# The Design Engine: Interactive Data System Design *a.k.a* The Automated Researcher

Stratos Idreos, Manos Athanassoulis, Demi Guo, Michael Kester,
Lukas Maas, Niv Dayan, Brian Hentchel, Graham Lustiber, Angelo Kastroulis

Harvard University

## ABSTRACT

We propose reconsidering the process of designing data systems, moving from a process that is primarily based on the intuition and experience of system architects, to a process where intuition and experience complement formal and systematic methods. We envision a *Design Engine* that allows architects to interactively design systems; they can easily combine design options, try out alternative designs at a fine granularity, get instant feedback on the impact of their design decisions, ask *what-if* design questions, get suggestions about good and bad designs, and even semi-automate the process of discovering entirely new and previously unexplored designs, that is, doing research.

## 1. INTRODUCTION & MOTIVATION

**Data Systems Design.** Designing new systems and variations of existing ones is a continuous process that has been active for as long as the database field [1, 15, 27]. Today it is more important than ever to design new and tailored systems given the ever growing types of data driven applications [41]. We refer to data system design as the *collection of design decisions that define the software architecture of a data system*. This includes all decisions that have to do with how we store and access data, what kind of functionality the system supports, and how data flows between operators.

Figure 1 depicts the path from design to deployment. Design is distinct from implementation. Implementation only comes when we have a promising design candidate. Typically, there are several cycles between design and implementation as architects may need to try out multiple designs before settling to the one that works well. This is because there is no good way to argue about the properties of such a complex design space accurately. In addition, design is distinct from tuning. Tuning refers to adjusting certain configurable parameters as allowed by the original design; it happens offline as part of the set-up process for a specific application by a database administrator or in the case of adaptive systems during query processing. Design, on the other hand, is the process of coming up with the original software architecture of the system, it is performed by the system architect and may expose one or more tuning knobs. Finally, by design we refer to both the process of figuring out a good design from the vast space of known solutions and to the process of finding a new design, i.e., doing data systems research.

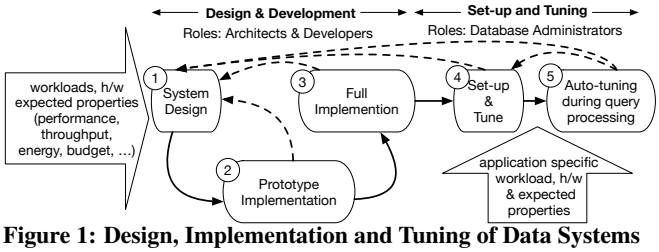


**Figure 1: Design, Implementation and Tuning of Data Systems**

**The Problem: Using (only) Human-driven Design.** There is extensive research on how to extend a system (typically by adding new data types) with pioneering works such as POSTGRES [40], Starburst [33], PROBE [16] and RAD [35]. In addition, past research has focused on how to easily synthesize and generate systems from high level components linked with clear APIs and embedded languages with works such as EXODUS [10], GENESIS [7], and with more dynamic solutions such as LegoBase [30]. Moreover, increasingly better debugging and performance profiling tools provide ideas on how to even treat system development as a data management problem [8]. All this research, though, centers around speeding up implementation. Architects still need to come up with good designs manually based on intuition and experience before being able to use any development tools; the fundamental problem of recognizing good and bad designs remains open [20]. For example, consider the following design questions.

1. Say we need a system for a specific workload (data & access patterns): Should we strip down a relational system? Should we build up a key-value store or main-memory system? Or should we design and build a new system from scratch?
2. Say the workload (read/write ratio) shifts (e.g., due to new application features): Should we use a different data layout for base data? Should we use differential updates? Should we use indexing? And if so, what kind?
3. Say we buy new flash drives with B% more bandwidth and add M% more system memory: Should we change the structure of our B-tree nodes? Should we change the merging strategy in our LSM-tree?
4. Say we want to improve throughput: Would it be beneficial to buy faster disks? Would it be beneficial to buy more memory? Or should we invest the same budget in redesigning a specific software component of the system?

**Complexity of the Vast Design Space.** The above is just a small sample of the hard design questions facing data system architects.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

We can rely on rules of thumb for some of the decisions [23] but even those need to be reconsidered frequently [22, 21, 18].

