systems & research project
prof. Stratos Idreos

HTTP://DASLAB.SEAS.HARVARD.EDU/CLASSES/CS265/
filtering data: point/range queries

index knows order about the data
initial state
columns in
insertion order

A B C

sorted A B C
initial state
columns in
insertion order

sorted A B C

propagate
order of A
\textbf{select max(D), min(E) from R where} (A>10 and A<40) and (B>20 and B<60)

\textbf{avoid scan} of A
\textbf{avoid TR} on B
work on a \textbf{restricted area}
across all columns
good for memory hierarchy

\begin{itemize}
\item binary search for 10 & 40
\item for all B values between pos1 & 2: if B>20 and B<60
\item mark bit vector at pos i
\item for each marked position
\item max(D)
\end{itemize}
select max(D), min(E) from R where (A > 10 and A < 40) or (B > 20 and B < 60)
queries that filter on A benefit

queries that filter on B benefit
ok and how do we build, search, update a tree index efficiently?

structure = complexity = if statements, random access, instruction misses, etc.

= no free lunch

node size, data organization fanout...

(with range partitioning)
searching internal node

(v1,p1) (v2,p2) (v3,p3) (v4,p4) (v5,p5) (v6,p6)…
what does a leaf contain

searching leaf nodes
sorted array

page size: 64K - holds 16K 4 byte ints
N elements, P pages
info to navigate lower level value-pointer

12,20  35,...  50,...

<12  >=12

1,2,3,...  12,15,17  20,...

page size: 64K - holds 16K 4 byte ints
N elements, P pages
page size: 64K - holds 16K 4 byte ints
N elements, P pages

4+4 bytes for each page (value+pointer)
64K/8= index 8K pages

info to navigate lower level value-pointer

1,2,3...
12,15,17
20,...
35,...
50,...

sorted array

1,2,3...
12,15,17
20,...
info to navigate lower level value-pointer

page size: 64K - holds 16K 4 byte ints
N elements, P pages

4+4 bytes for each page (value+pointer)
64K/8 = index 8K pages

can index 8K pages of the next level
A B-tree diagram showing:
- **root** node with values 30, 50
- **internal nodes** with values 12, 20, 35, 50
- **leaves** with values 1, 2, 3, 12, 15, 17, 20
- **fanout**
random accesses

height \( \log_{\text{fanout}} N \)

leaves

1,2,3…

12,15,17

20,…

…

internal nodes

root

30,50

fanout

12,20

35,…

50,…
get 15
get 15-25

leaves
updates

1,2,3… 12,15,17 20,…

2,5,6,3,2

7,8,3,5,4

good
systems project
LSM-tree based key-value store

Modern B-Tree Techniques
by Goetz Graefe,
Foundations and Trends in Databases, 2011

The Log-Structured Merge-Tree (LSM-Tree)
by Patrick E. O’Neil, Edward Cheng, Dieter Gawlick, Elizabeth J. O’Neil
sorted on key

(key-value, key-value, key-value, …)
buffer/mem table

sorted on key

(key-value, key-value, key-value, …)
buffer/mem table

sorted on key
(key-value, key-value, key-value, …)

MEMORY

DISK

Level 0, Level 1, …
buffer/mem table

(sorted on key)

(key-value, key-value, key-value, ...)

Level 0, Level 1, ...

MEMORY

DISK

bloom filters
buffer/mem table

(sorted on key)

(key-value, key-value, key-value, …)

MEMORY

bloom filters

DISK

leveled

tiered

Level 0, Level 1, …
new data & updates are always applied at the buffer and duplicates are OK

buffer/mem table

MEMORY

DISK

sorted on key

(key-value, key-value, key-value, …)

bloom filters

leveled

tiered

Level 0, Level 1, …
new data & updates are always applied at the buffer and duplicates are OK

API: put, get, range, ...
design options: size ratio, memory allocation, structure of levels, # of runs, key-value layout,

buffer/mem table

sorted on key
(key-value, key-value, key-value, ...)

