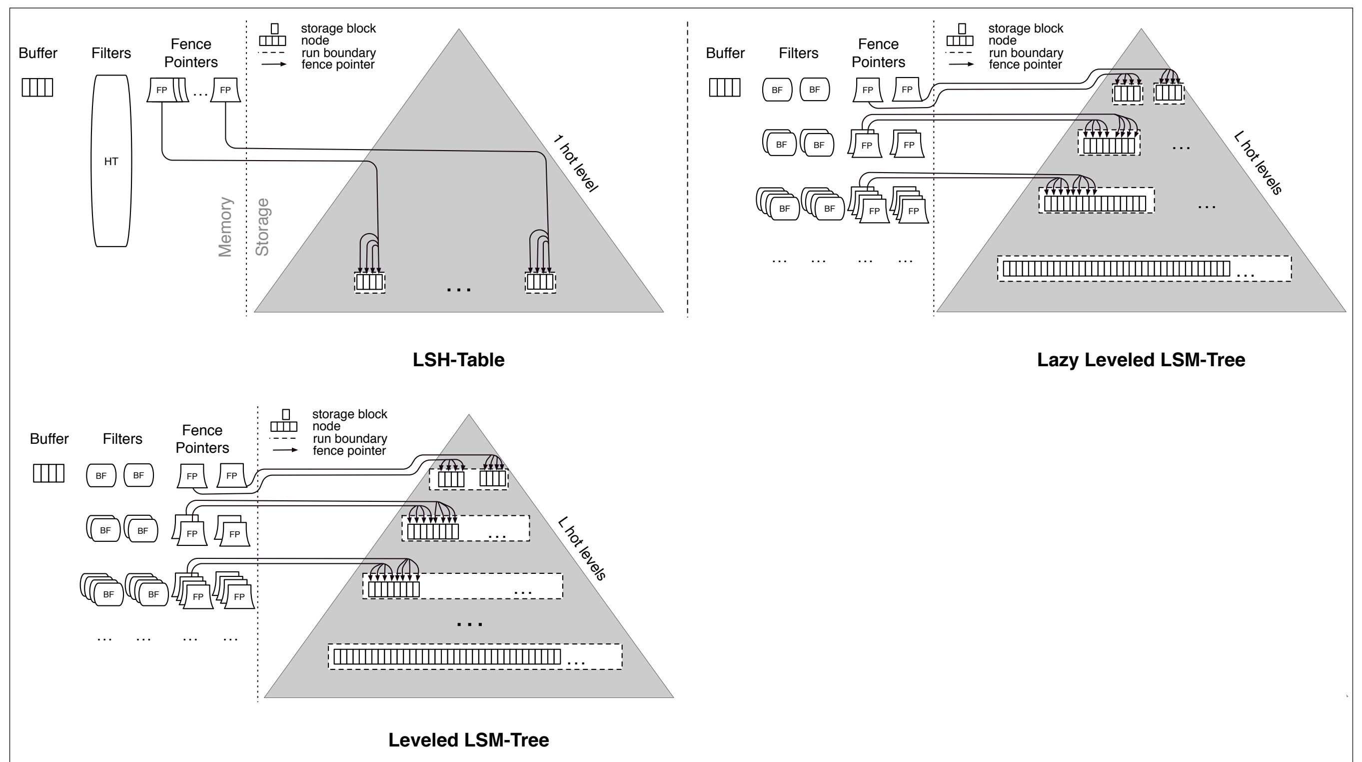

Examples of how different data structures are instantiated by the first design continuum. One of the proofs is that the closed form models actually work, i.e., they produce the expected cost for these designs.



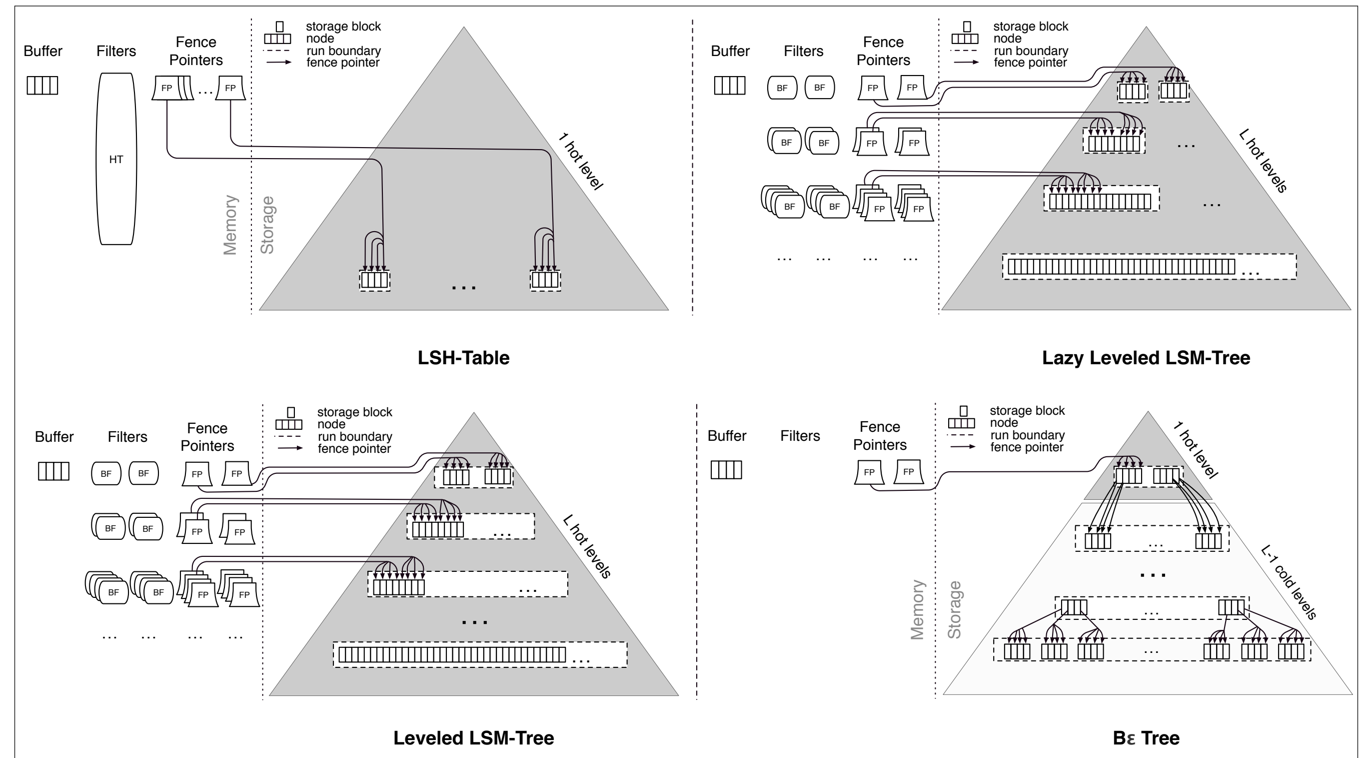

Visual examples of instantiating the super structure to individual tailored designs.



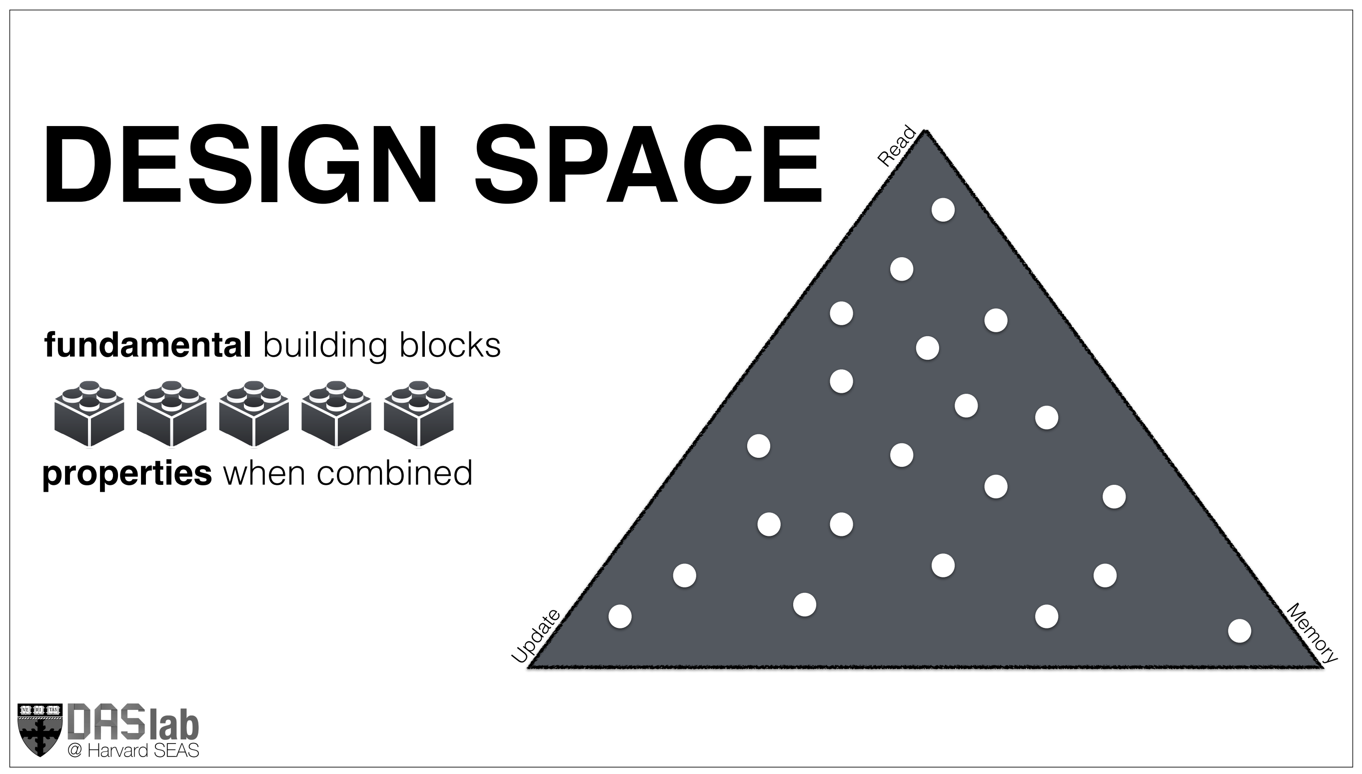

Visual examples of instantiating the super structure to individual tailored designs.