We can draw numerous examples where even very promising ideas are not easy to adopt and designing data systems becomes a multi-year process. A characteristic example is the column-store revolution of the past 10 years [1]. Many of the fundamental design concepts have been studied already in the 80s and 90s: storing data one column-at-a-time [14]; cache-conscious processing [39]; working over compressed data [19]; vectorization [37]; and a full columnar algebra [9]. This is not to say that there is nothing new with modern systems; quite the opposite [1], and of course modern hardware and data-driven applications created the right environment for these changes as well. Still, it took nearly forty years to adopt, adapt, and extend these ideas to a completely new system design that we now consider the obvious way to design systems for analytics. Roughly ten years after the community focused on systems tailored for analytics, several fundamental topics are still open, e.g., architecting an optimizer for main-memory systems.

Designing systems is an inherently complex process. Response time, throughput, energy consumption, hardware resources, budget, and application requirements are just some of the conflicting goals that matter. The design space is simply enormous.

*In an effort to enumerate the possible design options and their combinations D when designing a data store from scratch we found that* $D >> 10^{18}$.

Given this huge number of possible design combinations and the absence of design tools, the process of system design primarily relies on the intuition and experience of system architects to navigate the vast design space and bet on the right designs (we will explain this number in §2). Nevertheless, more often than not, we cannot know that a specific combination of design decisions works until we implement and test it. In addition, being certain that a design is the best for a specific case requires a comparison of all alternatives; in practice, this is infeasible as there are too many alternatives to manually test (or even list).

*System design and development costs a significant amount of time and money to get right [8, 13]. Given the vast design space, it often comes with uncertainty, compromise, and is hard to adjust when the environment changes (hardware, workload, SLAs).*

With new design ideas published every year, new hardware released or promised on a steady basis, and new applications with new requirements appearing as we collect more data, the design space only grows and becomes harder to reason about.

**The Design Engine.** In this paper, we propose to move toward a new design process where, in addition to intuition and experience, we also use tools tailored to assist with system design, bringing abstraction and structure to a complex process. We present the Design Engine that allows for storing, managing, and navigating fine-grained data systems design options in an interactive and (semi-) automatic way. Our vision is to make it easier and faster to:

1. Navigate the design space such that good (and bad) designs are not missed and systems can be designed quickly;
2. Adopt new design ideas and new hardware by making it easy to interactively test their impact on an existing system design;
3. Conduct data systems research by making it easy to discover and test new and unexplored design option combinations.

**Contributions.** We describe the research path and the basic components of the Design Engine (§2) as follows: (1) a way to structure fine-grained design options, their combinations and their cost components; (2) a hybrid model and experimental framework to interactively give feedback on the expected impact of design decisions across the whole system stack; (3) a design-suggestion mechanism that can auto-complete designs, discover bad design decisions in existing systems, and even unveil entirely new design combinations; and (4) a design language (and GUI) to interactively describe systems as sets of design options. We demonstrate several early examples that show strong potential for successful design navigation vs. completely manual design for core problems of system design. In addition, we demonstrate the potential for discovering and easily testing new design ideas by showcasing unexplored research areas that are naturally revealed by the Design Engine, including one idea that we already published in SIGMOD'16 (§3). Finally, we provide an interactive demo of the Design Engine (§3).

## 2. INTERACTIVE DESIGN

The engine helps architects navigate the vast design space by easing the design of any system (fixed or adaptive). The vision is captured by Fig. 2 and its main components are shown in Fig. 3.

**Design Library.** At the core of the engine lies the Design Library which maintains a structure of design options and their possible combinations (bottom left part of Figure 3). A design is defined as a set of design options that can be combined to cover a specific API. The granularity of the design options is a crucial parameter. For example, consider the following crude design space for access methods: $\{B\text{-}Tree, Array\}$. This only allows the inclusion of specific access methods. On the contrary, the Design Engine contains a fine-grained design library. For example, for array structures it includes options like ghost values (and with what frequency), partitioning (how many) and with which method (sorting, hashing, radix), decisions about ordering within each partition (sorting, clustering), compression options, indexing options to navigate the array or the partitions (tree, bloom filter, zonemap), and so on. Similarly, for tree structures the engine includes options such as fanout, whether to link leaves together, node split strategy, whether nodes should follow a read or write -optimized design, and so on. The same discussion holds for algorithmic design options, e.g., consider the numerous options to scan data: with and without predication, SIMD, BitWeaving, loop unrolling, etc.

