

Database System Architecture and Implementation

Module 0
Introduction and Overview
October 21, 2013

Module Overview

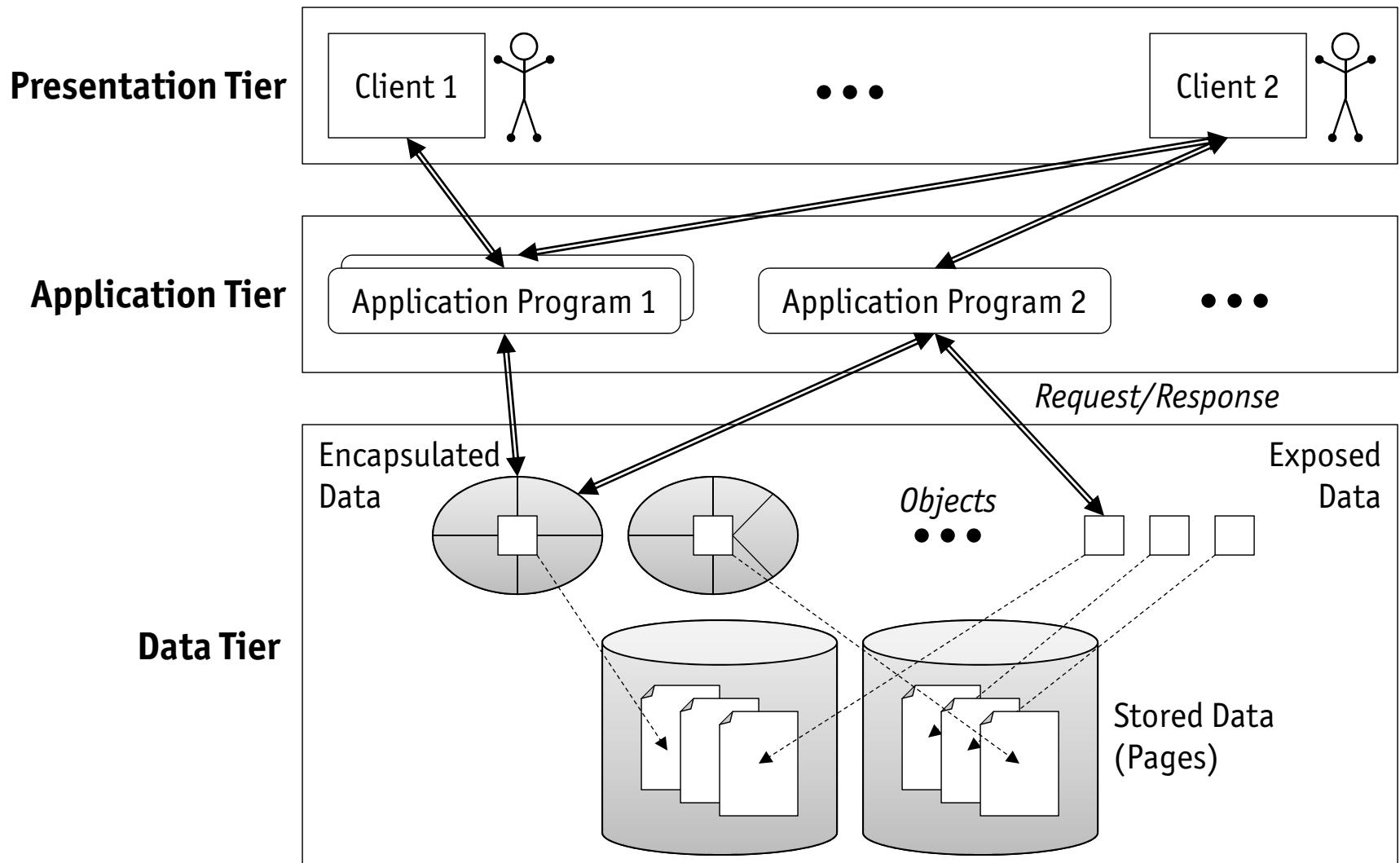
1. Course contents

1. Goals of this course
2. DBMS architecture overview
3. Layers of DBMS architecture

2. Course organization

1. Planned schedule
2. Personnel
3. Course resources
4. Exercises
5. Exam

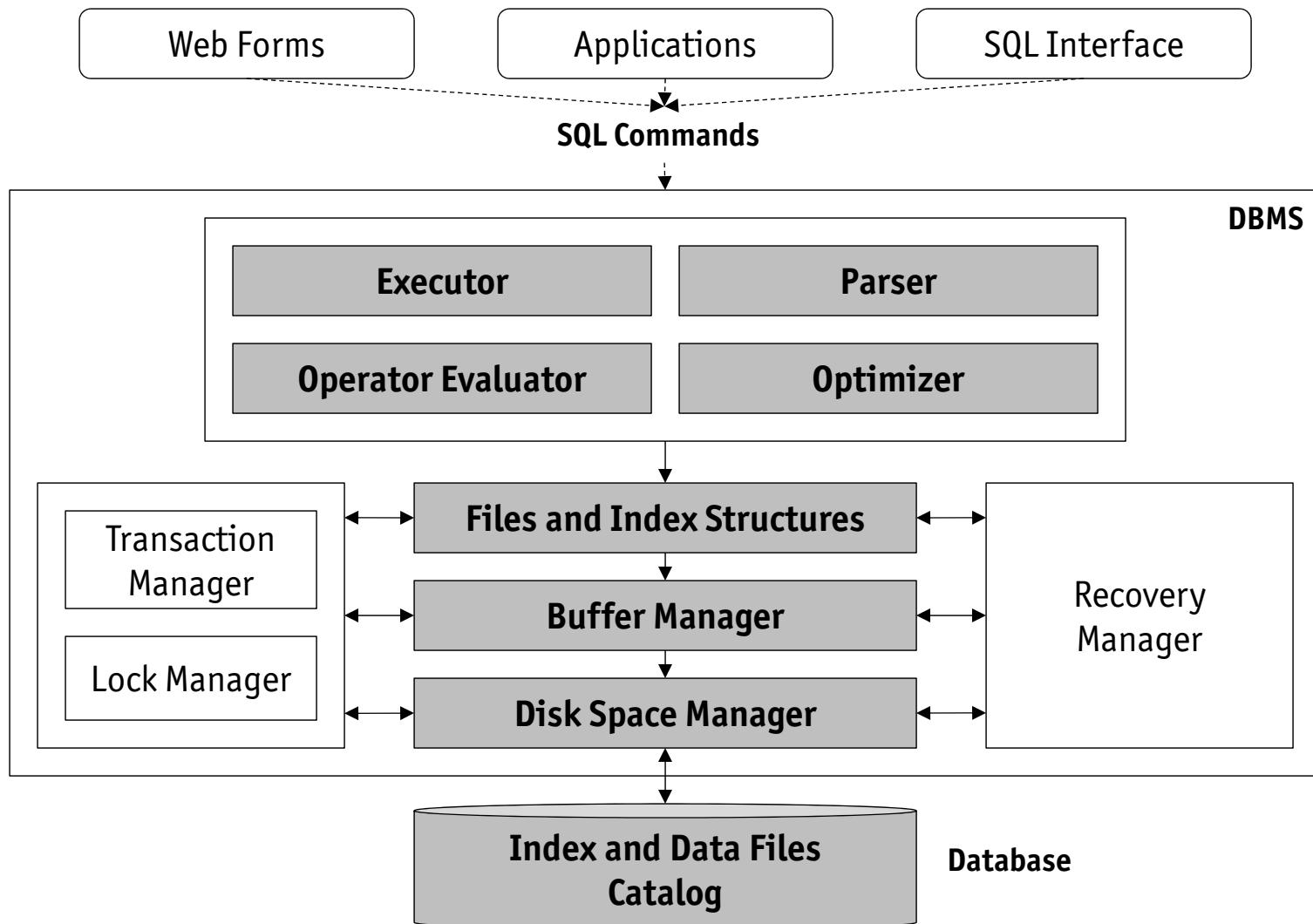
The Big Picture



Course Contents and Goals

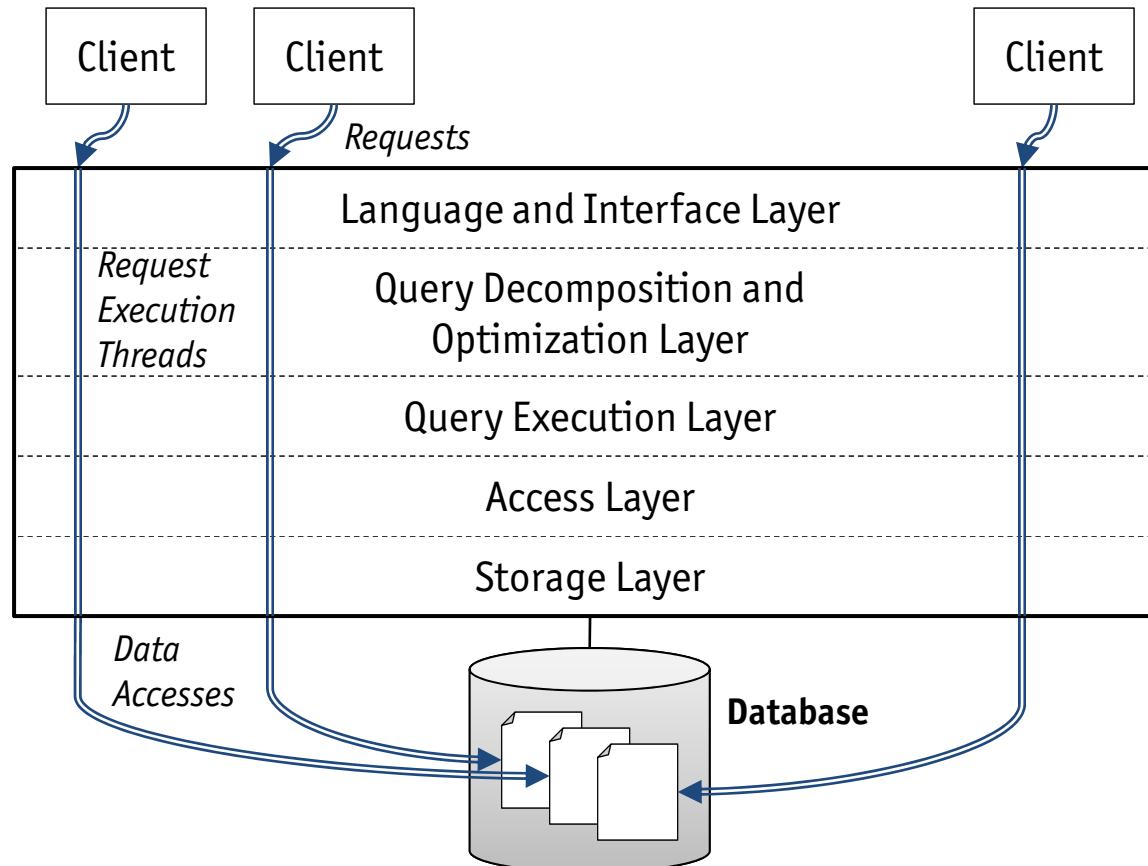
- Builds on introductory database course with focus on internals at system-level, rather than functionality at interface-level
- Learn how a DBMS can...
 - **organize and access files** on hard disks to minimize costly I/O traffic
 - **translate SQL statements** into efficient query execution plans
 - **sort/combine/filter data volumes** that exceed main memory size by far
 - be **tuned** for performance-critical applications
 - be evaluated by using **benchmarks**

DBMS Architecture



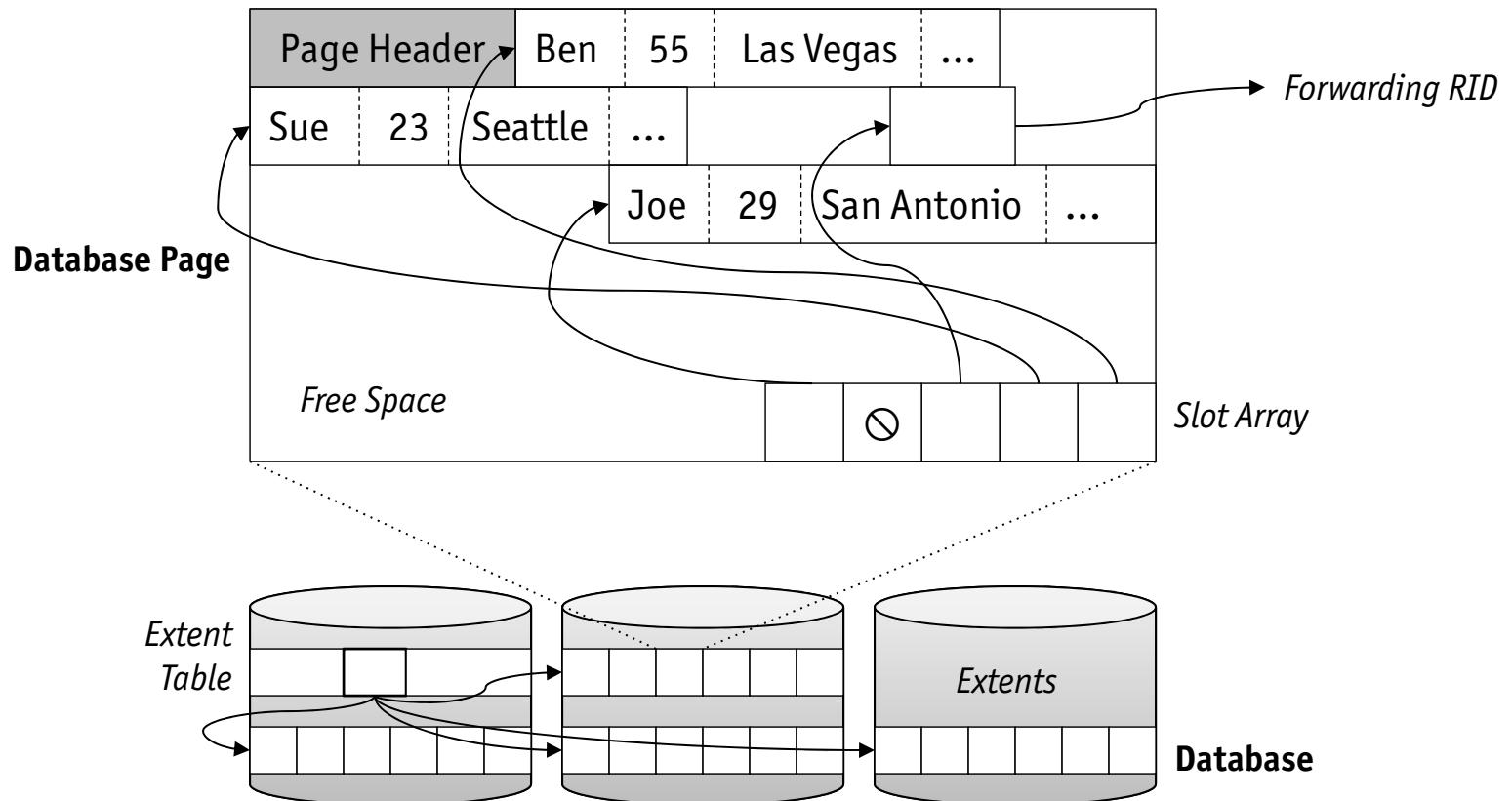
Layered DBMS Architecture

- DBMS implements its functionality in a layered architecture
 - incrementally add more abstractions with each layer
 - from **low-level** block I/O devices to **high-level** declarative user interface



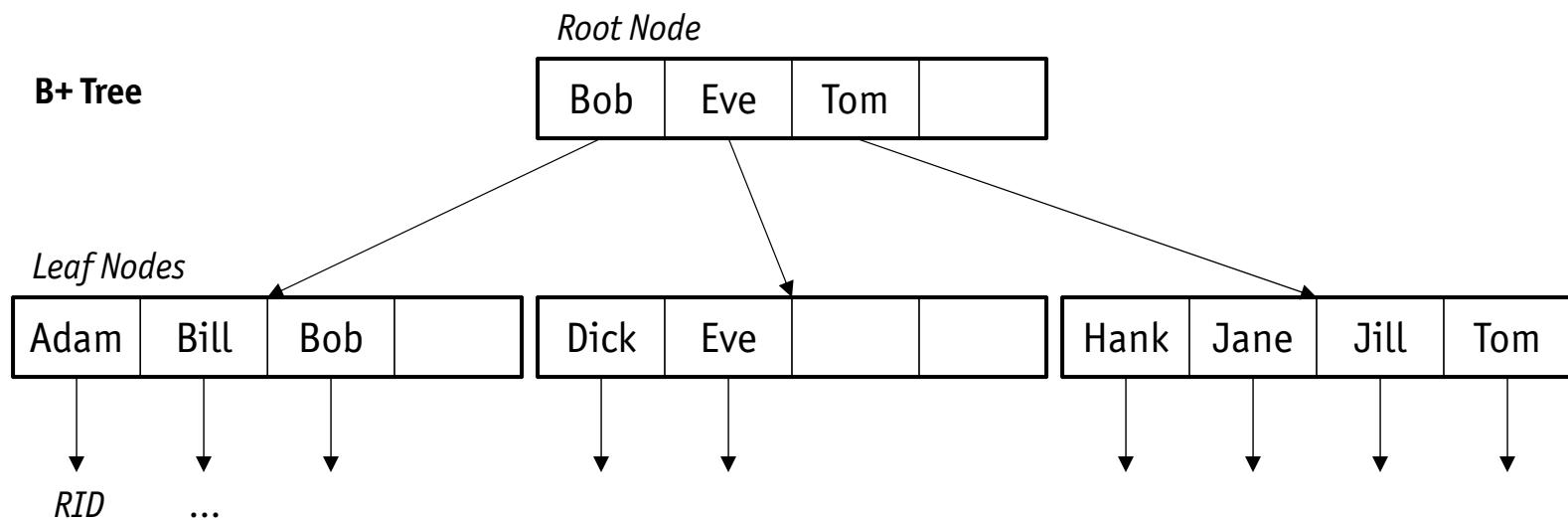
Storage Structures

- DBMS data structures are mapped to fixed-length blocks
 - basic I/O **unit of transfer** between main and secondary memory
 - true for **all** types of DBMS, i.e., relational, object-relational, etc.



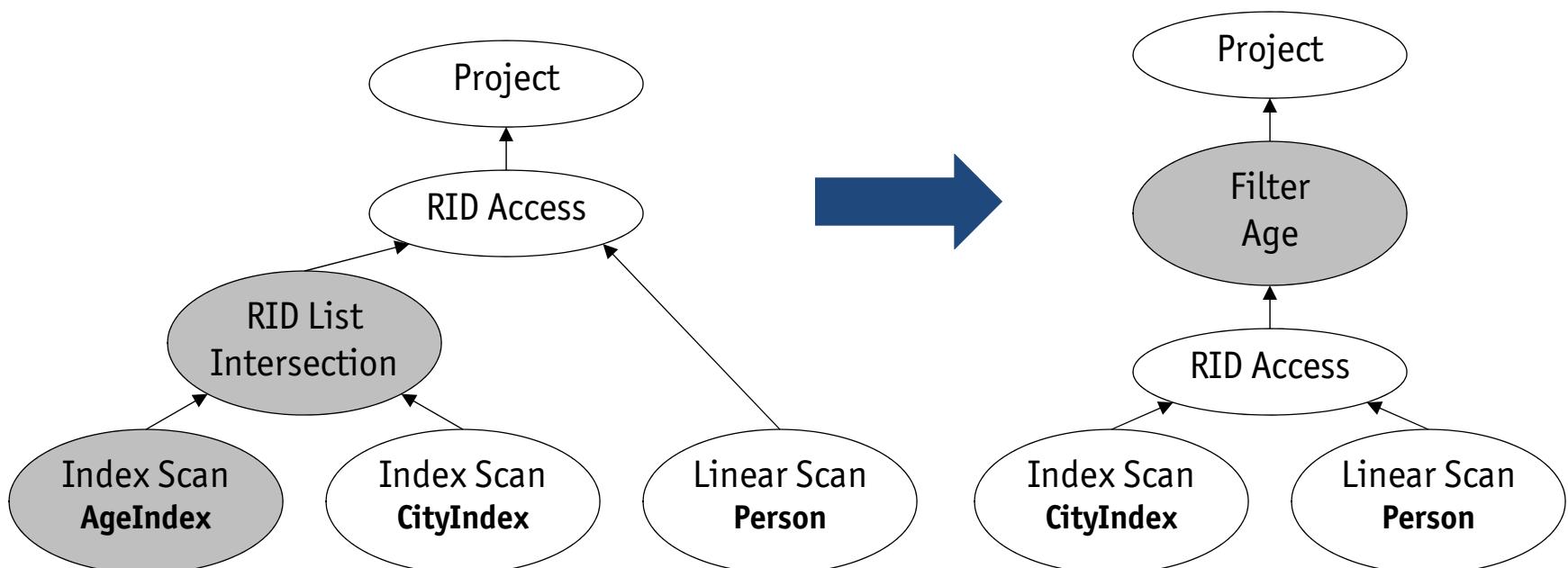
Access Paths

- DBMS indexing techniques enable fast content-based access
 - **tree-structured** and **hash-based** methods
 - **application-specific** methods



Query Processing

- DBMS processes queries and returns results
 - **declarative query specification**, e.g., expressed in SQL, is optimized and transformed into efficient **query execution plan** (QEP)
 - QEP is a **sequential** or **parallel** program that computes the query results



Planned Course Schedule

Week	Date	Topic	Exercise	Due
1	21.10.2013	Introduction	Disk Storage	28.10.2013
2	28.10.2013	File Organization and Indexing	Disk Management	4.11.2013
3	4.11.2013	Disks and Files	Buffer Management	18.11.2013
4	11.11.2013	Tree-Structured Indexing		
5	18.11.2013	Hash-Based Indexing	B+ Trees	25.11.2013
6	25.11.2013	Query Evaluation	Disk-based B+ Tree	9.12.2013
7	2.12.2013	External Sorting		
8	9.12.2013	Evaluating Relational Operators	Hashing	16.12.2013
9	16.12.2013	Query Optimization	External Sorting	6.1.2014
Christmas Break				
10	6.1.2014	Selectivity and Cost Estimation	Relational Operators	20.1.2014
11	13.1.2014	Nested and Recursive Queries		
12	20.1.2014	Interesting Orders	Cost and Estimations	27.1.2014
13	27.1.2014	Physical Database Design	Query Optimization	3.2.2014
14	3.2.2014	Database Tuning	Flashback	10.2.2014
15	10.2.2014	Benchmarking		

Personnel

- Jun.-Prof. Dr. Michael Grossniklaus

- Office E 207
 - E-mail michael.grossniklaus@uni-konstanz.de
 - Phone 4434
 - Office hours by appointment



- Andreas Weiler

- Office E 209
 - E-mail andreas.weiler@uni-konstanz.de
 - Phone 4449
 - Office hours always



About Myself

- Curriculum vitae
 - MSc and PhD (ETH Zurich, Switzerland)
 - Post-doc (Politecnico di Milano, Italy)
 - Post-doc (Portland State University, US)
 - Juniorprofessor (University of Konstanz, Germany)
- Areas of interest
 - graph data management and processing
 - object databases
 - data stream management systems
 - web engineering
- Previous work
 - context-aware data management
 - search computing

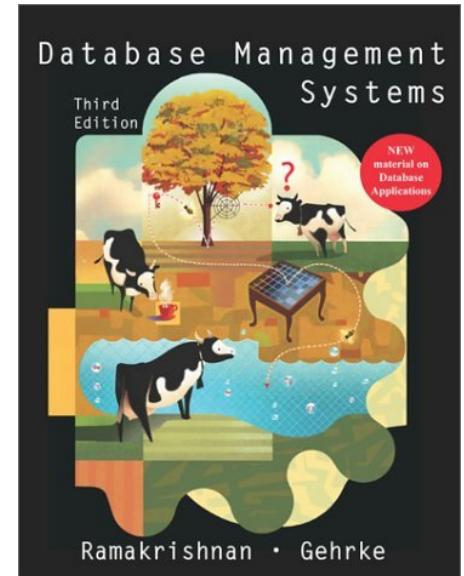


Course Resources

- Web site
 - <http://www.informatik.uni-konstanz.de/grossniklaus/education/dbmsinternals/>
- Registration
 - **LSF** for course-related e-mail
 - **StudIS** for admittance to the exam
- Literature
 - lectures slides can be downloaded from the course website
 - textbook (see next slide)
 - further readings will be published on the course website

Textbook

- *Raghu Ramakrishnan and Johannes Gehrke*
Database Management Systems (3rd Edition)
McGraw-Hill, 2002
 - course will focus on chapters 8 to 15 and 20
- Ten copies of the **2nd edition** of the book
are available at the university library
 - not much difference for the scope of this course
 - course will focus on chapters 7 to 14 and 16



Prerequisites

- The following skills are **mandatory** prerequisites to attend and successfully complete this course
- **Foundations of database systems** (INF-12040 or equivalent)
 - E/R model and conceptual design, relational model, relational algebra and calculus, query languages, database application programming, etc.
- **Computer systems** (INF-11740, INF-11880, or equivalent)
 - computer architecture, operating systems, compilers, networks, etc.
- **System programming** (INF-11930 or equivalent)
 - some of the exercises are based on a (somewhat) complex Java project
 - students **must have the ability to program in Java (read, understand, design, and write high-quality Java code)**
- **Key competences** (INF-10175 or equivalent)
 - Subversion, LaTeX, etc.

Rooms and Dates

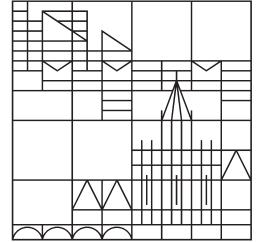
- Lecture
 - Monday 3:15 pm – 4:45 pm Room E 402
 - Wednesday 10:00 am – 10:45 pm Room R 512
- Exercise
 - Wednesday 10:45 am – 11:30 pm Room R 512

Exercises

- Assignments will be made available on the course website
 - solved in pairs of two students
- Electronic submission via e-mail only
 - **study questions:** PDF files only (no plain text, hand-written, Word, ...)
 - **programming exercises:** ZIP files that only contains relevant files
- Four two-week programming projects
 - based on Minibase for Java
 - opportunity to implement core components of a DBMS yourself
- Grading
 - one-week assignments give 20 points
 - two-week assignments give 40 points
 - 50% of total course grade

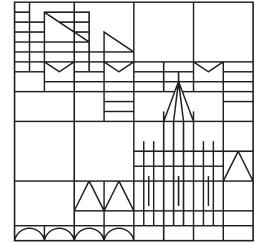
Exam

- Written exam
 - two-hour exam
 - closed-book exam, but one hand-written double-sided piece of A4 paper with notes is permitted
- Content
 - lecture and lecture slides
 - (written) exercises
 - relevant chapters from textbook
- Scheduling
 - first date February 20, 2014
 - second date April 16, 2014
- Grading
 - 50% of total course grade



Database System Architecture and Implementation

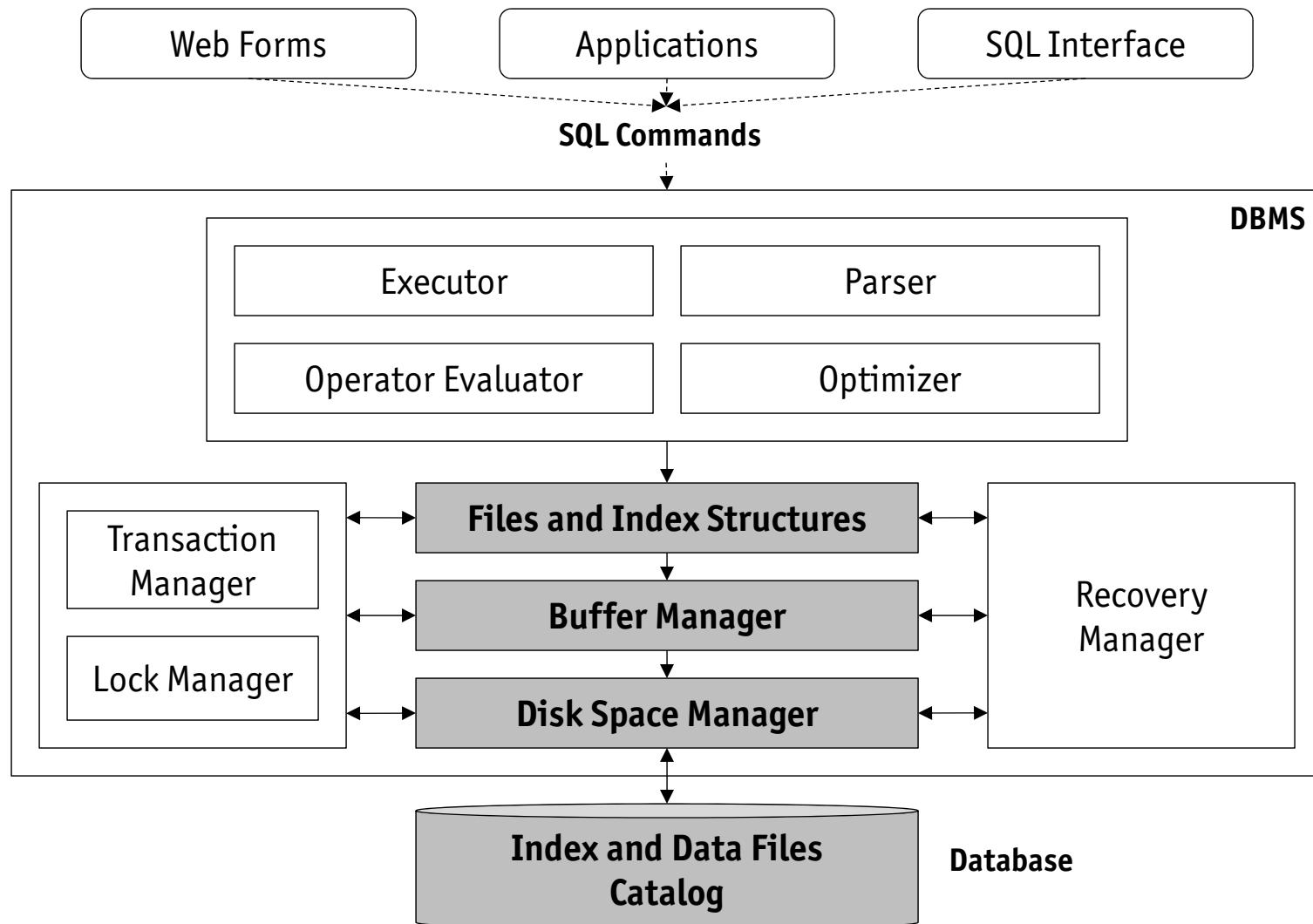
TO BE CONTINUED...



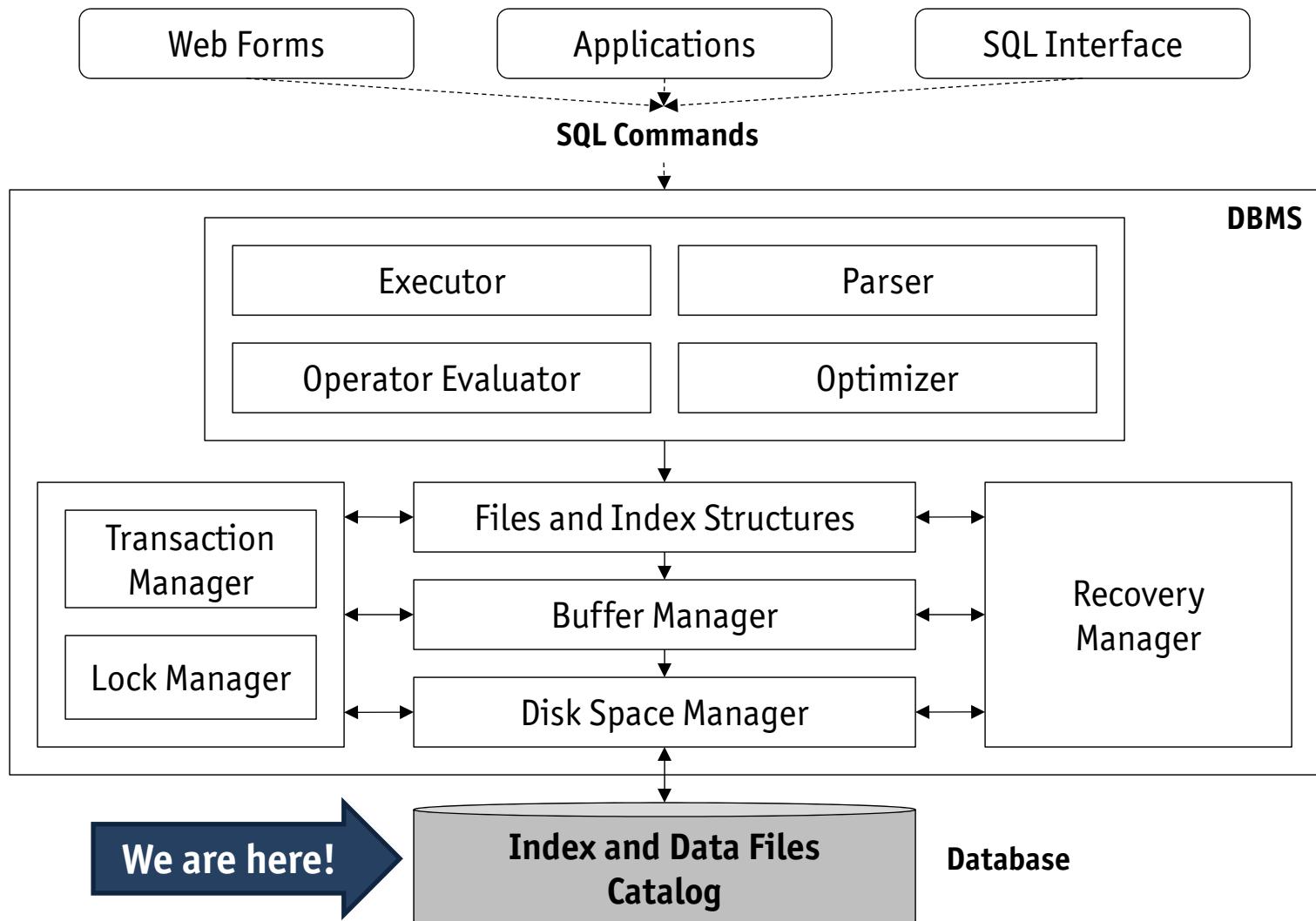
Database System Architecture and Implementation

Module 1
Storing Data: Disks and Files
October 21, 2013

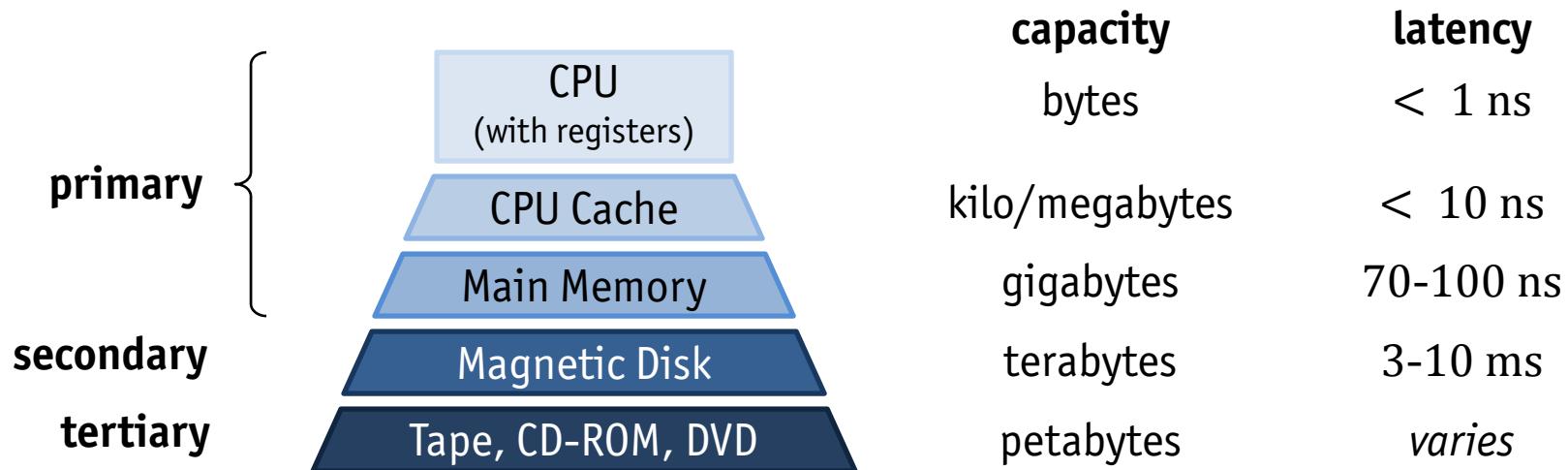
Module Overview



Orientation

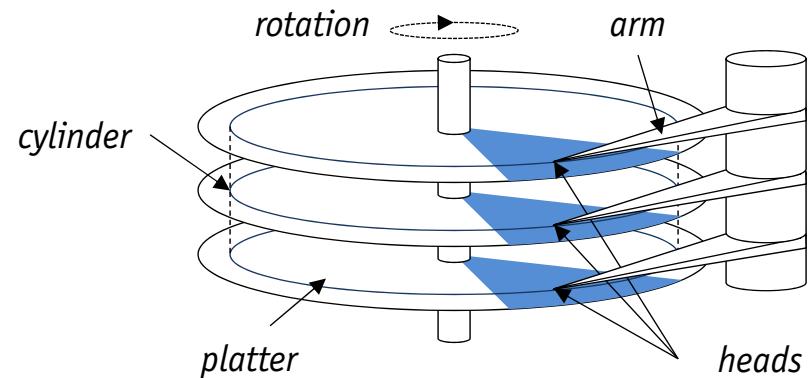
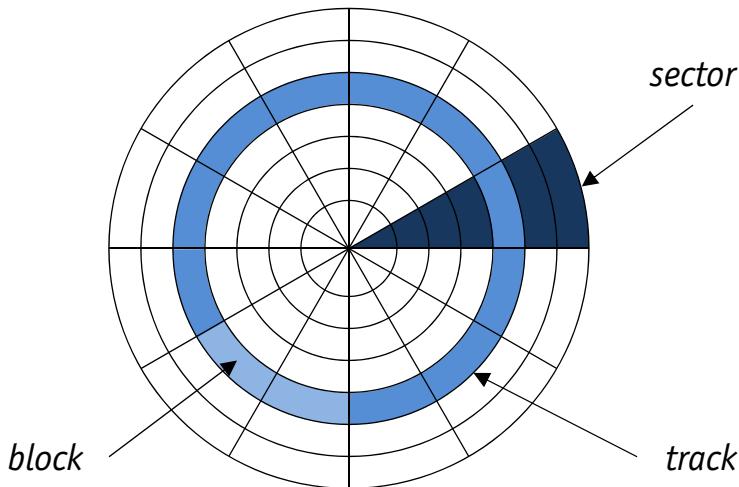


Memory Hierarchy



- Memory in off-the-shelf computers is organized in a hierarchy
 - **cost** of primary memory $\approx 100 \times$ cost of secondary storage of same size
 - **size** of address space in primary memory usually not large enough to map an entire database
- Only data stored in **non-volatile**, i.e., secondary, tertiary, storage persists across system shutdowns and crashes
- Goal is to **hide latency** by using the fast memory as **cache**

Magnetic Disks



- Data is arranged in concentric rings (**tracks**) on **platters** (one or two-sided)
- Tracks are divided into arc-shaped **sectors** (hardware characteristic)
- A **cylinder** is the set of all tracks with the same diameter
- A stepper motor moves an array of disk **arms** and **heads** from track to track as the disks steadily **rotate**
- Data is read from and written to disk one **block** at a time, which can be set to a multiple of sector size when formatting the disk, e.g., 4 kB or 8 kB

Access Time

- Data blocks can only be read and written if disk heads and platters are positioned accordingly
 - this design has implications on the **access time** required to read or write a given block
 - time to read or write data varies, depending on the location of the data
- **Definition:** access time is the sum $t = t_s + t_r + t_t$, where
 - **seek time** (t_s): disk heads have to be moved to desired track
 - **rotational delay** (t_r): disk controller has to wait for the desired block to rotate under disk head
 - **transfer time** (t_t): disk block data has to be written or read

Example

- Seagate Cheetah 15K.7
 - 4 disks, 8 heads, 512 kB/track, 600 GB capacity
 - rotational speed 15,000 RPM (revolutions per minute)
 - average seek time 3.4 ms
 - transfer rate \approx 163 MB/s



What is the access time to read an 8 kB block?

