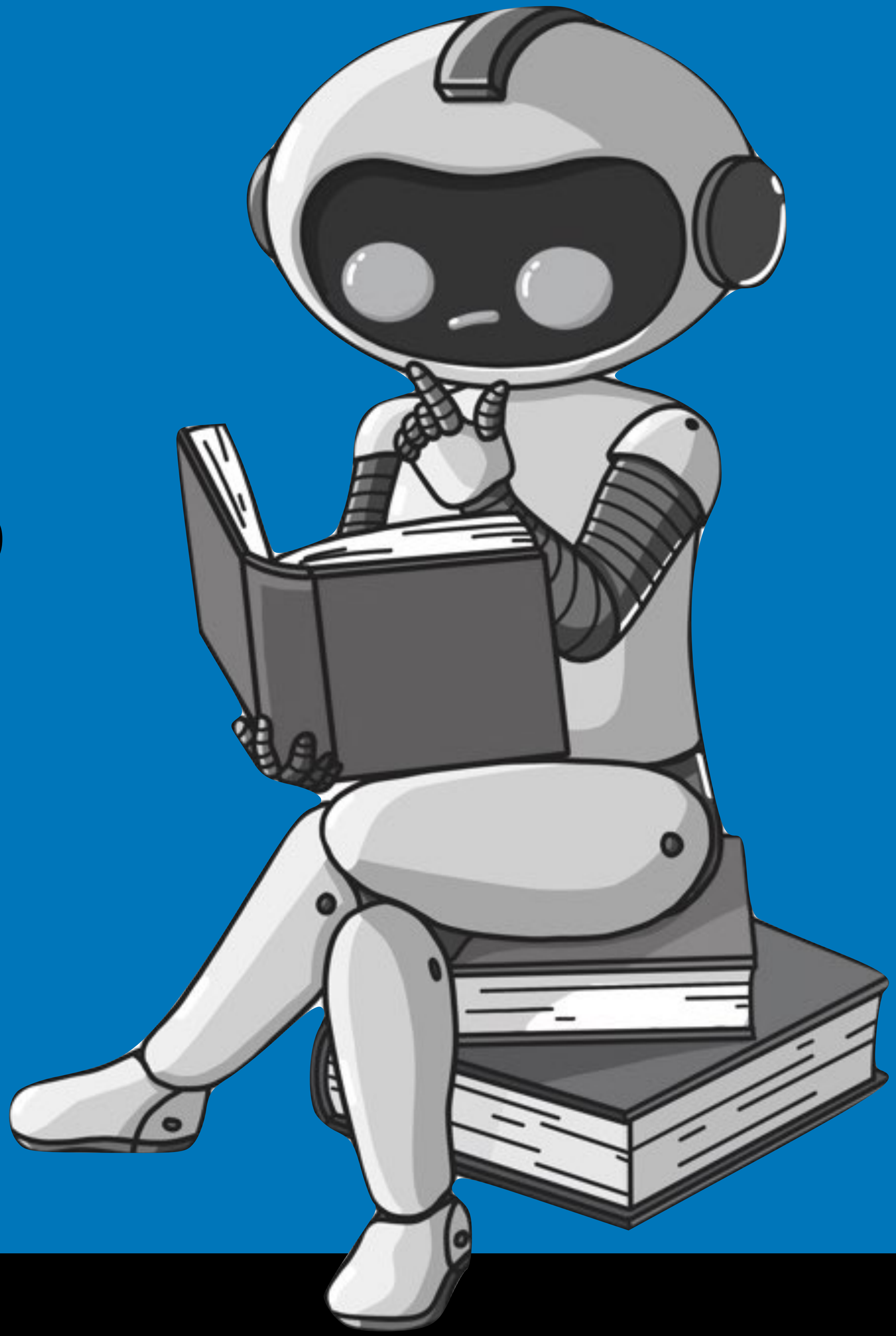
ALPHABETS, GRAMMARS, CALCULATORS, AND THE END OF HAND-CRAFTED SYSTEMS

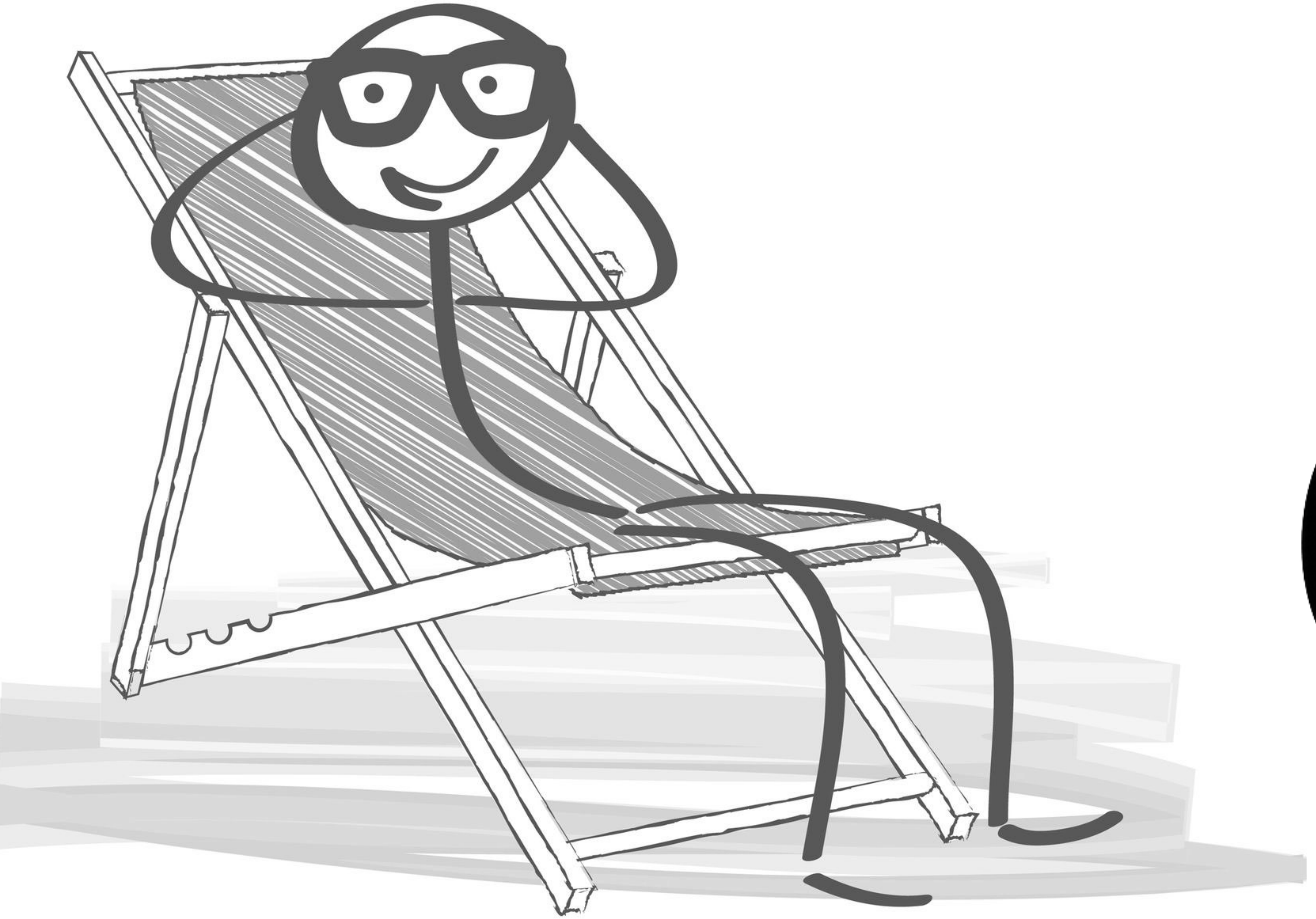
Stratos Idreos

DASlab @ Harvard SEAS



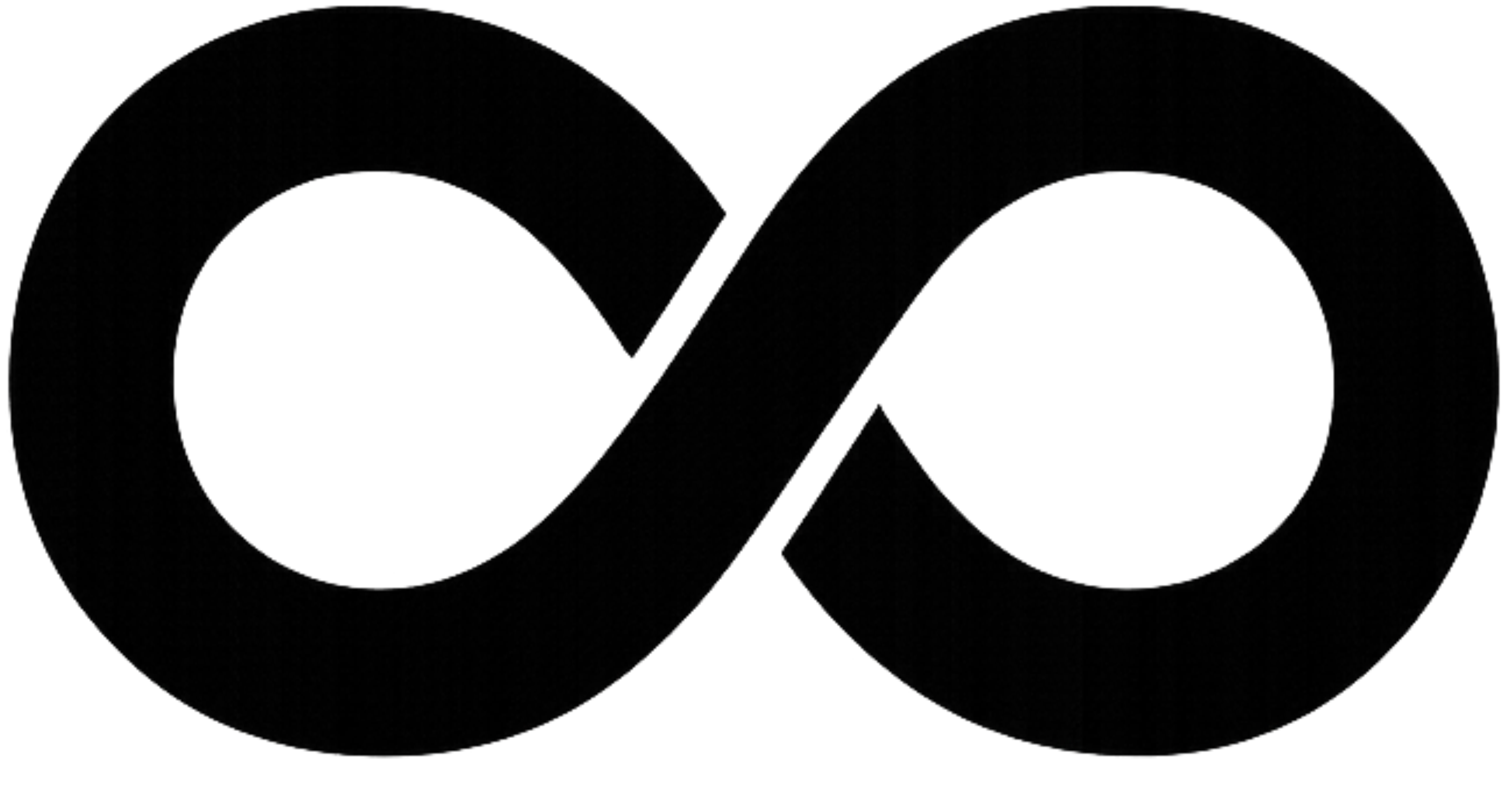
*What if we can **reason** about systems design?*

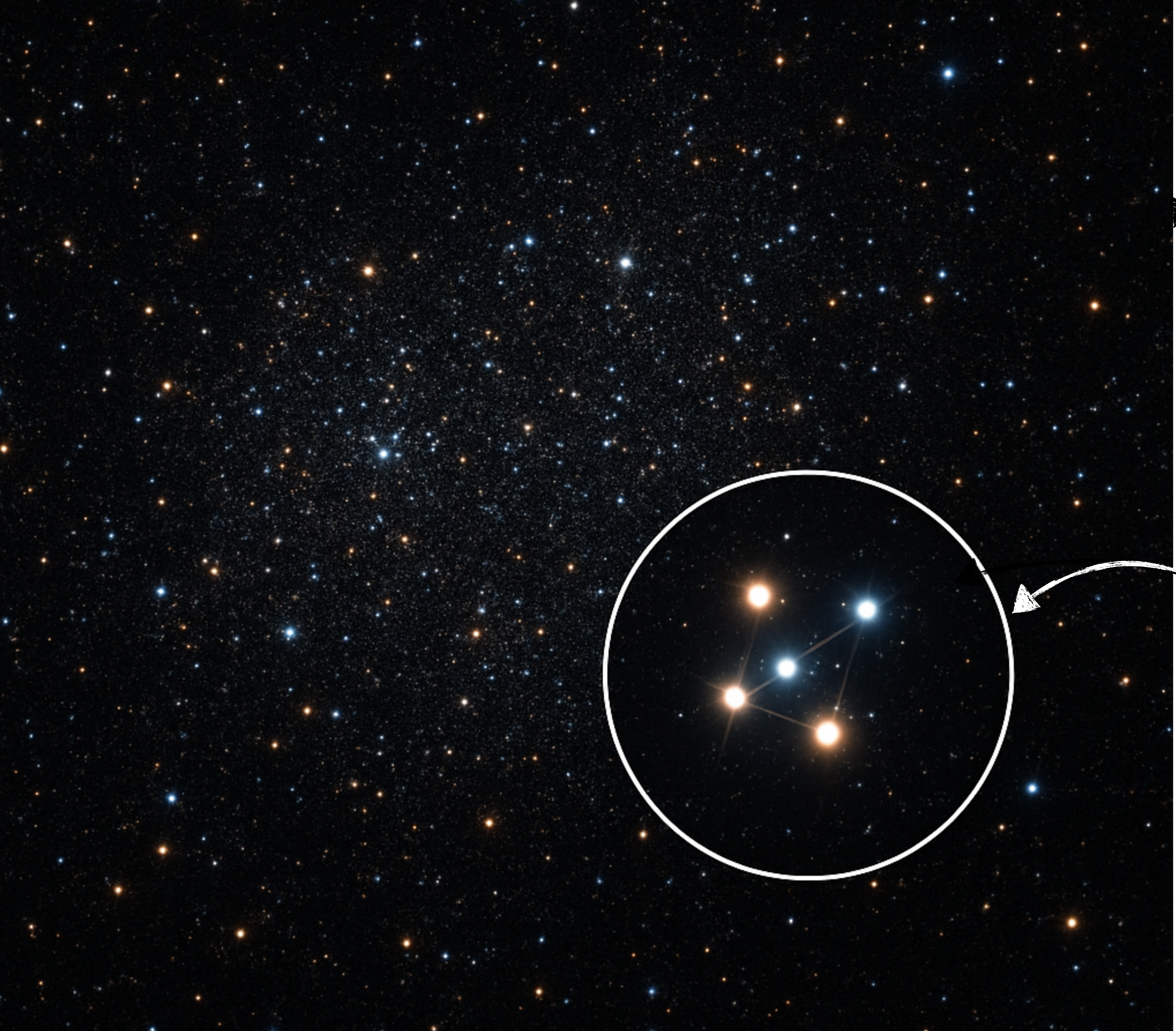
IS THIS GOING TO REPLACE RESEARCHERS?



NOPE!