With this fine-grained design space, we can create numerous variations of a tree or an array, and crucially we can mix and match design options across the whole design space. This is how we get the design space size number in Section 1 of $D >> 10^{18}$ design combinations. Once we add such fine-grained design options across the whole system stack, the number of possible design combinations explodes. In fact, this is a rather conservative estimate based on our current effort to shape the Design Library. As a basic intuition on how the design combinations quickly add up consider a rather small design space of just 100 design options (and for simplicity assume that all combinations are valid). Then if a system needs say a unique combination of just 20 design decisions out of 100 we quickly get $\binom{100}{20} > 10^{20}$ possible combinations to test (and of course numbers such as 100 and 20 above, while enough to navigate small subspaces, e.g., specific data structures, they are small numbers when it comes to full system design).

**Design Options.** For each design option $d$, the library may contain information about: (1) a name and a textual description, (2) a set of required design options for $d$ to work, (3) a set of valid design options that can be combined with $d$, (4) a set of invalid design options, (5) a set of tuning parameters for $d$ and their possible values, (6) an API, and (7) cost estimation components: (a) a model, (b) a code template and (c) an array of tradeoffs. Only 1 and 6 are

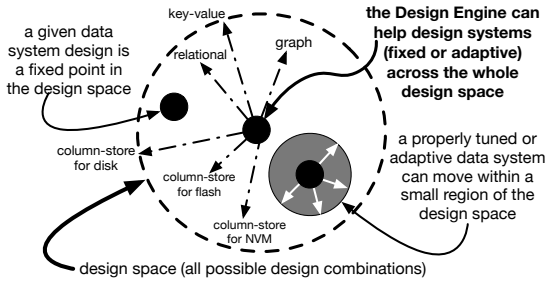


**Figure 2: Navigating the Design Space with the Design Engine.**

mandatory - in most cases it is expected that designers incrementally add more information about design options. For example, a design option to "binary search" requires a sorted data object (e.g., an array) and it is compatible with other design options over an array, such as including ghost values, or partitioning. However, it should not be used when the array is smaller than a cache line, or bigger than memory. It has an API that supports point and range queries. Lower level algorithmic design options may be used to potentially optimize binary searching an array. For example, how to choose the next pivot or loop unrolling do not have an API and do not affect the API in any higher level algorithm that they may be used for. Instead, they represent valid transformations of higher level design options and may lead to improvements depending on data and hardware specifications.

Design options are organized into Design Subspaces. In Figure 3, Partitioning, Logarithmic Design, etc. are all subspaces. Subspaces work recursively. Figure 3 shows how one level higher we can have even broader subspaces such as Access Methods (that contain all the above), Execution Strategies, and Optimization Strategies. Subspaces do not define the design space in the same way design options do; instead, they help illuminate an (evolving) structure of the design space and can be considered as "shortcuts or views" of part of the design space. Subspaces enable easier browsing of the design space by human designers who add intuition and provide hints to design algorithms (e.g., that a single solution may be picked from within a design space). Architects may define numerous subspaces to organize the design space. Each subspace is defined as a set of design options from the library and each design option may exist in more than one subspaces.

**Interactive Design.** The engine supports several types of "Design Queries" as shown in the middle box of Figure 3. For these to work, it needs to know the target environment/application and target properties of the system (top part of Figure 3). Knowing environment details such as properties of the memory hierarchy and CPUs, workload properties such as data/query samples and read/write ratios, and system goals such as minimizing latency or maximizing throughput, and SLAs, allows the engine to optimize for specific cases when it ranks design options.

*1) Design Suggestions.* Given an existing design, an architect may ask "what next steps are possible?" The engine returns all design combinations marked as valid with the current design. For example, the top part of Figure 3 shows an example of interactive suggestions for the structure of a hash table bucket. Design suggestions make it easy for designers to be aware of all valid design options at every step of the design.