Example

- Seagate Cheetah 15K.7
 - 4 disks, 8 heads, 512 kB/track, 600 GB capacity
 - rotational speed 15,000 RPM (revolutions per minute)
 - average seek time 3.4 ms
 - transfer rate $\approx 163 \text{ MB/s}$

✍ What is the access time to read an 8 kB block?

- average seek time $t_s = 3.40 \text{ ms}$
- average rotational delay ($\frac{1}{2} \times \frac{1}{15,000 \text{ min}^{-1}}$) $t_r = 2.00 \text{ ms}$
- transfer time for 8 kB ($\frac{8 \text{ kB}}{163 \text{ MB/s}}$) $t_t = 0.05 \text{ ms}$
- total **access time** for an 8 kB data block $t = 5.45 \text{ ms}$

↳ Recall that accessing a main memory location typically takes $\approx 70\text{-}100 \text{ ns}$!

Sequential vs. Random Access

Example: Read 1,000 blocks of size 8 kB

- The Seagate Cheetah 15K.7 stores an average of 512 kB per track, with a 0.2 ms track-to-track seek time. Hence, our 8 kB blocks are spread across 16 tracks.
- **Random access**

$$t_{rnd} = 1,000 \times 5.45 \text{ ms} = \mathbf{5.45 \text{ s}}$$

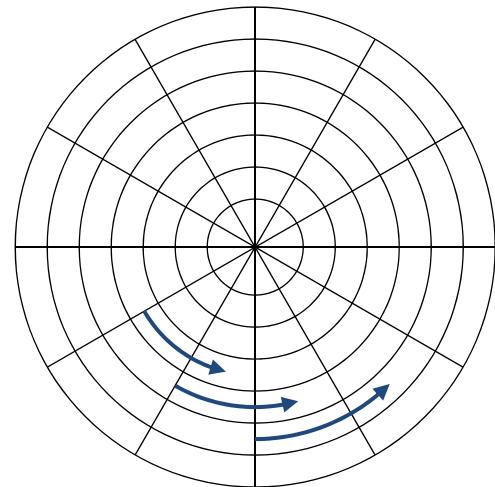
- **Sequential read of adjacent blocks**

$$\begin{aligned} t_{seq} &= t_s + t_r + 1,000 \times t_t + 16 \times t_{s,\text{track-to-track}} \\ &= 3.40 \text{ ms} + 2.00 \text{ ms} + 50 \text{ ms} + 3.2 \text{ ms} \approx \mathbf{58.6 \text{ ms}} \end{aligned}$$

- Sequential access is **much** faster than random access
- **Avoid** random access whenever possible
- As soon as we need at least $\frac{58.6 \text{ ms}}{5,450 \text{ ms}} = 1.07\%$ of a file, we better read the **entire** file sequentially

Performance Tricks

- Manufacturers use a number of tricks to improve performance
 - **track skewing:** align sector 0 of each track to avoid rotational delay using longer sequential scans
 - **request scheduling:** if multiple requests have to be served, chose the one that requires the smallest arm movement (SPTF: shortest positioning time first and elevator algorithms)
 - **zoning:** outer tracks are longer than inner ones and therefore can be divided into more sectors



Evolution of Hard Disk Technology

- Disks are a potential bottleneck for system performance
 - performance of CPU has improved $\approx 50\%$ per year
 - disk seek and rotational latencies have only marginally improved over the last years ($\approx 10\%$ per year)
- Sequential vs. random access ratio gets worse over time
 - throughput (i.e., transfer rates) improve by $\approx 50\%$ per year
 - hard disk capacity grows by $\approx 50\%$ every year

Example: Seagate Barracuda 7200.7 (5 years ago)

Read 1,000 blocks of 8 kB sequentially and randomly: 397 ms vs. 12,800 ms

Implications for DBMS

- Time for moving blocks to and from disk usually **dominates** time taken by a database operation
 - data must be **in memory** for DBMS to operate on it
 - a disk block is the **unit of data transfer** between disk and memory, which is called an I/O (input/output) operation
- DBMS takes geometry and mechanics of hard disk into account
 - transfer a whole track in one platter revolution
 - switch active disk head after each revolution
- This implies a **closeness measure** for data records r_1, r_2 on disk
 1. Place r_1 and r_2 in the same block (single I/O operation)
 2. Place r_2 inside a block adjacent to the block of r_1 on the **same track**
 3. Place r_2 in a block somewhere on the track of r_1
 4. Place r_2 in a track of the **same cylinder** as the track of r_1
 5. Place r_2 in a cylinder adjacent to the cylinder of r_1

Improving I/O Performance

1. Reduce **number** of I/O operations

- DBMS buffer: cache data in fast memory to hide latency
- physical database design: tables, indexes, ...

2. Reduce **duration** of I/O operations

- access neighboring disk blocks (**clustering**) and bulk I/O
- different I/O paths (**de-clustering**) with parallel access

Clustering vs. De-clustering

- Clustering
 - bulk I/O can be implemented on top of or inside disk controller
 - **advantages:** optimized seek time and rotational delay, minimized overhead (e.g., interrupt handling), no additional hardware needed
 - **disadvantage:** I/O path busy for a long time (\bullet^* concurrency)
 - **use case:** mid-sized data access (prefetching, sector buffering)
- De-clustering
 - advanced hardware or disk arrays (RAID systems)
 - **advantages:** parallel I/O operations, minimized transfer time with multiplied bandwidth
 - **disadvantages:** average seek time and rotational delay increased, blocking of parallel transactions, additional hardware needed
 - **use case:** large-size data access

RAID Systems

- Redundant Array of Independent Disks
 - storage system built as an **arrangement** of several disks
 - designed to **improve reliability** and **increase performance**
- Reliability
 - **redundancy**: replicate data onto multiple disks
 - leverage redundant information to improve **mean-time-to-failure**
- Performance
 - **data striping**: distribute data over disks
 - exploit parallel I/O to emulate a single large and very fast disk
- Implementation of RAID logic
 - **hardware RAID**: inside the disk subsystem or disk controller
 - **software RAID**: inside the operating system

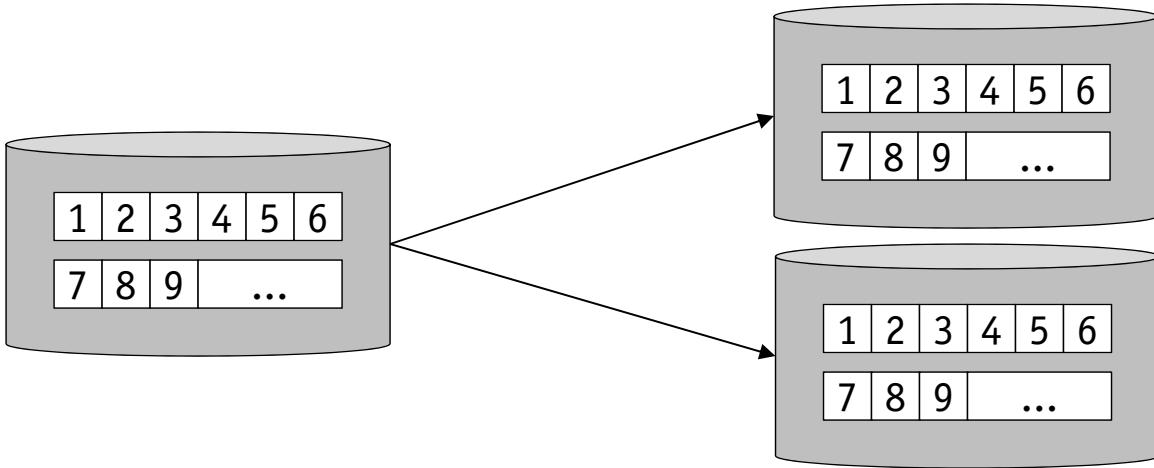
MTTF: Mean-Time-To-Failure

Example

If the MTTF of a single disk is 50,000 hours (\approx 5.7 years), then the MTTF of an array of 100 disks is only $50,000/100 = 500$ hours (\approx 21 days)!

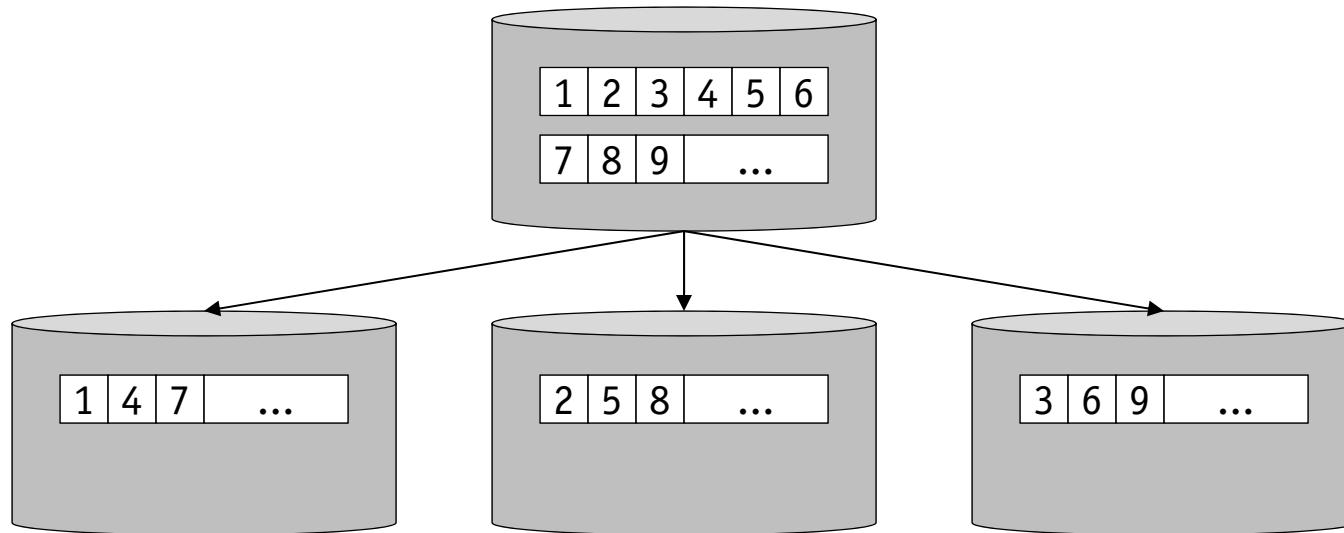
- Having more disks **increases** storage system performance, but also **decreases** overall storage system reliability
- 💣 Example is too simple!
 - disk failures do not occur independently, e.g., because of a “bad” batch
 - failure probability changes over time, i.e., disk failures are more likely to occur early and late in their lifetimes
- Use redundant information to reconstruct data of a failed disk
 - where is redundant information stored?
 - how is redundant information computed?

Disk Mirroring



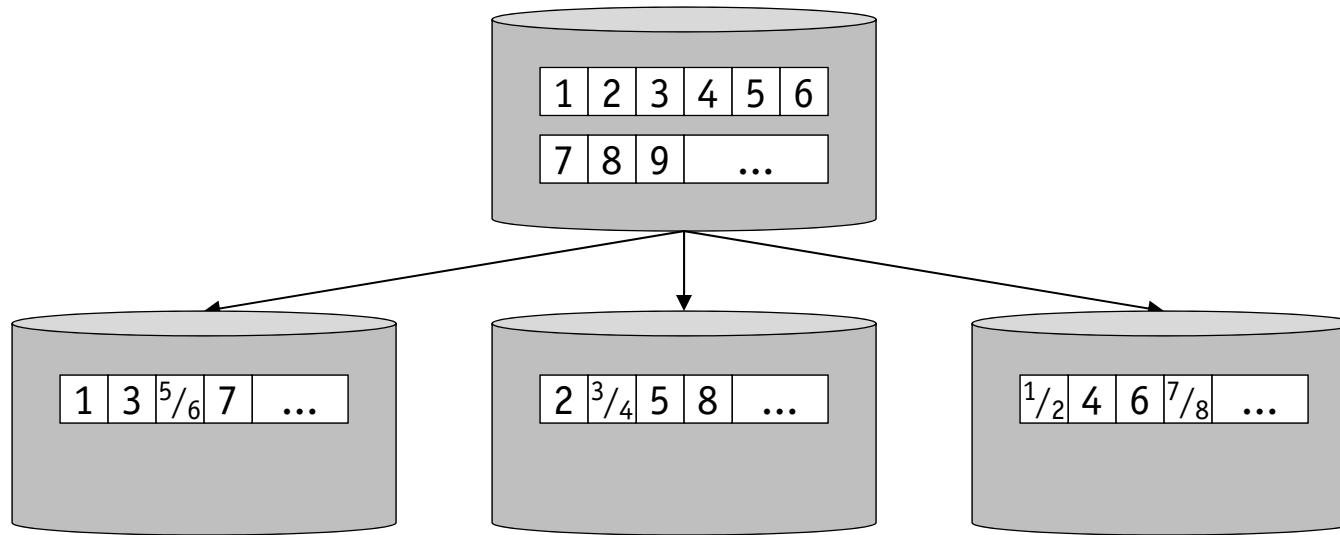
- Replicate copies of data onto multiple disks
 - write operations are executed on **all** disks
 - read operations are executed on **one** disk
- Benefits
 - **performance:** I/O parallelism only for reads
 - **reliability:** can survive failure of one disk

Disk Striping



- Partition data into equally-sized chunks of consecutive blocks
- Striping unit (i.e., chunk size) determines **degree of parallelism** for **single I/O** and **between different I/O operations**
 - **small chunks:** high intra-access parallelism, but many devices busy, i.e., not many I/O operations in parallel
 - **large chunks:** high inter-access parallelism, i.e., many I/O operations in parallel

Disk Striping with Parity



- Pure striping has a high failure risk as there is no redundancy
- Distribute data and parity information over ≥ 3 disks
- Benefits
 - **performance:** high I/O parallelism
 - **reliability:** depending on redundancy scheme, one or even two disks can fail at the same time

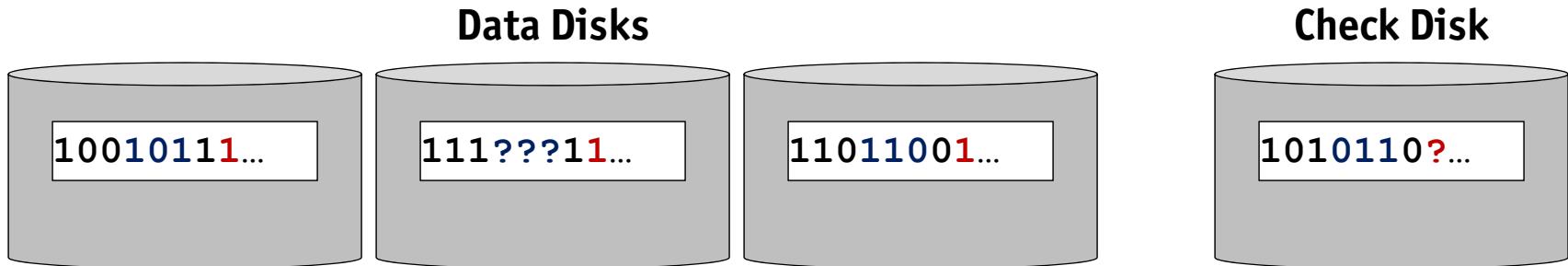
RAID Levels

- Based on the different options to maximize reliability and performance, so-called **RAID Levels** have been defined
- Striping unit (data interleaving)
 - **distribution:** how to scatter (primary) data across disks
 - **granularity:** fine (bits/bytes) or coarse (blocks)
- Computation and allocation of redundant information
 - **redundancy schemes:** parity, ECC, etc.
 - **distribution:** separate/few disks vs. all disks of the array
- Initially, five RAID Levels have been introduced, additional levels were later defined

Redundancy Schemes

- Parity scheme
 - requires one additional check disk
 - recovery from **one** disk is possible
- Hamming codes
 - requires $\log n$ check disks, assuming n data disks
 - recovery from **one** disk is possible, faulty disk can be identified
- Reed-Solomon codes
 - requires **two** check disks
 - recovery from **up to two** disks is possible

Parity Scheme



- Assume a disk array with D data disks ($D = 3$ above)
- **Initialization**
 - let $i(n)$ be the number of D data bits at the n th position that are **1**
 - n th **parity bit** on check disk is set to **1** if $i(n)$ is odd and to **0** otherwise
- **Recovery**
 - let $j(n)$ be the number of n th data bits that are **1** on $D - 1$ non-failed disks
 - if $j(n)$ is odd and the n th parity bit is **1**, or if $j(n)$ is even and the n th parity bit is **0**, then the value of the n th bit on the failed disk must be **0**
 - otherwise the value of the n th bit on the failed disk must be **1**

RAID Levels 0, 1, 10, and 2

- RAID Level 0: Non-redundant, just striping
 - least storage overhead (no redundancy)
 - **write**: no extra effort, **read**: not the best performance
- RAID Level 1: Mirroring
 - double necessary storage
 - **write**: doubles write access, **read**: optimized performance
- RAID Level 10 (a.k.a. 0+1): Striping and Mirroring
 - **write**: analogous to Level 1, **read**: improved performance over Level 0
- RAID Level 2: Error-Correcting Codes (ECC)
 - **initialization**: uses bit-level striping and Hamming code to compute ECC for data of n disks, which is stored onto $n - 1$ additional disks
 - **recovery**: determine lost disk by using $n - 1$ extra disk and correct (reconstruct) its content from one of those
 - large requests benefit from aggregated bandwidth (⌚ small requests)

RAID Levels 3, 4, 5, and 6

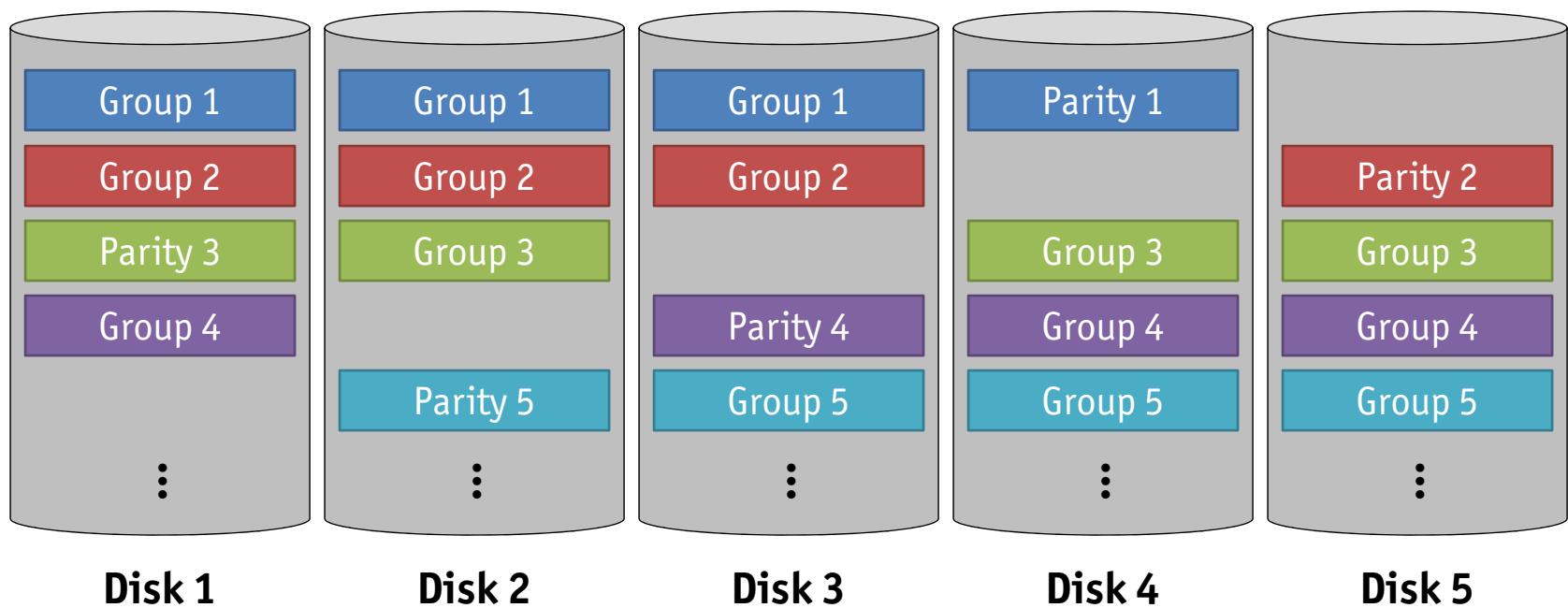
- RAID Level 3: Bit-Interleaved Parity
 - one parity disk suffices, since controller can easily identify faulty disk
 - distribute (primary) data bit-wise onto data disks
 - **read/write** go to all disks: no inter-I/O parallelism, but high bandwidth
- RAID Level 4: Block-Interleaved Parity
 - like RAID 3, but distribute data block-wise onto data disks
 - small **reads** go to one disk, but all **writes** go to one parity disk
- RAID Level 5: Block-Interleaved Distributed Parity
 - like RAID 4, but distribute parity blocks across all disks (load balancing)
 - best performance for small and large **reads** as well as large **writes**
- RAID Level 6: P+Q Redundancy
 - like RAID 5, but adds an additional parity block
 - performance equivalent to RAID 5, except that **small writes** involve more disks

Overview of RAID Levels



Parity Groups

- Parity is not necessarily computed across all disks of an array
- Possible to define parity groups (of same or different sizes)



Selecting RAID Levels

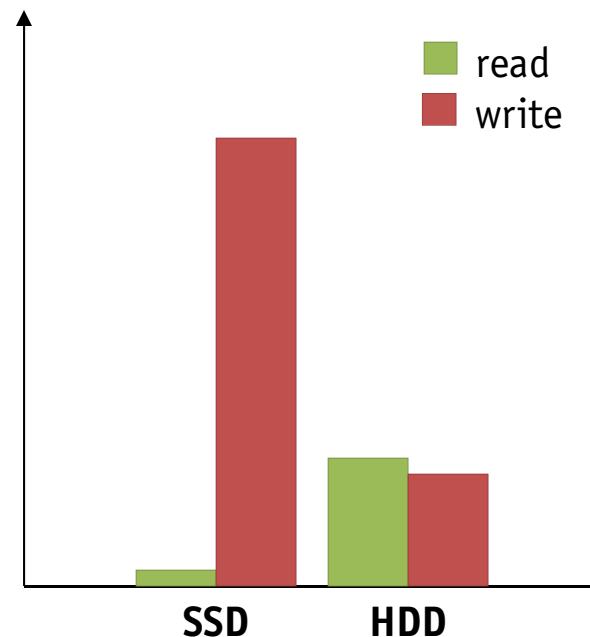
- RAID Level 0
 - improves overall performance at lowest cost
 - no provision against data loss
 - best write performance, since no redundancy
- RAID Level 10
 - superior to level 1
 - main application area is small storage subsystems, sometimes write-intensive applications
- RAID Level 1
 - most expensive variant
 - typically serialize to necessary I/O operations for writes to avoid data loss in case of power failure, etc.

Selecting RAID Levels

- RAID Level 2 and 4
 - always inferior to levels 3 and 5, respectively
- RAID Level 3
 - appropriate for workloads with large request for contiguous blocks
 - bad for many small requests of a single block
- RAID Level 5
 - good general-purpose solution
 - best performance (with redundancy) for small and large read as well as large write operations
- RAID Level 6
 - choice for higher level of reliability

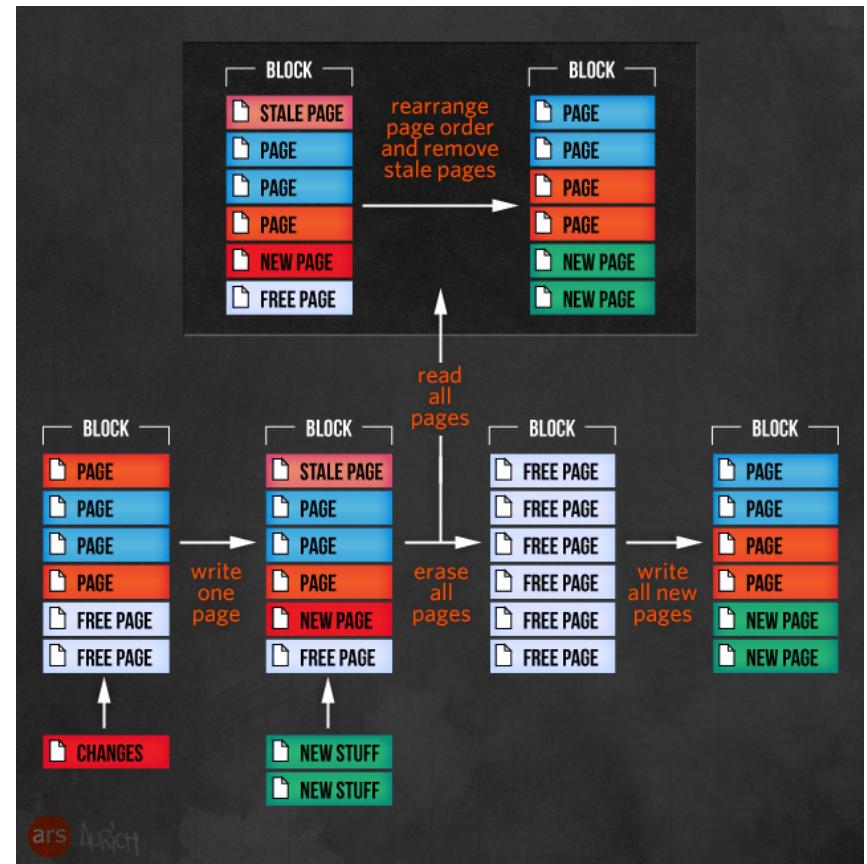
Solid-State Disks

- Solid-state disks (SSD) have emerged as an alternative to conventional hard disks (HDD)
- SSD provide **very low-latency read access** (< 0.01 ms)
- Random writes are **significantly slower** than on a HDD
 - (blocks of) pages have to be erased before they can be updated
 - once pages have been erased, sequentially writing them is almost as fast as reading



Solid-State Disks

- Page-level writes
 - typical **page size** is 128 kB
- Block-level deletes
 - SSD erase **blocks of pages**
 - block \approx 64 pages (8 MB)



Picture Credit: arstechnica.com (The SSD Revolution)

Example

- Seagate Pulsar.2
 - NAND flash memory, 800 GB capacity
 - standard 2.5" enclosure, no moving or rotating parts
 - data read and written in pages of size 128 kB
 - transfer rate \approx 370 MB/s



What is the access time to read an 8 kB block?

Example

- Seagate Pulsar.2
 - NAND flash memory, 800 GB capacity
 - standard 2.5" enclosure, no moving or rotating parts
 - data read and written in pages of size 128 kB
 - transfer rate \approx 370 MB/s

What is the access time to read an 8 kB block?

- no seek time $t_s = 0.00 \text{ ms}$
- no rotational delay $t_r = 0.00 \text{ ms}$
- transfer time for 8 kB ($\frac{128 \text{ kB}}{370 \text{ MB/s}}$) $t_t = 0.30 \text{ ms}$
- total **access time** for an 8 kB data block $t = 0.30 \text{ ms}$

Sequential vs. Random Access

Example: Read 1,000 blocks of size 8 kB

- The Seagate Pulsar.2 (sequentially) reads data in 128 kB chunks.

- **Random access**

$$t_{rnd} = 1,000 \times 0.30 \text{ ms} = \mathbf{0.30 \text{ s}}$$

- **Sequential read of adjacent blocks**

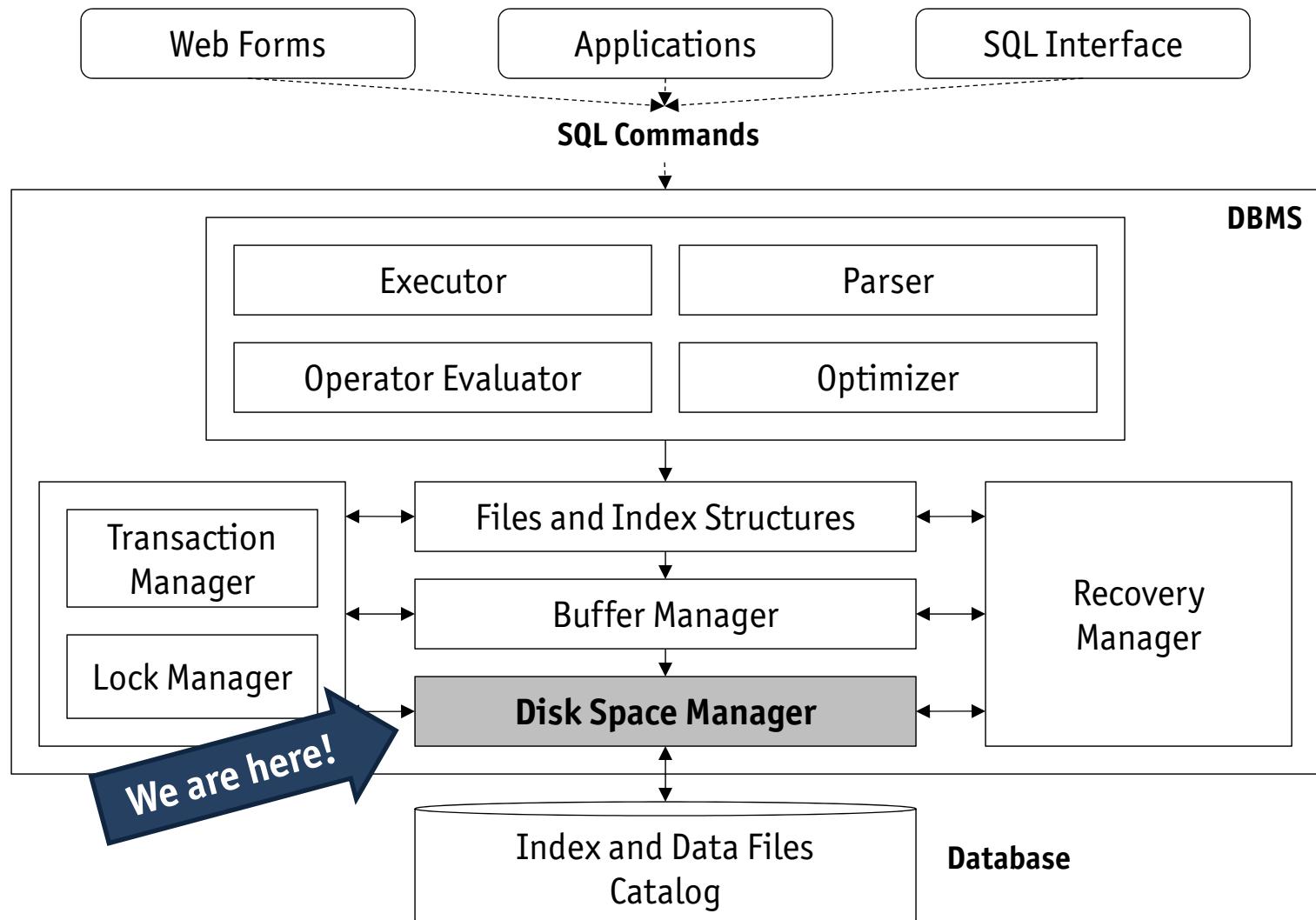
$$t_{seq} = \left\lceil \frac{1,000 \times 8 \text{ kB}}{128 \text{ kB}} \right\rceil \times t_t \approx \mathbf{18.9 \text{ ms}}$$

- Sequential access still beats random access
 - but random access is once again more feasible
- Adapting database technology to these characteristics is a current research topic

Network and Cloud-based Storage

- Today the network is **not** a bottleneck anymore
- Storage area network (SAN)
 - emulate interface of block-structured disks
 - hardware acceleration and simplified maintainability by abstracting from RAID or physical disk
 - fault tolerance and increased flexibility through multiple servers and storage resources
- Data management in the Cloud
 - University of Konstanz: Course INF-12820
 - M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska: **Building a Database on S3**. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pp. 251-264, 2008.

Orientation

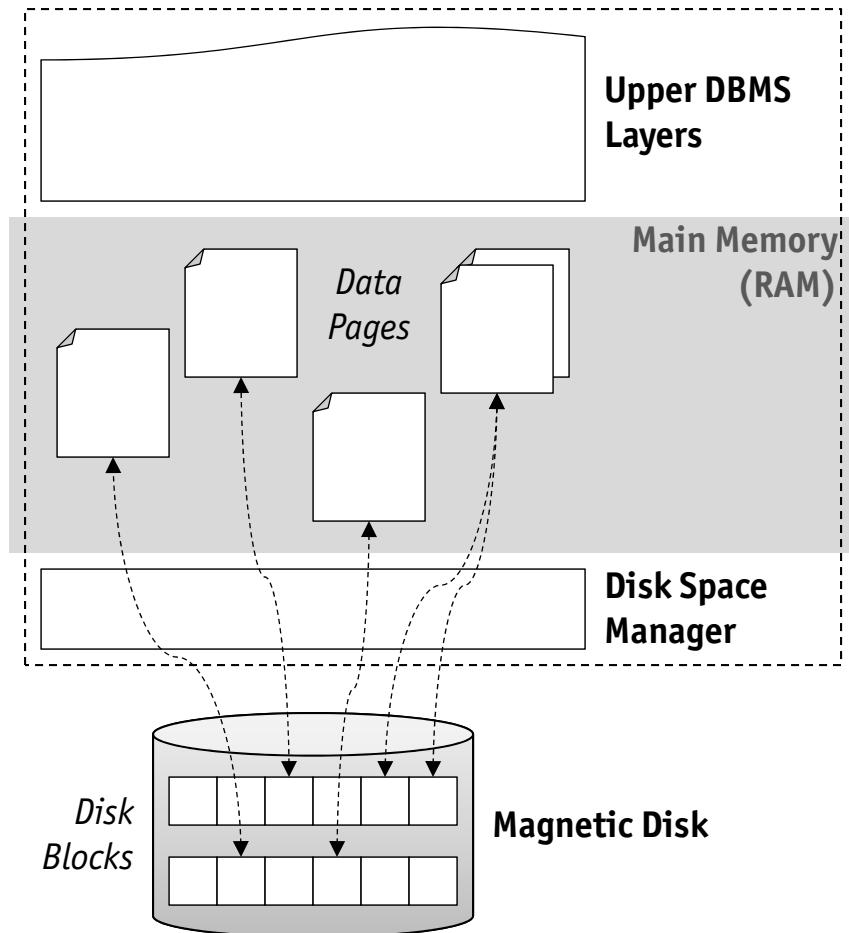


Disk Space Manager

- Abstracts from the gory details of the underlying storage
 - disk space manager talks to disk controller and initiates I/O operations
 - DBMS issues **allocate/deallocate** and **read/write** commands to disk space manager
- Provides the concept of a **page**
 - a page is a disk block that has been brought **into memory**
 - disk blocks and pages are of the **same size**
 - **sequences** of pages are mapped onto **contiguous sequences** of blocks
 - **unit of storage** for all system components in higher layers

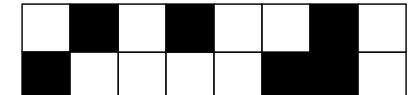
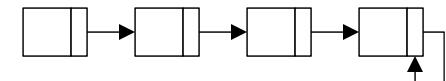
Disk Space Manager

- Locality-preserving mapping
 - track page locations and block usage internally
 - page number \leftrightarrow physical location
- Abstract physical location
 - OS file name and offset within that file
 - head, sector, and track of a hard disk drive
 - tape number and offset for data stored in tape library
 - ...



Managing Free Space

- Disk space manager also keeps track of **used** and **free blocks** to reclaim space that has been freed
 - blocks can usually be allocated contiguously on database/table creation
 - subsequent de-allocation and new allocation may create **holes**
- **Linked list** of free blocks
 1. keep a pointer to the **first free block** in a known location on disk
 2. when a block is no longer needed, append/prepend it to the list
 3. the **next** pointer may be stored in blocks themselves
- Free block **bitmap**
 1. reserve a block whose bytes are interpreted bitwise (bit $n = 0$: block n is free)
 2. toggle bit n whenever block n is (de-)allocated



Contiguous Sequences of Pages

Exercise

To exploit **sequential access**, it is useful to allocate **contiguous sequences** of pages

Which technique would you use, a linked list or a bitmap of free blocks?

