

Physical Data Organization and Indexing

Introduction to Databases
CompSci 316 Fall 2017



Announcements (Mon., Mar. 6)

- **Homework #3** to be posted today
 - will be updated after each lecture

Today:

- Finish physical organization
- Indexes

Recall: cost for DB = mostly I/O

- Reading from/writing to disk is a major source of cost
- It's all about reducing I/O's!
- Cache blocks from stable storage in memory
 - DBMS maintains a memory **buffer pool** of blocks
 - Reads/writes operate on these memory blocks
 - Dirty (updated) memory blocks are "flushed" back to stable storage
- Sequential I/O is much faster than random I/O
 - try to store records that are likely to be accessed together close to each other

Recall: different storage layouts

- Record layouts
 - how attributes are stored in a record
- Block layout
 - how records are stored in a block
 - block = unit of I/O
 - sometimes unit of I/O in terms of a "page", and a block can contain multiple pages
 - basic idea remains the same

Recall: Record layout

- **Record** = row or tuple in a table
 - "fixed format" dictated by table schema in relational databases
- Fixed-Length Records
- Variable-Length Records

Fixed-length fields

- All field lengths and offsets are constant
 - Computed from schema, stored in the system catalog
- Common to start fields at locations multiple of 4 or 8
- Often record starts with a header
 - pointer to the schema
 - length of the record
 - timestamp for last read/write
 - pointers to the fields

Example: `CREATE TABLE User(uid INT, name CHAR(20), age INT, pop FLOAT);`

0	4	24	28	36
142	Bart (padded with space)		10	0.9

Block layout for fixed length records

- Header may contain
 - links to some other "related" blocks, e.g. from overflow in indexes
 - information about the relation the block belongs to
 - directory for offset of each record
 - timestamp for last read/write
 - etc.

header	record1	record2	free space
--------	---------	---------	------	------------

Variable Length Records - motivation

- Data size may vary
 - address (up to 255 bytes, typically < 50 bytes), name
 - waste of space in fixed length
- Repeating fields
 - e.g. pointers for a many-many relationship
 - the number of references may vary
- Variable format records
 - do not know at the beginning (XML)
- Enormous fields
 - like videos
 - recall BLOBs from Lecture 13

Variable-length records

- Put all variable-length fields at the end after all fixed length fields (why?)
- Example: `CREATE TABLE User(uid INT, name VARCHAR(20), age INT, pop FLOAT, comment VARCHAR(100));`
- Approach 1: use field delimiters ("0' okay?)

0	4	8	16		
142	10	0.9	Bart 0	Weird kid !0	

- Approach 2: use an offset array

0	4	8	16	18	22	32
142	10	0.9	Bart	Weird kid!		

why no pointer to Bart?

- Pros/cons of approach 2?

Specific approaches

- NSM:
 - N-ary storage model
 - Standard row-major order
- PAX:
 - Partition Attributes Across
 - (if you are interested, see this: <http://www.pdl.cmu.edu/PDL-FTP/Database/pax.pdf>)
- Column store
 - Store records in column-major order

NSM

- Store records from the beginning of each block
- Use a directory at the end of each block
 - To locate records and manage free space
 - Necessary for variable-length records

Why store data and directory at two different ends?

So both can grow easily!

Options

- Reorganize after every update/delete to avoid fragmentation (gaps between records)
 - Need to rewrite half of the block on average
- A special case: What if records are fixed-length?
 - Option 1: reorganize after delete
 - Only need to move one record
 - Need a pointer to the beginning of free space
 - Option 2: do not reorganize after update
 - Need a bitmap indicating which slots are in use

Cache behavior of NSM

- Query: `SELECT uid FROM User WHERE pop > 0.8;`
- Assumptions: no index, and cache line size < record size
- Lots of cache misses
 - loads unnecessary attributes
 - uid and pop are not close enough by memory standards

142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3

142	Bart	10	
0.9	123	Milhouse	
10	0.2	857	Lisa
8	0.7		
456	Ralph	8	
0.3			

Cache

PAX

- Most queries only access a few columns
- Cluster values of the same columns in each block
 - When a particular column of a row is brought into the cache, the same column of the next row is brought in together

Reorganize after every update (for variable-length records only) and delete to keep fields together

(IS NOT NULL bitmap)

Beyond block layout: column stores

- The other extreme: store tables by columns instead of rows
- PAX affects data layout within a single page
 - e.g. one relation can store NSM, other PAX
 - or first do vertical partitioning, then use PAX for storing
- Advantages (and disadvantages) of PAX are magnified
 - Not only better cache performance, but also fewer I/O's for queries involving many rows but few columns
 - Aggressive compression to further reduce I/O's
- More disruptive changes to the DBMS architecture are required than PAX
 - Not only storage, but also query execution and optimization

Summary

- Storage hierarchy
 - Why I/O's dominate the cost of database operations
- Disk
 - Steps in completing a disk access
 - Sequential versus random accesses
- Record layout
 - Handling variable-length fields
 - Handling NULL
 - Handling modifications
- Block layout
 - NSM: the traditional layout (row store)
 - PAX: a layout that tries to improve cache performance
- Column store: NSM transposed, beyond blocks

Index

What are indexes for?