*2) What-if Design.* What-if design queries allow architects to ask about the impact of a design change on an existing design (adding/removing options, changing tuning parameters or the environment). The engine resolves the new design and gets its total cost in terms of the target objective function (response time, throughput, etc.) with respect to the input environment/workload variables. The architect can then judge whether the change is a step forward and

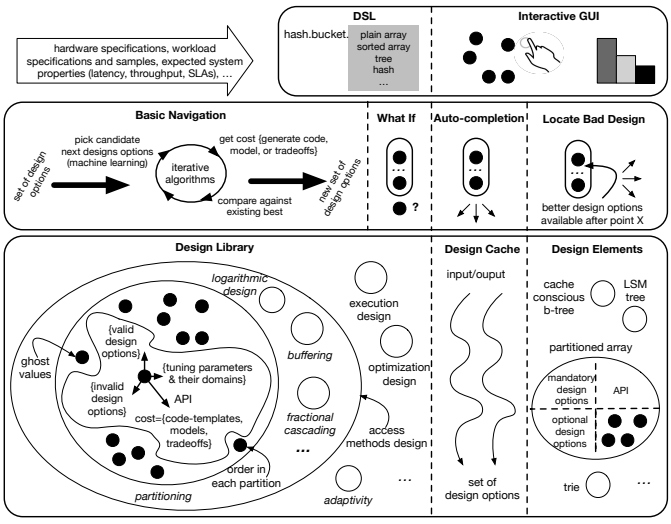


**Figure 3: Architecture of the Design Engine with examples of design options, spaces and elements.**

decide whether to move on or to backtrack (e.g., because a previous design combination a few steps back gave more desirable results or because given this result the designer develops a different design intuition to try out). Question 2 in Section 1 is such an example. The key is being able to argue about the impact of a design change. This comes from the three cost components of the design options: When models are available we use them to quickly estimate the effect of the design change; When code is available it can provide a more reliable estimate. That is, the engine materializes the code and tests the new design against the input environment and output properties required; If neither models nor code are available, then the engine uses the tradeoffs array. Tradeoffs give a high level balance for basic tradeoffs [4] such as whether a design change is expected to decrease or increase memory, write or read amplification. Even though we cannot give an exact cost with tradeoffs, we can estimate which design solutions are worth exploring.

*3) Auto-design.* The engine can auto-fill an existing design. For example, an architect may have already decided to create a hash table and subsequently assigns the design decisions about the shape and structure of the buckets to the engine. In its most generic form auto-design works in broader settings to find the best candidate design for a given API. For example, Question 3 in Section 1 is a combination of a what-if question with auto-completion, e.g., letting the engine auto-complete the structure of internal B-tree nodes choosing from any possible array designs.

The engine reduces the problem to a cost-based optimization problem inspired by evolutionary, genetic, and simulated annealing concepts to reach good design candidates. It goes iteratively through several designs until it stabilizes on a good selection (middle box in Figure 3). It starts with a set of $N$ solutions and computes their cost in parallel. A solution is a set of design options that supports the requested API. Some or all of these $N$ seed solutions can be chosen by the architect (or by query history) as good starting points - otherwise they are chosen randomly. The engine then crosses these solutions by randomly choosing pairs of solutions and valid design options from these pairs to create new solutions (children). Resulting children go under further mutations (with a given probability) by adding or removing additional design options. The next step is to decide whether to keep the resulting designs (for the next iteration of mutation and crossover actions). We use a simulated annealing approach where in every iteration the probability of choosing a design with a worse cost than before decreases so that initially we allow for more random walks in the design space. The

search stops when we go through $z$ iterations without improvement. The result of a search includes the top-k results so that architects may still manually evaluate which to use. Architects may also request and inspect the whole lineage of design choices. Even if it does not trigger any changes in the final design, having access to the top-k results and to the lineage improves the architect's understanding and intuition of the design space and it also helps spot opportunities to improve design option definitions (e.g., with better code templates or more accurate models).

*4) Find Design Errors.* Similarly, the engine can evaluate part or the whole design chosen by an architect to spot design errors (suboptimal designs) and to proppose better alternatives; Effectively this is resolved in the same way as described in the previous paragraph. That is, the engine finds the best design options it can to support the API for the target design and then it provides a diff between the old and new designs (set of design options and costs).