Contiguous Sequences of Pages

Exercise

To exploit **sequential access**, it is useful to allocate **contiguous sequences** of pages

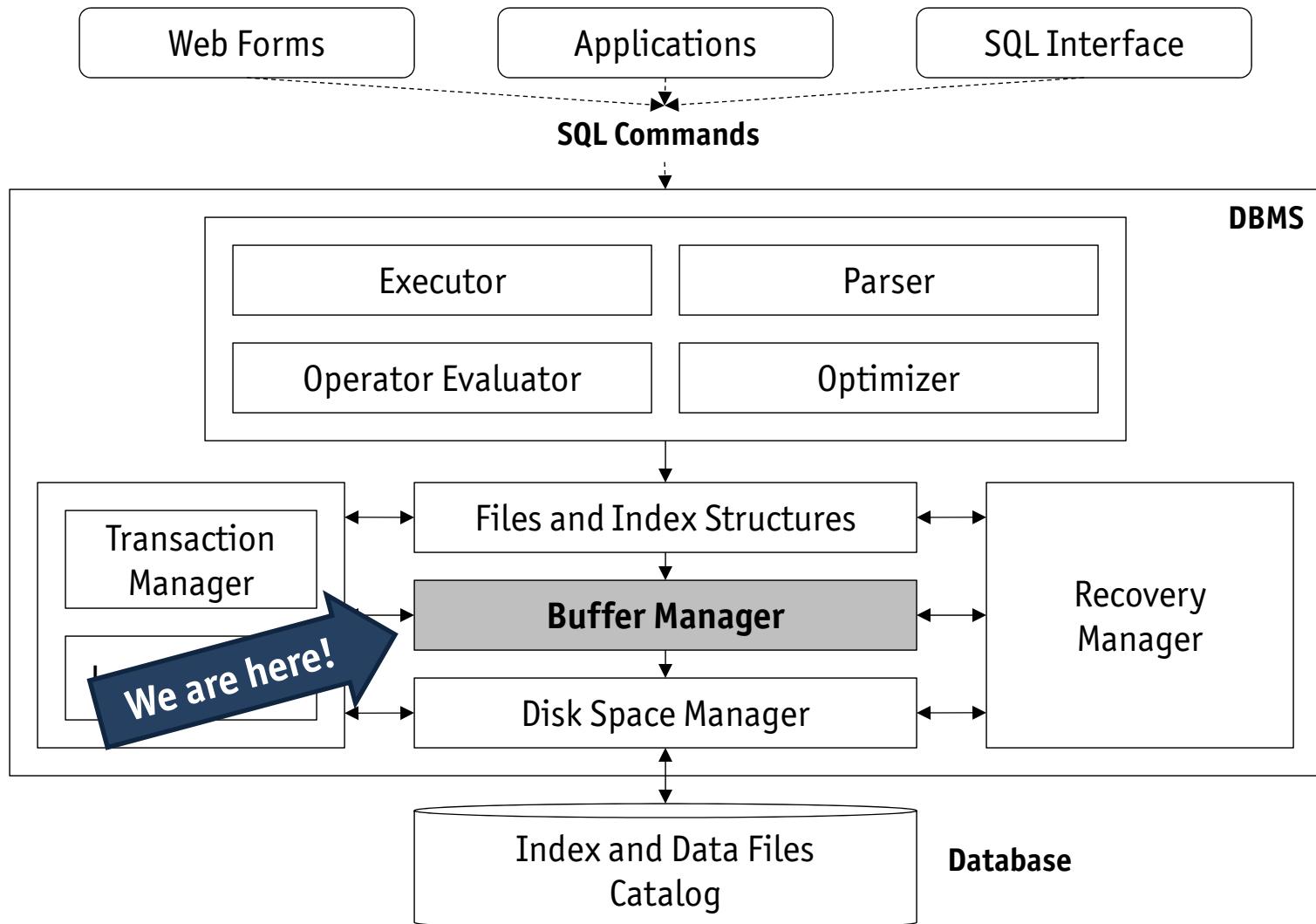
Which technique would you use, a linked list or a bitmap of free blocks?

- ↳ Free block bitmaps support fast identification of contiguous sequences of free blocks
- ↳ **Minibase** uses a bitmap of free blocks in its disk space manager (`DiskManager.java`)

Segments, Table Spaces, and Partitions

- Most DBMS do not manage the entire data store as **one** contiguous storage space
- Data store can be partitioned into smaller units
 - segments
 - table spaces
 - partitions
 - storage pools
 - *and many other names*
- These units are arranged in a layered stack on top of the physical disks
 - **simple use of OS file** $table \leftarrow [1:1] \rightarrow segment \leftarrow [1:1] \rightarrow (OS) file$
 - **more flexible** $table \leftarrow [n:1] \rightarrow segment \leftarrow [1:1] \rightarrow (OS) file$
 - **advanced** $table \leftarrow [n:1] \rightarrow segment \leftarrow [n:1] \rightarrow storage group\dots$
 $\dots \leftarrow [n:1] \rightarrow logical (OS) volume$

Orientation



Buffer Manager

Recall

size of database on secondary storage

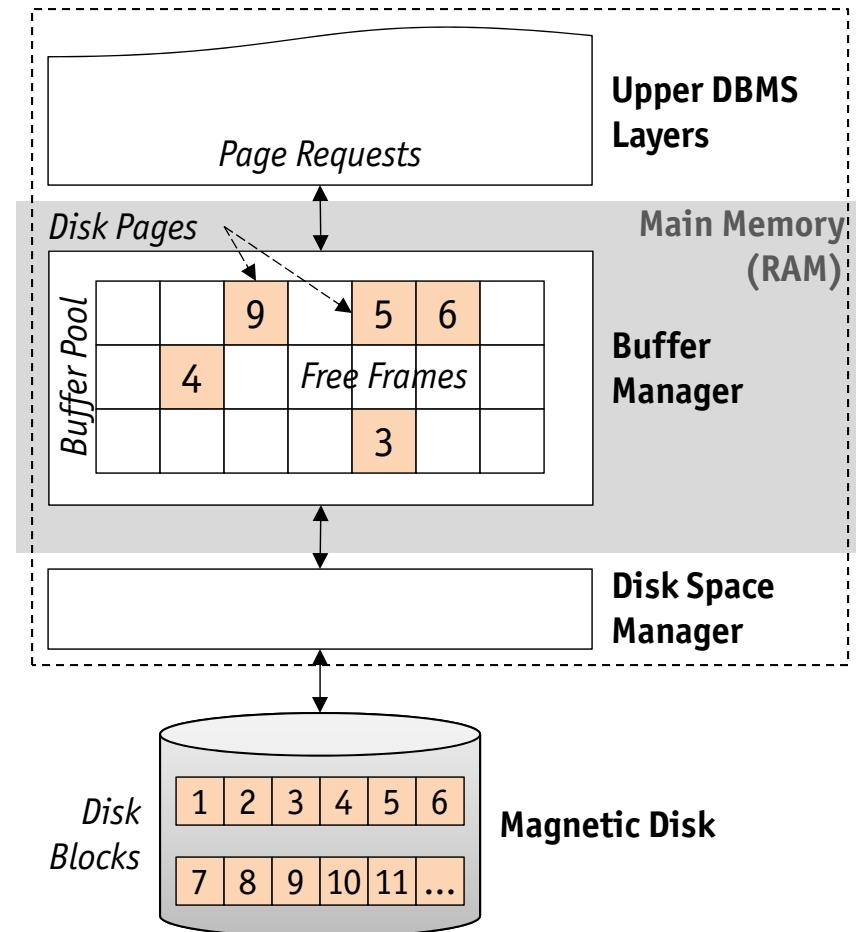
>>

size of available primary memory to hold user data

- To scan the entire pages of a 50 GB table (`SELECT * FROM ...`), the DBMS needs to
 - **bring pages into memory** as they are needed for processing
 - **overwrite (replace) pages** when they become obsolete and new pages are required
- The **buffer manager** mediates between external storage and main memory

Buffer Pool

- **Buffer pool** is a designated main memory area managed by the buffer manager
 - organized as a collection of **frames** (empty slots)
 - pages brought into memory and are loaded into frames as needed
 - a **replacement policy** decides, which page to evict when the buffer is full



Buffer Manager Interface

- Higher-level code requests (**pins**) pages from the buffer manager and releases (**unpins**) pages after use
- **pin (pageNo)**
 - request page number *pageNo* from the buffer manager
 - if necessary, load page into memory
 - mark page as clean ($\neg\text{dirty}$)
 - return a reference to the frame containing page number *pageNo*
- **unpin (pageNo, dirty)**
 - release page number *pageNo*, making it a candidate for eviction
 - if page was modified, *dirty* must be set to **true**

 Why do we need the *dirty* bit?

Buffer Manager Interface

- Higher-level code requests (**pins**) pages from the buffer manager and releases (**unpins**) pages after use
- **pin (pageNo)**
 - request page number *pageNo* from the buffer manager
 - if necessary, load page into memory
 - mark page as clean ($\neg\text{dirty}$)
 - return a reference to the frame containing page number *pageNo*
- **unpin (pageNo, dirty)**
 - release page number *pageNo*, making it a candidate for eviction
 - if page was modified, *dirty* must be set to **true**

Why do we need the *dirty* bit?

Only **modified** pages need to be written back to disk upon eviction.

Proper Nesting of `pin()` and `unpin()`

⌚ A read-only page operation

```
a ← pin(p) ;  
{ ...  
  read data on page at  
  memory address a  
  ...  
unpin(p, false) ;
```

⌚ A read/write page operation

```
a ← pin(p) ;  
{ ...  
  read and modify data (records)  
  on page at memory address a  
  ...  
unpin(p, true) ;
```

- All database transactions are required to properly “bracket” page operations using `pin()` and `unpin()` calls
- Proper bracketing enables the system keeps a count of active users (e.g., transactions) accessing a page (`pinCount`)

Writing Pages Back to Disk

- Pages are written back to disk when evicted from buffer pool
 - “clean” victim pages are not written back to disk
 - call to `unpin()` does not trigger any I/O operations, even if the `pinCount` of the page becomes 0
 - pages with `pinCount = 0` are simply a suitable **victim** for eviction

☞ Digression: Transaction Management

Wait! Stop! This makes no sense... If pages are written only when they are evicted from the buffer pool, then how can we ensure proper transaction management, i.e., guarantee durability?

⇒ A buffer manager typically offers at least one or more interface calls (e.g., `flushPage(p)`) to **force** a page p (synchronously) back to disk

Implementation of pin ()

Function pin (*pageNo*)

```
if buffer pool already contains pageNo then
    pinCount (pageNo)  $\leftarrow$  pinCount (pageNo) + 1
    return address of frame holding pageNo;
else
    select a victim frame v using the replacement policy
    if dirty (page in v) then
        write page in v to disk
    read page pageNo from disk into frame v
    pinCount (pageNo)  $\leftarrow$  1
    dirty (pageNo)  $\leftarrow$  false
    return address of frame v
```

Implementation of unpin ()

💻 Function unpin (*pageNo*, *dirty*)

```
pinCount(pageNo) ← pinCount(pageNo) - 1  
dirty(pageNo) ← dirty(pageNo) ∨ dirty
```

✍️ But... Why don't we write pages back to disk during unpin () ?

Implementation of unpin ()

💻 Function unpin (*pageNo, dirty*)

```
pinCount(pageNo) ← pinCount(pageNo) - 1  
dirty(pageNo) ← dirty(pageNo) ∨ dirty
```

✍ But... Why don't we write pages back to disk during unpin () ?

Of course, this is a possibility...

- ⊕ recovery from failure would be **a lot** simpler
 - ⊖ higher I/O cost (**every** page write implies a write to disk)
 - ⊖ **bad response** time for writing transactions
- ☞ This discussion is also known as **force** (or **write-through**) vs. **write-back**. Actual database systems typically implement write-back.

Transaction Management (oh no, not again...)

☞ Digression: Conflicting concurrent writes to a block

Assumptions

1. the same page p is requested by **more than one** transaction, i.e.,
the `pinCount(p) > 1`
2. those transactions perform **conflicting writes** on p

Conflicts of this kind are resolved by the system's **concurrency control**, a layer on top of the buffer manager. For more information, see courses "Database Systems" (INF-12040) and "Transactional Information Systems" (INF-11610).

- ☞ The buffer manager can assume that everything is in order whenever it receives an `unpin(p, true)` call.

Two Strategic Questions

1. **Buffer Allocation Problem:** How much precious buffer space should be allocated to each active transaction?
 - **static** assignment
 - **dynamic** assignment
 2. **Page Replacement Problem:** Which page will be replaced when a new request arrives and the buffer pool is full?
 - decide without knowledge of reference pattern
 - presume knowledge of (expected) reference pattern
- Additional complexity is introduced when we take into account that DBMS may manage “segments” of different page sizes
 - **one buffer pool:** good space utilization, but fragmentation problem
 - **multiple buffer pools:** no fragmentation, but worse utilization, global allocation/replacement strategy may get complicated
 - A possible solution is to support set-oriented **pin ({p})** calls

Buffer Allocation Policies

- **Problem:** allocate parts of buffer pool to each transaction (TX) or let replacement policy decide who gets how much space?
- **Local** policies...
 - allocate buffer frames to a specific TX **without** taking the reference behavior of concurrent TX into account
 - have to be supplemented with a mechanism for handling the allocation of buffer frames for shared pages
- **Global** policies...
 - consider **not only** the reference pattern of the transaction currently executing, **but also** the reference behavior of all other transactions
 - based allocation decision on data obtained from all transactions

Local vs. Global Policies

- **Properties** of a local policy
 - ⊕ one TX **cannot hurt** other transactions
 - ⊕ all TX are treated **equally**
 - ⊖ possibly, bad **overall** utilization of buffer space
 - ⊖ some TX may occupy vast amounts of buffer space with “old” pages, while others suffer from too little space (“**internal page thrashing**”)
- **Problem** with a global policy (*assume TX reading a huge relation sequentially*)
 - all page accesses are references to **newly loaded pages**
 - hence, almost **all other pages** are likely to be replaced (following a standard replacement policy)
 - other TX cannot proceed without loading their pages again (“**external page thrashing**”)

Buffer Allocation Policies

- **Global** one buffer pool for all transactions
- **Local** use various information (e.g., catalog, index, data, ...)
- **Local** each TX gets a certain fraction of the buffer pool
 - **static partitioning** *assign buffer budget once for each TX*
 - **dynamic partitioning** *adjust buffer budget of TX according to its past reference pattern or some kind of semantic information*
- It is also possible to apply **mixed** policies
 - have **different** buffer pools with **different** approaches
 - mixed policies **complicate** matters significantly!

Dynamic Buffer Allocation Policies

- **Local LRU** (*cf. LRU replacement policy, later*)
 - keep a separate **LRU-stack** for each active TX
 - keep a global **freelist** for pages not pinned by any TX

Strategy

1. replace a page from the freelist
2. replace a page from the LRU-stack of the requesting TX
3. replace a page from the TX with the largest LRU-stack

- **Working Set Model** (cf. virtual memory management of OS)

goal: avoid thrashing by allocating “just enough” buffers to each TX

approach: observe number of different page requests by each TX in a certain interval of time (window size τ)

- deduce “optimal” buffer budget from this observation
- allocate buffer budgets according to ratio between those optimal sizes

Buffer Replacement Policies

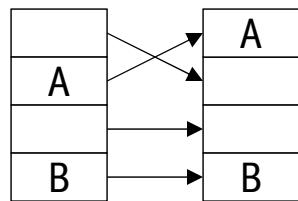
- Choice of **buffer replacement policy** (victim frame selection) can considerably affect DBMS performance
- Large number of policies in OS and DBMS

References	Criteria	Age of page in buffer		
		no	since last reference	total age
none	Random			FIFO
	last		LRU CLOCK GCLOCK(V1)	
	all	LFU	GCLOCK(V2) DG_CLOCK	LRD(V1)
				LRD(V2)

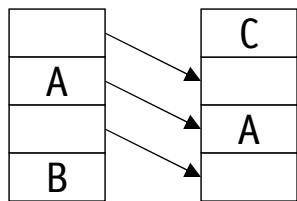
Criteria for victim selection used in some policies

Typical Buffer Replacement Policies

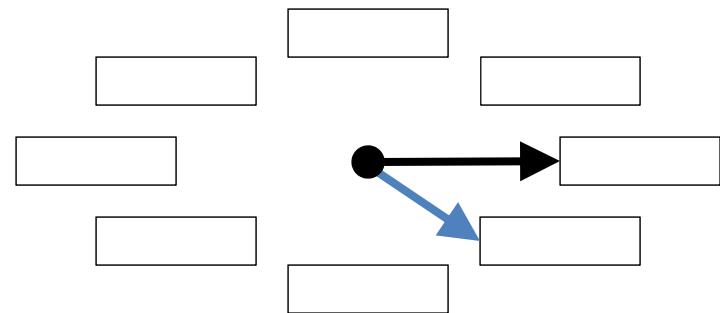
- DBMS typically implement more than one replacement policy
 - **FIFO** (“first in, first out”)
 - **LRU** (“least recently used”): evicts the page whose latest `unpin()` is longest ago
 - **LRU- k** : like LRU, but evicts the k latest `unpin()` call, not just the latest
 - **MRU** (“most recently used”): evicts the page that has been unpinned most recently
 - **LFU** (“least frequently used”)
 - **LRD** (“least reference density”)
 - **CLOCK** (“second chance”): simulates LRU with less overhead (no LRU queue reorganization on every frame reference)
 - **GCLOCK** (“generalized clock”)
 - **WS, HS** (“working set”, “hot set”)
 - **Random**: evicts a random page

LRU

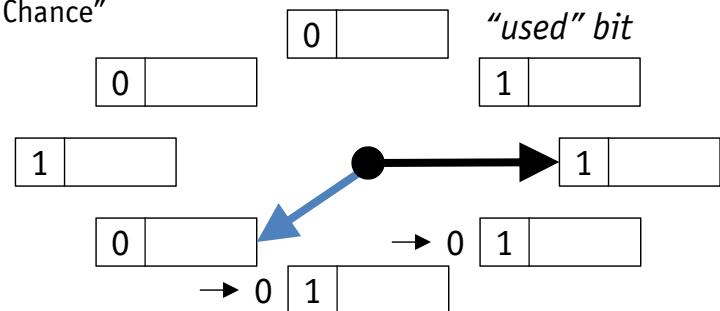
Reference to A in buffer



Reference to C not in buffer

FIFO**LFU***Victim page*

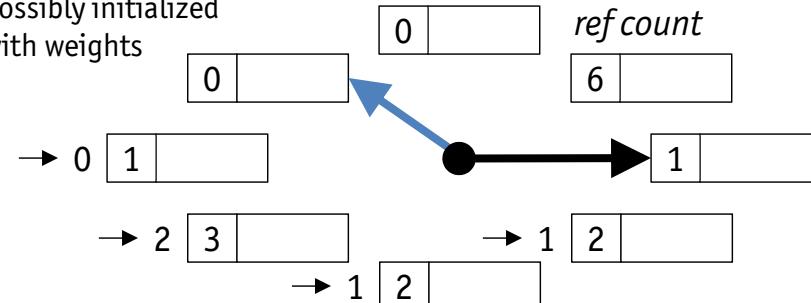
2	
3	
1	
3	
3	
6	
1	
3	

*or***CLOCK***"Second Chance"***LRD(V1)***Victim page*

	<i>rc</i>	<i>age</i>	
2	20		
3	26		
1	40		
3	45		
3	5		
6	2		
1	37		
3	17		

gc

50

GCLOCKpossibly initialized
with weights

Details of LRU and CLOCK Policy

Example: LRU Buffer Replacement Policy

1. keep a **queue** (often described as a **stack**) of pointers to frames
2. in **unpin (pageNo, dirty)** , append p to the **tail** of queue, if **pinCount (pageNo)** is decremented to 0
3. to find next victim, search through the queue from its **head** and find the first page p with **pinCount (pageNo) = 0**

Example: CLOCK Buffer Replacement Policy

1. number the N buffer frames $0 \dots N-1$, initialize **current** $\leftarrow 0$, maintain a bit array **referenced** [$0 \dots N-1$], which is initialized to all 0
2. in **pin (pageNo)** , do **referenced [pageNo] $\leftarrow 1$**
3. to find the next victim, consider page **current**:
if **pinCount (current) = 0** and **referenced [current] = 0**, **current** is the victim; otherwise, **referenced [current] $\leftarrow 0$** , **current $\leftarrow (\text{current} + 1) \bmod N$** ; repeat 3.

Heuristic Policies Can Fail

- These buffer replacement policies are **heuristics** only and can fail miserably in certain scenarios

Example: A Challenge for LRU

A number of transactions want to scan the same sequence of pages (as for example a repeated **SELECT * FROM R**). Assume a buffer pool capacity of 10 pages.

1. Let the size of relation R be 10 or less page.
How many I/O operations do you expect?
2. Let the size of the relation R be 11 pages.
What about the number of I/O operations in this case?

Details of LRD

- Record the following three parameters
 - $trc(t)$ total reference count of transaction t
 - $age(p)$ value of $trc(t)$ at the time of loading p into buffer
 - $rc(p)$ reference count of page p
- Update these parameters during a transaction's page references (**pin (pageNo)** calls)
- Compute **mean reference density** of a page p at time t as

$$rd(p, t) := \frac{rc(p)}{trc(t) - age(p)}, \text{ where } trc(t) - rc(p) \geq 1$$

- **Strategy** for victim selection
 - chose page with least reference density $rd(p, t)$
 - many variants exist, e.g., for gradually disregarding old references

Exploiting Semantic Knowledge

- **Background:** query compiler/optimizer already...
 - selects access plan, e.g., sequential scan vs. index
 - estimates number of page I/O operations for cost-based optimization
- **Idea:** use this information to determine query-specific, optimal buffer budget, i.e., a **Hot Set**
- Goals of **Query Hot Set** model
 - optimize overall system throughput
 - avoiding thrashing is the most important goal

Hot Set with Disjoint Page Sets

☛ Mode of Operation

1. only those queries are activated, whose Hot Set buffer budget can be satisfied immediately
2. queries with higher demand have to wait until their budget becomes available
3. within its own buffer budget, each transaction applies a local LRU policy

- Properties
 - ⊖ **no sharing** of buffered pages between transactions
 - ⊖ risk of **internal thrashing** when Hot Set estimates are wrong
 - ⊖ queries with large Hot Sets **block** following small queries (or, if bypassing is permitted, many small queries can lead to **starvation** of large queries)

Hot Set with Non-Disjoint Page Set

↷ Mode of Operation

1. queries allocate their budget stepwise, up to the size of their Hot Set
 2. local LRU stacks are used for replacement
 3. request for a page p
 - i. if found in **own LRU stack**: update LRU stack
 - ii. if found in **another transaction's LRU stack**: access page, but do not update the other LRU stack
 - iii. if found in **freelist**: push page on own LRU stack
 4. call to **unpin (pageNo)** : push page onto freelist stack
 5. filling empty buffer frames: taken from the bottom of the freelist stack
-
- As long as a page is in a local LRU stack, it cannot be replaced
 - If a page drops out of a local LRU stack, it is pushed onto freelist stack
 - A page is replaced only if it reaches the bottom of the freelist stack before some transaction pins it again

Priority Hints

- **Idea:** with `unpin (pageNo)`, a transaction gives one of two possible indications to the buffer manager
 - **preferred page** managed in a transaction-local partition
 - **ordinary page** managed in a global partition
- **Strategy:** when a page needs to be replaced...
 1. try to replace an ordinary page from the global partition using LRU
 2. replace a preferred page of the requesting transaction using MRU
- **Advantages**
 - much simpler than Hot Set, but similar performance
 - easy to deal with “too small” partitions

☞ Variant: Fixing and Hating Pages

Operator can **fix** a page if it may be useful in the near future (e.g., *nested-loop join*) or **hate** a page it will not access any time soon (e.g., *pages in a sequential scan*)

Prefetching

- Buffer manager can try to **anticipate** page requests
 - asynchronously read ahead even if only a single page is requested
 - improve performance by overlapping CPU and I/O operations
- Prefetching techniques
 - **prefetch lists: on-demand, asynchronous read-ahead**
e.g., when traversing the sequence set of an index, during a sequential scan of a relation
 - **heuristic (speculative) prefetching**
e.g., sequential n -block look-ahead (cf. drive or controller buffers in hard disks), semantically determined supersets, index prefetch, ...

Prefetching

The Real World

IBM DB2

- supports both **sequential** and **list** prefetch (prefetching a list of pages)
- **default prefetch size** is 32 4 kB pages (user-definable), but for some utilities (e.g., COPY, RUNSTAT) pages up to 64 4 kB are prefetched
- for small buffer pools (i.e., < 1000 buffers) prefetch adjusted to 8 or 16 pages
- prefetch size can be defined by user (sometimes it makes sense to prefetch 1000 pages)

Oracle 8

- uses prefetching for **sequential scan**, retrieving **large objects**, and certain **index scans**

Microsoft SQL Server

- supports prefetching for **sequential scan** and for scans along the leaf-level of a **B+ tree index**
- prefetch size can be adjusted during a scan
- extensive use of asynchronous (speculative) prefetching

Database vs. Operating System

- **Stop!** What you are describing is an **operating system (OS)**!
- Well, yes...
 - disk space management and buffer management very much look like **file management** and **virtual memory (VM)** in an operating system
- But, no...
 - DBMS can predict the **access patterns** of certain operators a lot better than the operating system (prefetching, priority hints, etc.)
 - **concurrency control** is based on protocols that prescribe the order in which pages have to be written back to disk
 - **technical reasons** can make operating systems tools unsuitable for a database (e.g., file size limitation, platform independence)

Double Buffering

- DBMS buffer manager within VM of DBMS server process can **interfere** with OS VM manager
 - **virtual page fault**: page resides in DBMS buffer, but the frame has been swapped out by operating system VM manager
↳ **one I/O operation** is necessary that is not visible to the DBMS
 - **buffer fault**: page does not reside in DBMS buffer, but frame is in physical memory
↳ regular DBMS page replacement requiring **one I/O operation**
 - **double page fault**: pages does not reside in DBMS buffer and frame has been swapped out of physical memory by operating system VM manager
↳ **two I/O operations** necessary: one to bring in the frame (OS) and another one to replace the page in that frame (DBMS)
- DBMS buffer needs to be **memory resident** in OS

Buffer Management in Practice

The Real World

↳ IBM DB2

- buffers can be partitioned into named pools
- each database, table, or index can be bound to a pool
- each pool uses FIFO, LRU or (variant of) Clock buffer replacement policy
- supports “hating” of pages

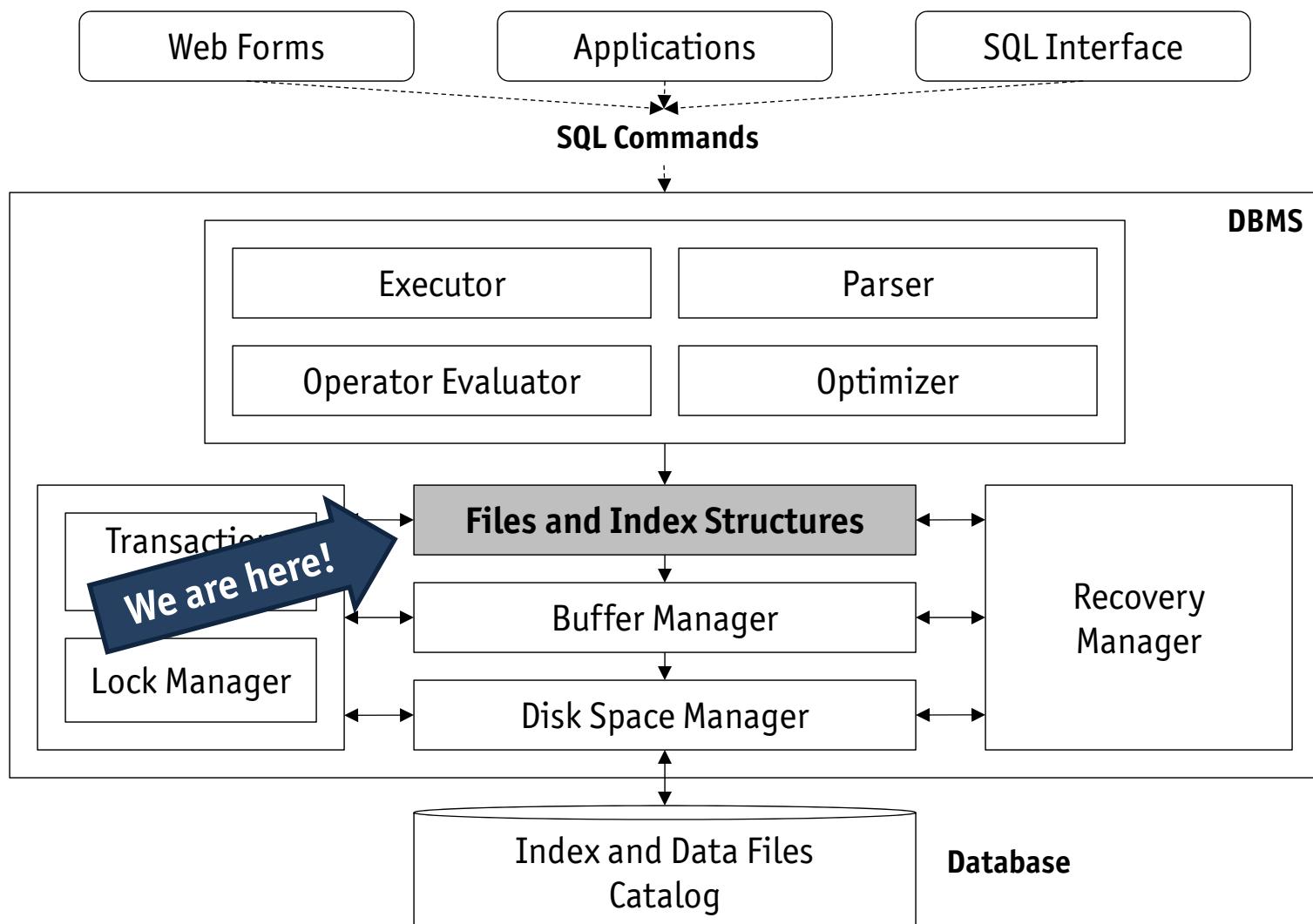
↳ Oracle 7

- maintains a single global buffer pool using LRU

↳ Microsoft SQL Server

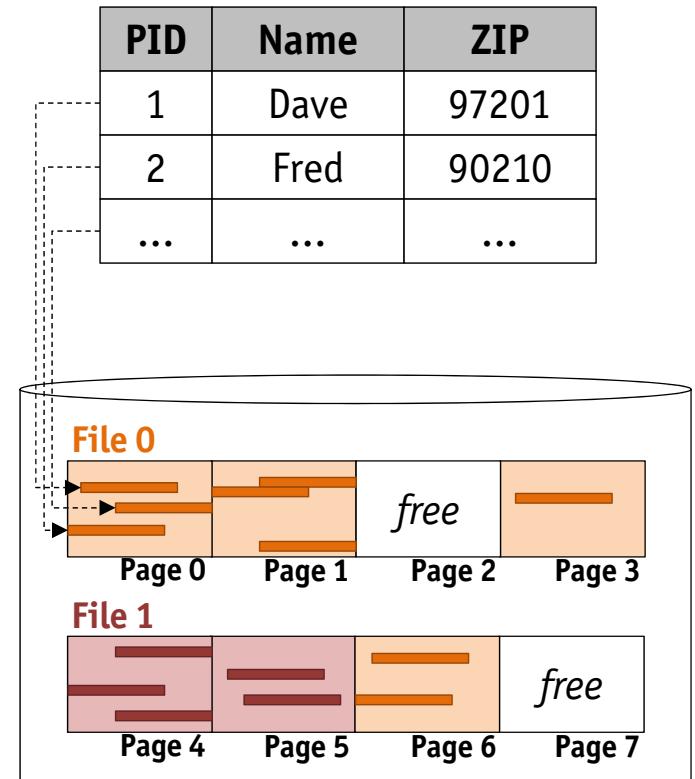
- uses a single buffer pool with Clock replacement
- supports a “reservation” of pages by queries that require large amount of memory (e.g., queries involving sorting or hashing)

Orientation



Database Files

- Focus change
 - **the road so far:** how does a DBMS manage pages?
 - **now:** how does a DBMS use pages to store data?
- On the conceptual level,
a **relational** DBMS manages
tables of rows and indexes
- On the physical level, these
data structures are implemented
as **files of records**
 - each file consists of **one or more pages**
 - each page contains **one or more records**
 - each record corresponds to **one row**



Database Heap Files

- The most important and most simple file structure in a database is the heap file
 - represents an **unordered** collection of records (cf. SQL semantics)
 - each record in a heap file has a **unique record identifier (*rid*)**
- Record ids (*rids*) are used like **record addresses** (or pointers)
 - internally, the heap file structure must be able to **map** a given rid to the page containing the record

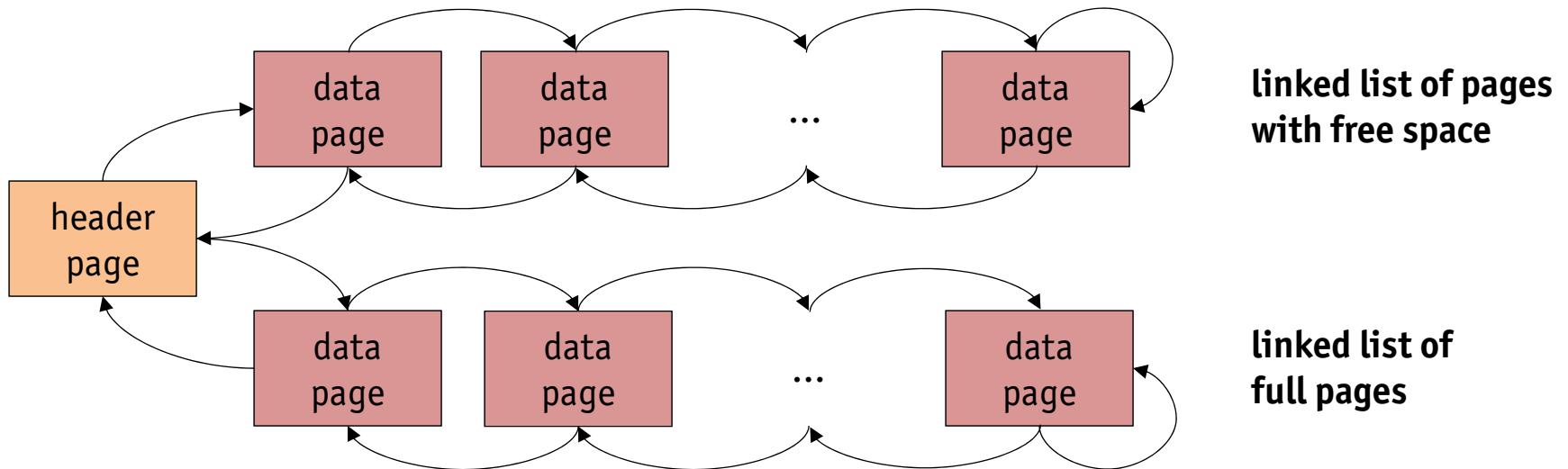
典型堆文件接口

- **create/destroy** heap file *f* named *n*: $f \leftarrow \text{createFile}(n)$ and $\text{deleteFile}(n)$
- **insert** record *r* and return its *rid*:
 $rid \leftarrow \text{insertRecord}(f, r)$
- **delete** a record with a given *rid*:
 $\text{deleteRecord}(f, rid)$
- **get** a record with a given *rid*:
 $r \leftarrow \text{getRecord}(f, rid)$
- initiate a **sequential scan** over the whole heap file:
 $\text{openScan}(f)$

Free Space Management

- Implications of the heap file interface
 - to support **openScan** (f) , the heap file structure has to **keep track of all pages in the file f**
 - to support **insertRecord** (f, r) efficiently, the heap file structure also needs to **keep track of all pages with free space in the file f**
- In this course, we will look at two simple structures that can offer this support
 - (doubly) **linked list** of pages
 - **directory** of pages

Linked List of Pages



Operation $f \leftarrow \text{createFile}(n)$

1. DBMS allocates a free page (called file **header page**) and stores an entry $\langle n, \text{header page} \rangle$ to a known location on disk
2. header page is initialized to point to two doubly linked list of pages: one containing **full pages** and one containing **pages with free space**
3. initially, both lists are **empty**

Linked List of Pages

↷ Operation $rid \leftarrow \text{insertRecord}(f, r)$

1. try to **find a page** p in the free list with space $> |r|$
2. should this fail ask the disk space manager **to allocate a new page** p
3. record r is **written** to page p
4. since generally $|r| \ll |p|$, p will belong to the **list of pages with free space**
5. a **unique** rid for r is computed and returned to the caller

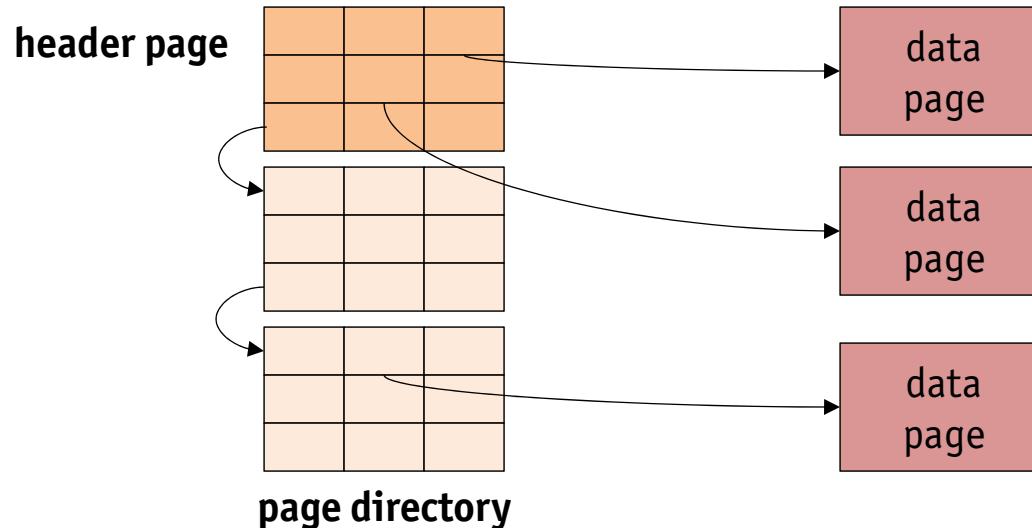
↷ Operation $\text{deleteRecord}(f, r)$

1. may result in **moving the containing page** from the list of full pages to the list of pages with free space
2. may even lead to **page deallocation** if the page is completely free after deletion

↷ Operation $\text{openScan}(f)$

1. **both** page lists have to be traversed

Directory of Pages



- **Header page** contains first page of a chain of **directory pages**
 - each entry in a directory page identifies a page of the file
 - $|page\ directory| \ll |data\ pages|$
- Free space management is also done through the directory
 - space vs. accuracy **trade-off**: from *open/closed* flag to exact information
 - for example, entries could be of the form $\langle page\ addr\ p, nfree \rangle$, where $nfree$ indicates **actual amount of free space** (e.g., in bytes) on page p

Free Space Management

- Keeping **exact** counter value (e.g., $nfree$) during updates may produce a performance bottleneck in multi-user operations
 - each transaction, whose update changes the amount of available free space, needs to update this meta-information in the directory
 - locking (or some other form of synchronization) needs to be applied to avoid lost updates
 - frequent updates of the same record lead to “hot data item”, introducing lock-wait queues and thus reducing degree of parallelism
- Keep **fuzzy information** on available free space in directory, e.g., $\lfloor nfree/8 \rfloor$ (units of 8 bytes or some other granularity)
 - directory only needs to be updated for “large” changes in the available free space on a page
 - page itself contains the exact information (of course)

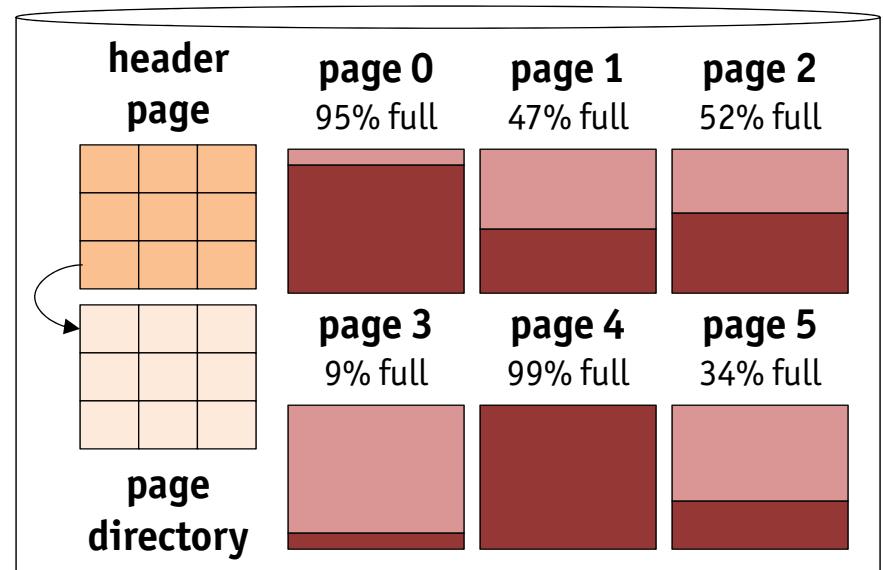
Record Insertion Strategies

- **Append Only**
 - always insert into the last page
 - otherwise, create a new page
- **Best Fit**
 - search from beginning and pick the page with the amount of free space that most closely matches the required space
 - reduces fragmentation, but requires search the entire list or directory
- **First Fit**
 - search from beginning and pick the first page with sufficient space
 - since first pages fill up quickly, system may waste a lot of search effort in these pages later on
- **Next Fit**
 - maintain cursor and continue searching where last search ended

Free Space Witnesses

- Accelerate search by remembering **witness pages**
 - classify pages into **buckets**
 - for each bucket, remember a **witness page**, i.e., any page that falls into that bucket
 - only perform standard best/first/next fit search, if no witness page is recorded for the specific bucket
 - populate witness information, e.g., as a side effect when searching

Bucket#	%Full	Witness
0	75%–100%	0
1	50%–75%	2
2	25%–50%	—
3	0%–25%	3



Linked List vs. Directory

I/O operations and free space management

For a file of 10,000 pages, give lower and upper bound for the number of I/O operations during an `insertRecord(f, r)` call for a heap file organized using

1. a **linked list** of pages

2. a **directory** of pages (1,000 directory entries/page)

Linked List vs. Directory

- Linked list of free pages
 - ⊕ easy to implement
 - ⊖ most pages will end up in the list of pages with free space
 - ⊖ might have to search many pages to place a (large) record
- Directory of free pages
 - ⊕ free space management more efficient
 - ⊖ memory overhead to host the page directory

Page Formats

- Locating the page containing a given *rid* is not the whole story
 - **internal structure of pages** plays a crucial role
 - we think of a page as sequence of **slots**, each of which contains a record

Generating sensible record ids (*rids*)

Given that rids are used like record addresses, what would be a feasible *rid* generation method?