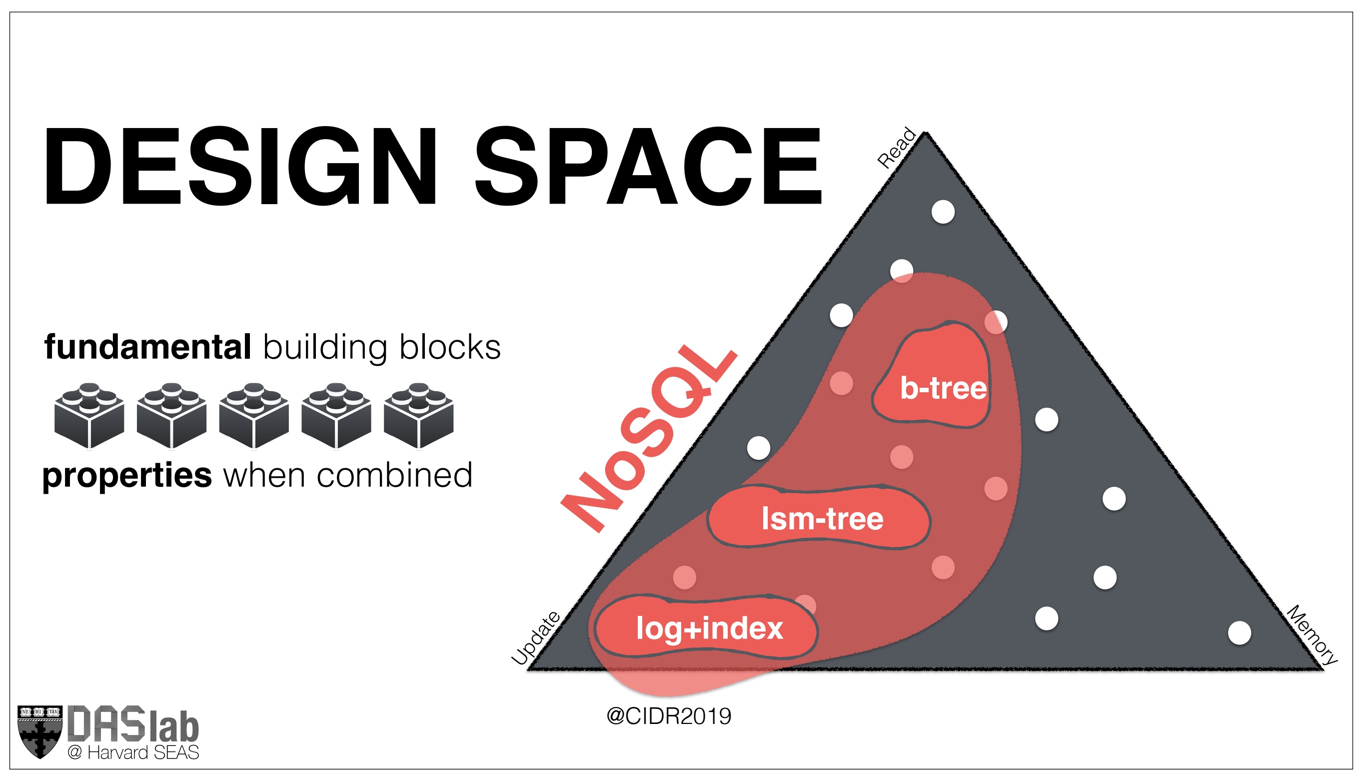

Visual examples of instantiating the super structure to individual tailored designs.



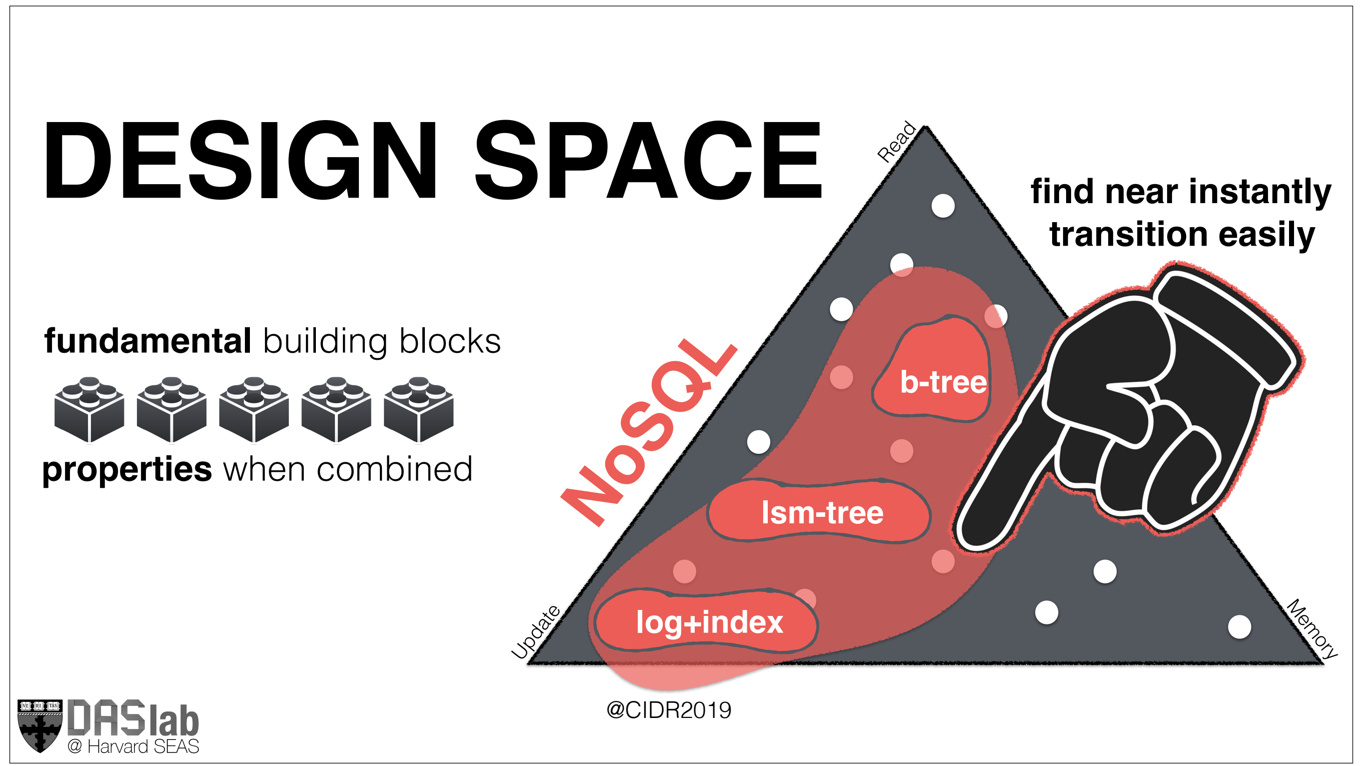

Visual examples of instantiating the super structure to individual tailored designs.



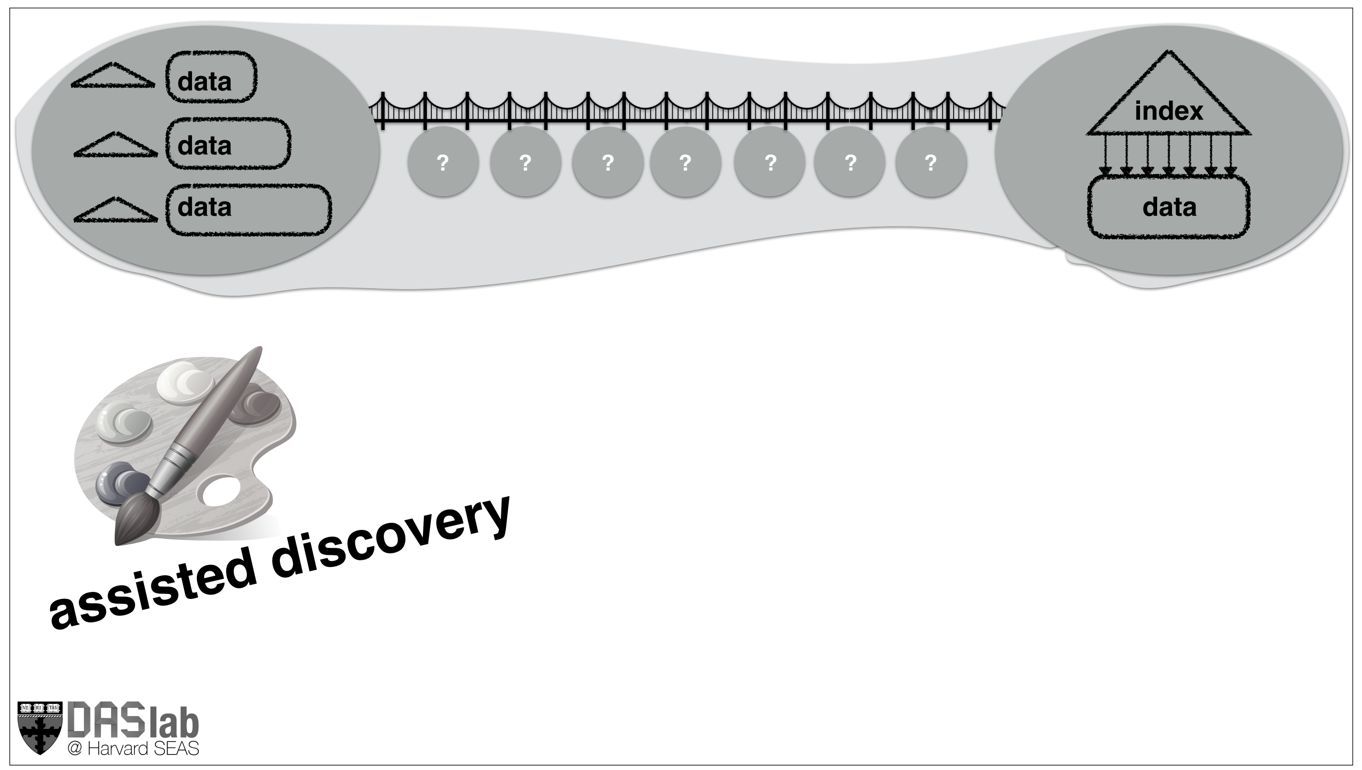

In the end what this means is that we can now instantly search within the design continuum for the best data structure! Because the closed form models are very fast to compute.