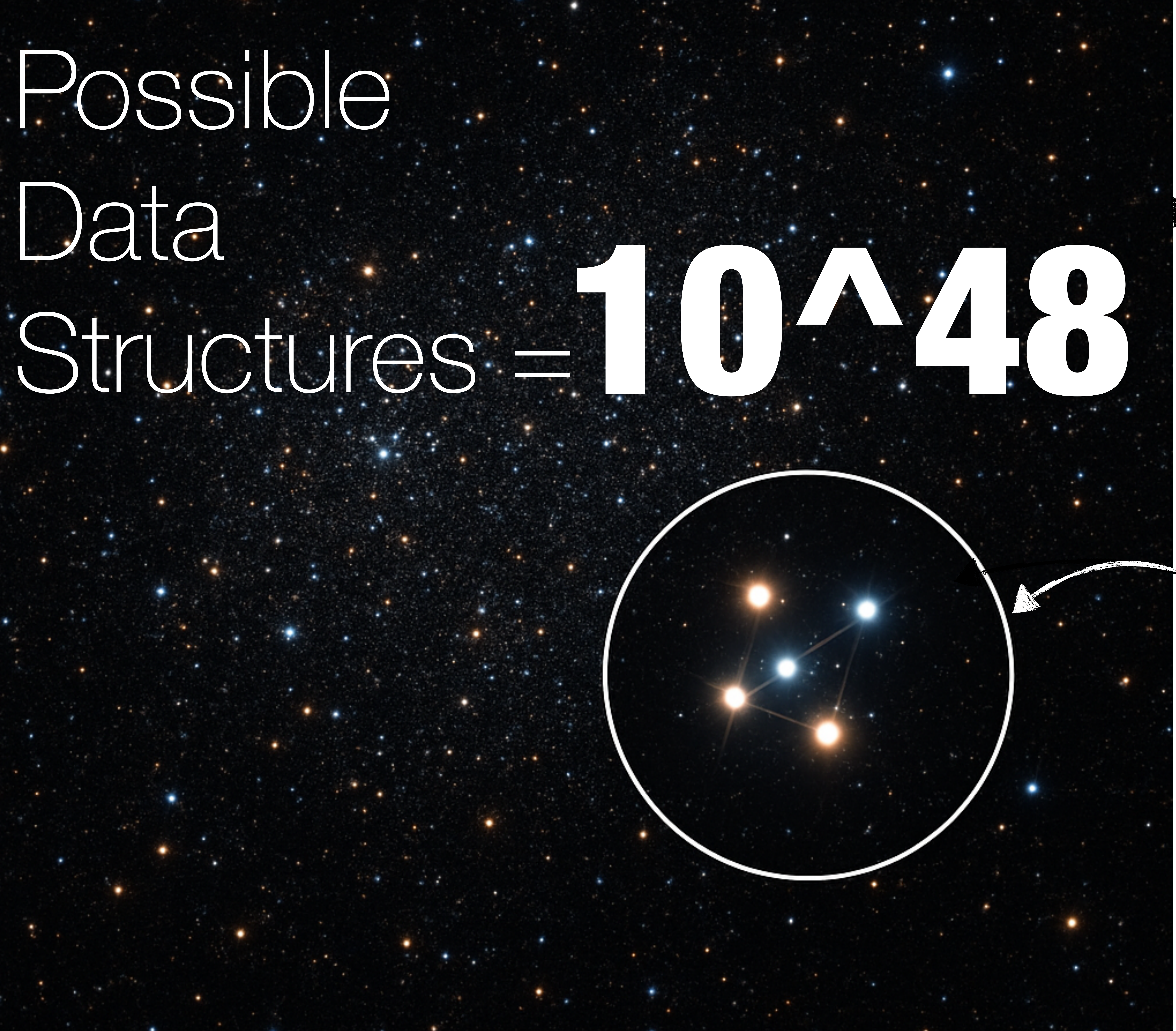
Infinite Research Topics





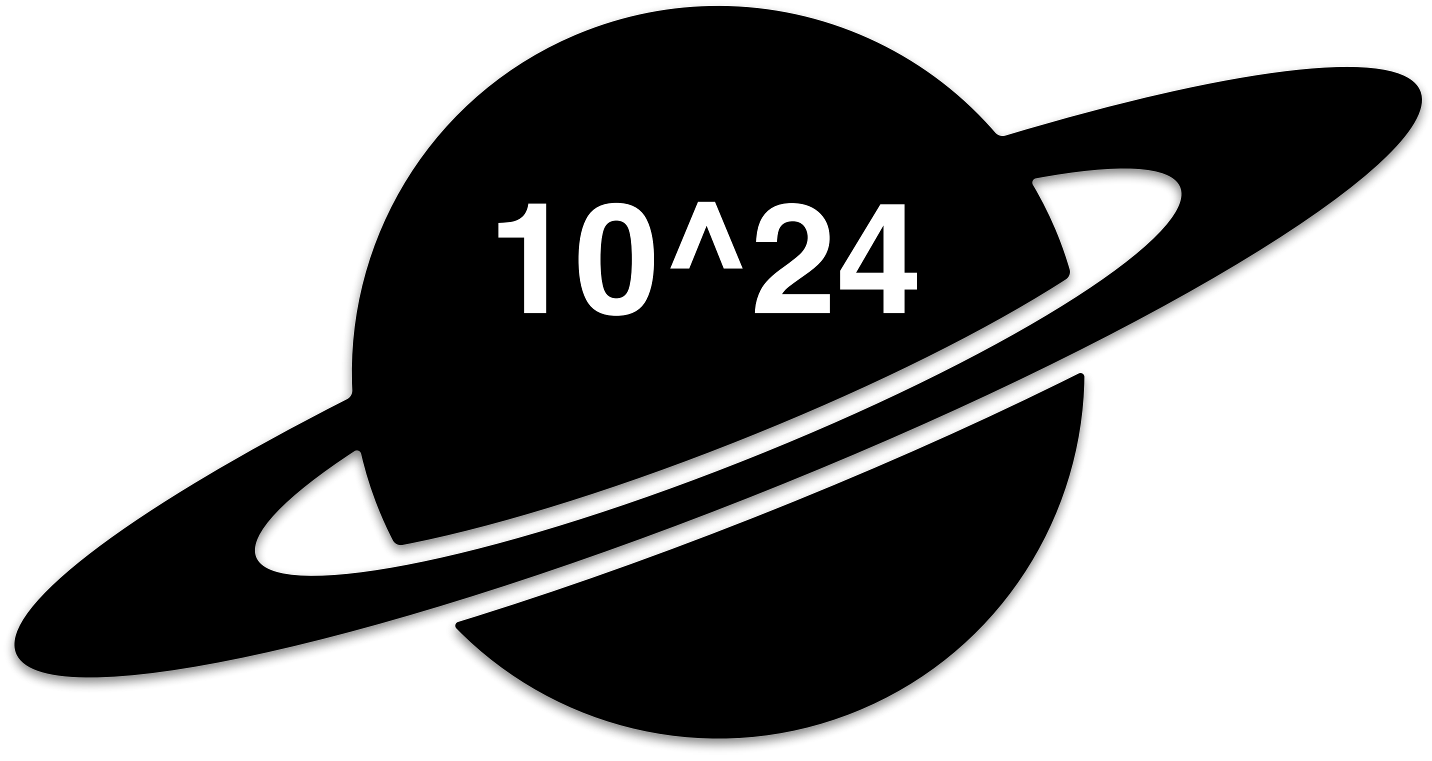
POSSIBLE SYSTEM DESIGNS

WHAT WE HAVE INVENTED SINCE THE DAWN OF COMPUTER SCIENCE

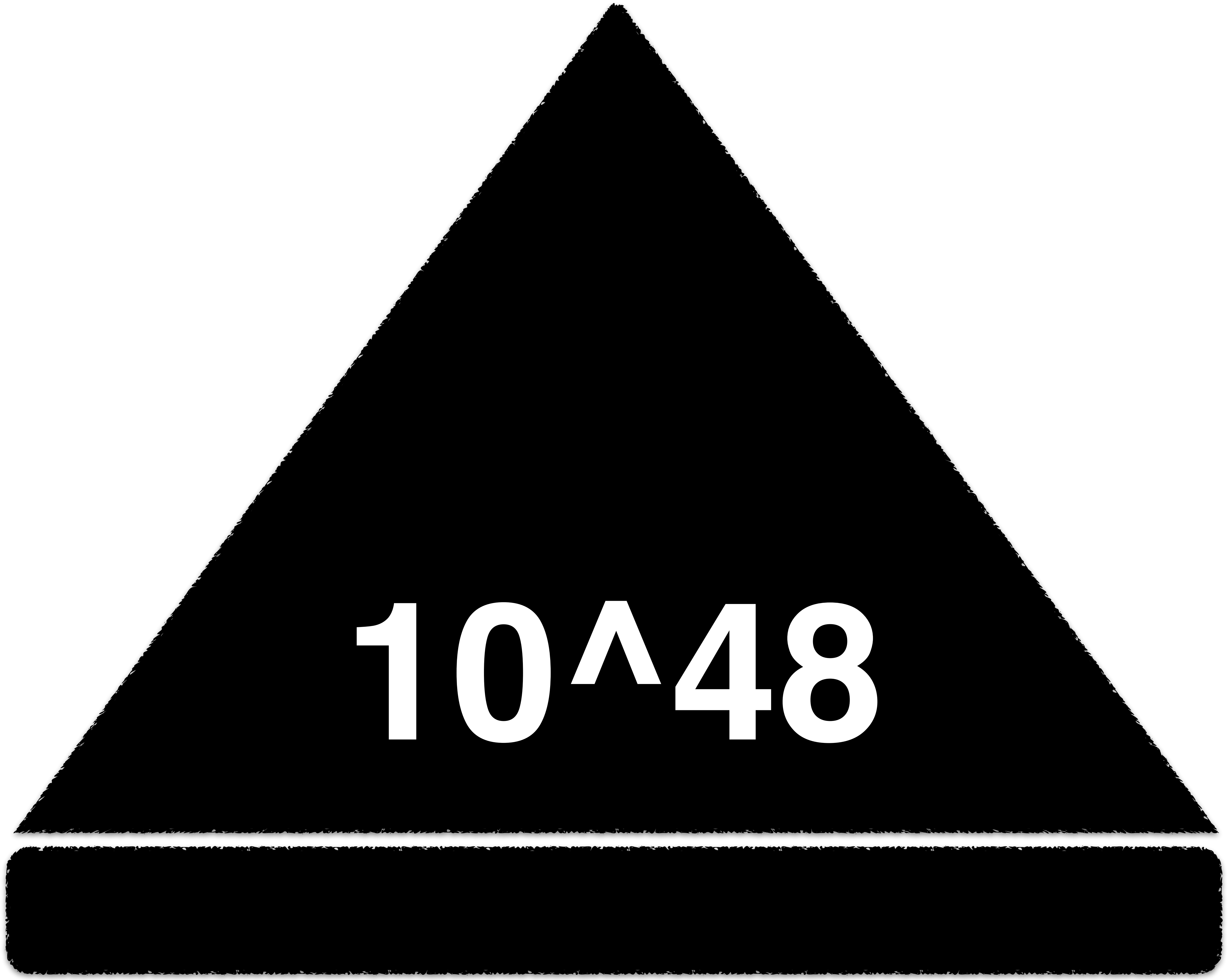


POSSIBLE
SYSTEM
DESIGNS

WHAT WE HAVE
INVENTED SINCE
THE DAWN OF
COMPUTER
SCIENCE

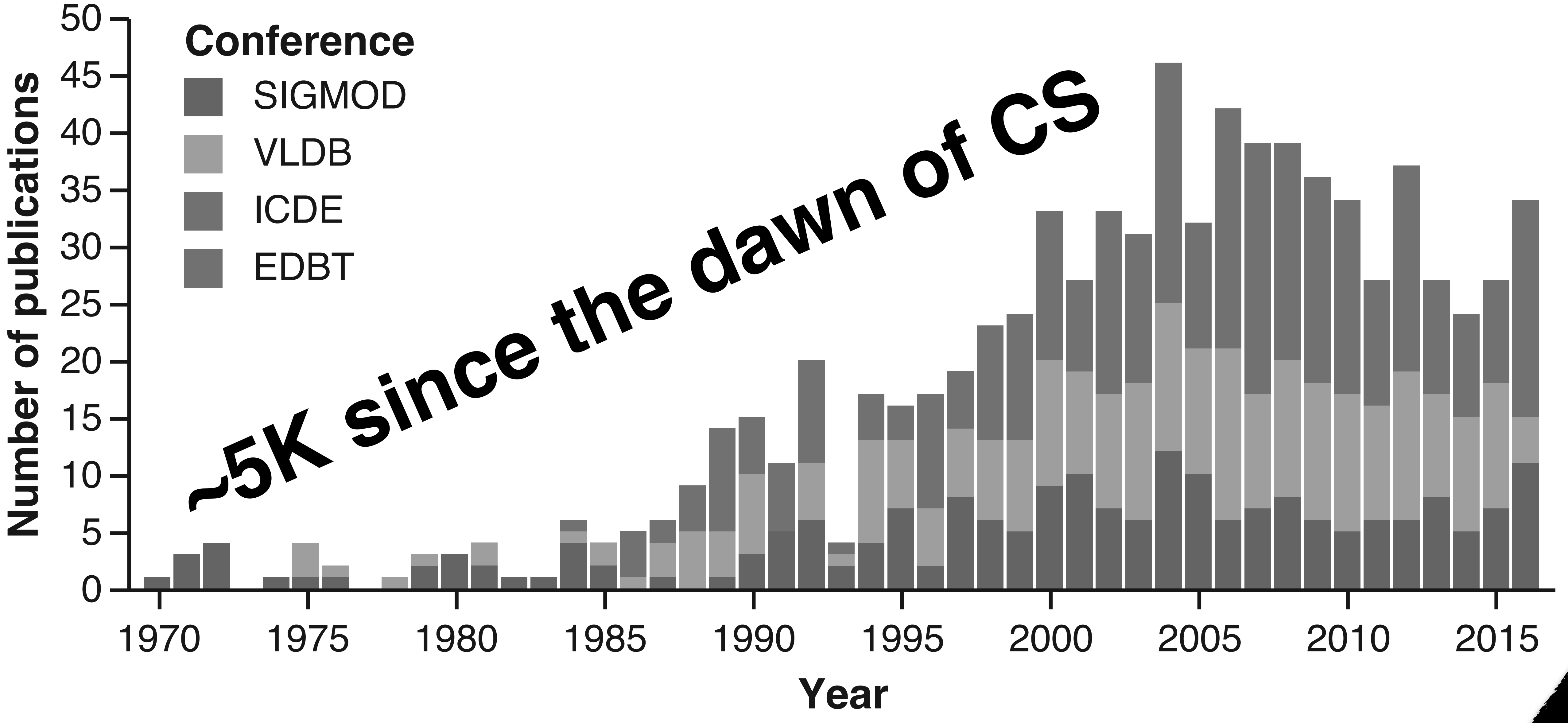


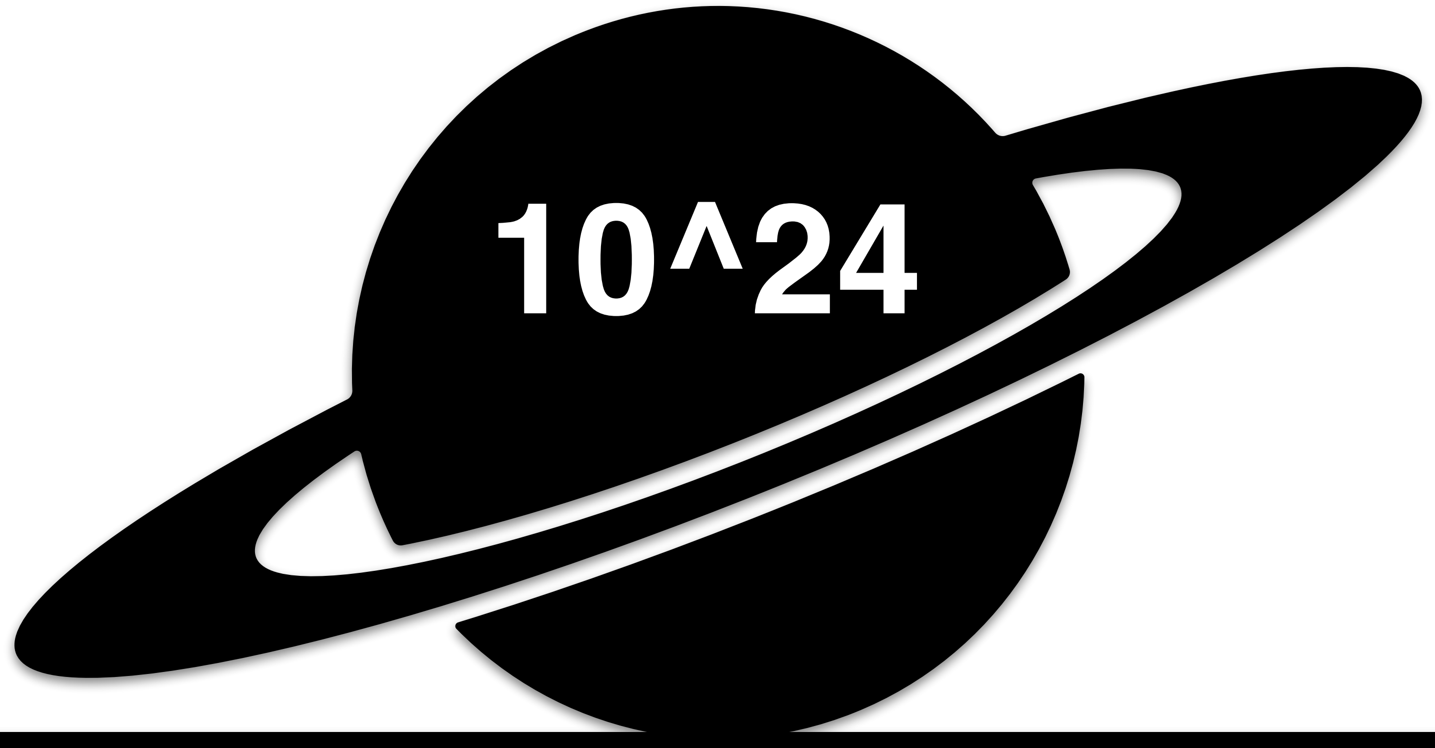
STARS ON THE SKY

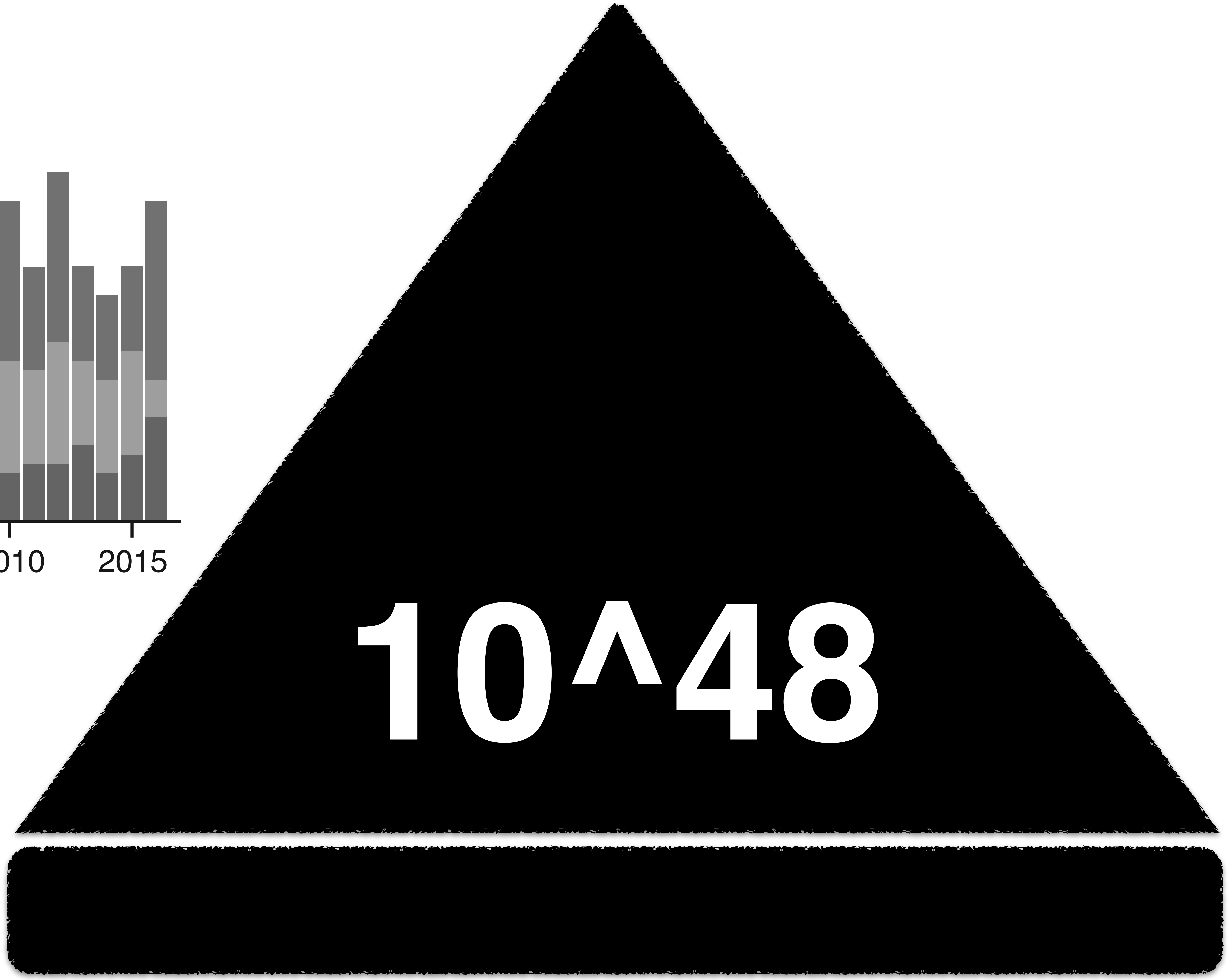


POSSIBLE DATA STRUCTURES

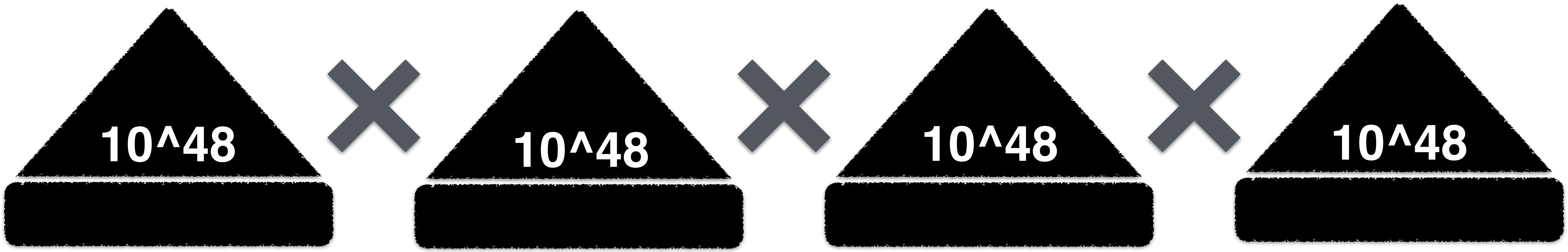
DASlab
@ Harvard SEAS







$10^{48} - 5 \times 10^{3} = 10^{48}$ **zero progress**



Key-value Stores/Databases/ML Systems

>10^100

the dinning
systems designers

phd

phd

phd

phd

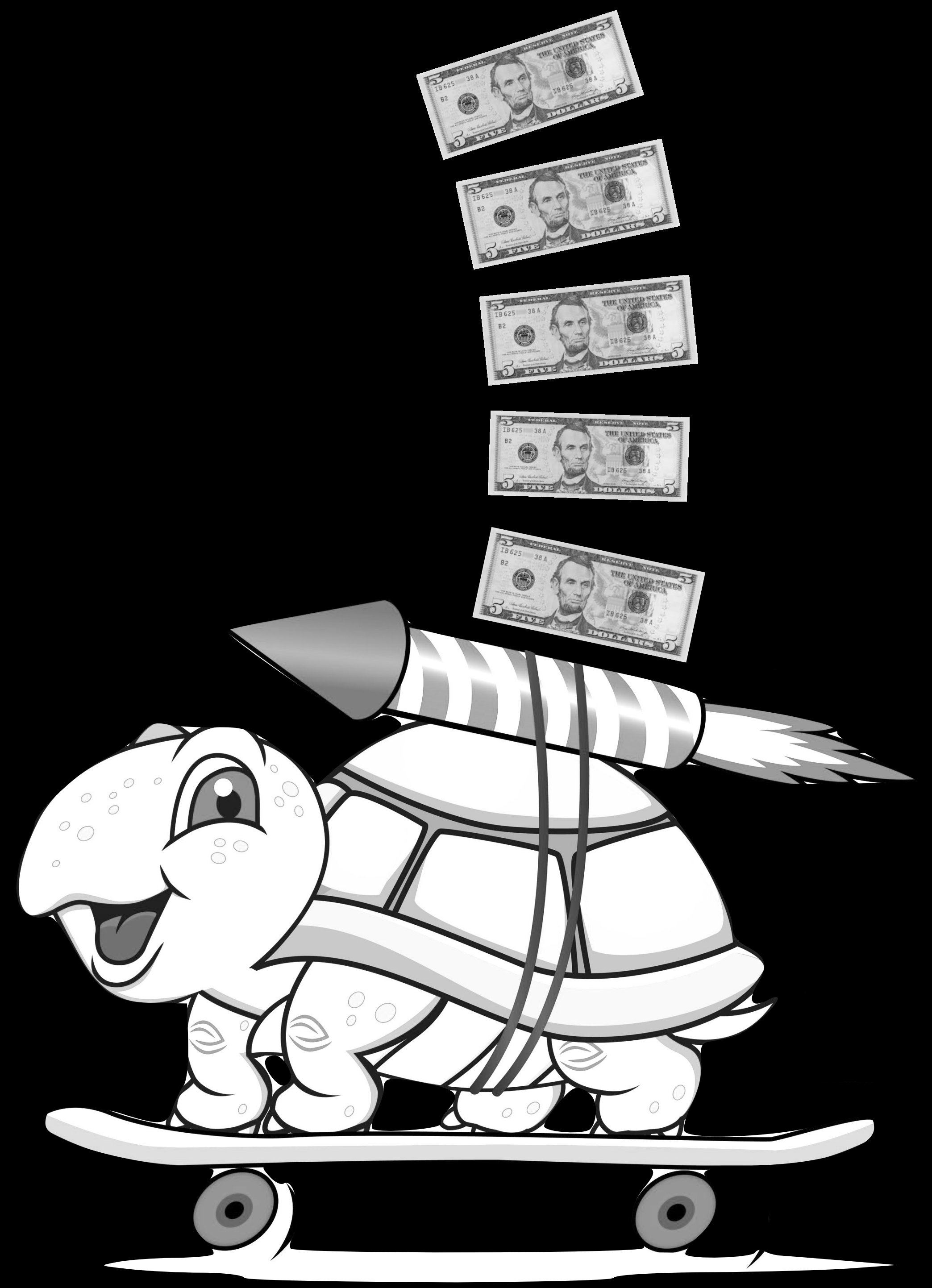
phd

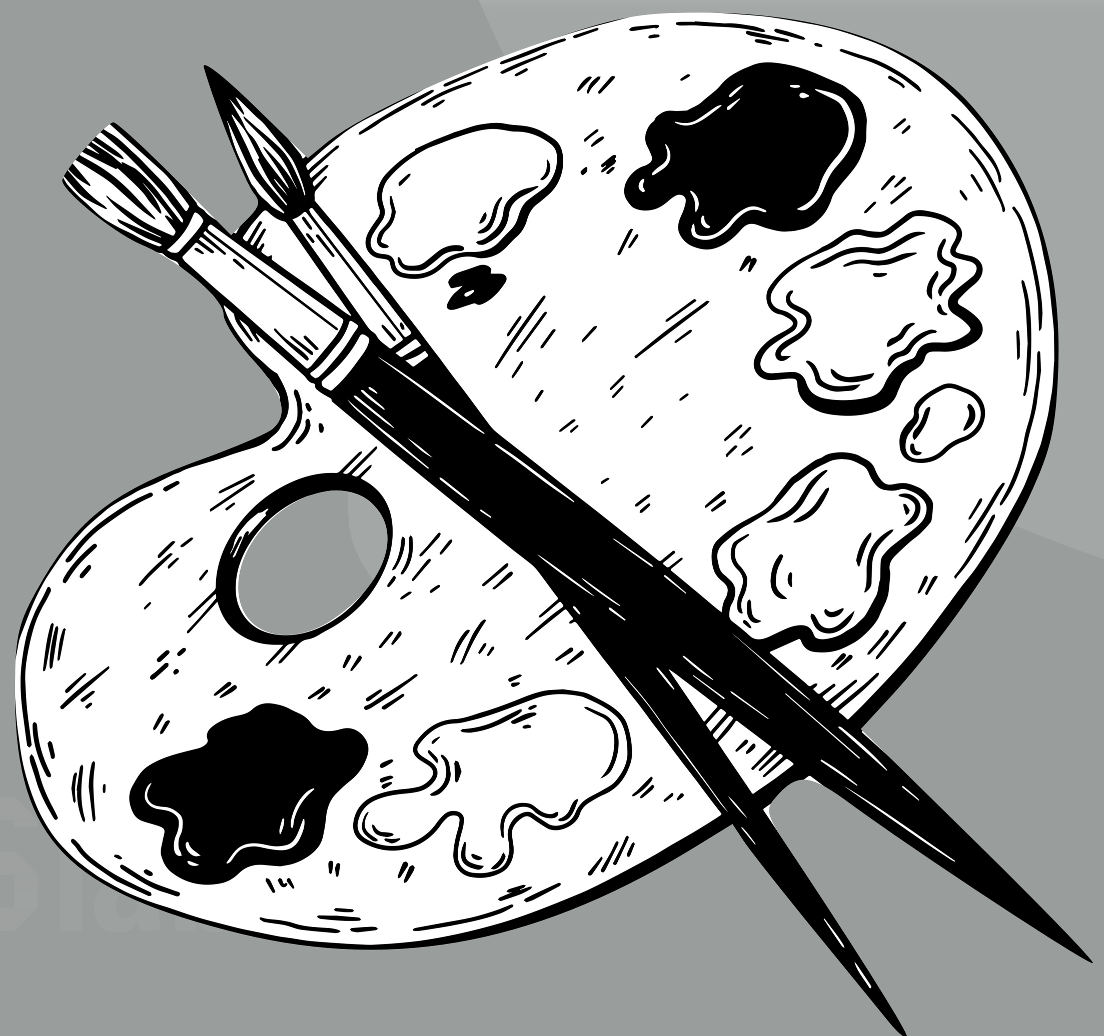
BUILD

GET *N* EXPERT DESIGNERS

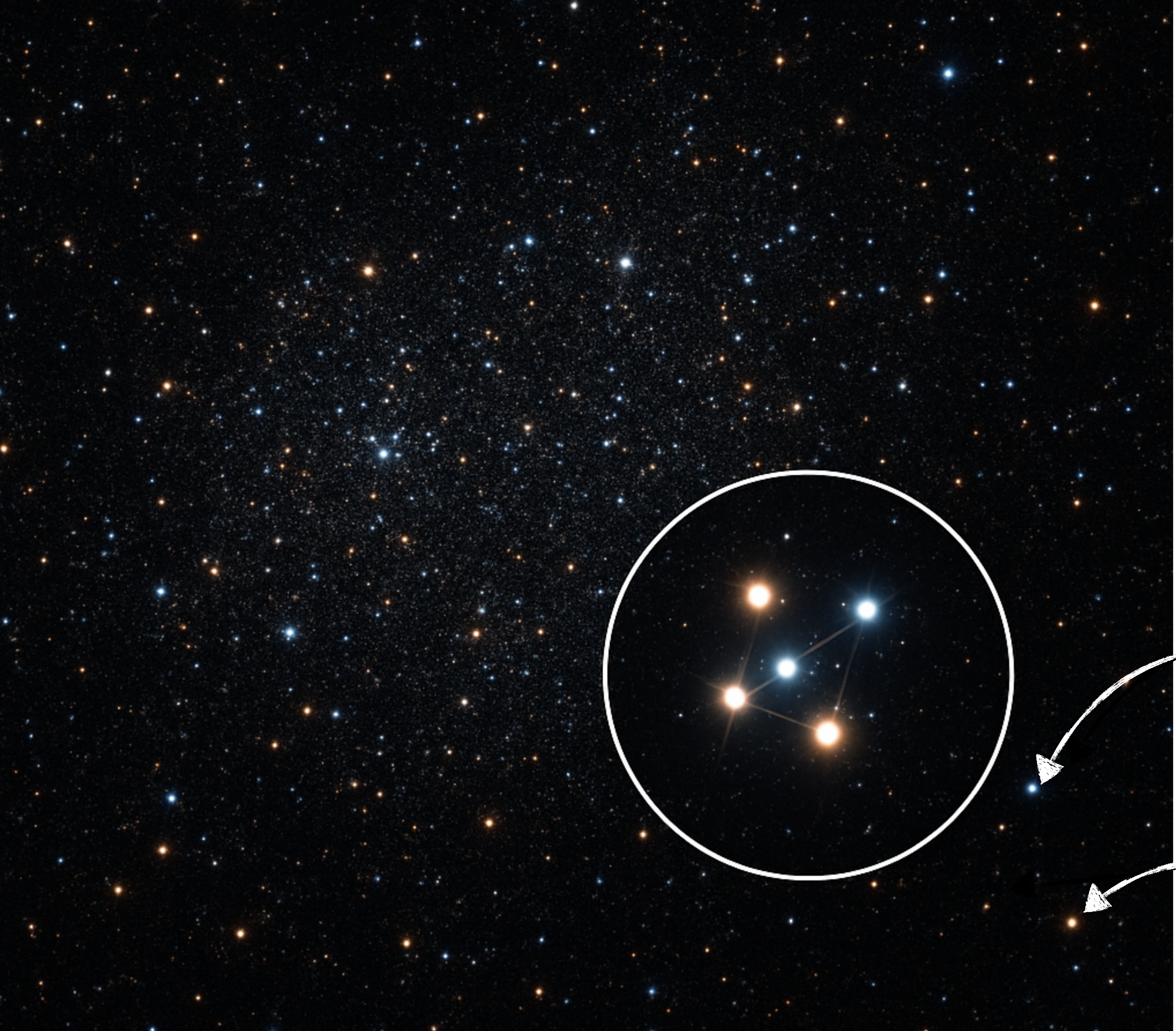
GIVE THEM *T* TIME

HOPE FOR THE BEST





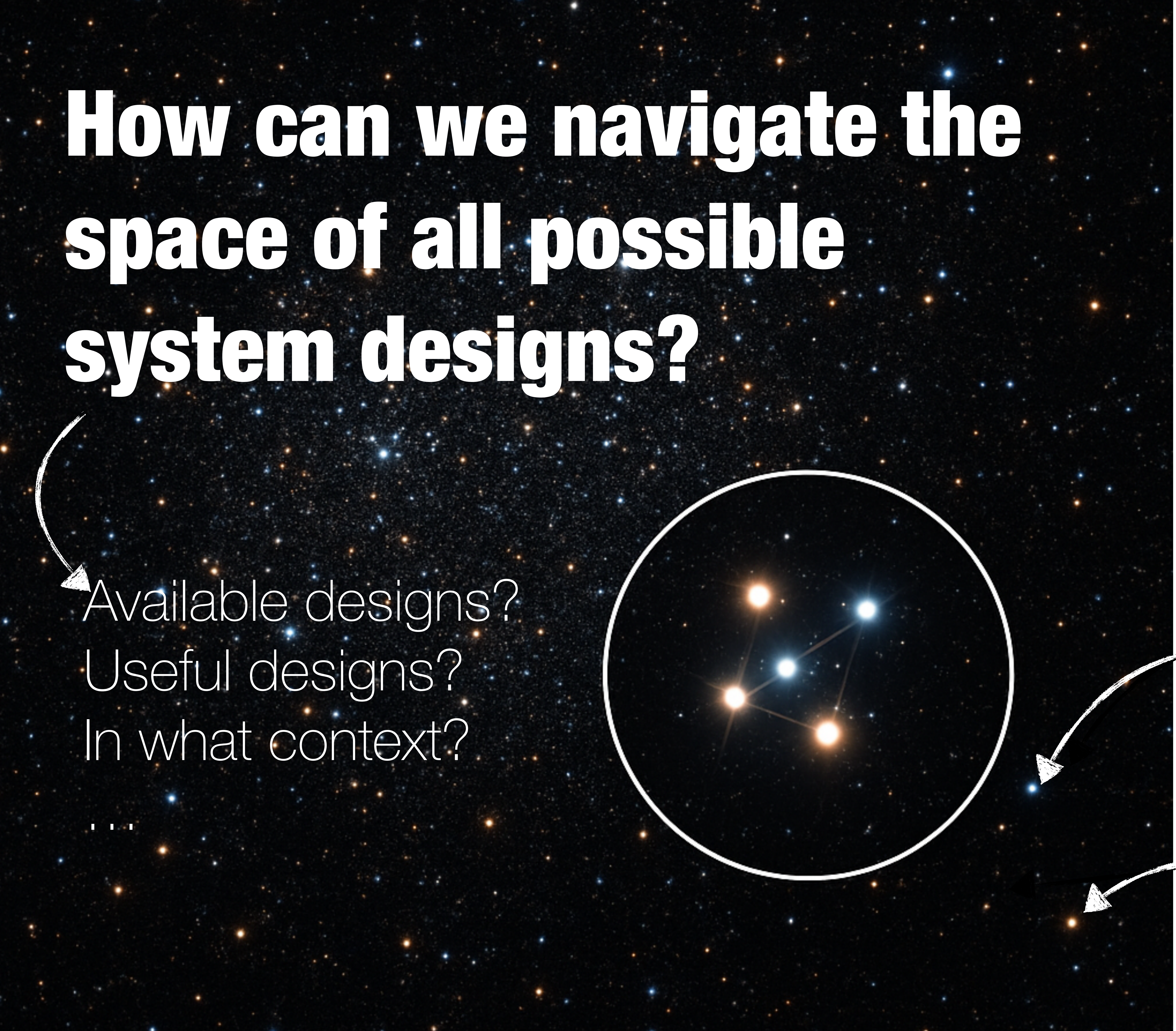
design is an art



Humans do not lack imagination …
but manual design exploration is too slow

EVERY PAPER
(1 DESIGN)
IS 1-2 YEARS

EVERY FULL
SYSTEM IS
7-10 YEARS

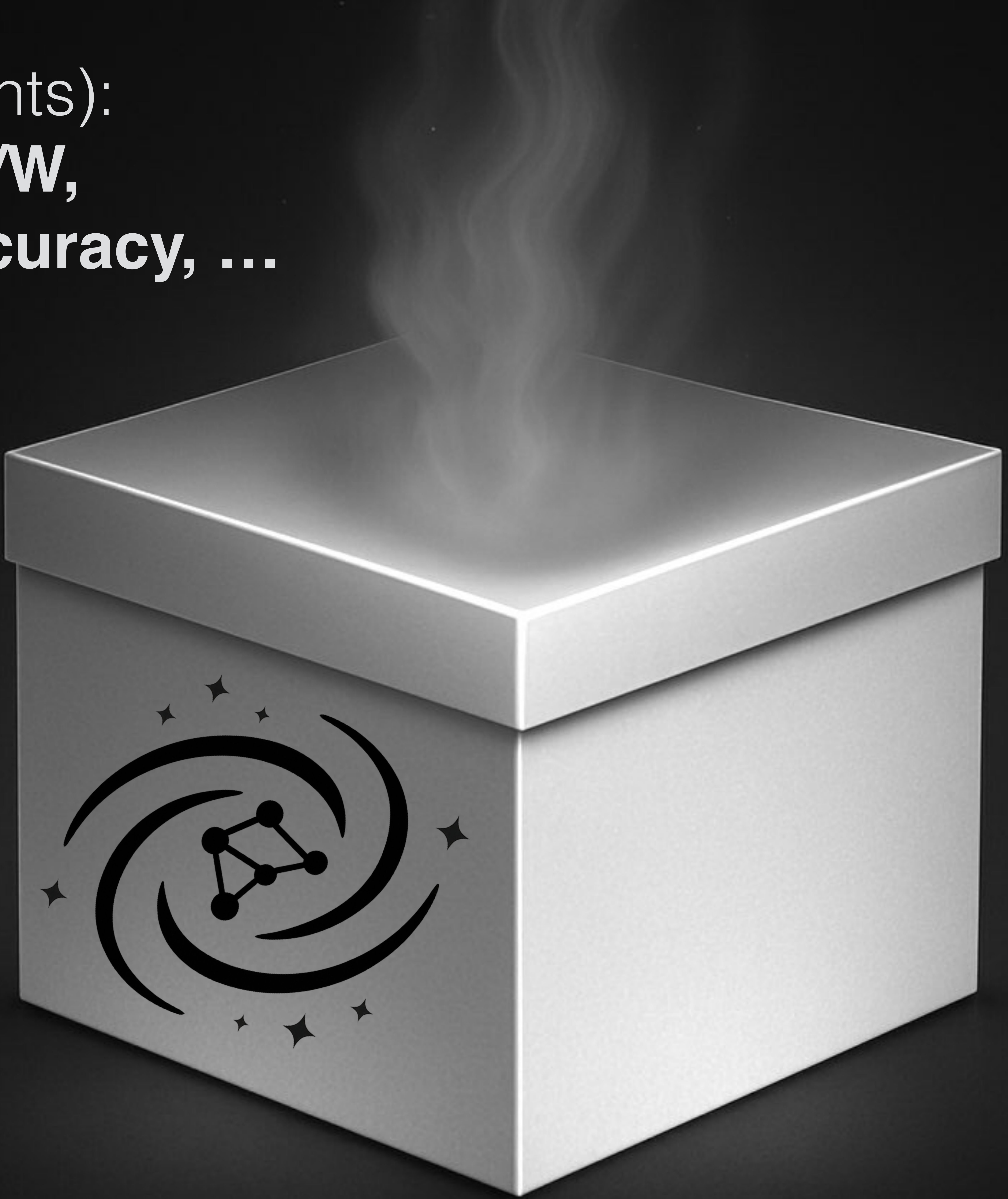


Humans do not lack imagination ...
but manual design exploration is too slow

EVERY PAPER
(1 DESIGN)
IS 1-2 YEARS

EVERY FULL
SYSTEM IS
7-10 YEARS

Input (Requirements):
Data, Queries, H/W,
Cost, Speed, Accuracy, …



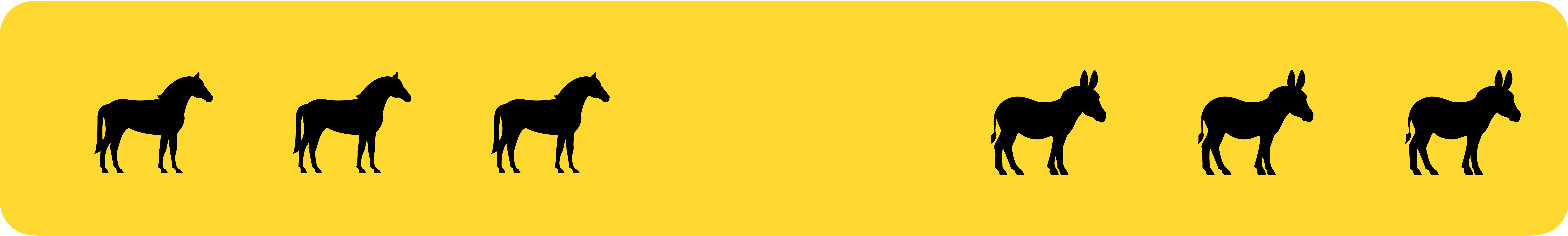
Output:
Tailored System

WHY DO WE NEED NEW SYSTEMS?

But first: *“What is a data system design?”*

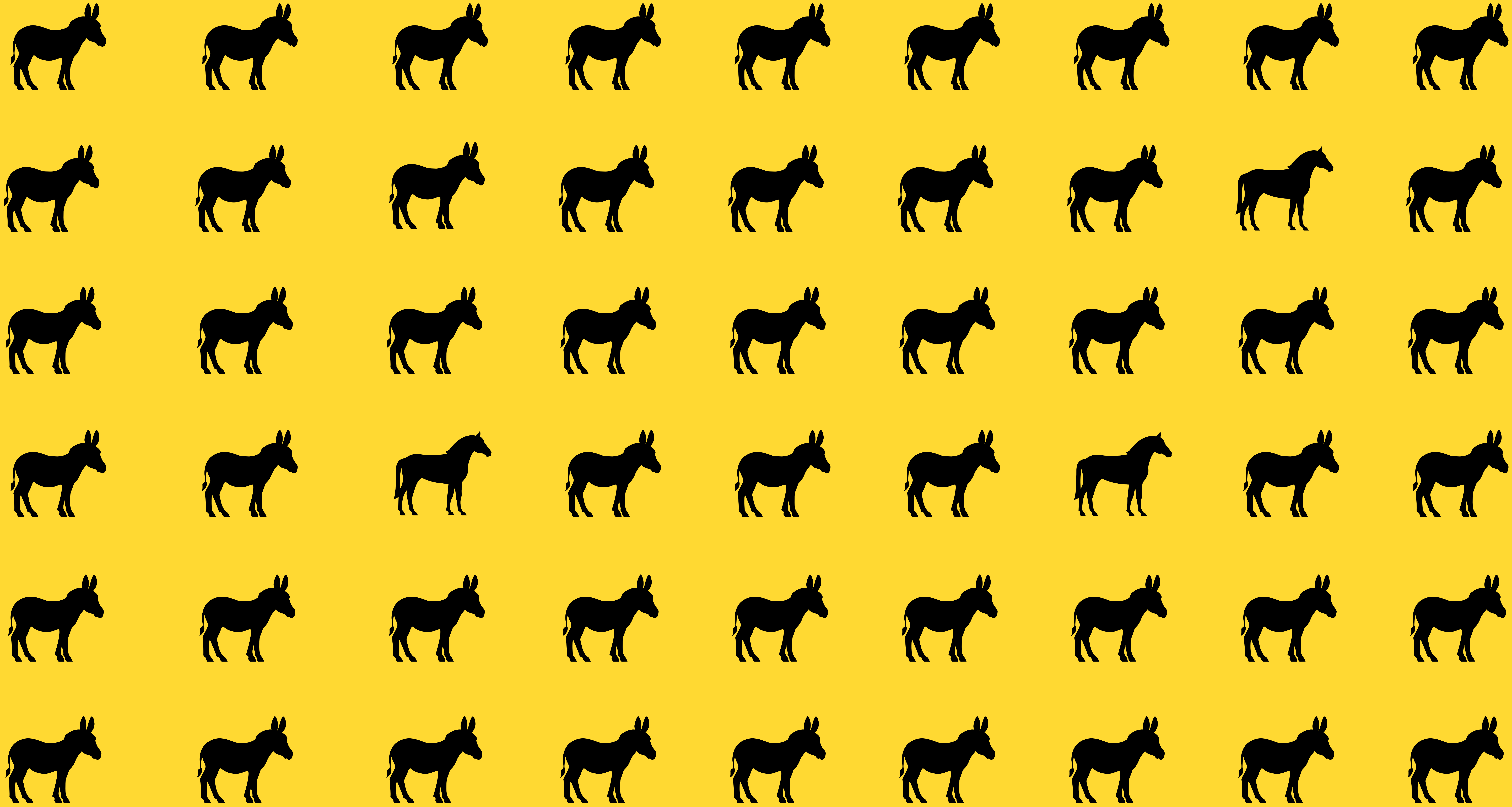
A TYPICAL BIG DATA TASK

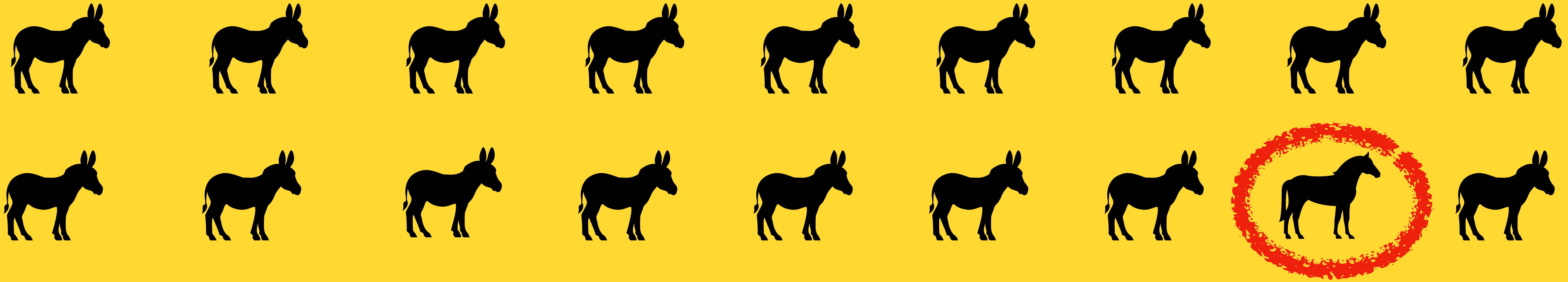
image analysis: e.g., detect the number of horses



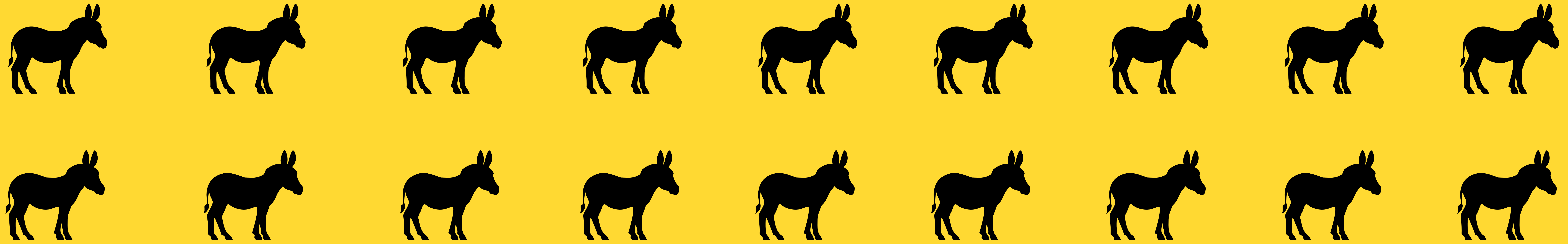
A TYPICAL BIG DATA TASK

image analysis: e.g., detect the number of horses

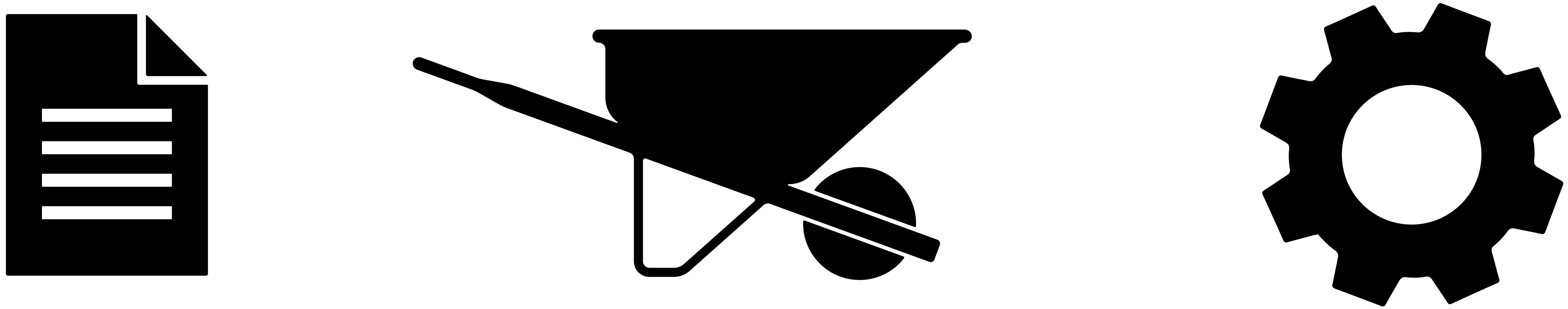




The size and organization of the data define how we can process requests



Three components in e2e system design

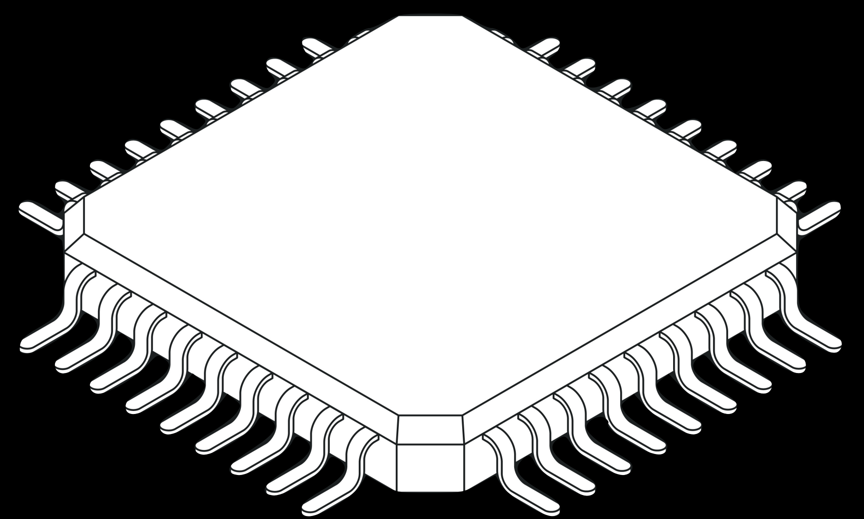


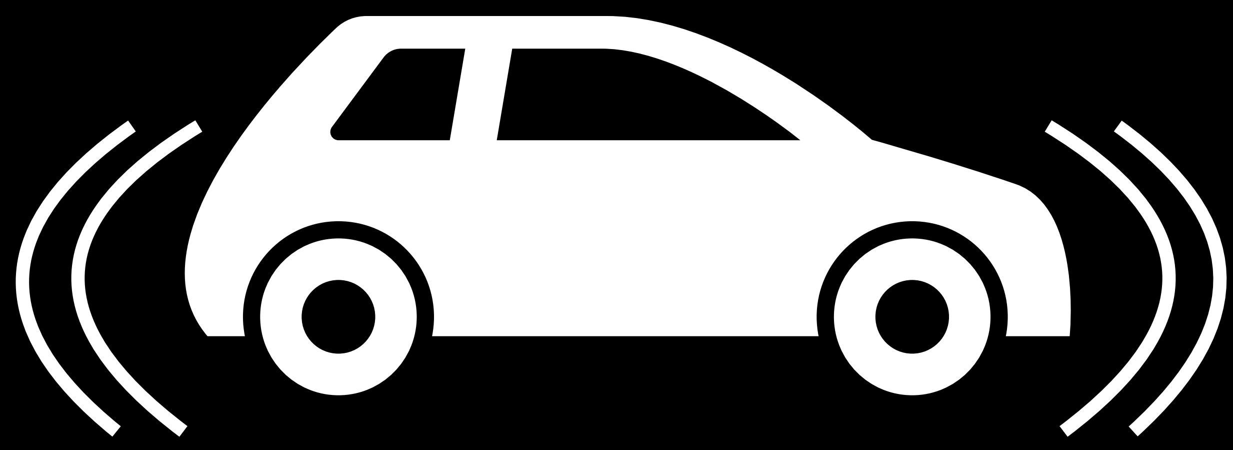
STORE MOVE PROCESS

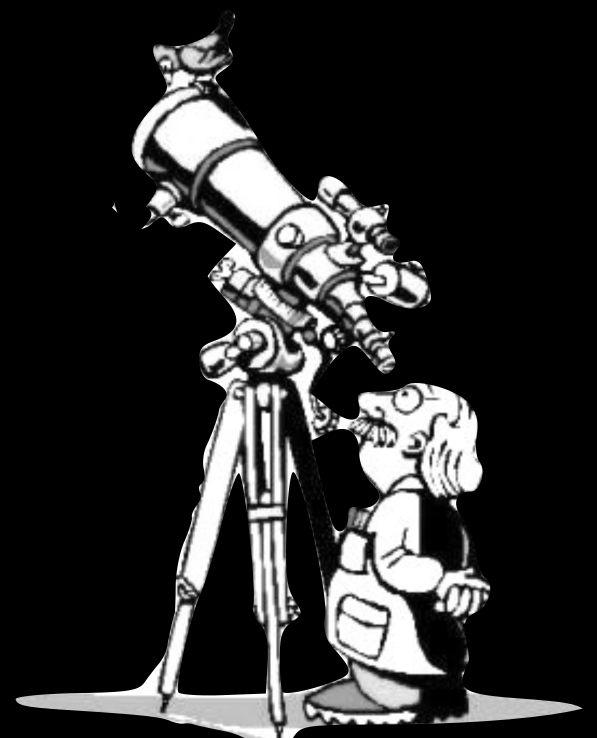
NEW PERFORMANCE/FEATURES

REDESIGN FROM STORAGE

BIG DATA/AI
NEW APPLICATIONS/PERFORMACE

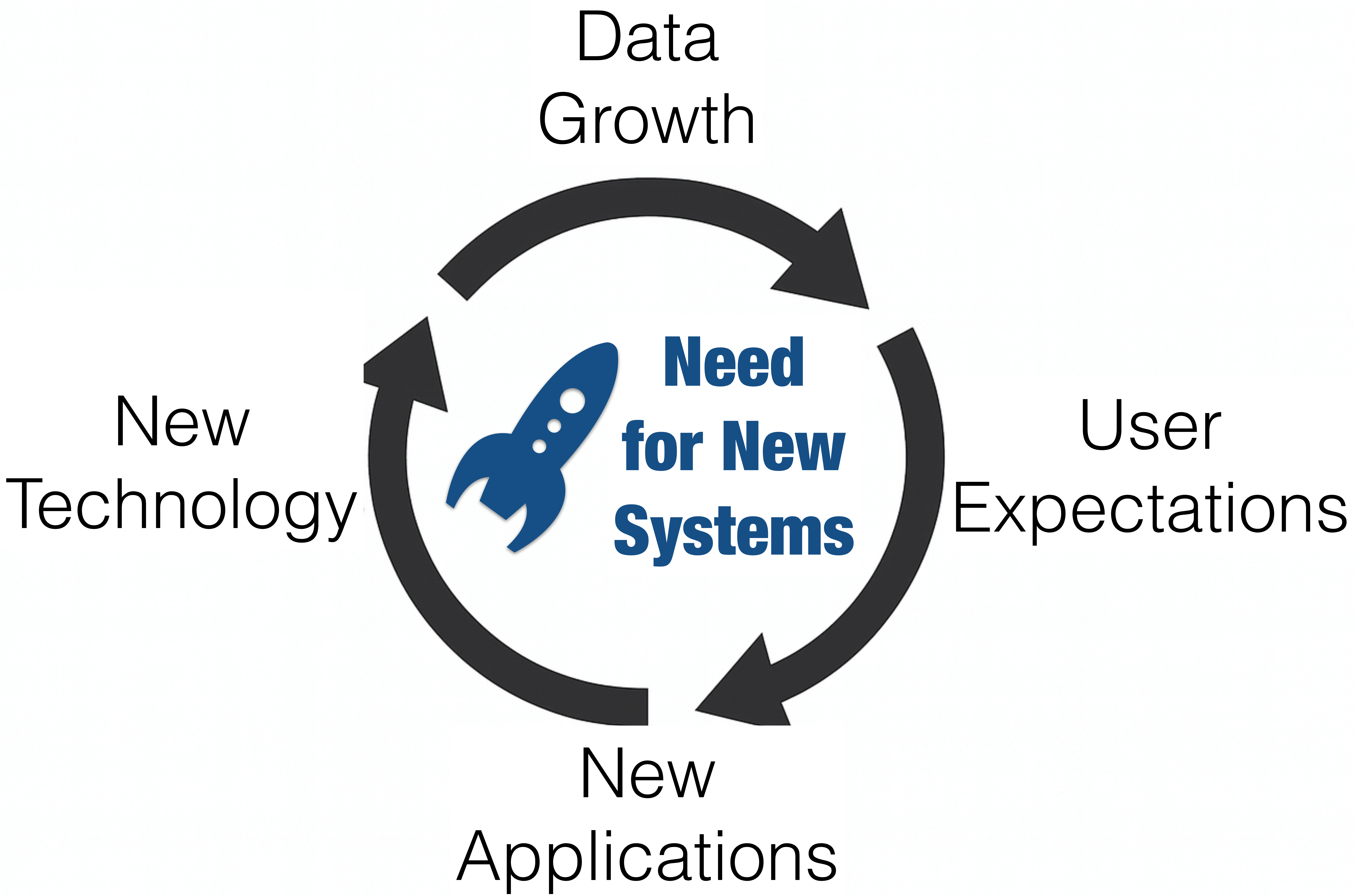






NEW SYSTEMS

DASlab
@ Harvard SEAS


Data
Growth
User
Expectations
New
Applications
New
Technology
Need
for New
Systems

“Getting a new data structure into
production takes years.
And by the time it’s ready,
your assumptions are already wrong.”

Mark Callaghan

“Getting a new data structure into production takes years.
And by the time it’s ready,
your assumptions are already wrong.”

Mark Callaghan

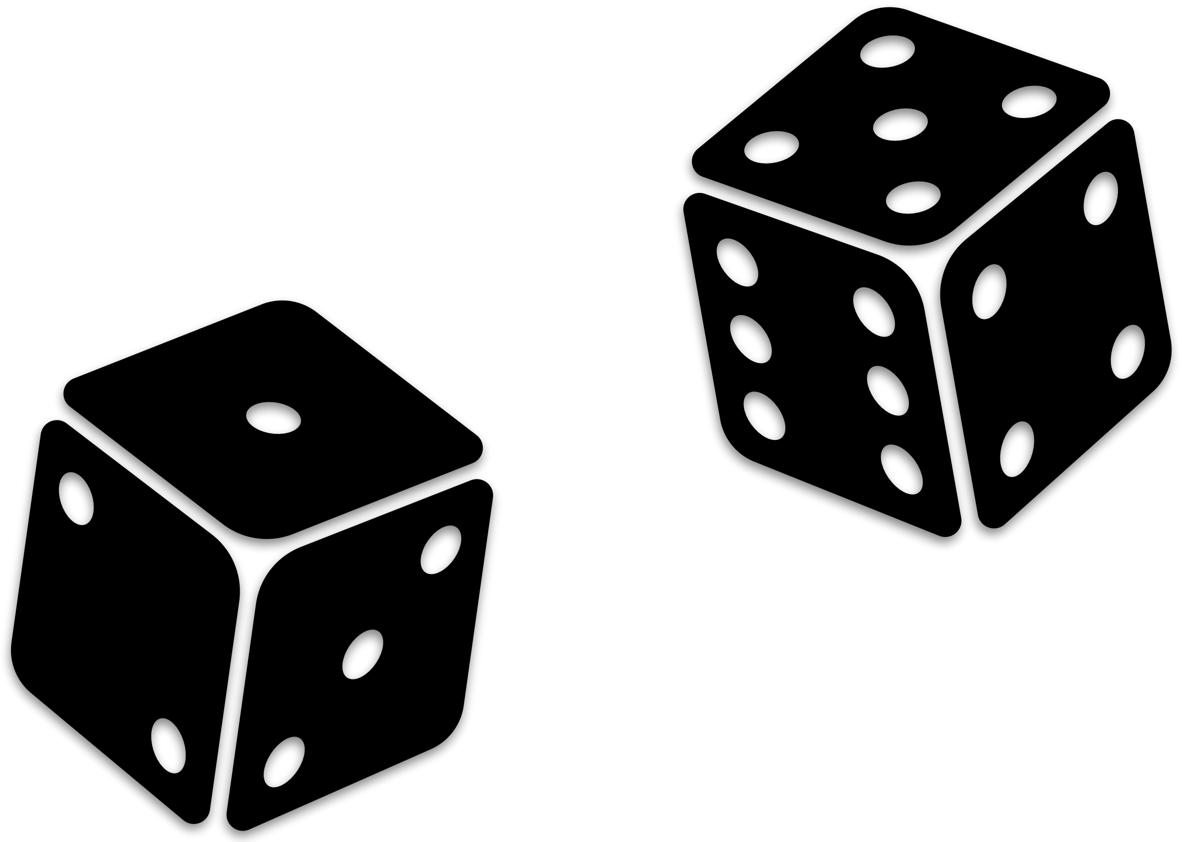
“We likely have the data to cure cancer.
We just do not know what query to ask.”

Martin Kersten

How do we design a data system that is **X times faster for a workload W**? 

 How do we design a data system that allows for control of **cloud cost**?

What happens if we introduce **new application feature** Y?



Should we **upgrade** to new version Z?

What will **break** our system?

Beyond Performance: WHAT IF design

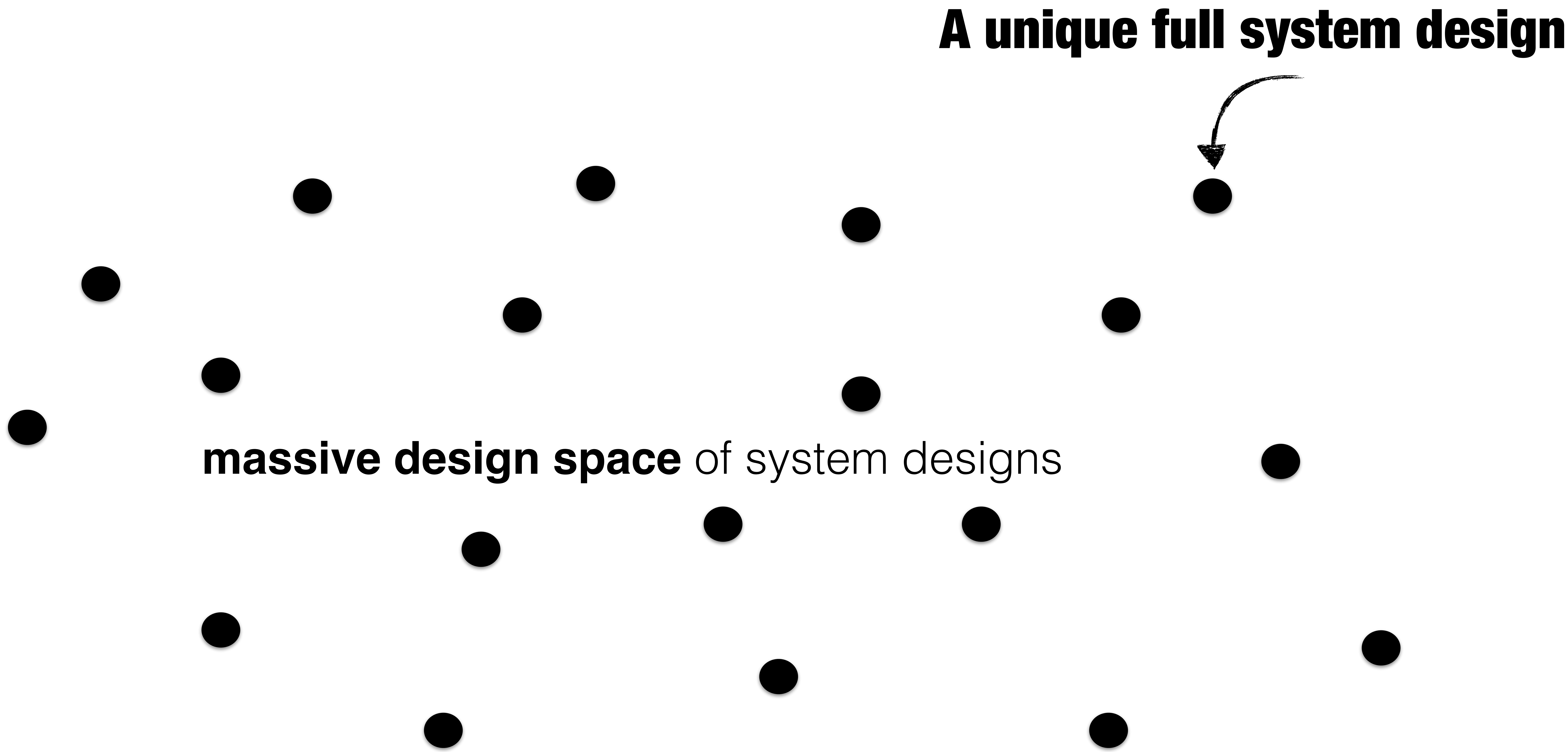
HOW ABOUT ADAPTIVE SYSTEMS?

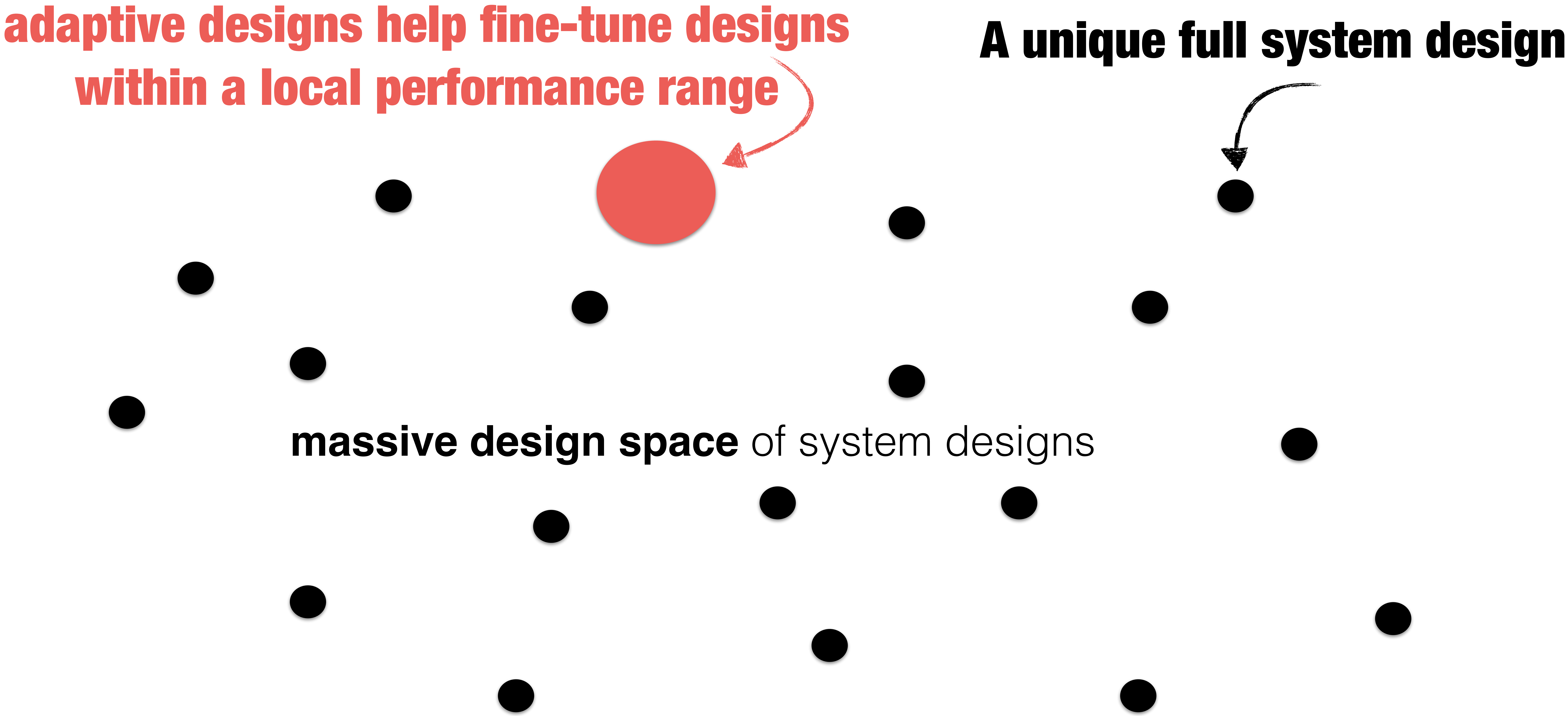
Databases= Adaptive (Optimizers, Auto-tuning)?
Cracking/Adaptive Indexing/Learned Components?…

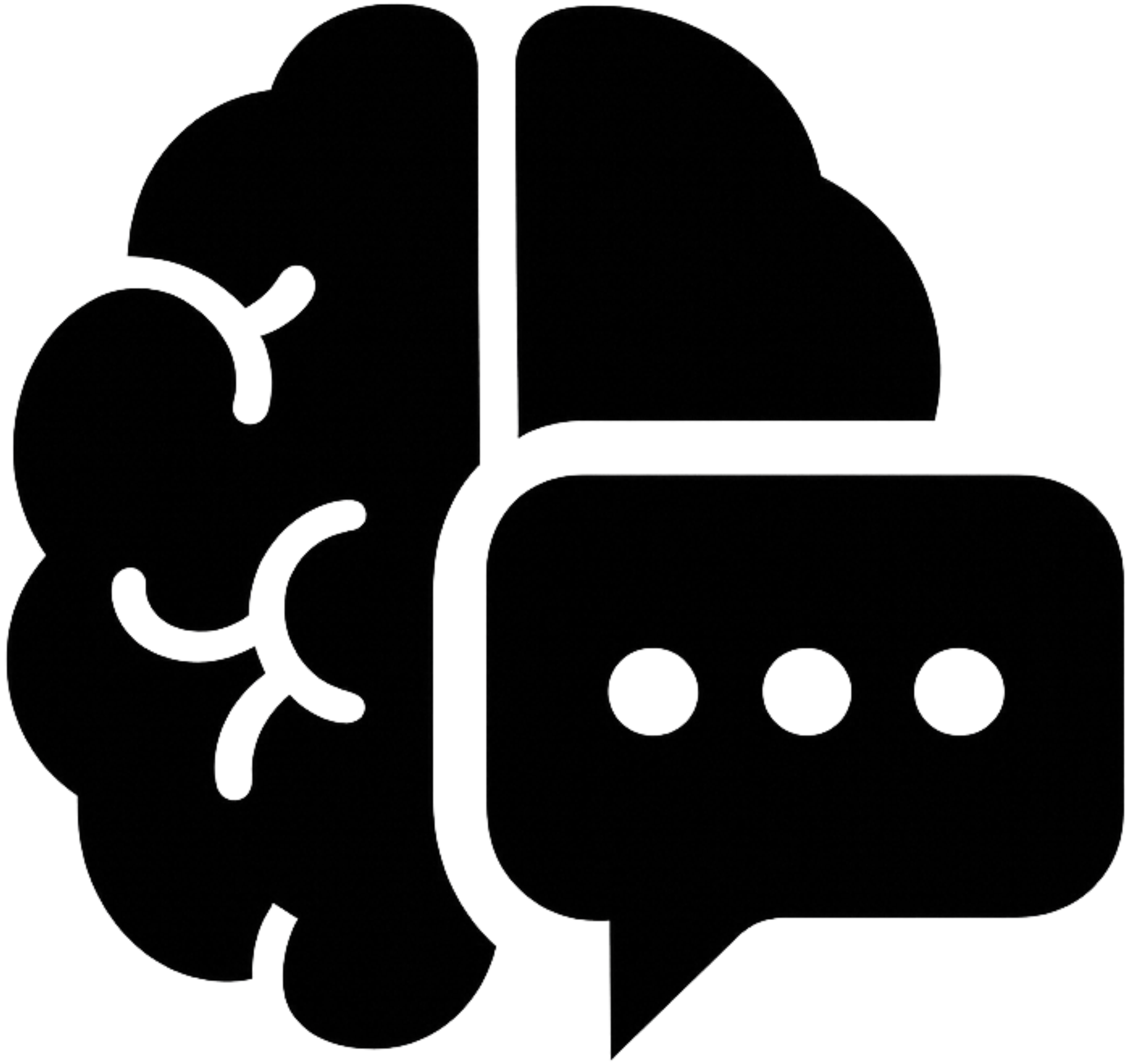
small variations

small-big variations

STORE **MOVE** **PROCESS**


A unique full system design
massive design space of system designs

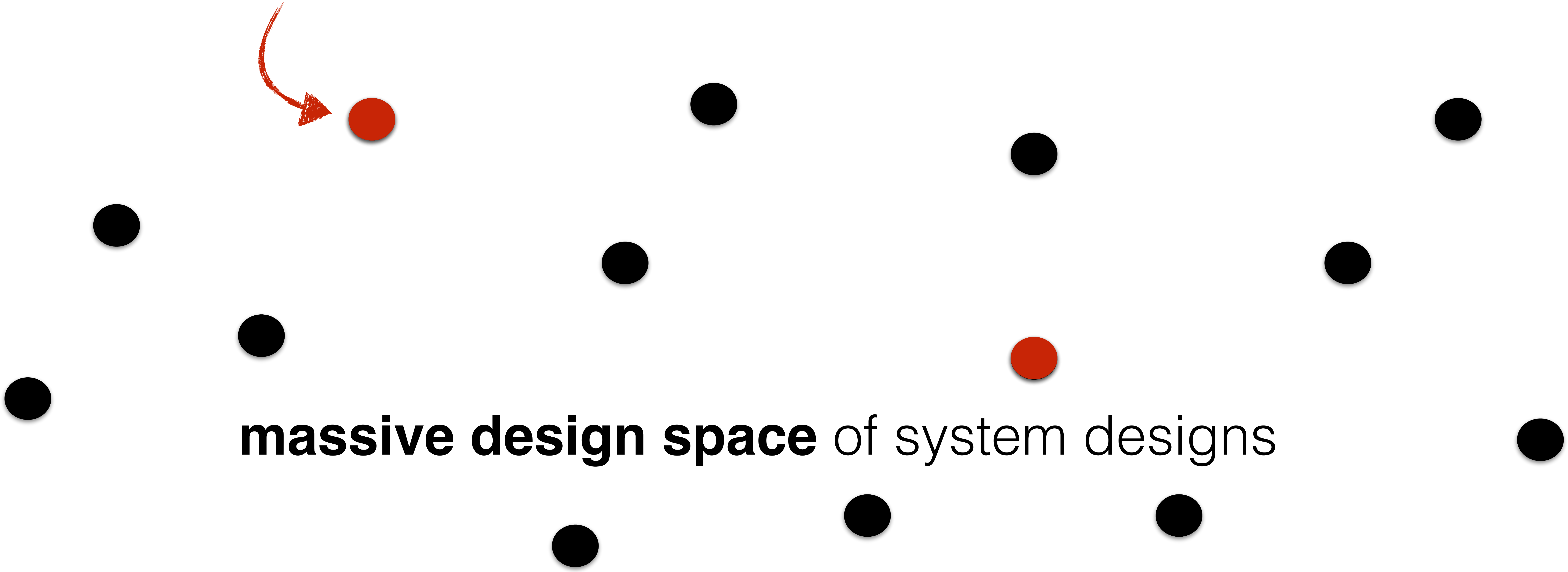

adaptive designs help fine-tune designs
within a local performance range
A unique full system design
massive design space of system designs



CAN LLMs OR NEW LARGE MODELS MAKE A DIFFERENCE?