Page Formats

- Locating the page containing a given *rid* is not the whole story
 - **internal structure of pages** plays a crucial role
 - we think of a page as sequence of **slots**, each of which contains a record

Generating sensible record ids (*rids*)

Given that rids are used like record addresses, what would be a feasible *rid* generation method?

- ⇒ Generate a composite *rid* consisting of the address of page *p* and the placement (offset/slot) of *r* inside *p*

$\langle \text{pageNo } p, \text{slotNo } r \rangle$

Record Ids in Commercial Systems

The Real World

- ↳ **IBM DB2, Oracle 8, and Microsoft SQL Server**
 - record id is implemented as a **page id** and a **slot number**

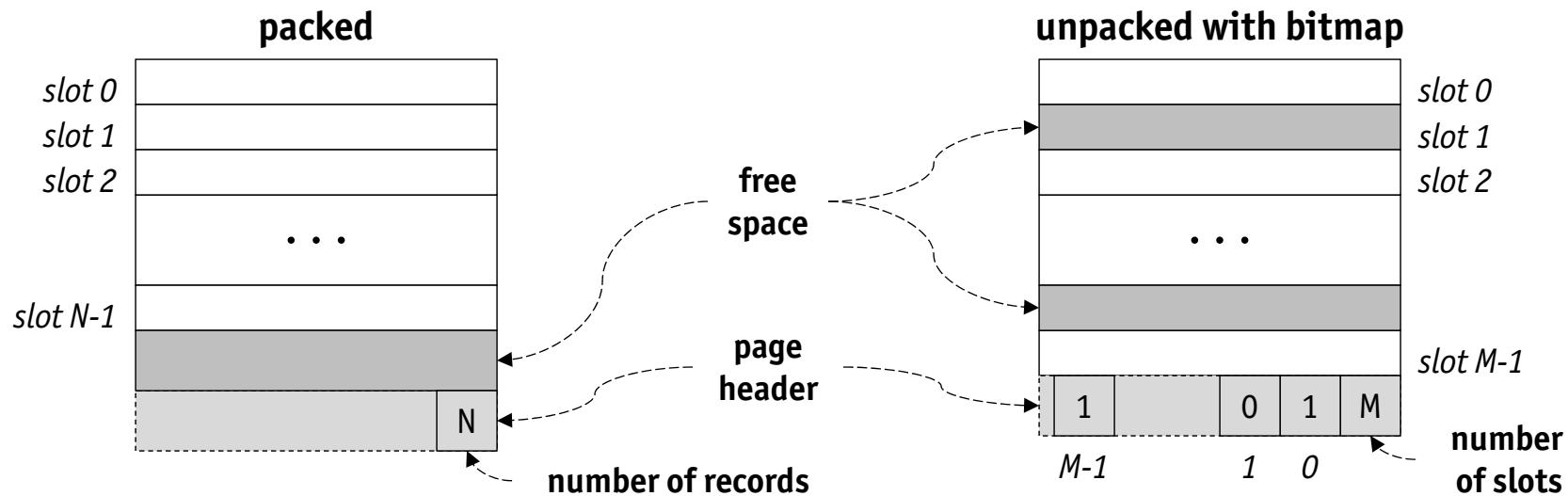
Fixed-Length Records

- All records on the page (in the file) are the **same size** s
 - **getRecord** ($f, \langle p, n \rangle$) : given the $rid \langle p, n \rangle$ we know that the record is to be found at (byte) offset $n \times s$ on page p
 - **deleteRecord** ($f, \langle p, n \rangle$) : copy the bytes of the last occupied slot on page p to offset $n \times s$, mark slot as free (page is **packed**, i.e., all occupied slots appear together at the start of the page)
 - **insertRecord** (f, r) : find a page p with free space $\geq s$ (see previous discussion) and copy r to the first free slot on p , then mark the slot as occupied

☛ Packed pages and deletions

One problem with packed pages remains as calling **deleteRecord** ($f, \langle p, n \rangle$) modifies the rid of a **different record** $\langle p, n' \rangle$ on the same page
☛ if any external references to this record exist, we need to chase through the whole database and **update rid** references $\langle p, n' \rangle \rightarrow \langle p, n \rangle$... **Bad!**

Free Slot Bitmap



- Avoid record copying and therefore rid modifications
 - `deleteRecord(f, <p, n>)` simply needs to set bit *n* in bitmap to 0
 - **no other rids affected**

✍ Page header or trailer?

In both page organization schemes, we have positioned the page header **at the end of the page**. How would you justify this design decision?

Inserting Fixed-Length Records

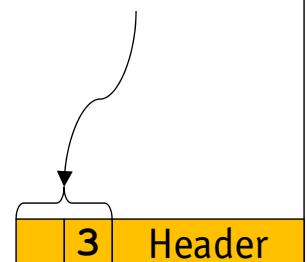
Creating a simple table in SQL

```
CREATE TABLE Persons (
    ID      INTEGER,
    NAME   CHAR(10),
    SEX    CHAR(1)
);
```

ID	NAME	SEX
2112	Walter.....	M
3927	Jesse.....	M
9453	Skyler.....	F

2	1	1	2	W	a	l	t	e	r					M	3	9	2	7	J	
e	s	s	e							M	9	4	5	3	S	k	y	1	e	r
				F																

number of records
in this page



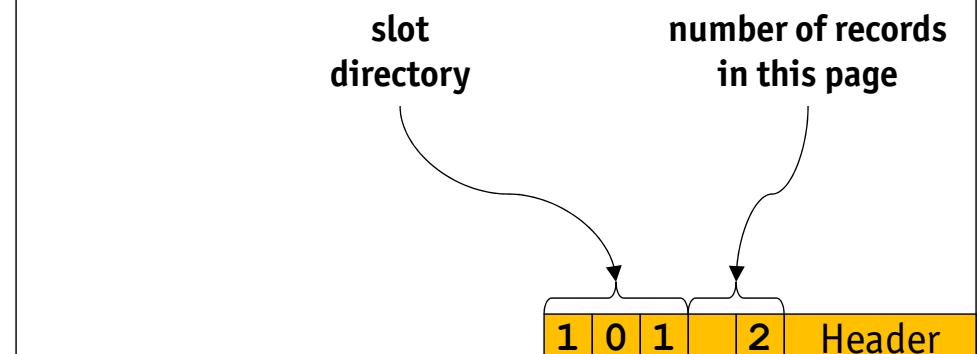
Deleting Fixed-Sized Records

Creating a simple table in SQL

```
CREATE TABLE Persons (
    ID      INTEGER,
    NAME   CHAR(10),
    SEX    CHAR(1)
);
```

ID	NAME	SEX
2112	Walter.....	M
3927	Jesse.....	M
9453	Skyler.....	F

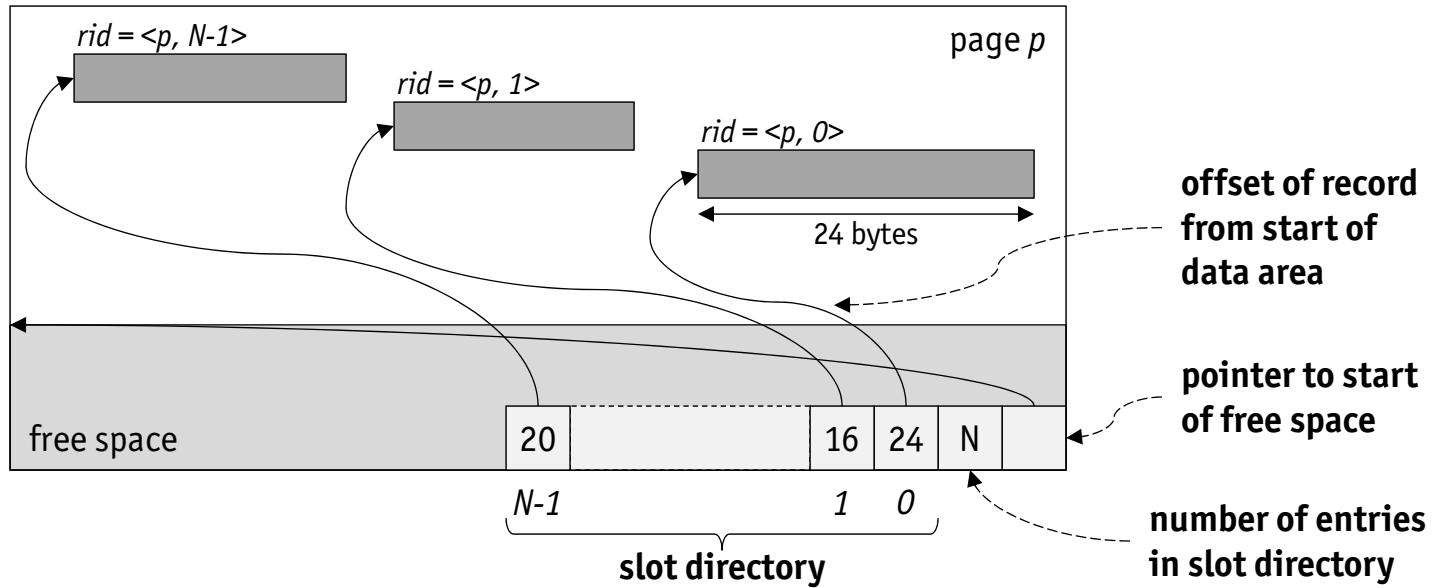
2	1	1	2	W	a	l	t	e	r					M	3	9	2	7	J
e	s	s	e							M	9	4	5	3	S	k	y	1	e
				F															



Variable-Length Records

- If records on a page are of varying size (e.g., SQL data type **VARCHAR (n)**), there can be **page fragmentation**
 - **insertRecord (f, r)** : needs to find an empty slot of size $\geq |r|$, such that the wasted space is **minimal**
 - compacting the remaining records to maintain a **contiguous area of free space** gets rid of holes produced by **deleteRecord (f, rid)**
- A solution is to maintain a **slot directory** on each page
 - similar free space management in a database heap file
 - contains entries $\langle \text{offset}, \text{length} \rangle$, where offset is measured in bytes from the start of data page
 - **deleteRecord ($f, \langle p, n \rangle$)** : set offset of directory entry n to -1 to indicate that entry can be reused subsequent **insertRecord (f, r)** calls, which hit page p

Variable-Length Records



Compaction of slot directory

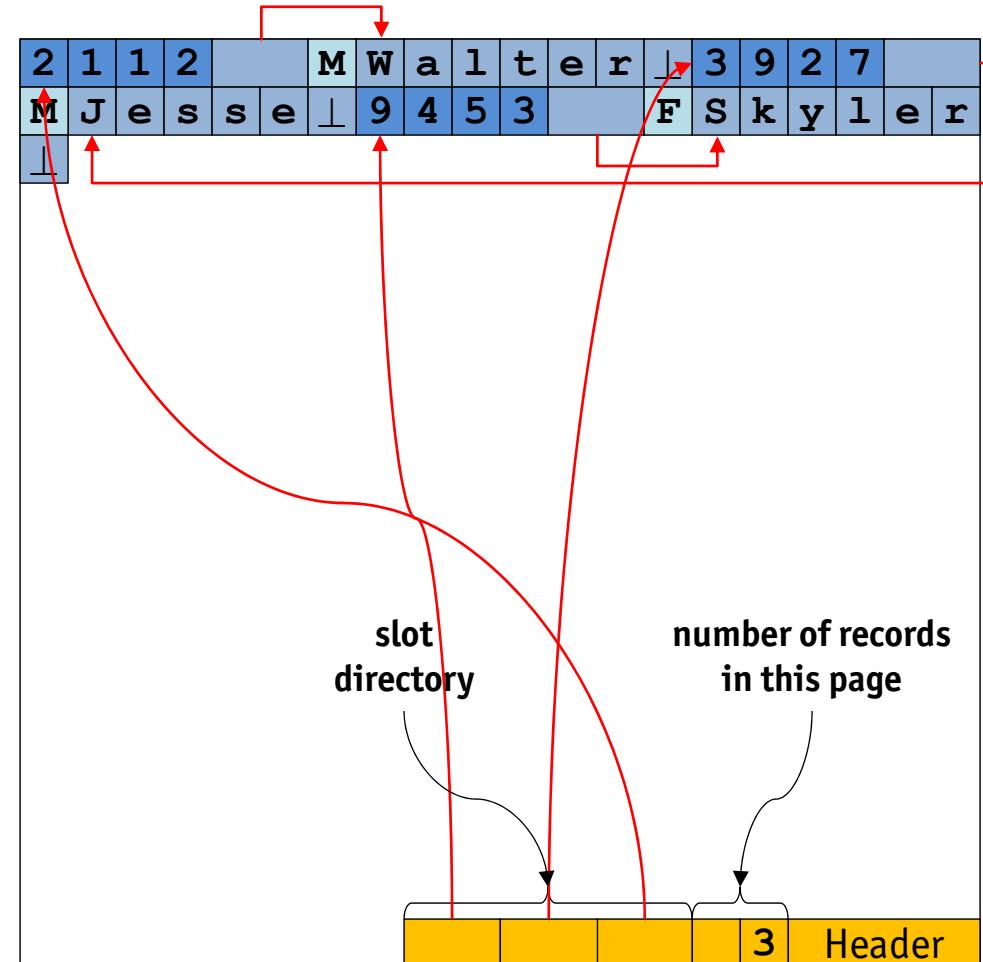
- **not allowed** in this scheme as this would again modify the rids of all records $\langle p, n' \rangle$, for $n' > m$
- if insertion are much more common than deletions, the directory size will nevertheless be **close to the actual number of records** stored on the page
- **record compaction** (defragmentation) is performed, of course

Inserting Variable-Length Records

Creating a simple table in SQL

```
CREATE TABLE Persons (
    ID      INTEGER,
    NAME   VARCHAR(10),
    SEX    CHAR(1)
);
```

ID	NAME	SEX
2112	Walter	M
3927	Jesse	M
9453	Skyler	F

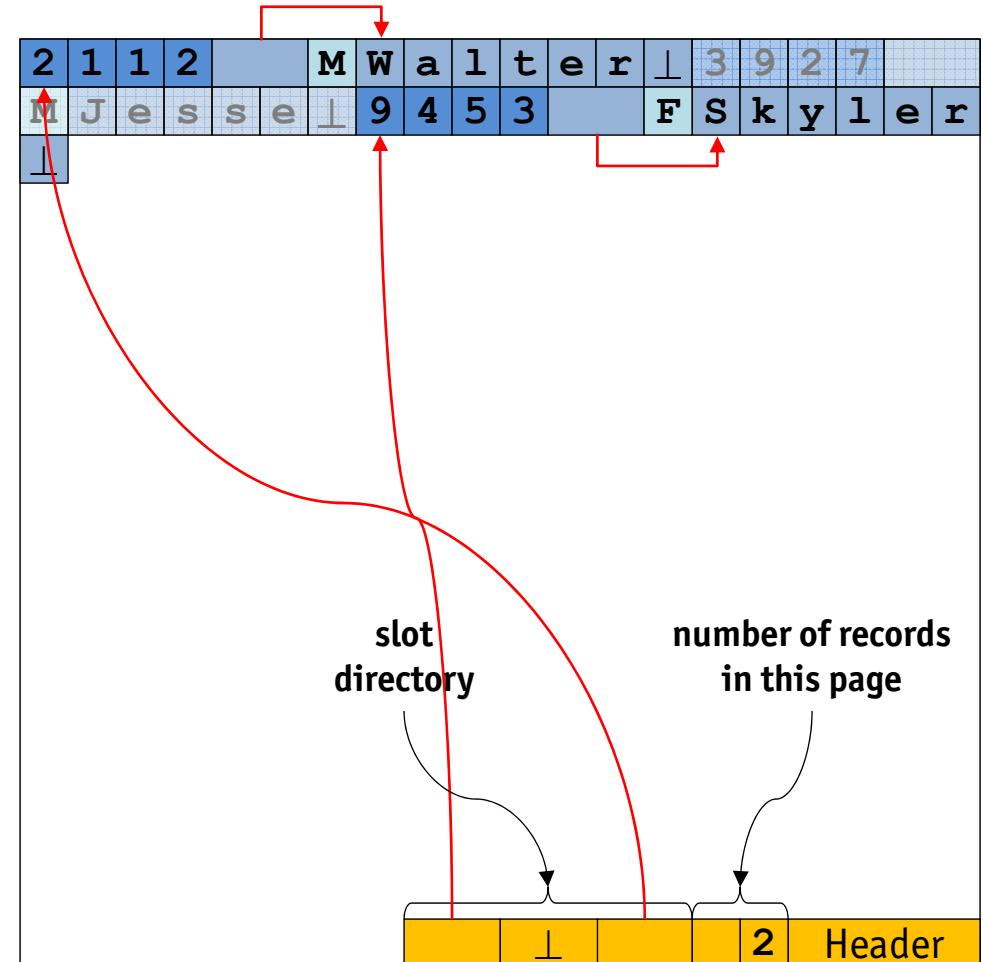


Deleting Variable-Length Records

Creating a simple table in SQL

```
CREATE TABLE Persons (
    ID      INTEGER,
    NAME   VARCHAR(10),
    SEX    CHAR(1)
);
```

ID	NAME	SEX
2112	Walter⊥	M
3927	Jesse⊥	M
9453	Skyler⊥	F

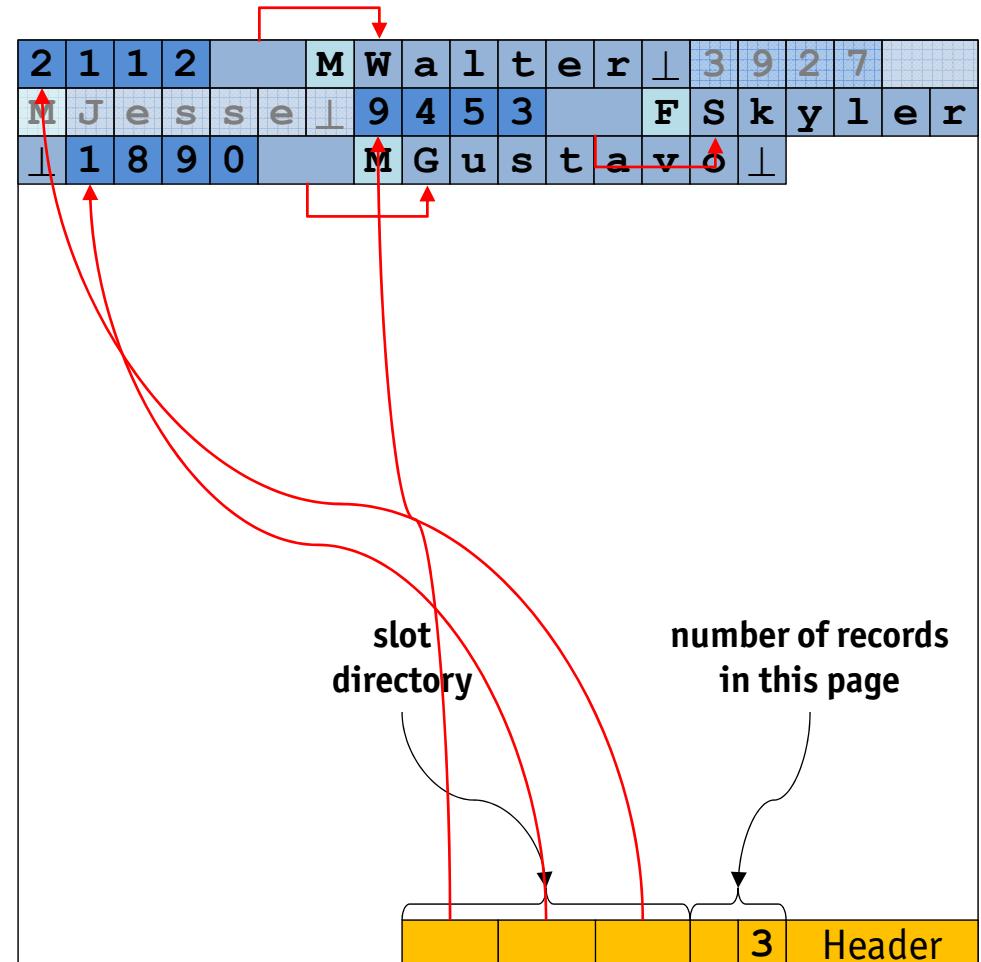


Inserting Variable-Length Records

Creating a simple table in SQL

```
CREATE TABLE Persons (
    ID      INTEGER,
    NAME   VARCHAR(10),
    SEX    CHAR(1)
);
```

ID	NAME	SEX
2112	Walter⊥	M
3927	Jesse⊥	M
9453	Skyler⊥	F
1890	Gustavo⊥	M



Record Formats

- Another focus change
 - **the road so far:** accessing records in a page
 - **now:** accessing fields (conceptually, attributes) in a record
- **Recall:** attributes are considered to be atomic in an RDBMS
- Record field type defines their **length**
 - **fixed-length**, e.g., **INTEGER**, **BIGINT**, **CHAR** (*n*) , **DATE**, ...
 - **variable-length**, e.g., **VARCHAR** (*n*) , **CLOB** (*n*) , ...
- On **CREATE TABLE**
 - DBMS computes field size information for the records of a file
 - thin information is then recorded in the **system catalog**

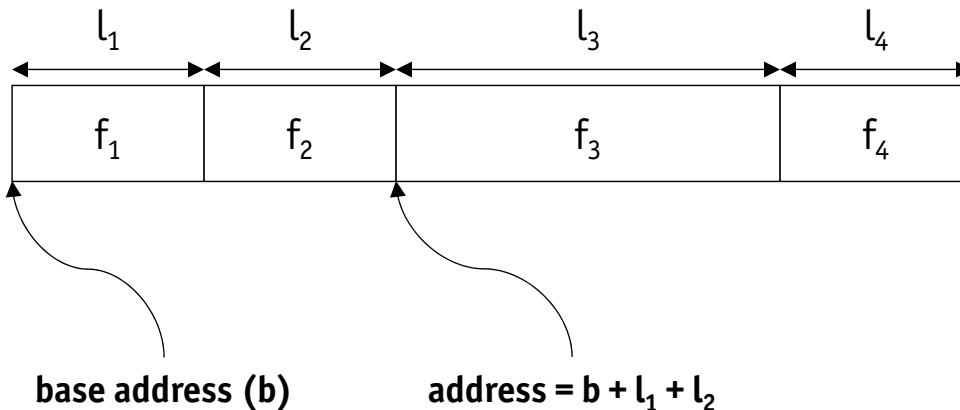
Record Field Sizes

The Real World

↳ Microsoft SQL Server

- **exact numerics**
`BIGINT` (8 bytes), `INT` (4 bytes), `SMALLINT` (2 bytes), `TINYINT` (1 byte)
- **approximate numerics**
`FLOAT (n)` (4-8 bytes), `REAL` (4 bytes)
- **date and time**
`DATE` (3 bytes), `TIME` (5 bytes), `DATETIME` (8 bytes)
- **character strings**
`CHAR (n)` (1-8,000 bytes), `VARCHAR (n)` (1-8,000 bytes, 2 GB for n = `max`),
`NTEXT` (1-2,147,483,647 bytes)
- **binary strings**
`BINARY (n)` (1-8000 bytes), `VARBINARY (n)` (1-8,000 bytes, 2 GB for
n = `max`), `IMAGE` (1-2,147,483,647 bytes)

Fixed-Length Fields



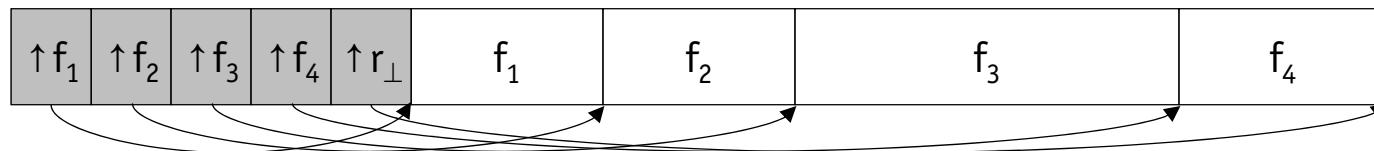
- Fixed-length record: each field has a **fixed length** and the number of fields is also **fixed**
 - fields can be stored **consecutively**
 - given the address of the record (b), the address of a particular field can be calculated using information about **lengths of preceding fields** (l_i)
 - this information is available from the DBMS **system catalog**

Variable-Length Fields

- Multiple variants exist to store records that contain variable-length fields
 - use a special **delimiter symbol** (\$) to separate record fields: accessing field f_n requires a scan over the bytes of fields $f_1 \dots f_{n-1}$

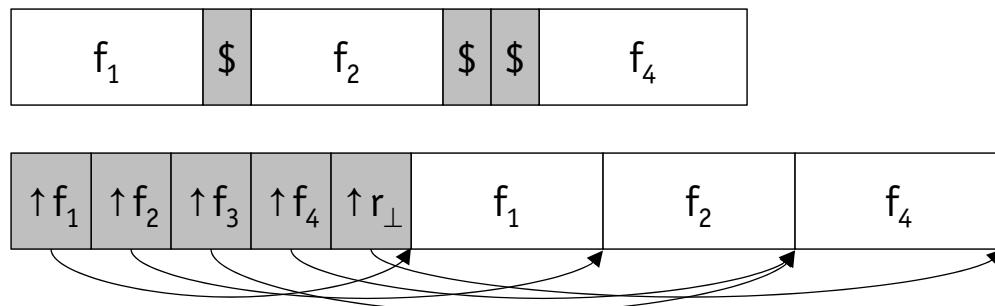


- for a record of n fields, use an **array** of $n + 1$ offsets pointing into the record (the last array entry marks the end of field f_n)



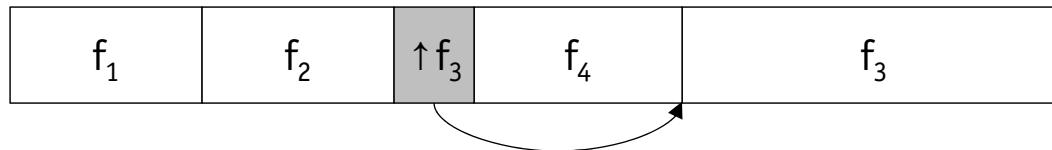
Delimiter vs. Array

- Array approach is typically superior
 - array overhead translates to **direct access** to any field
 - clean and compact way to deal with **null values (NULL in SQL)** by simply comparing pointers to **beginning** and **end** of field



Record Formats

- Another popular record format to support variable-length fields is a **combination** of the delimiter and array approach
 - variable-length fields are stored **at the end of the record**
 - fixed-length fields and pointers to variable-length fields are stored sequentially, starting **at the beginning of the record**



- Note that it may even make sense to use variable-length records to store fixed-length records
 - support for null values (see above)
 - schema evolution, i.e., adding or removing columns

Modifying Variable-Length Fields

Growing a record

Modifying a variable-length field may cause it to grow! How could the DBMS file manager cope with the following cases?

- 1. the updated record still fits on the page**

- 2. the updated record does not fit on the page anymore**

- 3. the updated record does not fit on any other page**

Modifying Variable-Length Fields

Growing a record

Modifying a variable-length field may cause it to grow! How could the DBMS file manager cope with the following cases?

1. the updated record still fits on the page

- ↳ all subsequent records must be shifted to make space for the modification

2. the updated record does not fit on the page anymore

- ↳ modified record must be moved to another page with enough space
- ↳ moving a record can cause problems as its *rid* will change
- ↳ must leave a “forwarding address” to the new location

3. the updated record does not fit on any other page

- ↳ modified record must be broken into smaller records
- ↳ smaller records can then be stored on pages with enough space
- ↳ smaller records are chained together across pages to enable retrieval

IBM DB2 Data Pages

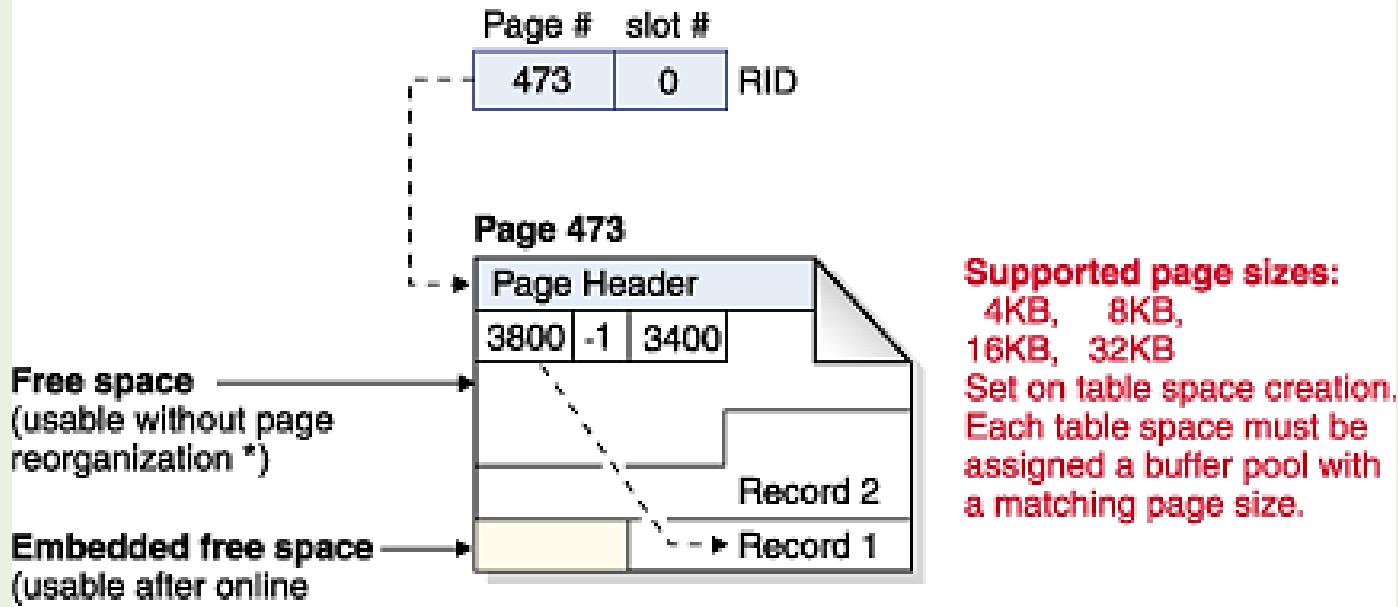
The Real World

- Support for **4 kB, 8 kB, 16 kB, and 32 kB data pages** in separate table spaces, buffer manager pages match in size
- **68 bytes** of database manager overhead per page, e.g., on a 4 kB page
 - **maximum user data:** 4,028 bytes
 - **maximum record size:** 4,005 bytes
- Records do **not** span pages
- **Maximum table size:** 512 GB (with 32 kB pages)
- **Maximum number of columns:** 1,012 (500 on a 4 kB page)
- **Maximum number of rows per page:** 255
- Columns of type **LONG VARCHAR, CLOB**, etc. maintained **outside** regular data pages, which contain descriptors only
- **Free space management:** first-fit order
 - free space map distributed on every 500th page in **free space control records**
 - records updated in-place if possible, otherwise uses **forwards**

IBM DB2 Data Pages

The Real World

Data page and RID format

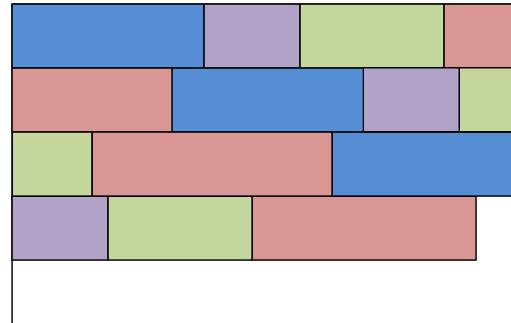
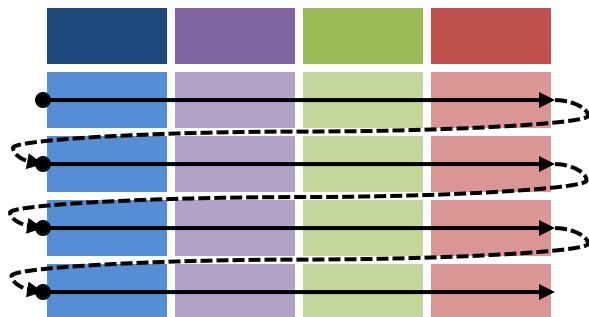


* Exception: Any space reserved by an uncommitted DELETE is not usable.