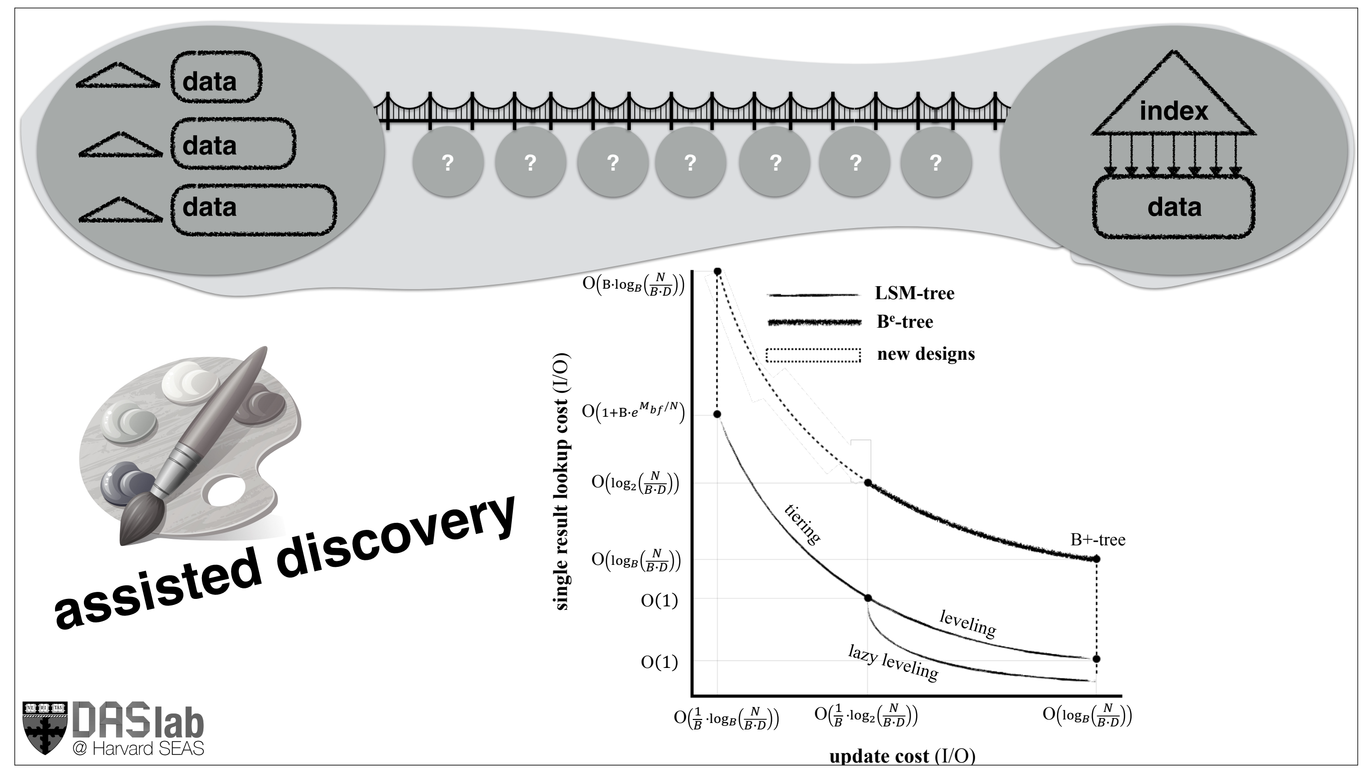

In the end what this means is that we can now instantly search within the design continuum for the best data structure! Because the closed form models are very fast to compute.



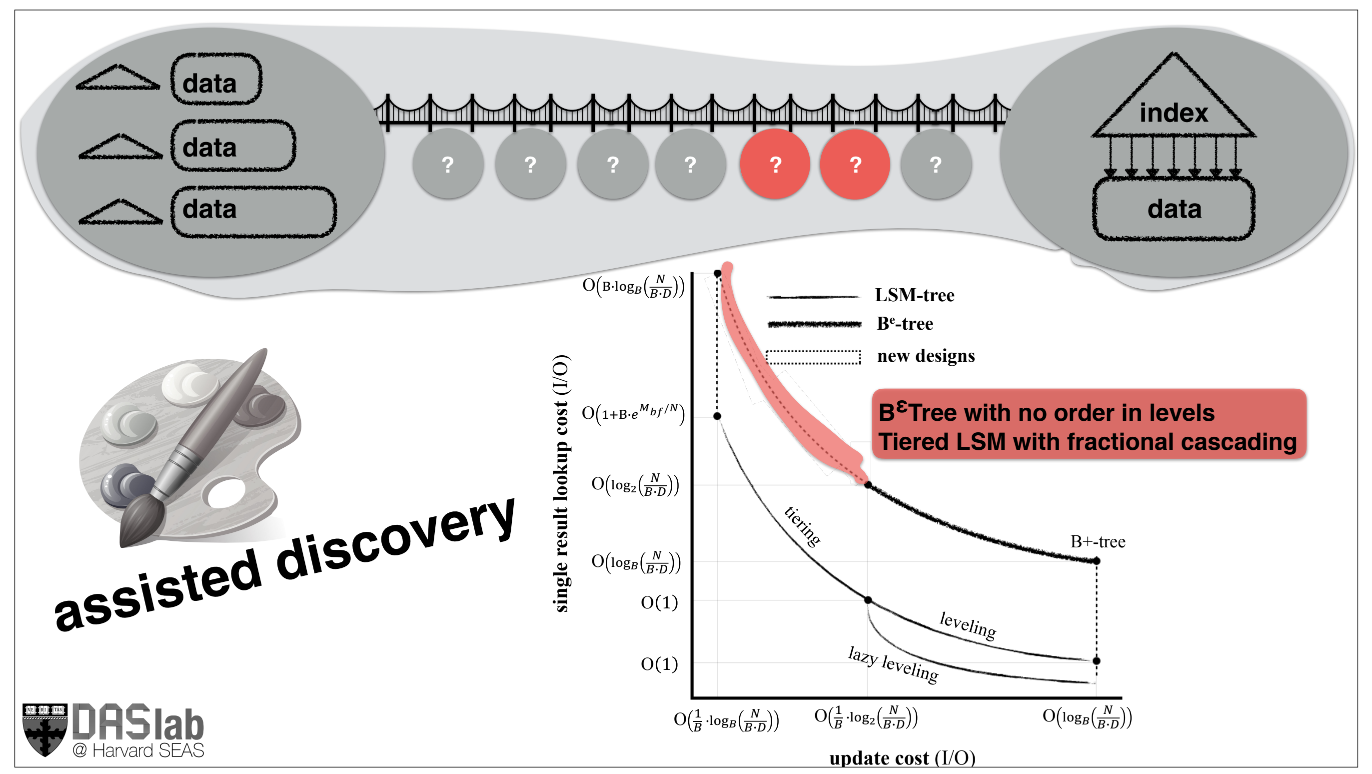

In the end what this means is that we can now instantly search within the design continuum for the best data structure! Because the closed form models are very fast to compute.



