DB101 – Storage formats

Goetz Graefe – Madison, Wis.

Agenda

- Introduction
- Devices, memory & storage hierarchy, caches & buffers
- Columnar storage, compression, zone filters
- Traditional row storage, primary & secondary indexes
- Domain indexes, master-detail clustering
- Log-structured merge-forests, stepped-merge forests
- Parallel & partitioned: mirrors, replicas, partitions, shards
- Free-space management, database catalogs
- Summary & conclusions

Topics omitted

1. Consistency checking – tables, indexes, databases, logs
2. Virtual, disaggregated, network-attached storage
3. Creation & maintenance algorithms, offline & online
4. Deferred maintenance of tables, views, and indexes
5. Automatic index tuning, ML @ DB
6. Physical database design as side effect of query execution
7. …
8. Data quality and cleaning

Memory and storage hierarchy

- CPU caches: L1, L2, LL – “rules of 5×”
  inclusive or exclusive, write-through or -back, LRU or ARC or …
- Main memory: 500 cycles to serve a cache fault
  traditional balance: 1 MIPS, 1 MB memory, 1 MB/s I/O; “5-min rule”
- Non-volatile memory: “imminent” for a decade
- SSD: 0.1 ms latency; flash translation layer
- Disk: 10 ms latency; 20 TB / 200 MB/s = 100,000 sec/scan
- RAID: 0: striped, 1: mirrored, 4: parity, 5: rotating, 6: double
- “Disaggregated” (network-attached & pooled) storage

Columnar storage

- Relation = rectangular table
- Queries retrieve & analyze specific columns in business intelligence, data analytics, “data wrangling” for ML

Speeding up large scans:

- Columnar storage
- Large transfers (I/O page)
- Parallelism
- Compression
- Zone filters

Compression in columnar storage

Compression feeds on repetition

- Storage efficiency (arrays without indirection)
- Dictionary encoding (best for strings)
- Frame-of-reference (best with correlation)
- Run-length encoding (best after sorting)
  - Equal values
  - Equal differences (e.g., in time series)

Zone filters

- Netezza: 3MB, single record type, separate min+max records
- Infobright: bit vector filters
- Zone filters: min+max, also second-to-min+max, bit vector

- bit vector filters: limited to exact-match look-up
- min+max $\Leftarrow$ correlation, e.g., order date & ship date
- second-to-min+max $\Leftarrow$ outliers

Row storage

- Page header + slot array (byte offsets within page)
  PageLSN, record count
  Poor man's normalized keys? Offset-value codes?
- Record header + slot array (byte offsets within record)
  Field count(s), null bits?

In-page compression:

- In-page dictionary
- Variable-size integers (e.g., counts)
- In-string dynamic compression