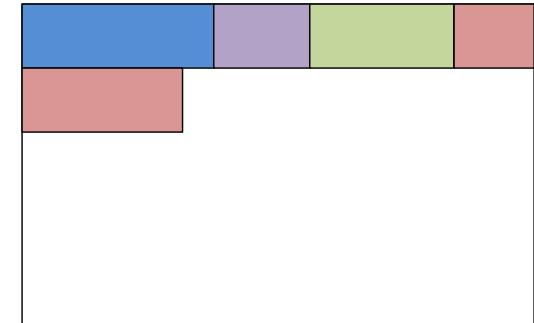
Picture Credit: <http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp>

Alternative Page Layouts

- So far, we populated data pages in **row-wise** order

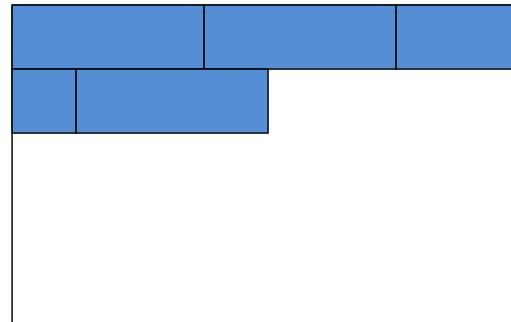
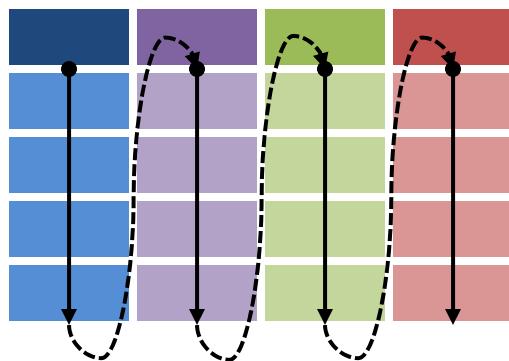


page 0

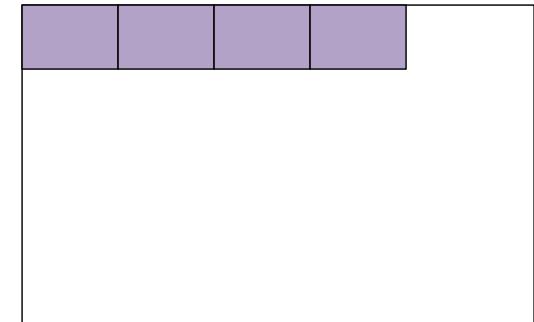


page 1

- We could also populate data pages in **column-wise** order



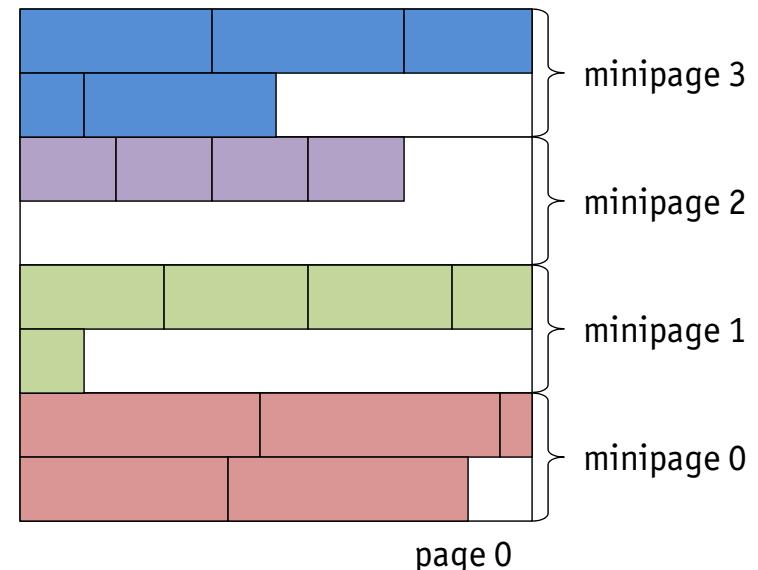
page 0



page 1

Alternative Page Layouts

- These two approaches are also known as **n-ary Storage Model (NSM)** and **Decomposition Storage Model (DSM)**
 - tuning known for certain workload types, e.g., OLAP
 - suitable for narrow projections and in-memory database systems
 - different behavior with respect to compression
 - a.k.a. **row-store** and **column-store**
- A hybrid approach is the **Partition Attributes Across (PAX)** layout
 - divide each page into **minipages**
 - group attributes into them



Addressing Schemes

- Criteria for “good” record ids (*rids*)
 - given an *rid*, it ideally takes **no more than one page I/O operation** to get the record itself
 - *rids* should **stable** under all circumstances, e.g., when a record is being **moved within a page or across pages**

 Why are these goals important to achieve?

 These goals are conflicting. Explain!

Addressing Schemes

- Criteria for “good” record ids (*rids*)
 - given an *rid*, it ideally takes **no more than one page I/O operation** to get the record itself
 - *rids* should **stable** under all circumstances, e.g., when a record is being **moved within a page or across pages**

Why are these goals important to achieve?

Consider the fact that rids are used as “persistent pointers” in a DBMS (indexes, directories, etc.). Changing the rid when a record is moved would leave a dangling pointer or require a lot of maintenance overhead.

These goals are conflicting. Explain!

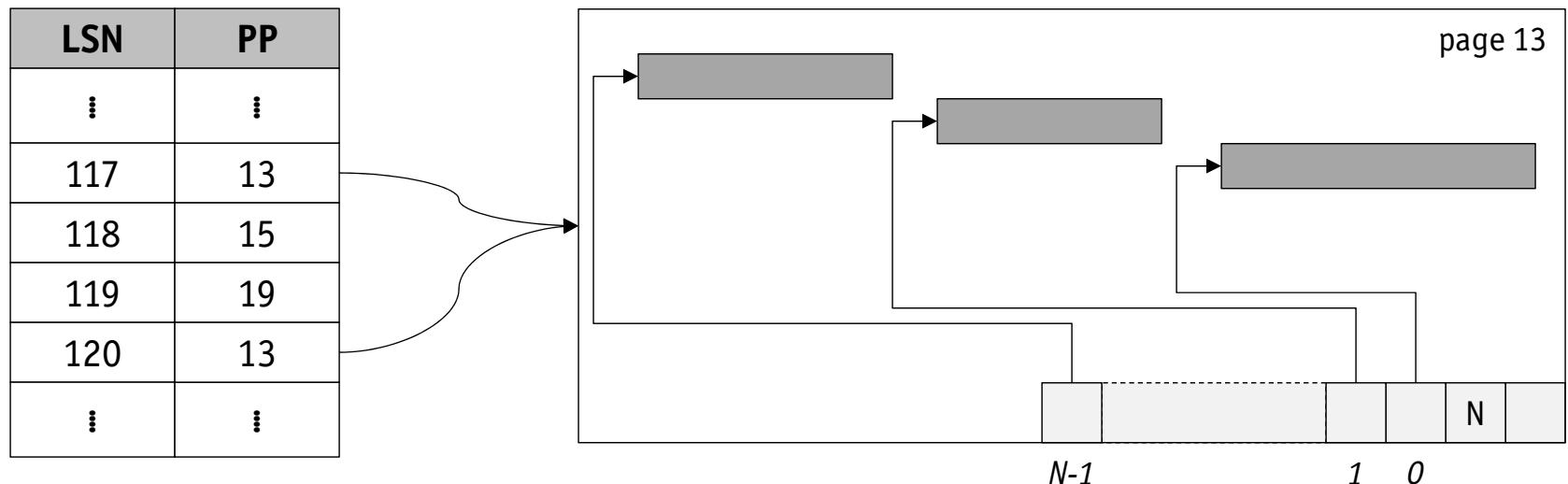
- **Efficiency** calls for “direct disk access”
- **Stability** calls for some kind of indirection

Direct Addressing

- **Relative Byte Address (RBA)**: think of a disk file as a persistent virtual address space and use byte-offset as rid
 - ⊕ very efficient access to pages and records within pages
 - ⊖ no stability at all w.r.t. to moving records
- **Page Pointers (PP)**: use disk page numbers as rid
 - ⊕ very efficient access to page, locating records within a page is also cheap (in-memory operation)
 - ⊖ stable w.r.t. to moving record within a page, but not when moving records across pages

Indirect Addressing

- **Logical Sequence Numbers (LSN):** assign logical numbers to records and use address translation table to map LSN to PP (or even RBA)
 - ⊕ full stability w.r.t. all relocations of records
 - ⊖ additional I/O operation to translation table (often in the buffer)

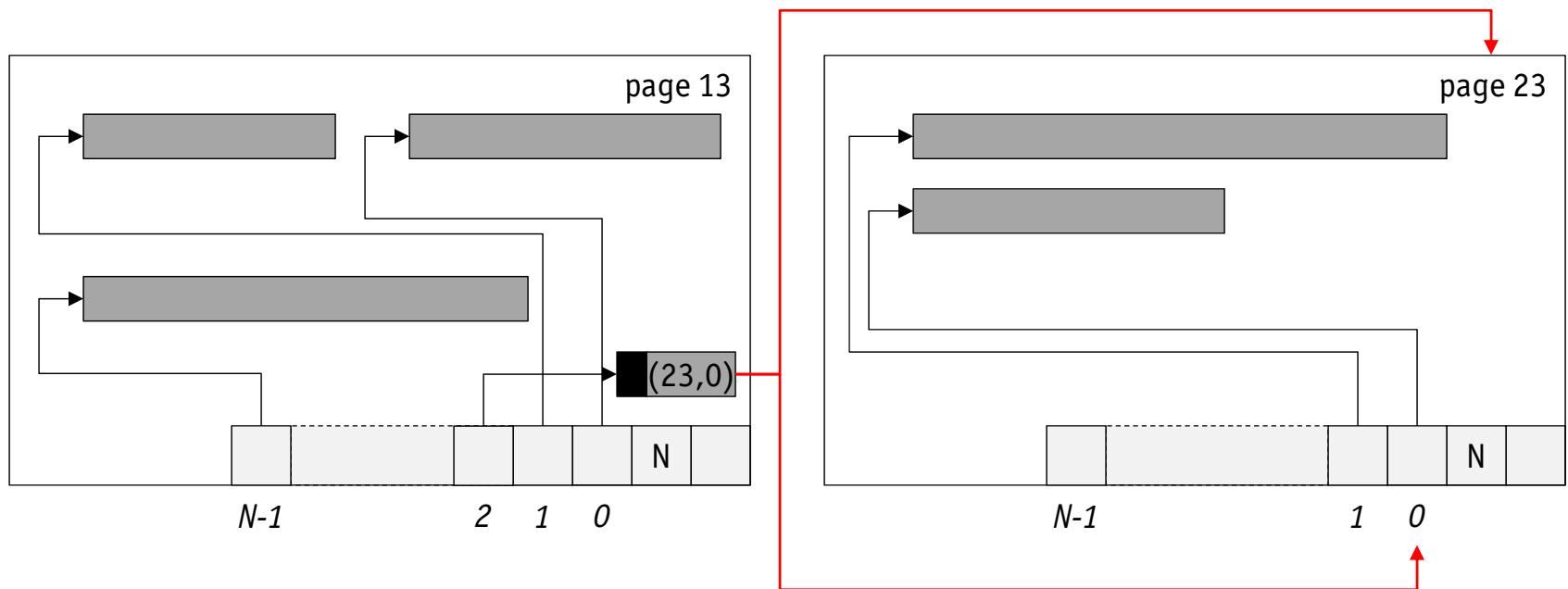


Fancy Indirect Addressing

- **LSN with Probable Page Pointers (LSN/PPP):** try to avoid extra I/O operations by adding a “probable” PP (PPP) to LSN, where PPP is PP at the time of insertion into database (if record is moved across pages, PPP is **not** updated)
 - ⊕ full stability w.r.t. all record relocations and PPP can save extra I/O operation, if still correct
 - ⊖ two additional page I/O operations if PPP is no longer valid: need to read “old” page to notice that record has moved, then need to read translation table to lookup new page number

Recall Initial Addressing Scheme

- Tuple Identifier (TID): use $\langle \text{pageNo}, \text{slotNo} \rangle$ pair as rid
 - slotNo is an index in a page-local offset array
 - this guarantees stability w.r.t. relocation within a page
 - to guarantee stability w.r.t. relocation across pages, leave a **forwarding address** on original page



Tuple Identifier Addressing Scheme

✍ But what happens when the record is moved again?

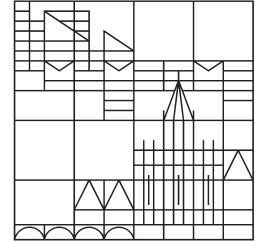
- Discussion
 - ⊕ full stability w.r.t. all relocations of records, no extra I/O operations due to indirection
 - ⊖ only one additional page I/O operation in case of forward pointer on original page
- Therefore, most DBMS use this addressing scheme!

Tuple Identifier Addressing Scheme

 But what happens when the record is moved again?

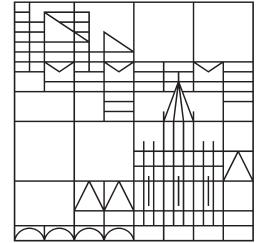
Avoid chains of forwarding addresses! Instead of leaving another forwarding address, update the forwarding address on the original page.

- Discussion
 - ⊕ full stability w.r.t. all relocations of records, no extra I/O operations due to indirection
 - ⊖ only one additional page I/O operation in case of forward pointer on original page
- Therefore, most DBMS use this addressing scheme!



Database System Architecture and Implementation

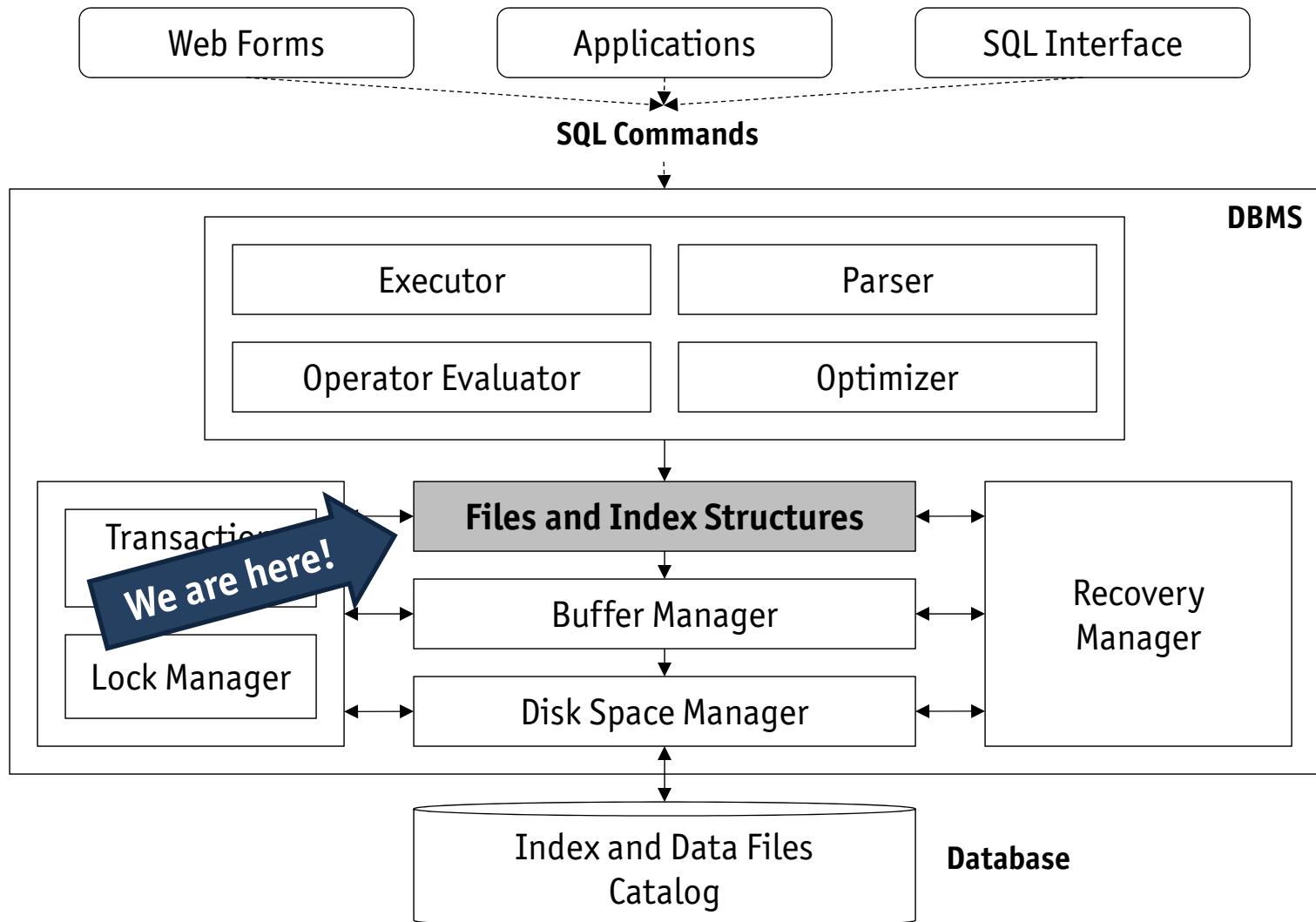
TO BE CONTINUED...



Database System Architecture and Implementation

Module 2
File Organizations and Indexing
November 11, 2013

Orientation



Recall Heap Files

- Heap files provide **just enough** structure to maintain a collection of records (of a table)
- The heap file **supports sequential (`openScan(·)`)** over the collection

 SQL query leading to a sequential scan

```
SELECT A, B  
FROM   R
```

- **No other operations** get specific support from heap files

Systematic File Organization

SQL queries calling for systematic file organization

```
SELECT A, B  
FROM R  
WHERE C > 45
```

```
SELECT A, B  
FROM R  
ORDER BY C ASC
```

- For the above queries, it would definitely be helpful if the SQL query processor could rely on a particular file organization of the records in the file for table **R**

Exercise

Which organization of records in the file for table **R** could speed up the evaluation of **both** queries above?

Systematic File Organization

SQL queries calling for systematic file organization

```
SELECT A, B  
FROM R  
WHERE C > 45
```

```
SELECT A, B  
FROM R  
ORDER BY C ASC
```

- For the above queries, it would definitely be helpful if the SQL query processor could rely on a particular file organization of the records in the file for table **R**

Exercise

Which organization of records in the file for table **R** could speed up the evaluation of **both** queries above?

- Allocate records of table **R** in ascending order of attribute **C** values
- Place records in neighboring pages
- (Only include columns **A**, **B**, and **C** in the records)

Module Overview

- Three different **file organizations**
 1. files containing **randomly ordered** records (heap files)
 2. files **sorted** on one or more record fields
 3. files **hashed** on one or more record fields
- Comparison of file organizations
 - simple **cost model**
 - application of cost model to **file operations**
- Introduction to **index** concept
 - clustered vs. unclustered indexes
 - dense vs. sparse indexes

Comparison of File Organizations

- Competition of three file organizations in five disciplines
 1. **scan**: read all records in a give file
 2. **search with equality test**
 3. **search with range selection** (upper or lower bound may be unspecified)
 4. **insert** a given record in the file, respecting the file's organization
 5. **delete** a record (identified by its *rid*), maintain the file's organization

SQL queries calling for equality test and range selection support

```
SELECT *
FROM   R
WHERE  C = 45
```

```
SELECT *
FROM   R
WHERE  A > 0 AND A < 100
```

Simple Cost Model

- A **cost model** is used to analyze the execution time of a given database operations
 - **block I/O** operations are typically a major cost factor
 - **CPU time** to account for searching inside a page, comparing a record field to selection constant, etc.
- To **estimate** the execution time of the five database operation, we introduce a **coarse** cost model
 - omits cost of network access
 - does not consider cache effects
 - neglects burst I/O
 - ...
- Cost models play an important role in **query optimization**

Simple Cost Model

Simple cost model parameters

Parameter	Description
b	number of pages in the file
r	number of records on a page
D	time to read/write a disk page
C	CPU time needed to process a record (e.g., compare a field value)
H	CPU time take to apply a function to a record (e.g., a comparison or hash function)

- Some typical values
 - $D \approx 15$ ms
 - $C \approx H \approx 0.1$ μ s

Back to the Future

☛ A simple hash function

A **hashed file** uses a **hash function** h to map a given record onto a specific page of the file.

Example: h uses the lower 3 bits of the first field (of type **INTEGER**) of the record to compute the corresponding page number.

$h(\langle 42, \text{true}, \text{'dog'} \rangle)$	\rightarrow	2	$(42 = 101010_2)$
$h(\langle 14, \text{true}, \text{'cat'} \rangle)$	\rightarrow	6	$(14 = 1110_2)$
$h(\langle 26, \text{false}, \text{'mouse'} \rangle)$	\rightarrow	2	$(26 = 11010_2)$

- The hash function determines the page number only, record placement inside a page is not prescribed
- If a page p is filled to capacity, a chain of overflow pages is maintained to store additional records with $h(\langle \dots \rangle) = p$
- To avoid immediate overflowing when a new record is inserted, pages are typically filled to 80% only when a heap file is initially (re)organized into a hash file

Cost of Scan

1. Heap file

Scanning the records of a file involves **reading all b pages** as well as **processing each of the r records on each page**.

$$Scan_{heap} = b \cdot (D + r \cdot C)$$

2. Sorted file

The sort order **does not help** here. However, the scan retrieves the record in sorted order (which can be a big plus for subsequent operators).

$$Scan_{sort} = b \cdot (D + r \cdot C)$$

3. Hashed file

The hash function **does not help**. We simply scan from the beginning (skipping over the spare free space typically found in hashed files).

$$Scan_{hash} = \underbrace{(100/80)}_{=1.25} \cdot b \cdot (D + r \cdot C)$$

Hashed File

Scanning a hashed file

In which order does a scan of a hashed file retrieve its records?

Cost of Search with Equality Test

1. Heap file

The equality test is Ⓐ on a *primary key* or Ⓑ *not on a primary key*.

- Ⓐ $\text{Search}_{\text{heap}} = \frac{1}{2} \cdot b \cdot (D + r \cdot C)$
- Ⓑ $\text{Search}_{\text{heap}} = b \cdot (D + r \cdot C)$

2. Sorted file (sorted on A)

We assume the equality test is on **A**. The sort order enables the use of **binary search**.

$$\text{Search}_{\text{sort}} = \log_2 b \cdot D + \log_2 r \cdot C$$

If more than one record qualifies, all other matches are stored **right after** the first hit.

 **Nevertheless, no DBMS will implement binary search for value lookup**

Why?

Cost of Search with Equality Test

3. Hashed file (hashed on A)

We assume the equality test is on **A**. **Hashed files support equality searching best.** The hash function directly leads to the page containing the hit (overflows chains are ignored here).

The equality test is ① on a *primary key* or ② *not on a primary key*

① $Search_{hash} = H + D + \frac{1}{2} \cdot r \cdot C$

② $Search_{hash} = H + D + r \cdot C$

Note that there is **no dependence** on the file size b here. All qualifying records live on the same page or, if present, in its overflow pages.

Cost of Search with Range Selection

1. Heap file

Qualifying records can appear anywhere in the file.

$$Range_{heap} = b \cdot (D + r \cdot C)$$

2. Sorted file (sorted on **A**)

Use equality search with **A** = *lower*, then sequentially scan the file until a record with **A** > *upper* is found.

$$Range_{sort} = \log_2 b \cdot D + \log_2 r \cdot C + \lfloor \frac{n}{r} \rfloor \cdot D + n \cdot C$$

where *n* denotes the number of hits in the range.

3. Hashed file (hashed on **A**)

Hashing offers no help as hash functions are designed to scatter records all over the hashed file, e.g., $h(\langle 7, \dots \rangle) = 7, h(\langle 8, \dots \rangle) = 0$.

$$Range_{hash} = 1.25 \cdot b \cdot (D + r \cdot C)$$

Cost of Insert

1. Heap file

The record can be added to some arbitrary page, e.g., the last page. This involves reading and writing the page.

$$Insert_{heap} = 2 \cdot D + C$$

2. Sorted file

On average, the new record will belong in the middle of the file. Then, all subsequent records in the latter half of the file must be shifted.

$$Insert_{sort} = \underbrace{\log_2 b \cdot D + \log_2 r \cdot C}_{\text{search}} + \underbrace{\frac{1}{2} \cdot b \cdot (2 \cdot D + r \cdot C)}_{\text{shift latter half}}$$

3. Hashed file

Search for the record, then read and write the page determined by the hash function (assume spare 20% space is sufficient to hold new record)

$$Insert_{hash} = \underbrace{H + D}_{\text{search}} + C + D$$

Cost of Delete

1. Heap file

Assume that the file is not compacted after the record is found and removed, i.e., the file uses free space management.

$$Delete_{heap} = \underbrace{D}_{\substack{\text{search} \\ \text{by rid}}} + C + D$$

2. Sorted file

Access the record's page and then (on average) shift the latter half of the file to compact it.

$$Delete_{sort} = D + \underbrace{\frac{1}{2} \cdot b \cdot (2 \cdot D + r \cdot C)}_{\text{shift latter half}}$$

3. Hashed file

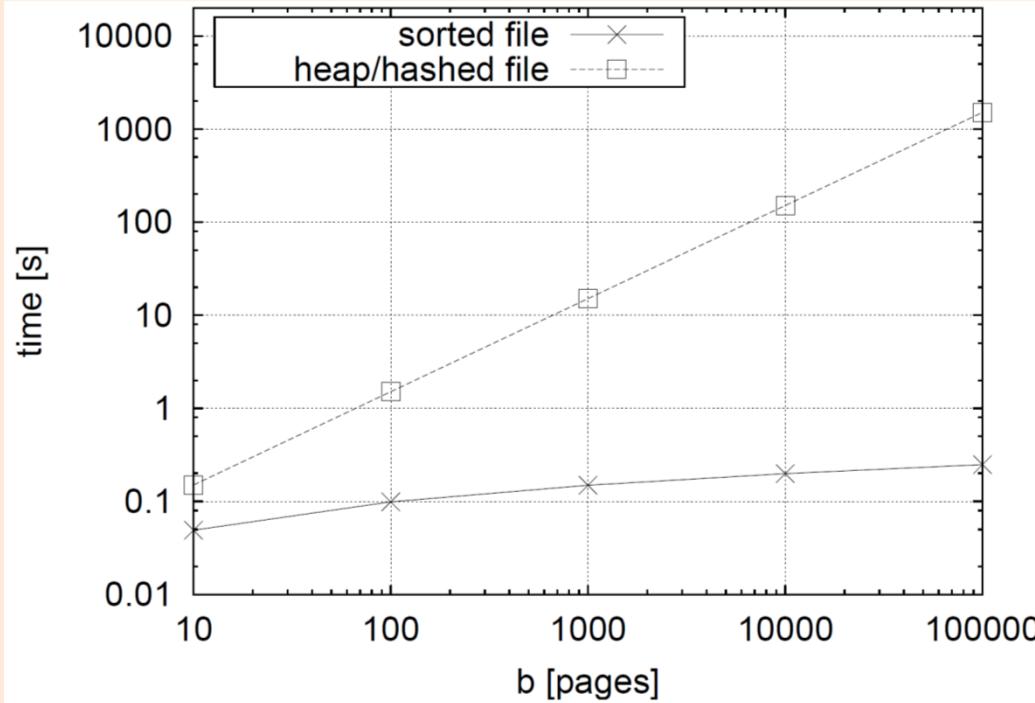
Page access by rid is faster than the hash function: same cost as heap file.

$$Delete_{hash} = D + C + D$$

Performance Comparison

- Performance of **range selections** for files of increasing size ($D = 15 \text{ ms}$, $C = 0.1 \mu\text{s}$, $r = 100$, $n = 10$)

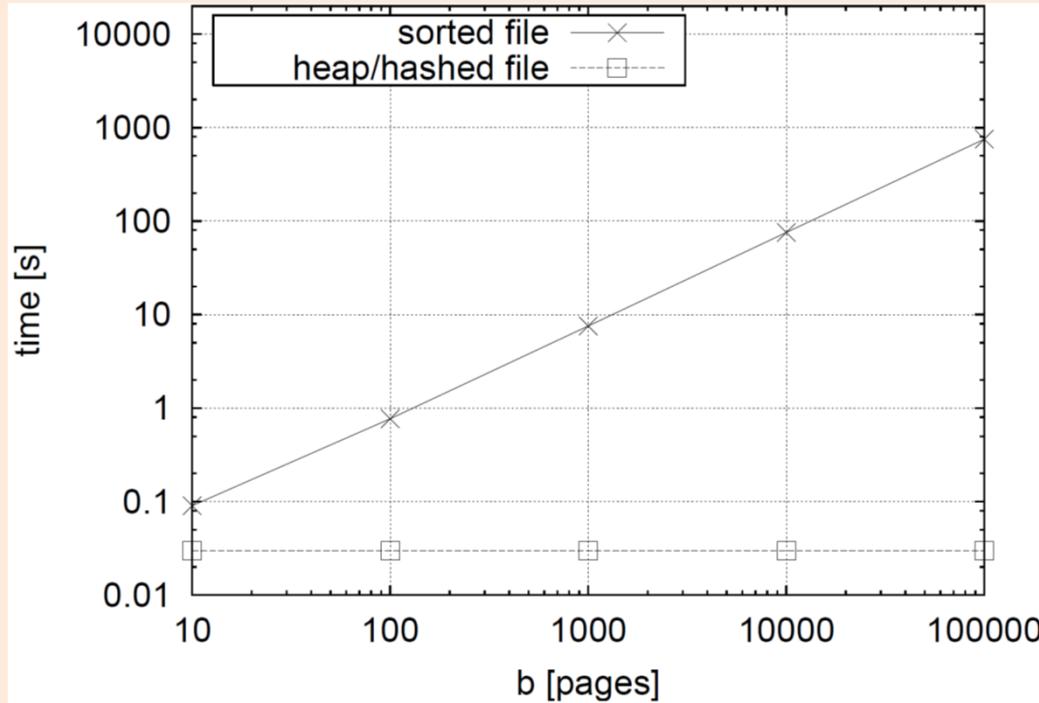
Performance graph



Performance Comparison

- Performance of **deletions** for files of increasing size
($D = 15 \text{ ms}$, $C = 0.1 \mu\text{s}$, $r = 100$, $n = 1$)

Performance graph



And the Winner Is...

- There is **no single file organization** that responds equally fast to all five operations
- This is a **dilemma** because more advanced file organizations can make a real difference in speed (see previous slides)
- There exist **index structures** which offer all advantages of a sorted file *and* support insertions/deletions efficiently (at the cost of a modest space overhead): **B+ trees**
- Before discussing B+ trees in detail, the following introduces the **index concept** in general

Indexes

- If the basic organization of a file does not support a specific operation, we can **additionally** maintain an auxiliary structure, an **index**, which adds the needed support

Example

```
SELECT A, B, C  
FROM   R  
WHERE  A > 0 AND A < 100
```

If the file for table **R** is sorted on **C**, it **cannot** be used to evaluate *Q* more efficiently. A solution is to add an index *that supports range queries* on **A**.

Indexes

- A DBMS uses indexes like **guides**, where each guide is specialized to accelerate searches on a specific attribute (or a combination of attributes) of the records in its associated file

☞ Usage of index on attribute **A**



1. Query index for the location of a record with **A** = k (k is the **search key**)
2. The index responds with an associated **index entry** k^*
(k^* contains enough information to access the actual record in the file)
3. Read the actual record by using the guiding information in k^* : the record will have an **A**-field with value k .

The Small Print: Only “exact match” indexes will return records that contain the value k for field **A**. In the more general case of “similarity” indexes, the records are not guaranteed to contain the value k , they are only candidates for having this value.

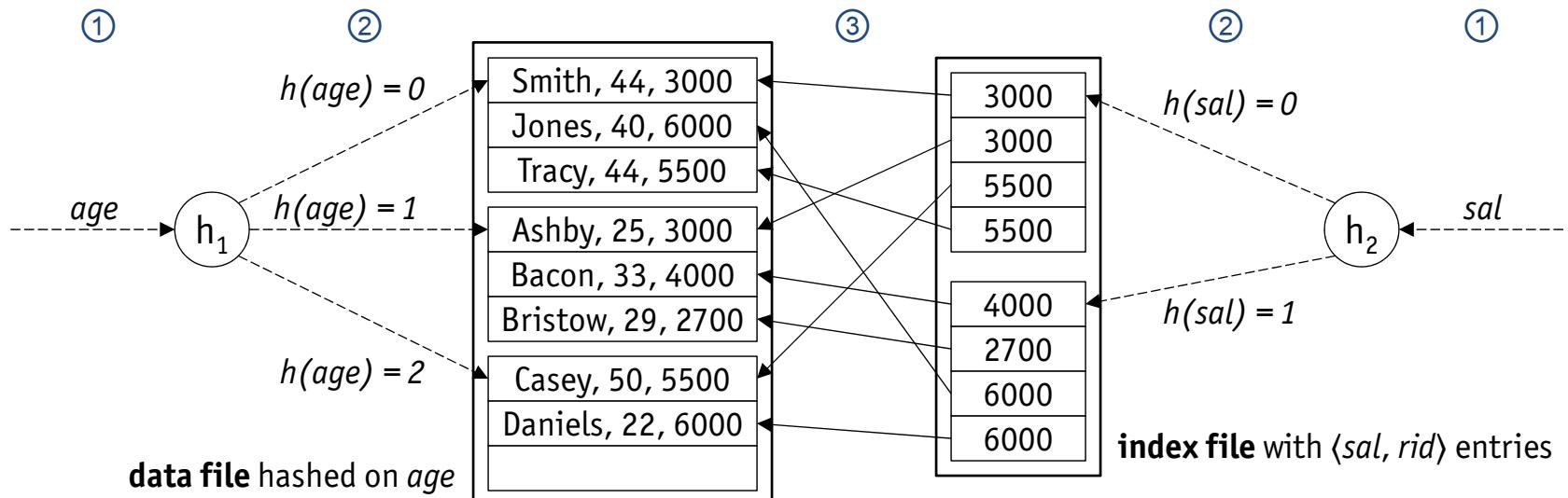
Index Entries

Index Entry Design

Variant	Index entry k^*
①	$\langle k, \langle \dots, \mathbf{A} = k, \dots \rangle \rangle$
②	$\langle k, rid \rangle$
③	$\langle k, [rid_1, rid_2, \dots] \rangle$

- Remarks
 - With variant ①, there is no need to store the data records in addition to the index—the index itself is a special file organization
 - If we build multiple indexes for a file, at most one of these should use variant ① to avoid redundant storage of records
 - Variants ② and ③ use $rid(s)$ to point into the actual data file
 - Variant ③ leads to fewer index entries if multiple records match a search key k , but index entries are of variable length

Index Example



- **Data file** contains $\langle \text{name}, \text{age}, \text{sal} \rangle$ records and is hashed on age, using hash function h_1 (index entry variant ①)
- **Index file** contains $\langle \text{sal}, \text{rid} \rangle$ index entries (variant ②), pointing to data file (hash function h_2)
- This file organization plus index **efficiently** supports equality searches on both key *age* and key *sal*

Clustered vs. Unclustered Indexes

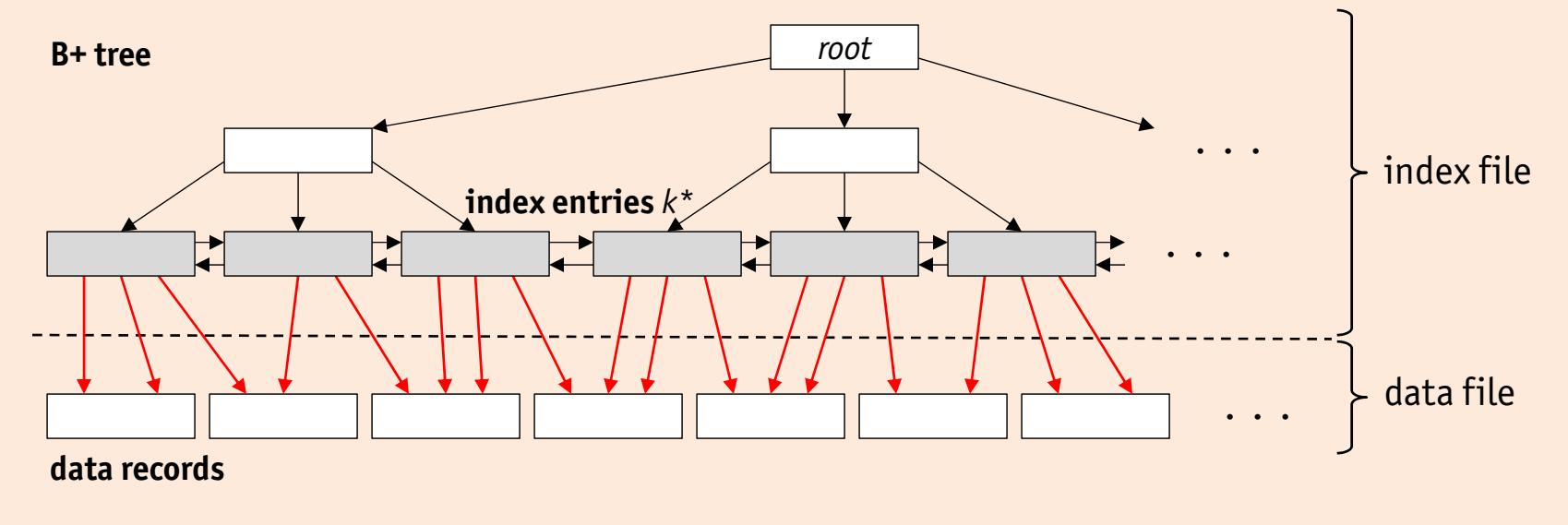
- Suppose, **range selections** such as $lower \leq A \leq upper$ need to be supported on records of a data file
- If an index on field **A** is maintained, range selection queries could be evaluated using the following algorithm
 1. **query the index** once for a record with field **A** = $lower$
 2. **sequentially scan the data file** from there until we encounter a record with field **A** > $upper$

 Name the assumption!

Which important assumption does the above algorithm make in order for this **switch from index to data file** to work efficiently?