MEMORY

DISK

bloom filters

levelled

tiered

Level 0, Level 1, ...
new data & updates are always applied at the buffer and duplicates are OK

API: put, get, range, ...
design options: size ratio, memory allocation, structure of levels, # of runs, key-value layout,

buffer/mem table

TWO MILESTONES

a) One query at a time and
b) concurrent queries

support specific performance goals
provide design analysis and optimizations
final paper/presentation/demo
check project description, API, tests, goals
systems project is the default for everyone
numerous open research projects on top of that

research project
open to cs165 students and to students
that finish the systems project quickly
be able to query the data immediately & with good performance
be able to query the data immediately & with good performance

raw data → explore data and gain knowledge “immediately”
Initialization

Querying

Data layout

Idle time

Workload knowledge

External tools

Human driven
every query is treated as an advice on how data should be stored
continuous, lightweight actions to co-locate relevant data

every query is treated as an advice on how data should be stored
NORMALIZED DATA

good for updates, storage
but we need joins

DENORMALIZED DATA

only fast scans
but expensive to create,
storage & updates
normalized data
normalized data

possible denormalized space
continuously physically reorganize data based on incoming query patterns (joins)

normalized data

possible denormalized space
continuously physically reorganize data based on incoming query patterns (joins)

denormalized fragments
queries only need to fast scan

possible denormalized space
continuously physically reorganize data based on incoming query patterns (joins)

normalized data

denormalized fragments
queries only need to fast scan

possible denormalized space
continuously physically reorganize data based on incoming query patterns (joins)

denormalized fragments queries only need to fast scan

possible denormalized space
continuously physically reorganize data based on incoming query patterns (joins)

denormalized fragments
queries only need to fast scan

possible denormalized space

normalized data
but requires time, workload knowledge & stability
but requires time, workload knowledge & stability
# of queries

response time

standard

optimal

adaptive

but requires
time, workload knowledge & stability
WHY DO WE NEED TO GO BEYOND ADAPTIVE STORAGE
WHY DO WE NEED TO GO BEYOND ADAPTIVE STORAGE

data system X
WHY DO WE NEED TO GO BEYOND ADAPTIVE STORAGE

data system X

adaptive data system Y
HOW CAN WE EASILY NAVIGATE THE WHOLE DESIGN SPACE?
workloads, h/w expected properties (performance, throughput, energy, budget, ...)

1. System Design
   - Roles: Architects & Developers

2. Prototype Implementation

3. Full Implementation

4. Set-up & Tune
   - Roles: Database Administrators

5. Auto-tuning during query processing
   - application specific workload, h/w & expected properties
1) get N smart people into a “room”
2) give them T time
3) hope for the best
1) get N smart people into a “room”
2) give them T time
3) hope for the best

MAGICAL BUT

slow
prone to errors
usually no guarantees
hard to change
say we need a system for workload X (data/access patterns):
    should we strip down a relational system?
    should we build up a key-value store or main-memory system?
    should we build something from scratch?
say we need a system for workload X (data/access patterns):
 should we strip down a relational system?
 should we build up a key-value store or main-memory system?
 should we build something from scratch?

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 need a system for workload X (data/access patterns):
    should we strip down a relational system?
    should we build up a key-value store or main-memory system?
    should we build something from scratch?

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 need a system for workload X (data/access patterns):
  should we strip down a relational system?
  should we build up a key-value store or main-memory system?
  should we build something from scratch?

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 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?
  or just spend the same budget on improving software?
conflicting goals
(hardware and requirements change continuously and rapidly)

moving target

application requirements

hardware

budget

performance

energy profile

even if we now have better:
languages, compilers, libraries, & “the cloud”
Disk memory

A B C D
A B C D → option1 → A B C → row-store engine
e.g., modern main-memory optimized systems:
first ideas in 80s (who remembers khoshafian?),
first advanced architectures in 90s,
first rather complete designs in early 2000s,
mainstream industry adoption 2010+
still limited indexing, limited cost based optimizations, …
how many options in designing a relational system or key-value store?

system design: all decisions/ideas that go into a db architecture (e.g., link leaves of b-tree, add N holes in vector, use bloom filter for runs of LSM tree, # of bits in Bloom filter, etc.)
how many options in designing a relational system or key-value store?