“N-ary” in-page storage organization

0962	Jane	30
7658	John	45
3589	Jim	20
5523	Susan	52

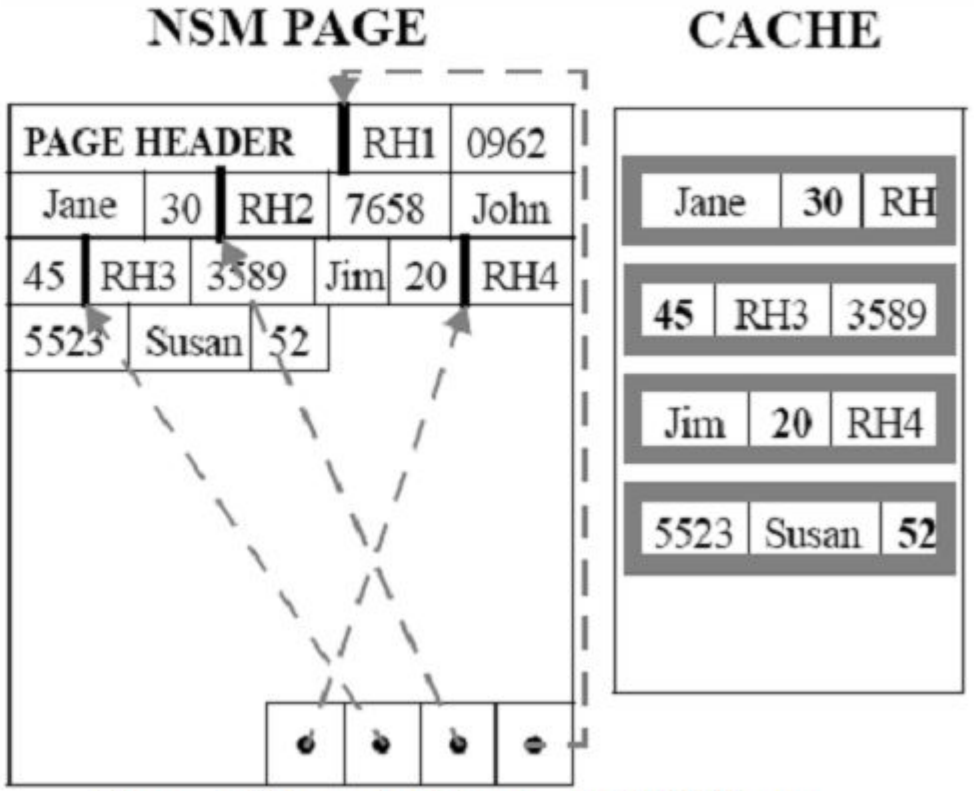


Figure 1. The cache behavior of NSM (from [ADH 01]).

PAX “partitioned across” in-page organization

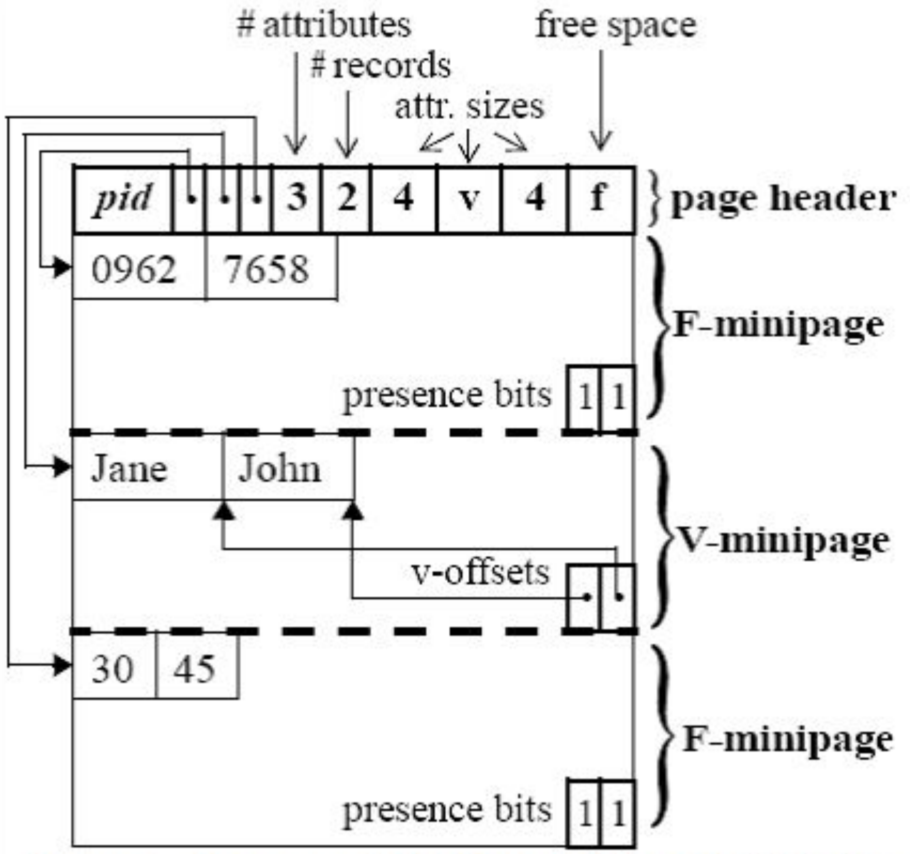


Figure 3. An example PAX page (from [ADH 01]).

