# Dingsbums

## A Data Storage System

Christoph Schwering

RWTH Aachen

July 25, 2009

# Data Model

- sparse table, record oriented
- record $r_k = (k, (A_{k,1}, v_{k,1}), (A_{k,2}, v_{k,2}), \ldots)$
  - $k$ is key, identifies record $r_k$: $k \mapsto r_k$
  - $A_i$ is attribute: $(k, A_{k,i}) \mapsto v_{k,i}$
  - $(A_{k,i}, v_{k,i})$ can be added/removed to/from record
- opt: attr + time: $(A_{k,i}, v_{k,i}) \rightsquigarrow (A_{k,i}, t_{k,i}, v_{k,i})$

# Data Model (cont.)

- sparse table, record oriented
- record $r_k = (k, (A_{k,1}, v_{k,1}), (A_{k,2}, v_{k,2}), \ldots)$
  - $k$ is key, identifies record $r_k$: $k \mapsto r_k$
  - $A_i$ is attribute: $(k, A_{k,i}) \mapsto v_{k,i}$
  - $(A_{k,i}, v_{k,i})$ can be added/removed to/from record
- opt: attr + time: $(A_{k,i}, v_{k,i}) \rightsquigarrow (A_{k,i}, t_{k,i}, v_{k,i})$

## Data Model

- sparse, 2-dim. [3-dim.] map: $(k, A, t) \mapsto v$
- lightweight attributes, rather belong to record than table
- btw it's read optimized (at least not write optimized)

# Data Model (cont.)

## Data Model

- sparse, 2-dim. [3-dim.] map: $(k, A, t) \mapsto v$
- lightweight attributes, rather belong to record than table
- btw it's read optimized (at least not write optimized)

in comparison to relational model:

- sparse: $r_{k_1}, r_{k_2}$ can have distinct attributes
- attributes refer to tuple rather than relation
  attributes are created on-the-fly
- less typed: $r_{k_1}[A]$ can have different type than $r_{k_2}[A]$
- weaker constraints: no integrity, only one key

# Intended Hardware Audience

- cheap PCs
  - Google cheap $\neq$ student cheap
- Gigabit Ethernet

- current testbed: two computers à
  - two cores with 2.6 GHz
  - 2 resp. 4 GB memory
  - 640 GB disk
  - Gigabit Ethernet
  - Linux

# Index File

- sorted index-structure for disk-memory
- stores key / value pairs
- keys are unique
- key / value pairs have variable bounded length

# Index File (cont.)

- $B^+$-Tree variant, implemented in Ada 95
- highly parameterizable:
  - key type, value type
  - key and value serialization $\Rightarrow$ compression
  - IO implementation
- max entry size $\cong$ 1000 bytes

# Heap File

- unsorted storage on disk
- item is interpreted as byte sequence
- items have variable unbounded length
- items tend to be large, e.g. 4 KB or 2 MB or so
- goal: read quickly, i.e. minimize disk seeks

# Heap File (cont.)

- *chunk*: seq. of consecutive blocks
- each value stored as 1 chunk
- indexes:
  - info index: chunk addr $\mapsto$ used/free & length
  - free index: free chunk size $\mapsto$ chunk addr

- read is very fast
  (i) look up length in info index
  (ii) seek position in data file, read chunk
- write & delete are a little bit more complicated
  - free chunk administration

# Data Layout

- vertical partitioning
  - group attributes to families
  - the attrs of a family share the index and heap file
  - attr + pattern matching $\Rightarrow$ family (like in Haskell)
  - one index [+ heap] $\leftrightarrow$ one column family
- horizontal partitioning
  - index and heap files are distributed over the cluster nodes

# Network File Abstraction

- segment address space in chunks, e.g. 64 MB
  - this chunk $\neq$ heap chunk
- copy of chunk index stored at every node:

$$S = 2^{26} = 128M$$

$$\begin{aligned}
\text{Address} &\mapsto \text{Node} \\
0 \ldots (1S - 1) &\mapsto 2 \\
1S \ldots (2S - 1) &\mapsto 1 \\
2S \ldots (3S - 1) &\mapsto 3 \\
&\ldots
\end{aligned}$$

- local caching at nodes
- failover-redundant, self-managing

# Network File Abstraction (cont.)

operations:

- read:
  - (i) ask local cache
  - (ii) request block from node
- write
  - (i) broadcast (addr,block); handle on receiving nodes:
  - (ii) if addr in local cache, invalidate copy
  - (iii) if addr on disk, write block to disk
- seek_new
  - (i) if last chunk on node has a free block, take this
  - (ii) create a new chunk on node, take first block
- locks: read lock local, write/certify lock must be confirmed by each node

# Network File Abstraction (cont.)

self-management, redundancy:

- ping
- backoff copies

# Chunk File System

- goal: minimize seeks
- very simple
  - no meta data
  - no directories
  - no POSIX-API etc., just a library
- chunks
  - minimal allocation unit
  - chunks are large, e.g. 512 MB
  - this chunk $\neq$ network FS chunk $\neq$ heap chunk
- file $=$ filename $+$ list of chunks

# Compression in $B^+$-Tree

- entries $(k, A, t)$ primarily sorted by record key $k$
- record key $k$ of type string in many tables

- no compression: very fast :)
- prefix compression: should be quite good in many cases, especially when keys are reversed URLs `com.dingbum.search/foo/bar.html`
- delta compression: current Levenshtein implementation rather slow $\mathcal{O}(n^2)$