Yes, amazing reasoning, but ….
need tons of data, money, time

few existing designs and mostly complex closed-source code



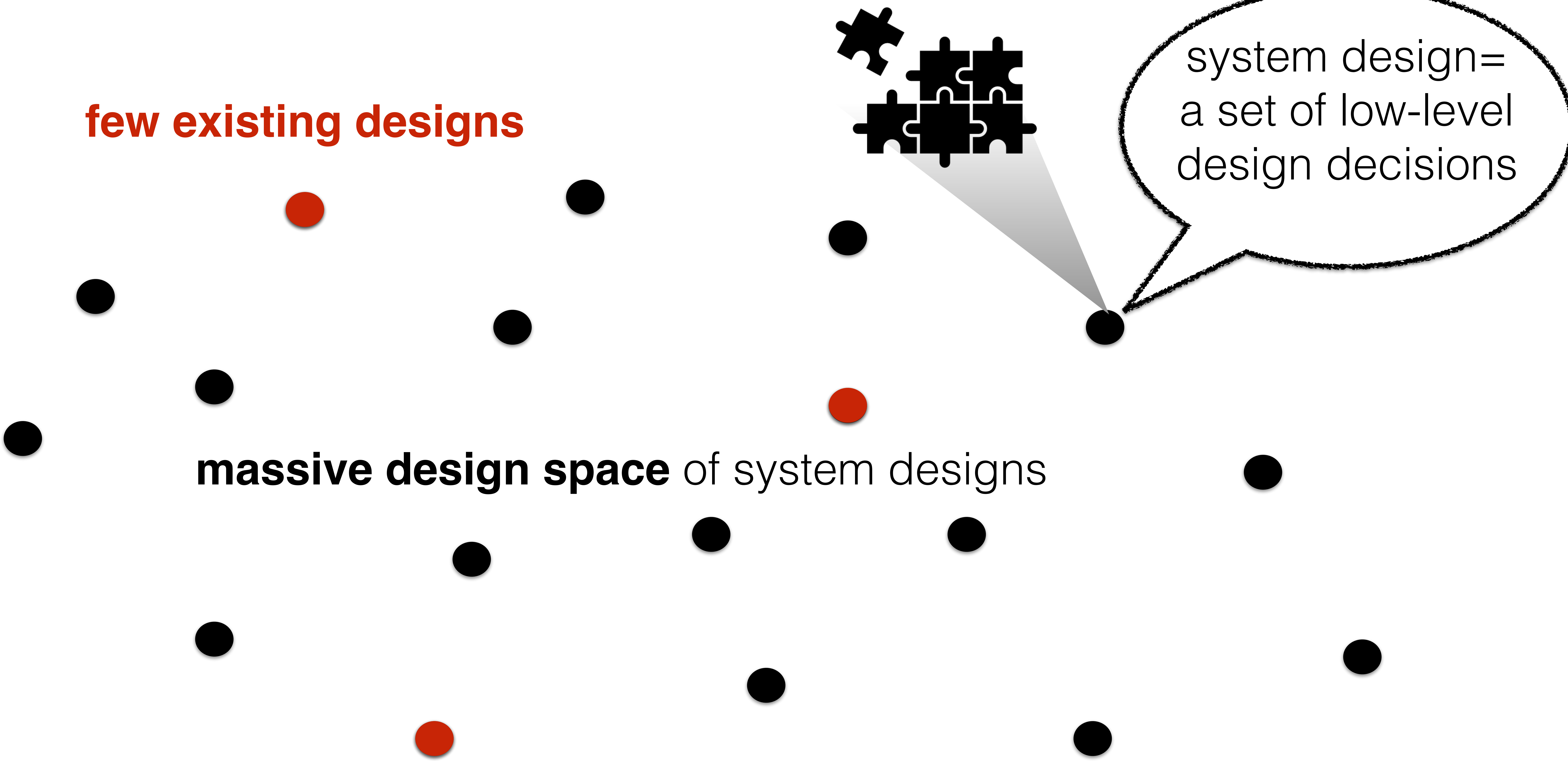
NO WEBSCALE DATA FOR SYSTEM DESIGN

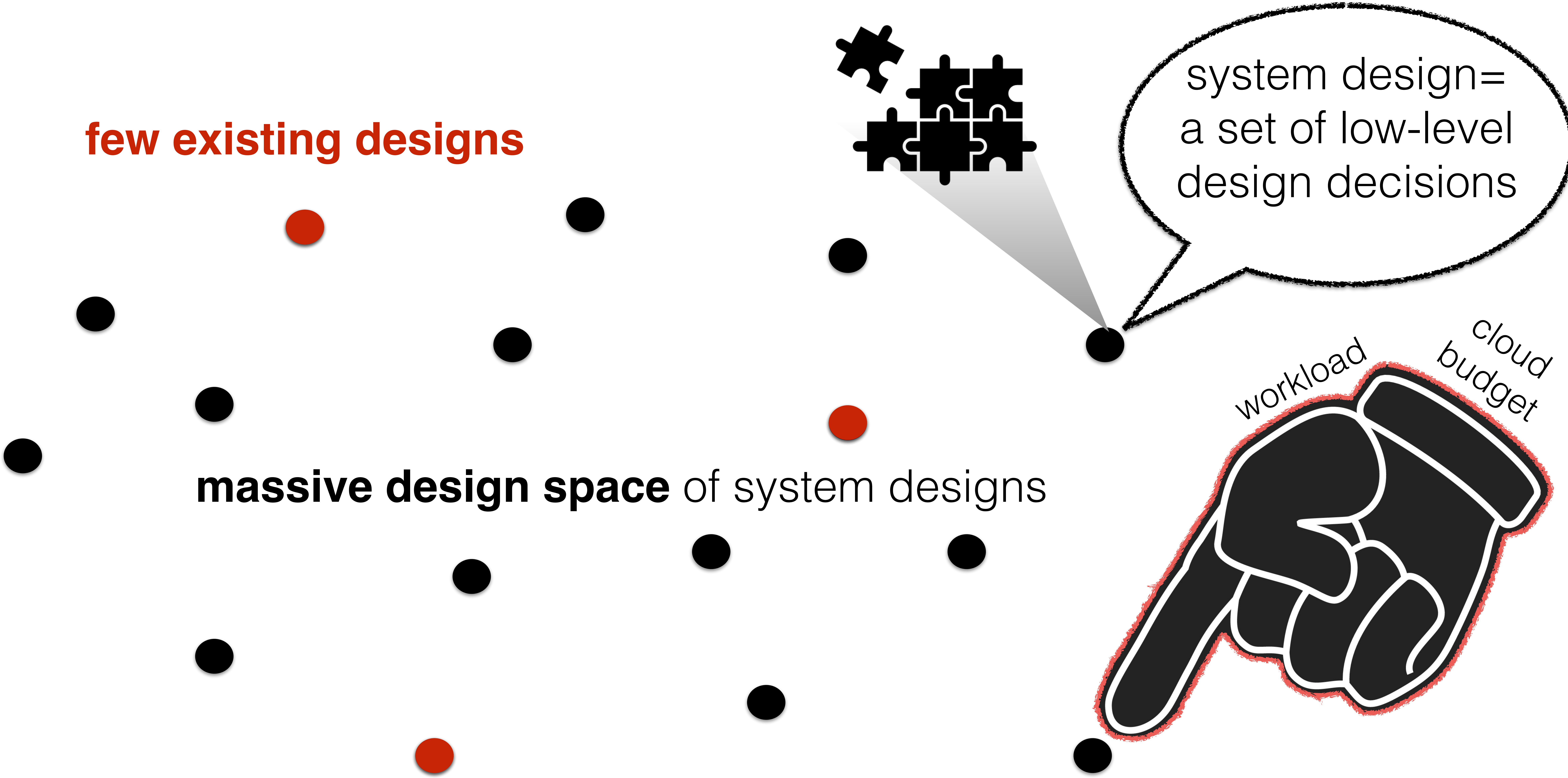
Adaptive designs & large models are part of the solution, but we also need something else to:

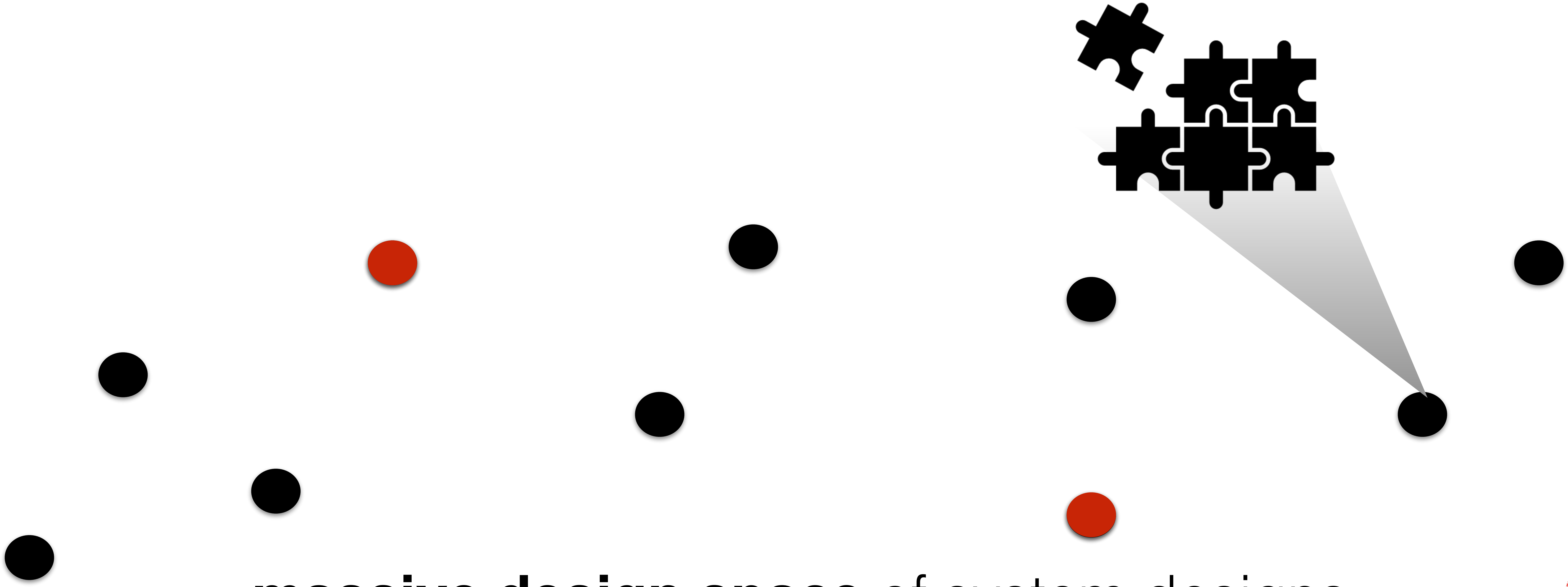
FIND FAST THE BEST POSSIBLE DESIGN

SELF-DESIGNING SYSTEMS

automatically invent & build the perfect system for any new application


few existing designs
system design=
a set of low-level
design decisions
massive design space of system designs


few existing designs
system design=
a set of low-level
design decisions
massive design space of system designs
workload
cloud
budget



massive design space of system designs

reasoning: understand all the design decisions & their impact

AUTO DESIGN IS AS OLD AS COMPUTER SCIENCE

Rob Tarjan, Turing Award 1986

“IS THERE A CALCULUS OF DATA STRUCTURES

by which one can choose the appropriate representation and techniques for a given problem?” (SIAM,1978)

[*P* vs *NP, average case, constant factors vs asymptotic, low bounds*]

IS THERE A CALCULUS OF DATA SYSTEMS?

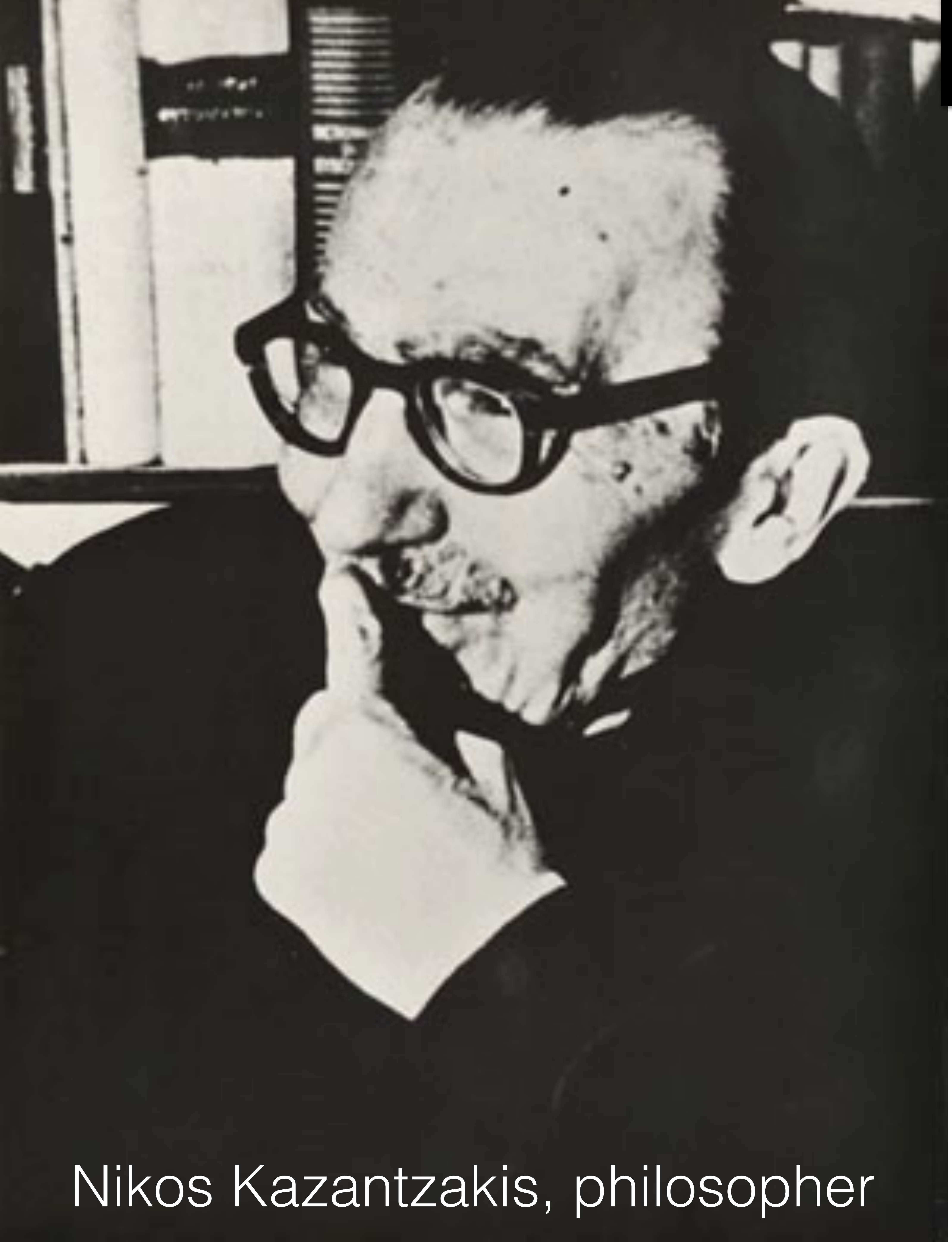
Rob Tarjan, Turing Award 1986

“IS THERE A CALCULUS OF DATA STRUCTURES

by which one can choose the appropriate representation and techniques for a given problem?” (SIAM,1978)

[*P* vs *NP, average case, constant factors vs asymptotic, low bounds*] the **grammar** of data systems design

the **grammar** of data systems design



Nikos Kazantzakis, philosopher

action *is* *for* *nothing* *holy*

hope *the* *most* *of*

fear *form*

am *free*

ultimate *I* *theory*

action is

the most holy

of

form

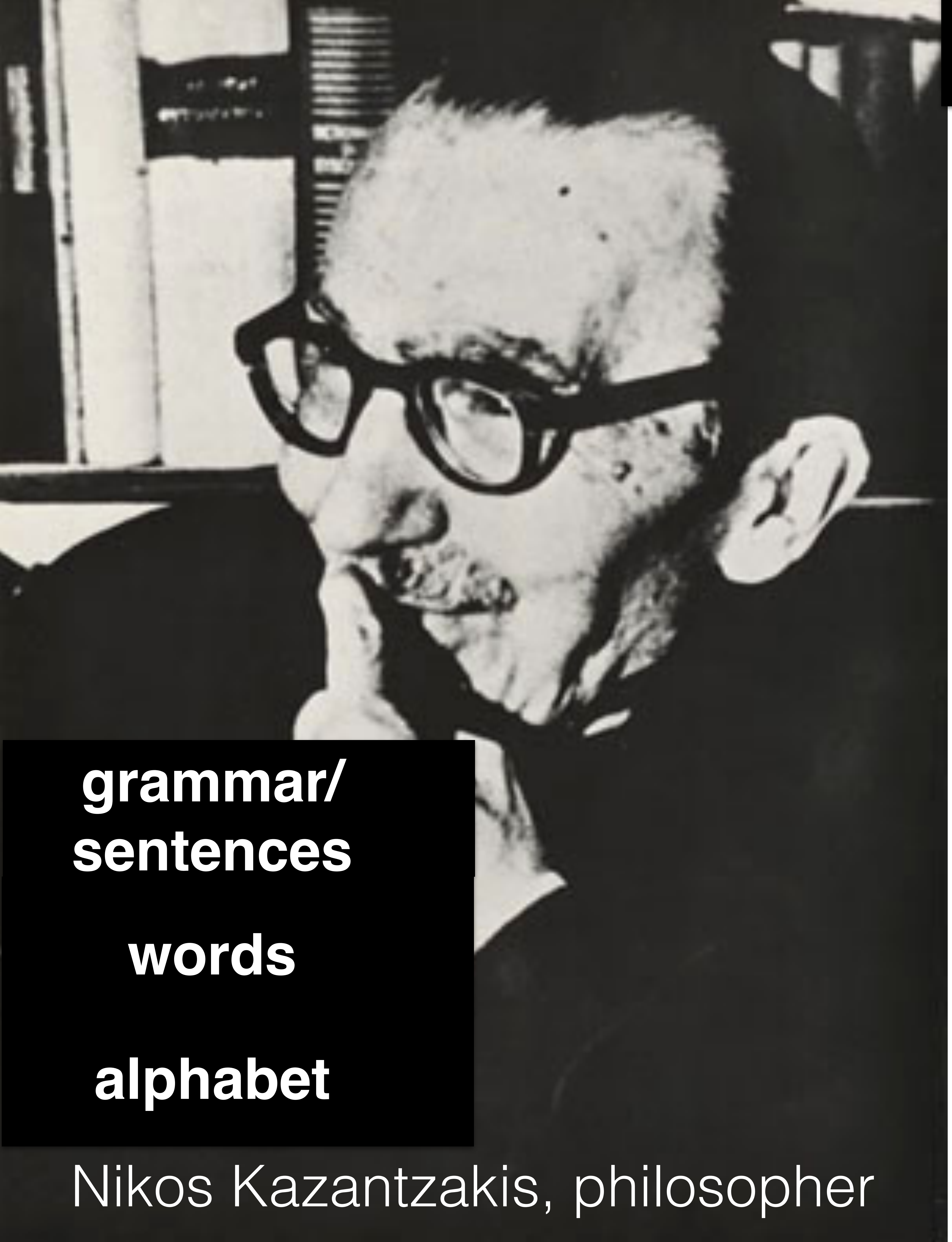
ultimate theory

I hope for nothing

I fear nothing

I am free

the **grammar** of data systems design



Nikos Kazantzakis, philosopher

action *is*

the *most* *holy* *of*

form

ultimate *theory*

I hope for nothing

I fear nothing

I am free

the **grammar** of data systems design

grammar/ sentences	**interactions**
words	**data structures**
alphabet	**principles**

Nikos Kazantzakis, philosopher

action is
the most holy
of
form
ultimate theory

I hope for nothing
I fear nothing
I am free

the **grammar** of data systems design

grammar/ sentences	**interactions**
words	**data structures**
alphabet	**principles**

Nikos Kazantzakis, philosopher

action is
the most holy
of
form
ultimate theory

NEW

I hope for nothing
I fear nothing
I am free

the **grammar** of data systems design

grammar/ sentences	**interactions**
words	**data structures**
alphabet	**principles**

Nikos Kazantzakis, philosopher

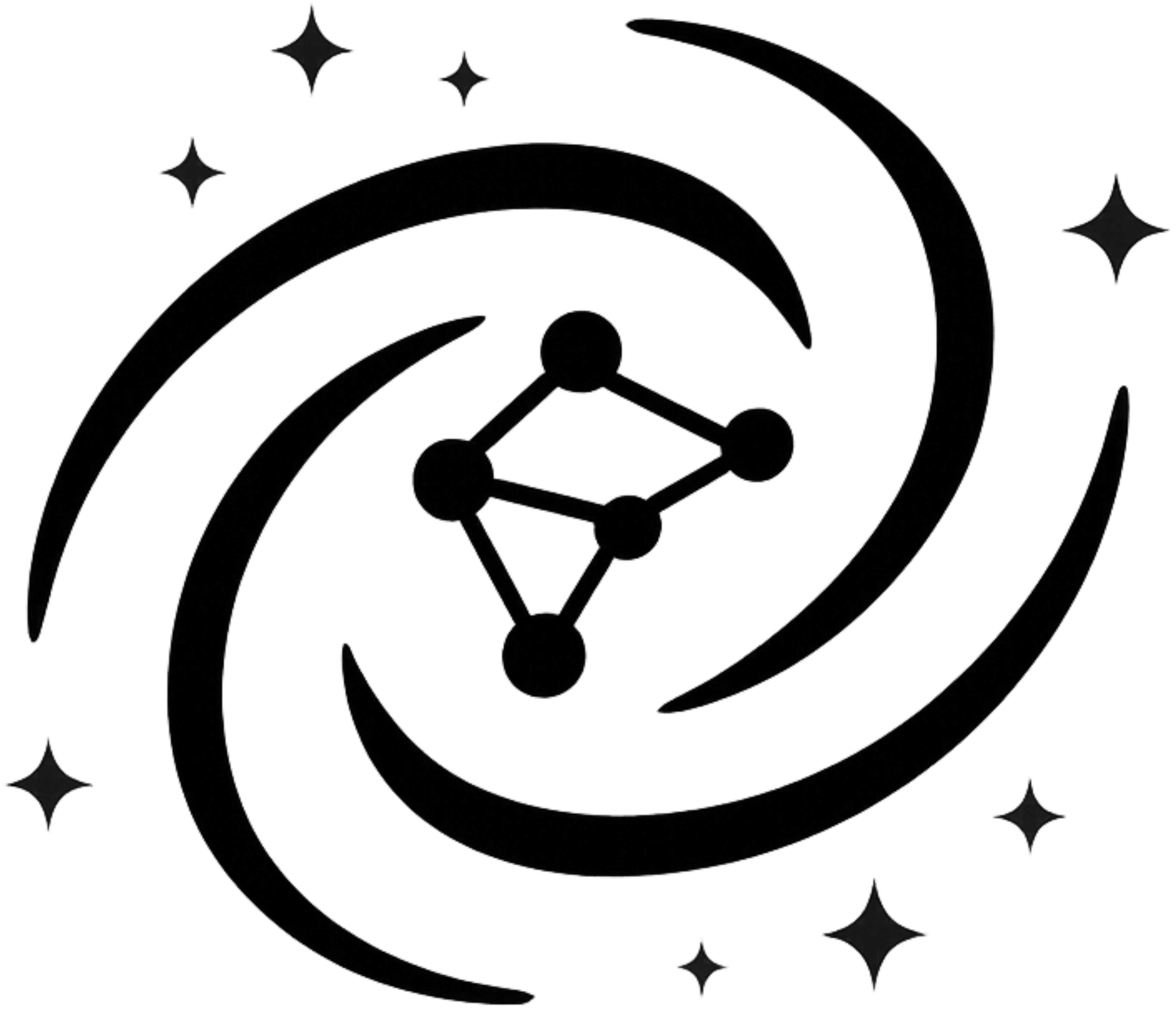
action is the most holy form of theory

which are “all” possible *data systems* **we may ever invent?**

I hope for nothing

I fear nothing

I am free



There exist trillions of possible designs for:

1) **Data Structures**
2) **Key-value Stores**
3) **Storage for Image AI**
4) **Large Model Training Algorithms**

…and we can navigate their design space

DESIGN SPACE | COST ESTIMATION | SEARCH

Which are all design principles & how they “connect” to synthesize all possible designs?

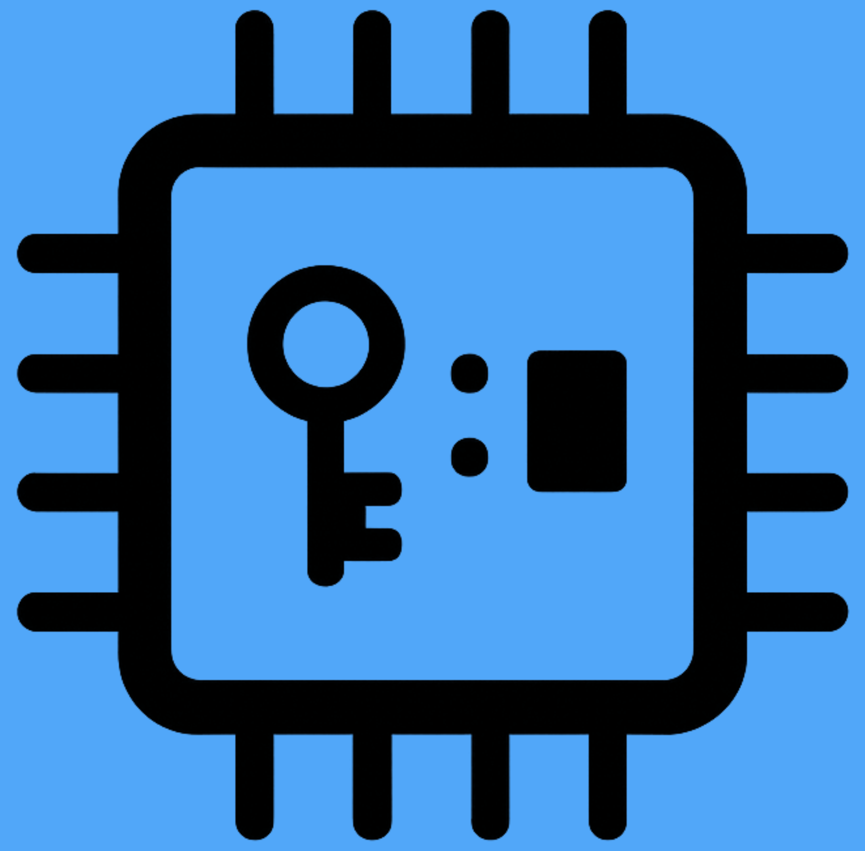
If we implemented in the best possible way 2 designs, how would they behave on data X & hardware Y?

DESIGN SPACE

COST ESTIMATION

SEARCH

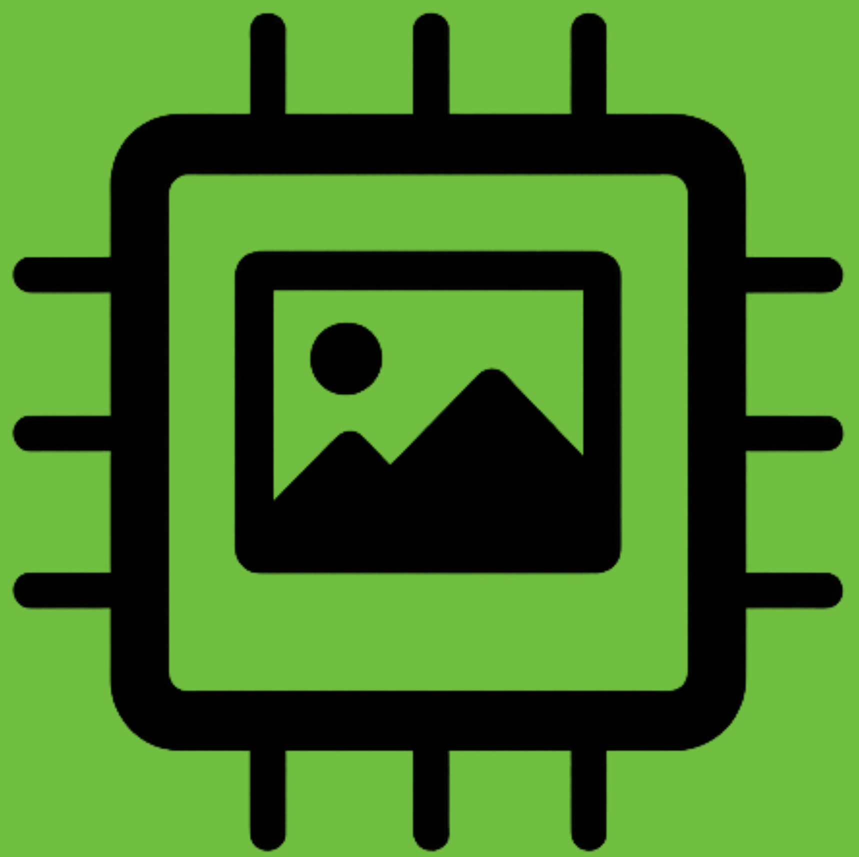
Cosine



1000x faster
key-value stores

SIGMOD’18/24, VLDB’22

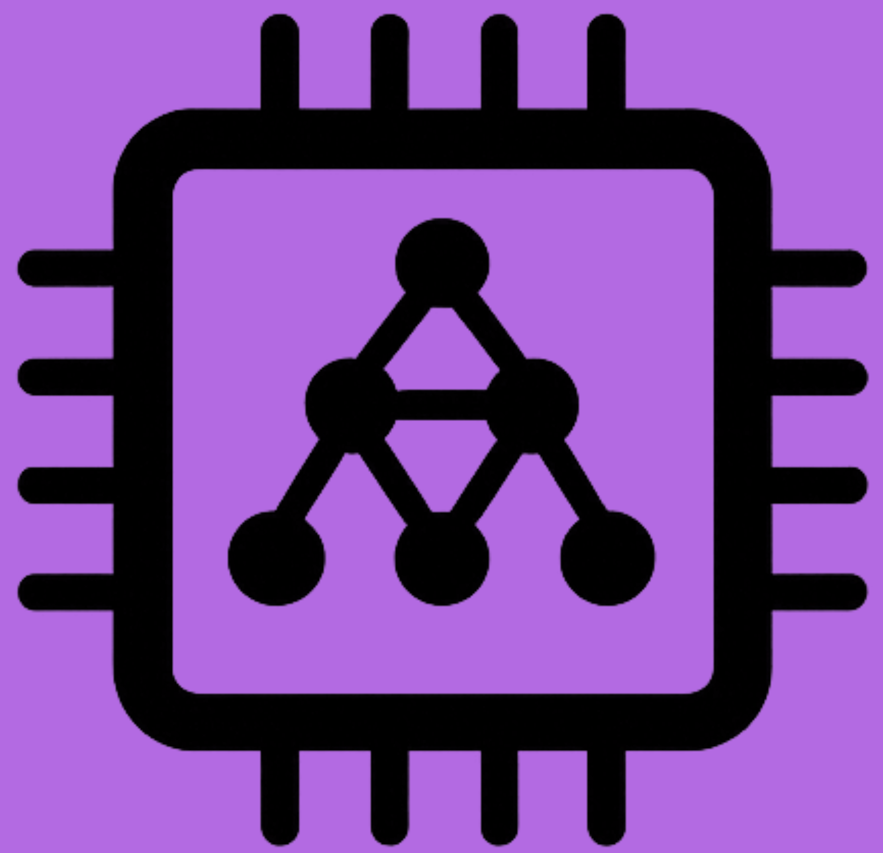
Image Calculator



10x faster
image AI inference

SIGMOD’24

TorchTitan



3x faster
large model training

MLsys 2023, ICLR’25

DESIGN SPACE

COST ESTIMATION

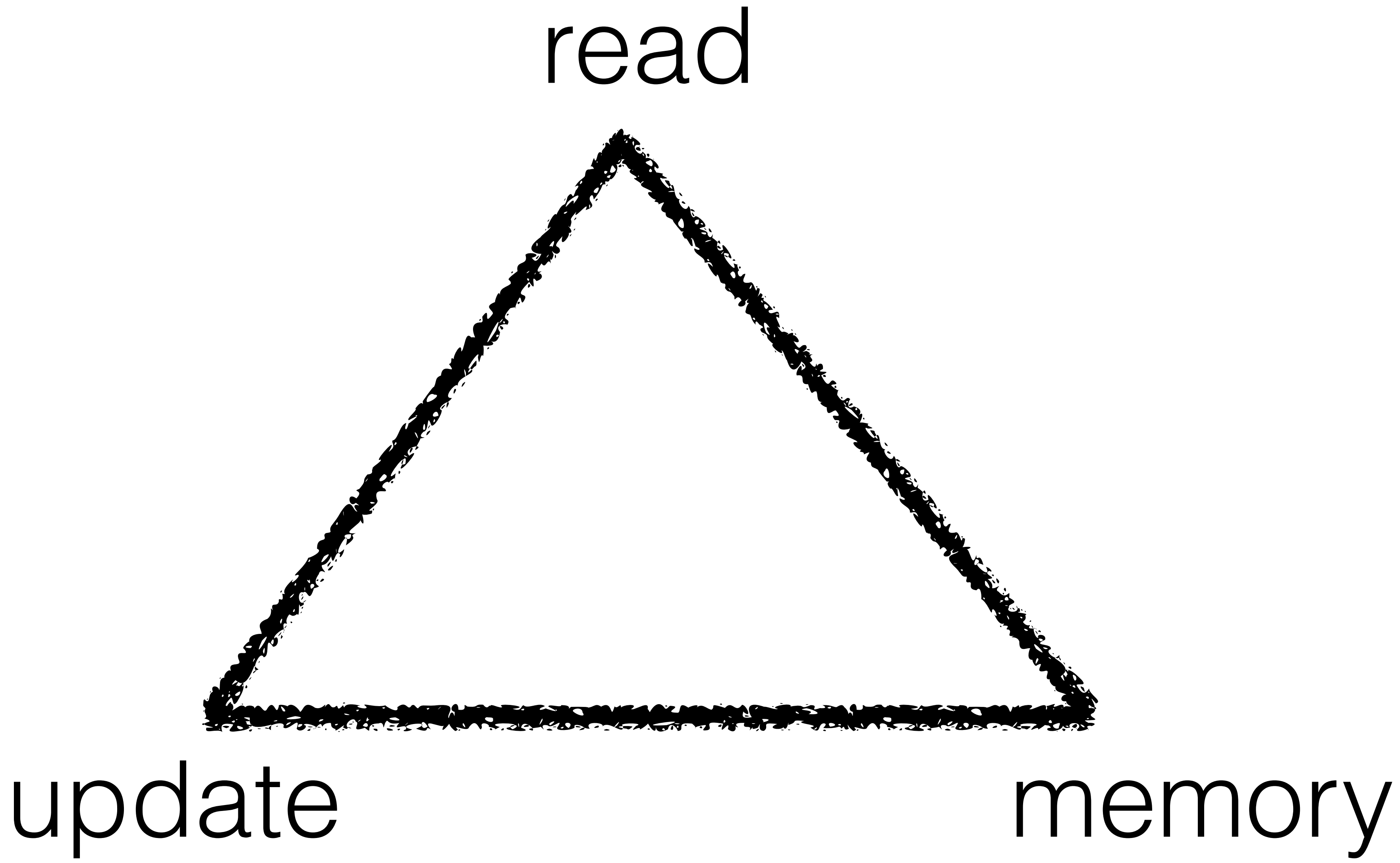
SEARCH

no perfect structure

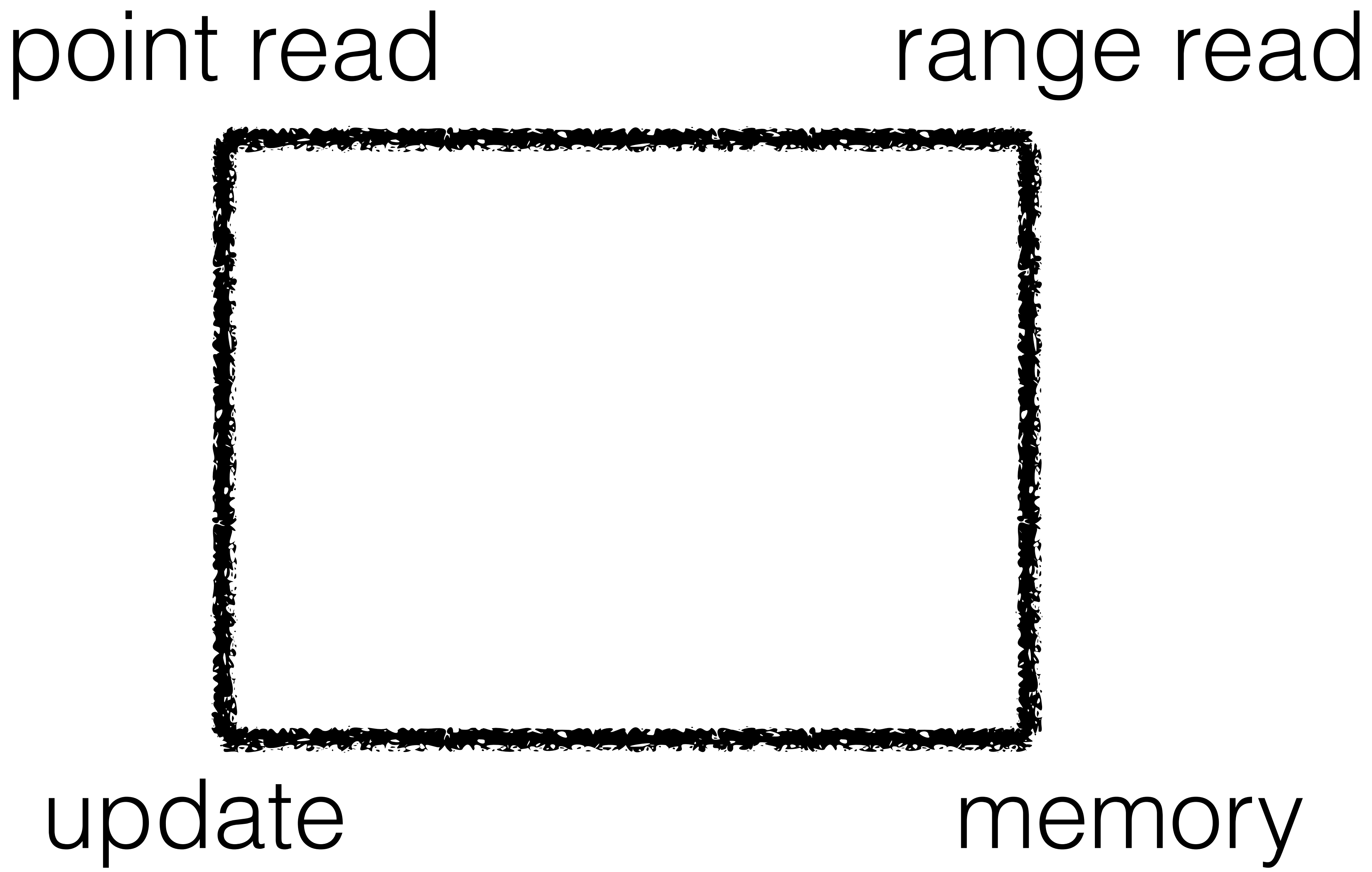
EDBT 2016
SIGMOD 2016

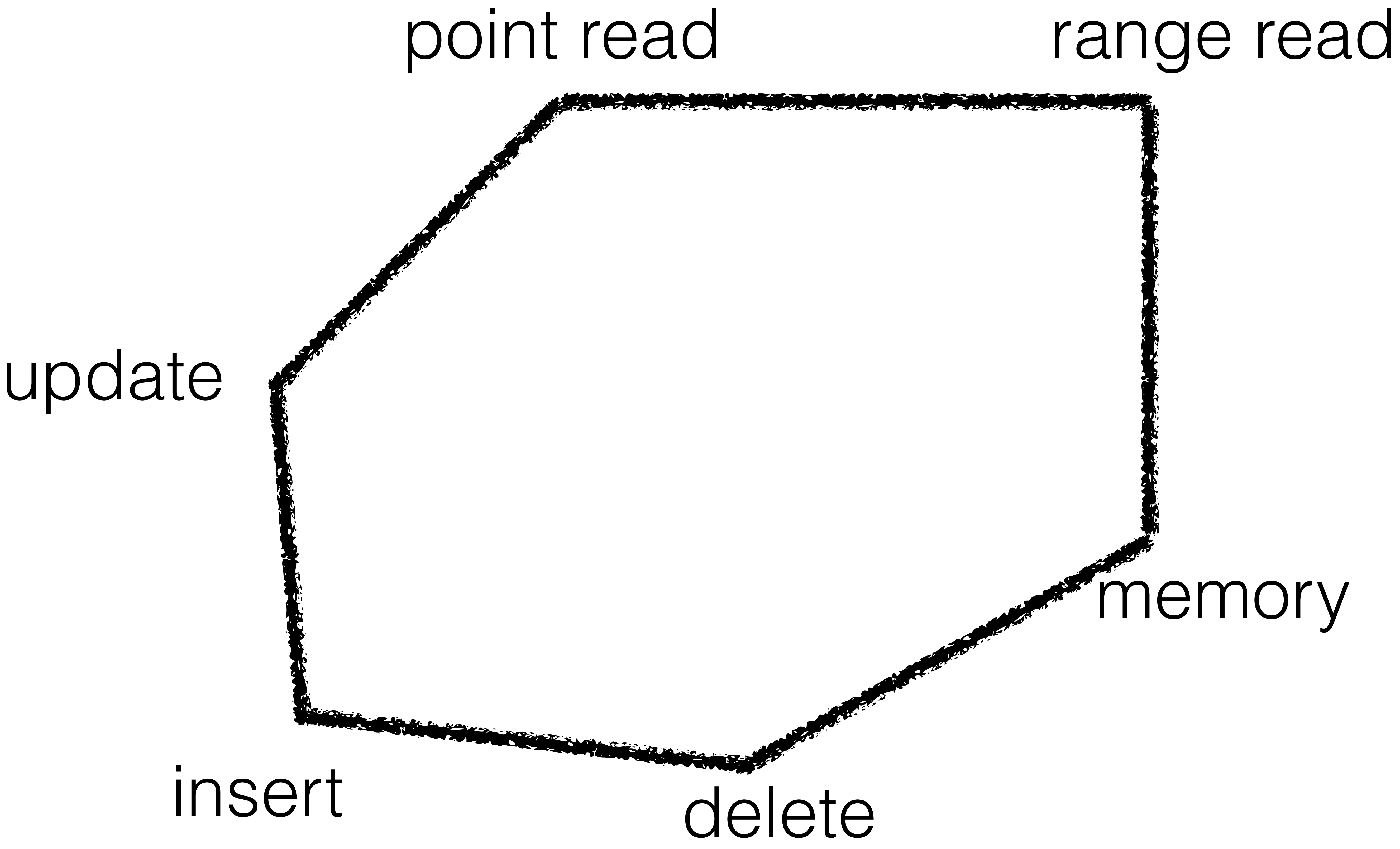
DASlab
@ Harvard SEAS

no perfect structure



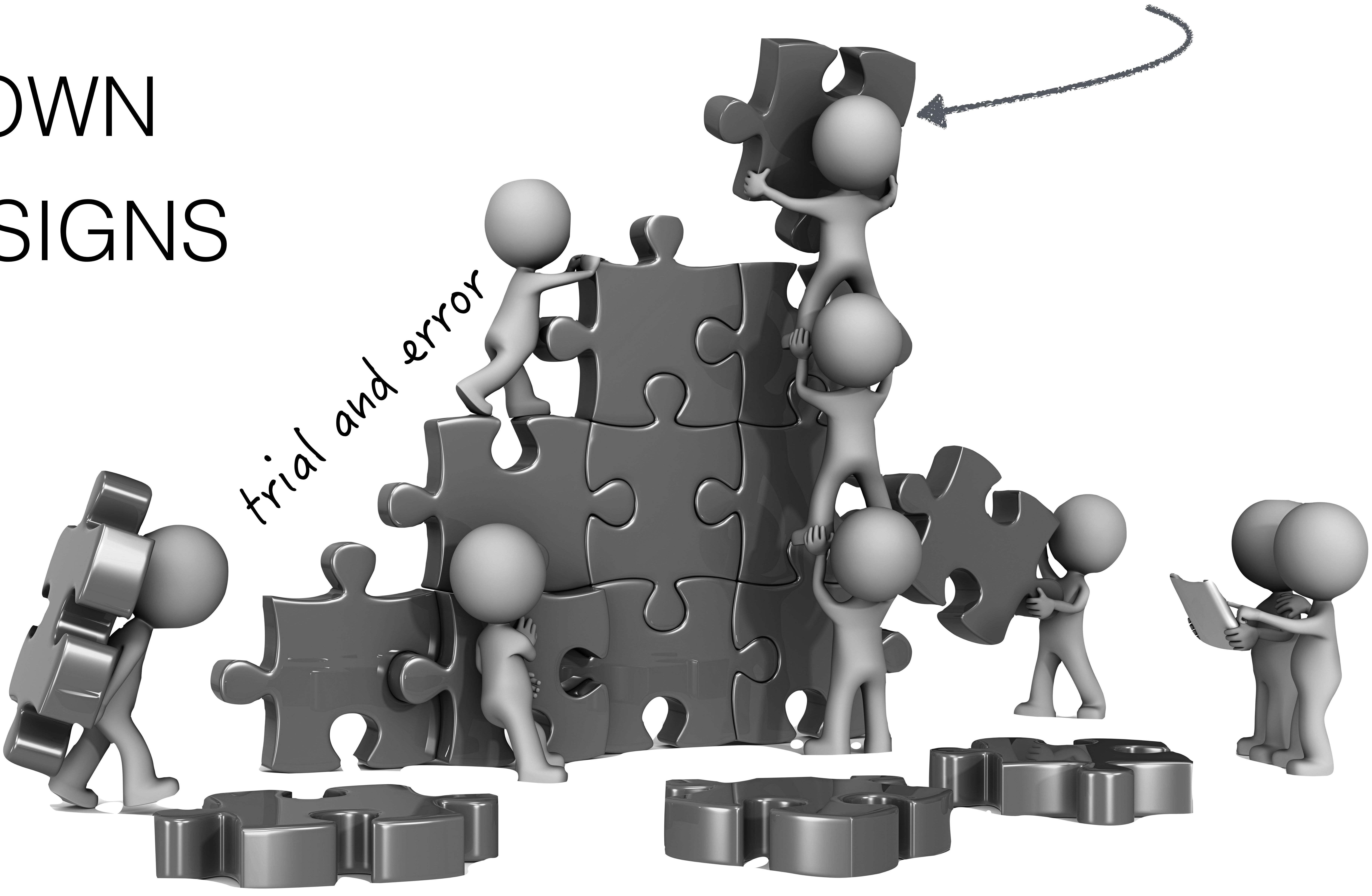
DASlab
@ Harvard SEAS


point read
range read
update
memory


point read
range read
update
memory
insert
delete

FIRST PRINCIPLE: DESIGN CONCEPT THAT IS NOT POSSIBLE OR MEANINGFUL TO BREAK FURTHER

BREAK DOWN KNOWN DESIGNS

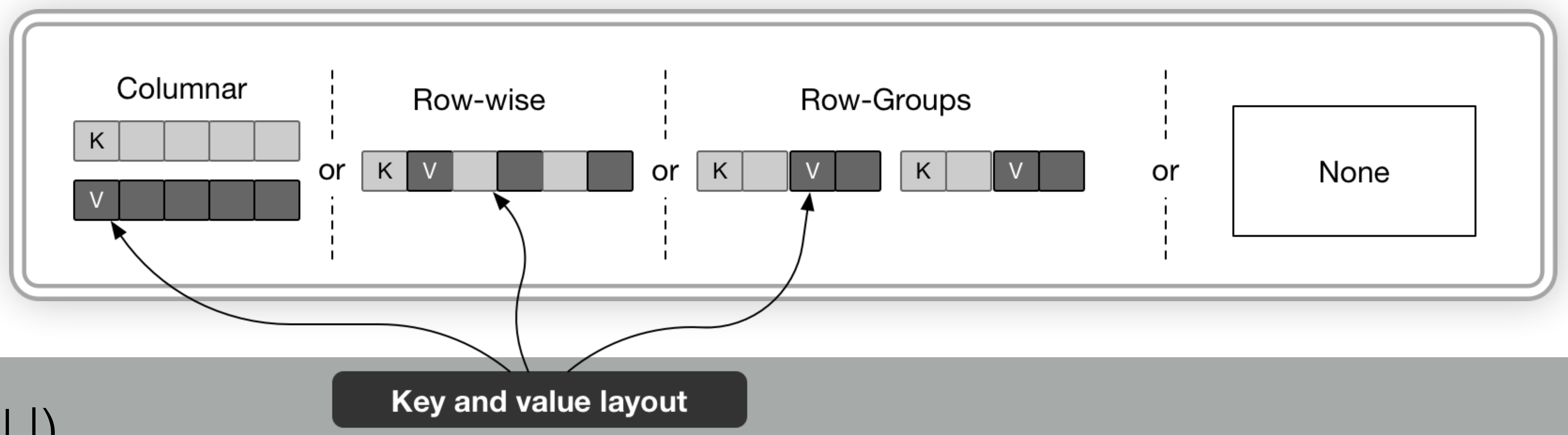


DASlab @ Harvard SEAS

Are keys retained? (yes, no, function)

Are values retained?