PAX cache efficiency

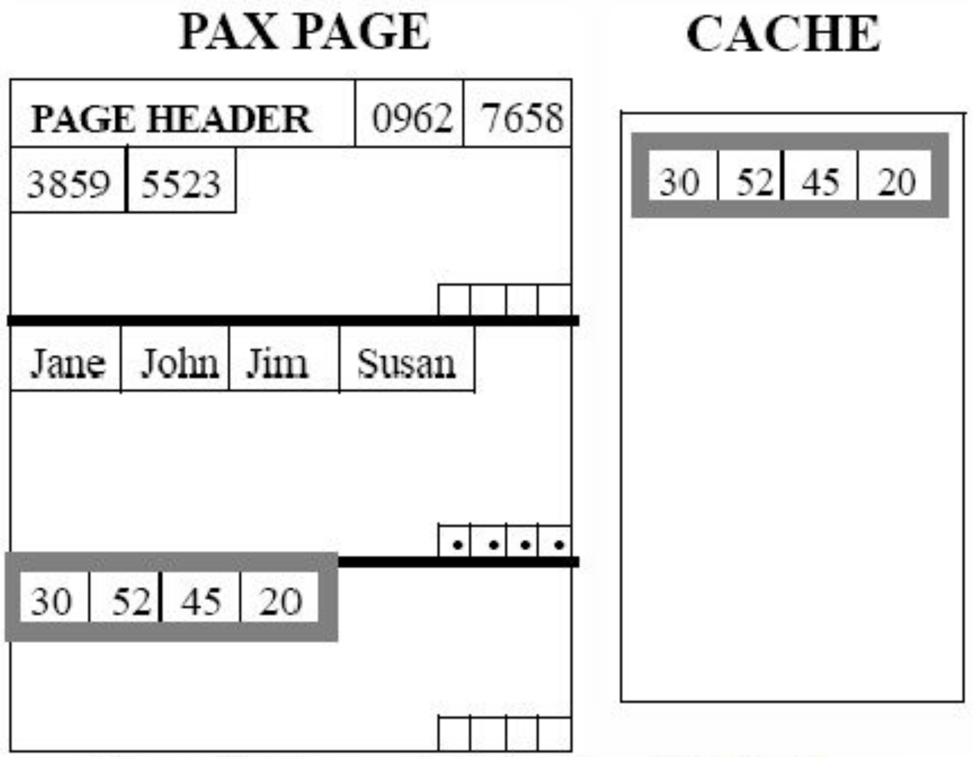


Figure 2. The cache behavior of PAX (from [ADH 01]).

Indexes

- Map keys to values
  C++ "map" templates, "key-value stores"
- Single-column index: key → row identifier
- Compound index: (key, key…) → row identifier
- Join index: key → row identifier + row identifier
- Domain index: key → table + row identifier
- Non-unique index: … → { … }
- Compressed index:
- Bitmap index: { integer }

Databases vs key-value stores

Recall "sharing structured data:"

- No schema, no integrity constraints
  No logical or physical data independence
  No secondary indexes, no materialized views
- No privacy, security, or retention policies
- No query processing (only "get" and "put" in b-trees)
- No (general) transactions, e.g.:
  No multi-row transactions, no cross-index consistency
  No controlled redundancy, no distributed transactions
  No phantom protection

Primary and secondary indexes

- Primary: all columns, defines row identifier
  aka clustered index, index-organized table
- Secondary: key → row identifier
  - Non-unique ⇒ multiple entries, { row identifier }, both
    Bitmap = set of integers, hybrid, "word-aligned hybrid"
    Compression of list vs bitmap
  - "Include" columns, e.g., all foreign keys
    "Covering index" for "index-only retrieval"
    Foreign keys: navigating index-to-index, row-to-row

Domain index

- e.g., CustId → { tableId, { row identifier } }
  all information about a customer in a single index search: accounts, invoices, payments, shipments, returns, reviews, appointments, email, letters, phone calls, online activity, loyalty program, referrals, …
- Not offered much, not used much – not strictly relational
- Alternative: (CustId, TableId) → { row identifier }
- Refinement: merged index (multiple record types)