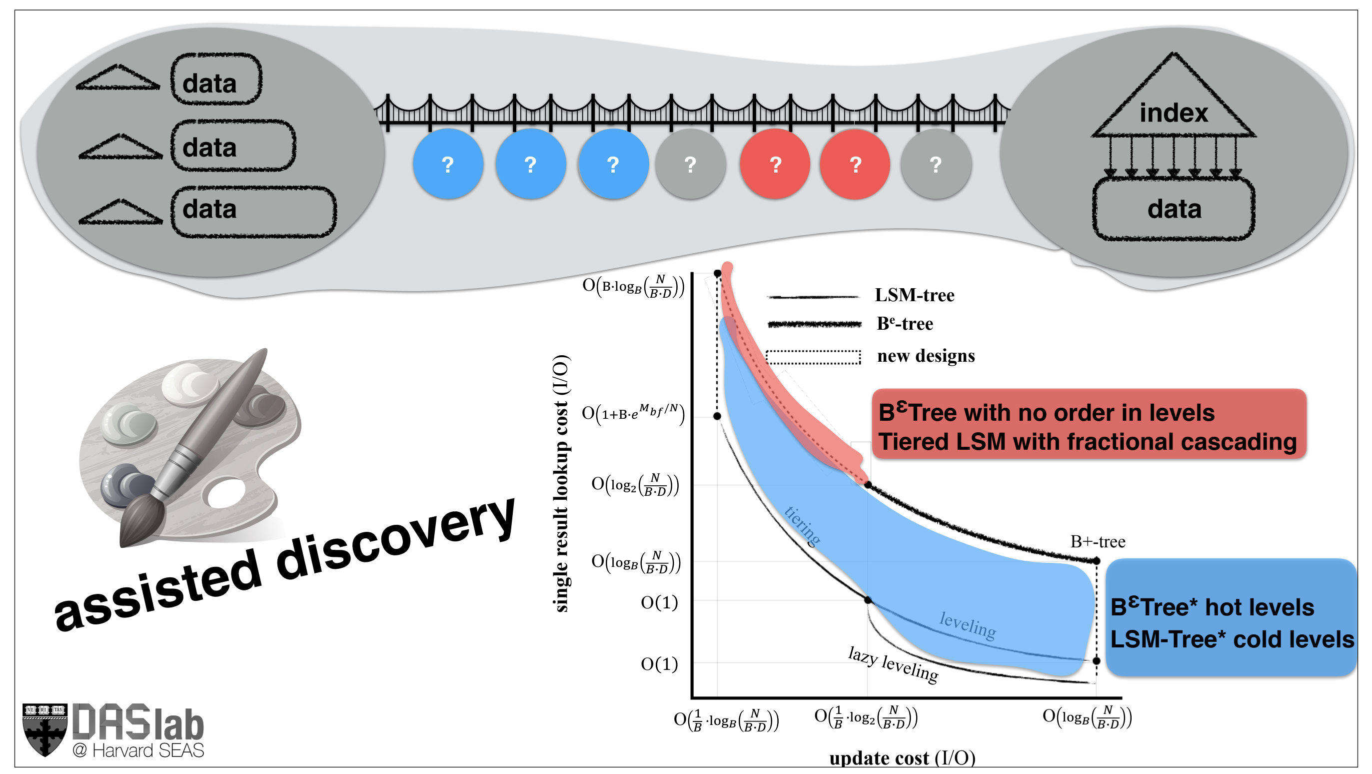

The continuum can be used to “invent” new data structures! It naturally shows us designs which are not known or published in any way in the literature but they are derived as part of the primitives in the continuum. The graph on this slide shows a performance continuum using the models of our design continuum and highlighting how  b-tree, be-tree and LSM-tree designs behave. B+tree is a single point while Be-tree and LSM-tree cover a larger area depending on their tuning and so are represented by a line. However, the continuum allows us to see many more designs and also know how they will behave. For example, we can complete the be-tree line with a design that does not have order within the tree levels. This makes it more write optimized. Similarly we can fill the area in between be-tree and LSM-tree with hybrid designs where some of the levels resemble be-tree designs and some resemble LSM-tree designs and thus providing hybrid properties which were not possible before. Now not only achieving this properties is possible but more crucially we can reliably predict what the behavior would be.



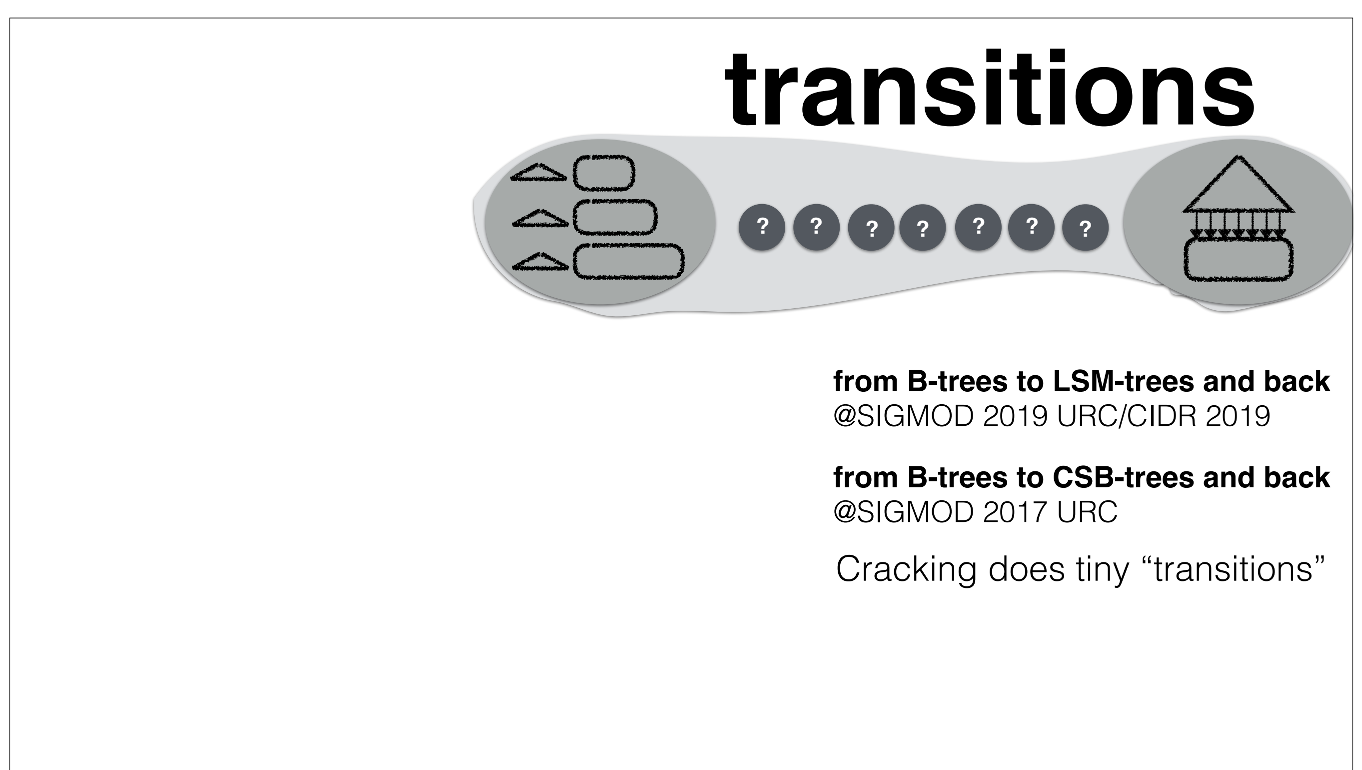

The continuum can be used to “invent” new data structures! It naturally shows us designs which are not known or published in any way in the literature but they are derived as part of the primitives in the continuum. The graph on this slide shows a performance continuum using the models of our design continuum and highlighting how  b-tree, be-tree and LSM-tree designs behave. B+tree is a single point while Be-tree and LSM-tree cover a larger area depending on their tuning and so are represented by a line. However, the continuum allows us to see many more designs and also know how they will behave. For example, we can complete the be-tree line with a design that does not have order within the tree levels. This makes it more write optimized. Similarly we can fill the area in between be-tree and LSM-tree with hybrid designs where some of the levels resemble be-tree designs and some resemble LSM-tree designs and thus providing hybrid properties which were not possible before. Now not only achieving this properties is possible but more crucially we can reliably predict what the behavior would be.



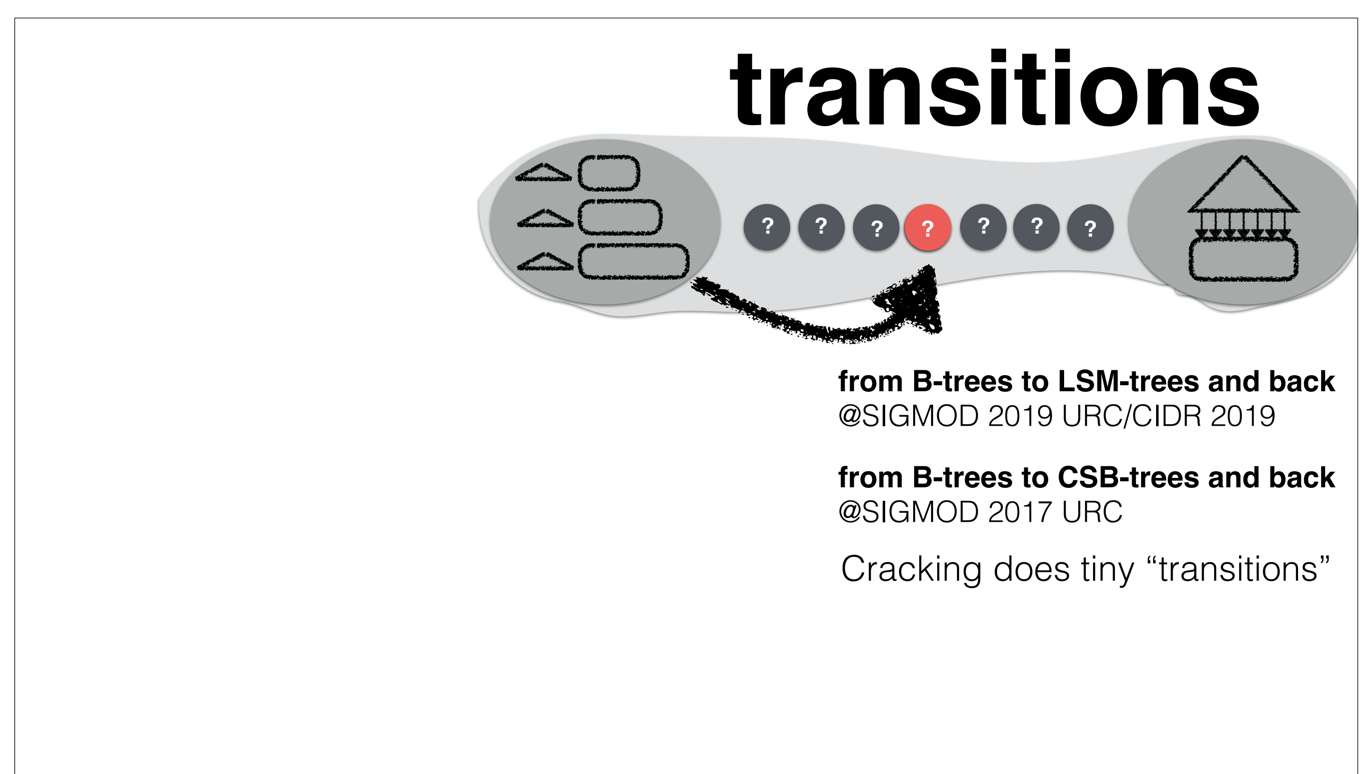

The continuum can be used to “invent” new data structures! It naturally shows us designs which are not known or published in any way in the literature but they are derived as part of the primitives in the continuum. The graph on this slide shows a performance continuum using the models of our design continuum and highlighting how  b-tree, be-tree and LSM-tree designs behave. B+tree is a single point while Be-tree and LSM-tree cover a larger area depending on their tuning and so are represented by a line. However, the continuum allows us to see many more designs and also know how they will behave. For example, we can complete the be-tree line with a design that does not have order within the tree levels. This makes it more write optimized. Similarly we can fill the area in between be-tree and LSM-tree with hybrid designs where some of the levels resemble be-tree designs and some resemble LSM-tree designs and thus providing hybrid properties which were not possible before. Now not only achieving this properties is possible but more crucially we can reliably predict what the behavior would be.