Utilization? (e.g., >50%)



Fanout (fixed/functional | unlimited | terminal |)

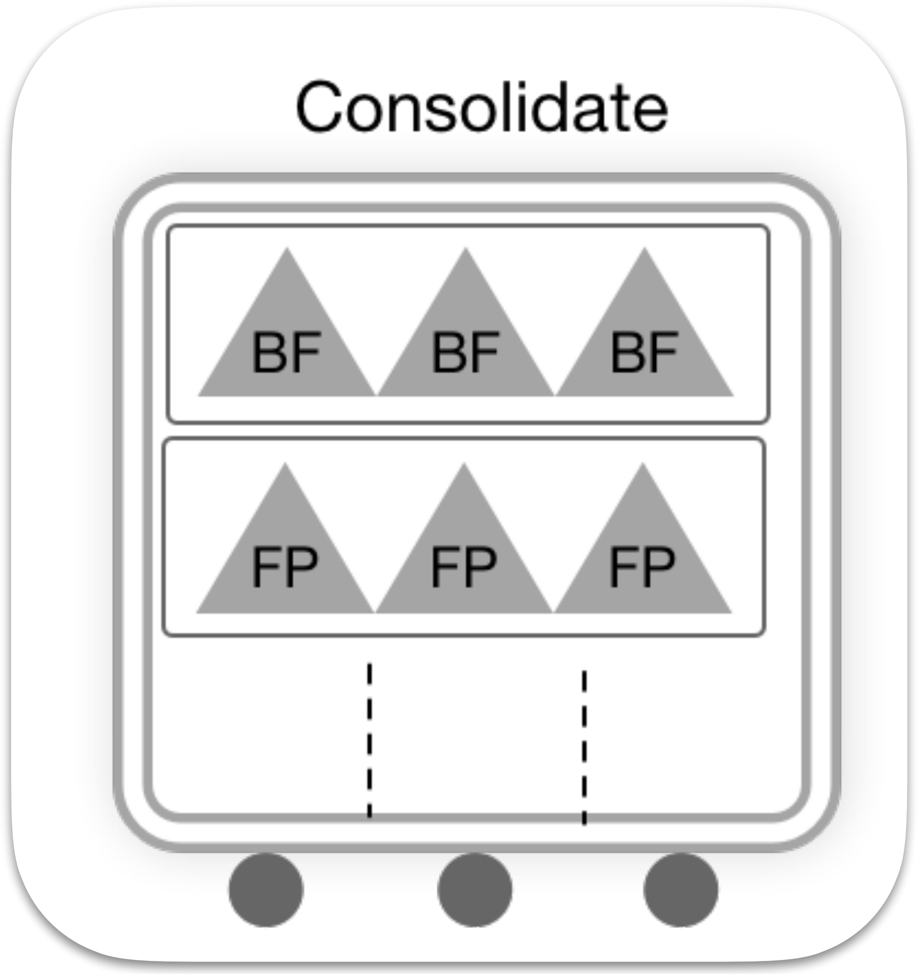
Key partitioning (none(fw-append | bw-append) | sorted | range() | radix() | function (func) | temporal(…))

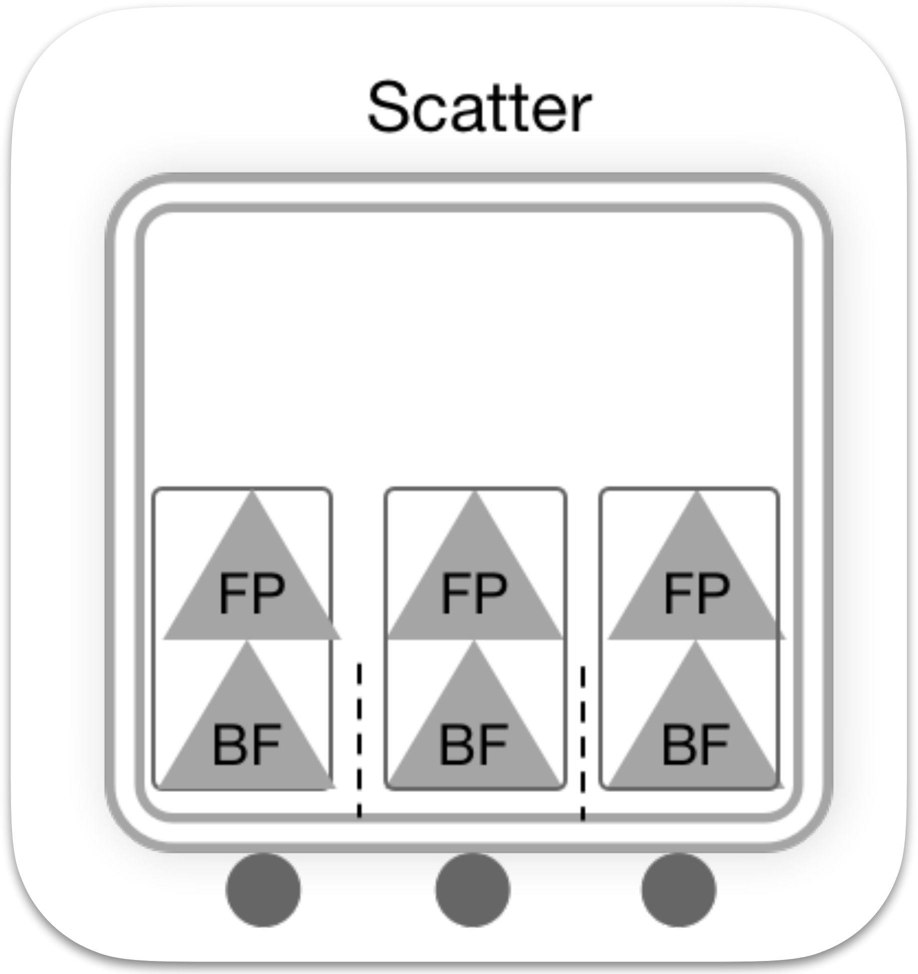
Zone Maps (min | max | both | exact | off)

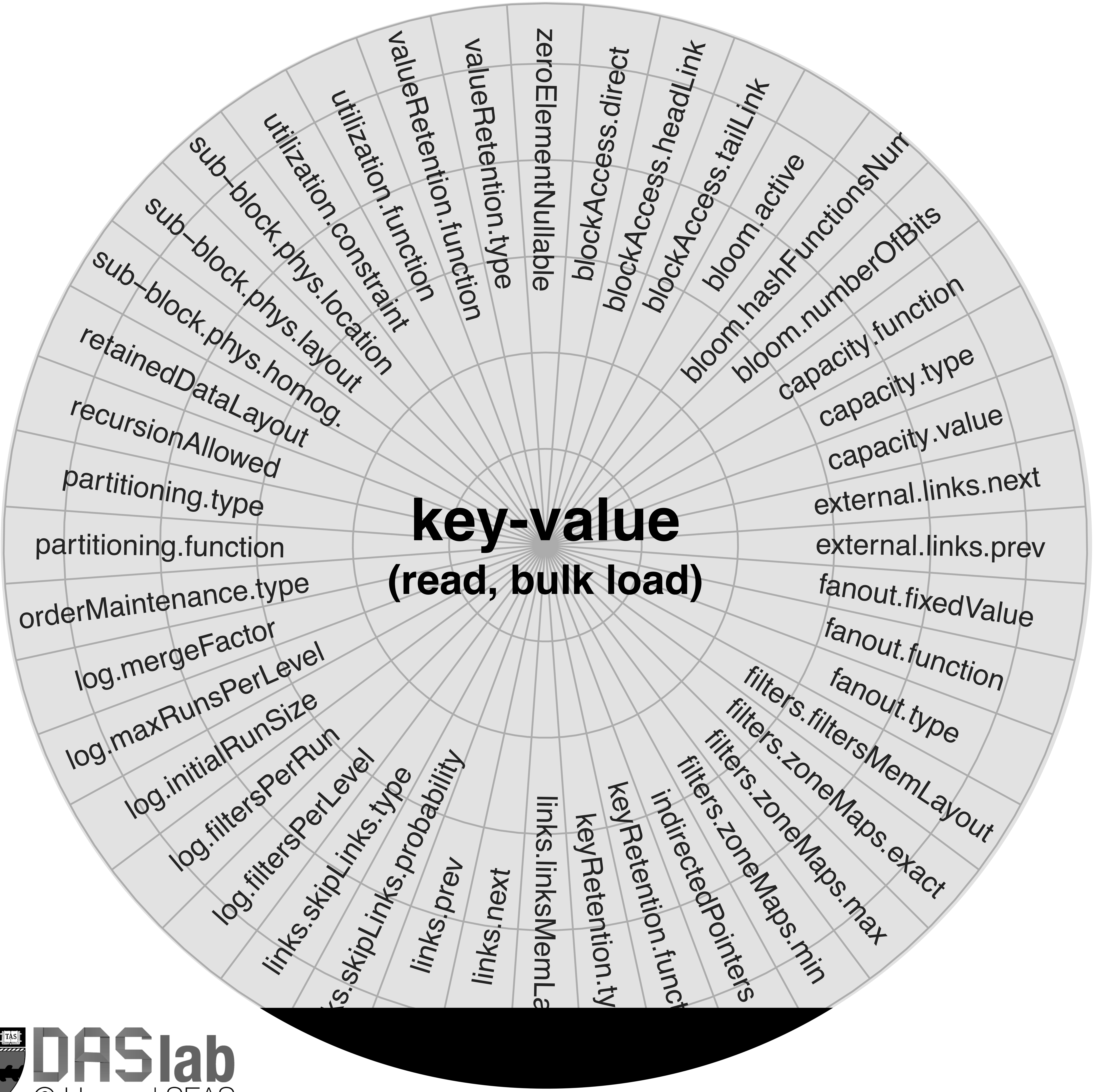
Bloom filters (off | on(num_hashes: int, num_bits: int))

Filters layout (consolidate | scatter)

Links layout (consolidate | scatter)





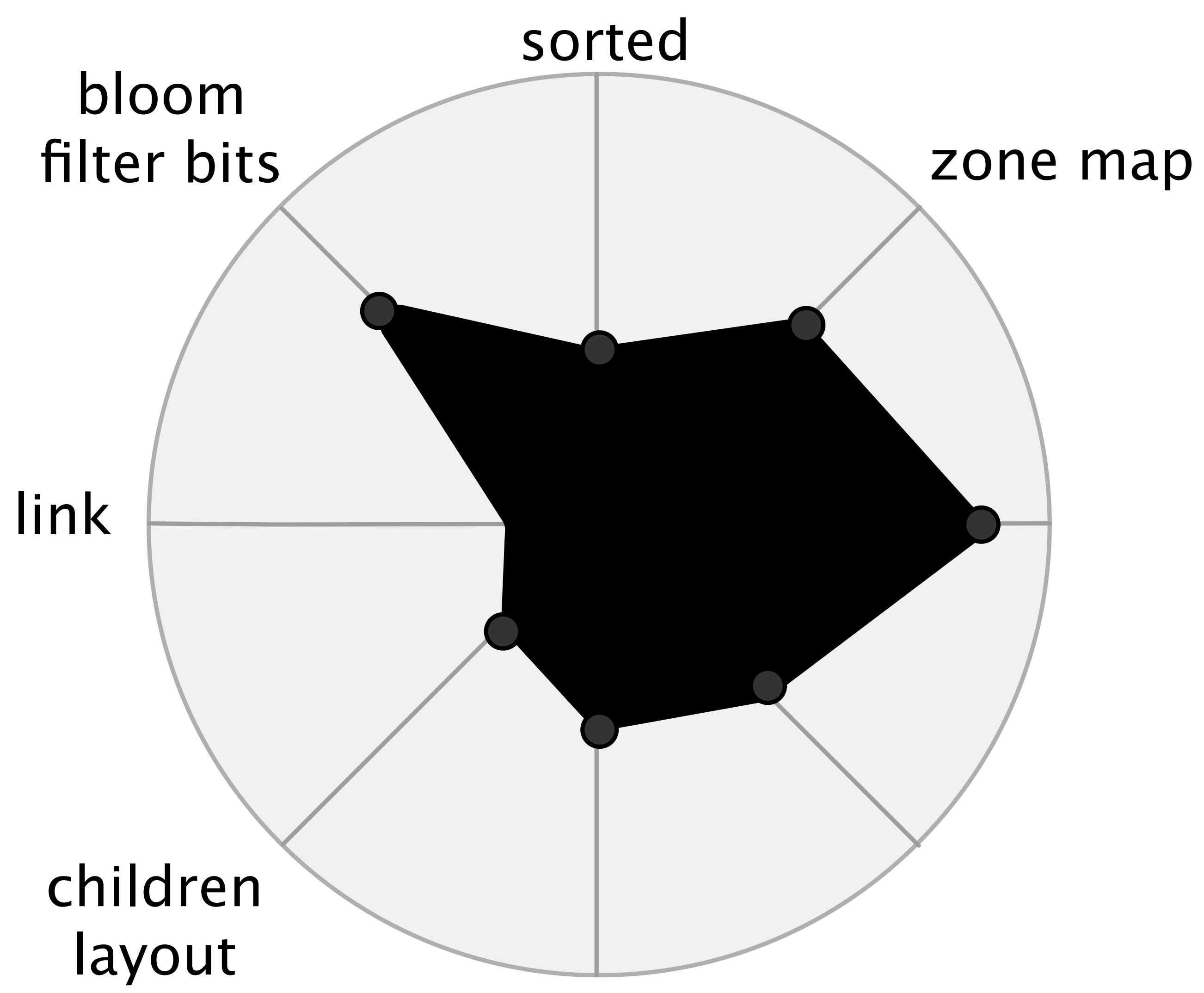


DESCRIBE ONE DATA BLOCK AT A TIME

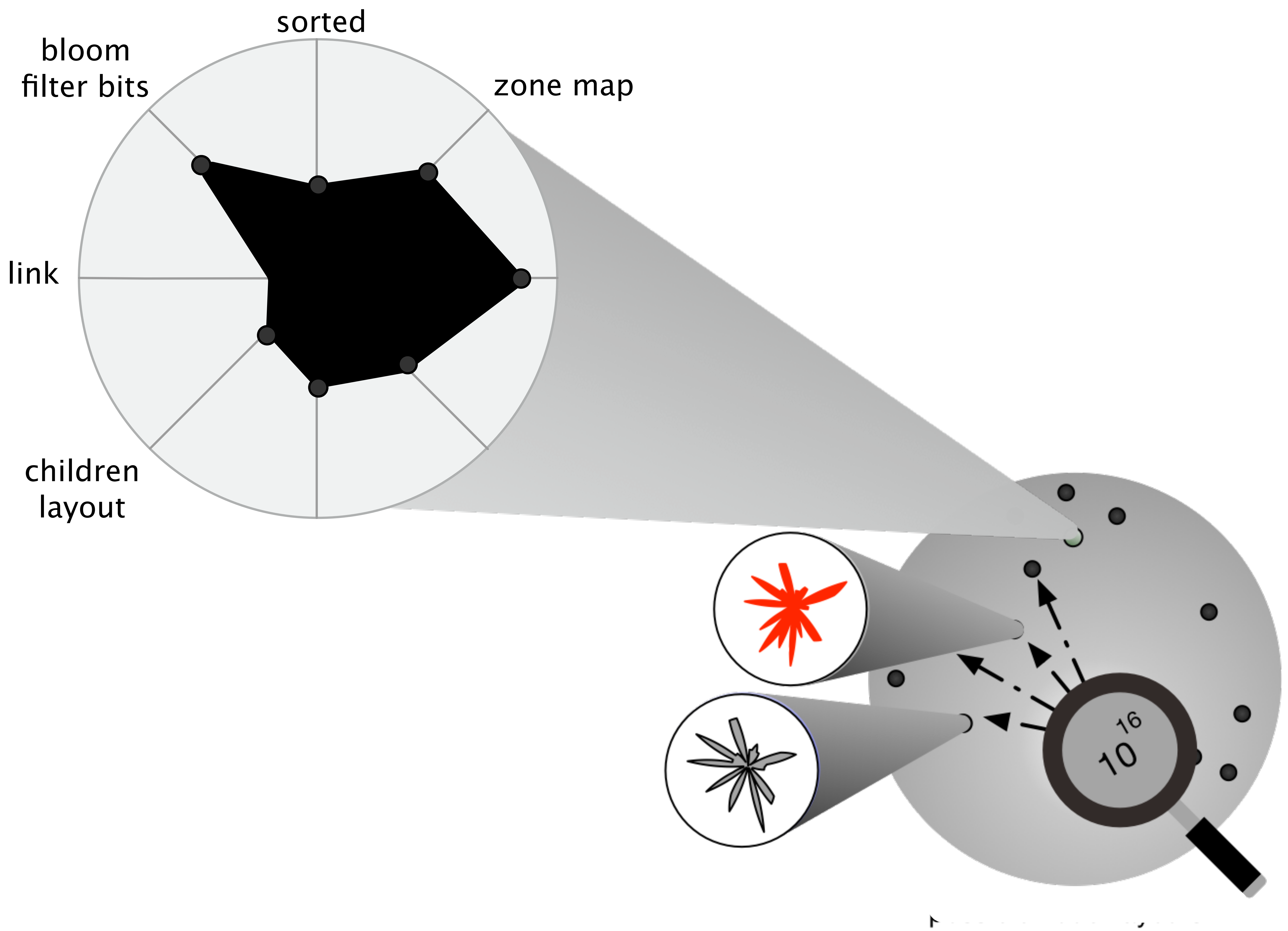
AS A SET OF CONCEPTS

physical layout and domain partitioning

SETS OF CONCEPTS

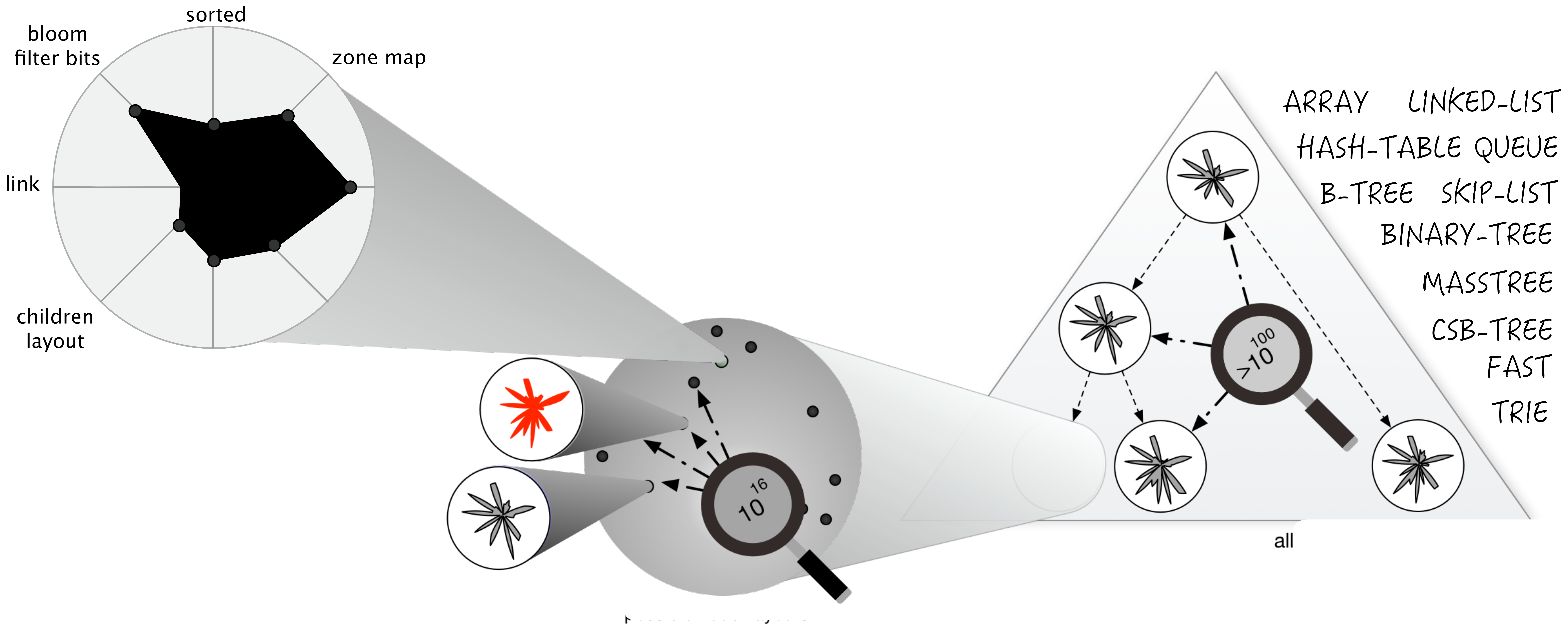


SETS OF CONCEPTS   POSSIBLE NODE DESIGNS



DASlab
@ Harvard SEAS

SETS OF CONCEPTS   POSSIBLE NODE DESIGNS   POSSIBLE STRUCTURES



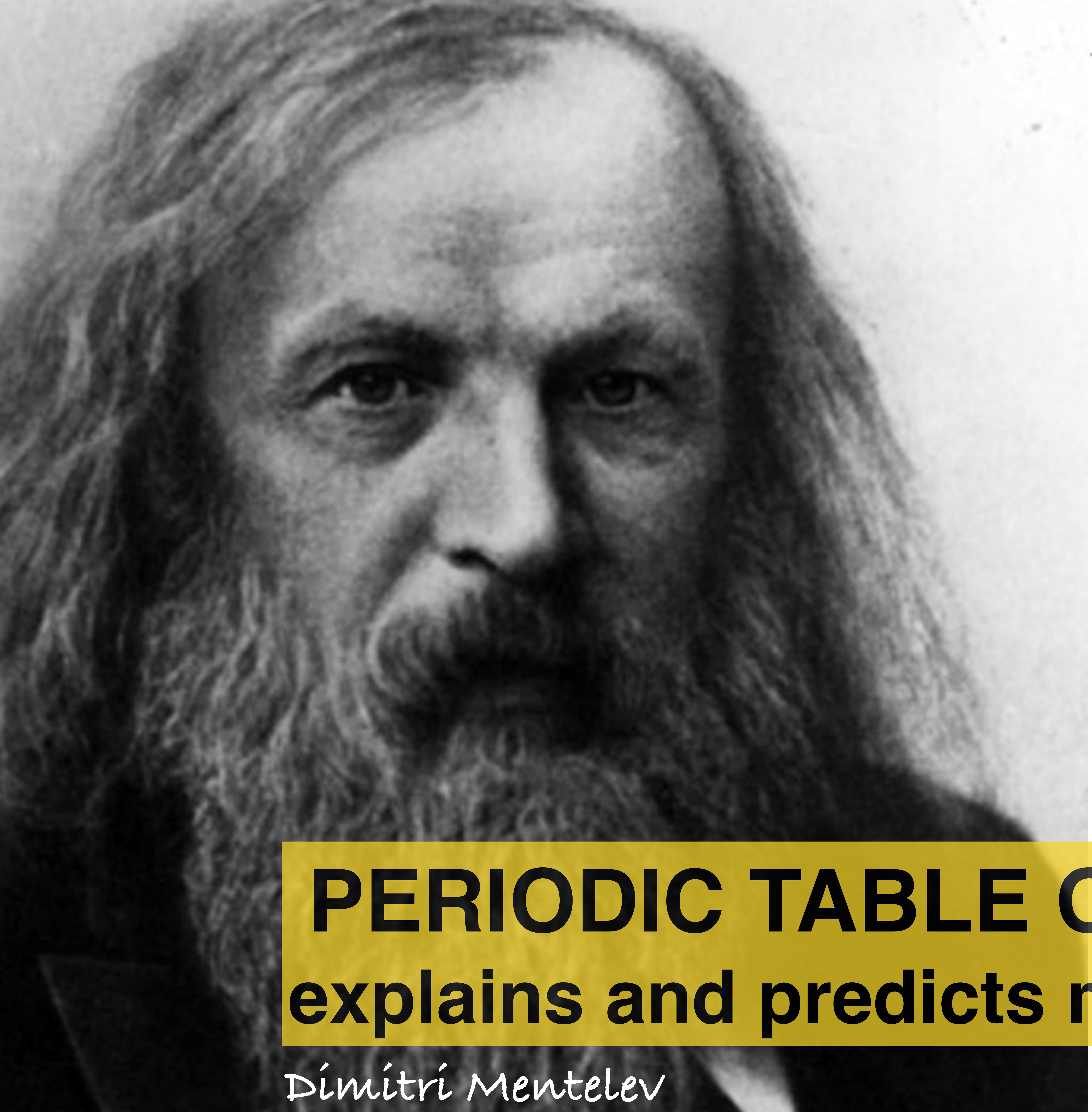
Design Primitives to Auto Generate Trillions of Data Structures

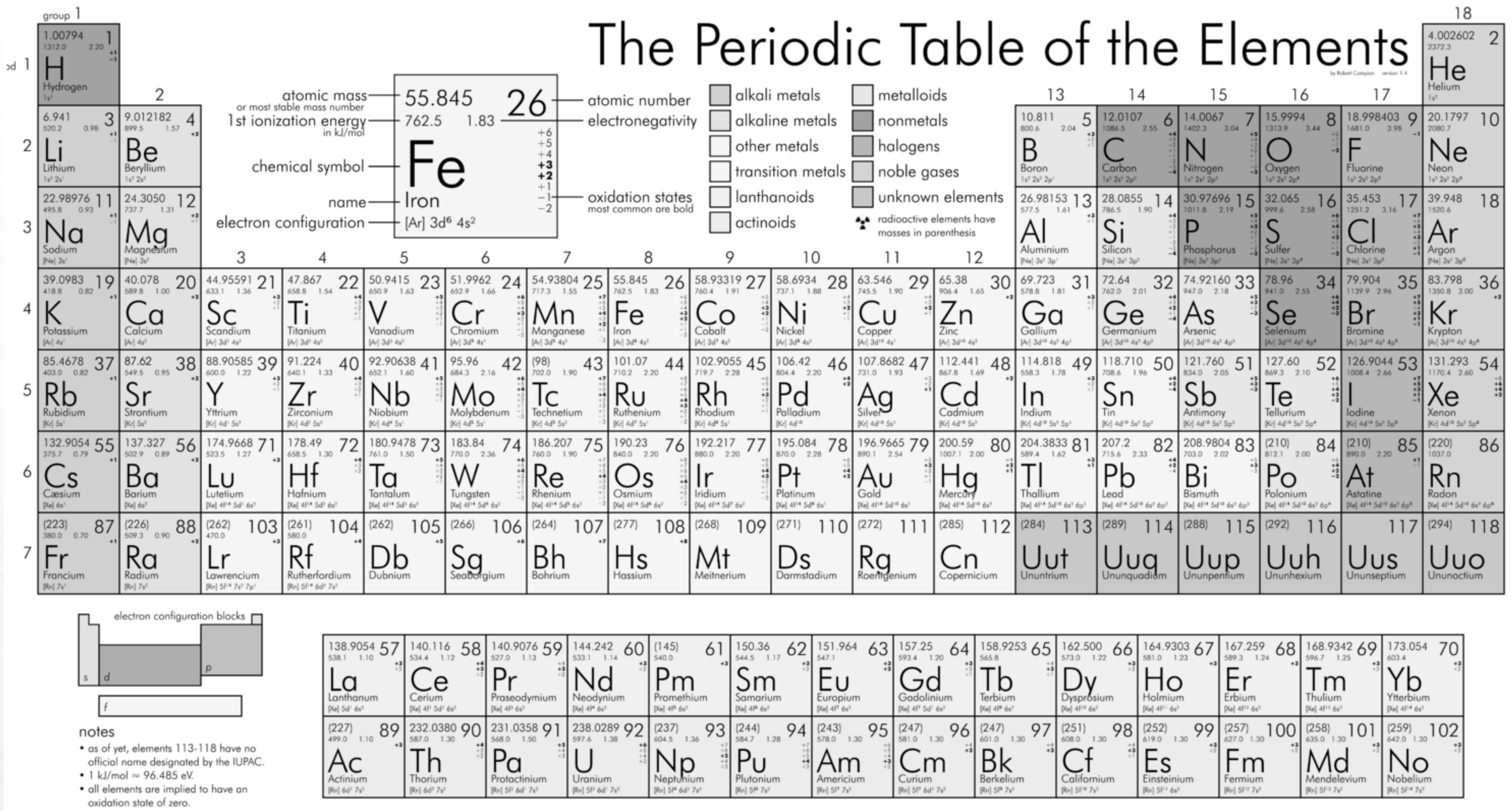
Unless otherwise specified, we use a reduced default values domain of 100 values for integers, 10 values for doubles, and 1 value for functions.

Categories		Primitive	Domain	size	Hash Table	LPL		B+Tree/CSB+Tree/FAST			
					H	LL	UDP	B+	CSB+	FAST	ODP
Node organization	1	**Key retention.** <u>*No:*</u> node contains no real key data, e.g., intermediate nodes of b+trees and linked lists. <u>*Yes:*</u> contains complete key data, e.g., nodes of b-trees, and arrays. <u>*Function:*</u> 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.** <u>*No:*</u> node contains no real value data, e.g., intermediate nodes of b+trees, and linked lists. <u>*Yes:*</u> contains complete value data, e.g., nodes of b-trees, and arrays. <u>*Function:*</u> 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. <u>Rules:</u> 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. <u>Rules:</u> 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) | mited | 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)
	11	**Key partitioning.** Set if there is a pre-defined key partitioning imposed, e.g. the									

Partiti[on]		… or balanced (i.e., all sub-blocks have the same size), unrestricted or functional. Rules: requires key partitioning != none.	fixed(value: int) | balanced | [unre]stricted | function(func) *(up to 10 different fixed capacity values are considered)*	13	unrestri[cted]	fixed(25…		balance[d]	balance[d]	balance[d]	
	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. 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. Rules: requires fanout/radix != terminal.	BFS | BFS layer(level-grouping: [i]nt) | scatter *(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. Rules: requires fanout/radix != terminal.	boolean	2	true	true		true	true	true	
	19	**Sub-block consolidation.** Single children are merged with their parents. 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). 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). 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. Rules: requires fanout/radix != terminal.	yes(func) | no	3	no	no		yes(logn)	yes(logn)	yes(logn)	

Categories





structures elements based on atomic number,
electron configuration, and recurring chemical properties

PERIODIC TABLE OF ELEMENTS

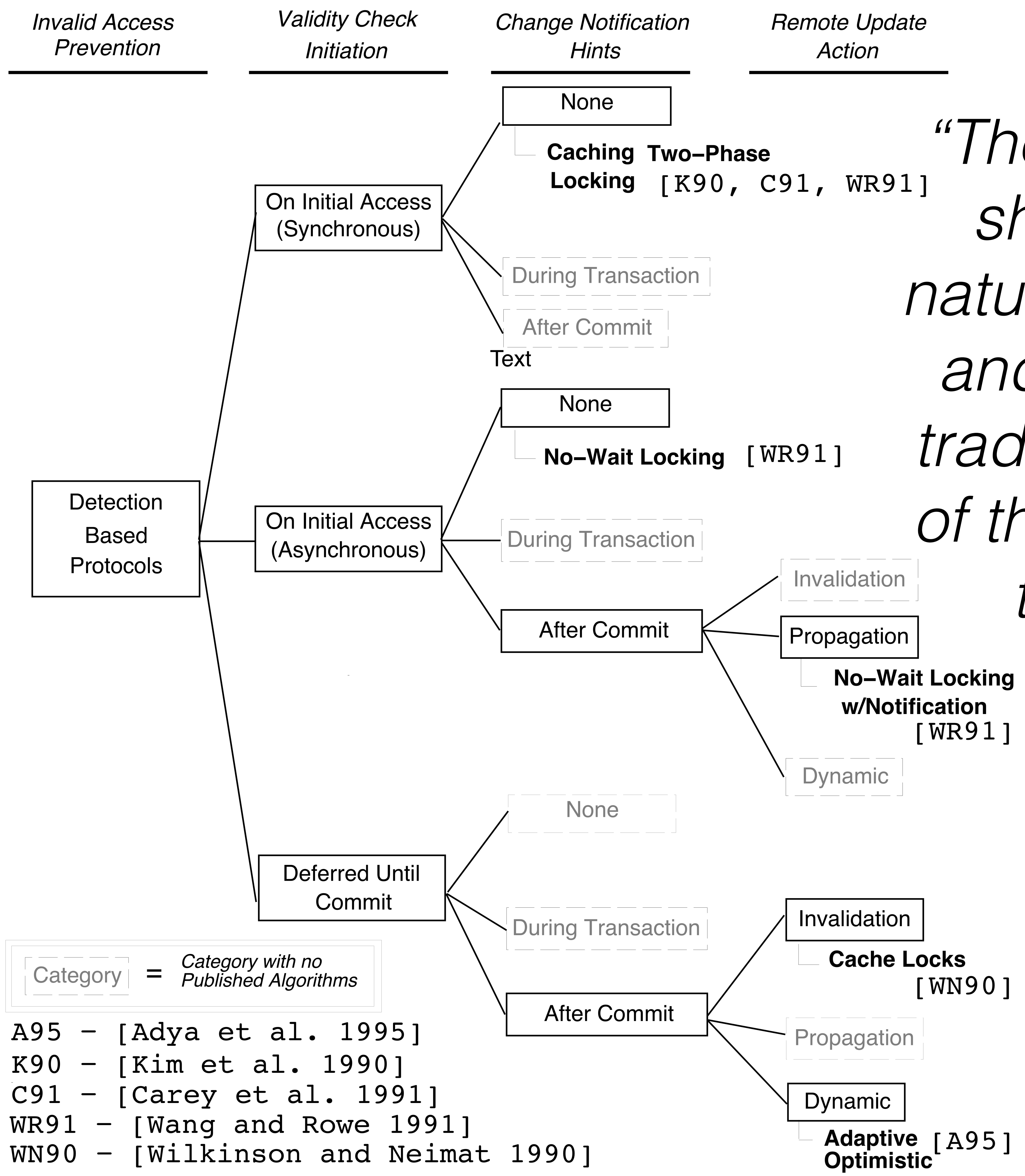
explains and predicts missing elements

Dimitri Mentelev

原子番号（陽子数）
元素名（和名）
元素記号
水素
H
超重元素
113
nihonium
理化学研
Kosuke Morita

the 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	R U M
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	



"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 ESTIMATION

SEARCH

HOW TO JUDGE A DESIGN?

1

COMPLEXITY ANALYSIS

2

IMPLEMENTATION & TESTING

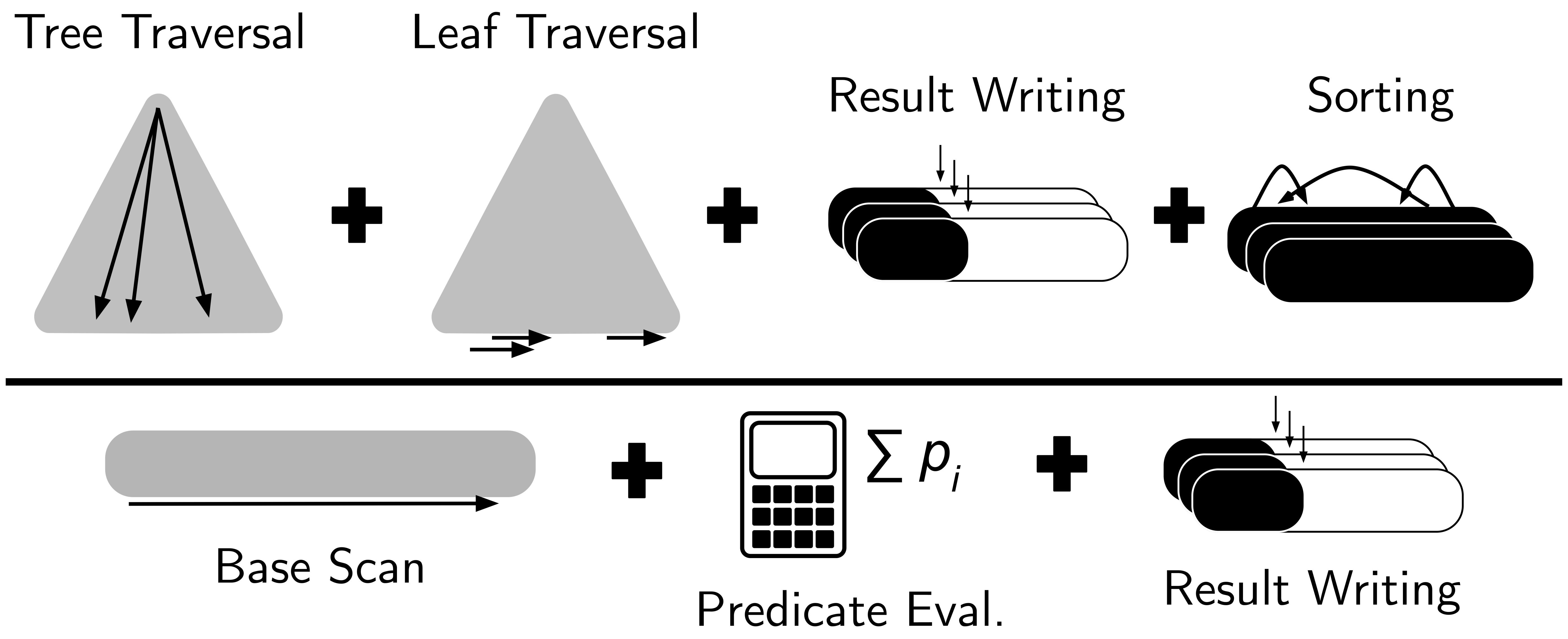
3

GENERALIZED MODELS

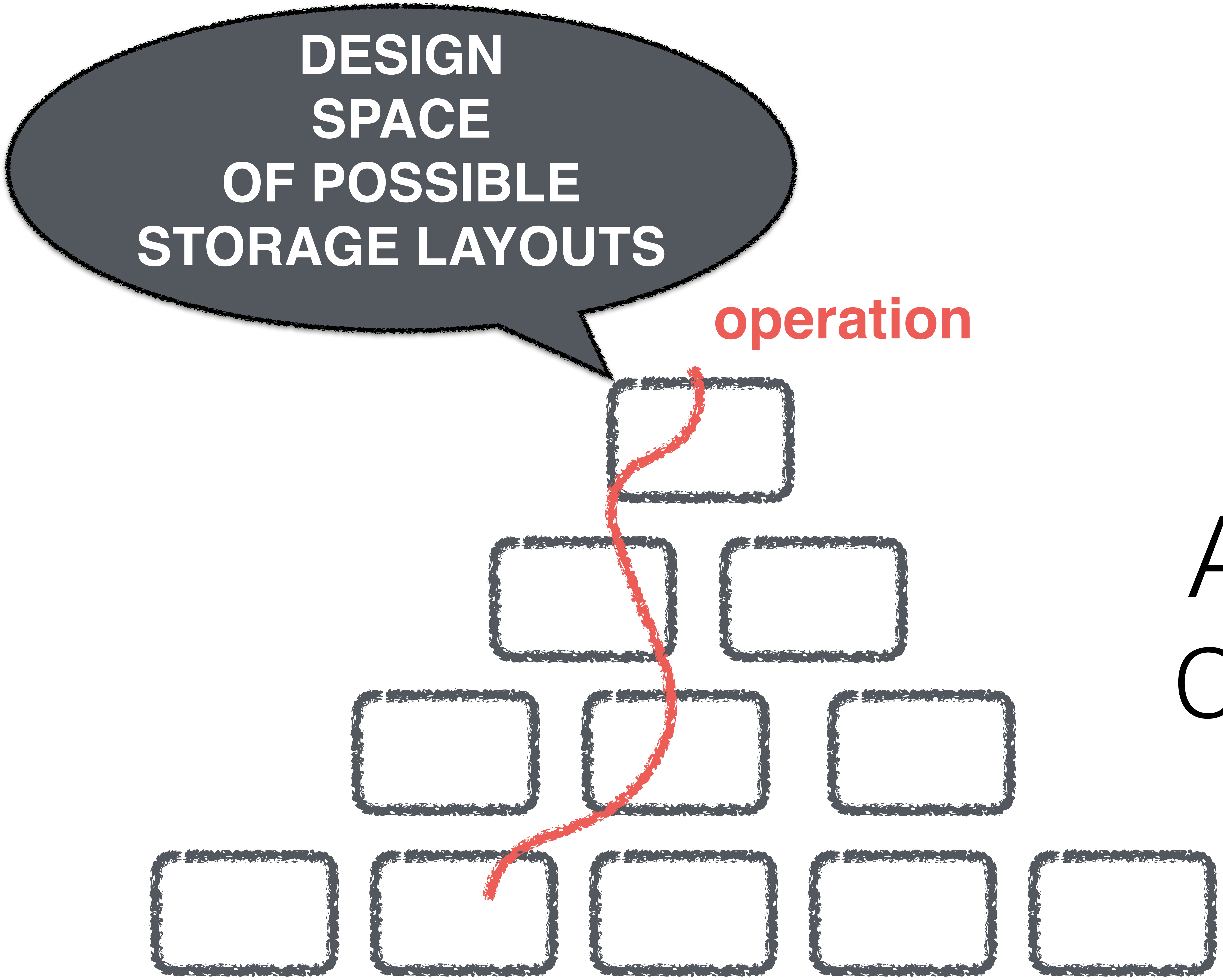
HARD & SLOW

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

Access path selection @SIGMOD2017

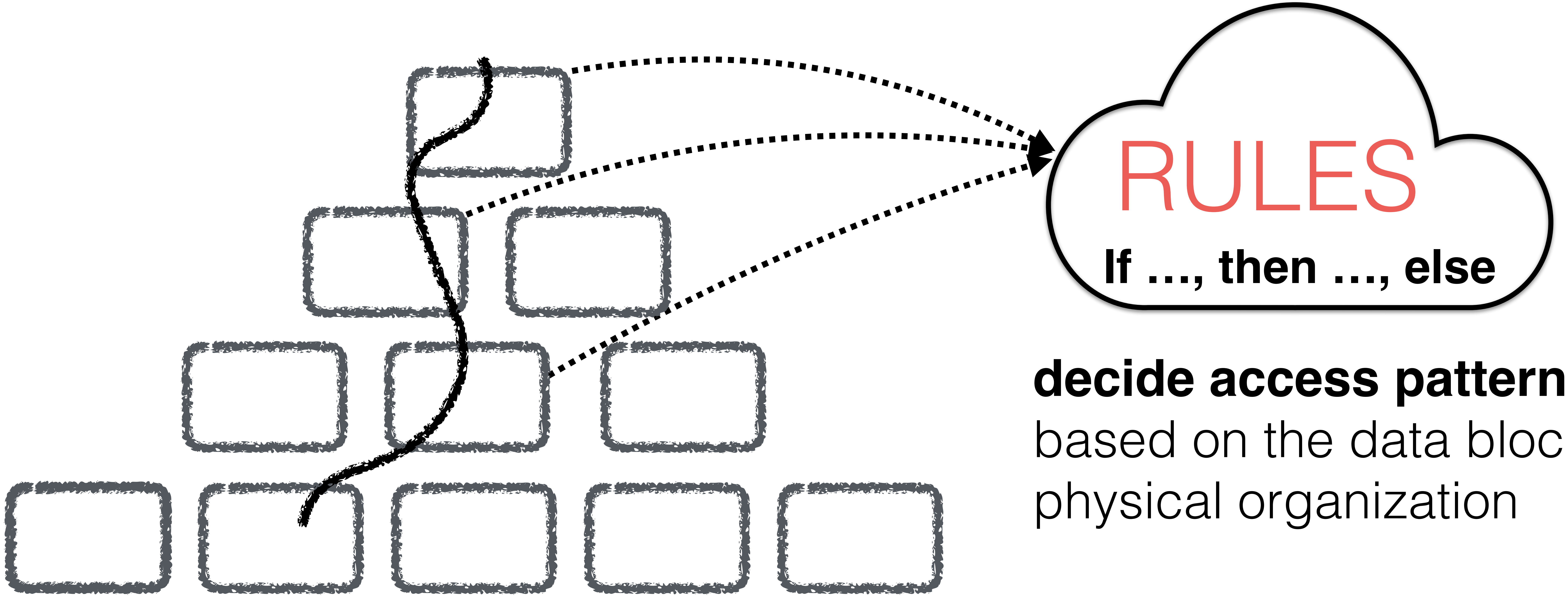


Workload	q	number of queries
	s_i	selectivity of query i
	S_{tot}	total selectivity of the workload
Dataset	N	data size (tuples per column)
	ts	tuple size (bytes per tuple)
Hardware	C_A	L1 cache access (sec)
	C_M	LLC miss: memory access (sec)
	BW_S	scanning bandwidth (GB/s)
	BW_R	result writing bandwidth (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)