> 1 QUINTILLION

(and counting…)

system design: all decisions/ideas that go into a db architecture (e.g., link leaves of b-tree, add N holes in vector, use bloom filter for runs of LSM tree, # of bits in Bloom filter, etc.)
(semi-)automating the more “mundane” steps

the creative process is about:
1) tuning past ideas
2) combining past ideas
3) coming up with completely new ideas
easily utilize past concepts/designs
P. O’Neil, E. Cheng, D. Gawlick, E, O’Neil
The log-structured merge-tree (LSM-tree)

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

P. O’Neil, E. Cheng, D. Gawlick, E, O'Neil
The log-structured merge-tree (LSM-tree)
How to model/abstract design decisions?

How much can happen automatically (auto-design)?
How to model/abstract design decisions?

How much can happen automatically (auto-design)?

move from design based on intuition & experience only to a more formal and systematic way to design systems
<table>
<thead>
<tr>
<th></th>
<th>Partitioning</th>
<th>Logarithmic Design</th>
<th>Fractional Cascading</th>
<th>Log-Structured</th>
<th>Buffering</th>
<th>Differential Updates</th>
<th>Sparse Indexing</th>
<th>Adativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>predicted behavior</strong></td>
<td>R↓ U↑ M↑</td>
<td>R↓ U↑ M↑</td>
<td>R↓ U↑ M↑</td>
<td>R↓ U↑ M↑</td>
<td>R↓ U↑ M↑</td>
<td>R↓ U↑ M↑</td>
<td>R↓ U↑ M↑</td>
<td></td>
</tr>
<tr>
<td><strong>existing designs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>B-trees &amp; Variants</strong></td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td><strong>Tries &amp; Variants</strong></td>
<td>DONE</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td><strong>LSM-Trees &amp; Variants</strong></td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td><strong>Differential Files</strong></td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td><strong>Membership Tests</strong></td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
<td>OPEN</td>
</tr>
<tr>
<td><strong>Zone maps &amp; Variants</strong></td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
<td>OPEN</td>
</tr>
<tr>
<td><strong>Bitmaps &amp; Variants</strong></td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td><strong>Hashing</strong></td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td><strong>Base Data &amp; Columns</strong></td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
</tr>
</tbody>
</table>

"elements"
<table>
<thead>
<tr>
<th></th>
<th>Partitioning</th>
<th>Logarithmic Design</th>
<th>Fractional Cascading</th>
<th>Log-Structured Buffering</th>
<th>Differential Updates</th>
<th>Sparse Indexing</th>
<th>Adaptivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-trees &amp; Variants</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Tries &amp; Variants</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>LSM-Trees &amp; Variants</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Differential Files</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Membership Tests</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Zone maps &amp; Variants</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
<td>OPEN</td>
</tr>
<tr>
<td>Bitmaps &amp; Variants</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Hashing</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Base Data &amp; Columns</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
</tr>
</tbody>
</table>

predicted behavior
existing designs

studied/published
studied? unpublished
studied and failed? no application?
### Predicted Behavior of Existing Designs

<table>
<thead>
<tr>
<th></th>
<th>Partitioning</th>
<th>Logarithmic Design</th>
<th>Fractional Cascading</th>
<th>Log-Structured</th>
<th>Buffering</th>
<th>Differential Updates</th>
<th>Sparse Indexing</th>
<th>Adaptness</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-trees &amp; Variants</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Tries &amp; Variants</td>
<td>DONE</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>LSM-Trees &amp; Variants</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Differential Files</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Membership Tests</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
<td>OPEN</td>
</tr>
<tr>
<td>Zone maps &amp; Variants</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
<td>OPEN</td>
</tr>
<tr>
<td>Bitmaps &amp; Variants</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Hashing</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Base Data &amp; Columns</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
</tr>
</tbody>
</table>