- Given a value, locate the record(s) with this value
`SELECT * FROM R WHERE A = value;`
`SELECT * FROM R, S WHERE R.A = S.B;`
- Find data by other search criteria, e.g.
 - Range search
`SELECT * FROM R WHERE A > value;`
 - Keyword search

} Focus of this lecture

database indexing Search

Index classification

- Dense vs. Sparse
- Clustered vs. unclustered
- Primary vs. Secondary
- Tree-based vs. Hash-based
 - we will only do tree indexes in 316

• Discussion on structure of indexes and pages : on whiteboard

Dense and sparse indexes

- Dense:** one index entry for each search key value
 - One entry may "point" to multiple records (e.g., two users named Jessica)
- Sparse:** one index entry for each block
 - Records must be **clustered** according to the search key

123	Milhouse	10	0.2
142	Bart	10	0.9
279	Jessica	10	0.9
345	Martin	8	2.3
456	Ralph	8	0.3
512	Nelson	10	0.4
679	Sherri	10	0.6
697	Terri	10	0.6
857	Lisa	8	0.7
912	Windel	8	0.5
997	Jessica	8	0.5

Dense versus sparse indexes

- Index size
 - Sparse index is smaller
- Requirement on records
 - Records must be clustered for sparse index
- Lookup
 - Sparse index is smaller and may fit in memory
 - Dense index can directly tell if a record exists
- Update
 - Easier for sparse index

Primary and secondary indexes

- Primary index**
 - Created for the **primary key** of a table
 - Records are usually clustered by the primary key
 - Can be sparse
- Secondary index**
 - Usually dense
- SQL
 - PRIMARY KEY declaration automatically creates a primary index, UNIQUE key automatically creates a secondary index
 - Additional secondary index can be created on non-key attribute(s):
`CREATE INDEX UserPopIndex ON User(pop);`

ISAM

- What if an index is still too big?
 - Put a another (sparse) index on top of that!
 ☞ **ISAM (Index Sequential Access Method)**, more or less

Example: look up 197

Updates with ISAM

Example: insert 107
Example: delete 129

Index blocks: 100, 200, ..., 901

Data blocks: 100, 108, 119, 121; 123, 129; 192, 197; 200, 202; 901, 907; 996, 997

107 Overflow block

- Overflow chains and empty data blocks degrade performance
 - Worst case: most records go into one long chain, so lookups require scanning all data!

B⁺-tree

- A hierarchy of nodes with intervals
- **Balanced** (more or less): good performance guarantee
- **Disk-based**: one node per block; large fan-out

Max fan-out: 4

Sample B⁺-tree nodes

to keys $100 \leq k$

Non-leaf: 120, 150, 180 (Max fan-out: 4)

to keys $100 \leq k < 120$ to keys $120 \leq k < 150$ to keys $150 \leq k < 180$ to keys $180 \leq k$

Leaf: 120, 130 (to next leaf node in sequence)

to records with these k values; or, store records directly in leaves

B⁺-tree balancing properties

- Height constraint: all leaves at the same lowest level
- Fan-out constraint: all nodes at least half full (except root)

	Max # pointers	Max # keys	Min # active pointers	Min # keys
Non-leaf	f	$f - 1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil - 1$
Root	f	$f - 1$	2	1
Leaf	f	$f - 1$	$\lfloor f/2 \rfloor$	$\lfloor f/2 \rfloor$