ALGORITHM &
COST SYNTHESIS

DASlab
@ Harvard SEAS



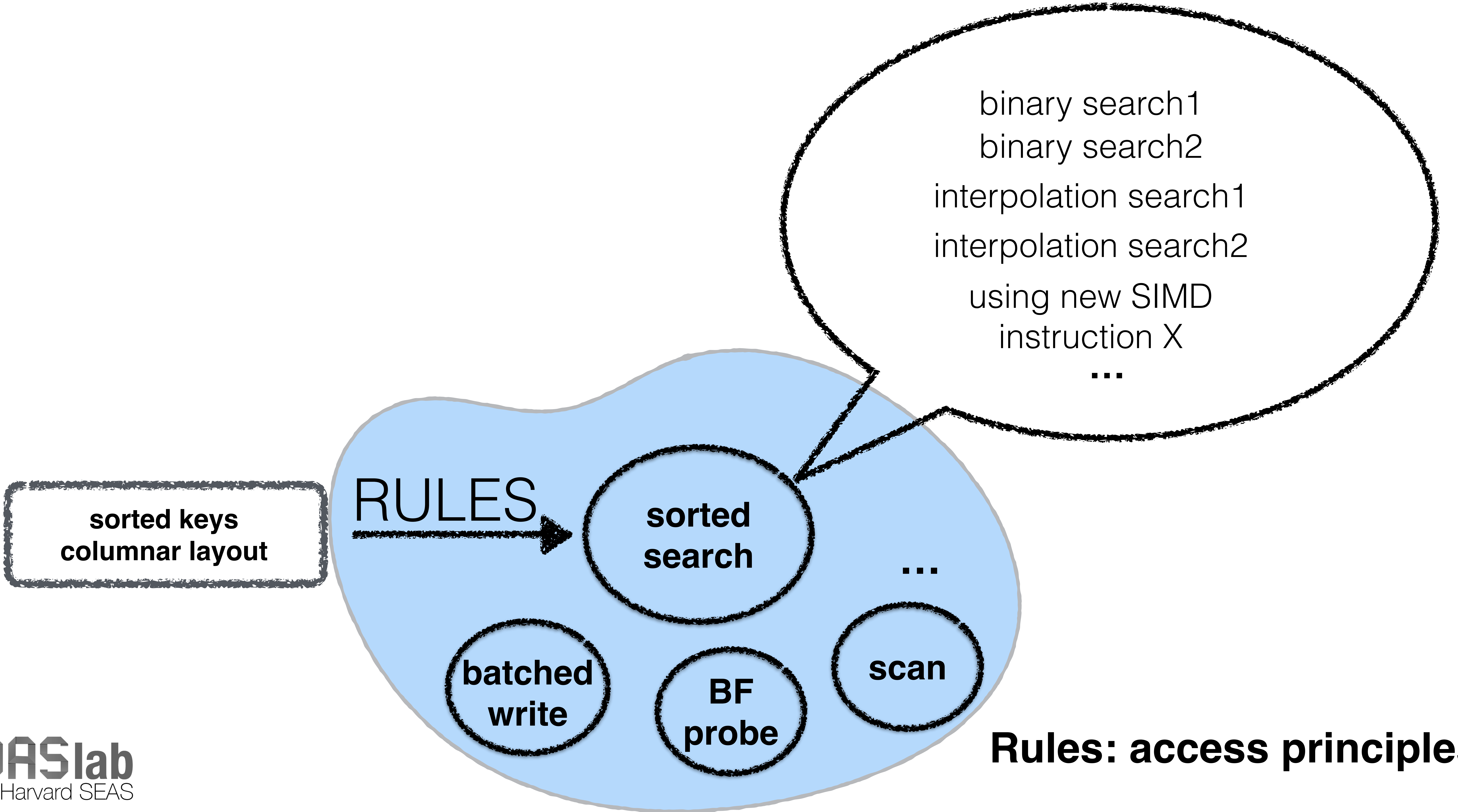
decide access pattern
based on the data block's
physical organization


sorted keys
columnar layout
RULES
sorted
search

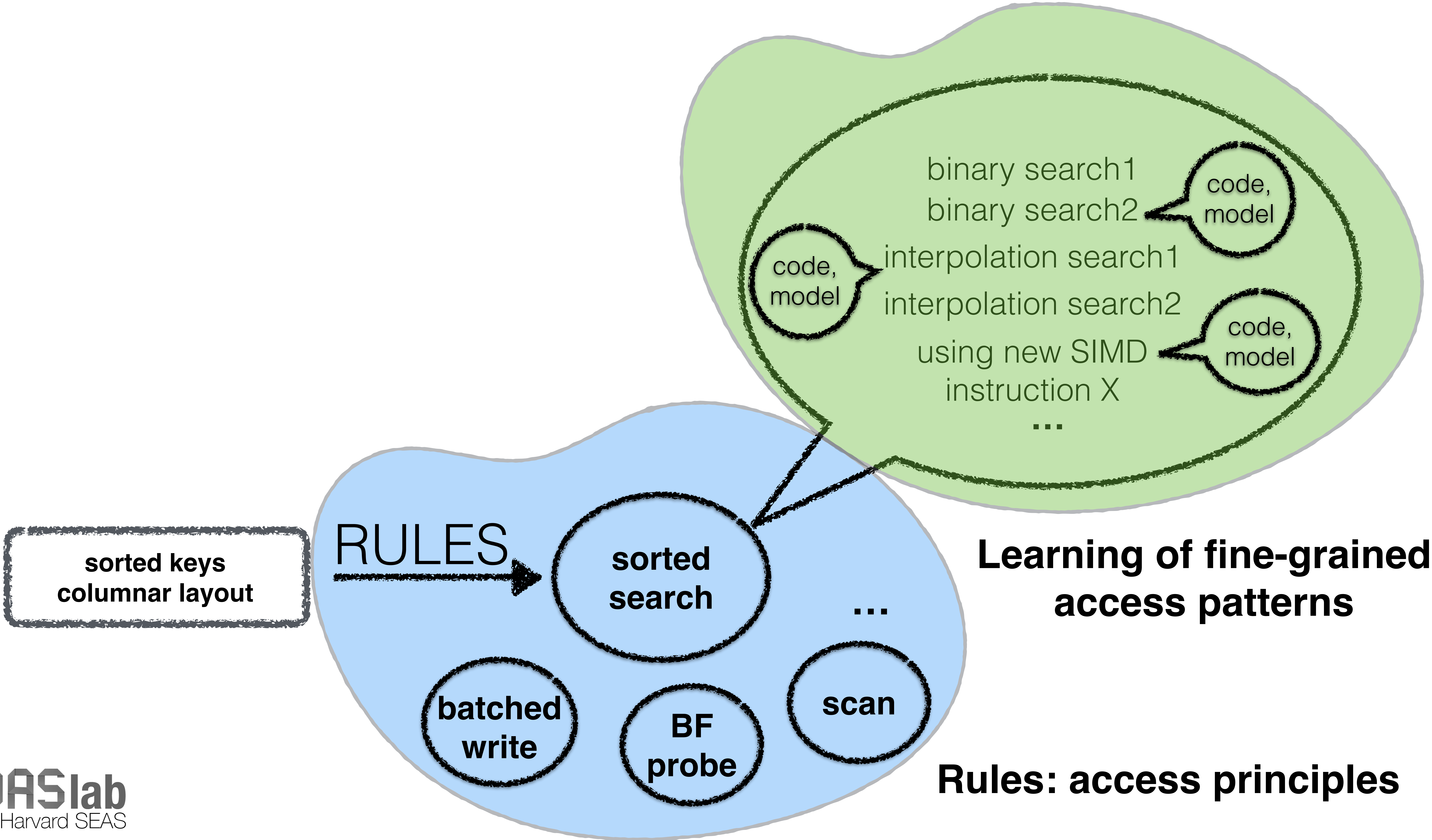
DEPENDS ON **HARDWARE ENGINEERING**

binary search1
binary search2
interpolation search1
interpolation search2
using new SIMD instruction X
...

sorted keys columnar layout → RULES → **sorted search**



Rules: access principles



Learning of fine-grained access patterns

Rules: access principles

DASlab @ Harvard SEAS

SYNTHESIS FROM LEARNED MODELS

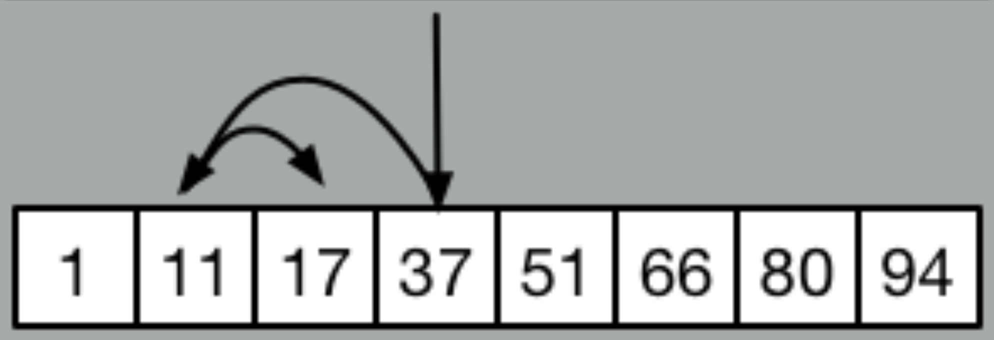
coding, modeling, generalized models, and a touch of ML

1. MINIMAL CODE

e.g., binary search

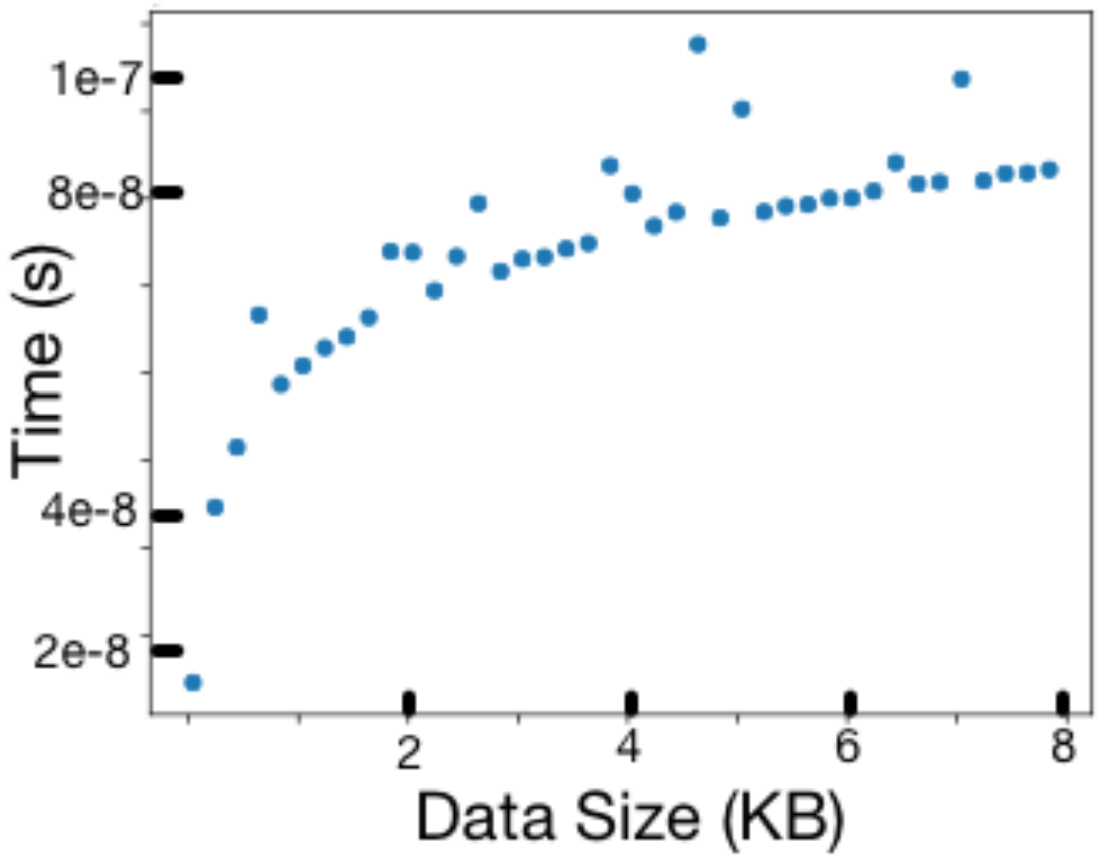
C++

```
if (data[middle] < search_val) {
    low = middle + 1;
} else {
    high = middle;
}
middle = (low + high)/2;
```



Run

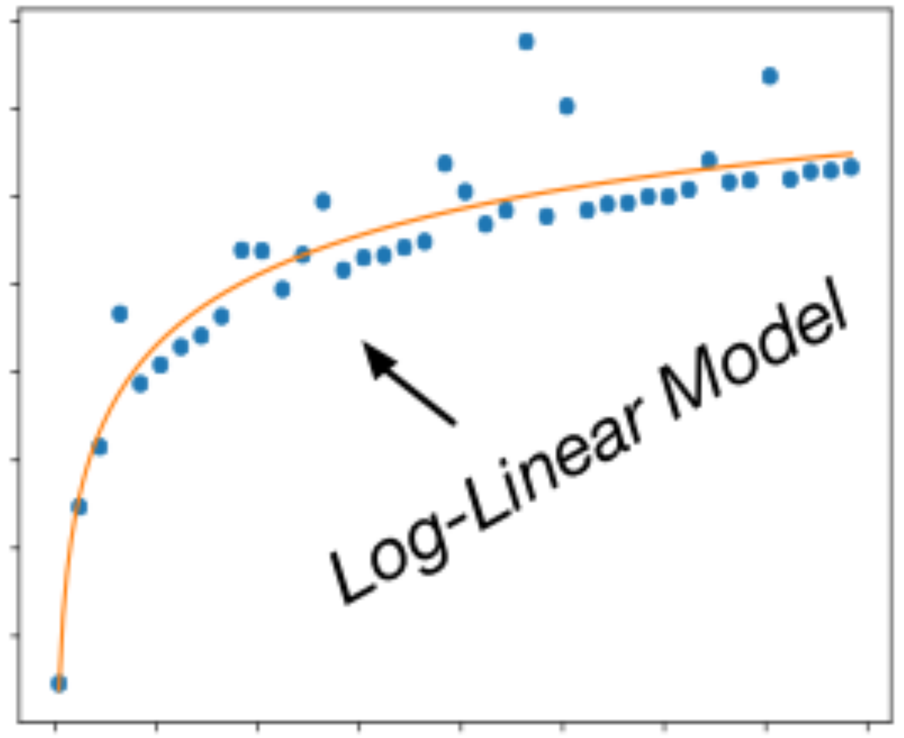
2. BENCHMARK



$f(x)$

Train

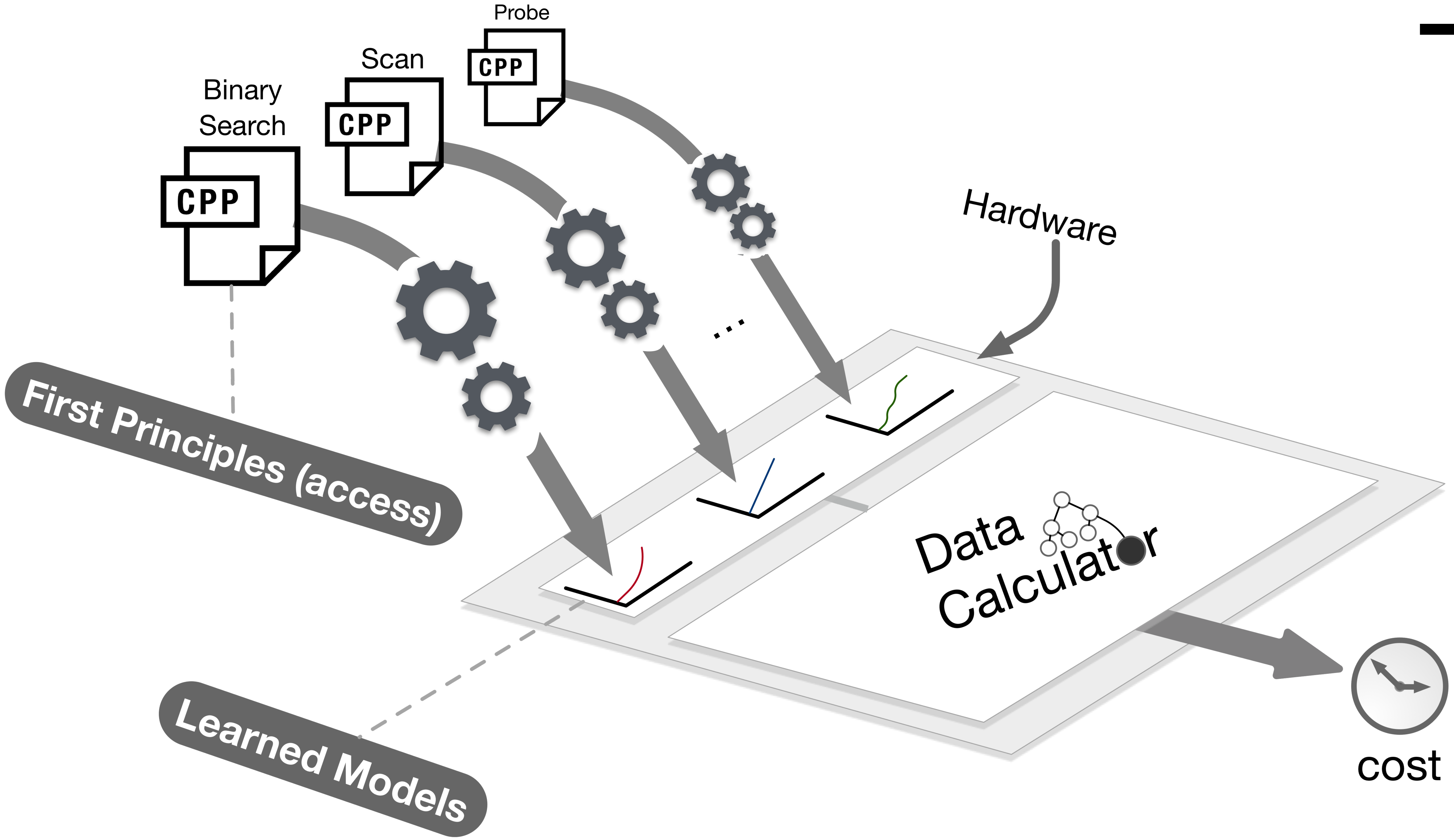
3. FIT MODEL



$$f(x) = ax + b\log x + c$$

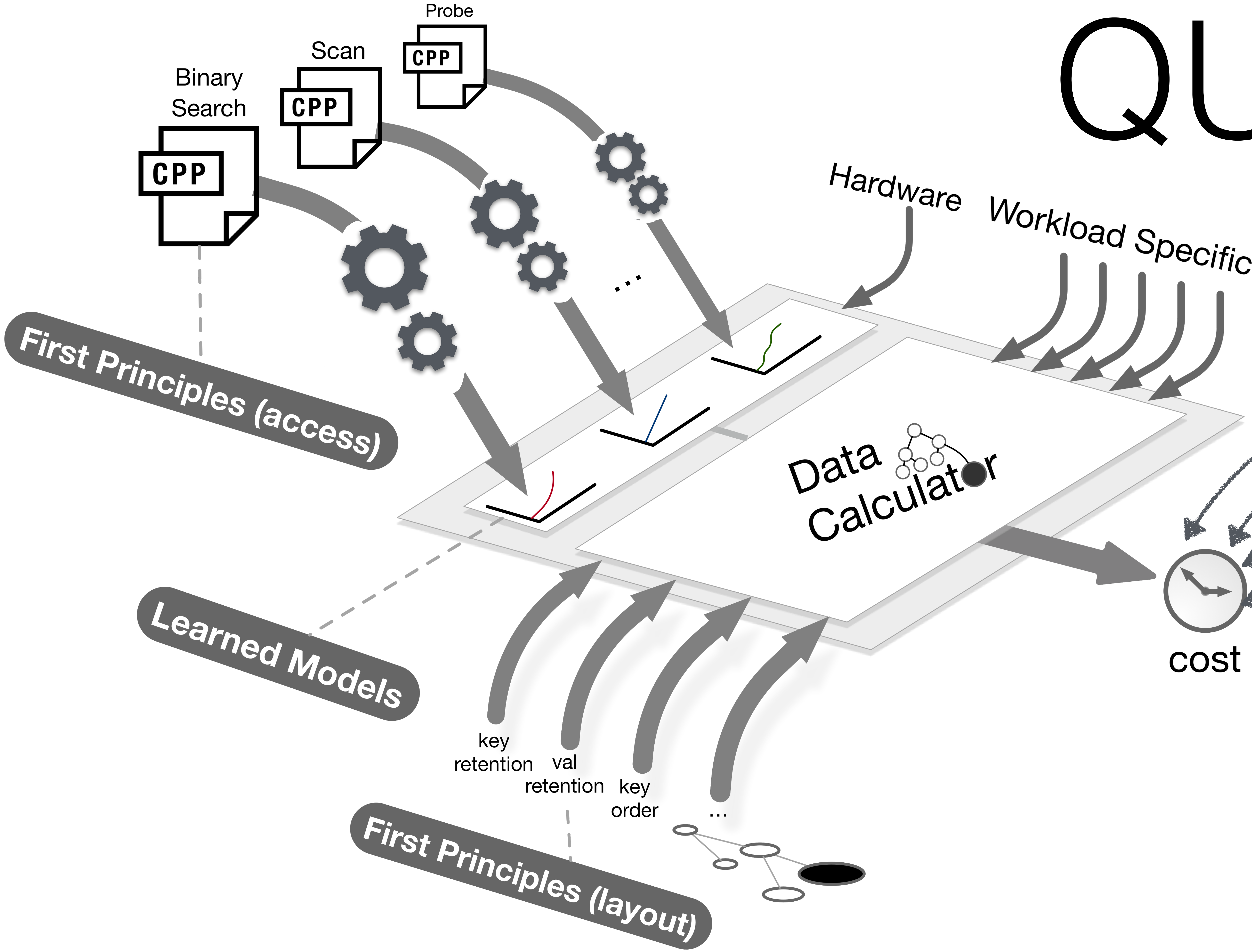
FOLDING ALGORITHMIC, ENGINEERING, AND H/W, PROPERTIES INTO THE COEFFICIENTS

TRAINING



Probe
CPP
Scan
CPP
Binary Search
CPP
Hardware
...
First Principles (access)
Data Calculator
Learned Models
cost

QUEYRYING



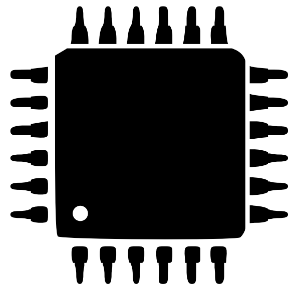
FOR EACH OPERATION

FOR EACH NODE

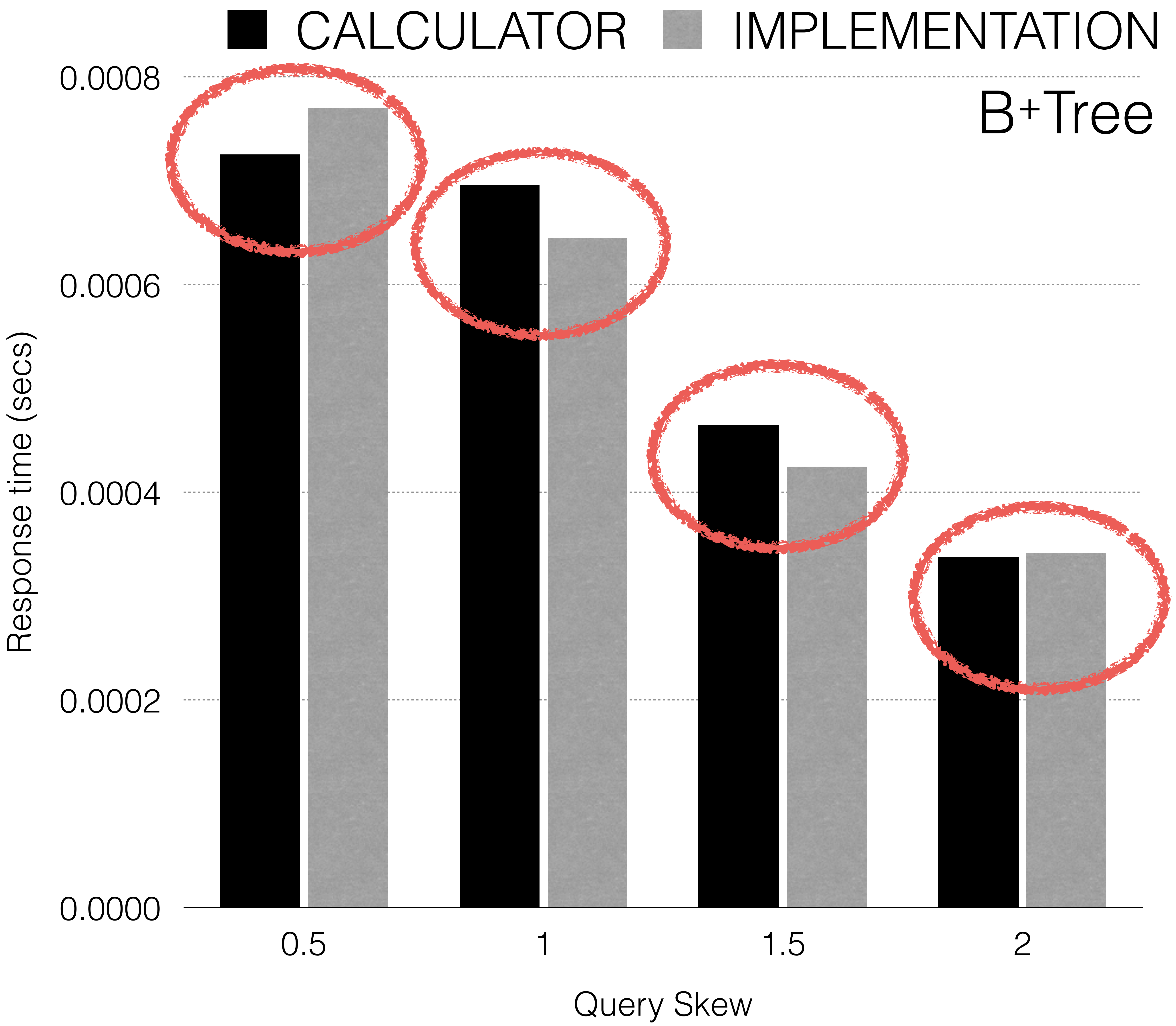
1. Decide access **strategy** (L1) based on node design
2. Decide exact access strategy **implementation** (L2) based on available models
3. **Get cost** for chosen model

DASlab @ Harvard SEAS

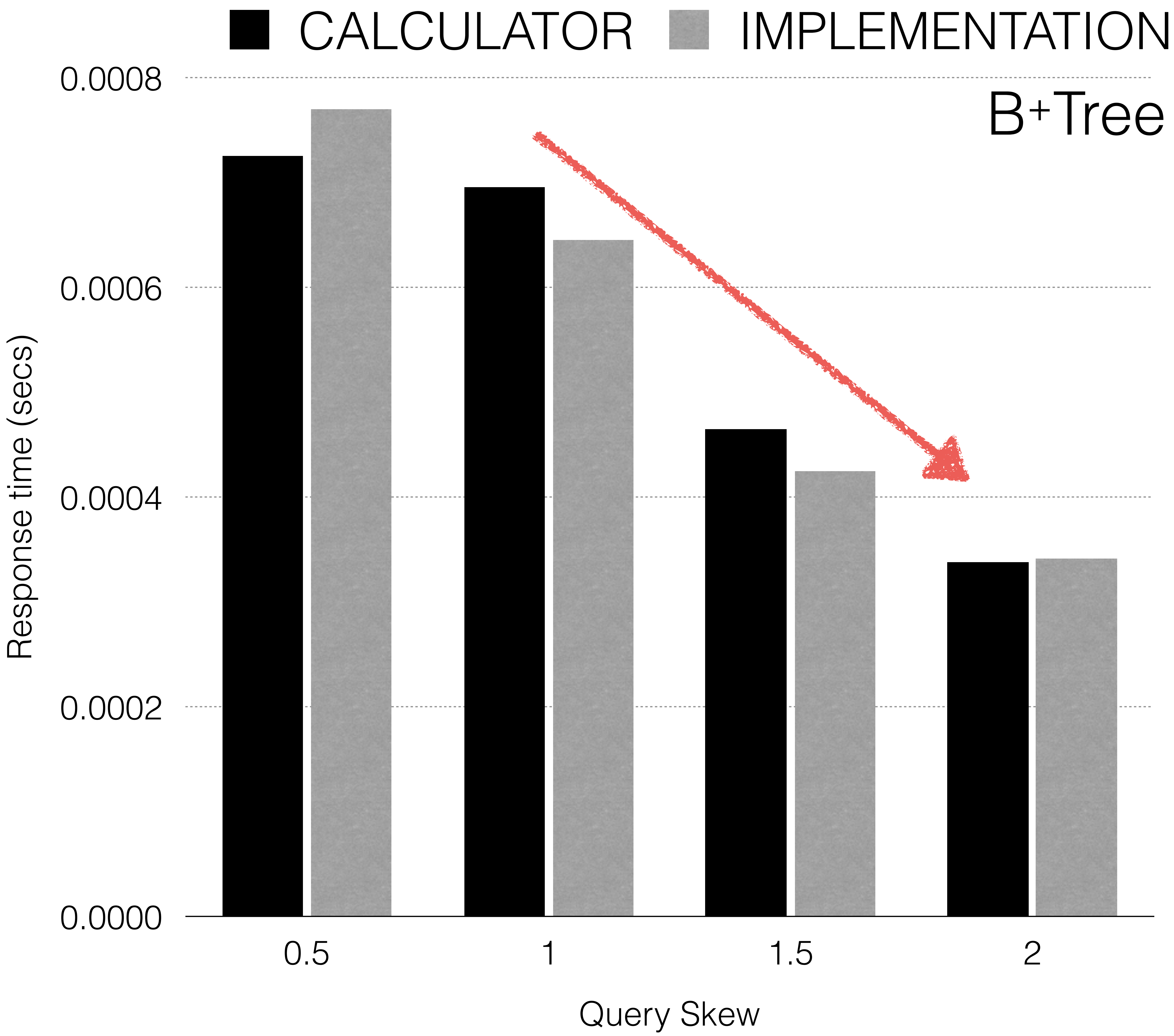
CAN WE COMPUTE PERFORMANCE ACCURATELY?

layout spec → **DC** → cost **VS** **C++** →  → cost

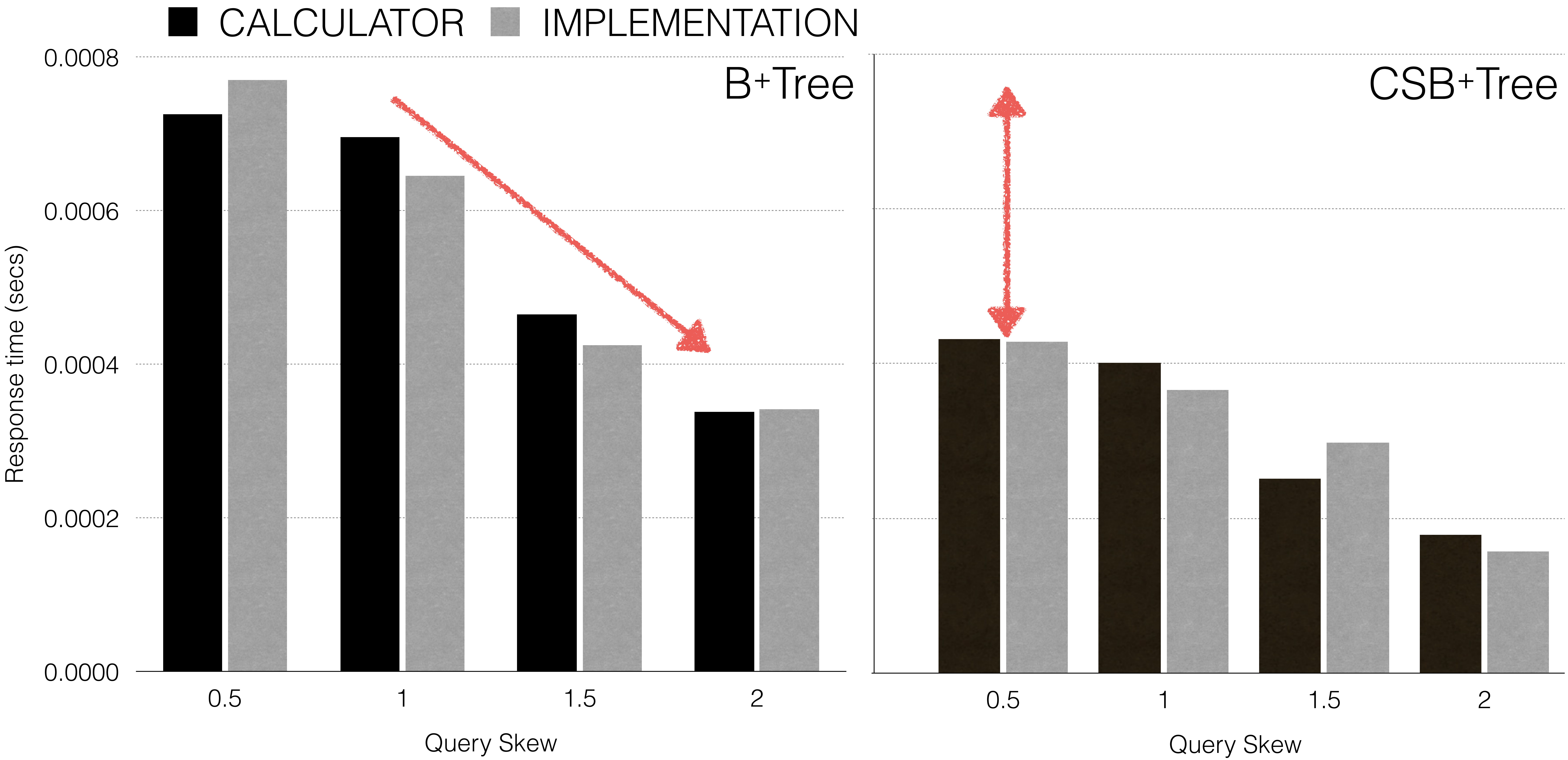
(same workload, hardware, data)



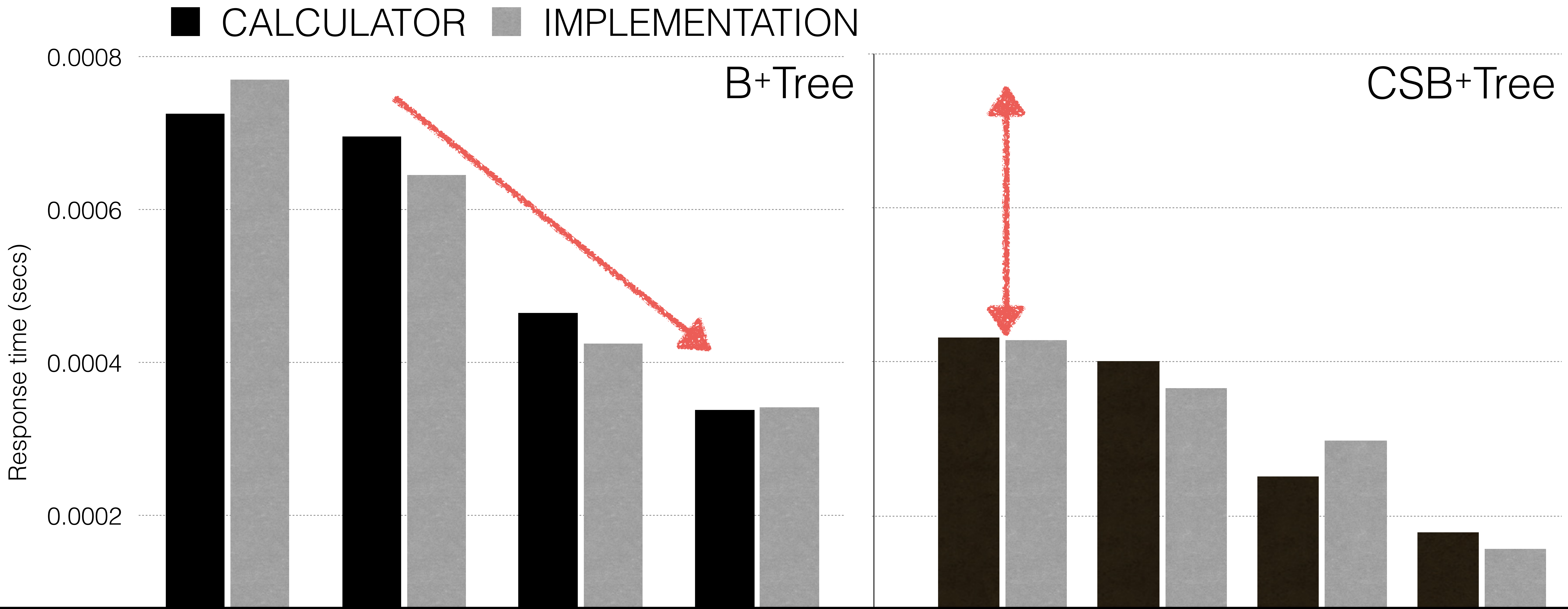
{10M (uniform) k-v pairs, 100 point queries (skewed)}



{10M (uniform) k-v pairs, 100 point queries (skewed)}



{10M (uniform) k-v pairs, 100 point queries (skewed)}



IF IT WORKS FOR >>1 KNOWN DESIGNS WE CAN TRUST IT FOR NEW ONES

DESIGN SPACE

COST ESTIMATION

SEARCH

DASlab
@ Harvard SEAS

Key-value Stores/Databases/ML Systems

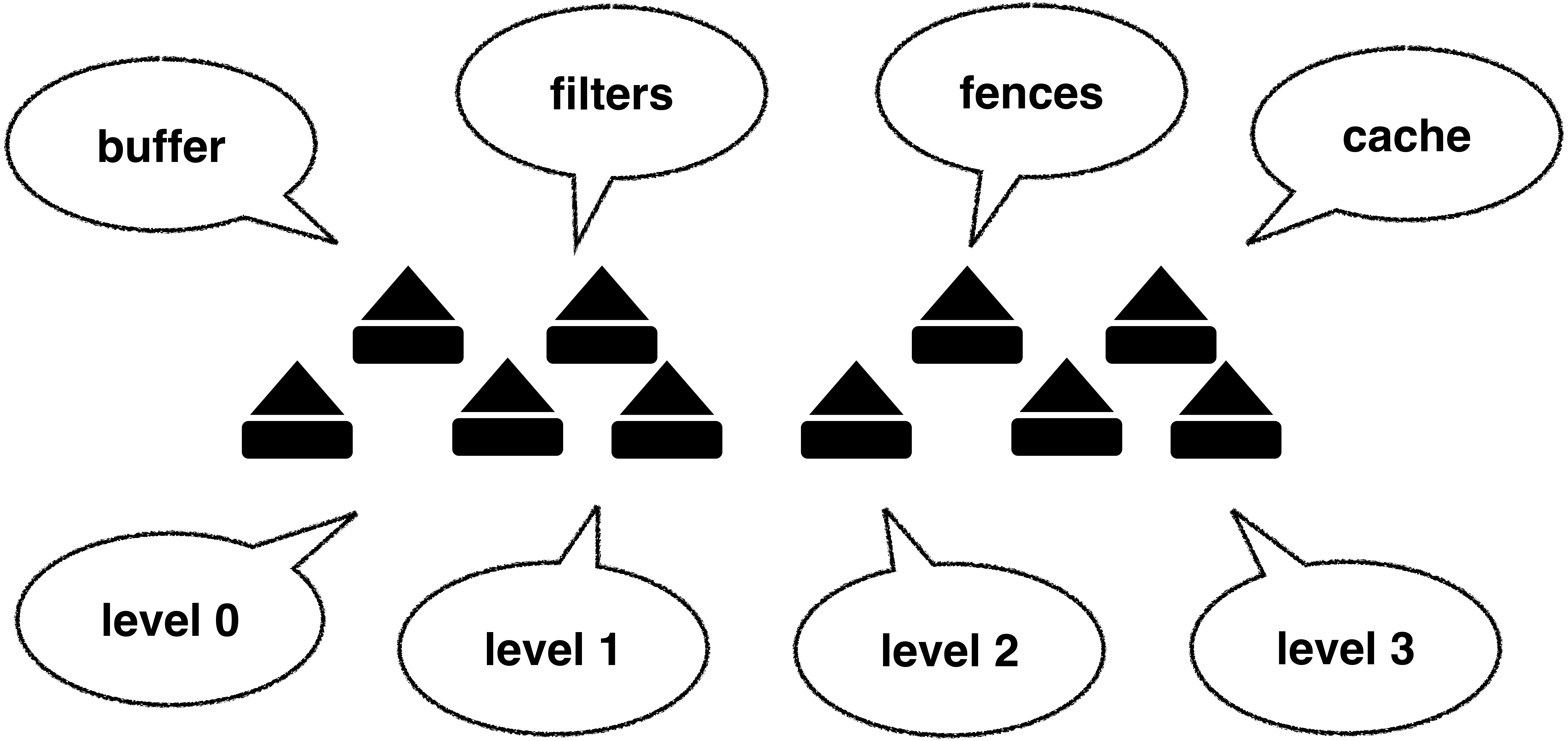
>10^100

NoSQL systems are the backbone of the BigData and AI era

LSM-tree
KV-stores

FACEBOOK, AMAZON, GOOGLE, TWITTER, LINKEDIN

MACHINE LEARNING, SQL, CRYPTO, SCIENCE

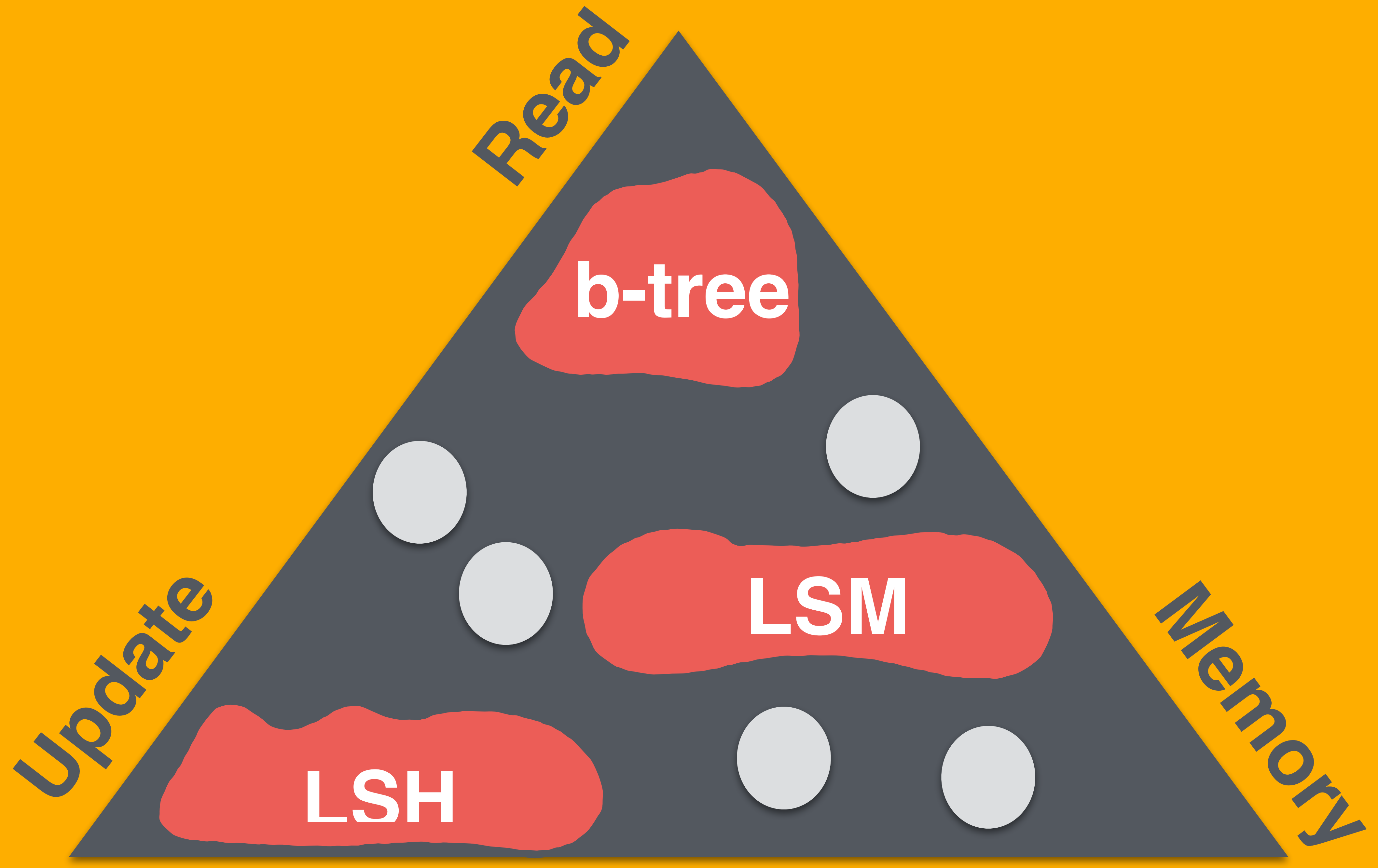


DASlab @ Harvard SEAS

NoSQL systems are the backbone of the BigData and AI era

LSM-tree
KV-stores

FACEBOOK, AMAZON, GOOGLE, TWITTER, LINKEDIN
MACHINE LEARNING, SQL, CRYPTO, SCIENCE



diverse data structures

interactions

hardware

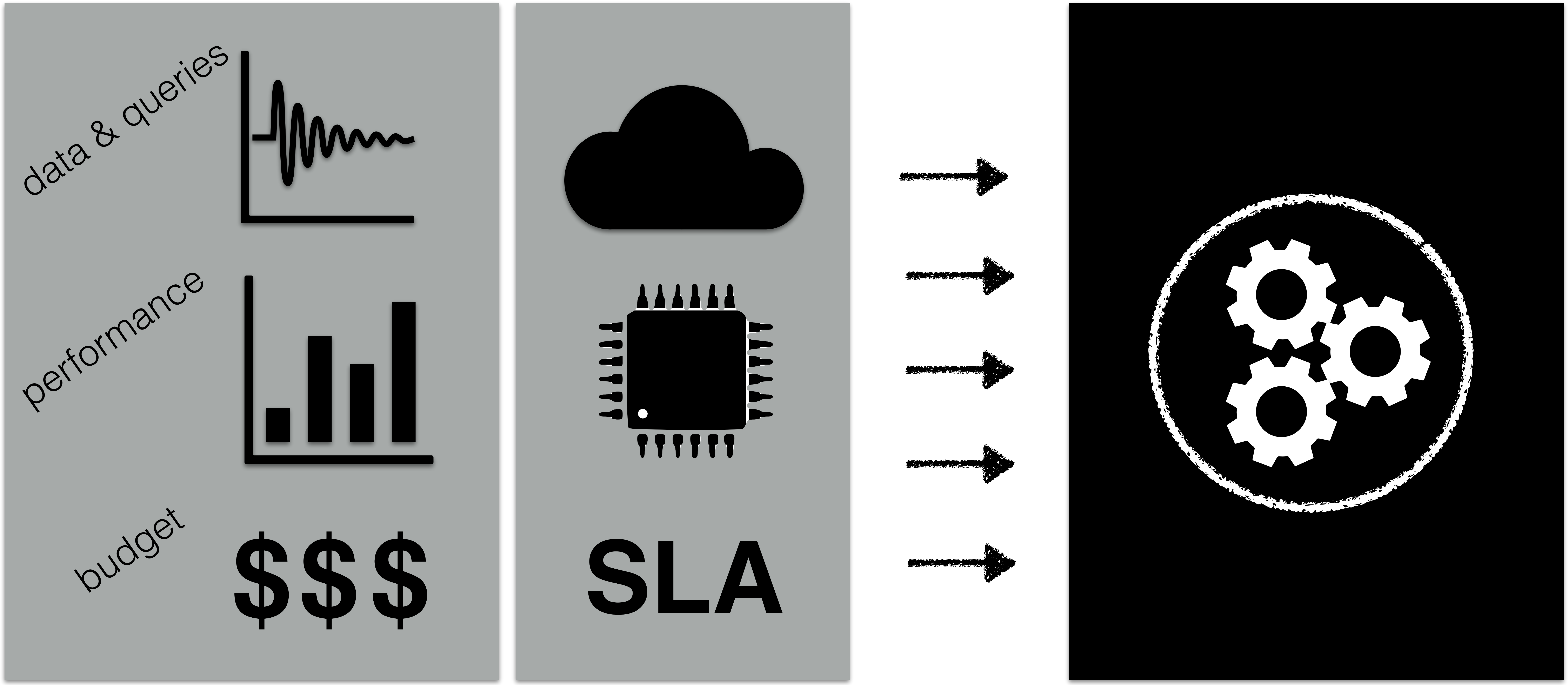
parallelism

robustness cloud cost SLAs

There exist three core variations of NoSQL KV-stores

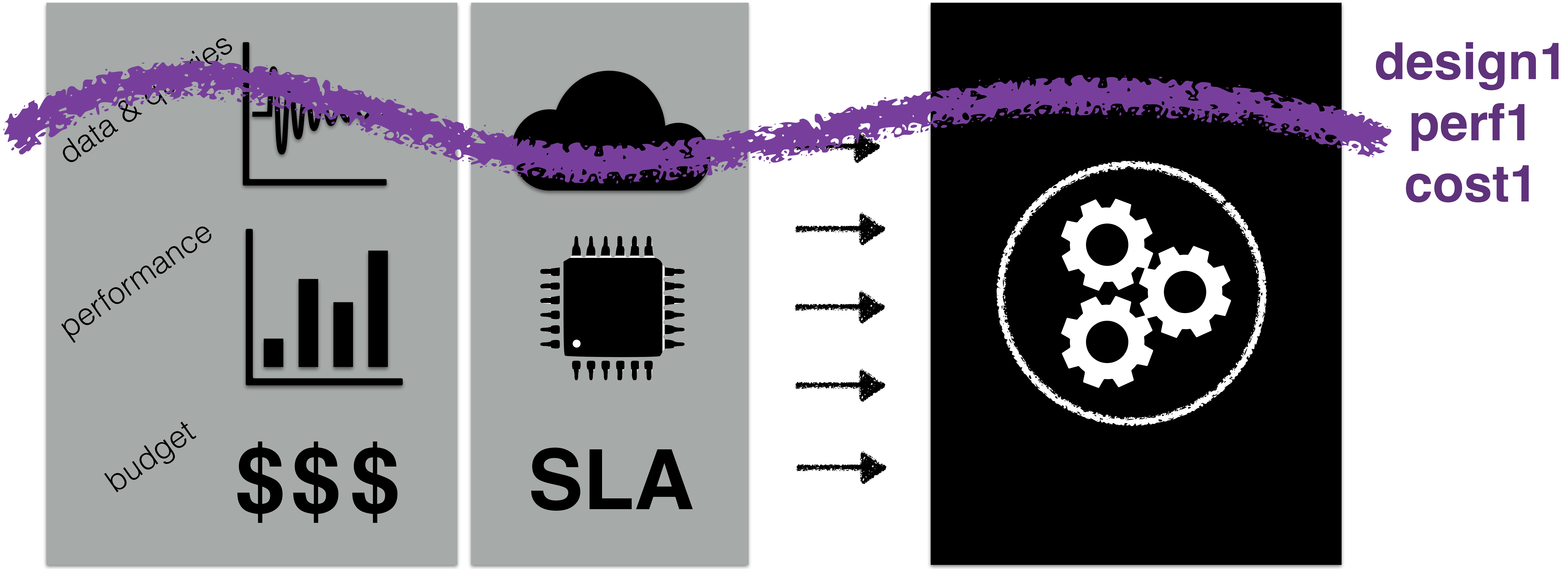
But there is a massive possible set of designs