To boost interactivity when resolving complex designs, the engine provides a running estimate of how much of the design has been resolved so far and how long it may take to complete (in terms of the portion of the relevant design space that we have searched through). Results are also incrementally updated and thus designers can stop bad designs from taking too long to compute. By continuously trying to add/remove design options, ask for autocompletion, and change the input/output requirements, architects can navigate a vast design area and ask design questions (§1).

*Models Vs. Code.* The engine uses models and code templates to estimate design costs. While it is an open challenge to develop models that can describe sophisticated systems accurately, our intuition is that even if we achieve accuracy, it will be too complex, too expensive to compute and too hard to maintain. For example optimization of many of the design subspaces in data systems is already known to be NP-hard (such as optimal partitioning). Thus, we purposely build the Design Engine as a hybrid model/code-generation infrastructure; both processes return a cost that can be taken into account by the search algorithms. Models are used opportunistically whenever they are available and tradeoffs are used to speed up search when possible or to prioritize next steps. Code-generation is used in all other cases and it is also used as a feedback loop to verify and update the models and tradeoffs known for a design option.

*Design Cache and Elements.* To help bootstrap design queries the engine contains a Design Cache and Design Elements (bottom box of Figure 3). The engine caches past designs that have been found to work well for a given API and their input/output requirements. Then, these desings seed candidate solutions in the future for similar requirements. Similarly, a Design Element represents known (sub)designs that architects may want to reuse. Typical examples are standard access methods that can be reused in numerous designs as standard recipes or starting points.

*Storage.* The engine currently uses a columnar-like format to store design options. This gives a good reference performance given that the access paths of most of our design algorithms require iterating over a specific property of several design options at a time. There are numerous opportunities for tailored access methods mainly by adaptively and proactively collocating design options that are most likely to be part of a design.

**Discovering New Solutions.** An attractive property of the Design Engine is the ability to track novel design combinations that have not been studied before but may be appropriate for the input/output required. There are three ways that systems researchers create a new design and do research in general. The first one is about *tuning existing designs* in a different way than before. The second one is a significantly more complex process and involves *combining existing design options in a novel way*. That is, being able to argue about existing research and how parts of it could be combined in different ways to reach new properties. The third one, is about *creating a new concept*. This is the most profound (but also the most rare) type of research contribution; it creates opportunities to rethink the research space as the new concept can potentially be combined with every existing concept.

The Design Engine can automate the first two creative processes. This becomes possible by the fine-grained Design Library and the ability to map and try out any combination of existing design options and their tuning parameters with an estimation about the expected properties of the design. Essentially, it is all about *cross-pollinating* design options across the whole design space whenever the same API applies. To give a simple but characteristic example, consider plain arrays. In practice they form the most fundamental component of most data structures; nodes in trees or tries, buckets in hash tables, etc. Any idea that can improve an array in any way (e.g., BitWeaving for scanning the array [32]), automatically becomes a candidate design option for all data structures that use array-like components. Effectively, this allows data system researchers to easily spot new promising designs and frees them to focus more of their creative power on the hardest part of the creative process: coming up with new concepts.

**Design Language and GUI.** The engine supports a high level language that captures design options, subspaces, elements and their tuning parameters. The language allows the description of designs as sets of design options. For example, portions of a hash table access method may be described as *hash.bucket.sorted* which signifies that we define buckets of a hash table to be sorted. Similar to auto-completion features found in modern IDEs the engine can run auto-completion to show possible design options at every step of the design (see example at top box of Figure 3). It allows designers to quickly describe a new design or the design of an existing system and test it with the engine. The same functionality is provided by a GUI with which architects can quickly design systems, try out alternative designs, get recommendations for good designs and alerts for bad designs via a drag-and-drop interface (a screenshot of the current prototype is shown in Figure 6).

**Extensibility.** The Design Library can be extended at any time with additional design options to incorporate additional concepts/ideas, increasing the number of design combinations we can choose from. Architects are able to extend the Design Engine with new types of Design Options or by adding better code templates, models, and more complete connections for existing ones. Effectively extending the Design Library becomes the most critical job for database architects and the prime creative process; any new Design Option added reflects a new concept that can potentially be combined with all existing Design Options. The Design Engine may also be extended to consider further performance metrics - this is a trickier extension as it requires potentially updating the model of every design option. Once this is done, any design query on this metric can be achieved in an interactive way. Given the vast design space, we will open-source the engine so that all researchers and designers can update it and thus we can quickly incorporate new research and understanding about the design space to a common Design Library.