Lookups

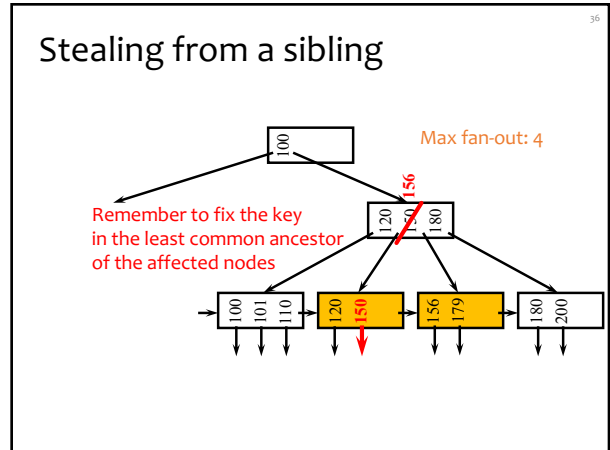
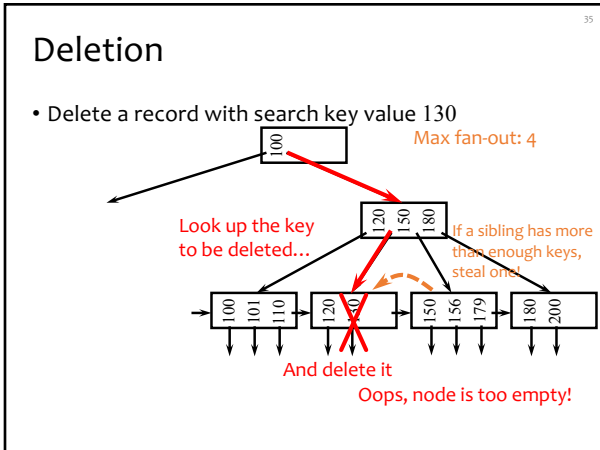
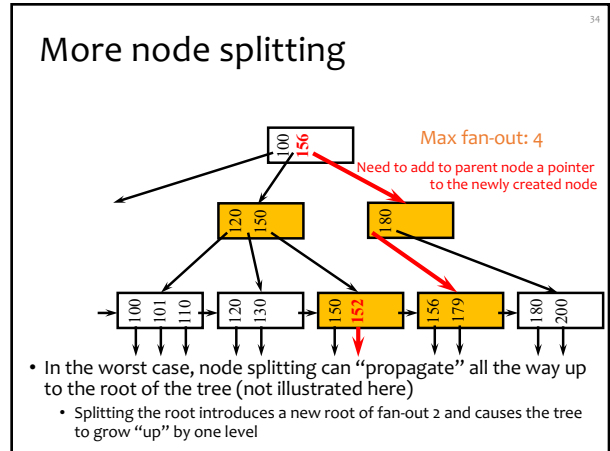
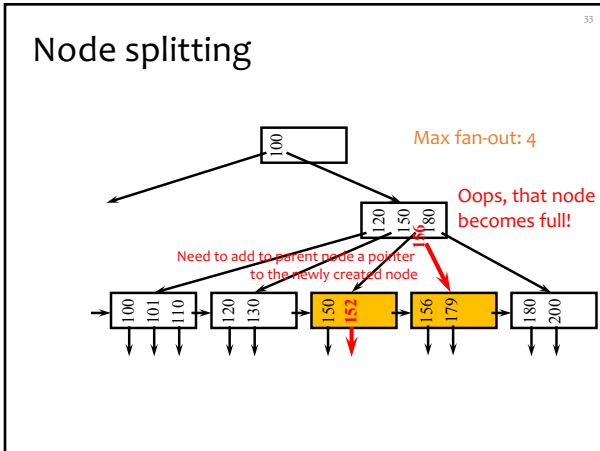
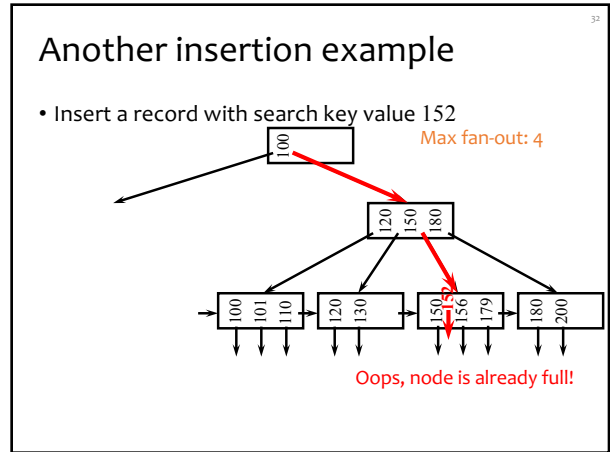
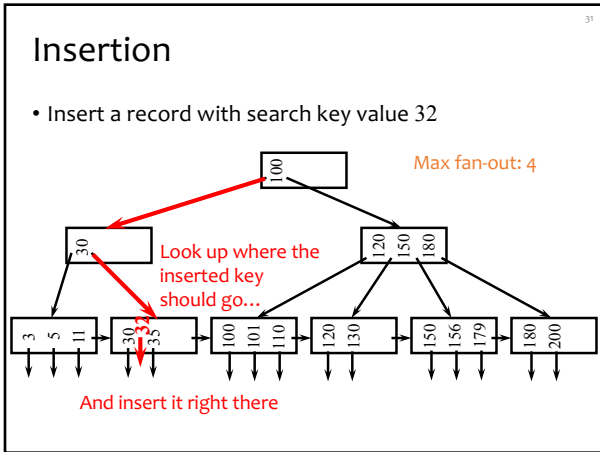
- SELECT * FROM R WHERE $k = 179$;
- SELECT * FROM R WHERE $k = 32$;