DASlab
@ Harvard SEAS



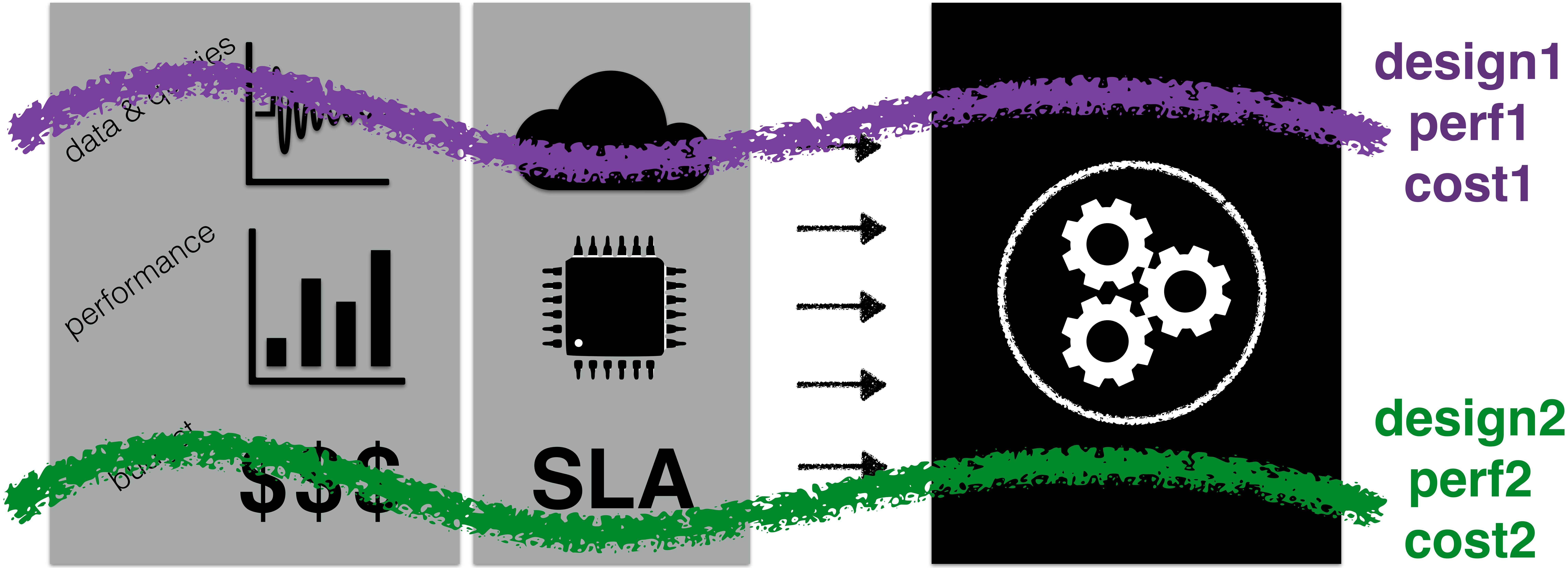
what-if reasoning

DASlab
@ Harvard SEAS



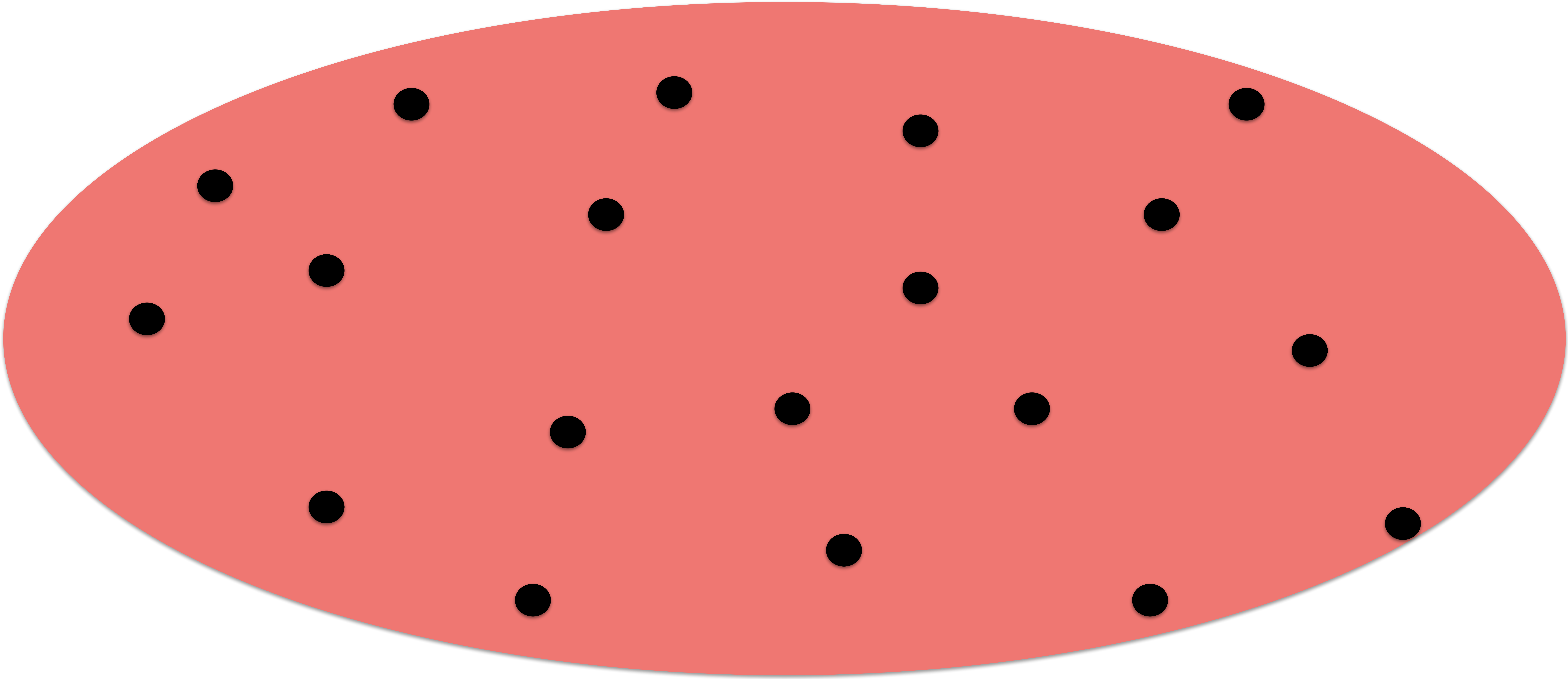
what-if reasoning

DASlab @ Harvard SEAS

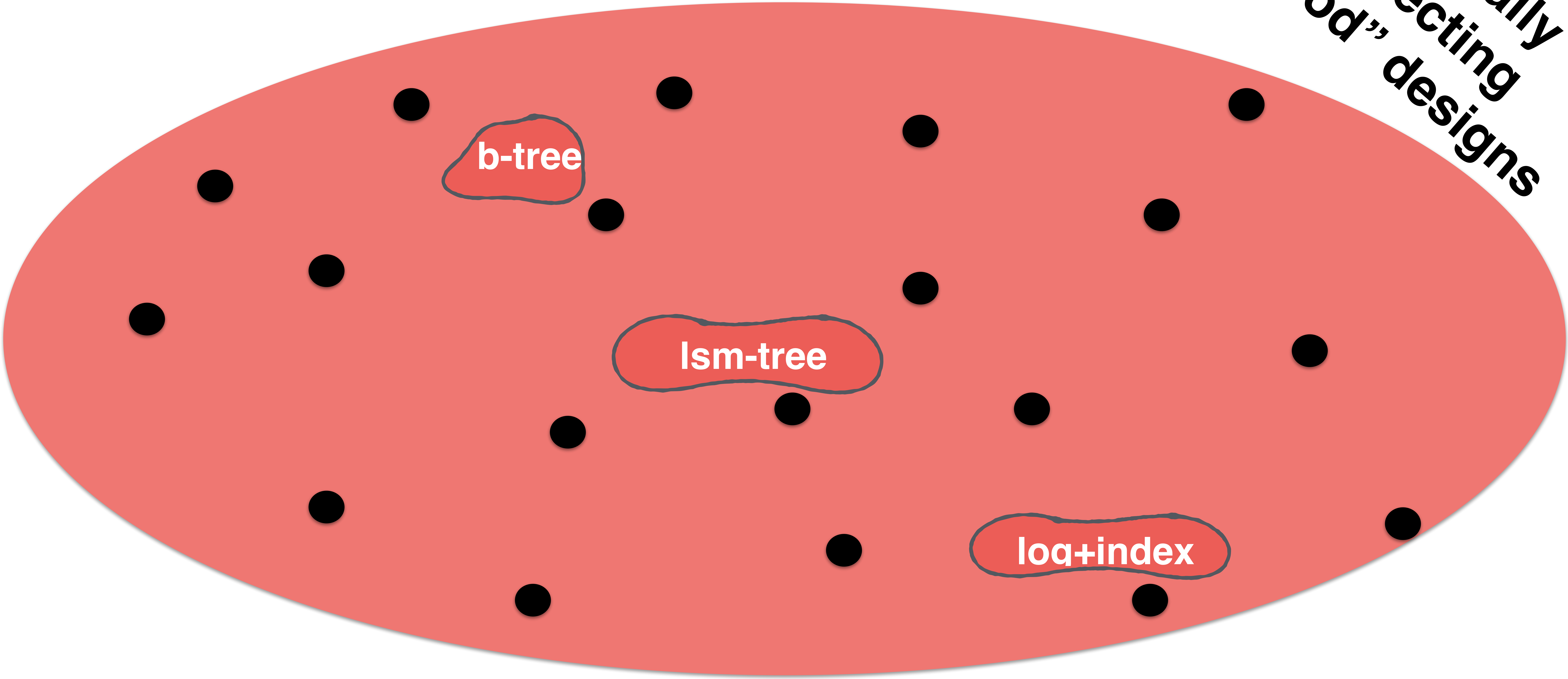


what-if reasoning

shrinking the massive design space

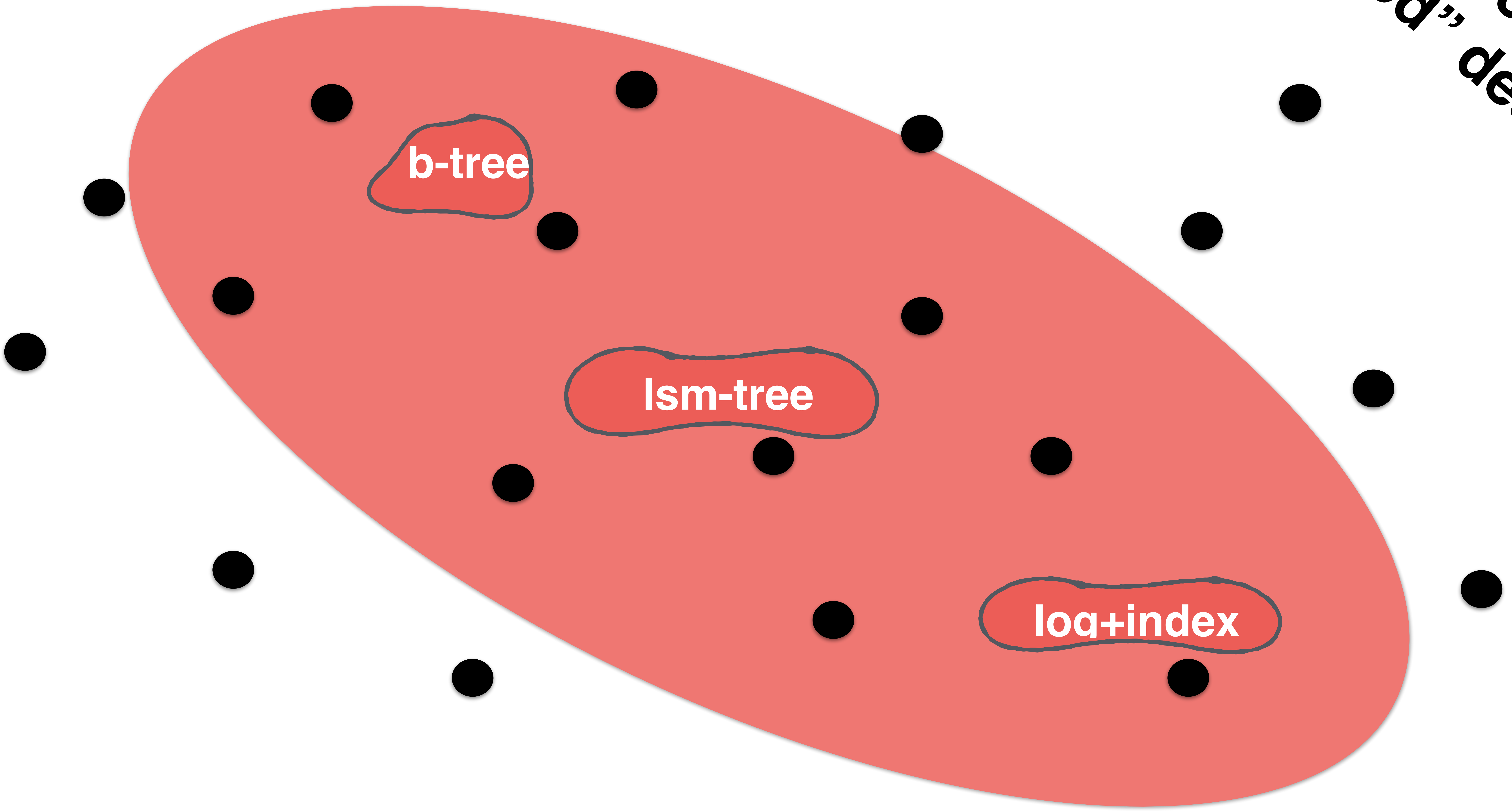


manually selecting "good" designs



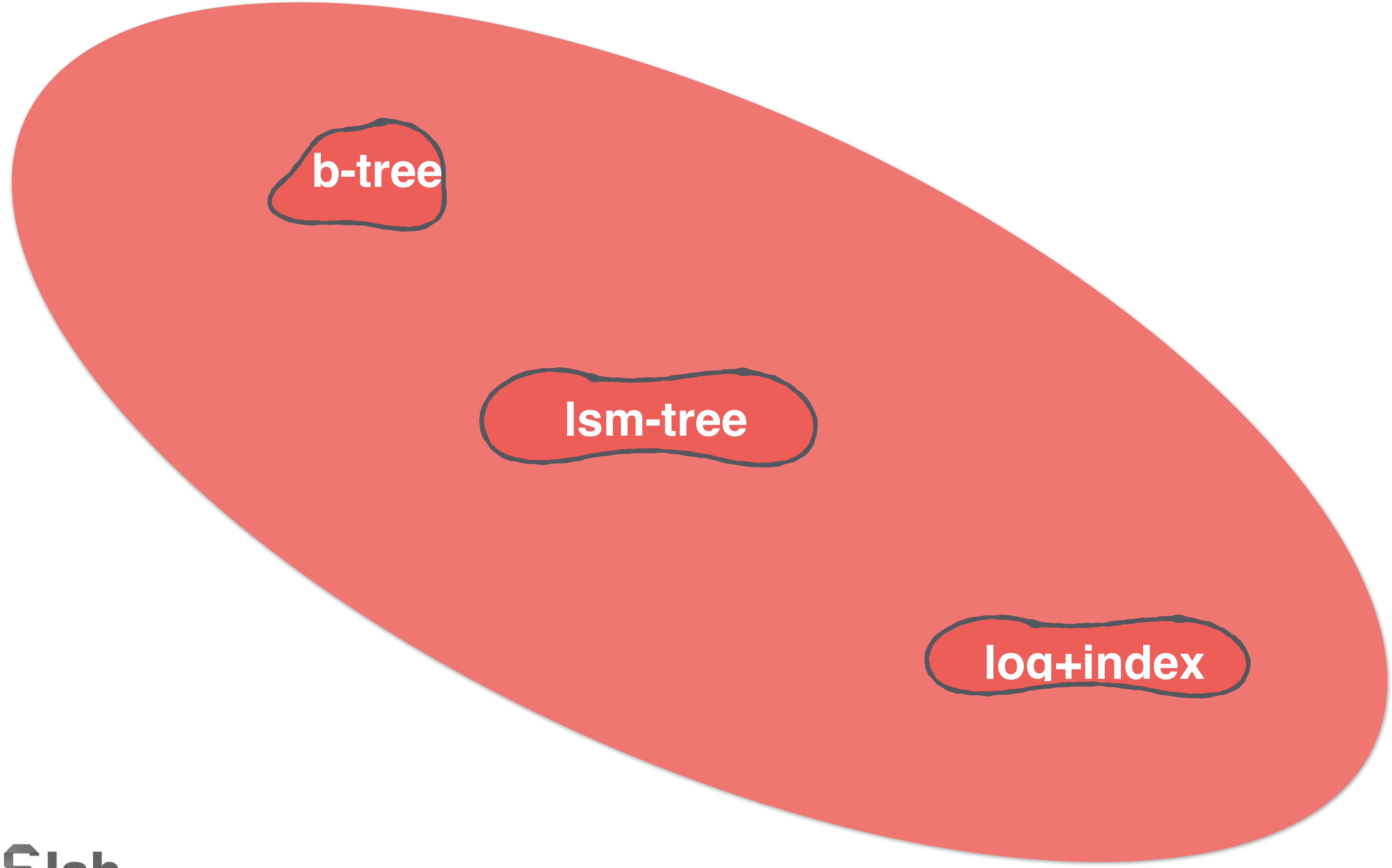
DASlab @ Harvard SEAS

manually selecting "good" designs



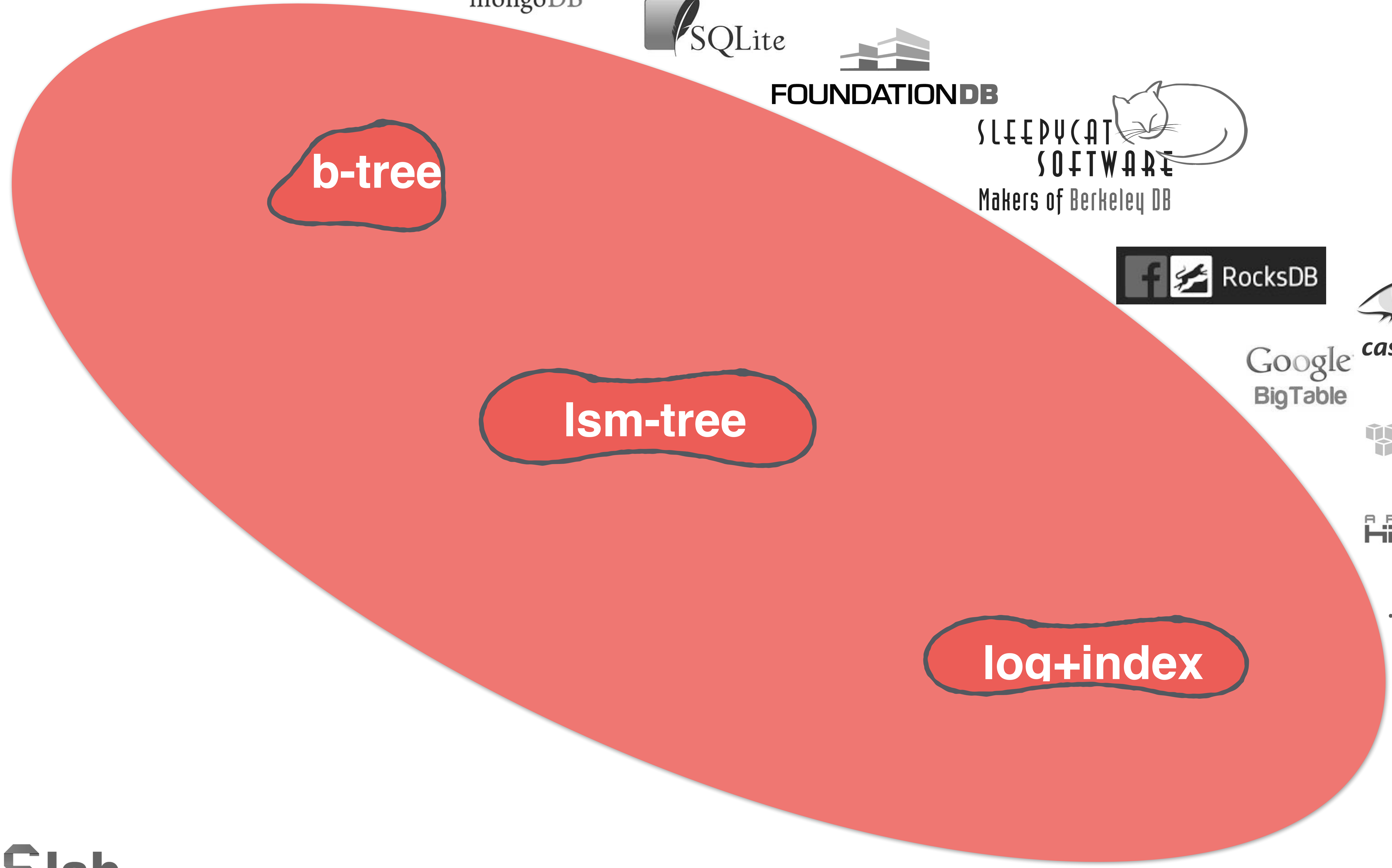
DASlab @ Harvard SEAS

manually selecting “good” designs



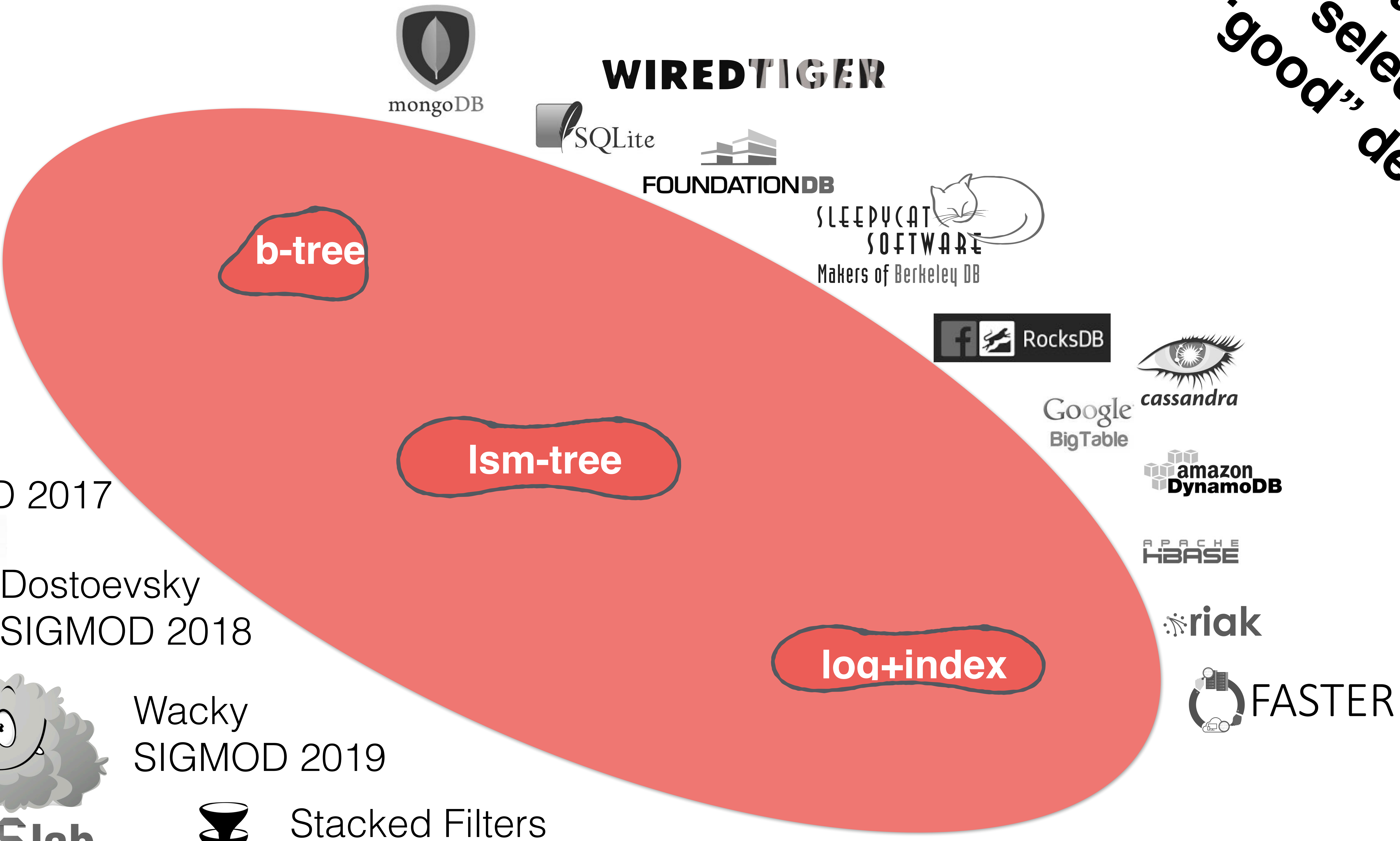
DASlab @ Harvard SEAS

manually selecting “good” designs



mongoDB
WIREDTIGER
SQLite
FOUNDATIONDB
SLEEPYCAT SOFTWARE Makers of Berkeley DB
RocksDB
cassandra
Google BigTable
amazon DynamoDB
APACHE HBASE
riak
FASTER

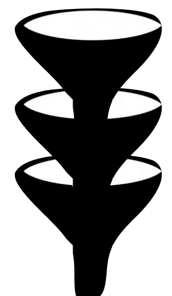
manually selecting “good” designs



Monkey
SIGMOD 2017

Dostoevsky
SIGMOD 2018

Wacky
SIGMOD 2019

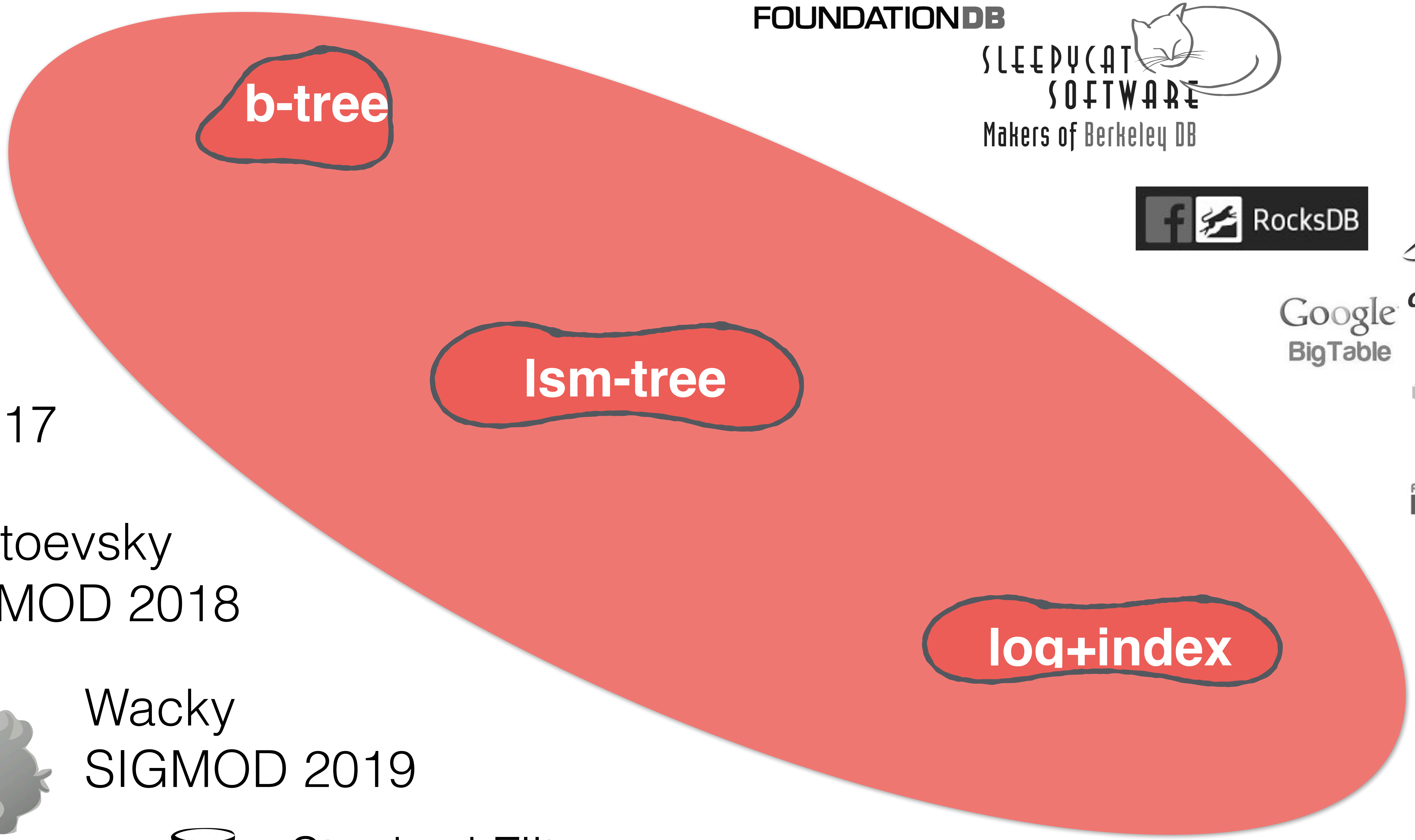


Stacked Filters
SIGMOD 2020

DASlab
@ Harvard SEAS

manually selecting “good” designs

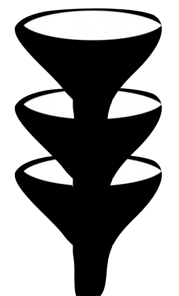
mongoDB
WIREDTIGER
SQLite
FOUNDATIONDB
SLEEPYCAT SOFTWARE
Makers of Berkeley DB
RocksDB
cassandra
Google BigTable
amazon DynamoDB
APACHE HBASE
riak
FASTER



Monkey
SIGMOD 2017

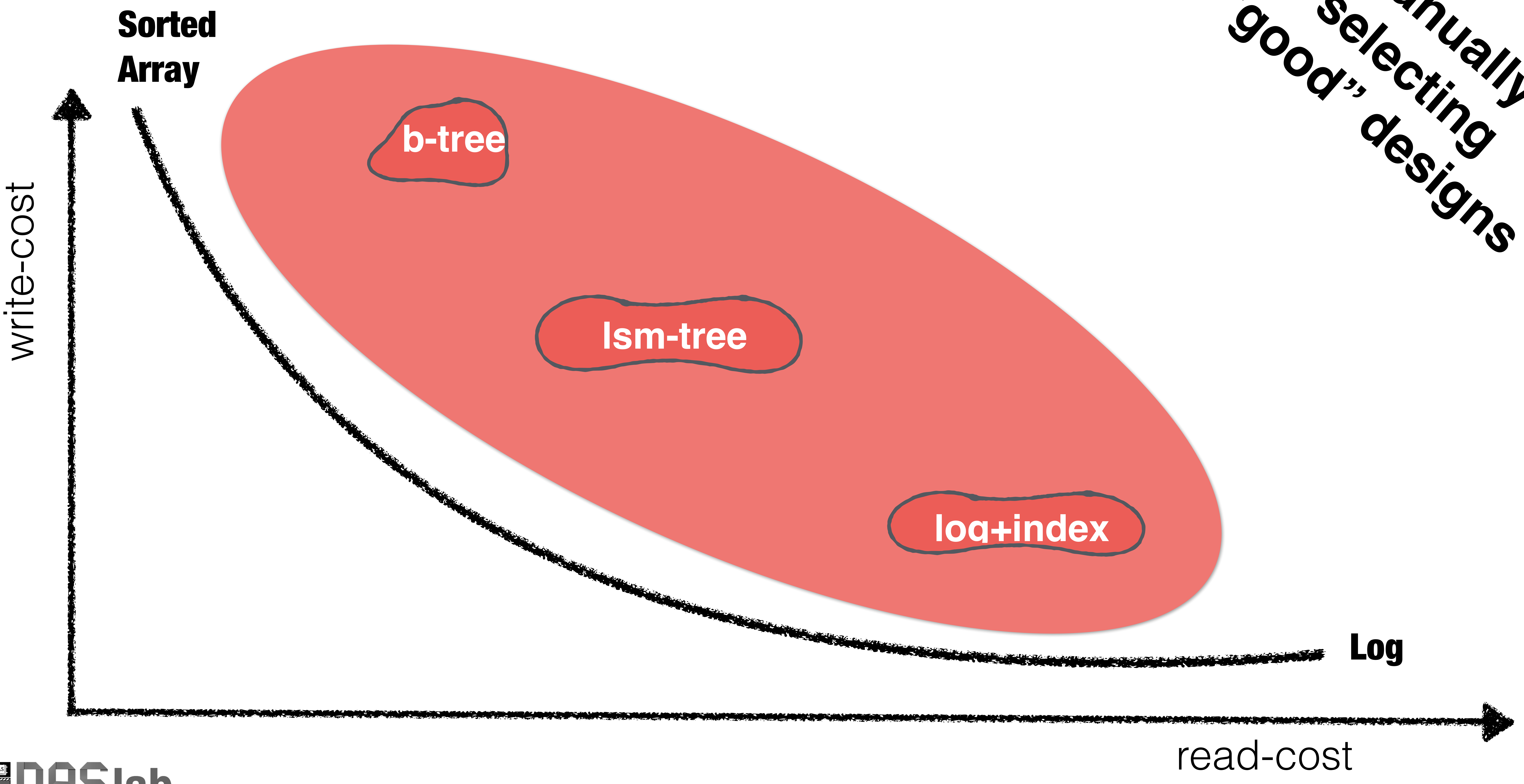
Dostoevsky
SIGMOD 2018

Wacky
SIGMOD 2019



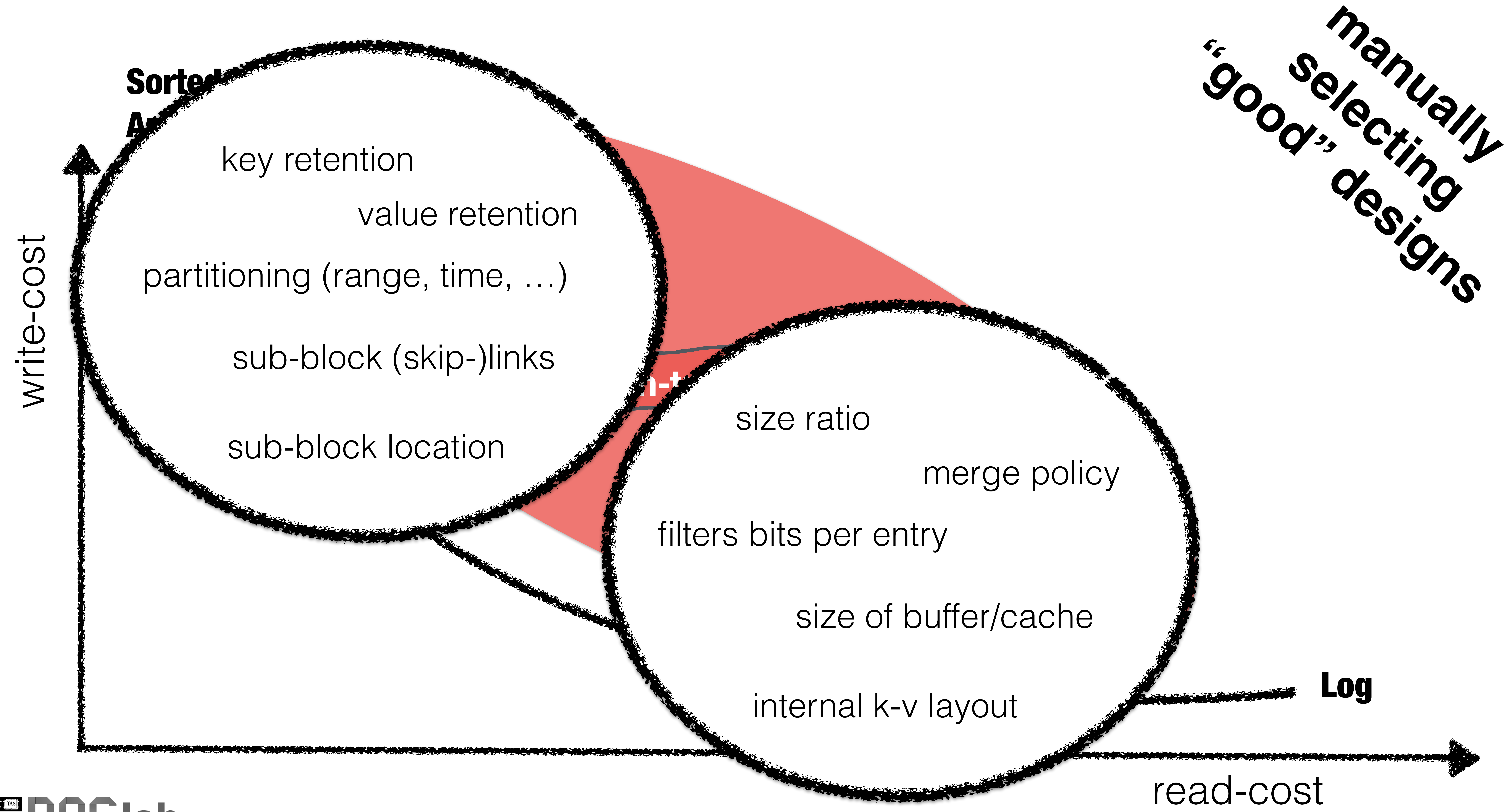
Stacked Filters
SIGMOD 2020

DASlab
@ Harvard SEAS

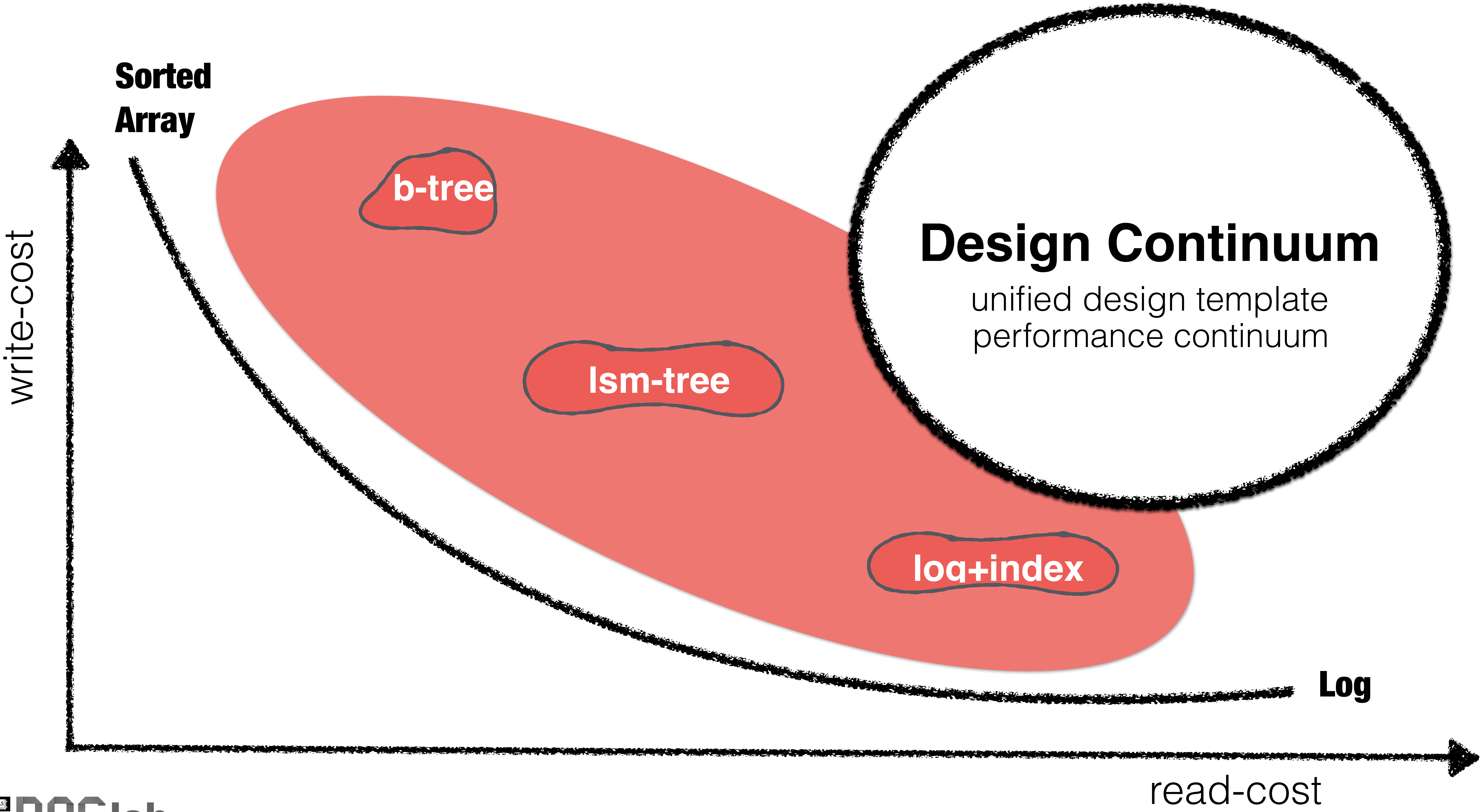


manually selecting "good" designs

DASlab @ Harvard SEAS


manually selecting "good" designs
Sorted Array
key retention
value retention
partitioning (range, time, ...)
sub-block (skip-)links
sub-block location
size ratio
merge policy
filters bits per entry
size of buffer/cache
internal k-v layout
Log
write-cost
read-cost


Sorted Array
write-cost
b-tree
lsm-tree
log+index
Log
read-cost
key retention
value retention
partitioning (range, time, …)
sub-block (skip-)links
sub-block location
DASlab
@ Harvard SEAS



Sorted Array
b-tree
Design Continuum
unified design template
performance continuum
write-cost
lsm-tree
log+index
Log
read-cost

Cosine
@PVLDB2022

Cloud-cost Optimized

Self Designing

Key-value Store

			Design Abstractions of Template	Type/Domain	Example templates for diverse data structures			
					LSM variants	B-Tree variants	LSH variants	A new design
LAYOUT PRIMITIVES	Design and hardware specification	*initialized by search through engine design space*	1. **Key size:** Denotes the size of keys in the workload.	unsigned int	auto-configured from the sample workload			
			2. **Value size:** Denotes the size of values in the workload. All values are accepted as variable-length strings.	string/slice *max size set to 1 GB*	auto-configured from the sample workload			
			3. **Size ratio (T):** The maximum number of entries in a block (e.g. growth factor in LSM trees or fanout of B-trees.	unsigned integer | function (func)	[2,.. 32]	[32, 64, 128, 256, ..]	[1000, 1001, …] (T is large)	2
			4. **Runs per hot level (K):** At what capacity hot levels are compacted. Rule: should be less than size ratio.	unsigned int	[1.. T]		[T-1]	7
			5. **Runs per cold level (Z):** At what capacity cold levels are compacted. Rule: should be less than size ratio.	unsigned int	[1.. T]	[1]		32
			6. **Logical block size (B):** Number of consecutive disk blocks.	unsigned int	[2048, 4096, …]			
			7. **Buffer capacity** (M_B)**:** Denotes the amount of memory allocated to in-memory buffer/memtables. Configurable w.r.t file size.	64-bit floating point | function (func)	[64 MB, 128 MB, …]	[1 MB, 2 MB, …]	[64 MB, 128 MB, …]	h/w dependent
			8. **Indexes**(M_{FP})**:** Amount of memory allocated to indexes (fence pointers/hashtables).	64-bit floating point | function (func)	memory to cover L	memory for first level	memory for hash table	h/w dependent
			9. **Bloom filter memory** (M_{BF})**:** Denotes the bits/entry assigned to Bloom filters.	64-bit float | func(FPR)	10 bits/key			func(FPR)
ALGORITHMIC ABSTRACTIONS	Data access	*derived with empirically verified rules*	10. **Bloom filter design:** Denotes the granularity of Bloom filters, e.g., one Bloom filter instance per block or per file or per run. The default is file.	block | file | run	file			file
			11. **Compaction/Restructuring algorithm:** Full does level-to-level compaction; partial is file-to-file; and hybrid uses both full and partial at separate levels.	partial | full | hybrid	full, partial	partial	partial	hybrid
			12. **Run strategy:** Denotes which run to be picked for compaction (only for partial/ hybrid compaction).	first | last_full | fullest	first, fullest, last_full		first	fullest
			13. **File picking strategy:** Denotes which file to be picked for compaction (for partial/ hybrid compaction). For LSM-trees we set default to dense_fp as it empirically works the best. B-trees pick the first file found to be full. LSH-table restructures at the granularity of runs.	oldest_merged | oldest_flushed | dense_fp | sparse_fp | choose_first	dense_fp	choose_first		dense_fp (hot), choose_first (cold)
	Parallelism		14. **Merge threshold:** If a level is more than x% full, a compaction is triggered.	64-bit floating point	[0.7..1]	0.5		0.75
			15. **Full compaction levels:** Denotes how many levels will have full compaction (only for hybrid compaction). The default is set to 2.	unsigned integer | function (func)	[1..L]			L-Y (from optimal config)
			16. **No. of CPUs:** Number of available cores to use in a VM.	unsigned int	Use all available cores			
			17. **No of threads:** Denotes how many threads are used to process the workload.	unsigned int	Use 1 thread per CPU core			

Storage engine template in Cosine and example initializations for diverse storage engine designs.