**Opportunities and Challenges.** In this section, we touched on some of the most basic research challenges and required steps to bootstrap the vision of the Design Engine. There are many more open topics. Characteristic examples include taking into account network cost and data allocation for distributed solutions, taking into account dollar cost of development, developing debugging and profiling tools for the Design Engine itself, being able to automatically map existing systems to the design space of the engine and even automatically recognizing new design concepts.

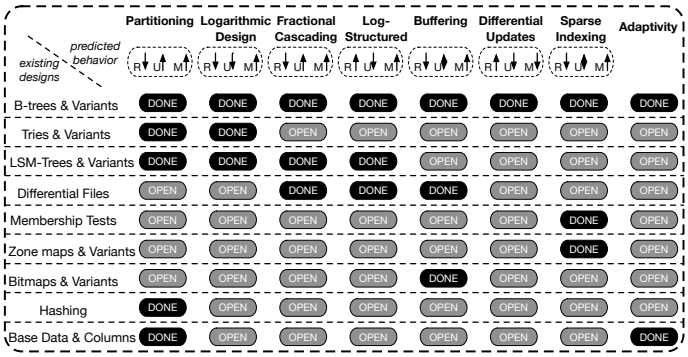


**Figure 5: Discovering new access methods designs.**


**Figure 6: The Design Engine in action.**

# 3. EVALUATION & DEMO

We now demonstrate that the Design Engine has strong potential to improve the process of system design. The way to judge the effectiveness of the Design Engine spans numerous metrics. We focus on three metrics here: (1) speed of the process of design, (2) quality of design, and (3) research opportunity. The benchmark (competition) for all of the above is human-only design.

**Architect vs. Engine.** We compare human-only design vs. engine-assisted design in the task of spotting design errors and providing better designs. To keep the experiment manageable, we tested human-only design for analyzing a design of ten options only. This is for the space of finding a good layout for an array of data given a specific data set, hardware configuration and read/write ratio. The designer had to evaluate a design based on a specific set of options: whether to partition the array and if so how; the order within each partition; including ghost values or not and, if so, how many per partition, etc. Then we changed the hardware specifications and asked if the design should change again and if so, how.

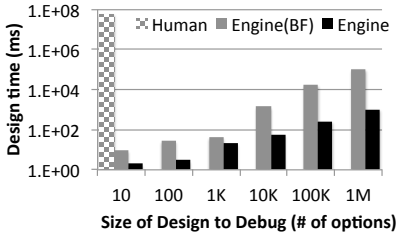


**Figure 4: Discovering errors.**

Figure 4 shows the results. We show two bars for the engine. The first one uses a brute force approach to test all possible design combinations. The second one (termed "Engine"), represents our current search algorithm based on the evolutionary and simulated annealing concepts to reduce the amount of design combinations it needs to consider. With both algorithms the engine provides a design with a 20% better cost several orders of magnitude more quickly than the human designer (we put an upper limit of eight hours for human-only design). We then repeated the experiment considering larger parts of the design space of data systems (without using a human designer). Figure 4 shows that the difference between using brute force and an evolutionary approach becomes increasingly more significant with the latter being two orders of magnitude faster than brute force when we try to spot errors in designs of one million design options. Even with one million design options, the engine can find the right design in just one second, while the human designer needed eight hours for only ten design options (the engine runs on a standard quad core modern server with all data hot in memory - Intel Xeon CPU E7-4820 v2, 2GHz, 16MB L3 cache with 8 hardware threads).

**Design Discovery.** Next, we discuss the strong potential for the discovery of new ideas. We show this by demonstrating that by having a fine-grained structure of the design space, new designs become apparent and all we have to do is navigate the space. Figure 5 shows the subspace of the data systems design space that has to do with access methods. We can use the seven subspaces in this figure to construct, and explain the behavior of, existing access method designs. "DONE" corresponds to a design that has been published in the literature. Observe that most existing access methods do not add new concepts; rather they are novel design combinations of existing options. "OPEN" corresponds to a design combination that has yet to be examined (to the best of our knowledge).