Max fan-out: 4

Range query

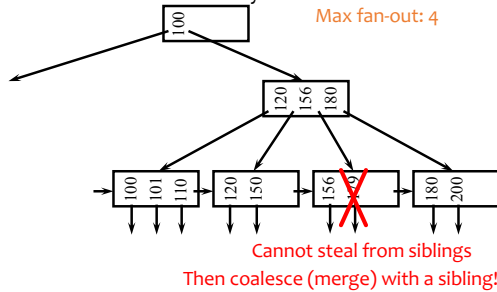
- SELECT * FROM R WHERE $k > 32$ AND $k < 179$;

Max fan-out: 4

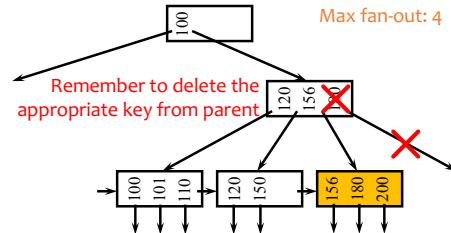


Another deletion example

- Delete a record with search key value 179



Coalescing



- Deletion can “propagate” all the way up to the root of the tree (not illustrated here)
 - When the root becomes empty, the tree “shrinks” by one level

Performance analysis

- How many I/O's are required for each operation?
 - h , the height of the tree (more or less)
 - Plus one or two to manipulate actual records
 - Plus $O(h)$ for reorganization (rare if f is large)
 - Minus one if we cache the root in memory
- How big is h ?
 - Roughly $\log_{fanout} N$, where N is the number of records
 - B⁺-tree properties guarantee that fan-out is least $f/2$ for all non-root nodes
 - Fan-out is typically large (in hundreds)—many keys and pointers can fit into one block
 - A 4-level B⁺-tree is enough for “typical” tables

B⁺-tree in practice

- Complex reorganization for deletion often is not implemented (e.g., Oracle)
 - Leave nodes less than half full and periodically reorganize
- Most commercial DBMS use B⁺-tree instead of hashing-based indexes because B⁺-tree handles range queries

The Halloween Problem

- Story from the early days of System R...

```
UPDATE Payroll
SET salary = salary * 1.1
WHERE salary <= 25000;
```

- There is a B⁺-tree index on *Payroll(salary)*
- The update never stopped (why?)
- Solutions?
 - Scan index in reverse, or
 - Before update, scan index to create a “to-do” list, or
 - During update, maintain a “done” list, or
 - Tag every row with transaction/statement id

B⁺-tree versus ISAM

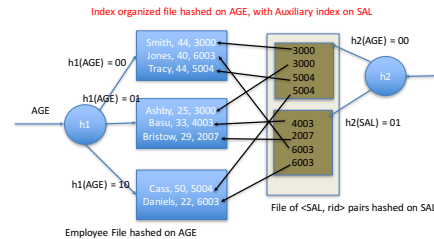
- ISAM is more static; B⁺-tree is more dynamic
- ISAM can be more compact (at least initially)
 - Fewer levels and I/O's than B⁺-tree
- Overtime, ISAM may not be balanced
 - Cannot provide guaranteed performance as B⁺-tree does

B⁺-tree versus B-tree

- B-tree: why not store records (or record pointers) in non-leaf nodes?
 - These records can be accessed with fewer I/O's
- Problems?
 - Storing more data in a node decreases fan-out and increases h
 - Records in leaves require more I/O's to access
 - Vast majority of the records live in leaves!

B+ tree vs. Hash-based indexes

- Extensible hashing, linear hashing, etc.
- **Can only handle “=” in join or selection**
- **Cannot handle inequality predicates**



Beyond ISAM, B-, and B⁺-trees, and hash

- Other tree-based indexes: R-trees and variants, GiST, etc.
 - How about binary tree?



- Text indexes: inverted-list index, suffix arrays, etc.
- Other tricks: bitmap index, bit-sliced index, etc.