>90% SIGMOD/VLDB/ICDE papers fit the table, their existence and properties can be predicted.
### Design Properties + Impact

<table>
<thead>
<tr>
<th>Access Methods</th>
<th>Partitioning</th>
<th>Logarithmic Design</th>
<th>Fractional Cascading</th>
<th>Log-Structured</th>
<th>Buffering</th>
<th>Differential Updates</th>
<th>Sparse Indexing</th>
<th>Adativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-trees &amp; Variants</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Tries &amp; Variants</td>
<td>DONE</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>LSM-Trees &amp; Variants</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Differential Files</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Membership Tests</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
</tr>
<tr>
<td>Zone maps &amp; Variants</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Bitmaps &amp; Variants</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Hashing</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
</tr>
<tr>
<td>Base Data &amp; Columns</td>
<td>DONE</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>DONE</td>
</tr>
</tbody>
</table>

>90% SIGMOD/VLDB/ICDE papers fit the table, their existence and properties can be predicted after 50+ years of research on access methods we have barely scratched the surface!
mapping the “DNA” of data systems

model impact & side-effects
map possible combinations
DSL/algebra/search over design space
it is a huge graph

what-if design
(hardware/software/performance)

find “optimal” solution for X given
design so far
or find errors in an existing design

ht=hashtable(…)
ht.bucket.

options:
plain array
sorted array
tree
trie
clustered
…
INTERACTIVE DATA SYSTEM DESIGN/TUNING/TESTING

analog: your IDE and GDB for system design

Stratos Idreos
self-designing systems (deep adaptive systems)
harder: transitioning cost 
+ limited search time

INTERACTIVE DATA SYSTEM DESIGN/TUNING/TESTING

analogy: your IDE and GDB for system design

Stratos Idreos
fine-grained decomposition of design components
a system is a set of design decisions
fine-grained decomposition of design components

A system is a set of design decisions

e.g., should we add ghost values in an in-memory contiguous array and how often?
models & testing

metrics: latency, throughput, energy

best wins

Read
Update
Memory

min
max
min
when do models stop

metrics:
- latency
- throughput
- energy

best wins
1. write/extend modules in a high level language (optimizations)

2. modules = storage/execution/data flow

3. try out >>1 designs (sets of modules)
understanding and navigating focused design spaces
building DSLs for DS subspaces, Read/Update/Memory/Concurrency
always optimal key-value store: 
a log, a sorted vector and an LSM Tree
are extremes of the same design continuum (b-tree is also not far away)
research project: navigate the design space of key-value stores
   study side-effects, transition costs, and procedures
   on top of Facebook’s RocksDB

themes:
1) drop cache,
2) size ratio,
3) level structure,
4) memory allocation across mem table/cache/bloom filters,
5) handle r/w spikes
6) optimize range queries
research project: navigate the design space of key-value stores
study side-effects, transition costs, and procedures on top of Facebook’s RocksDB

themes:
1) drop cache,
2) size ratio,
3) level structure,
4) memory allocation across mem table/cache/bloom filters,
5) handle r/w spikes
6) optimize range queries

FIRST STEPS:
form groups of three
understand the design space of LSM-trees
pick 2-3 of the goals above and start brainstorming
define extreme designs and behavior
sketch transition procedure and triggers
metric of success, goal
get familiar with RocksDB internals
research project: navigate the design space of key-value stores
study side-effects, transition costs, and procedures
on top of Facebook’s RocksDB

themes:
1) drop cache,
2) size ratio,
3) level structure,
4) memory allocation across mem table/cache/bloom filters,
5) handle r/w spikes
6) optimize range queries

FIRST STEPS:
form groups of three
understand the design space of LSM-trees
pick 2-3 of the goals above and start brainstorming
COME TO OH
FINALIZE PROPOSAL IN NEXT 3 WEEKS
metric of success, goal
get familiar with RocksDB internals
systems & research project

BIG DATA SYSTEMS

prof. Stratos Idreos