Clustered vs. Unclustered Indexes

Index over data file with matching sort order



- Remark
 - in a B+ tree, for example, the index entries k^* stored in the leaves are sorted on the key k

Clustered vs. Unclustered Indexes

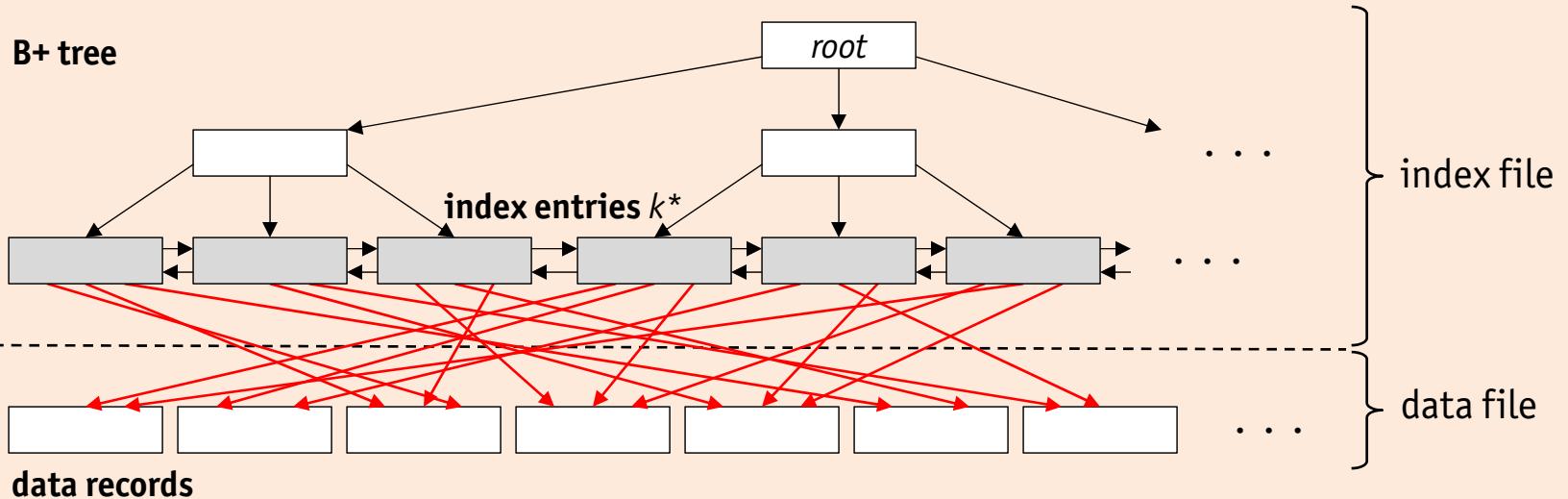
Definition: Clustered Index

If the **data file** associated with an index **is sorted on the index search key**, the index is said to be **clustered**

- In general, the cost for a range selection grows tremendously if the index on **A** is **unclustered**
 - proximity of index entries **does not** imply proximity of data records
 - as before, the index can be queried for a record with **A = lower**
 - however, to continue the scan it is necessary to **revisit the index entries**, which point to **data pages scattered** all over the data file
- Remarks
 - an index that uses entries k^* of variant ①, is clustered by definition
 - a data file can have at most one clustered index (but any number of unclustered indexes)

Clustered vs. Unclustered Indexes

Unclustered index



Clustered vs. Unclustered Indexes

Variant ① in Oracle 8i

```
CREATE TABLE ... ( ... PRIMARY KEY ( ... )) ORGANIZATION INDEX;
```

Clustered indexes in DB2

Create a clustered index **IXR** on table **R**, index key is attribute **A**

```
CREATE INDEX IXR ON R(A ASC) CLUSTER;
```

From the DB2 V9.5 manual

*"[CLUSTER] specifies that the index is **the** clustering index of the table. The **cluster factor** of a clustering index is maintained or improved dynamically as data is inserted into the associated table, by attempting to **insert new rows physically close to the rows for which the key values of this index are in the same range**. Only one clustering index may exist for a table so CLUSTER may not be specified if it was used in the definition of any existing index on the table (SQLSTATE 55012). A clustering index may not be created on a table that is defined to use append mode (SQLSTATE 428D8)."*

Clustered vs. Unclustered Indexes

Cluster a table based on an existing index in PostgreSQL

Reorganize the rows of table **R** so that their physical order matches the *existing* index **IXR**

```
CLUSTER R USING IXR;
```

- If **IXR** indexes attribute **A** of **R**, rows will be sorted in ascending **A** order
- Range queries will touch less pages, which additionally, will be physically adjacent
- **Note:** Generally, future insertions will compromise the perfect **A** order
 - may issue **CLUSTER R** again to re-cluster
 - in **CREATE TABLE**, use **WITH (fillfactor=f)**, $f \in 10\dots100$, to reserve space for subsequent insertions
- The SQL-92 and SQL-99 standard do not include any statement for the specification (creation, dropping) of index structures
 - SQL **does not even require** SQL systems to provide indexes at all!
 - almost all SQL implementations support one or more kinds of indexes

Dense vs. Sparse Indexes

- Another advantage of a **clustered index** is the fact that it can be designed to be **space efficient**

Definition: Sparse Index

To keep the size of the index small, maintain **one index entry k^* per data file page** (not one index entry per data record). The key k is the **smallest key** on that page.

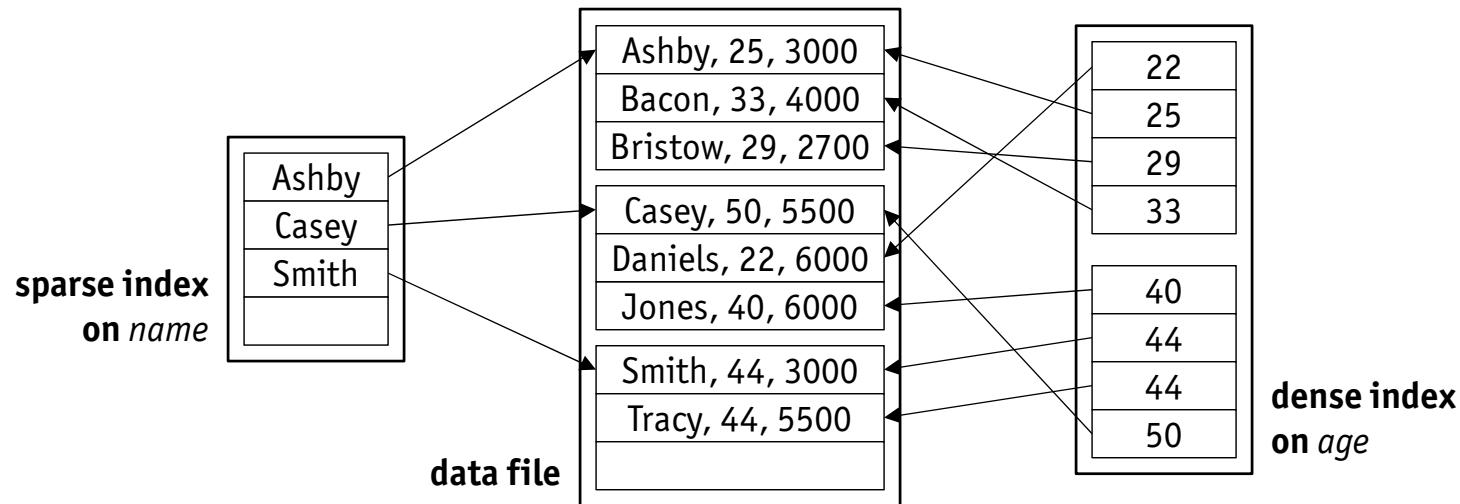
Indexes of this kind are called **sparse**. Otherwise indexes are referred to as **dense**.

Search a record with field **A = k** in a sparse A-index

- Locate the largest index entry k'^* such that $k'^* \leq k'$
- Then access the page pointed to by k'^*
- Scan this page (and the following pages, if needed) to find records with $\langle \dots, \mathbf{A} = k, \dots \rangle$.

Since the data file is clustered (i.e., sorted) on field **A**, matching records are guaranteed to be found in proximity

Dense vs. Sparse Index Example



- Again, the data file contains $\langle \text{name}, \text{age}, \text{sal} \rangle$ records
- Two indexes are maintained for the data file
 - **clustered sparse index** on field *name*
 - **unclustered dense index** on field *age*
- Both indexes use entry variant 2 to point into the data file

Dense vs. Sparse Indexes

- Final remarks
 - sparse indexes need 2-3 orders of magnitude **less space** than dense indexes
 - it is **not possible** to build a sparse index that is unclustered (i.e., there is at most one sparse index per file)

SQL queries and index exploitation

How do you propose to evaluate the following SQL queries?

- ```
SELECT MAX(age)
 FROM employees
```
- ```
SELECT MAX(name)
      FROM employees
```

Primary vs. Secondary Indexes

Terminology

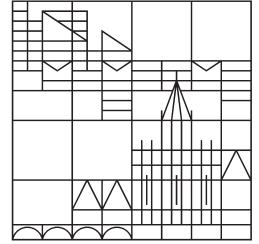
In the literature, there is often a distinction between **primary** (mostly used for indexes on the primary key) and **secondary** (mostly used for indexes on other attributes) indexes.

This terminology, however, is not very uniform and some text books may use those terms for different properties.

For example, some text books use **primary** to denote variant ① of indexes, whereas **secondary** is used to characterize the other two variants ② and ③.

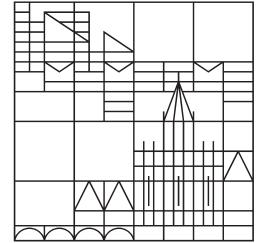
Multi-Attribute Indexes

- Each of the indexing techniques sketched so far can be applied to a **combination of attribute values** in a straightforward way
 - concatenate indexed attributes to form an index key,
e.g., `<lastname, firstname> → searchkey`
 - define index on **searchkey**
 - index will support lookup based on both attribute values,
e.g., ... `WHERE lastname='Doe' AND firstname='John'` ...
 - possibly, it will also support lookup based on a “prefix” of values,
e.g., ... `WHERE lastname='Doe'` ...
- So-called **multi-dimensional indexes** provide support for **symmetric** lookups for all subsets of the indexed attributes
- Numerous such indexes have been proposed, in particular for geographical and geometric applications



Database System Architecture and Implementation

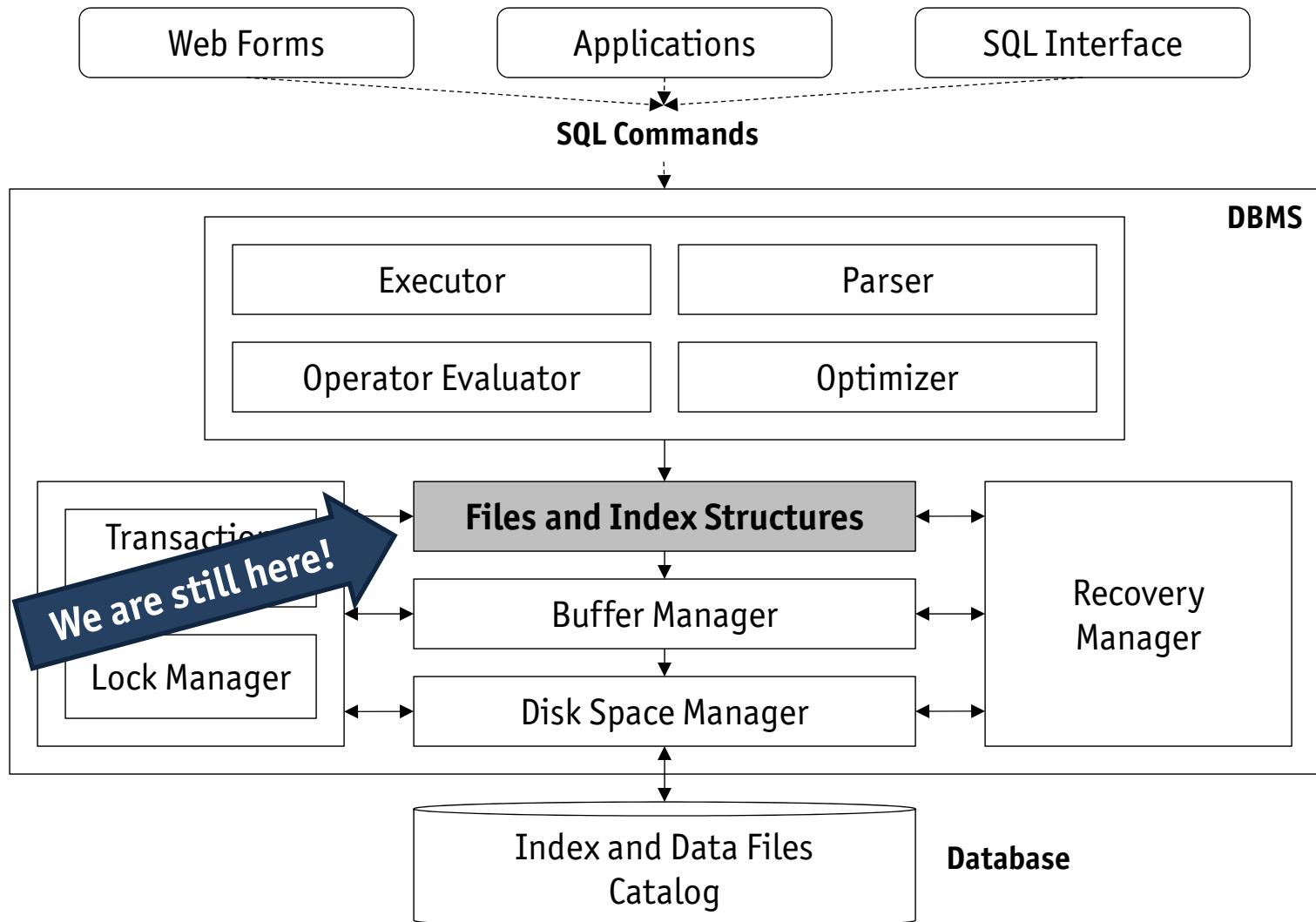
TO BE CONTINUED...



Database System Architecture and Implementation

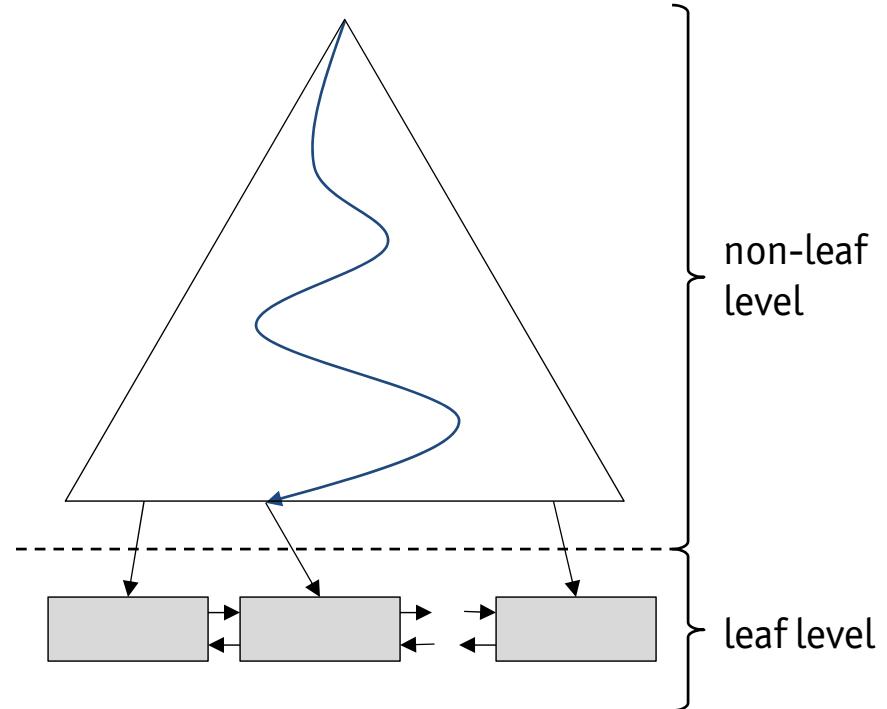
Module 3
Tree-Structured Indexes
November 13, 2013

Orientation



Module Overview

- Binary search
- ISAM
- B+ trees
 - search, insert, and delete
 - duplicates
 - key compression
 - bulk loading



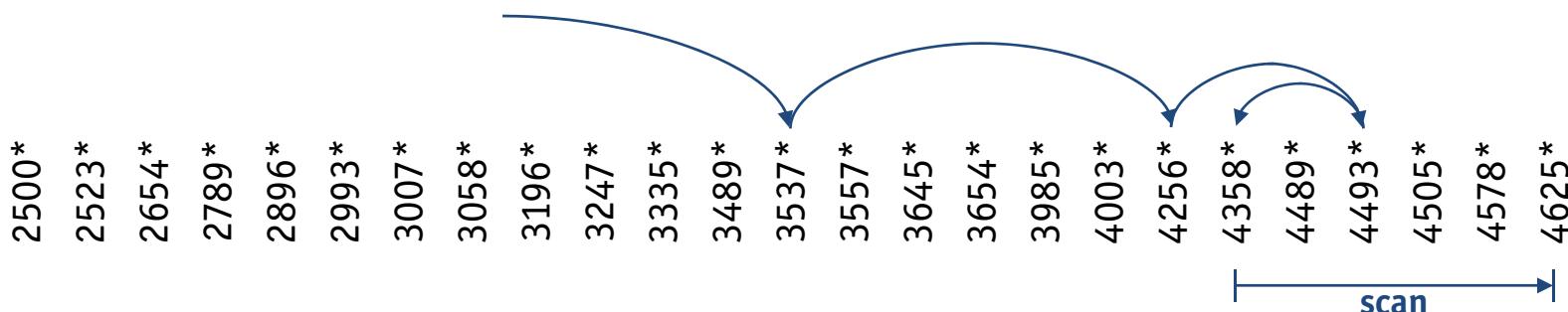
Binary Search

💻 How could we prepare for such queries and evaluate them efficiently

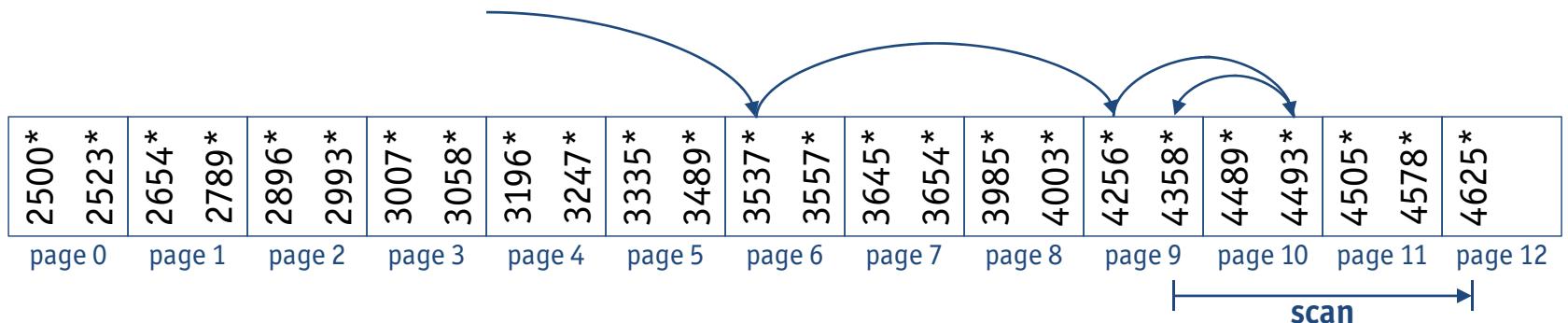
```
SELECT *
FROM   Employees
WHERE  Sal BETWEEN 4300 AND 4600
```

- We could
 1. sort the table on disk (in **Sal**-order)
 2. use **binary search** to find the first qualifying tuple, then scan as long as **Sal < 4600**

Again, let k^* denote the full record with key k



Binary Search



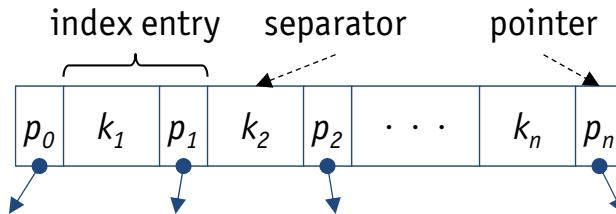
- Page I/O operations
 - ⊕ during the **scan phase**, pages are **accessed sequentially**
 - ⊖ during the **search phase**, $\log_2(\#tuples)$ need to be read
 - ⊖ about **the same number of pages** as tuples need to be read!
- Binary search is that it makes **far, unpredictable jumps**, which largely defeat page prefetching

Tree-Structured Indexing

- Intuition
 - improve binary search by introducing an **auxiliary structure** that only contains **one record per page** of the original (data) file
 - use this idea recursively until all records fit into **one single page**
- This simple idea naturally leads to a **tree-structured** organization of the indexes
 - ISAM
 - B+ trees
- Tree-structures indexes are particularly useful if **range selections** (and thus sorted file scans) need to be supported

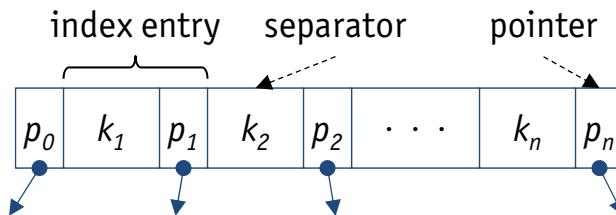
Indexed Sequential Access Method

- ISAM
 - acts as **static replacement** for the binary search phase
 - reads **considerable fewer pages** than binary search
- To support range selections on field **A**
 1. in addition to the **A**-sorted data file, maintain an **index file** with entries (records) of the following form



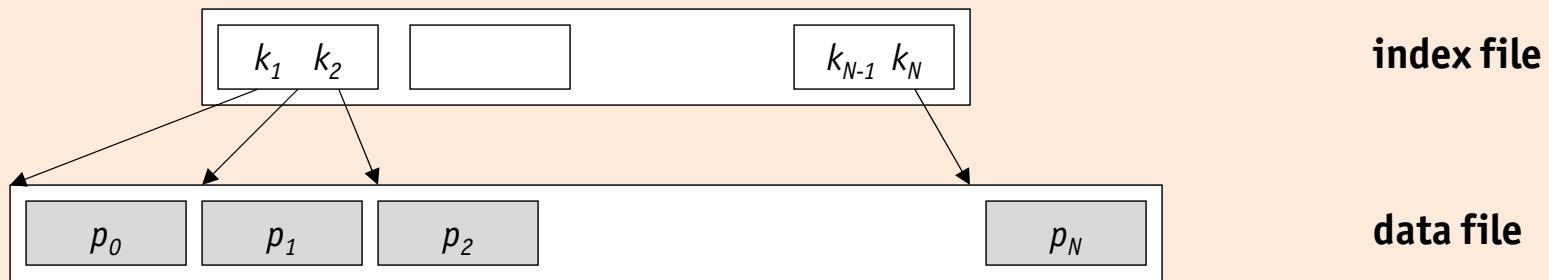
2. ISAM leads to **sparse** index structures, since in an index entry $\langle k_i, \uparrow p_i \rangle$ key k_i is the first (i.e., minimal) **A**-value on the data file page pointed to by p_i , where p_i is the page number

Indexed Sequential Access Method



- 3. in the index file, the k_i serve as separators between the contents of pages p_{i-1} and p_i
- 4. it is guaranteed that $k_{i-1} < k_i$ for $i = 2, \dots, n$
- We obtain a **one-level ISAM structure**

☞ One-level ISAM structure for $N + 1$ pages



Searching in ISAM

SQL query with range selection on field A

```
SELECT *
FROM   R
WHERE  A BETWEEN lower AND upper
```

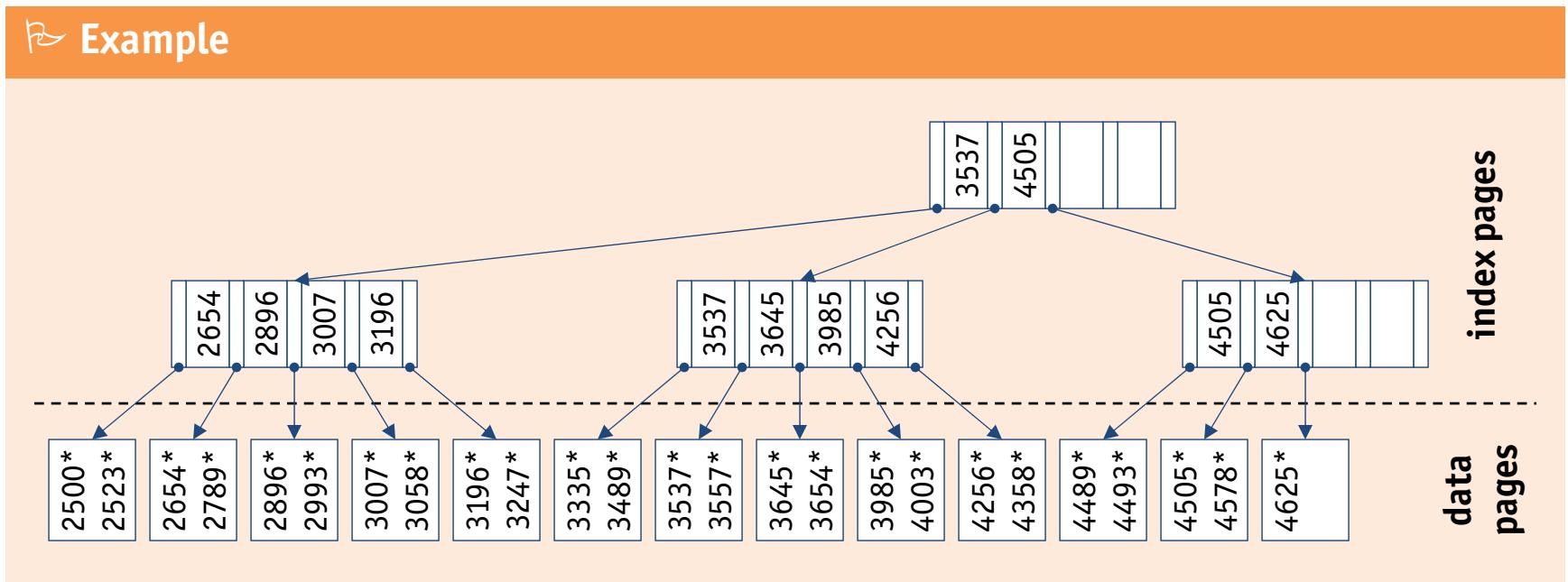
- To support range selection
 1. conduct a **binary search on the index file** for a key of value *lower*
 2. start a **sequential scan of the data file** from the page pointed to by the index entry and scan until field **A** exceeds *upper*
- Index file size is likely to be **much smaller** than data file size
 - searching the index is far more efficient than searching the data file
 - however, for large data files, even the index file might be too large to support fast searches

Multi-Level ISAM Structure

- **Recursively** apply the index creation step
 - treat the top-most index level like the data file and add an additional index layer on top
 - repeat until the top-most index layer fits into a single page (root page)
- This recursive index creation scheme leads to a **tree-structured** hierarchy of index levels

Multi-Level ISAM Structure

Example

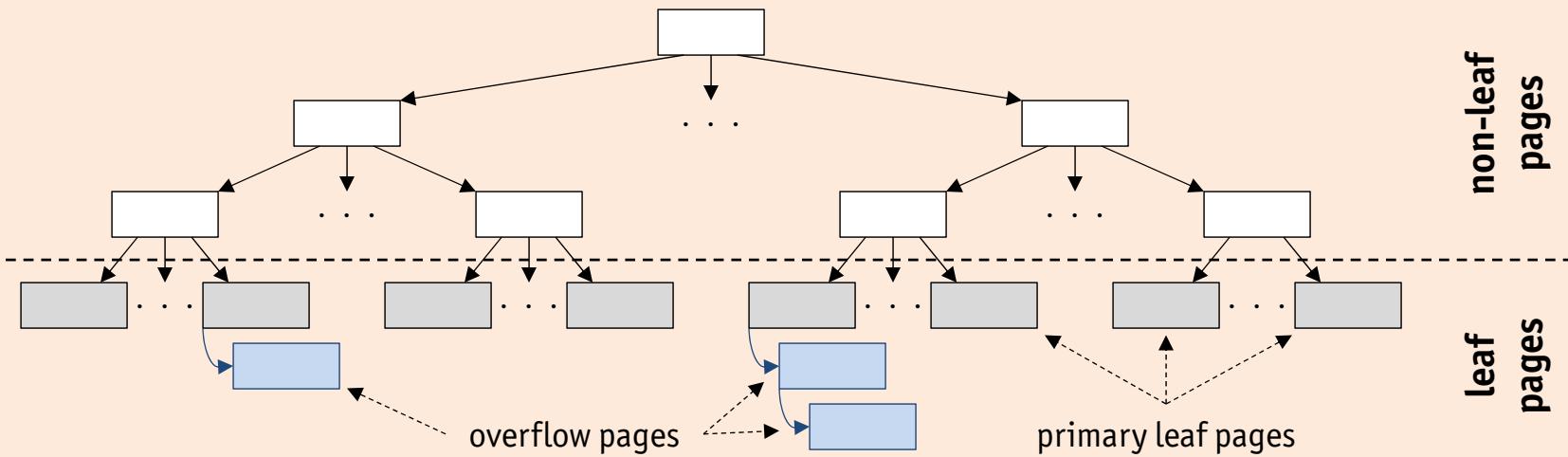


- Each ISAM tree node corresponds to **one page** (disk block)
- ISAM structure for a give data file is created **bottom up**
 1. sort the data file on the search key field
 2. create the index leaf level
 3. if top-most index level contains more than one page, repeat

ISAM Overflow Pages

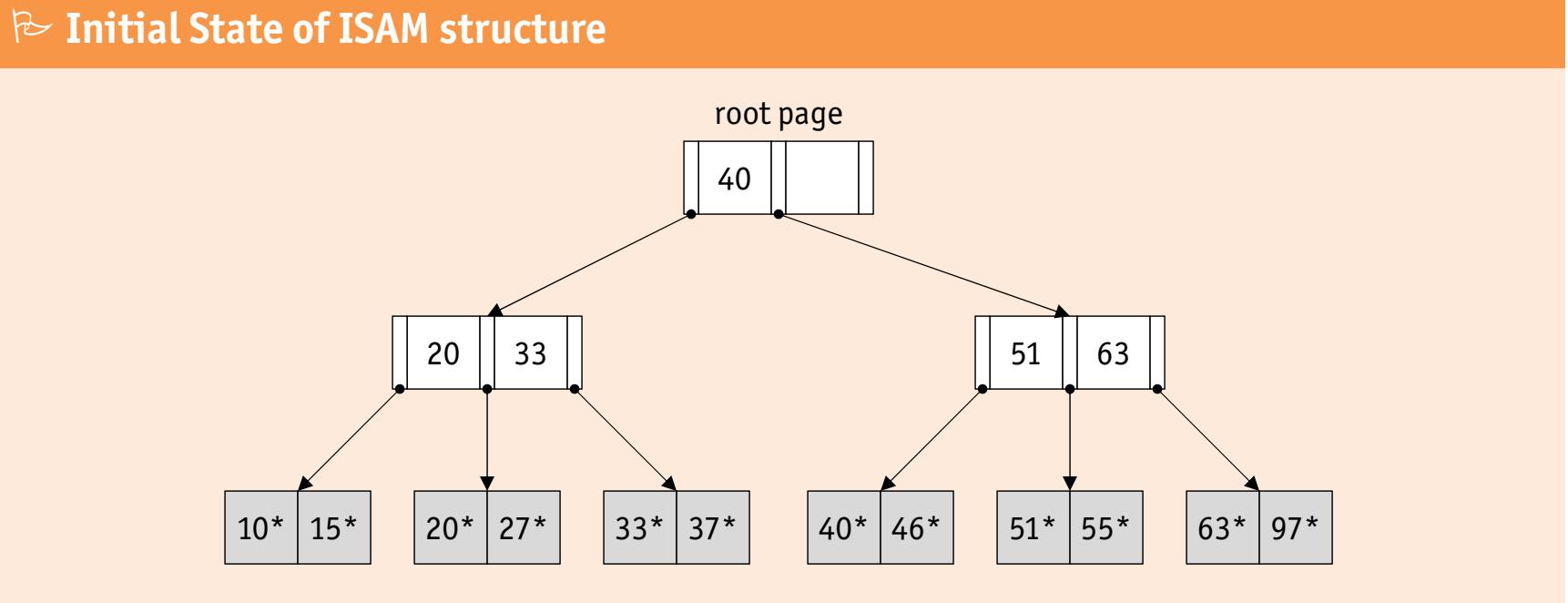
- The upper levels of the ISAM tree always remain **static**: updates in the data file **do not affect** the upper tree levels
 - if **space is available** on the corresponding leaf page, insert record there
 - otherwise, create and maintain a chain of **overflow pages** hanging off the full primary leaf page (overflow pages are **not ordered** in general)
- Over time, **search performance in ISAM can degrade**

Multi-level ISAM structure with overflow pages



ISAM Example: Initial State

- Each page can hold **two** index entries **plus one** (the left-most) page pointer



ISAM Example: Insertions

ISAM structure after insertion of data records with keys 23, 48, 41, and 42

non-leaf
pages

root page

40

20 33

51 63

primary
leaf pages

10* 15*

20* 27*

33* 37*

40* 46*

51* 55*

63* 97*

overflow
pages

23*

48* 41*

42*

ISAM Example: Deletions

ISAM structure after deletion of data records with keys 42, 51, and 97

non-leaf
pages

root page

40

20 33

51 63

primary
leaf pages

10* 15*

20* 27*

33* 37*

40* 46*

55*

63*

overflow
pages

23*

48* 41*

Is ISAM Too Static?

- Recall that ISAM structure is **static**
 - non-leaf levels are **not touched at all** by updates to the data file
 - may lead to **orphaned index key entries**, which do not appear in the index leaf level (e.g., key value **51** on the previous slide)

Orphaned index key entries

Does an index key entry like **51** (on the previous slide) cause problems during index key searches?

- To preserve the **separator property** of index key entries, it is necessary to maintain overflow chains
- ISAM may **lose balance** after heavy updating, which complicates the life for the query optimizer

Is ISAM Too Static?

- Recall that ISAM structure is **static**
 - non-leaf levels are **not touched at all** by updates to the data file
 - may lead to **orphaned index key entries**, which do not appear in the index leaf level (e.g., key value **51** on the previous slide)

Orphaned index key entries

Does an index key entry like **51** (on the previous slide) cause problems during index key searches?

☞ No, since the index keys maintain their separator property.

- To preserve the **separator property** of index key entries, it is necessary to maintain overflow chains
- ISAM may **lose balance** after heavy updating, which complicates the life for the query optimizer

Static Is Not All Bad

- Leaving **free space** during index creation reduces the insertion/overflow problem (typically $\approx 20\%$ free space)
- Since ISAM indexes are static, **pages do not need to be locked** during concurrent index access
 - locking can be a serious **bottleneck** in dynamic tree indexes (particularly near the root node)
- ISAM may be the index of choice for **relatively static** data

ISAM-style implementations

- ↳ **MySQL**
 - implements and extends ISAM as MyISAM, which is the default storage engine
- ↳ **Berkeley DB**
- ↳ **Microsoft Access**

Fan-Out

Definition

The average number of children for a non-leaf node is called the **fan-out** of the tree. If every non-leaf node has n children, a tree of height h has n^h leaf pages.

⇒ In practice, nodes do not have the same number of children, but using the **average value** F for n is a good **approximation** to the number of leaf pages F^h .

Exercise: Number of children

Why can non-leaf nodes have **different** numbers of children?

Fan-Out

Definition

The average number of children for a non-leaf node is called the **fan-out** of the tree. If every non-leaf node has n children, a tree of height h has n^h leaf pages.

⇒ In practice, nodes do not have the same number of children, but using the **average value** F for n is a good **approximation** to the number of leaf pages F^h .

Exercise: Number of children

Why can non-leaf nodes have **different** numbers of children?

⇒ Index entries k^* can be of variable length if the index is built on a variable-length key. Additionally, index entries k^* of variant ① and ③ can be of variable length because variable-length records or lists of *rids* are stored in the index entries.

Cost of Searches in ISAM

- Let N be the number of pages in the data file and let F denote the fan-out of the ISAM tree
 - when the index search begins, the search space is of size N
 - with the help of the root page, the index search is guided to a sub-tree of size

$$N \cdot 1/F$$

- as the index search continues down the tree, the search space is repeatedly reduced by a factor of F

$$N \cdot 1/F \cdot 1/F \cdots$$

- the index search ends after s steps, when the search space has been reduced to size 1 (i.e., when it reaches the index leaf level and hits the data page that contains the desired record)

$$N \cdot (1/F)^s \stackrel{\text{def}}{=} 1 \Leftrightarrow s = \log_F N$$

Cost of Searches in ISAM

Exercise: Binary search vs. tree

Assume a data file consists of 100 million leaf pages. How many page I/O operations will it take to find a value using ① **binary search** and ② an **ISAM tree with fan-out 100**?

Cost of Searches in ISAM

Exercise: Binary search vs. tree

Assume a data file consists of 100 million leaf pages. How many page I/O operations will it take to find a value using ① **binary search** and ② an **ISAM tree with fan-out 100**?