B-trees

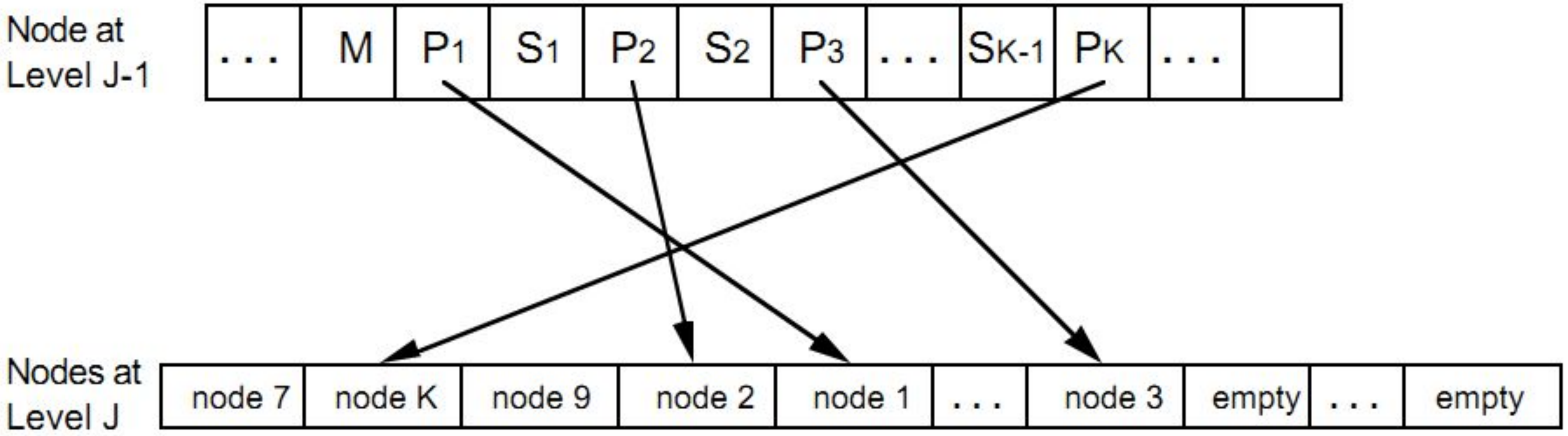
- 1970, 1972: b-tree, b^*-tree, b^+-tree…
- Today: data records in leaves, “guide” keys in branches
  Some: prefix & suffix truncation
- Future: additional information with branch keys?
  - ExpectedPageLSN → self-repairing b-trees
  - “Small materialized aggregates” – efficient materialized views
  - Zone filters (on values, maximum difference, etc.)
  - …

B-tree structure

- Creation: sort future index entries, build left-to-right leaf-to-root
- Index updates: sort future index entries, “merge into”
- O’Neil’s SB-tree (a b-tree within a b-tree)
- Write-optimized b-trees
  - Cheap page movement during write
  - Pointer swizzling in the buffer pool
  - Continuous comprehensive self-testing – “fence” keys
- Foster b-trees – low latching requirements

O’Neil’s SB-tree [1992]

- Allocate disk space in large blocks, allocate neighboring leaf nodes within blocks
- Split blocks 2→3 for ~85% utilization per page & per block
- Fast index-order scans, e.g., large range queries



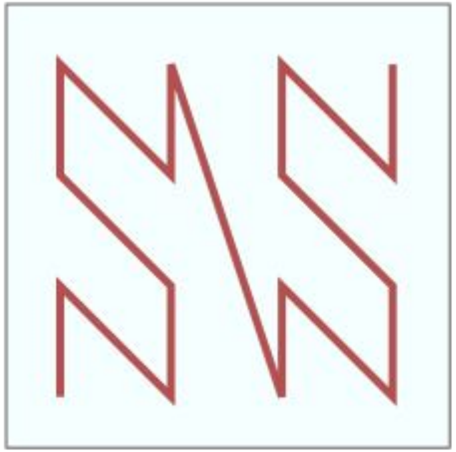
B-trees for correlated columns

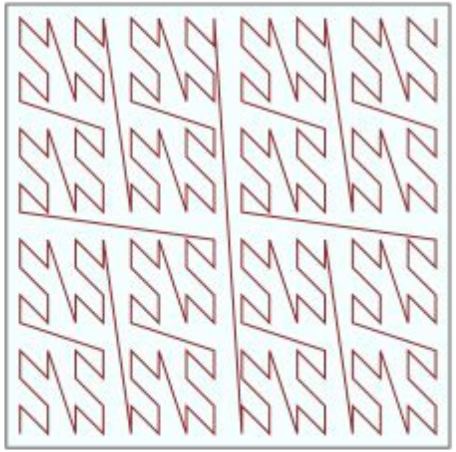
- e.g., TPC-H lineitem ship date, receipt date, commit date
  dates & times often correlated in business intelligence
- In a b-tree index on “ship date” …
  … add min+max information on “receipt date” and “commit date”
  ⇒ reasonably efficient search on any date column
- … add min+max of “transit days” and “spare days”
  ⇒ opportunities for compression with variable-size values