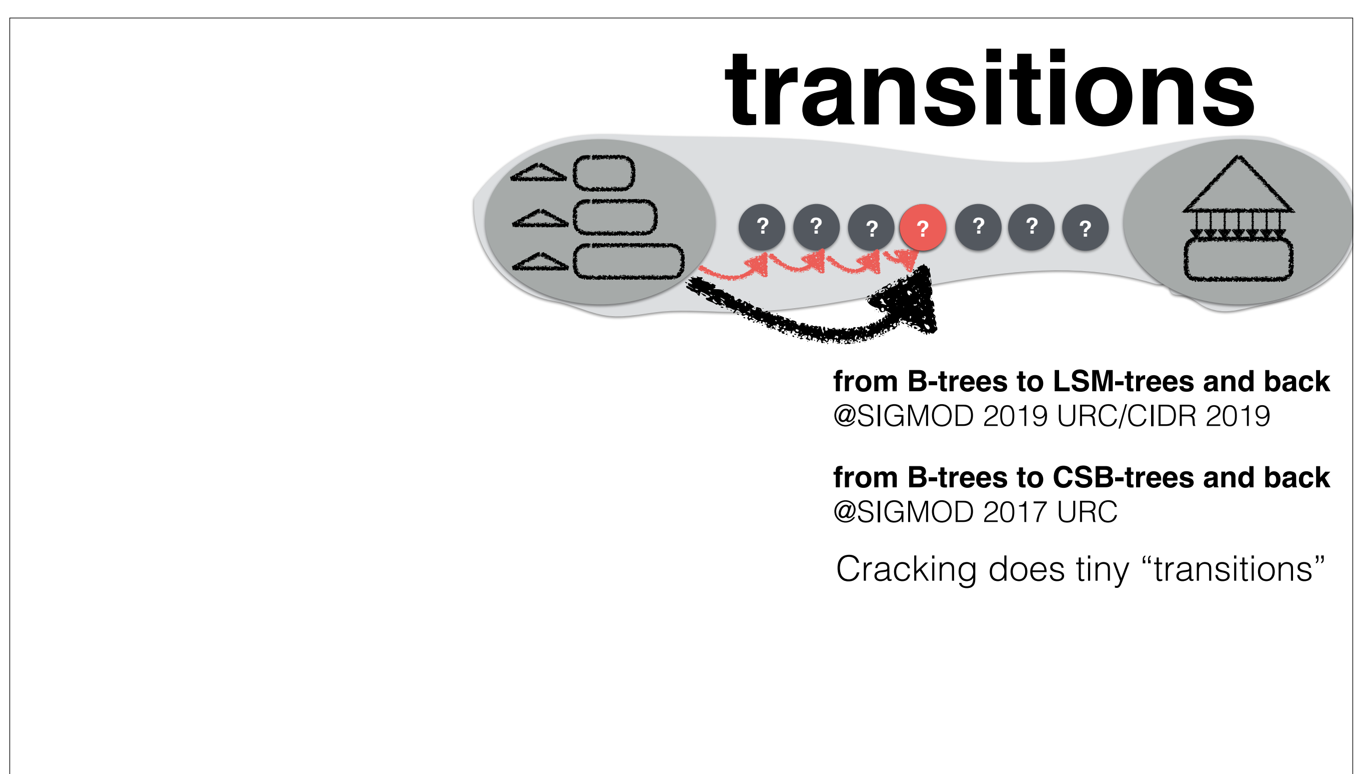

The continuum can be used to “invent” new data structures! It naturally shows us designs which are not known or published in any way in the literature but they are derived as part of the primitives in the continuum. The graph on this slide shows a performance continuum using the models of our design continuum and highlighting how  b-tree, be-tree and LSM-tree designs behave. B+tree is a single point while Be-tree and LSM-tree cover a larger area depending on their tuning and so are represented by a line. However, the continuum allows us to see many more designs and also know how they will behave. For example, we can complete the be-tree line with a design that does not have order within the tree levels. This makes it more write optimized. Similarly we can fill the area in between be-tree and LSM-tree with hybrid designs where some of the levels resemble be-tree designs and some resemble LSM-tree designs and thus providing hybrid properties which were not possible before. Now not only achieving this properties is possible but more crucially we can reliably predict what the behavior would be.



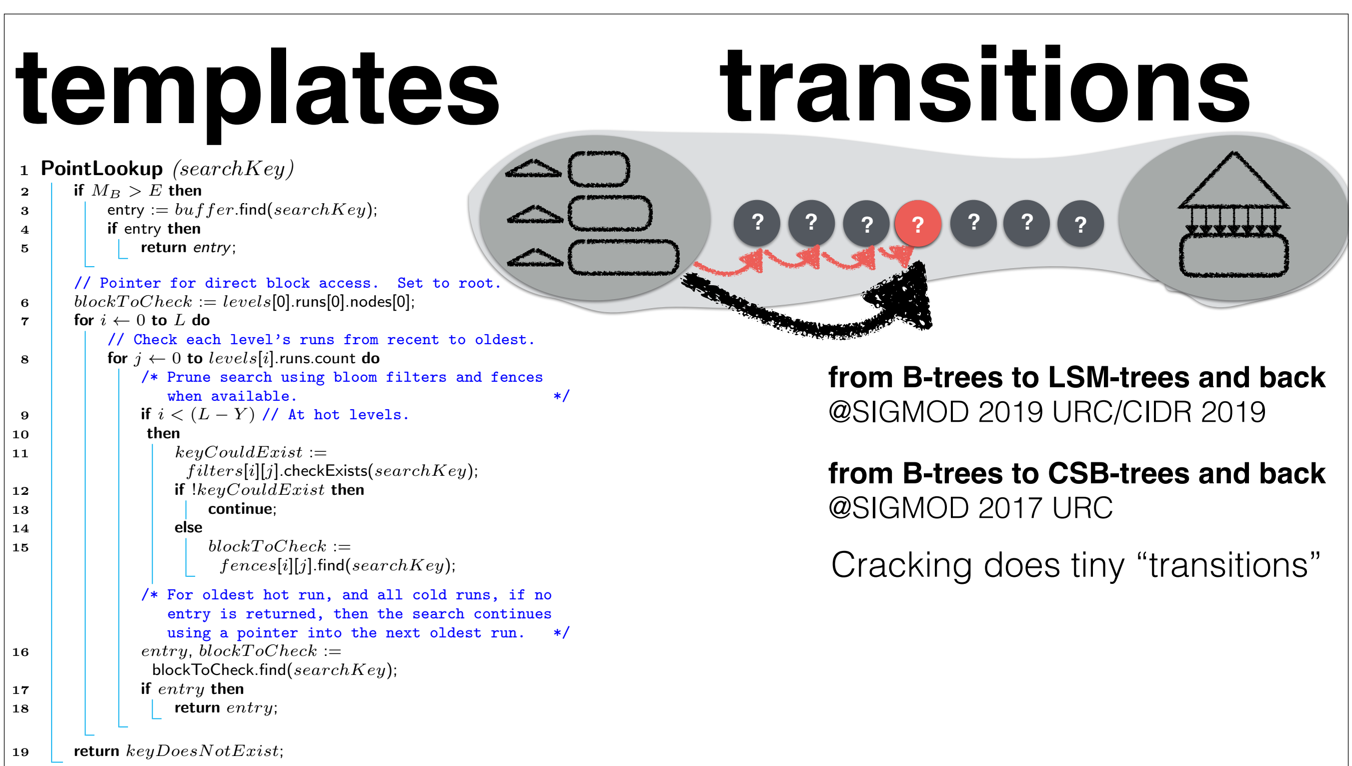

The fact that all designs in the continuum are represented effectively by the same super structure, means that we can write templates that can instantiate any of the structures, leading to efficient implementations that can be derived automatically. Early results of this idea shows the potential.