Cosine
@PVLDB2022

Cloud-cost Optimized

Self Designing

Key-value Store

				Design Abstractions of Template	**Type/Domain**	**Example templates for diverse data structures**			
						LSM variants	**B-Tree variants**	**LSH variants**	**A new design**
LAYOUT PRIMITIVES	Design and hardware specification	*initialized by search through engine design space*	1.	**Key size:** Denotes the size of keys in the workload.	unsigned int	auto-configured from the sample workload			
LAYOUT PRIMITIVES	Design and hardware specification	*initialized by search through engine design space*	2.	**Value size:** Denotes the size of values in the workload. All values are accepted as variable-length strings.	string/slice *max size set to 1 GB*	auto-configured from the sample workload			
LAYOUT PRIMITIVES	Design and hardware specification	*initialized by search through engine design space*	3.	**Size ratio (T):** The maximum number of entries in a block (e.g. growth factor in LSM trees or fanout of B-trees.	unsigned integer | function (func)	[2,.. 32]	[32, 64, 128, 256, ..]	[1000, 1001, …] (T is large)	2
LAYOUT PRIMITIVES	Design and hardware specification	*initialized by search through engine design space*	4.	**Runs per hot level (K):** At what capacity hot levels are compacted. Rule: should be less than size ratio.	unsigned int	[1.. T]		[T-1]	7
LAYOUT PRIMITIVES	Design and hardware specification	*initialized by search through engine design space*	5.	**Runs per cold level (Z):** At what capacity cold levels are compacted. Rule: should be less than size ratio.	unsigned int	[1.. T]	[1]		32
LAYOUT PRIMITIVES	Design and hardware specification	*initialized by search through engine design space*	6.	**Logical block size (B):** Number of consecutive disk blocks.	unsigned int	[2048, 4096, …]			
LAYOUT PRIMITIVES	Design and hardware specification	*initialized by search through engine design space*	7.	**Buffer capacity** (M_B)**:** Denotes the amount of memory allocated to in-memory buffer/memtables. Configurable w.r.t file size.	64-bit floating point | function (func)	[64 MB, 128 MB, …]	[1 MB, 2 MB, …]	[64 MB, 128 MB, …]	h/w dependent
LAYOUT PRIMITIVES	Design and hardware specification	*initialized by search through engine design space*	8.	**Indexes**(M_{FP})**:** Amount of memory allocated to indexes (fence pointers/hashtables).	64-bit floating point | function (func)	memory to cover L	memory for first level	memory for hash table	h/w dependent
LAYOUT PRIMITIVES	Design and hardware specification	*initialized by search through engine design space*	9.	**Bloom filter memory** (M_{BF})**:** Denotes the bits/entry assigned to Bloom filters.	64-bit float | func(FPR)	10 bits/key			func(FPR)
ALGORITHMIC ABSTRACTIONS	Data access	*derived with empirically verified rules*	10.	**Bloom filter design:** Denotes the granularity of Bloom filters, e.g., one Bloom filter instance per block or per file or per run. The default is file.	block | file | run	file			file
ALGORITHMIC ABSTRACTIONS	Data access	*derived with empirically verified rules*	11.	**Compaction/Restructuring algorithm:** Full does level-to-level compaction; partial is file-to-file; and hybrid uses both full and partial at separate levels.	partial | full | hybrid	full, partial	partial	partial	hybrid
ALGORITHMIC ABSTRACTIONS	Data access	*derived with empirically verified rules*	12.	**Run strategy:** Denotes which run to be picked for compaction (only for partial/ hybrid compaction).	first | last_full | fullest	first, fullest, last_full		first	fullest
ALGORITHMIC ABSTRACTIONS	Data access	*derived with empirically verified rules*	13.	**File picking strategy:** Denotes which file to be picked for compaction (for partial/ hybrid compaction). For LSM-trees we set default to dense_fp as it empirically works the best. B-trees pick the first file found to be full. LSH-table restructures at the granularity of runs.	oldest_merged | oldest_flushed | dense_fp | sparse_fp | choose_first	dense_fp	choose_first		dense_fp (hot), choose_first (cold)
ALGORITHMIC ABSTRACTIONS	Parallelism	*derived with empirically verified rules*	14.	**Merge threshold:** If a level is more than x% full, a compaction is triggered.	64-bit floating point	[0.7..1]	0.5		0.75
ALGORITHMIC ABSTRACTIONS	Parallelism	*derived with empirically verified rules*	15.	**Full compaction levels:** Denotes how many levels will have full compaction (only for hybrid compaction). The default is set to 2.	unsigned integer | function (func)	[1..L]			L-Y (from optimal config)
ALGORITHMIC ABSTRACTIONS	Parallelism	*derived with empirically verified rules*	16.	**No. of CPUs:** Number of available cores to use in a VM.	unsigned int	Use all available cores			
ALGORITHMIC ABSTRACTIONS	Parallelism	*derived with empirically verified rules*	17.	**No of threads:** Denotes how many threads are used to process the workload.	unsigned int	Use 1 thread per CPU core			

Storage engine template in Cosine and example initializations for diverse storage engine designs.

Cosine
@PVLDB2022

Cloud-cost Optimized

Self Designing

Key-value Store

LAYOUT PRIMITIVES — Design and hardware specification — *initialized by search through engine design space* (rows 1–9)

ALGORITHMIC ABSTRACTIONS — Data access / Parallelism — *derived with empirically verified rules* (rows 10–17)

	Design Abstractions of Template	Type/Domain	Example templates for diverse data structures: LSM variants	B-Tree variants	LSH variants	A new design
1.	**Key size:** Denotes the size of keys in the workload.	unsigned int	auto-configured from the sample workload			
2.	**Value size:** Denotes the size of values in the workload. All values are accepted as variable-length strings.	string/slice *max size set to 1 GB*	auto-configured from the sample workload			
3.	**Size ratio (T):** The maximum number of entries in a block (e.g. growth factor in LSM trees or fanout of B-trees.	unsigned integer | function (func)	[2,.. 32]	[32, 64, 128, 256, ..]	[1000, 1001, …] (T is large)	2
4.	**Runs per hot level (K):** At what capacity hot levels are compacted. Rule: should be less than size ratio.	unsigned int	[1.. T]		[T-1]	7
5.	**Runs per cold level (Z):** At what capacity cold levels are compacted. Rule: should be less than size ratio.	unsigned int	[1.. T]	[1]		32
6.	**Logical block size (B):** Number of consecutive disk blocks.	unsigned int	[2048, 4096, …]			
7.	**Buffer capacity** (M_B)**:** Denotes the amount of memory allocated to in-memory buffer/memtables. Configurable w.r.t file size.	64-bit floating point | function (func)	[64 MB, 128 MB, …]	[1 MB, 2 MB, …]	[64 MB, 128 MB, …]	h/w dependent
8.	**Indexes**(M_{FP})**:** Amount of memory allocated to indexes (fence pointers/hashtables).	64-bit floating point | function (func)	memory to cover L	memory for first level	memory for hash table	h/w dependent
9.	**Bloom filter memory** (M_{BF})**:** Denotes the bits/entry assigned to Bloom filters.	64-bit float | func(FPR)	10 bits/key			func(FPR)
10.	**Bloom filter design:** Denotes the granularity of Bloom filters, e.g., one Bloom filter instance per block or per file or per run. The default is file.	block | file | run	file			file
11.	**Compaction/Restructuring algorithm:** Full does level-to-level compaction; partial is file-to-file; and hybrid uses both full and partial at separate levels.	partial | full | hybrid	full, partial	partial	partial	hybrid
12.	**Run strategy:** Denotes which run to be picked for compaction (only for partial/ hybrid compaction).	first | last_full | fullest	first, fullest, last_full		first	fullest
13.	**File picking strategy:** Denotes which file to be picked for compaction (for partial/ hybrid compaction). For LSM-trees we set default to dense_fp as it empirically works the best. B-trees pick the first file found to be full. LSH-table restructures at the granularity of runs.	oldest_merged | oldest_flushed | dense_fp | sparse_fp | choose_first	dense_fp	choose_first		dense_fp (hot), choose_first (cold)
14.	**Merge threshold:** If a level is more than x% full, a compaction is triggered.	64-bit floating point	[0.7..1]	0.5		0.75
15.	**Full compaction levels:** Denotes how many levels will have full compaction (only for hybrid compaction). The default is set to 2.	unsigned integer | function (func)	[1..L]			L-Y (from optimal config)
16.	**No. of CPUs:** Number of available cores to use in a VM.	unsigned int	Use all available cores			
17.	**No of threads:** Denotes how many threads are used to process the workload.	unsigned int	Use 1 thread per CPU core			

Storage engine template in Cosine and example initializations for diverse storage engine designs.

Cosine
@PVLDB2022

Cloud-cost Optimized

Self Designing

Key-value Store

			#	Design Abstractions of Template	Type/Domain	Example templates for diverse data structures: LSM variants	B-Tree variants	LSH variants	A new design
LAYOUT PRIMITIVES	Design and hardware specification	*initialized by search through engine design space*	1.	**Key size:** Denotes the size of keys in the workload.	unsigned int	auto-configured from the sample workload			
			2.	**Value size:** Denotes the size of values in the workload. All values are accepted as variable-length strings.	string/slice *max size set to 1 GB*	auto-configured from the sample workload			
			3.	**Size ratio (T):** The maximum number of entries in a block (e.g. growth factor in LSM trees or fanout of B-trees.	unsigned integer | function (func)	[2,.. 32]	[32, 64, 128, 256, ..]	[1000, 1001, …] (T is large)	2
			4.	**Runs per hot level (K):** At what capacity hot levels are compacted. Rule: should be less than size ratio.	unsigned int	[1.. T]		[T-1]	7
			5.	**Runs per cold level (Z):** At what capacity cold levels are compacted. Rule: should be less than size ratio.	unsigned int	[1.. T]	[1]		32
			6.	**Logical block size (B):** Number of consecutive disk blocks.	unsigned int	[2048, 4096, …]			
			7.	**Buffer capacity** (M_B)**:** Denotes the amount of memory allocated to in-memory buffer/memtables. Configurable w.r.t file size.	64-bit floating point | function (func)	[64 MB, 128 MB, …]	[1 MB, 2 MB, …]	[64 MB, 128 MB, …]	h/w dependent
			8.	**Indexes** (M_{FP})**:** Amount of memory allocated to indexes (fence pointers/hashtables).	64-bit floating point | function (func)	memory to cover L	memory for first level	memory for hash table	h/w dependent
			9.	**Bloom filter memory** (M_{BF})**:** Denotes the bits/entry assigned to Bloom filters.	64-bit float | func(FPR)	10 bits/key			func(FPR)
ALGORITHMIC ABSTRACTIONS	Data access	*derived with empirically verified rules*	10.	**Bloom filter design:** Denotes the granularity of Bloom filters, e.g., one Bloom filter instance per block or per file or per run. The default is file.	block | file | run	file			file
			11.	**Compaction/Restructuring algorithm:** Full does level-to-level compaction; partial is file-to-file; and hybrid uses both full and partial at separate levels.	partial | full | hybrid	full, partial	partial	partial	hybrid
			12.	**Run strategy:** Denotes which run to be picked for compaction (only for partial/ hybrid compaction).	first | last_full | fullest	first, fullest, last_full		first	fullest
			13.	**File picking strategy:** Denotes which file to be picked for compaction (for partial/ hybrid compaction). For LSM-trees we set default to dense_fp as it empirically works the best. B-trees pick the first file found to be full. LSH-table restructures at the granularity of runs.	oldest_merged | oldest_flushed | dense_fp | sparse_fp | choose_first	dense_fp	choose_first		dense_fp (hot), choose_first (cold)
			14.	**Merge threshold:** If a level is more than x% full, a compaction is triggered.	64-bit floating point	[0.7..1]	0.5		0.75
			15.	**Full compaction levels:** Denotes how many levels will have full compaction (only for hybrid compaction). The default is set to 2.	unsigned integer | function (func)	[1..L]			L-Y (from optimal config)
	Parallelism		16.	**No. of CPUs:** Number of available cores to use in a VM.	unsigned int	Use all available cores			
			17.	**No of threads:** Denotes how many threads are used to process the workload.	unsigned int	Use 1 thread per CPU core			

Storage engine template in Cosine and example initializations for diverse storage engine designs.

Cosine
@PVLDB2022

Cloud-cost Optimized

Self Designing

Key-value Store

			#	Design Abstractions of Template	Type/Domain	Example templates for diverse data structures: LSM variants	B-Tree variants	LSH variants	A new design
LAYOUT PRIMITIVES	Design and hardware specification	*initialized by search through engine design space*	1.	**Key size:** Denotes the size of keys in the workload.	unsigned int	auto-configured from the sample workload			
			2.	**Value size:** Denotes the size of values in the workload. All values are accepted as variable-length strings.	string/slice *max size set to 1 GB*	auto-configured from the sample workload			
			3.	**Size ratio (T):** The maximum number of entries in a block (e.g. growth factor in LSM trees or fanout of B-trees.	unsigned integer | function (func)	[2,.. 32]	[32, 64, 128, 256, ..]	[1000, 1001, …] (T is large)	2
			4.	**Runs per hot level (K):** At what capacity hot levels are compacted. Rule: should be less than size ratio.	unsigned int	[1.. T]		[T-1]	7
			5.	**Runs per cold level (Z):** At what capacity cold levels are compacted. Rule: should be less than size ratio.	unsigned int	[1.. T]	[1]		32
			6.	**Logical block size (B):** Number of consecutive disk blocks.	unsigned int	[2048, 4096, …]			
			7.	**Buffer capacity** (M_B)**:** Denotes the amount of memory allocated to in-memory buffer/memtables. Configurable w.r.t file size.	64-bit floating point | function (func)	[64 MB, 128 MB, …]	[1 MB, 2 MB, …]	[64 MB, 128 MB, …]	h/w dependent
			8.	**Indexes**(M_{FP})**:** Amount of memory allocated to indexes (fence pointers/hashtables).	64-bit floating point | function (func)	memory to cover L	memory for first level	memory for hash table	h/w dependent
			9.	**Bloom filter memory** (M_{BF})**:** Denotes the bits/entry assigned to Bloom filters.	64-bit float | func(FPR)	10 bits/key			func(FPR)
ALGORITHMIC ABSTRACTIONS	Data access	*derived with empirically verified rules*	10.	**Bloom filter design:** Denotes the granularity of Bloom filters, e.g., one Bloom filter instance per block or per file or per run. The default is file.	block | file | run	file			file
			11.	**Compaction/Restructuring algorithm:** Full does level-to-level compaction; partial is file-to-file; and hybrid uses both full and partial at separate levels.	partial | full | hybrid	full, partial	partial	partial	hybrid
			12.	**Run strategy:** Denotes which run to be picked for compaction (only for partial/ hybrid compaction).	first | last_full | fullest	first, fullest, last_full		first	fullest
			13.	**File picking strategy:** Denotes which file to be picked for compaction (for partial/ hybrid compaction). For LSM-trees we set default to dense_fp as it empirically works the best. B-trees pick the first file found to be full. LSH-table restructures at the granularity of runs.	oldest_merged | oldest_flushed | dense_fp | sparse_fp | choose_first	dense_fp	choose_first		dense_fp (hot), choose_first (cold)
			14.	**Merge threshold:** If a level is more than x% full, a compaction is triggered.	64-bit floating point	[0.7..1]	0.5		0.75
	Parallelism		15.	**Full compaction levels:** Denotes how many levels will have full compaction (only for hybrid compaction). The default is set to 2.	unsigned integer | function (func)	[1..L]			L-Y (from optimal config)
			16.	**No. of CPUs:** Number of available cores to use in a VM.	unsigned int	Use all available cores			
			17.	**No of threads:** Denotes how many threads are used to process the workload.	unsigned int	Use 1 thread per CPU core			

Storage engine template in Cosine and example initializations for diverse storage engine designs.

from write to read optimized

Design Continuums @CDIR2019

Designs / Terms	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^{ϵ}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	$O(\frac{N \cdot E}{M_B} \cdot \frac{s}{B})$	$O(\frac{N \cdot E}{M_B} \cdot \frac{s}{B})$	$O(T \cdot \frac{s}{B})$	$O(\frac{s}{B})$	$O(\frac{s}{B})$	$O(\frac{s}{B})$	$O(\frac{s}{B})$	$O(\frac{s}{B})$	$O(\frac{s}{B})$	$O(\frac{s}{B})$

unified design

storage engine template

DASlab @ Harvard SEAS

unified design
storage engine template

analytical I/O model

unified closed form

learned CPU model

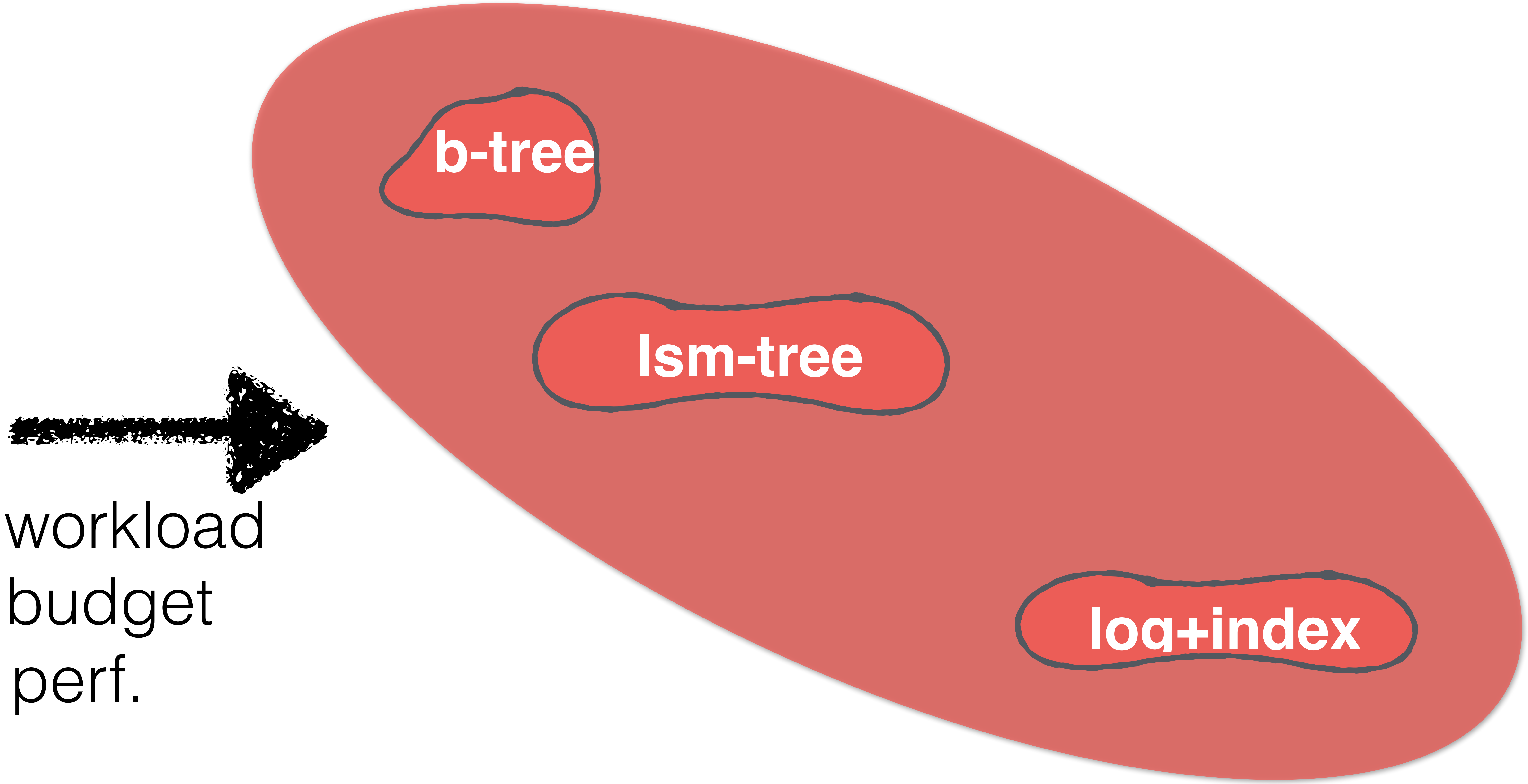
h/w, parallelism

cloud cost mapping & SLA

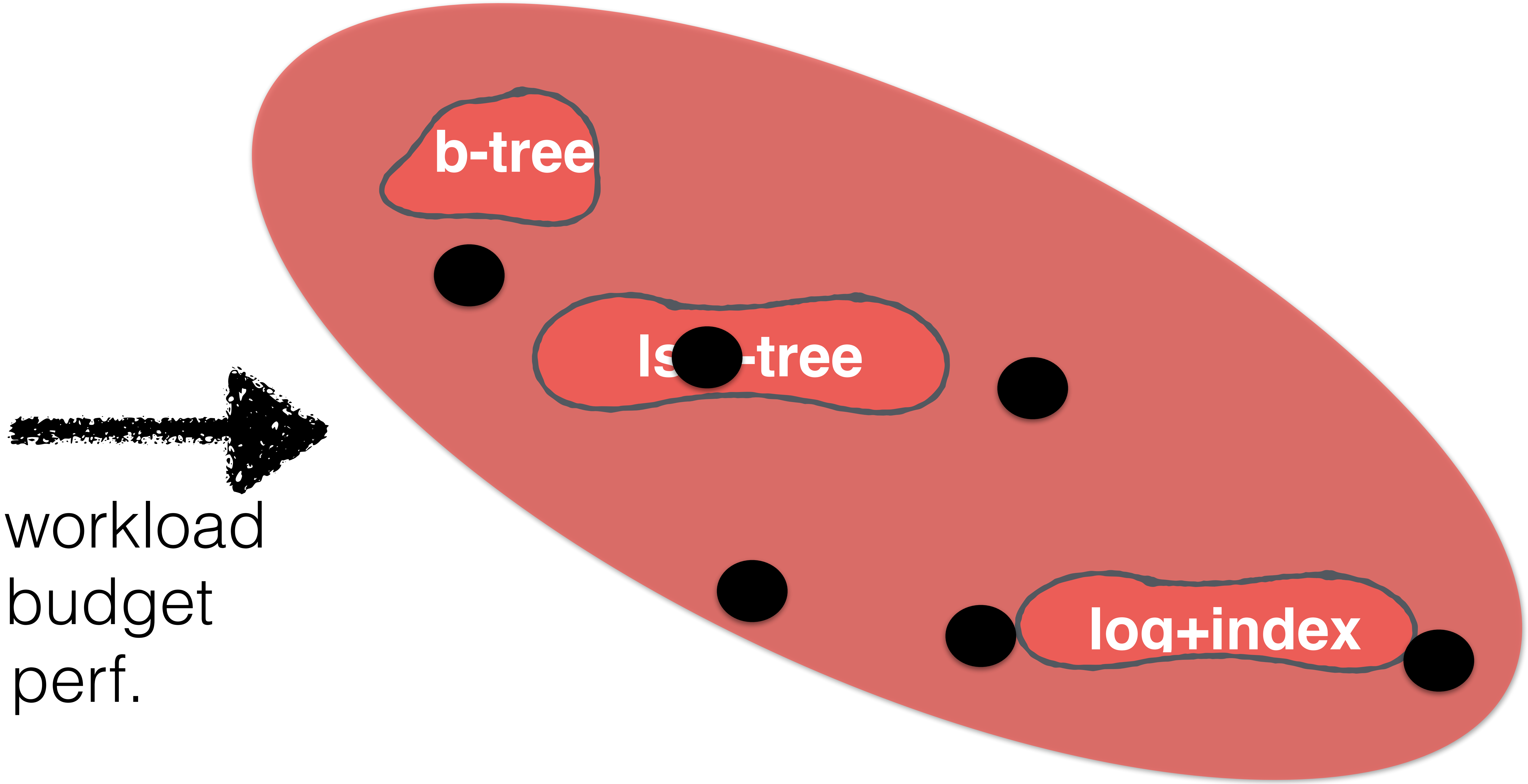
AWS, Azure, Google

workload
budget
perf.

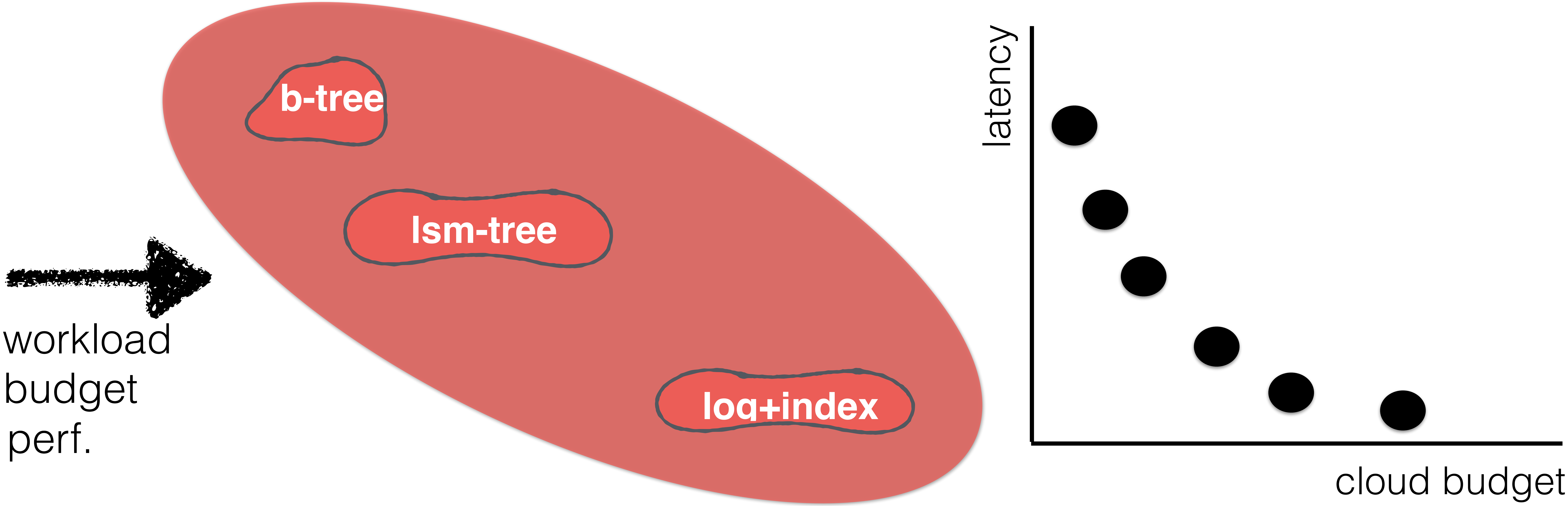
analytical I/O model	**learned CPU model**	**cloud cost mapping & SLA**
unified closed form	h/w, parallelism	AWS, Azure, Google



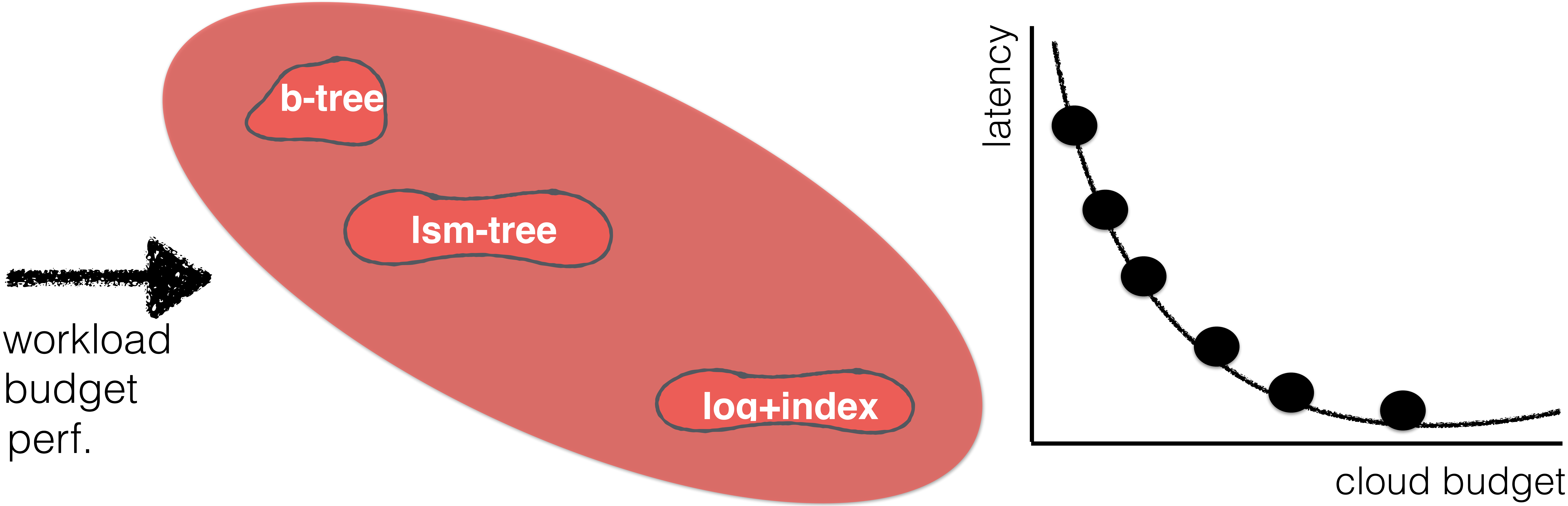
analytical I/O model	learned CPU model	cloud cost mapping & SLA	
unified closed form	h/w, parallelism	AWS, Azure, Google	



analytical I/O model	**learned CPU model**	**cloud cost mapping & SLA**
unified closed form	h/w, parallelism	AWS, Azure, Google



analytical I/O model	**learned CPU model**	**cloud cost mapping & SLA**
unified closed form	h/w, parallelism	AWS, Azure, Google



analytical I/O model

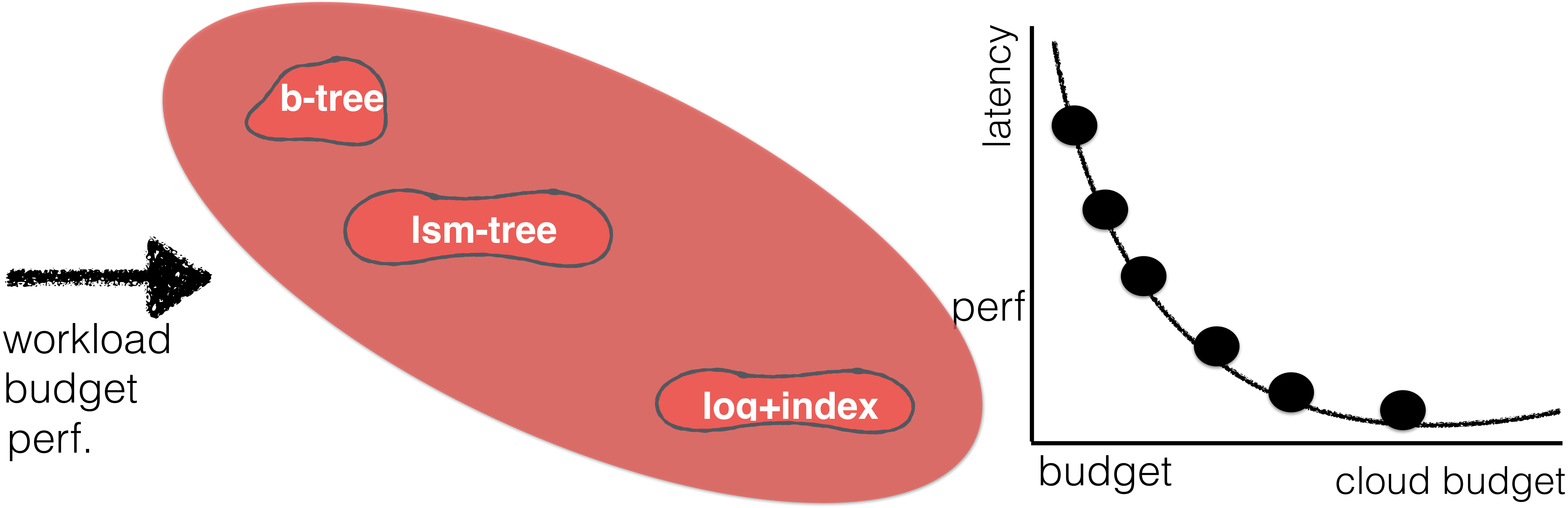
unified closed form

learned CPU model

h/w, parallelism

cloud cost mapping & SLA

AWS, Azure, Google



analytical I/O model

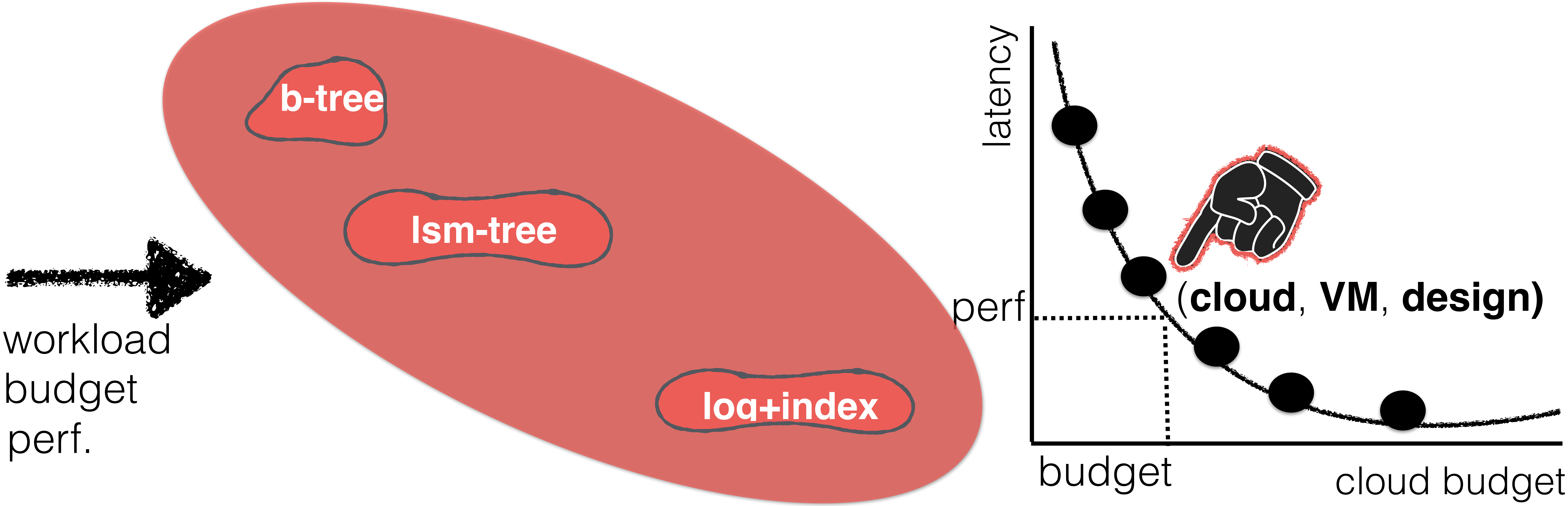
unified closed form

learned CPU model

h/w, parallelism

cloud cost mapping & SLA

AWS, Azure, Google



analytical I/O model

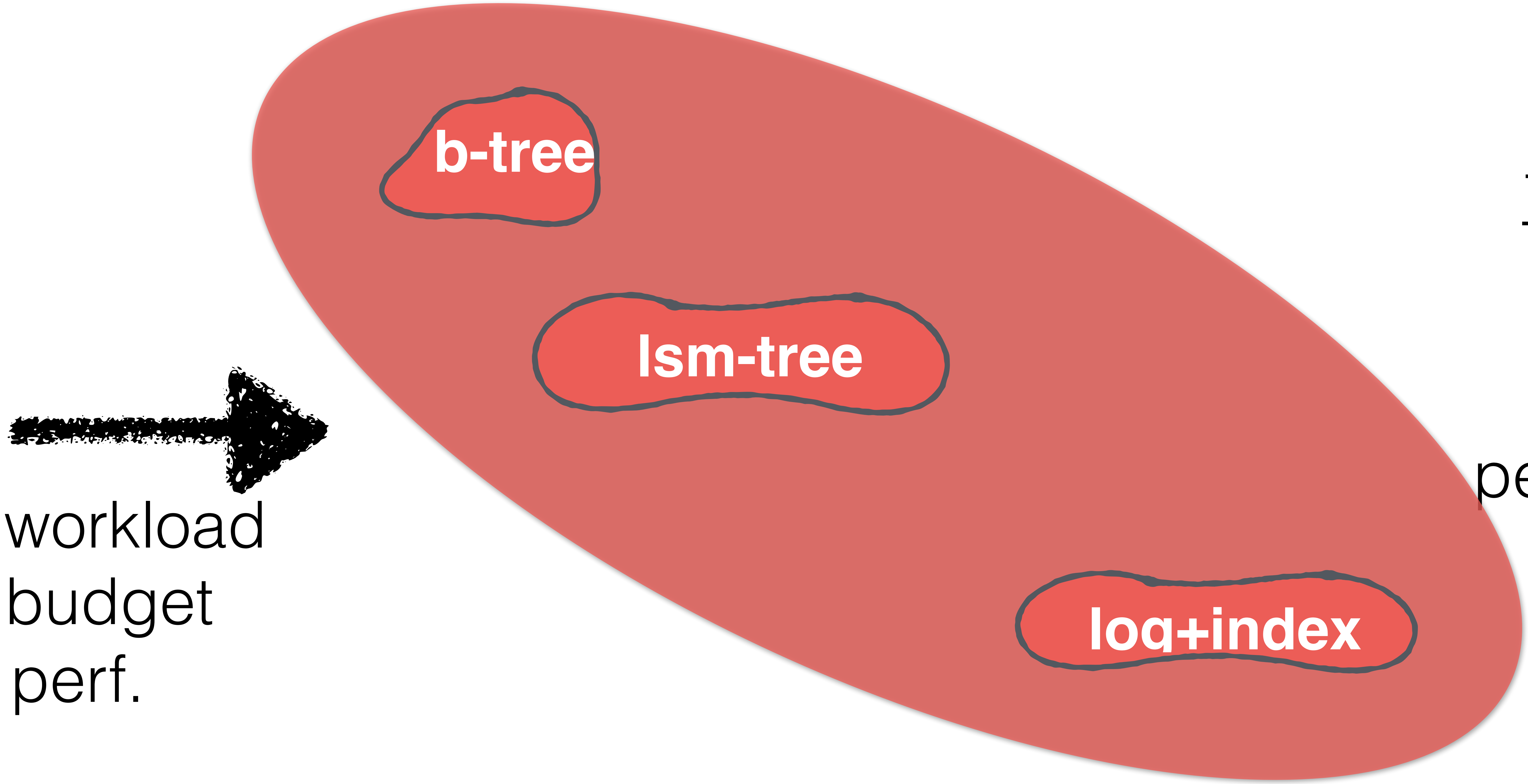
unified closed form

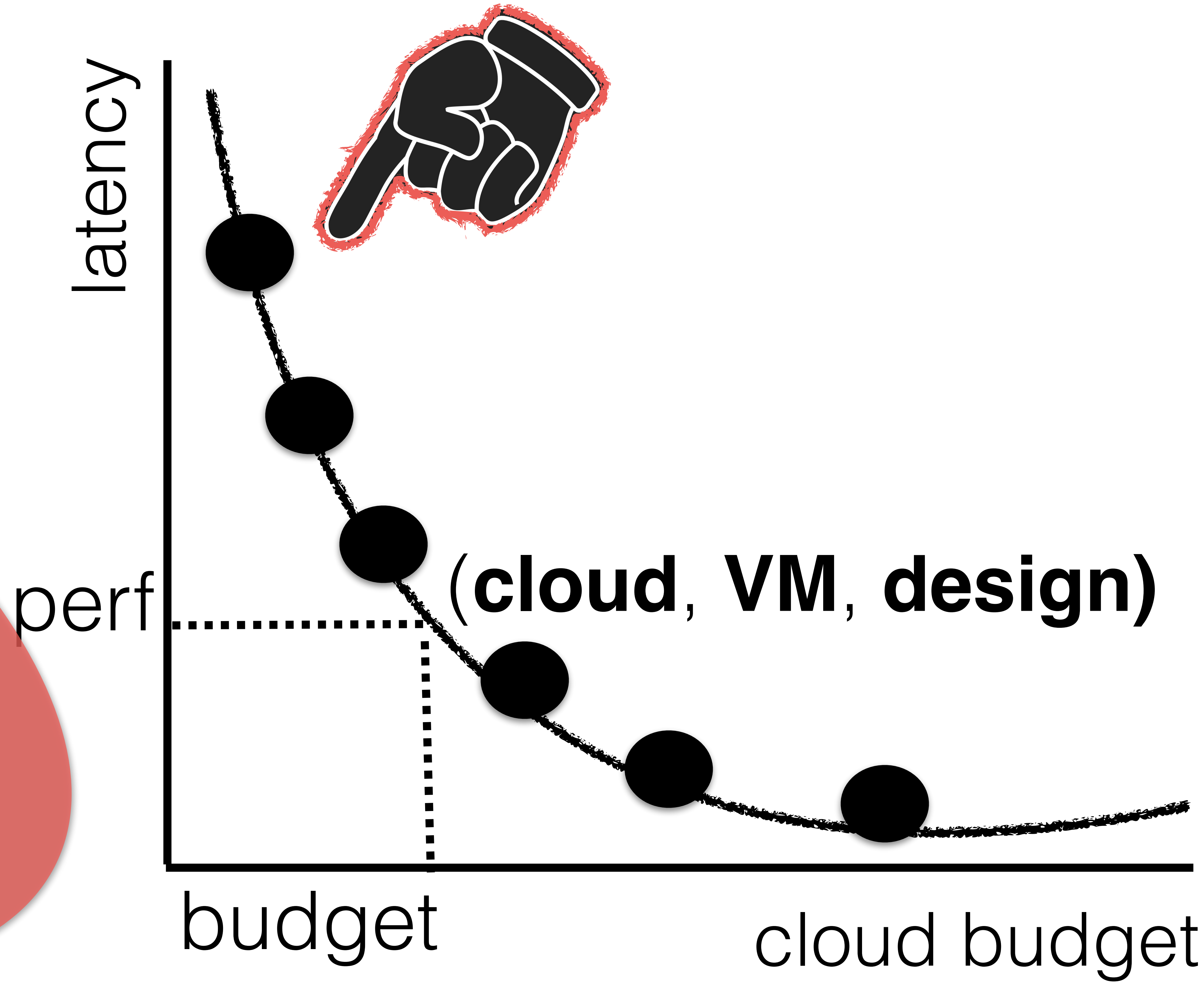
learned CPU model

h/w, parallelism

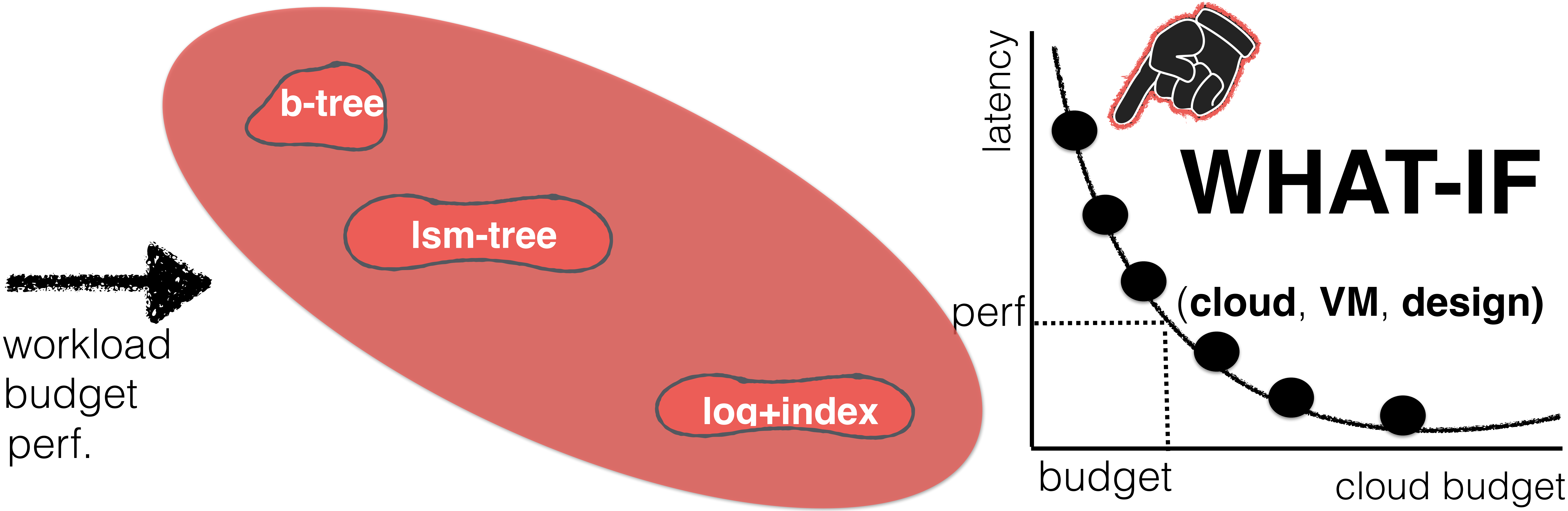
cloud cost mapping & SLA

AWS, Azure, Google

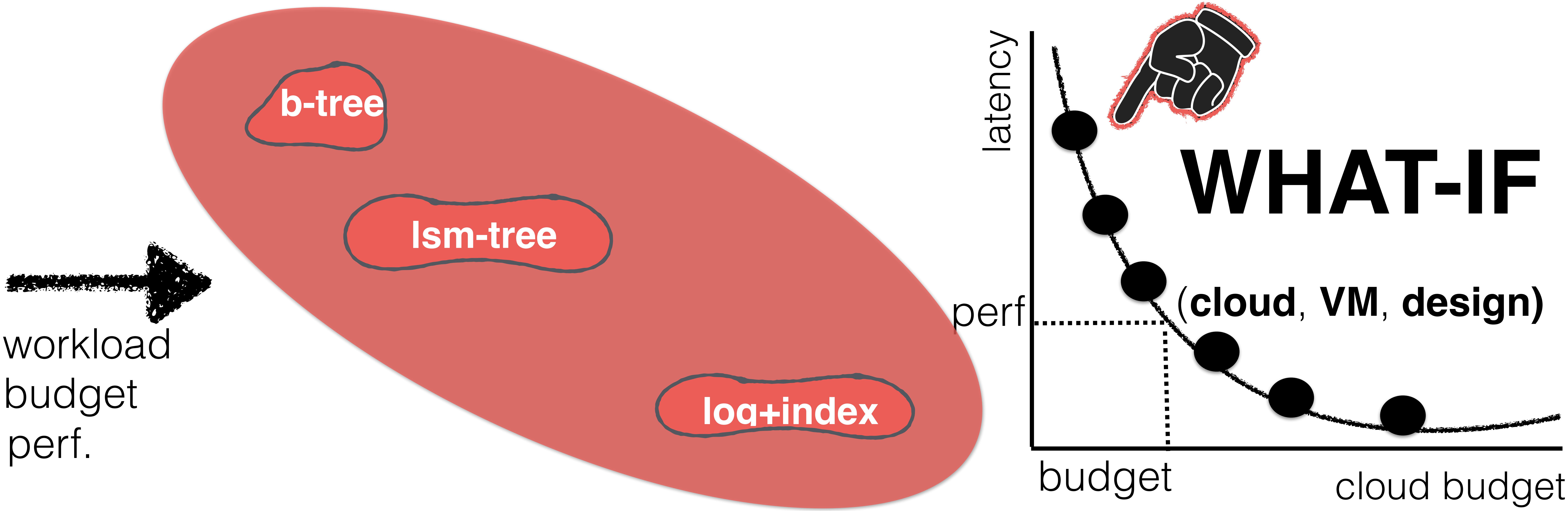




analytical I/O model	**learned CPU model**	**cloud cost mapping & SLA**
unified closed form	h/w, parallelism	AWS, Azure, Google