- ① **binary search**

$$\log_2(100,000,000) \approx 25 \text{ page I/O operations}$$

- ② **ISAM tree with fan-out 100**

$$\log_{100}(100,000,000) = 4 \text{ page I/O operations}$$

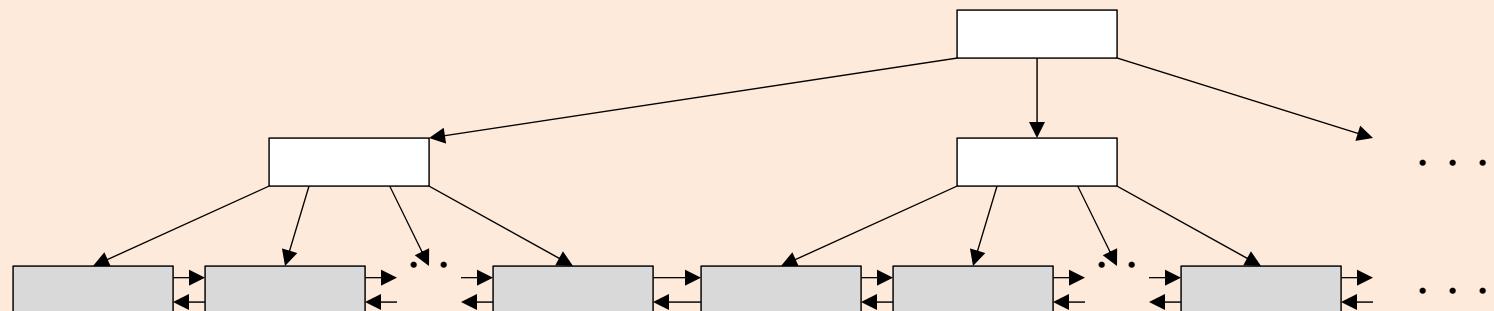
B+ Tree Properties

- The **B+ tree index structure** is derived from the ISAM index structure, but is fully dynamic w.r.t. updates
 - search performance is only dependent on the **height** of the B+ tree (because of a high fan-out, the height rarely exceeds 3)
 - B+ trees **remains balanced, no overflow chains** develop
 - B+ trees support efficient **insert/delete operations**, where the underlying data file can grow/shrink dynamically
 - B+ tree nodes (with the exception of the root node) are **guaranteed to have a minimum occupancy of 50%** (typically 66%)

B+ Trees Structure

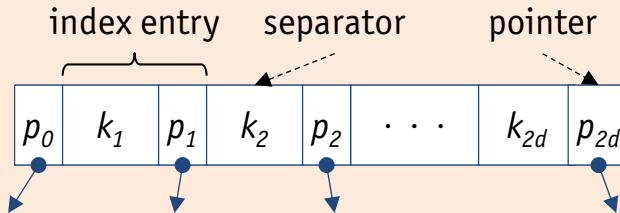
- Differences between B+ tree structure and ISAM structure
 - leaf nodes are connected to form a **doubly-linked list**, the so-called **sequence set**
 - ↳ *not a strict requirement, but implemented in most systems*
 - leaves may contain **actual data records** (variant ①) or just **references to records** on data pages (variants ② and ③)
 - ↳ *instead, ISAM leaves are the data pages themselves*

Sketch of B+ tree structure (data pages not shown)



B+ Tree Non-Leaf Nodes

↷ B+ inner (non-leaf) node



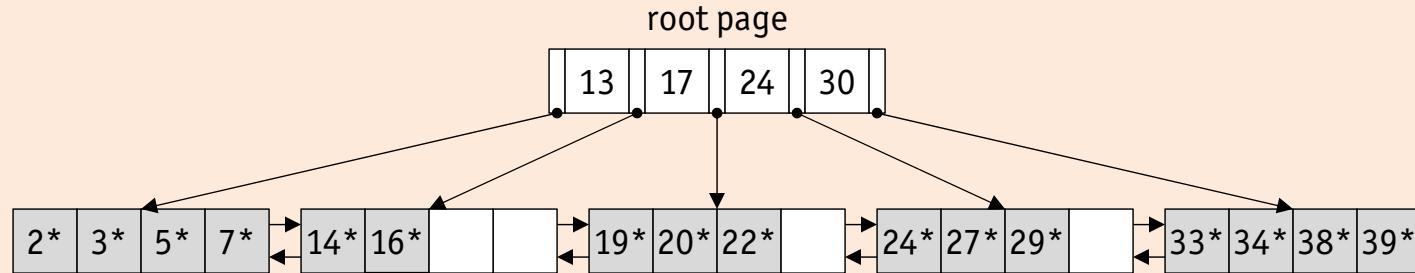
- B+ tree non-leaf nodes use the same internal layout as inner ISAM nodes
 - the **minimum** and **maximum number of entries** n is bounded by the **order d** of the B+ tree
$$d \leq n \leq 2 \cdot d \text{ (root node: } 1 \leq n \leq 2 \cdot d\text{)}$$
 - a node contains $n + 1$ pointers, where pointer p_i ($1 \leq i \leq n - 1$) points to a sub-tree in which all key values k are such that
$$k_i \leq k < k_{i+1}$$
 $(p_0 \text{ points to a sub-tree with key values } < k_1, p_{2d} \text{ points to a sub-tree with key values } \geq k_{2d})$

B+ Tree Leaf Nodes

- B+ tree leaf nodes contain pointers to data **records** (not **pages**)
- A **leaf node entry** with key value k is denoted as k^* as before
- All index entry variants ①, ②, and ③ can be used to implement the leaf entries
 - for variant ①, the B+ tree represents the index as well as the data file itself and leaf node entries therefore look like
$$k_i^* = \langle k_i, \langle \dots \rangle \rangle$$
 - for variants ② and ③, the B+ tree is managed in a file separate from the actual data file and leaf node entries look like
$$k_i^* = \langle k_i, rid \rangle$$
$$k_i^* = \langle k_i, [rid_1, rid_2, \dots] \rangle$$

B+ Tree Search

Example of a B+ tree with order $d = 2$



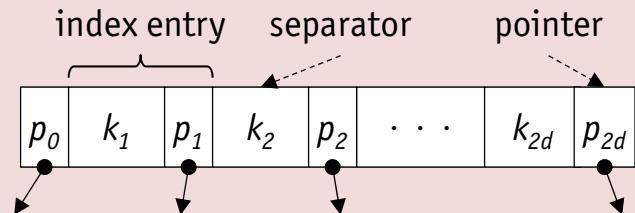
- Each node contains between 2 and 4 entries (order $d = 2$)
- Example of B+ tree searches
 - for entry 5*, follow the left-most child pointer, since $5 < 13$
 - for entries 14* or 15*, follow the second pointer, since $13 \leq 14 < 17$ and $13 \leq 15 < 17$ (because 15* cannot be found on the appropriate leaf, it can be concluded that it is not present in the tree)
 - for entry 24*, follow the fourth child pointer, since $24 \leq 24 < 30$

B+ Tree Search

Searching in a B+ tree

```
function search (k) : ↑node
    return treeSearch (root, k)
end

function treeSearch (↑node, k) : ↑node
    if node is a leaf node then return ↑node
    else
        if  $k < k_1$  then return treeSearch ( $p_0$ , k) ;
        else
            if  $k \geq k_{2d}$  then return treeSearch ( $p_{2d}$ , k) ;
            else
                find  $i$  such that  $k_i \leq k < k_{i+1}$  ;
                return treeSearch ( $p_i$ , k)
        end
    end
```



B+ Tree Insert

- B+ trees remain **balanced** regardless of the updates performed
 - **invariant:** all paths from the root to any leaf must be of **equal length**
 - insertions and deletions have to **preserve** this invariant

Basic principle of insertion into a B+ tree with order d

To insert a record with key k

1. start with root node and **recursively** insert entry into appropriate child node
2. descend down tree until **leaf node is found**, where entry belongs
(let n denote the leaf node to hold the record and m the number of entries in n)
3. if $m < 2 \cdot d$, there is capacity left in n and k^* **can be stored in leaf node n**
 **Otherwise...?**

- We *cannot* start an overflow chain hanging off p as this solution would violate the balancing invariant
- We *cannot* place k^* elsewhere (even close to n) as the cost of **search (k)** should only be dependent on the tree's height

B+ Tree Insert

☛ Splitting nodes

If a node n is full, it must be **split**

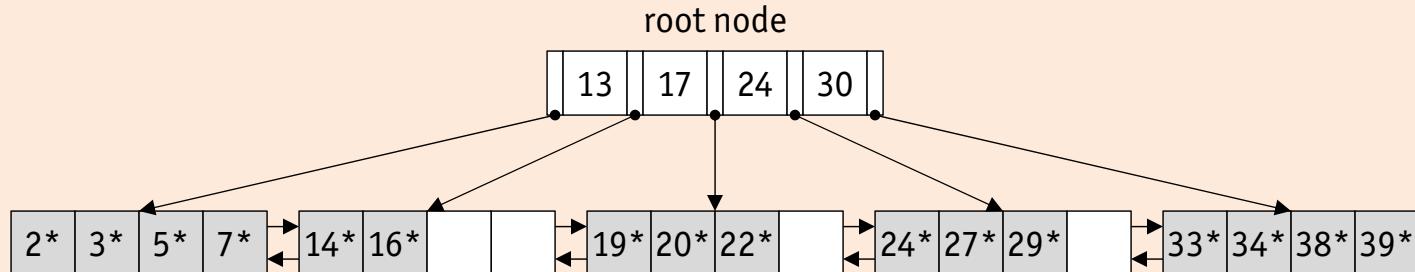
1. create a **new node n'**
2. distribute the entries of n and the new entry k **over n and n'**
3. insert an entry $\uparrow n'$ pointing to the new node n' **into its parent**

Splitting can therefore **propagate up the tree**. If the root has to be split, a new root is created and the **height of the tree increases** by 1.

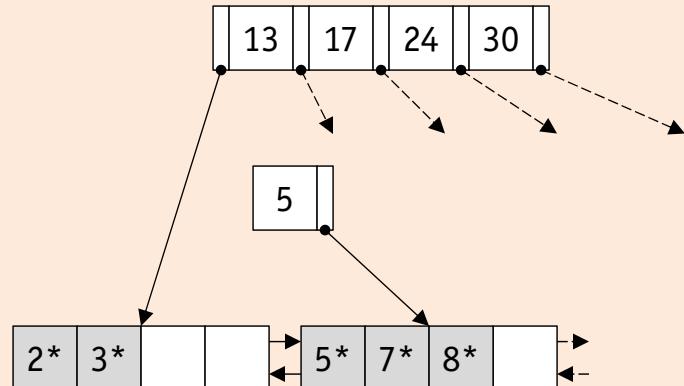
B+ Tree Insert

Example: Insertion into a B+ tree with order $d = 2$

1. insert record with key $k = 8$ into the following B+ tree



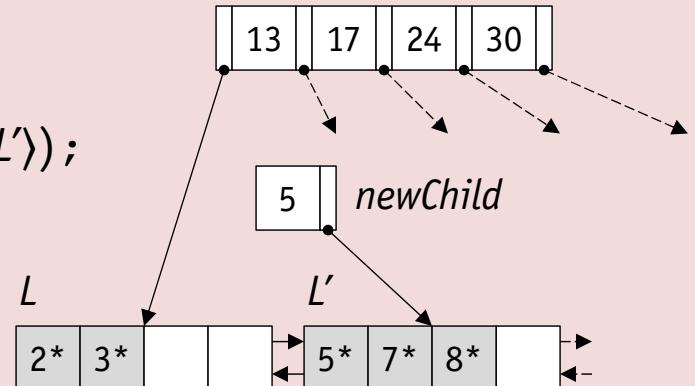
2. the new record has to be inserted into the left-most leaf node n
3. since n is **already full**, it has to be split
4. create a **new leaf node n'**
5. entries 2^* and 3^* remain on n , whereas entries 5^* , 7^* and 8^* (new) go into n'
6. key $k' = 5$ is the **new separator** between nodes n and n' and has to be **inserted into their parent** (copy up)



B+ Tree Insert

💻 Insert into B+ tree of degree d (leaf nodes)

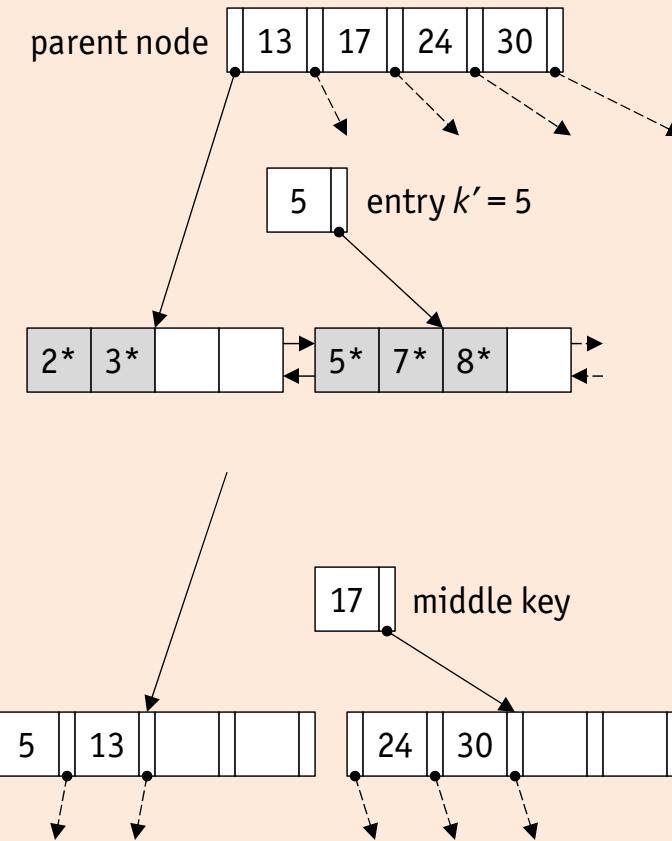
```
function insert ( $\uparrow$ node,  $k^*$ ) :  $\uparrow$ newChild
    if node is a non-leaf node, say  $N$  then ...
    if node is a leaf node, say  $L$  then
        if  $L$  has space then
            put  $k^*$  on  $L$ ;  $\uparrow$ newChild  $\leftarrow$  null; return;
        else
            split  $L$ : first  $d$  entries stay, rest move to new node  $L'$ ;
            put  $k^*$  on  $L$  or  $L'$ ;
            set sibling pointers in  $L$  and  $L'$ ;
             $\uparrow$ newChild  $\leftarrow$  @(<smallest key value on  $L'$ ,  $\uparrow L'$ >);
        return;
    endproc;
```



B+ Tree Insert

Example: Insertion into a B+ tree with order $d = 2$ (cont'd)

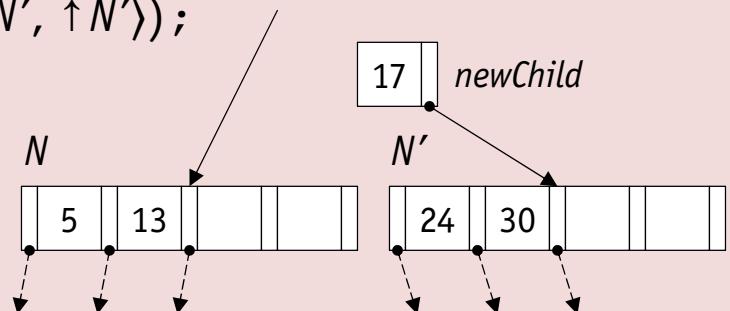
7. to insert entry $k' = 5$ into parent node, **another split** has to occur
8. parent node is **also full** since it already has $2d$ keys and $2d + 1$ pointers
9. with the new entry, there is a **total** of $2d + 1$ keys and $2d + 2$ pointers
10. form **two minimally full non-leaf nodes**, each containing d keys and $d + 1$ pointers, **plus an extra key**, the **middle key**
11. middle key plus pointer to second non-leaf node constitute a **new index entry**
12. new index entry has to be **inserted into parent** of split non-leaf node (push up)



B+ Tree Insert

💻 Insert into B+ tree of degree d (non-leaf nodes)

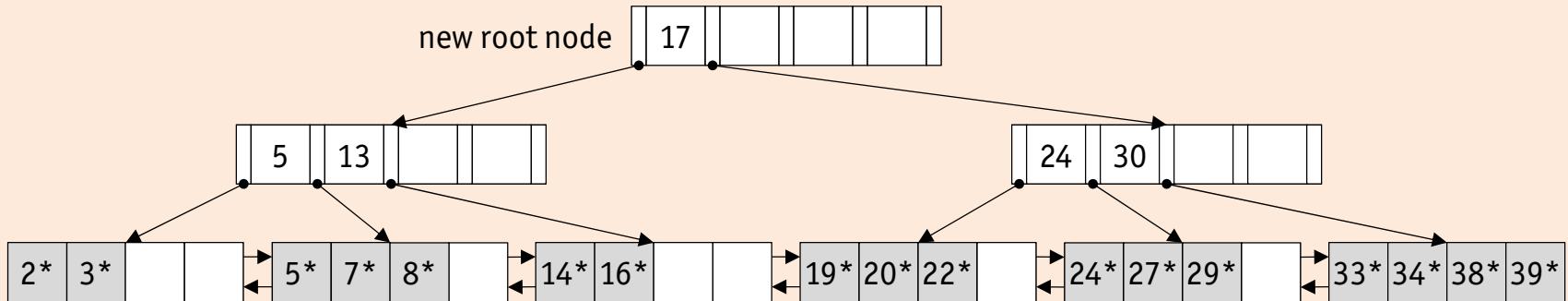
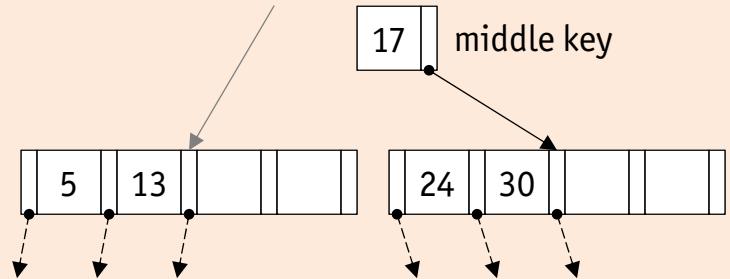
```
function insert ( $\uparrow$ node,  $k^*$ ) :  $\uparrow$ newChild
    if node is a non-leaf node, say  $N$  then
        find  $i$  such that  $k_i \leq k < k_{i+1}$ ;
         $\uparrow$ newChild = insert ( $p_i$ ,  $k$ );
        if  $\uparrow$ newChild is null then return;
        else
            if  $N$  has space then put newChild on it;  $\uparrow$ newChild  $\leftarrow$  null; return;
            else
                split  $N$ : first  $d$  key values and  $d + 1$  pointers stay,
                    last  $d$  key values and  $d + 1$  pointers move to new node  $N'$ ;
                 $\uparrow$ newChild  $\leftarrow$  @(<smallest key value on  $N'$ ,  $\uparrow N'$ >);
                if  $N$  is root then ...
            return;
    if node is a leaf node, say  $L$  then ...
endproc;
```



B+ Tree Insert

Example: Insertion into a B+ tree with order $d = 2$ (cont'd)

13. parent node that was split, was the (old) **root node** of the B+ tree
14. create a **new root node** to hold the entry that distinguishes the two split index pages



B+ Tree Insert

💻 Insert into B+ tree of degree d (root node)

```
function insert ( $\uparrow$ node,  $k^*$ ) :  $\uparrow$ newChild
    if node is a non-leaf node, say  $N$  then
        ...
        split  $N$ : first  $d$  key values and  $d + 1$  pointers stay,
                  last  $d$  key values and  $d + 1$  pointers move to new node  $N'$ ;
         $\uparrow$ newChild  $\leftarrow$  @( $\langle$ smallest key value on  $N'$ ,  $\uparrow N'\rangle$ );
        if  $N$  is root then
            create new node with  $\langle \uparrow N$ ,  $newChild\rangle$ ;
            make the tree's root node pointer point to the new node
        return;
    if node is a leaf node, say  $L$  then ...
endproc;
```

B+ Tree Root Node Split

- Splitting starts at the leaf level and continues upward as long as index nodes are fully occupied
- Eventually, the root node might be split
 - root node is the only node that may have an occupancy of < 50%
 - tree height only increases if the root is split



How often do you expect a root split to happen?

B+ Tree Root Node Split

- Splitting starts at the leaf level and continues upward as long as index nodes are fully occupied
- Eventually, the root node might be split
 - root node is the only node that may have an occupancy of < 50%
 - tree height only increases if the root is split

How often do you expect a root split to happen?

Assume a B+ tree over 8 byte integers with 4 kB pages and with pointers encoded as 8 byte integers.

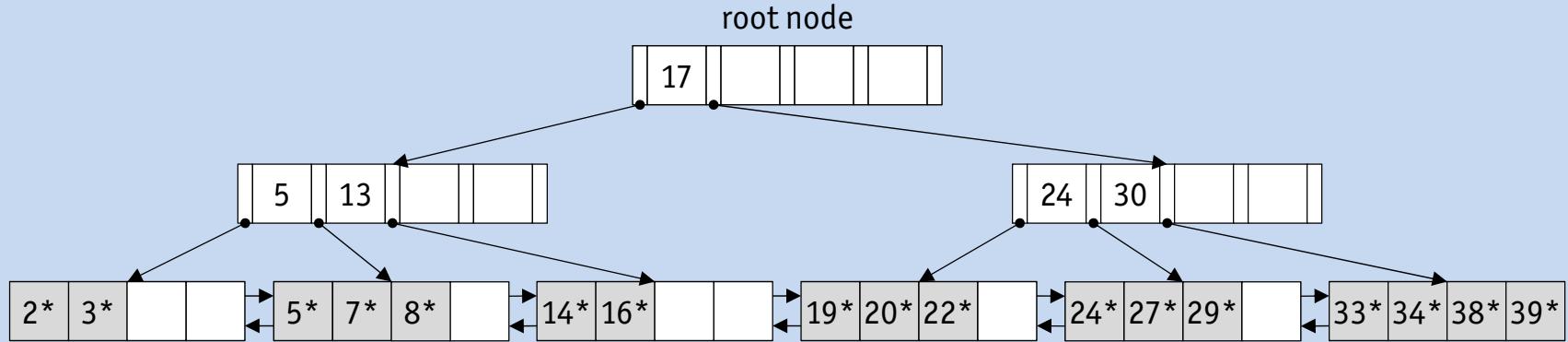
- 128-256 index entries/page (fan-out F)
- an index of height h indexes at least 128^h records, typically more

h	#records
2	16,000
3	2,000,000
4	250,000,000

B+ Tree Insert

✍ Further key insertions

How does the insertion of records with keys $k = 23$ and $k = 40$ alter the B+ tree?

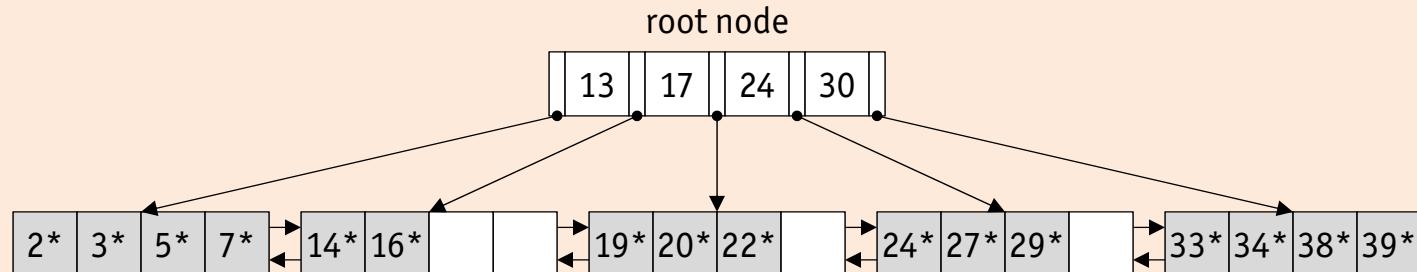


B+ Tree Insert with Redistribution

- Redistribution further improves average occupancy in a B+ tree
 - before a node n is split, its entries are **redistributed** with a sibling
 - a **sibling** of a node n is a node that is immediately to the left or right of N and has the same parent as n

Example: Insertion with redistribution into a B+ tree with order $d = 2$

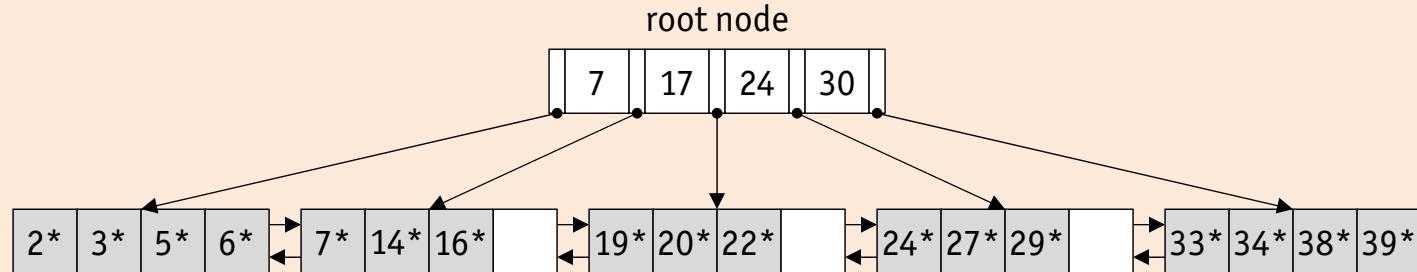
- insert record with key $k = 6$ into the following B+ tree



- the new record has to be inserted into the left-most leaf node, say n , which is **full**
- however, the (only) sibling of n **only has two entries and can accommodate more**
- therefore, insert of $k = 6$ can be handled with a **redistribution**

B+ Tree Insert with Redistribution

Example: Insertion with redistribution into a B+ tree with order $d = 2$



5. redistribution “**rotates**” values through the parent node from node n to its sibling

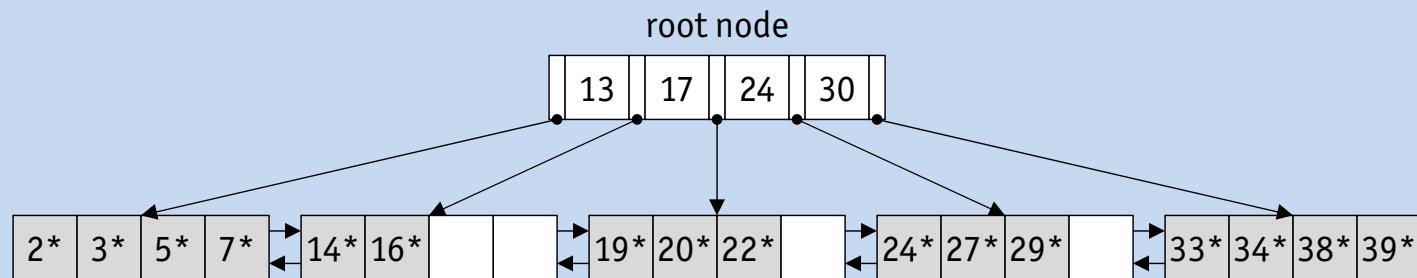
B+ Tree Insert with Redistribution

Redistribution makes a difference

Insert a record with key $k = 30$

- Ⓐ without redistribution
- Ⓑ using leaf-level redistribution

into the B+ tree shown below. How does the tree change?



B+ Tree Delete

- B+ tree deletion algorithm follows the same basic principle as the insertion algorithm

Basic principle of deletion from a B+ tree with order d

To delete a record with key k

1. start with root node and **recursively** delete entry from appropriate child node
2. descend down tree until **leaf node is found**, where entry is stored (let n denote the leaf node that holds the record and m the number of entries in n)
3. if $m > d$, n does not have minimum occupancy and k^* **can simply be deleted from leaf node n**
 **Otherwise...?**

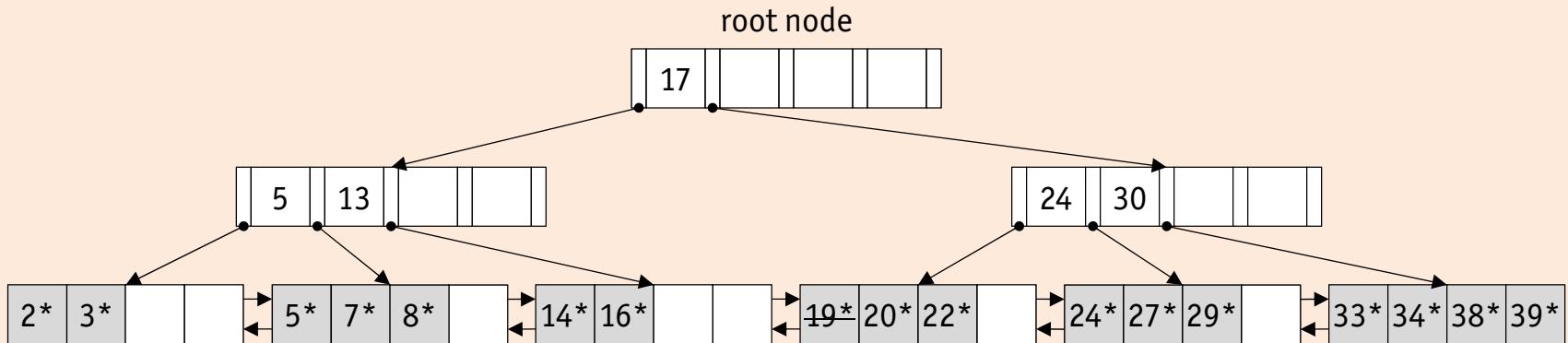
B+ Tree Delete

- Two techniques to handle the case that number of entries m of a node n falls under the **minimum occupancy threshold d**
- Redistribution
 - redistribute entries between n and an adjacent sibling
 - update parent to reflect redistribution: **change entry** pointing to second node to lowest search key in second node
- Merge
 - merge node n with an adjacent sibling
 - update parent to reflect merge: **delete entry** pointing to second node
 - if last entry in root is deleted, the height of the tree decreases by 1

B+ Tree Delete

Example: Deletion from a B+ tree with order $d = 2$

1. delete record with key $k = 19$ (i.e., entry 19^*) from the following B+ tree

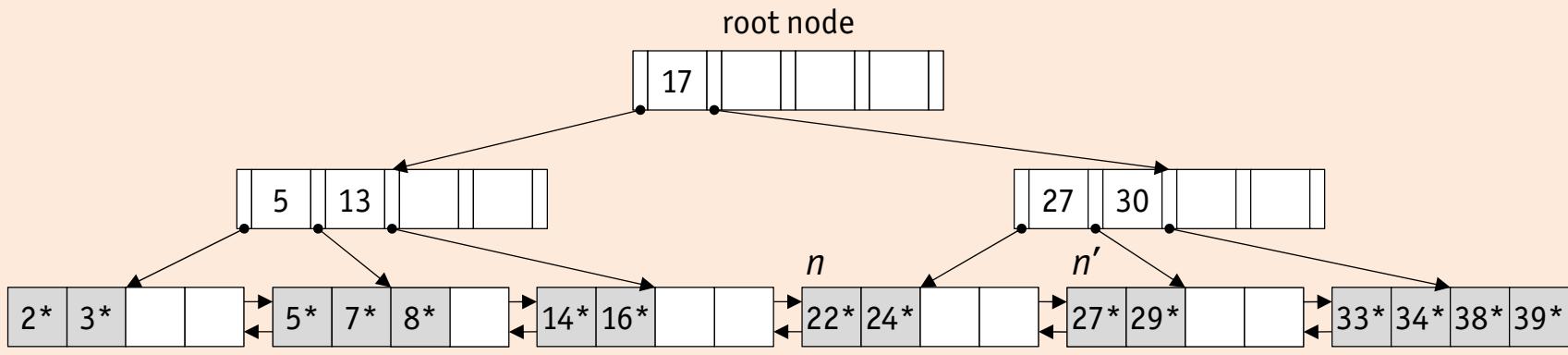
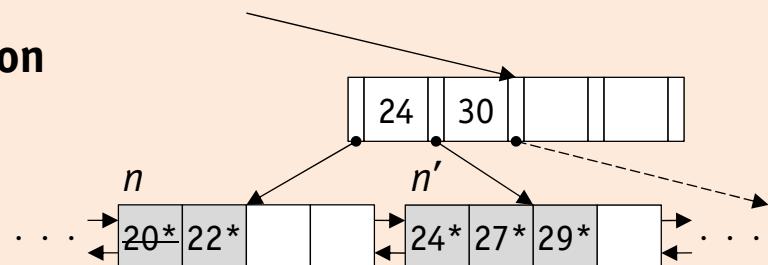


2. recursive tree traversal ends at leaf node n containing entries 19^* , 20^* , and 22^*
3. since $m = 3 > 2$, there is **no node underflow** in n after removal and entry 19^* can safely be deleted

B+ Tree Delete

Example: Deletion from a B+ tree with order $d = 2$ (cont'd)

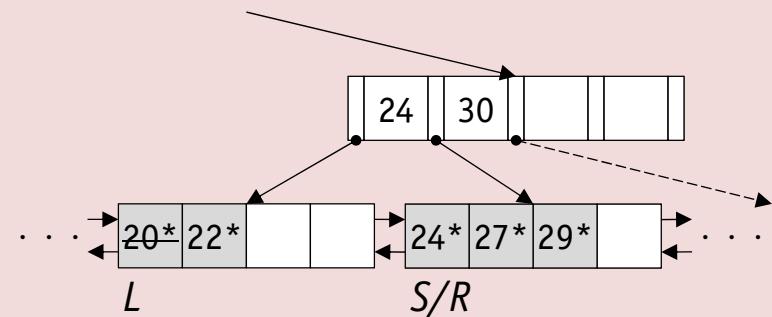
4. subsequent deletion of record with key $k = 20$ (i.e., entry 20^*) results in **underflow** of node n as it already has minimal occupancy $d = 2$
5. since the (only) sibling n' of n has $3 > 2$ entries (24^* , 27^* , and 29^*), **redistribution** can be used to deal with the underflow of n
6. move entry 24^* to n and copy up the **new splitting key 27**, which is the new smallest key value on n'



B+ Tree Delete

>Delete from B+ tree of degree d (leaf nodes: redistribution)

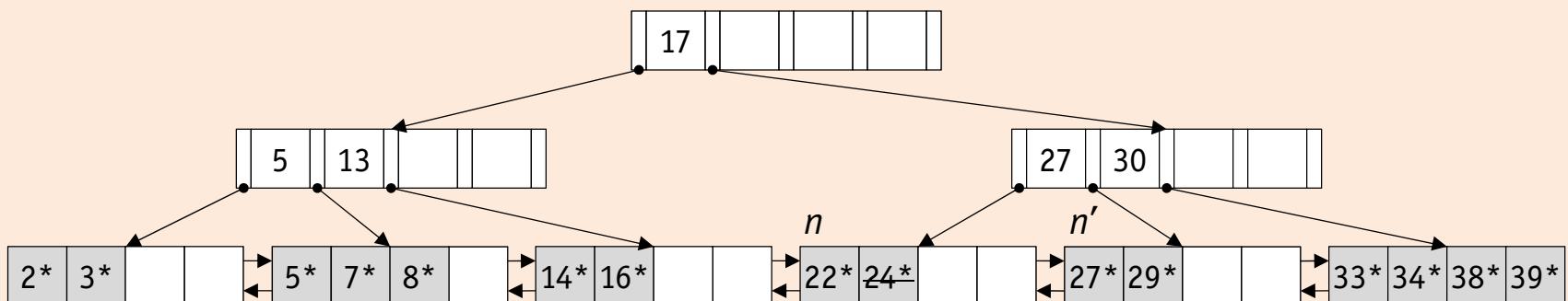
```
function delete ( $\uparrow parent$ ,  $\uparrow node$ ,  $k^*$ ) :  $\uparrow oldChild$ 
    if  $node$  is a non-leaf node, say  $N$  then ...
    if  $node$  is a leaf node, say  $L$  then
        if  $L$  has entries to spare then remove  $k^*$ ;  $\uparrow oldChild \leftarrow null$ ; return;
        else get a sibling  $S$  of  $L$ ;
            if  $S$  has extra entries then
                redistribute entries evenly between  $L$  and  $S$ ;
                find entry for right node, say  $R$ , in parent;
                replace key value in parent by new low-key value in  $R$ ;
                 $\uparrow oldChild \leftarrow null$ ; return;
            else ...
    endproc;
```



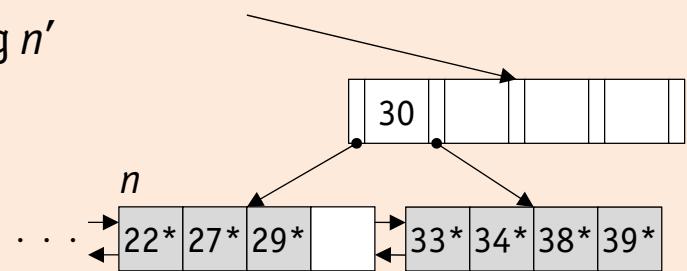
B+ Tree Delete

Example: Deletion from a B+ tree with order $d = 2$ (cont'd)

7. suppose record with key $k = 24$ (i.e., entry 24^*) is deleted next
root node



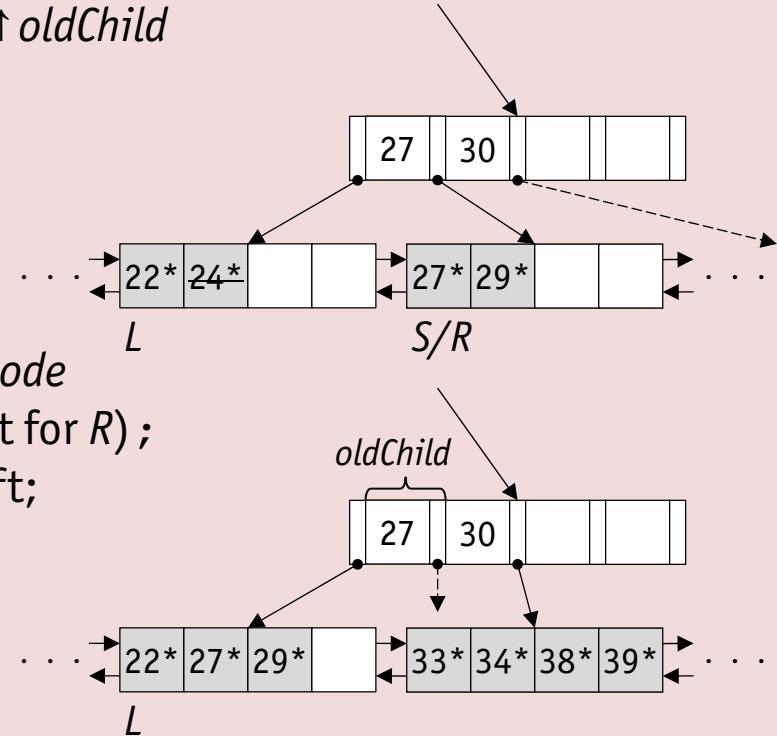
8. again, leaf-node n **underflows** as it only contains $1 < 2$ entries after deletion
9. redistribution is **not an option** as (only) sibling n' of n just contains two entries (27^* and 29^*)
10. together n and n' contain $3 > 2$ entries and can therefore be **merged**: move entries 27^* and 29^* from n' to n , then delete node n'
11. note that separator 27 between n and n' is no longer needed and therefore **discarded (recursively deleted)** from parent



B+ Tree Delete

>Delete from B+ tree of degree d (leaf nodes: merge)

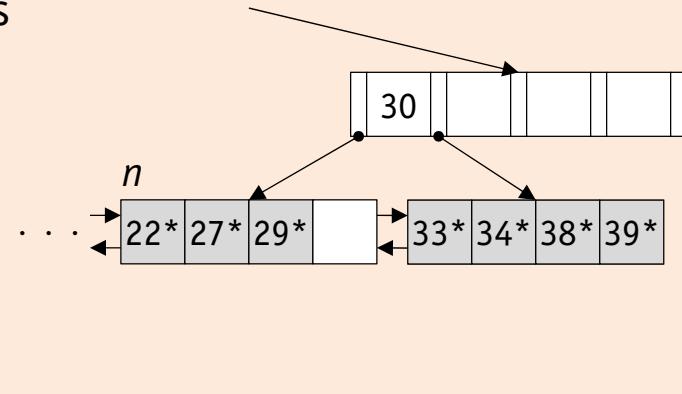
```
function delete (↑parent, ↑node, k*) : ↑oldChild
    if node is a non-leaf node, say  $N$  then ...
    if node is a leaf node, say  $L$  then
        if  $L$  has entries to spare then ...
        else ...
            if  $S$  has extra entries then ...
            else merge  $L$  and  $S$ , let  $R$  be the right node
                ↑oldChild ← @ (current entry in parent for  $R$ );
                move all entries from  $R$  to node on left;
                discard empty node  $R$ ;
                adjust sibling pointers;
    return;
endproc;
```



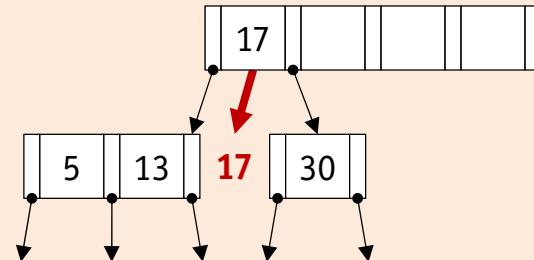
B+ Tree Delete

Example: Deletion from a B+ tree with order $d = 2$ (cont'd)

12. now, parent of n **underflows** as it only contains $1 < 2$ entries after deletion of entry $\langle 27, \uparrow n' \rangle$
 13. redistribution is **not an option** as its sibling just contains two entries (**5** and **13**)
 14. therefore, **merge the nodes** into a new node with $d + (d - 1)$ keys and $d + 1 + d$ pointers
- $\left[\begin{matrix} 22^* \\ 27^* \\ 29^* \end{matrix} \right] \quad \left[\begin{matrix} 33^* \\ 34^* \\ 38^* \\ 39^* \end{matrix} \right]$
- $\underbrace{\hspace{1cm}}_{\text{left}} \quad \underbrace{\hspace{1cm}}_{\text{right}} \quad \underbrace{\hspace{1cm}}_{\text{left}} \quad \underbrace{\hspace{1cm}}_{\text{right}}$