Even after several decades of research there are many open design combinations. The actual slots are more numerous than those shown in the table. For readability the table only shows subspaces instead of the design options included in these subspaces. By utilizing this structure, the Design Engine allows effortless navigation and recognizes promising design combinations and their potential behavior, covering two of the three creative research processes (tuning and combinations). In fact, we have already published one such solution inspired by the design structure [5] where we effectively used the combination of the differential updates, sparse indexing and adaptivity concepts (along with design options in these subspaces) to come up with a new bitmap-based access method that works well for both reads and writes. Similarly, we are currently investigating further opportunities that come out of observations as we construct individual subspaces for the engine. For example: (1) from hashing we have discovered new solutions that can treat buckets of a hash-table independently to tune their structure/size specifically for the incoming workload in each bucket, (2) from tree indexing we discovered an opportunity to treat subtrees independently and blend cache conscious designs that are read optimized with typical B-Tree designs that are more update friendly for different paths of a tree that receive different read/write ratios.

**Demo.** As part of our presentation we will include a demo of the Design Engine which will allow the audience to interactively browse the data system design space to quickly design systems, spot errors, experience what-if design and design suggestions, and compete with the engine for speed and quality of design. A screenshot of the current engine GUI is shown in Figure 6 where it shows the result on a "show me next design steps" -like query when designing an access method (the positive weight on a result design option means the cost will increase while negative is better). Up to date information about the Design Engine is maintained on its website: `http://daslab.seas.harvard.edu/design-engine/`.

# 4. RELATED WORK

This work is inspired from and builds on numerous past and current research lines both in the database community and outside. Due to limited space we only touch briefly on some of the stronger connections (extensible systems and systems synthesis).

Extensible systems have been studied across numerous areas of computer science. In databases the main premise is making it easy to add new data types and functionality to a system such that we can support new applications with minimal effort [16, 17, 33, 35, 40]. "Easy" means that developers can do this relatively quickly due to the careful interfaces provided and then, if done right, sophisticated database features such as concurrency control work out of the box for the new data types. Data system synthesizers extended this concept; The core idea is to have components that can be synthesized into full systems and interfaces that allow for autogenerating the final system code [7, 10, 30]. The same concept has been studied in other areas such as software engineering [38], computer architecture [36], and networks [31].

These lines of work have laid the foundations for the on-going effort to make system development easy. Our emphasis, though, is system design which comes before the development phase. Naturally, these two phases are strongly interconnected. For example, easier development also makes the process of designing easier because with such tools a developer/designer can more easily go through a bigger number of designs. Still, though, design now has to happen manually and relies primarily on intuition (and repeated experimentation) given the sheer amount of design combinations and conflicting goals. We see these lines of work as complementary to attack the bigger problem of "design and development" of data systems. Design Engines can help narrow down or eliminate the need to go through multiple designs and allow designers to focus quickly on the right design to develop.

From a technical point of view, part of our work relies on components that can be synthesized which is also seen in past modular systems. However, past approaches have used big building blocks to generate systems while we use fine-grained modules with additional structure about costs and design connections to be able to navigate the whole design space for design errors and design opportunities instead of only synthesis. Similar principles hold in keeping interfaces clean and simple [12] and our additional challenge here is to develop the infrastructure to interactively navigate the now massive space of modules. Conceptually, the work on Magic for layout on integrated circuits [36] comes closer to our goals. Magic uses a set of design rules to quickly verify transistor designs so they can be simulated by designers. Our proposal pushes interactive design one step further to incorporate cost estimation as part of the design phase by being able to estimate the cost of adding or removing individual design options which in turn also allows us to build design algorithms for automatic discovery of good and bad designs instead of having to build and test the complete design.

There are many more works in the database community from which we draw inspiration. A not complete list includes work on modeling [34], tuning via experiments [6], learning [2], flexible structures [26], what-if tuning [11], adaptive systems [3, 24, 28] and adaptation via iterative algorithms [25, 29]. These works mainly focus on tuning the knobs of an existing design and so they are complementary to our work that focuses on generating the initial design (see also Fig 1 & 3).