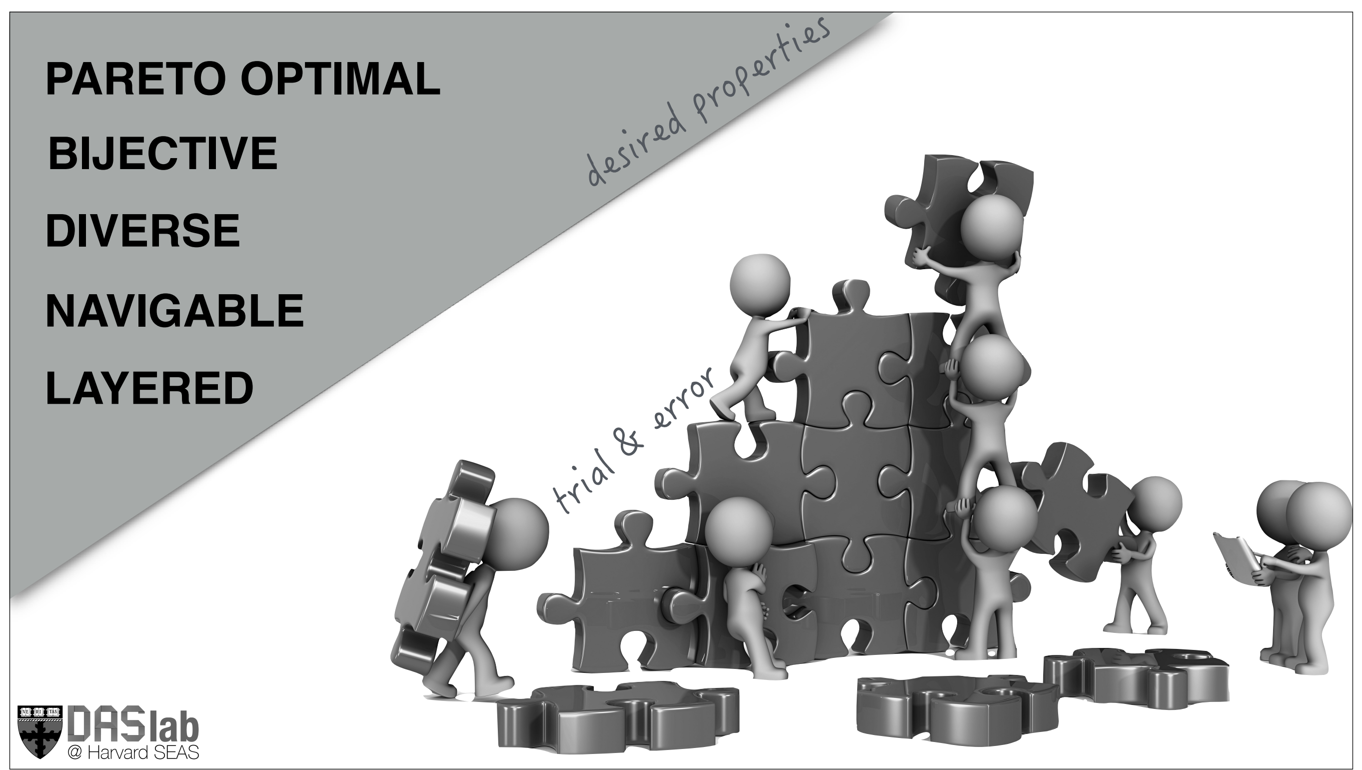

The fact that all designs in the continuum are represented effectively by the same super structure, means that we can write templates that can instantiate any of the structures, leading to efficient implementations that can be derived automatically. Early results of this idea shows the potential.



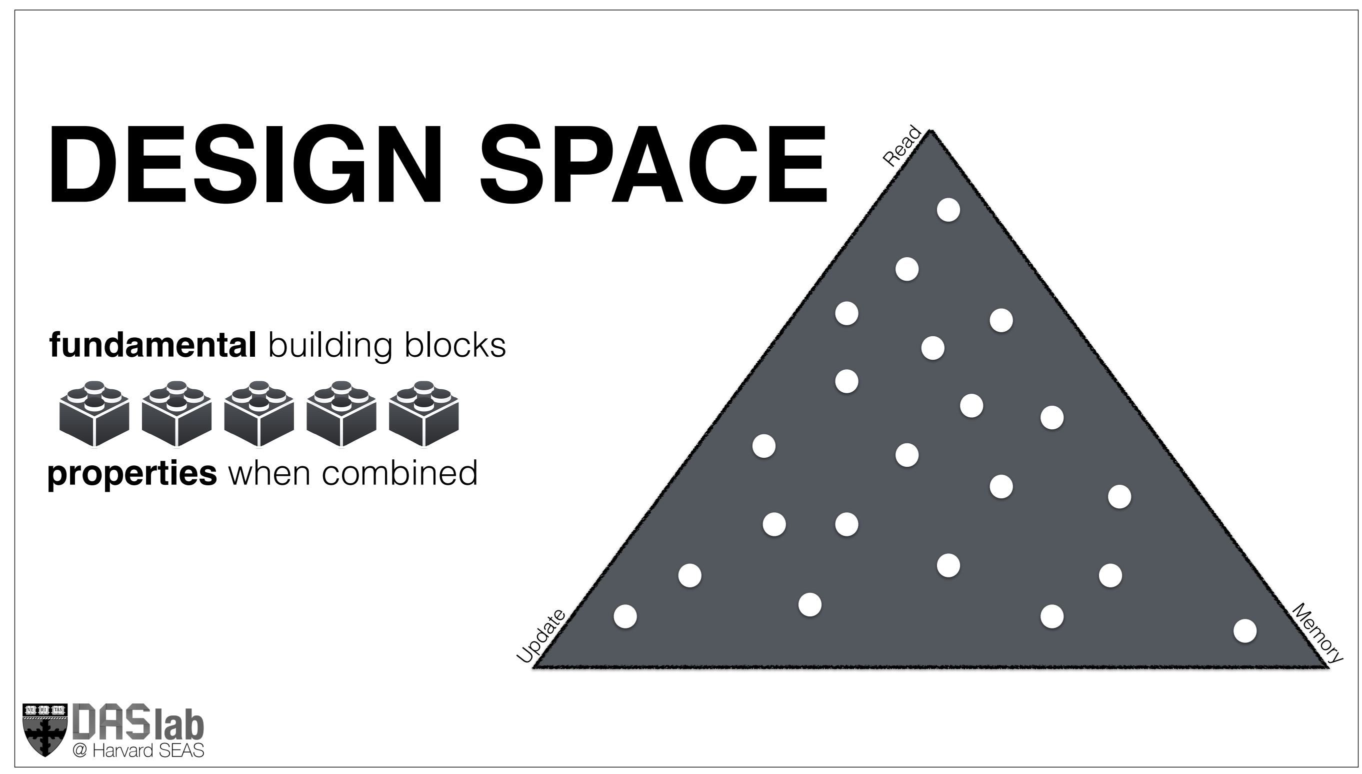

The fact that all designs in the continuum are represented effectively by the same super structure, means that we can write templates that can instantiate any of the structures, leading to efficient implementations that can be derived automatically. Early results of this idea shows the potential.



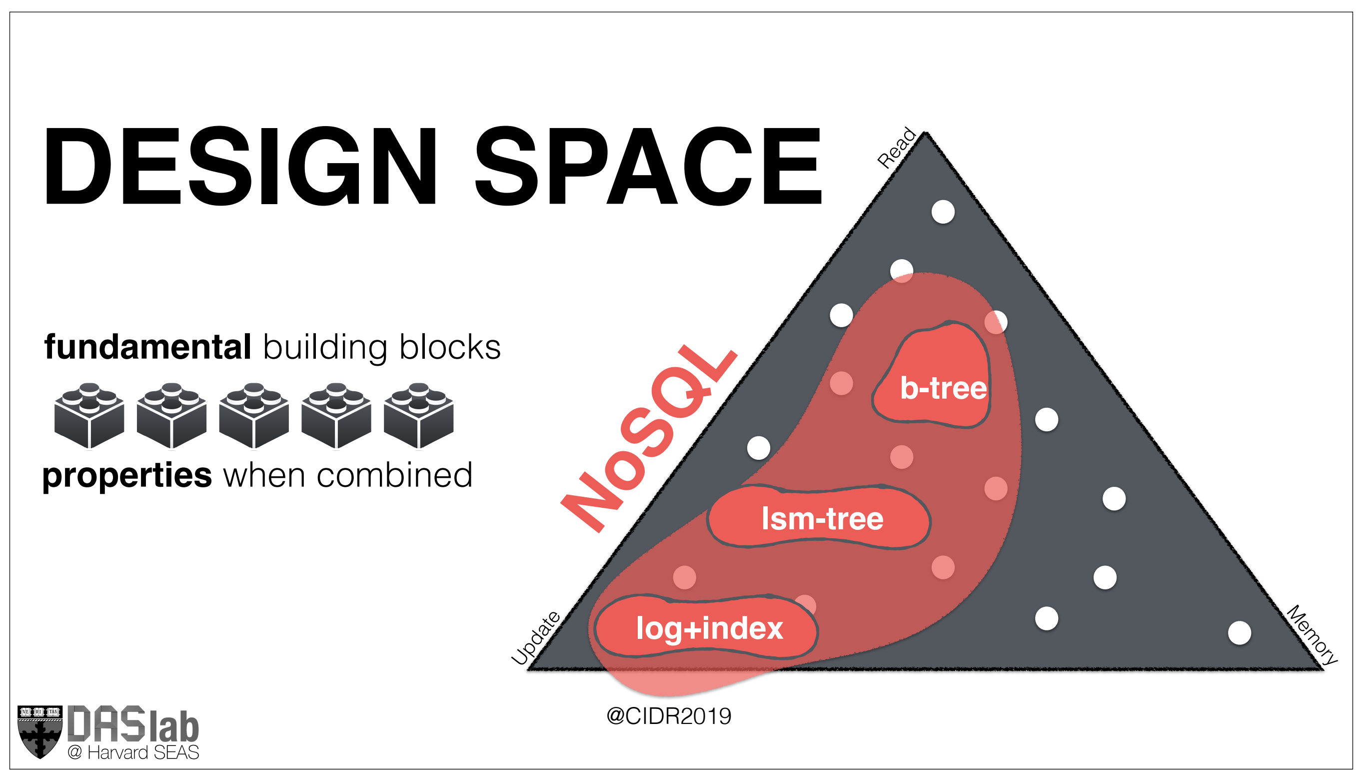

The fact that all designs in the continuum are represented effectively by the same super structure, means that we can write templates that can instantiate any of the structures, leading to efficient implementations that can be derived automatically. Early results of this idea shows the potential.