analytical I/O model	**learned CPU model**	**cloud cost mapping & SLA**
unified closed form	h/w, parallelism	AWS, Azure, Google



analytical I/O model

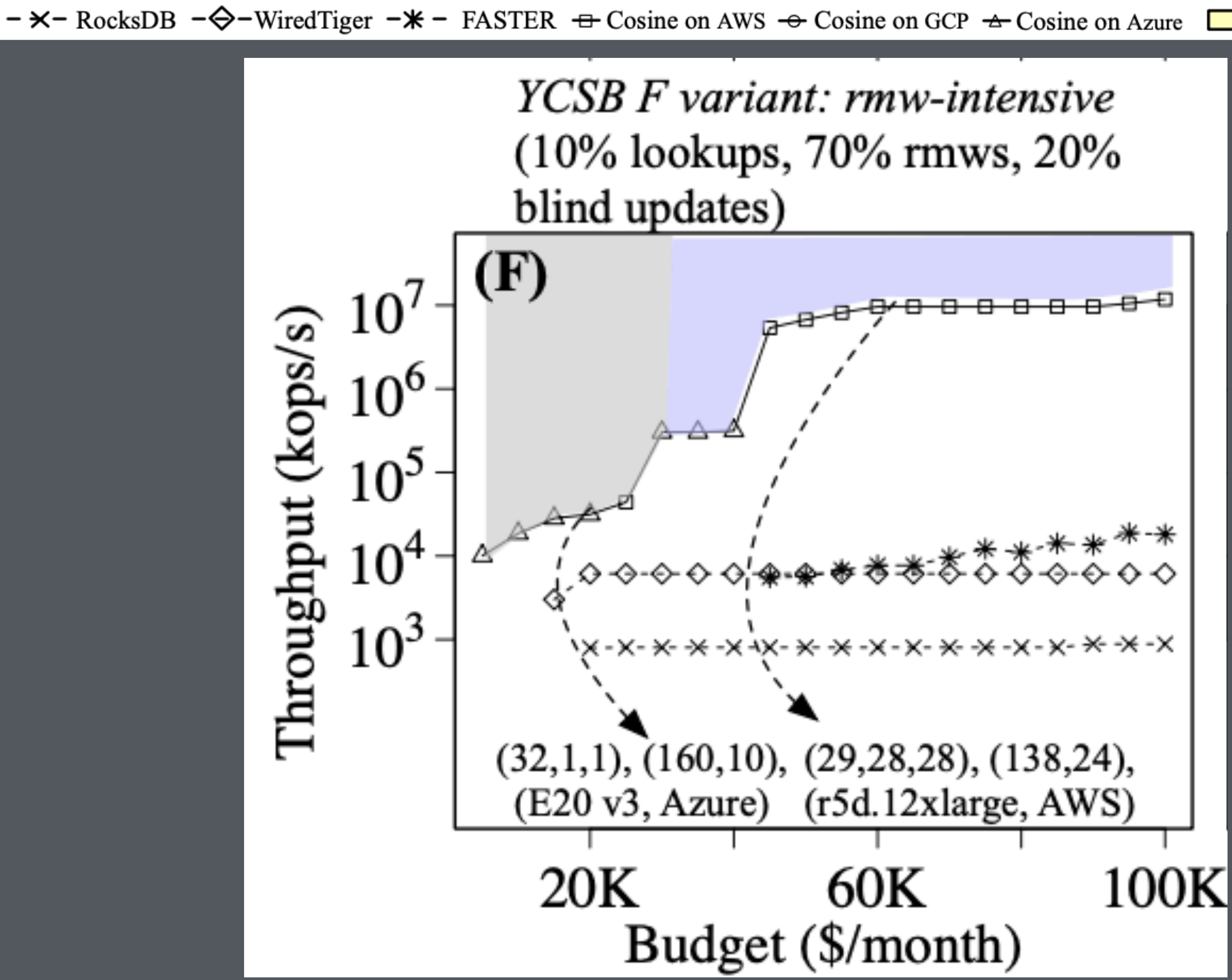
unified closed form

learned CPU model

h/w, parallelism

cloud cost mapping & SLA

AWS, Azure, Google



COSINE

State of the Art
Meta, Microsoft, Mongo

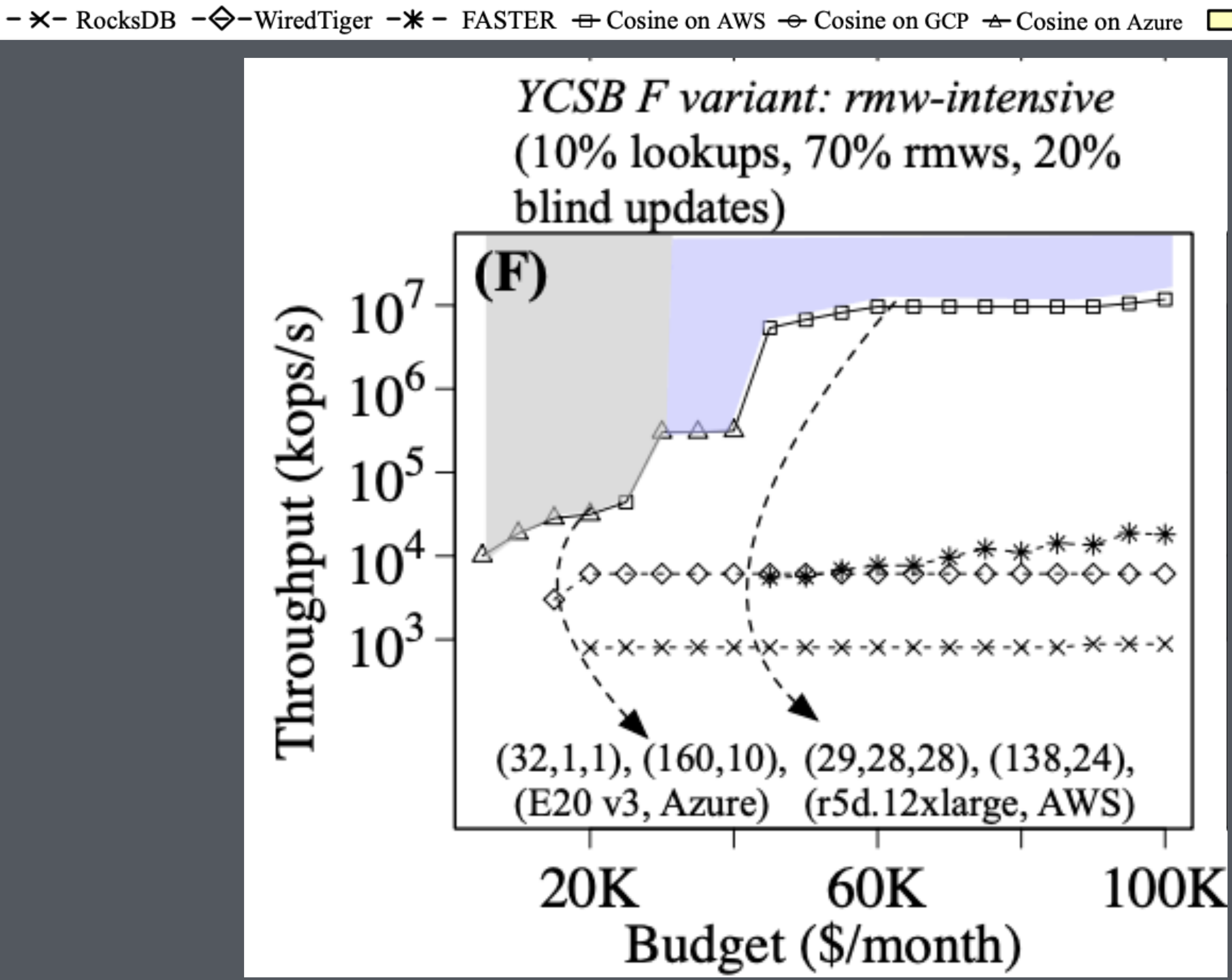
DASlab @ Harvard SEAS

COSINE

Better throughput/cost

State of the Art
Meta, Microsoft, Mongo

@ Harvard SEAS

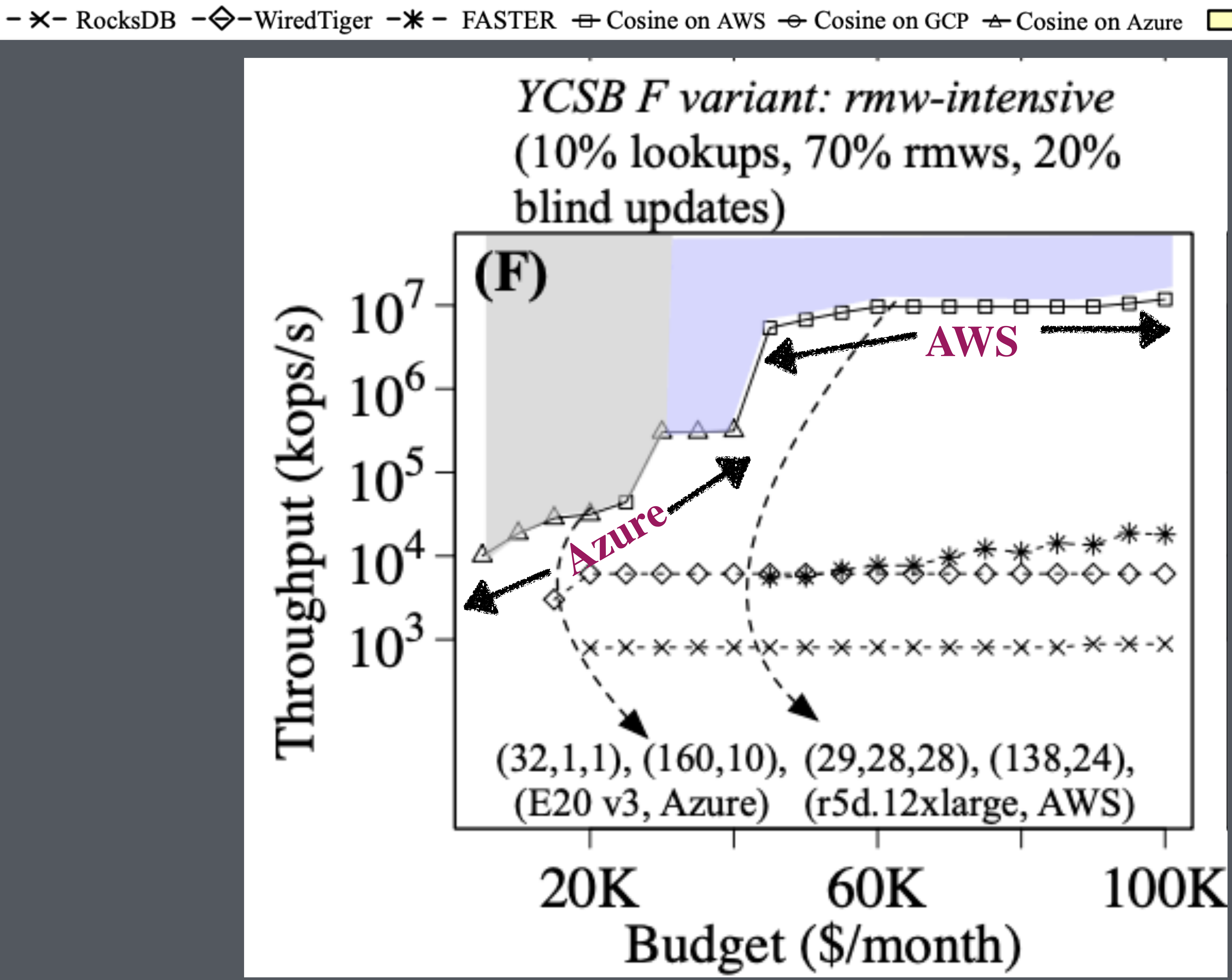


COSINE

Better throughput/cost

Self-designs

State of the Art
Meta, Microsoft, Mongo



COSINE

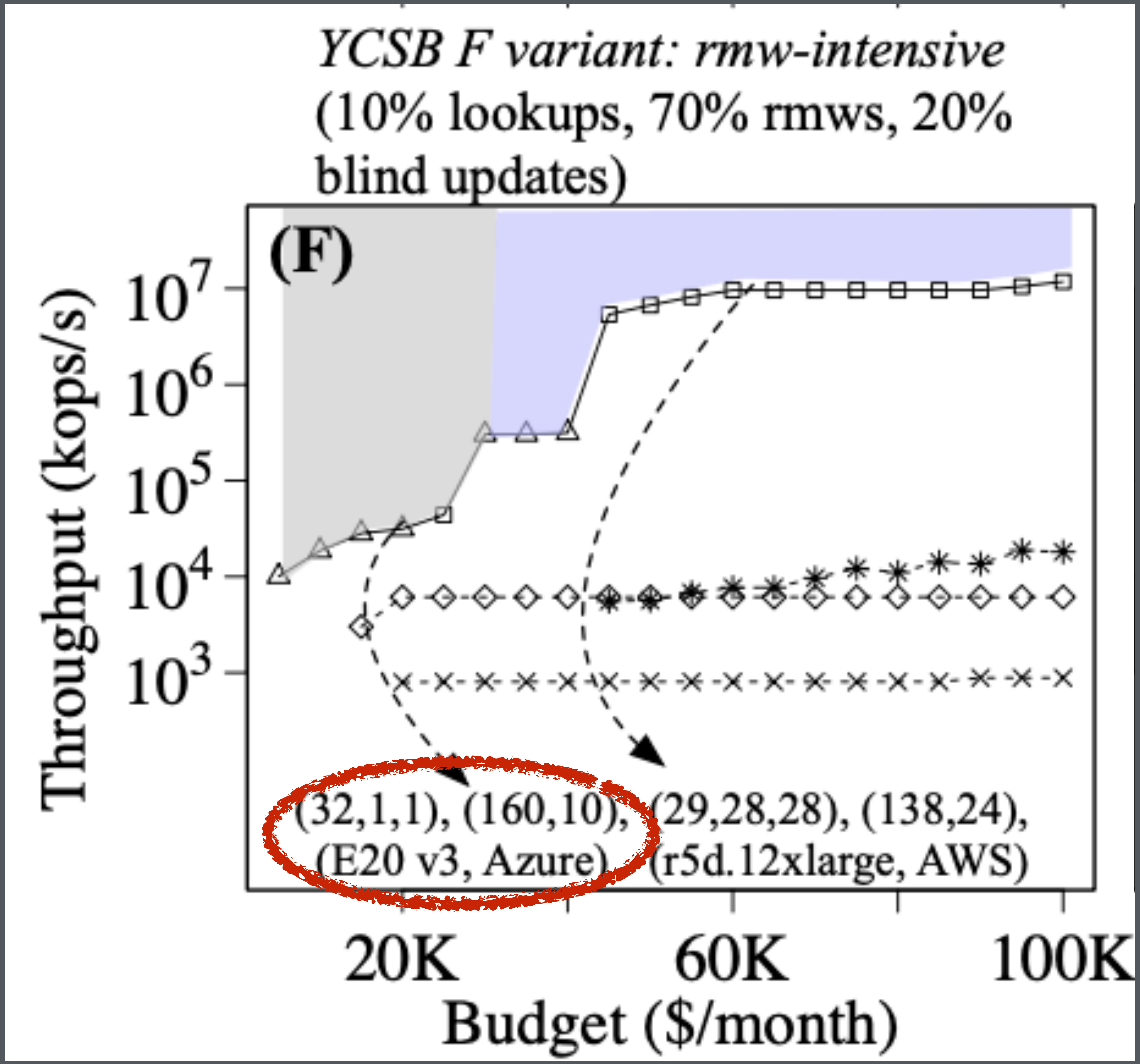
Better throughput/cost

Self-designs

State of the Art
Meta, Microsoft, Mongo

DASlab
@ Harvard SEAS

- × - RocksDB - ◇ - WiredTiger - ✳ - FASTER - □ - Cosine on AWS - ○ - Cosine on GCP - △ - Cosine on Azure | LSM class | B-tree class | LSH class | Hybrid class



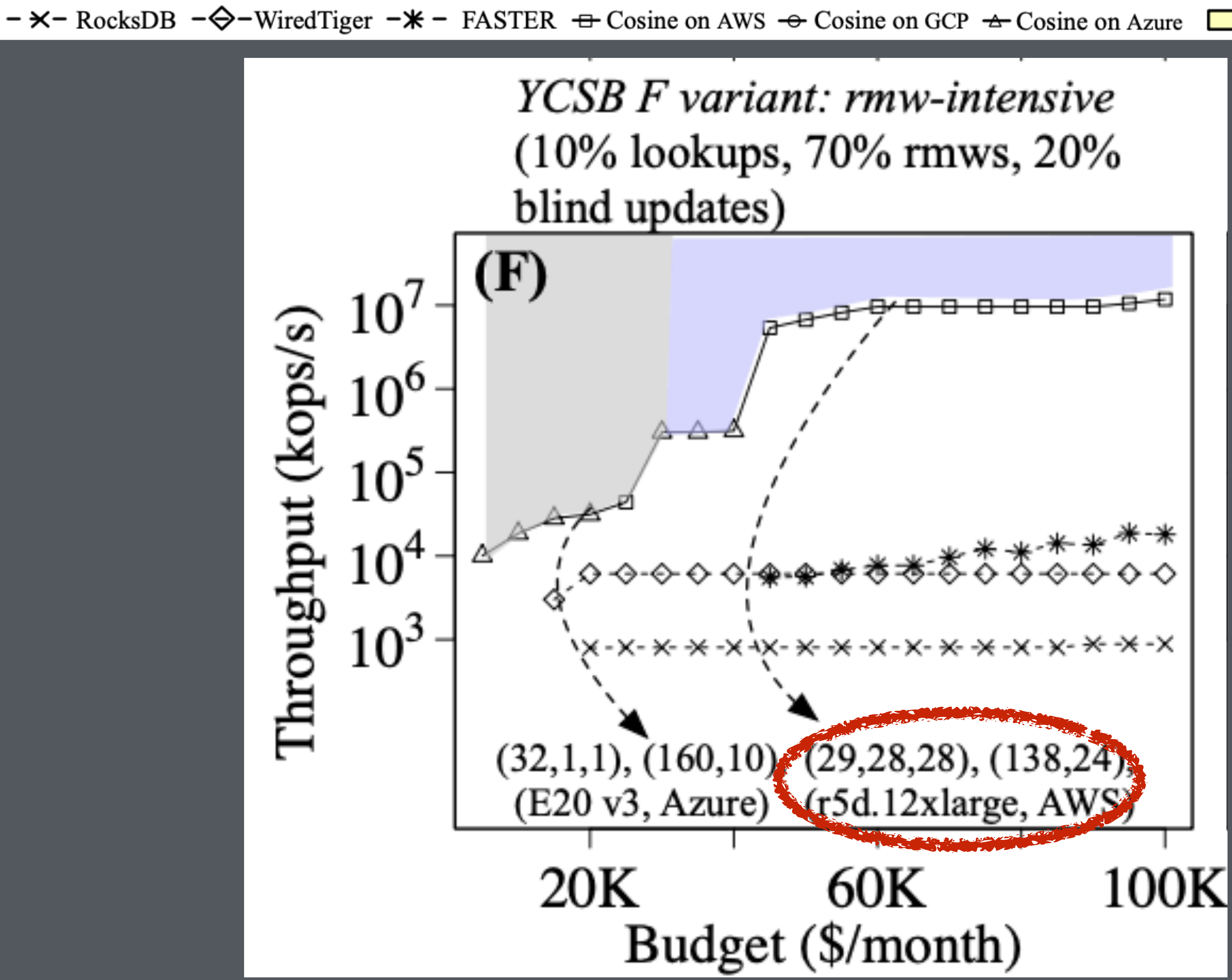
COSINE

Better throughput/cost

Self-designs

{ State of the Art
Meta, Microsoft, Mongo

DASlab @ Harvard SEAS



COSINE

Better throughput/cost

Self-designs

State of the Art
Meta, Microsoft, Mongo

DASlab @ Harvard SEAS

beats top systems across all workloads by self-designing new systems

workload/budget diversity

DASlab @ Harvard SEAS

IMAGE AI STORAGE

new primitives

accuracy

DESIGN SPACE

COST ESTIMATION

SEARCH


Remove rows/
columns
Processing
granularity
Sampling
Partitioning
Pruning
Quantization
Remove
unuseful data
Magnitude
reduction

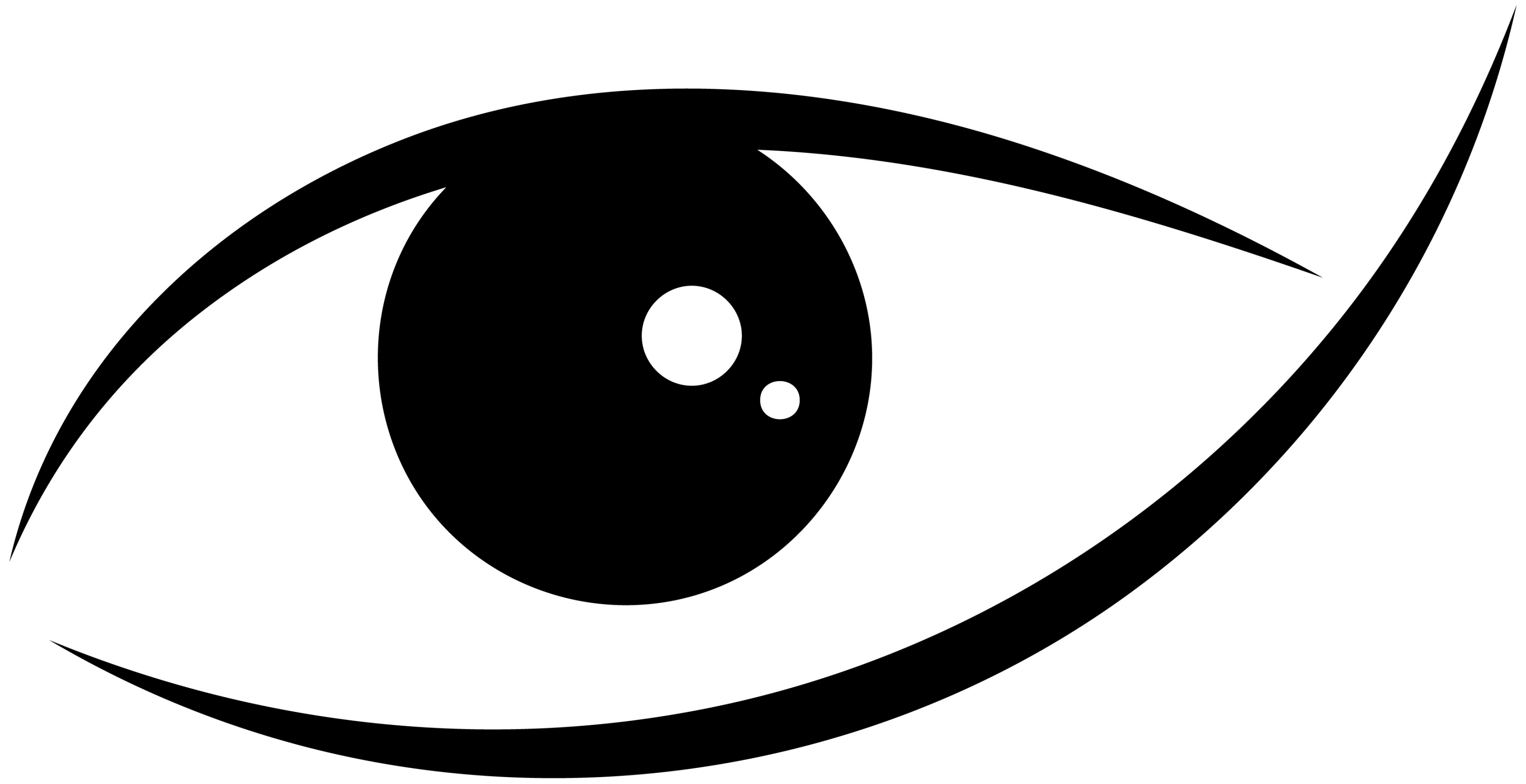
How do machines store images today?

How do machines store images today?

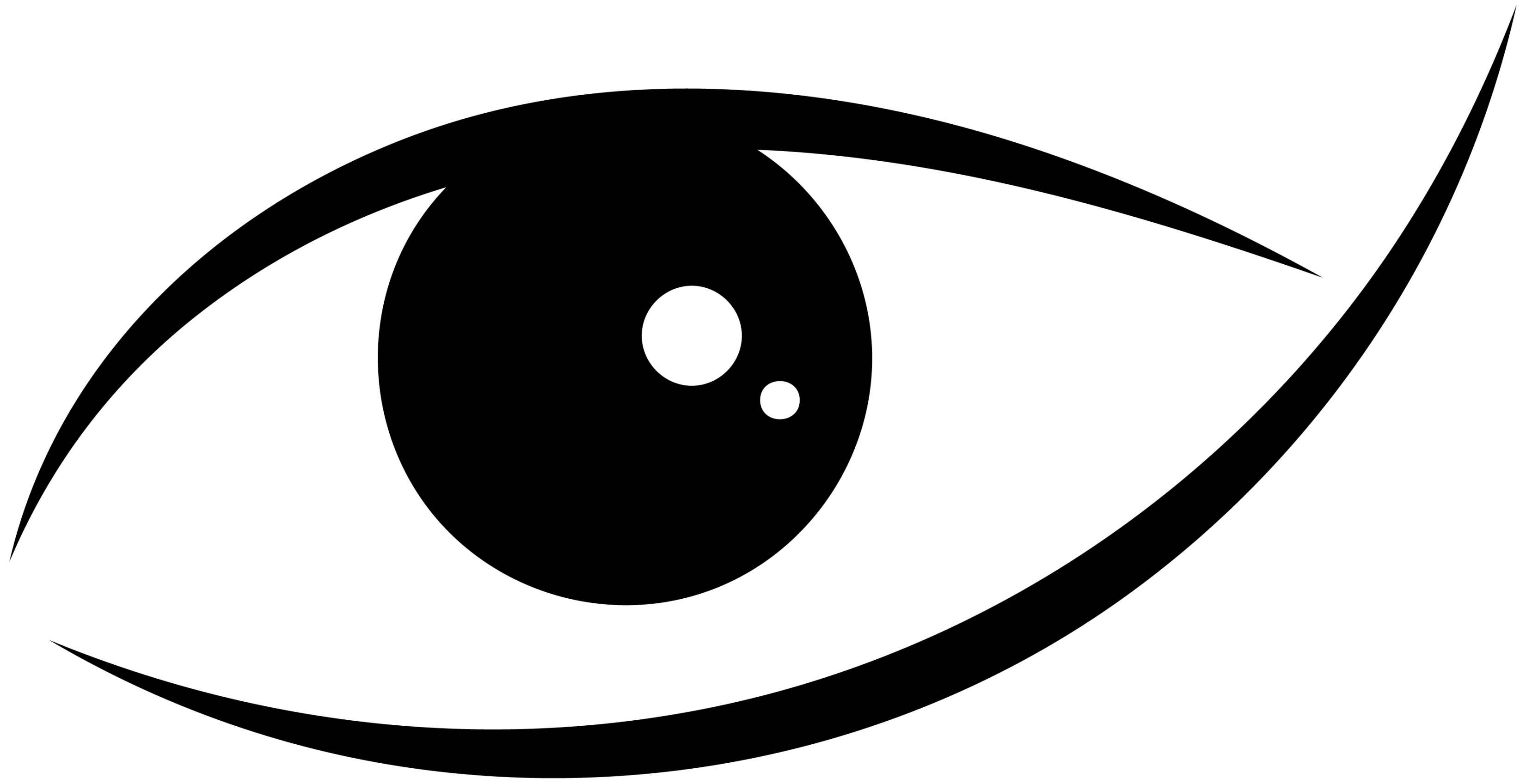
JPEG

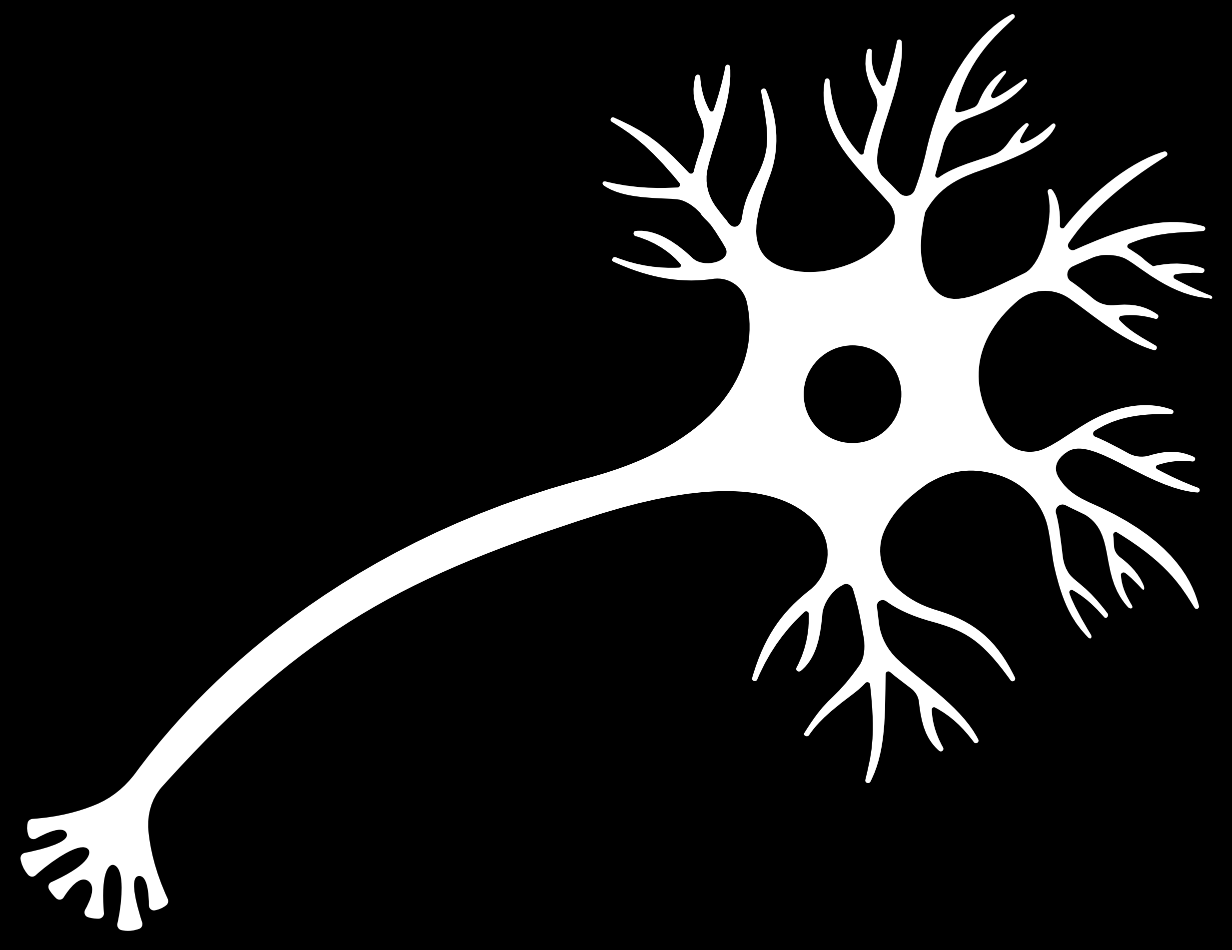
Joint Photographic Experts Group

JPEG is tailored for the properties of the human eye



JPEG is tailored for the properties of the human eye



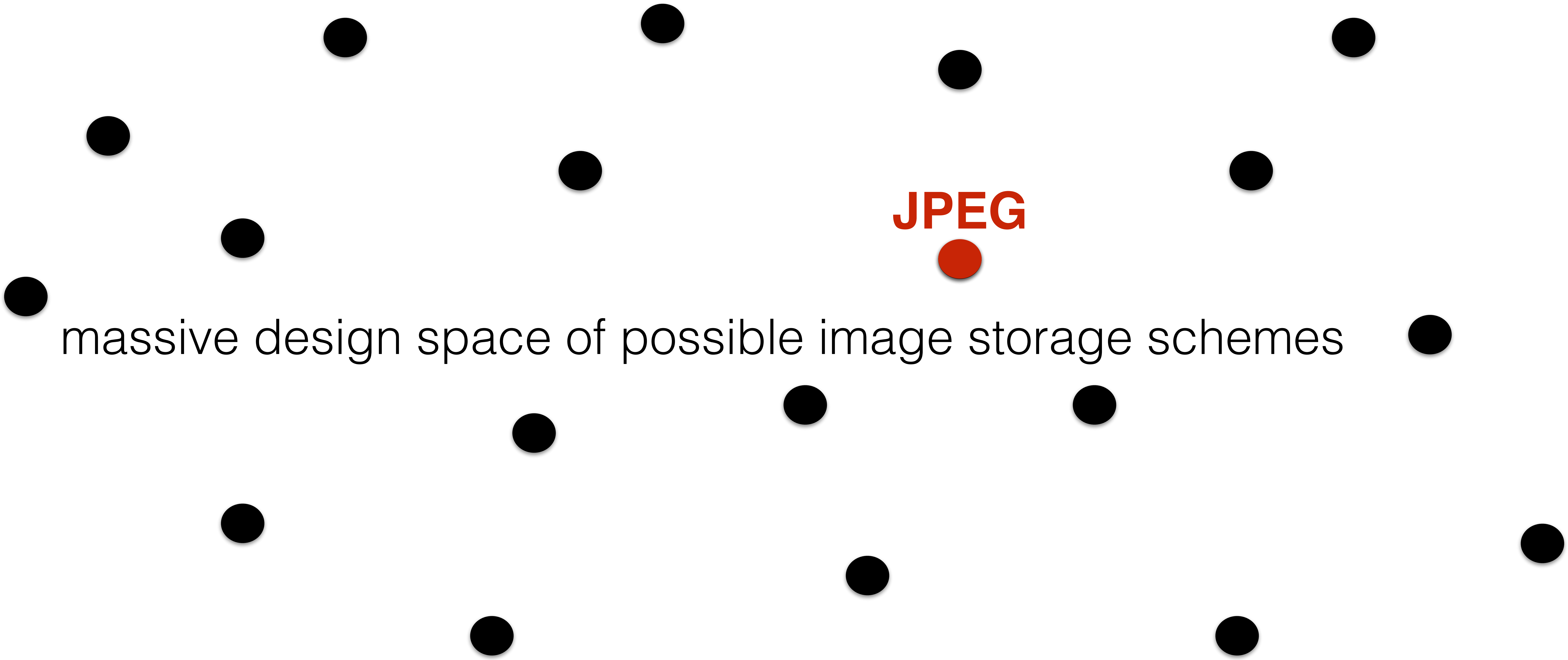


images for AI are seen by algorithms, not humans

10^150 designs

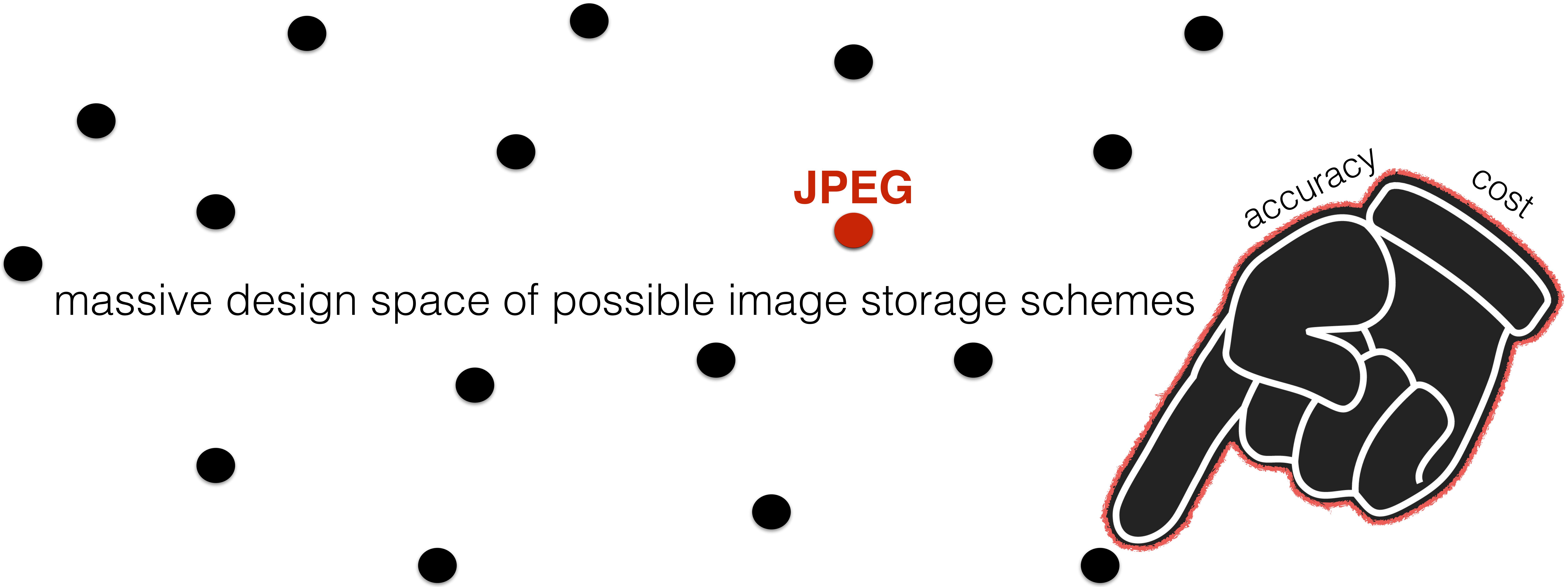
massive design space of possible image storage schemes

10^150 designs

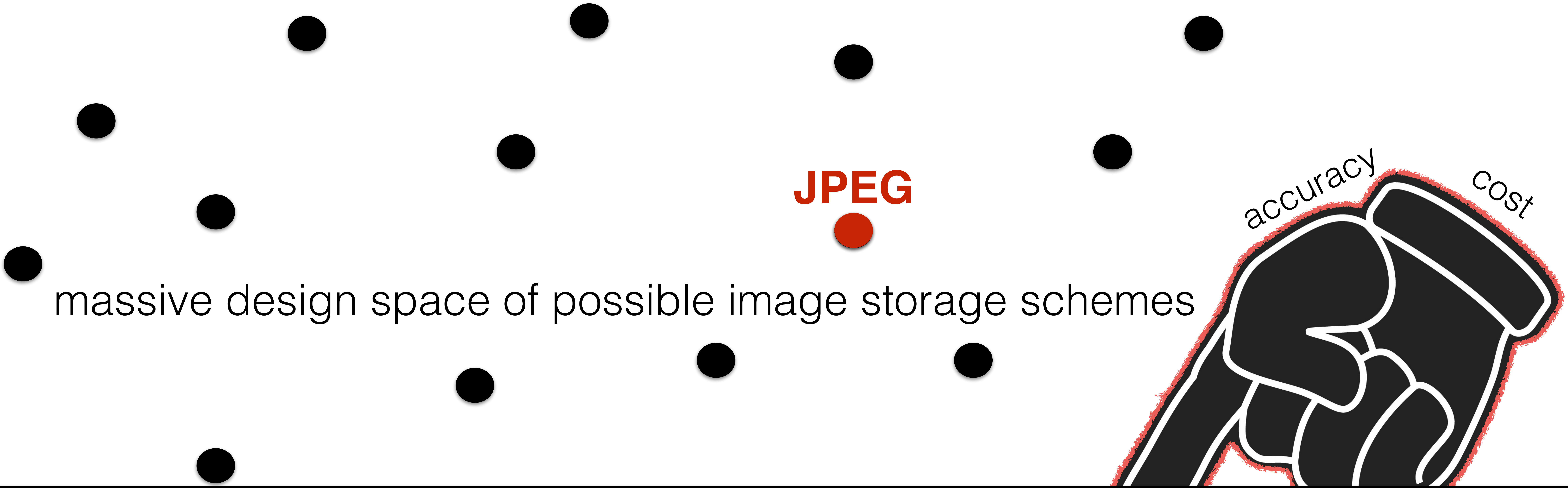


massive design space of possible image storage schemes

10^150 designs



10^150 designs



keep only the required bits

10x faster inference

LARGE MODEL TRAINING

DESIGN SPACE | COST ESTIMATION | SEARCH

GPU-based
Distributed

TorchTitan

- No failed training
- New algos at scale

DASlab
@ Harvard SEAS

S. BING YAO
models/advisors

DON BATORY
modular synthesis

JOE HELLERSTEIN
extensible indexing

STEFAN MANEGOLD
model synthesis

from one design at a time to design spaces

DESIGN SPACE | COST ESTIMATION | SEARCH

Tailored Systems, Accelerate Research

DASlab
@ Harvard SEAS

Next Steps

Research Topics for all data fields

Long-term: A Generilized Design Space Engine

Short-term: LLMs Design Space

Startup: E2E Data-centric AI Platform

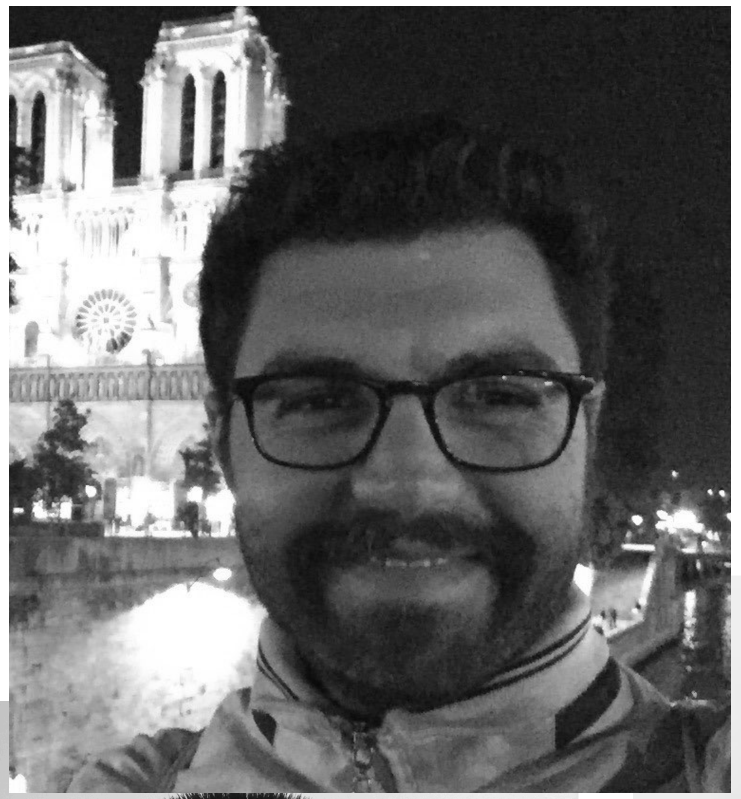






DASlab
@ Harvard SEAS

daslab.seas.harvard.edu




THANKS!