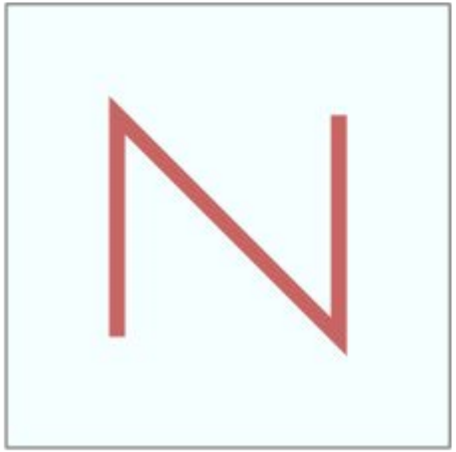
B-trees on hash value and on Z-values

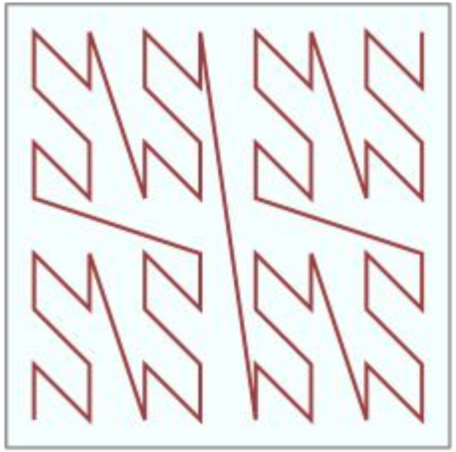
1. 0 dimensions ⇒ b-tree on hash values
   index creation: sort on hash values
   index intersection etc.: merge join on hash values
   concurrency control: phantom protection on gaps between hash values
2. spatial, temporal, spatio-temporal ⇒ b-tree on space-filling curve
   Z-values: bit interleaving
   Peano, Hilbert, Moore, … curves

Z-curve: bit interleaving









Log-structured merge-forests

- 1996/97: in-memory $\Rightarrow$ fast, on-disk $\Rightarrow$ slow: hybrid!
- Streams or bursts of insertions $\Leftarrow$
  - Update $\rightarrow$ insertion of replacement record
  - Deletion $\rightarrow$ insertion of “anti-matter” record
  - Range deletion $\rightarrow$ special anti-matter

Log-structured merge-forests: alternative views

- “Repair” b-trees by spill+restart $\Rightarrow$ binary merge steps
- “External merge sort” of b-tree entries $\Rightarrow$ bursts of wide merge steps
- “Never-ending sort” $\Rightarrow$ space budget per merge level
- Adaptive merging $\Rightarrow$ enforce side effects of queries

LSM-forest as “write-optimized store”

- C-store/Vertica (HP, HPE, Microfocus)
  read-optimized column store (no indexes)
  write-optimized row store (in-memory index?)
- When sorting and merging for maximal compression, what is the extra cost (CPU, storage, data movement) for b-tree branch nodes?
- What is the achievable benefit?

Differences…

	External merge sort	LSM-forest
Minimal, initial unit	In-memory run	In-memory b-tree
On-storage format	Sorted run file	B-tree
Efficient access	Scan, merge	Search, merge
Merge fan-in F	Memory / page size	Merge efficient vs query performance
Merge logic, depth	Tournament tree, $\log_F(I/M)$	
Merge schedule	Bottom-up, after run generation	As needed, any level any time

Partitioning

- Orthogonal to “partitions” (runs) in LSM-forest
- Focus on parallelism & manageability
- Partitioning per…
  - … table (or multi-version) $\Rightarrow$ “local” indexes $\Rightarrow$ local index-to-index navigation, index intersection, etc. $\Rightarrow$ little communication, high parallelism $\Rightarrow$ good response time, bad resource consumption
  - … index $\Rightarrow$ “global” indexes $\Rightarrow$ …
- Co-partitioning on PKs and FKs $\Rightarrow$ efficient local joins

Partitioning

- Round-robin & random partitioning
- Range-partitioning $\leftrightarrow$ distribution skew
- Hash-partitioning $\leftrightarrow$ duplicate skew
- Hybrid partitioning: ranges of hash values
- Hybrid partitioning: hashing small key ranges
  - + distribution skew, + duplicate skew
  - Small query range $\Rightarrow$ few parallel scans, low latency
  - Large query range $\Rightarrow$ many scans, high bandwidth

Free-space management

- Bitmaps with high concurrency
  - Per device or file (guides “backup” logic)
  - Per index (or partition) (guides “drop” logic)
- Manipulated in system transactions only
  - Model a b-tree with uncompressed bitmaps?
- Priority in restart and restore

Summary

- Devices, memory & storage hierarchy
- B-trees for row storage and column storage
- Columns for scans (compression, parallelism, zone filters)
  $\Rightarrow$ OLAP
- Rows for index-to-index, record-to-record navigation
  $\Rightarrow$ OLTP