# 5. SUMMARY

We present the vision of the Design Engine to bring structure and abstraction to the complex and lengthy process of data systems design. The Design Engine allows data systems architects to interactively and semi-automatically navigate complex design decisions when designing or re-designing systems. The core innovation is a fine-grained and extensible structure of the design space along with algorithms and rules on how to navigate the vast number of choices interactively and automatically to get instant feedback during design. This brings numerous opportunities, including: being able to quickly spin off tailored systems for new applications; easily understanding the potential of new hardware and modifying systems to better utilize it; making it easy to discover novel combinations of design options that have not been studied before; making it easy to adopt new ideas in existing systems; and in general providing a unified framework that can accelerate systems research.

# 6. REFERENCES


[1] D. J. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases*, 5(3):197–280, 2013.

[2] M. R. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. J. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. Brainwash: A Data System for Feature Engineering. In *CIDR*, 2013.

[3] J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD*, 2016.

[4] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. In *EDBT*, 2016.

[5] M. Athanassoulis, Z. Yan, and S. Idreos. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *SIGMOD*, 2016.

[6] S. Babu, N. Borisov, S. Duan, H. Herodotou, and V. Thummala. Automated Experiment-Driven Management of (Database) Systems. In *HotOS*, 2009.

[7] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An Extensible Database Management System. *TSE*, 14(11), 1988.

[8] P. A. Bernstein and D. B. Lomet. CASE Requirements for Extensible Database Systems. *IEEE DEBULL*, 10(2):2–9, 1987.

[9] P. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, 1999.

[10] M. J. Carey and D. J. DeWitt. An Overview of the EXODUS Project. *IEEE DEBULL*, 10(2):47–54, 1987.

[11] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, 1997.

[12] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *VLDB*, 2000.

[13] A. Cheung. Towards Generating Application-Specific Data Management Systems. In *CIDR*, 2015.

[14] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *SIGMOD*, 1985.

[15] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation Techniques for Main Memory Database Systems. In *SIGMOD*, 1984.

[16] D. Goldhirsch and J. A. Orenstein. Extensibility in the PROBE Database System. *IEEE DEBULL*, 10(2):24–31, 1987.

[17] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *TKDE'94*, 6(1).

[18] G. Graefe. The five-minute rule twenty years later. In *DAMON*, 2007.

[19] G. Graefe and L. D. Shapiro. Data Compression and Database Performance. In *SAC*, 1991.

[20] J. Gray. What Next? A Few Remaining Problems in Information Technlogy. *ACM SIGMOD Digital Symposium Collection*, 2(2), 2000.

[21] J. Gray and G. Graefe. The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. *SIGMOD Rec.*, 26(4):63–68, 1997.

[22] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disc accesses and the 5 byte rule for trading memory for CPU time. *Tandem Computers - Technical Report*, 1986.

[23] J. Gray and P. J. Shenoy. Rules of Thumb in Data Engineering. In *ICDE*, 2000.

[24] R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, 2003.

[25] M. Heimel, M. Kiefer, and V. Markl. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *SIGMOD*, 2015.

[26] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *VLDB*, 1995.

[27] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a Database System. *Found. Trends Databases*, 1(2):141–259, 2007.

[28] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.

[29] Y. E. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. *SIGMOD'87*.

[30] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-Level Language. *PVLDB*, 7(10):853–864, 2014.

[31] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *TOCS*, 18(3):263–297, 2000.

[32] Y. Li and J. M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*, 2013.

[33] J. McPherson and H. Pirahesh. An Overview of Extensibility in Starburst. *IEEE DEBULL*, 10(2):32–39, 1987.

[34] A. Mönkeberg, P. Zabback, C. Hasse, and G. Weikum. The COMFORT Prototype: A Step Toward Automated Database Performance Tuning. In *SIGMOD*, 1993.

[35] S. L. Orborn. Extensible Databases and RAD. *IEEE DEBULL*, 10(2):10–15, 1987.

[36] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor. Magic: A VLSI Layout System. In *DAC*, 1984.

[37] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *ICDE*, 2001.

[38] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *TSE*, 5(2):128–138, 1979.

[39] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *VLDB*, 1994.

[40] M. Stonebraker, J. Anton, and M. Hirohama. Extendability in POSTGRES. *IEEE DEBULL*, 10(2):16–23, 1987.

[41] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*, 2005.