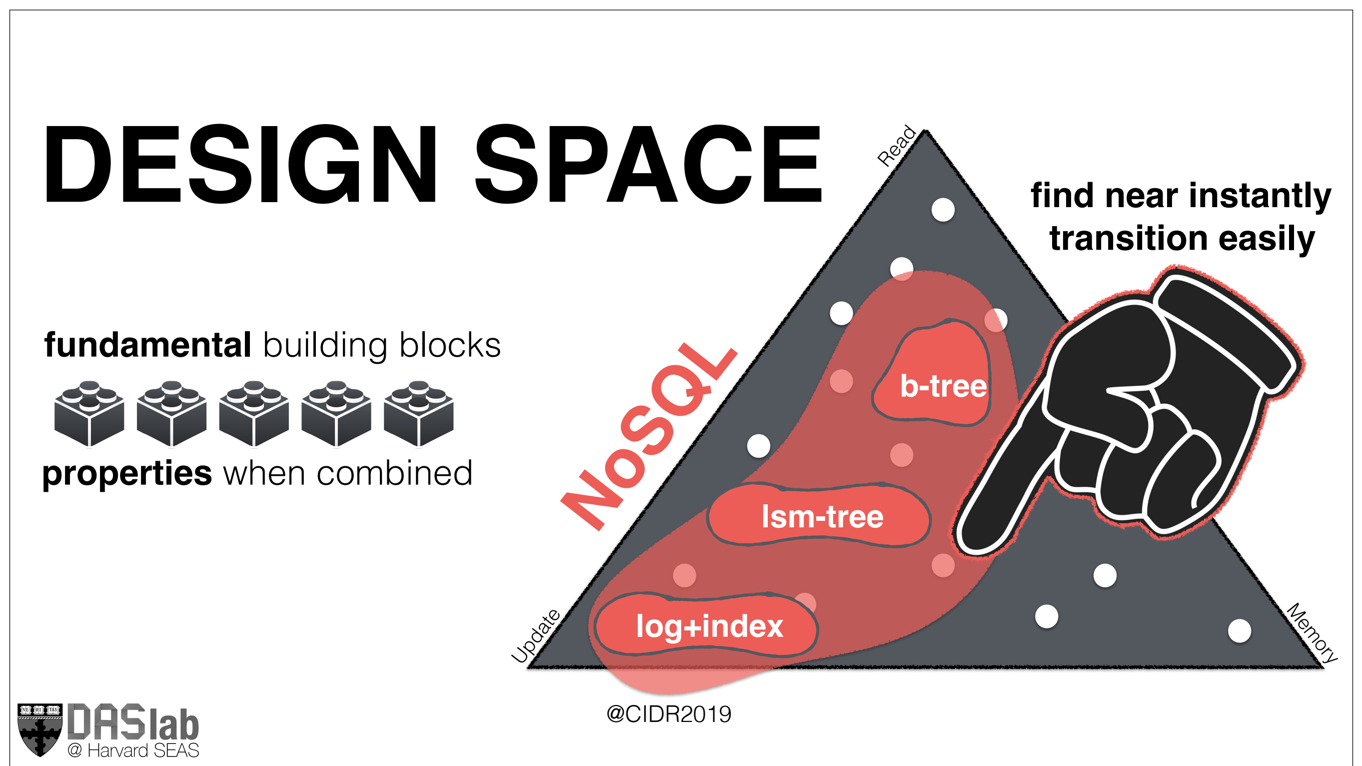

Properties that a design continuums should have. It should contain only designs that are ideally optimal for a given scenario or in other words a continuum should not contain designs that are never optimal for a workload because then we make it more complex without any gain. Second there should be only a single mapping from a workload to a design. Third, at the same time a useful continuum contains diverse designs that can span diverse performance properties. Fourth, we should keep the structure of the continuum in a form that is possible to navigate with simple rules. Finally, when the previous point is not possible, we can create layers in the continuum where we can make optimal decisions locally in an area before moving to the next layer of design decisions. This last part is not for free: we do lose the ability to represent certain designs, but navigability is essential for a design continuum to be useful.



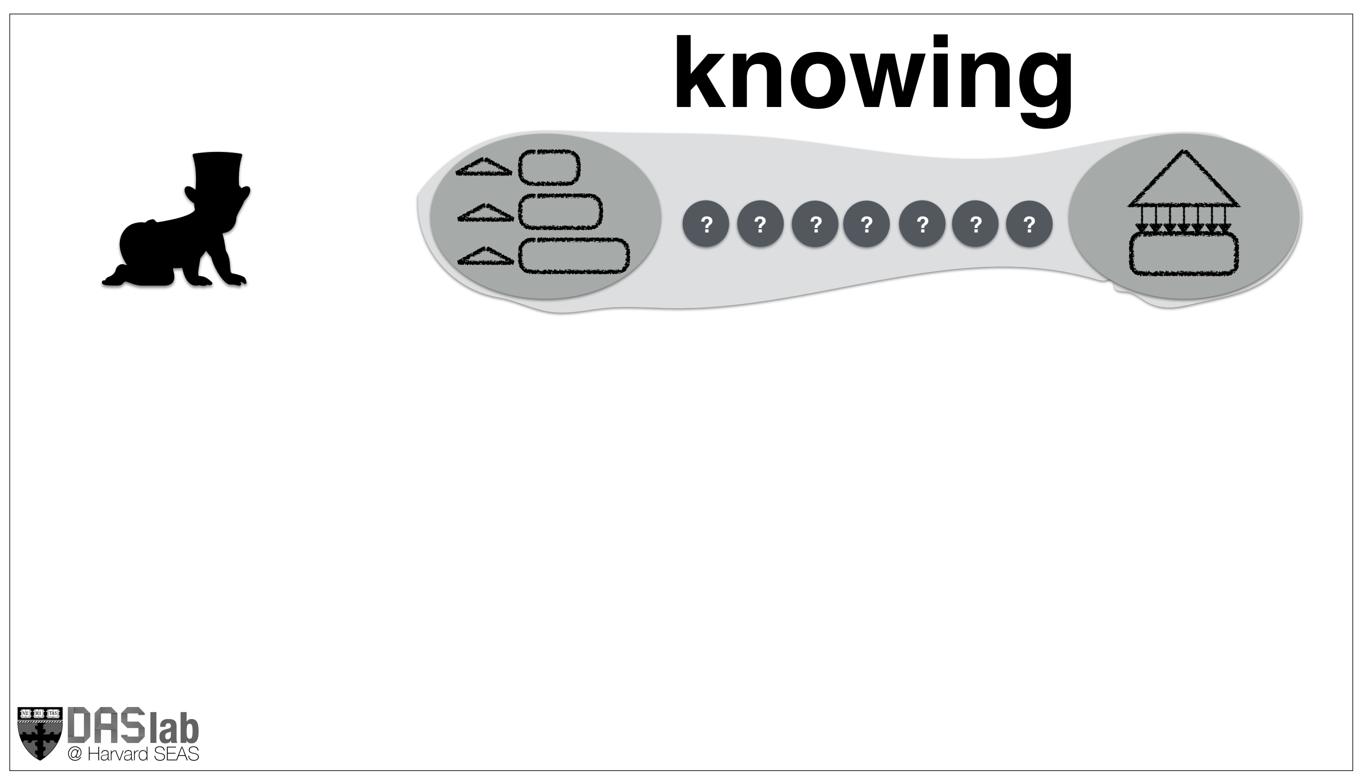

Overall the first design continuum spanning core key-value store data structures means that we can now search instantly within this critical design space.



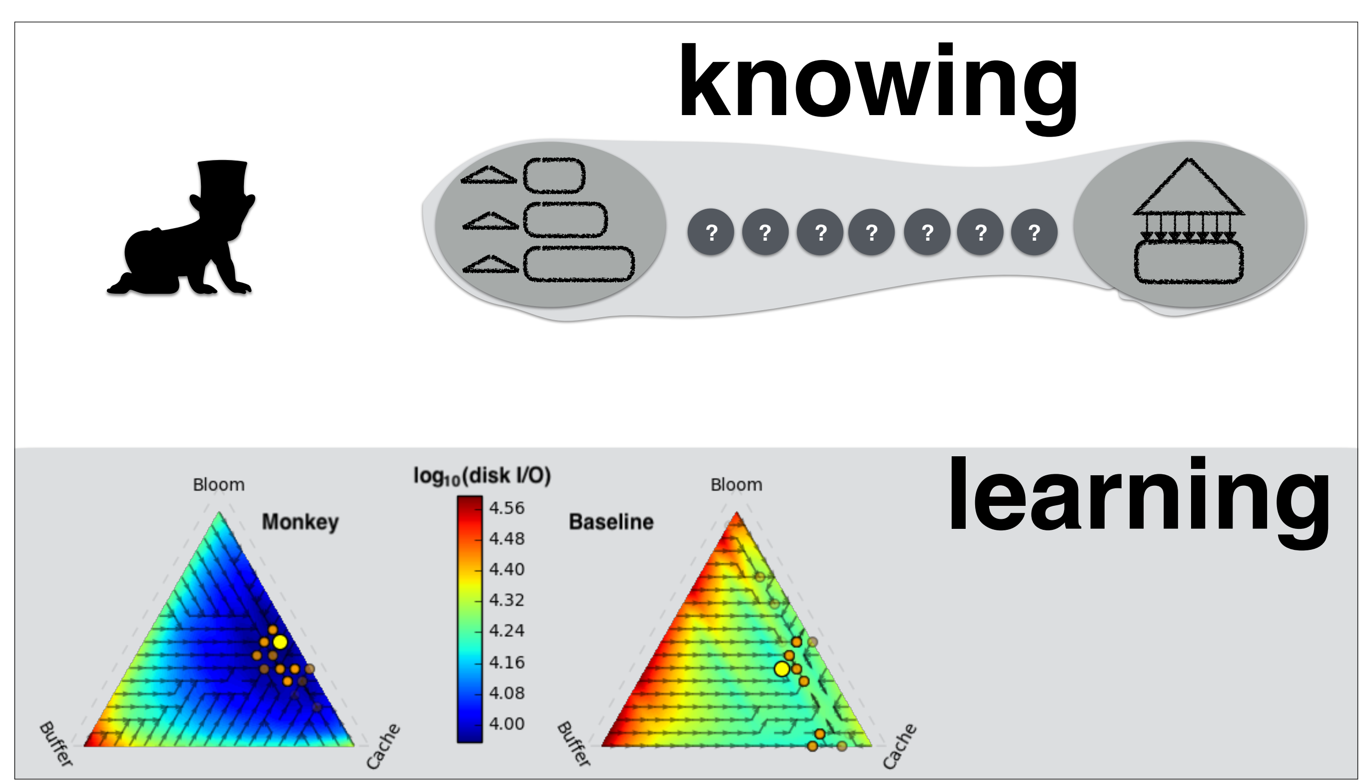

Overall the first design continuum spanning core key-value store data structures means that we can now search instantly within this critical design space.



