Design Tradeoffs of Data Access Methods

Manos Athanassoulis and Stratos Idreos
declarative interface
ask "what" you want

the system decides "how" to best store and access data

db system
data system kernel: a collection of access methods
an access method is a way to store and access data
an access method is a way to store and access data

**layout**
- e.g., array

**structure**
- unordered

**navigation**
- scan
an access method is a way to store and access data

<table>
<thead>
<tr>
<th>Layout</th>
<th>E.g., Array</th>
<th>E.g., Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>Unordered</td>
<td>Ordered</td>
</tr>
<tr>
<td>Navigation</td>
<td>Scan</td>
<td>Binary Search</td>
</tr>
</tbody>
</table>
isn't this a solved problem?
isn’t this a solved problem?

access method design is now as important as ever
today

data systems are nearly everywhere...

continuous need for new and tailored data systems
data systems are nearly everywhere…

continuous need for new and tailored data systems

today

tomorrow
today

data systems are nearly everywhere…

continuous need for new and tailored data systems

 tomorrow
disk   memory

A   B   C   D
A B C D

option 1

row-store engine

A B C

disk

memory
disk

memory

option1

A B C D

option2

A

column-store engine

row-store engine

how many more new access methods to design?
how many more new access methods to design?

it is not about radical new designs only!

design, tuning and variations
say the workload (read/write ratio) shifts (e.g., due to app features):
should we use a different data layout for base data - diff updates?
should we use different indexing or no indexing?
say the workload (read/write ratio) shifts (e.g., due to app features):
should we use a different data layout for base data - diff updates?
should we use different indexing or no indexing?

say we buy new hardware X (flash/memory):
should we change the size of b-tree nodes?
should we change the merging strategy in our LSM-tree?
say we buy new hardware X (flash/memory):
should we change the size of b-tree nodes?
should we change the merging strategy in our LSM-tree?

say the workload (read/write ratio) shifts (e.g., due to app features):
should we use a different data layout for base data - diff updates?
should we use different indexing or no indexing?

say we want to improve response time:
would it be beneficial if we would buy faster flash disks?
would it be beneficial if we buy more memory?
conflicting goals
(hardware and requirements change continuously and rapidly)

application requirements

moving target

budget

performance

hardware

energy profile
move from design based on intuition & experience only to a more formal and systematic way to design systems
goals and structure of the tutorial
structure design space & tradeoffs
highlight open problems towards easy to design methods
goals and structure of the tutorial
structure design space & tradeoffs
highlight open problems towards easy to design methods

basic tradeoffs goals & vision  
~30 min

design space  
~40 min

[slides available at daslab.seas.harvard.edu]
target audience = beginner to expert

no new designs but new connections & structure
NOT JUST SQL
+
operating systems, no sql, sciences
hardware is a big drive of access method (re)design (and it continuously evolves)
it is not just memory and disk
we want to move as few data items as possible
all the way up to the CPU
random access & page-based access

need to only read $x$... but have to read all of page 1

data value $x$

page1 page2 page3 ...
what is the perfect access method?
what is the perfect access method?

no single answer; it depends
what is the perfect access method?

no single answer; it depends

what is the application
read patterns
write patterns
reads/writes ratios
hardware (CPU, memory, etc)
SLAs
a perfect access method for reads (point queries)
a perfect access method for reads (point queries)
a perfect access method for reads (point queries)
a perfect access method for reads (point queries)
a perfect access method for reads (point queries)
a perfect access method for reads (point queries) but with no memory overhead

binary search to find(x)
a perfect access method for reads (point queries) but with no memory overhead

binary search to find(x)

sorted

reads
updates
memory
a perfect access method for reads (point queries) but with no memory overhead

binary search to find(x)

sorted

reads

updates

memory
a perfect access method for reads (point queries) but with no memory overhead

binary search to find(x)

reads
updates
memory
a perfect access method for reads (point queries) but with no memory overhead

binary search to find(x)

sorted
a perfect access method for writes (point writes)

update(x) → update log
a perfect access method for writes (point writes)

update(x)

reads
updates
memory

update log
a perfect access method for writes (point writes)
a perfect access method for writes (point writes)
a perfect access method for writes (point writes)
it all starts with how we store data

every bit matters
basic tradeoffs

Reads

Updates

Memory

RUM conjecture, EDBT 2016
Reads

Updates

Memory

Read

Update

Memory

read-optimized

update & memory optimized

memory-optimized
study basic access methods design components
	how they affect the RUM tradeoffs
how are they combined in existing access methods
study basic access methods design components

how they affect the RUM tradeoffs

how are they combined in existing access methods

Part 2
can we make it easy to design/tune access methods?
1 easily utilize past concepts
The log-structured merge-tree (LSM-tree)

do not miss out on cool ideas and concepts
2 do not miss out on cool ideas and concepts

The log-structured merge-tree (LSM-tree)

Google publishes BigTable

# of citations

move from design based on intuition & experience only to a more formal and systematic way to design systems
construct access methods out of basic components (and their tradeoffs) e.g., scan*, tree*, bloom filters, bitmaps, hash tables, etc.
INTERACTIVE DATA SYSTEM DESIGN/TUNING/TESTING
once we have a “complete” & navigable set of design modules

learn from: s/w engineering, modular dbs, compilers,
goes all the way back to basic texts
once we have a “complete” & navigable set of design modules

learn from: s/w engineering, modular dbs, compilers,
goes all the way back to basic texts

easy to change/adapt

easy to design
once we have a “complete” & navigable set of design modules

learn from: s/w engineering, modular dbs, compilers, goes all the way back to basic texts

easy to change/adapt

easy to design

universal
development
platform

testing
once we have a “complete” & navigable set of design modules

learn from: s/w engineering, modular dbs, compilers, goes all the way back to basic texts

easy to change/adapt

easy to design

universal development platform

testing

discovery of new combinations of design options
Part 2: observe how papers fill in gaps in the structure and existing open gaps