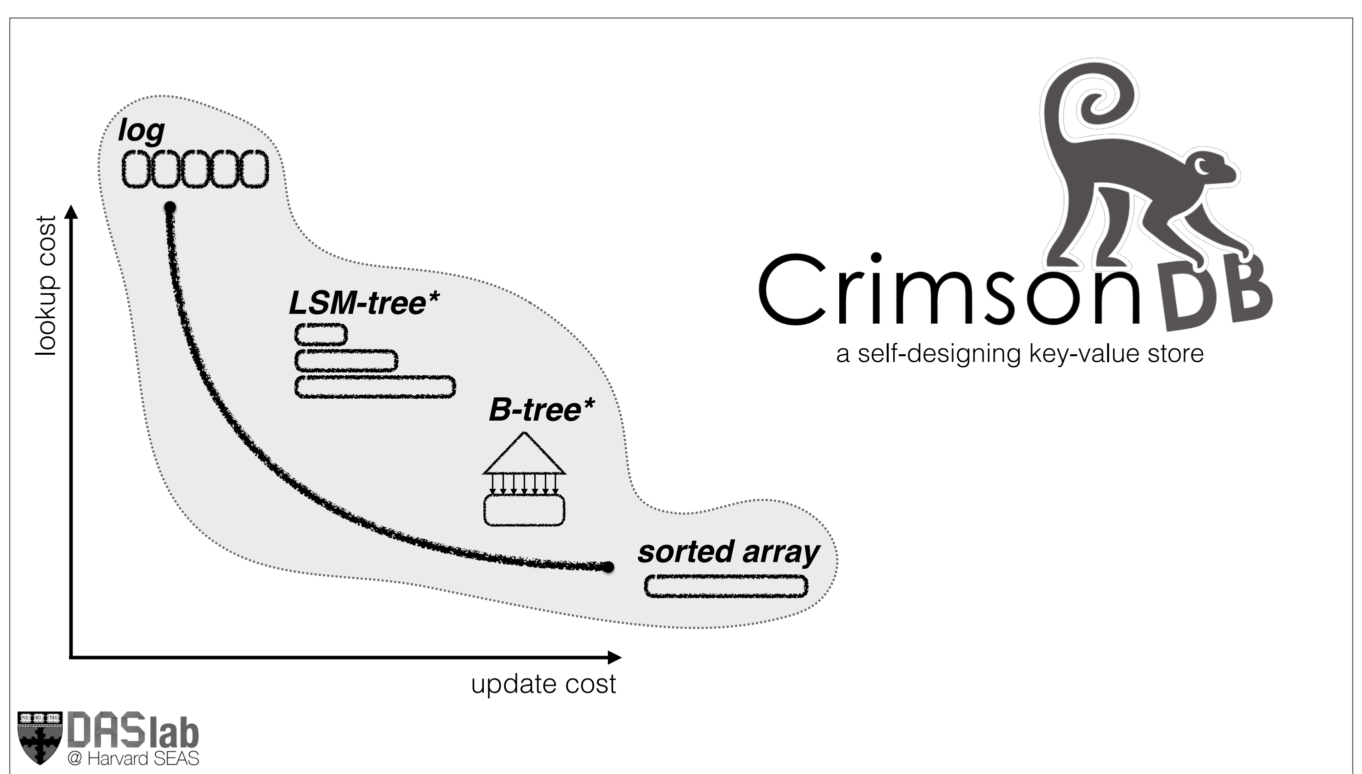

Overall the first design continuum spanning core key-value store data structures means that we can now search instantly within this critical design space.



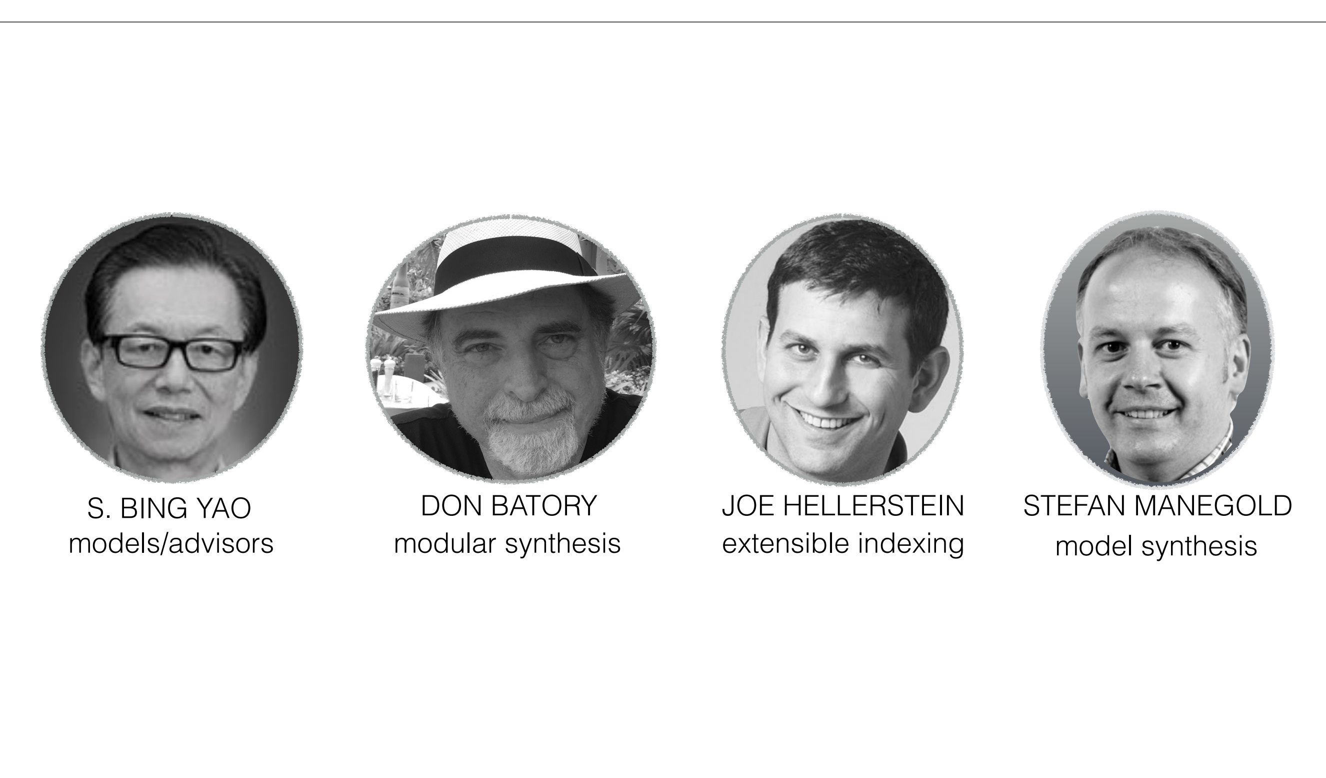

There is a limit to what we can actually represent with models. We do not know where the limit really is. But there seems to be one…If we cannot represent some critical design decisions with a closed form model, then we can fall back to learning which is an essential tool to complete complex designs. One such decision is how much cache we should use in the key-value store design. While in the first continuum we demonstrated that we can make concrete decisions based on models on critical memory components such as buffer size, bit per entry for the bloom filters and fence pointers, the last memory component of how much cache we should use is extremely hard to capture accurately its behavior in a closed form model. At least we have not been successful yet. The reason is the strong dependency on workload. On the other hand, we can reasonably simply learn a model for how much memory we should devote to the cache! And then spread the rest of the memory to the rest of the in-memory components using the continuum.




There is a limit to what we can actually represent with models. We do not know where the limit really is. But there seems to be one…If we cannot represent some critical design decisions with a closed form model, then we can fall back to learning which is an essential tool to complete complex designs. One such decision is how much cache we should use in the key-value store design. While in the first continuum we demonstrated that we can make concrete decisions based on models on critical memory components such as buffer size, bit per entry for the bloom filters and fence pointers, the last memory component of how much cache we should use is extremely hard to capture accurately its behavior in a closed form model. At least we have not been successful yet. The reason is the strong dependency on workload. On the other hand, we can reasonably simply learn a model for how much memory we should devote to the cache! And then spread the rest of the memory to the rest of the in-memory components using the continuum.




We are now building a key-value store system using these principles. CrimsonDB can “shape” itself to resemble arbitrary data structure designs to much workload and hardware.




While there are several people who have influenced this work there are 4 that stand out for the creative and ground breaking work that pushed the limits of data structure design and put the foundations to be able to think about automated design. S. Bing Yao was the first person to think about storage advisors and already in the 70s built elaborate hardware conscious generalized models with which we could choose among many candidate designs. Don Batory has been the pioneer of modular systems across many fields and among other things pushed the agenda on high level languages for data structure design backed by a system that would synthesize arbitrary designs. Joe Hellerstein worked on GiST which effectively was a templated data structure platform that could be instantiated in many forms and be very easily extended to support new data types. Stefan Manegold was the first person to work on generalized models for the much harder problem of in-memory processing and showed the first signs that we can actually synthesize the cost of complex algorithms (database operators in this case) from simpler components that we learn by running a simple set of experiments on the desired hardware.

Lots and lots of wonderful students have worked in this line of ideas over the last 5-6 years.

SESSION 2

**Learned Algorithms and Data Structures**

models replacing traditional designs

model benefits in storage & response time

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

SESSION 2

**Learned Algorithms and Data Structures**

models replacing traditional designs

model benefits in storage & response time

THANKS!