15. since a complete node needs to contain $2d$ keys and $2d + 1$ pointers, a **key value is missing**
16. missing key value is **pulled down** (i.e., deleted) from the parent to complete the merged node



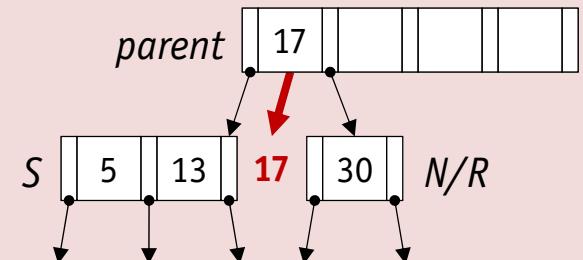
B+ Tree Delete

>Delete from B+ tree of degree d (non-leaf nodes: merge)

```

function delete ( $\uparrow parent$ ,  $\uparrow node$ ,  $k^*$ ) :  $\uparrow oldChild$ 
  if  $node$  is a non-leaf node, say  $N$  then
    find  $i$  such that  $k_i \leq k < k_{i+1}$ ;
     $\uparrow oldChild = \text{delete}(p_i, k)$  ;
    if  $\uparrow oldChild$  is null then return ;
    else
      remove  $oldChild$  from  $N$  ;
      if  $N$  has entries to spare then  $\uparrow oldChild \leftarrow \text{null}$  ; return ;
      else get a sibling  $S$  of  $N$ , using  $\uparrow parent$ 
        if  $S$  has extra entries then ...
        else merge  $L$  and  $S$ , let  $R$  be the right node
           $\uparrow oldChild \leftarrow @(\text{current entry in parent for } R)$  ;
          pull splitting key from parent down into left node ;
          move all entries from  $R$  to left node; discard empty node  $R$  ; return ;
      if  $node$  is a leaf node, say  $L$  then ...
endproc ;

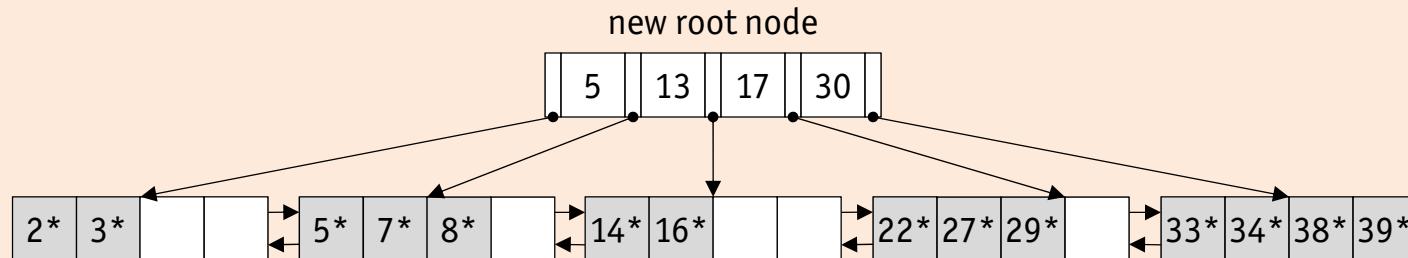
```



B+ Tree Delete

Example: Deletion from a B+ tree with order $d = 2$ (cont'd)

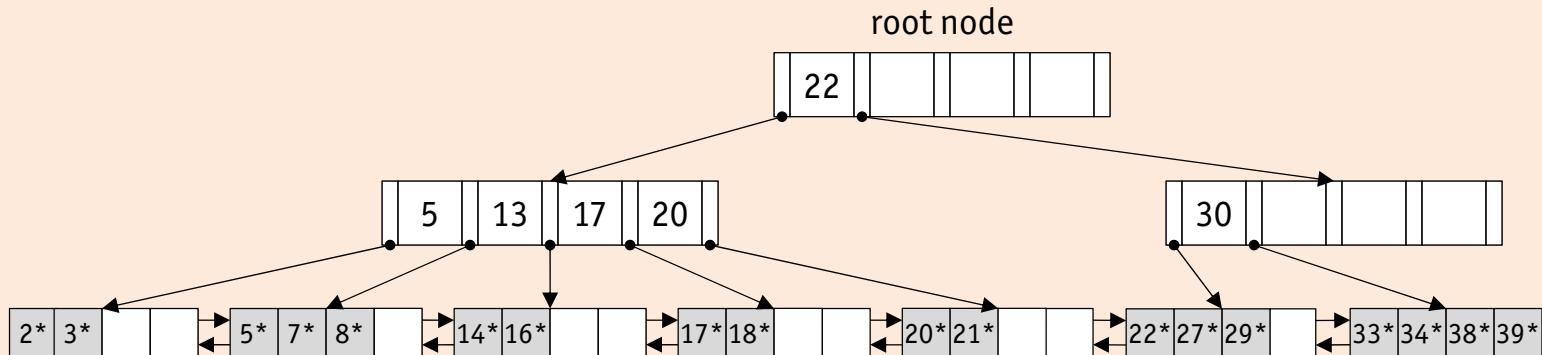
17. since the last remaining entry in the root was discarded, the merged node becomes the **new root**



B+ Tree Delete

Example: Deletion from a B+ tree with order $d = 2$ (cont'd)

18. suppose the following B+ tree is encountered **during** deletion



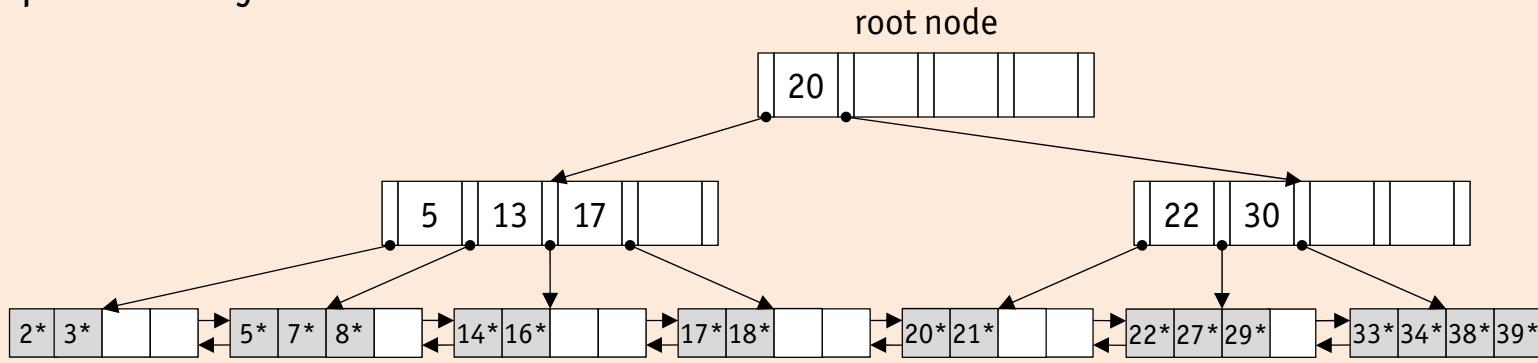
19. notice that the non-leaf node with entry 30 underflows

20. but its (only) sibling has two entries (17 and 20) to spare

B+ Tree Delete

Example: Deletion from a B+ tree with order $d = 2$ (cont'd)

21. redistribute entries by “**rotating**” entry **20** through the parent and pushing former parent entry **22** down



B+ Tree Delete

>Delete from B+ tree of degree d (non-leaf nodes: redistribution)

```
function delete ( $\uparrow parent$ ,  $\uparrow node$ ,  $k^*$ ) :  $\uparrow oldChild$ 
    if  $node$  is a non-leaf node, say  $N$  then ...
        if  $\uparrow oldChild$  is null then ...
        else ...
            if  $N$  has entries to spare then ...
            else get a sibling  $S$  of  $N$ , using  $\uparrow parent$ 
                if  $S$  has extra entries then
                    redistribute entries evenly between  $N$  and  $S$  through  $parent$ ;
                     $\uparrow oldChild \leftarrow$  null; return;
                else ...
            if  $node$  is a leaf node, say  $L$  then ...
    endproc;
```

Merge and Redistribution Effort

- Actual DBMS implementations often avoid the cost of merging and/or redistribution by relaxing the minimum occupancy rule

B+ tree deletion in DB2

- System parameter **MINPCTUSED** (*minimum percent used*) controls when the kernel should try a **leaf node merge** ("online index reorg"): particularly simple because of the sequence set pointers connecting adjacent leaves
- Non-leaf nodes are never merged:** only a "full index reorg" merges non-leaf nodes
- To improve concurrency, deleted index entries are merely **marked as deleted** and only removed later (IBM DB2 UDB type-2 indexes)

B+ Trees and Duplicates

- As discussed here, B+ tree **search**, **insert**, (and **delete**) procedures ignore the presence of **duplicate** key values
- This assumption is often reasonable
 - if the key field is a **primary key** for the data file (i.e., for the associated relation), the search keys k are unique by definition

Treatment of duplicate keys in DB2

Since duplicate keys add to the B+ tree complexity, IBM DB2 **forces uniqueness** by forming a composite key of the form $\langle k, id \rangle$, where id is the unique tuple identity of the data record with key k

Tuple identities are

1. **system-maintained** unique identifiers for each tuple in a table
2. **not** dependent on tuple order
3. **immutable**

B+ Trees and Duplicates

Other approaches alter the B+ tree implementation to add real support for duplicates

1. Use variant ③ to represent the index data entries k^*

$$k^* = \langle k, [rid_1, rid_2, \dots] \rangle$$

- each duplicate record with key field k makes the list of $rids$ grow
- key k is not repeated stored, which saves space
- B+ tree search and maintenance routines largely unaffected
- index data entry size varies, which affect the B+ tree **order** concept
- implemented, for example, in IBM Informix Dynamic Server

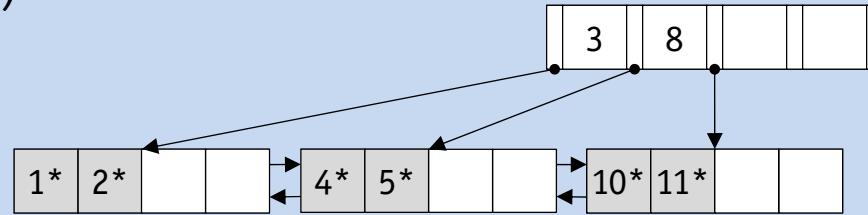
2. Treat duplicate value like any other value in the **insert** and **delete** procedures

- doing so affects the **search** procedure
- see example on following slides

B+ Trees and Duplicates

✍ Example: Impact on duplicate insertion on search (k)

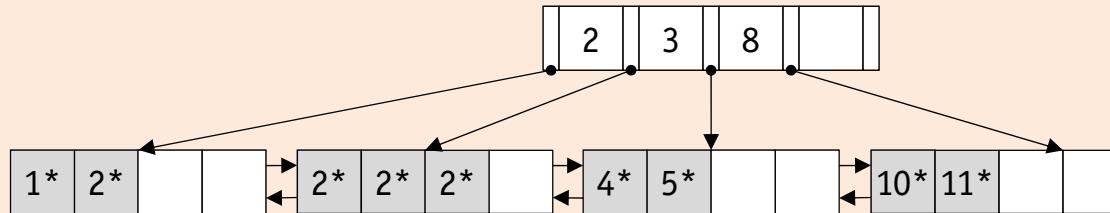
Insert **three records** with key $k = 2$ into the following B+ tree of order $d = 2$ (without using redistribution)



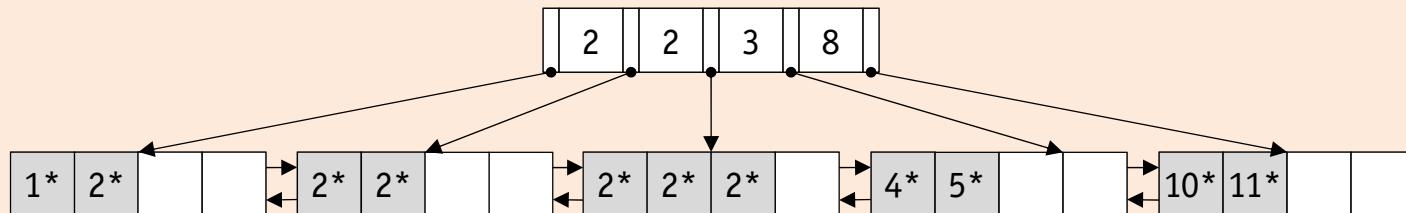
B+ Trees and Duplicates

☛ Example: Impact on duplicate insertion on search (k)

Below the B+ tree that results from the exercise on the previous slide is shown



The same B+ tree after inserting **another two records** with key $k = 2$, is shown below



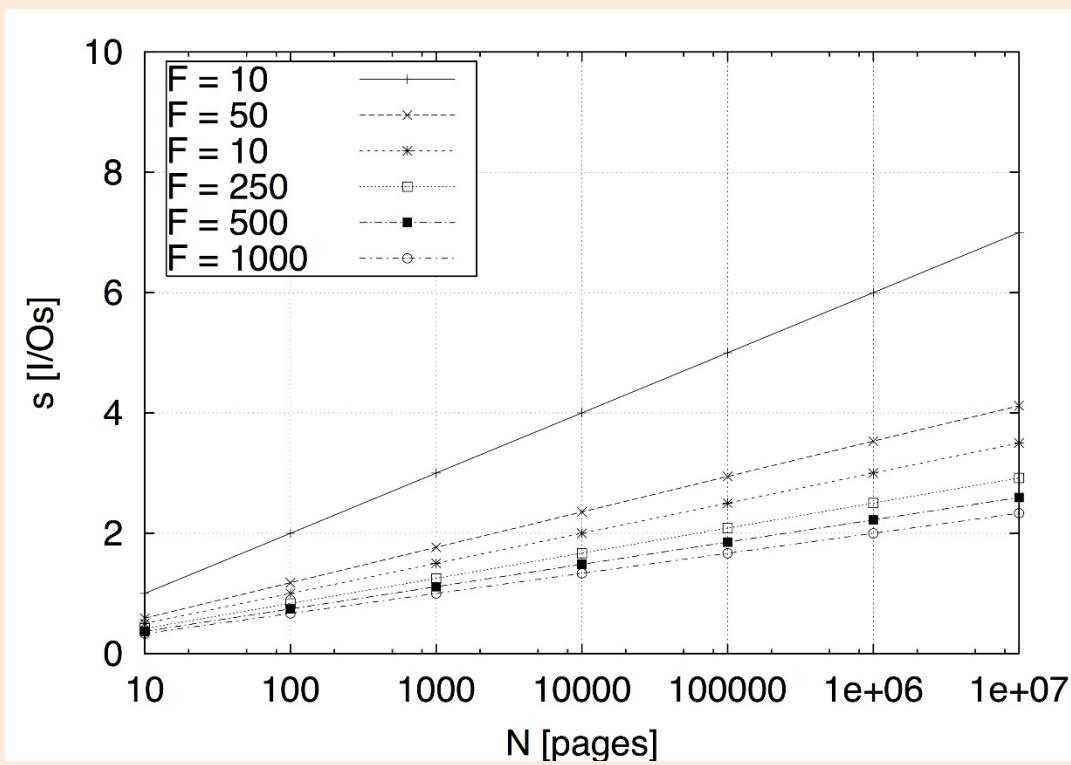
☛ **search (k)**

- **non-leaf nodes:** follow the left-most node pointer p_i , such that $k_i \leq k \leq k_{i+1}$
- **leaf nodes:** also check right sibling (and its right sibling and its right sibling and...)

Key Compression in B+ Trees

- Recall that the **fan-out** F is a deciding factor in the search I/O effort s in an ISAM or B+ tree for a file of N pages: $s = \log_F N$

☞ Tree index search effort dependent on fan-out F



⇒ It clearly pays off to invest effort and **try to maximize the fan-out F** of a given B+ tree implementation

Key Compression in B+ Trees

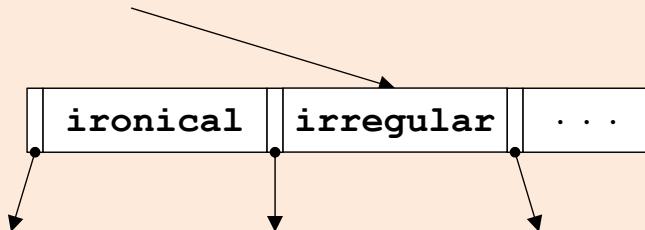
- Index entries in non-leaf B+ tree nodes are pairs $\langle k_i, \uparrow p_i \rangle$
 - **size of page pointers** depends on pointer representation of DBMS or hardware specifics
 - $|\uparrow p_i| \ll |k_i|$, especially for key field types like **CHAR** (·) or **VARCHAR** (·)
- To **minimize key size**, recall that key values in non-leaf nodes only direct calls to the appropriate leaf pages
 - actual key values are **not** needed
 - **arbitrary values** could be chosen as long as the separator property is maintained
 - for text attributes, a good choice can be **prefixes** of key values

Excerpt of search (k)

```
if  $k < k_1$  then ...  
else  
  if  $k \geq k_{2d}$  then ...  
  else  
    find  $i$  such that  $k_i \leq k < k_{i+1}$ ;  
    ...
```

Key Compression in B+ Trees

Example: Searching a B+ tree node with `VARCHAR(·)` keys

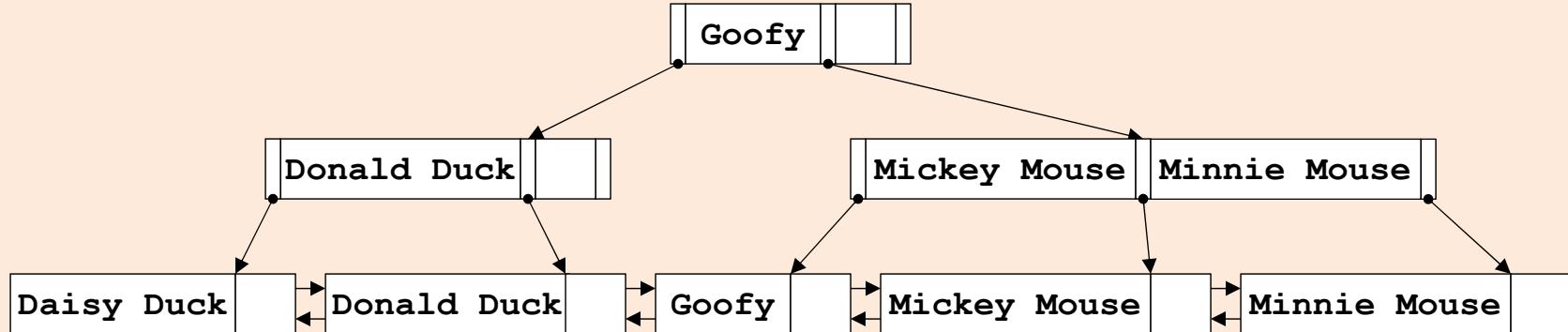


- To guide searches across this B+ tree node, it is sufficient to store the **prefixes `iro`** and **`irr`**
- B+ tree semantics must be preserved
 - all index entries stored in the sub-tree left of `iro` have keys $k < \text{iro}$
 - all index entries stored in the sub-tree right of `iro` have keys $k \geq \text{iro}$ (and $k < \text{irr}$)

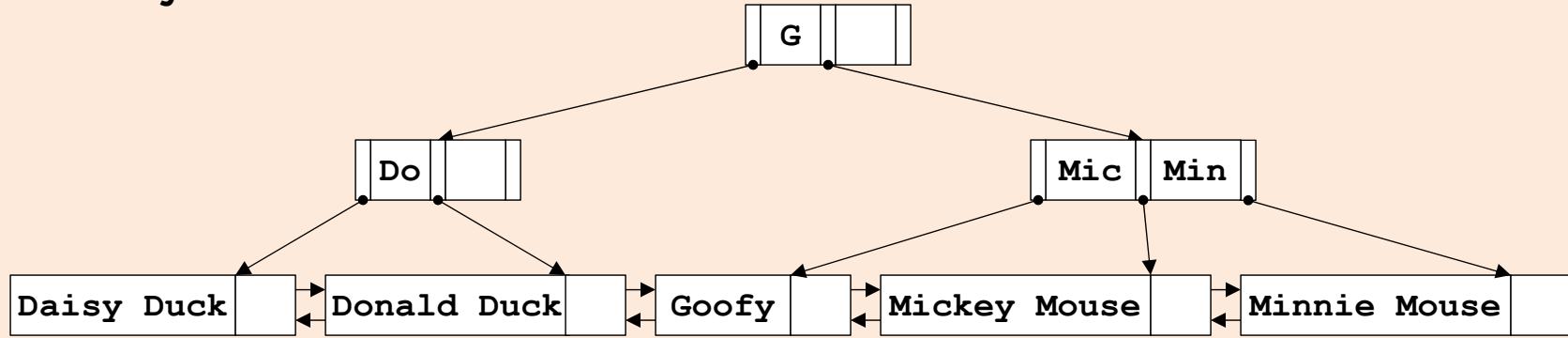
B+ Tree Key Suffix Truncation

Example: Key suffix truncation in a B+ tree with order $d = 1$

Before key suffix truncation



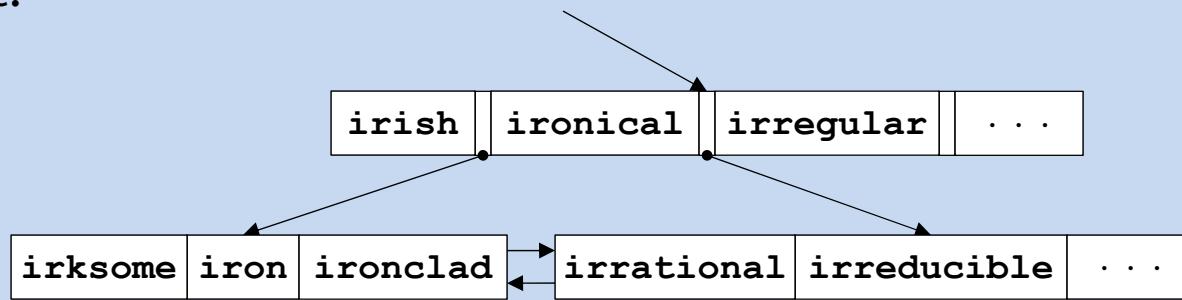
After key suffix truncation



B+ Tree Key Suffix Truncation

✍ Key suffix truncation

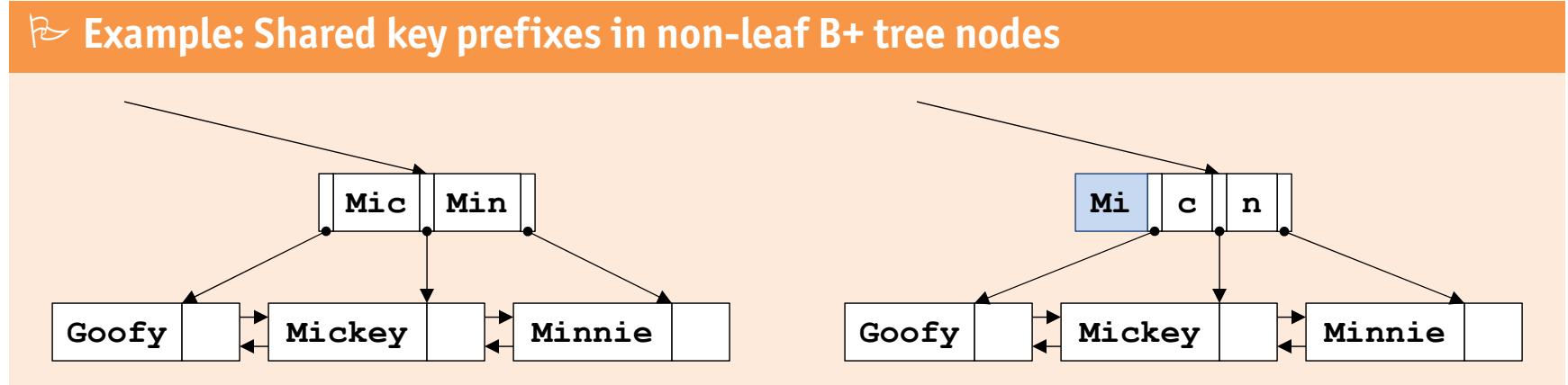
How would a B+ tree key compressor alter the key entries in the non-leaf node of this B+ tree excerpt?



Key Prefix Compression in B+ Trees

- Keys within a B+ tree node often share a **common prefix**

☛ Example: Shared key prefixes in non-leaf B+ tree nodes



- **Key prefix compression**
 - store common prefix only once (e.g., as “key” k_0)
 - keys have become highly discriminative now
- Violating the 50% occupancy rule can help to improve the effectiveness of prefix compression

B+ Tree Bulk Loading

⌚ Database log: table and index creation

```
CREATE TABLE t1 (id INT, text VARCHAR(10));
```

... insert 1,000,000 rows into table **t1** ...

```
CREATE INDEX t1_idx ON t1 (id ASC);
```

- Last SQL command initiates one million B+ tree **insert(·)** calls, a so-called index **bulk load**
- DBMS will traverse the growing B+ tree index from its root down to the leaf pages one million times

✍ This is bad...

...but at least, it is not as bad as swapping the order of row insertion and index creation. Why?

B+ Tree Bulk Loading

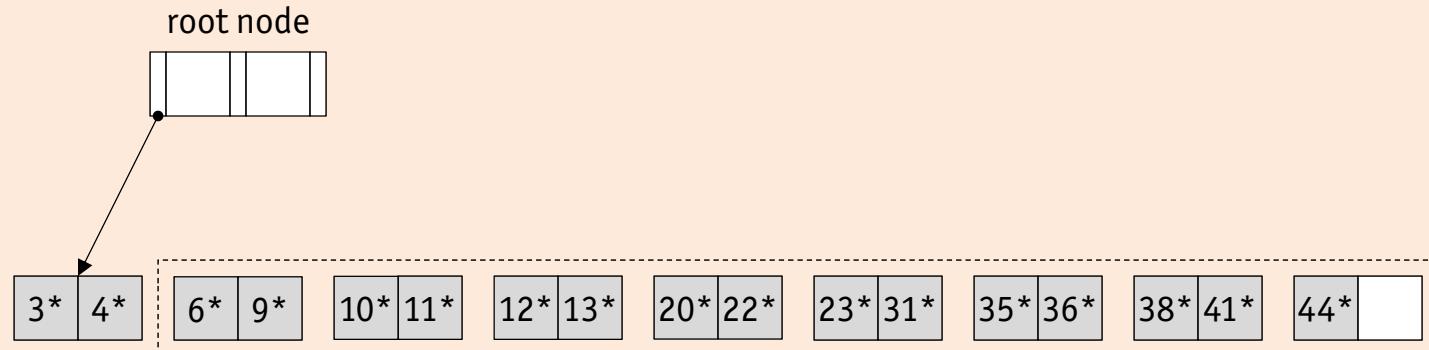
- Most DBMS provide a B+ tree **bulk loading utility** to reduce the cost of operations like the one on the previous slide

B+ tree bulk loading algorithm

1. for each record with key k in the data file, create a **sorted list** of pages of index leaf entries k^*
Note: for index variants ② or ③ this does **not** imply to sort the data file itself
(variant ① effectively creates a clustered index)
2. allocate an **empty index root node** and let its p_0 node pointer point to the first page of the sorted k^* entries

B+ Tree Bulk Loading

✍ Example: State of bulk load of a B+ tree with order $d = 1$ after Step 2



Index leaf pages that are not yet in the B+ tree are **framed**

✍ B+ tree bulk loading continued

Can you anticipate how the bulk loading process will proceed from this point?

B+ Tree Bulk Loading

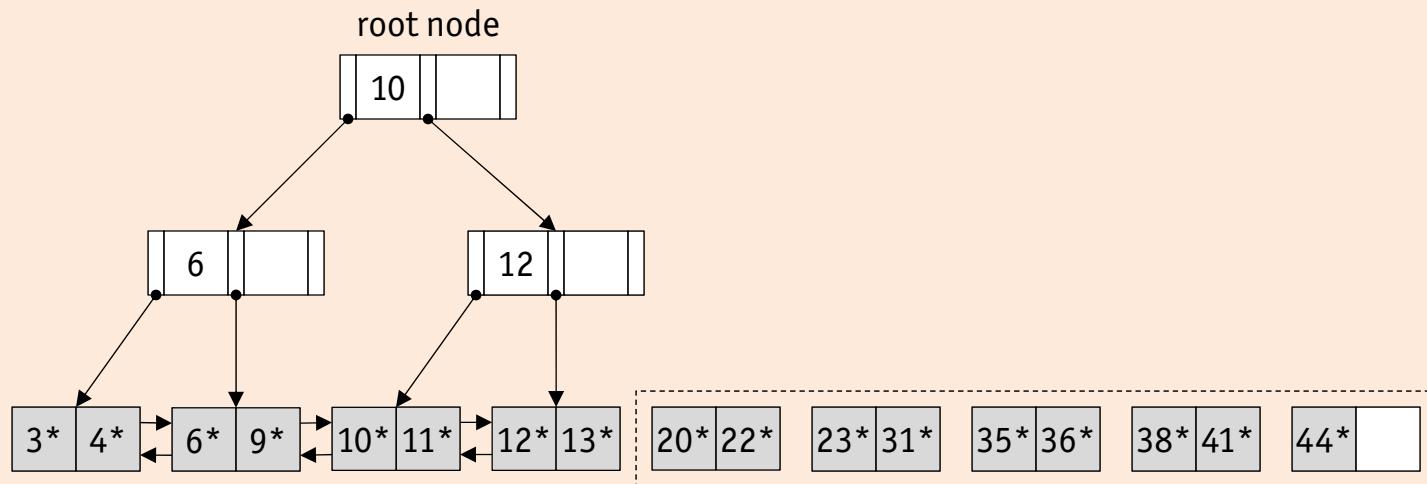
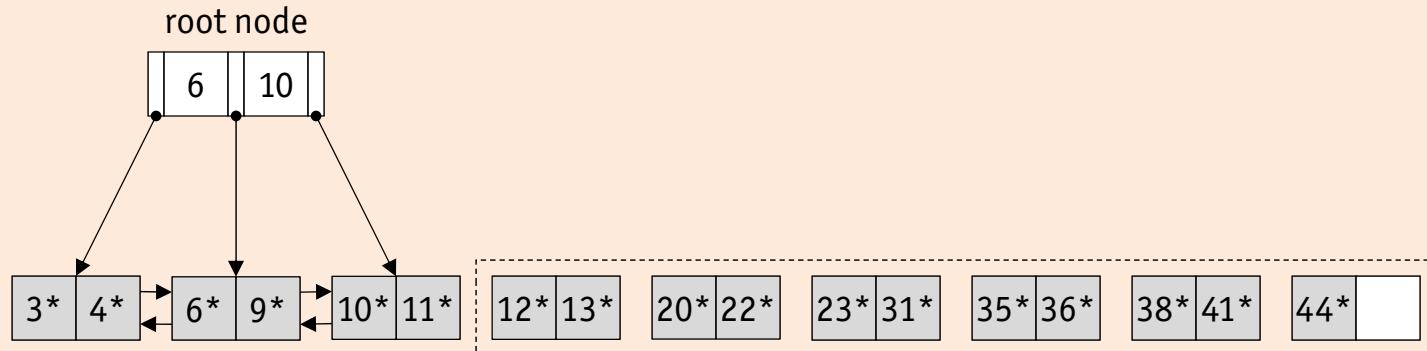
- As the k^* are sorted, any insertion will hit the **right-most index node** (just above the leaf level)
- A specialized **bulk_insert(·)** procedure avoids B+ tree root-to-leaf traversals altogether

☞ B+ tree bulk loading algorithm (cont'd)

3. for each leaf level node n , insert the index entry
 $\langle \text{minimum key on } n, \uparrow n \rangle$
into the right-most index node just above the leaf level
- ☞ the right-most node is filled **left-to-right**, splits only occur on the **right-most paths** from the leaf level up to the root

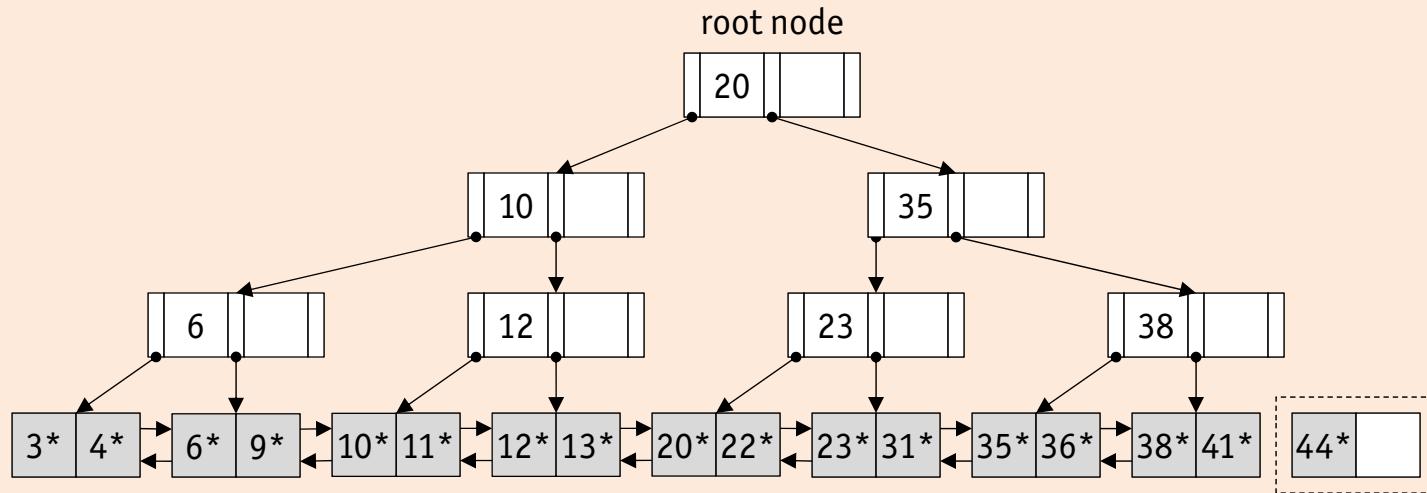
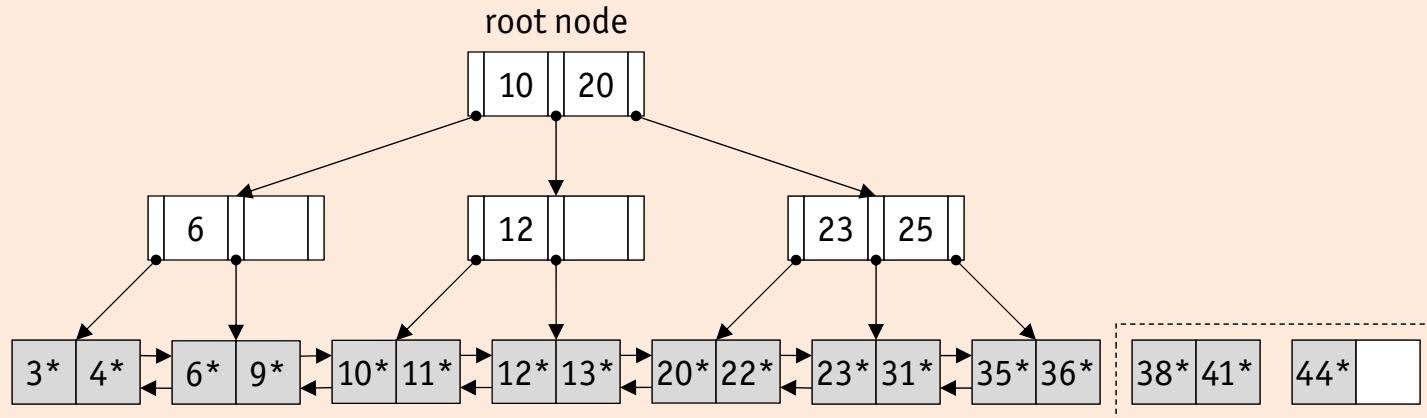
B+ Tree Bulk Loading

↷ Example: State of bulk load of a B+ tree with order $d = 1$ (cont'd)



B+ Tree Bulk Loading

Example: State of bulk load of a B+ tree with order $d = 1$ (cont'd)



B+ Tree Bulk Loading

- Bulk-loading is more (time) efficient
 - tree traversals are saved
 - less page I/O operations are necessary, i.e., buffer pool is used more effectively
- As seen in the example, bulk-loading is also more space-efficient as all leaf nodes are filled up completely

Space efficiency of bulk-loading

How would the resulting tree in the previous example look like, if you used the standard `insert(·)` routine on the sorted list of index entries k^* ?

B+ Tree Bulk Loading

- Bulk-loading is more (time) efficient
 - tree traversals are saved
 - less page I/O operations are necessary, i.e., buffer pool is used more effectively
- As seen in the example, bulk-loading is also more space-efficient as all leaf nodes are filled up completely

Space efficiency of bulk-loading

How would the resulting tree in the previous example look like, if you used the standard `insert(·)` routine on the sorted list of index entries k^* ?

↳ inserting sorted data into a B+ tree yields minimum occupancy of (only) d entries in all nodes

A Note on B+ Tree Order

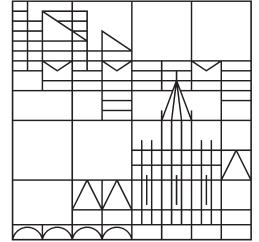
- Recall that B+ tree definition uses the concept of **order d**
- Order concept is useful for presenting B+ tree algorithms, but is hardly every used in practical implementations
 - key values may often be of variable length
 - duplicates may lead to variable number of rids in an index entry k^* according to variant ③
 - leaf and non-leaf nodes may have different capacities due to index entries of variant ①
 - key compression may introduce variable-length separator values
- Therefore, the order concept is relaxed in practice and replaced with a physical space criterion, e.g., every node needs to be at least half-full

A Note on Clustered Indexes

- Recall that a clustered index stores actual data records inside the index structure (variant ① entries)
- In case of a B+ tree index, splitting and merging leaf nodes **moves data records** from one page to another
 - depending on the addressing scheme used, *rid* of a record may change if it is moved to another page
 - even with the TID addressing scheme (records can be moved within a pages, uses forwarding address to deal with moves across pages), the performance overhead may be intolerable
 - some systems use the search key of the clustered index as a (location independent) record address for other, non-clustered indexes in order to avoid having to update other indexes or to avoid many forwards

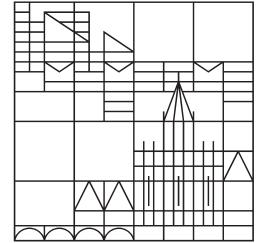
B+ Tree Invariants

- **Order:** d
- **Occupancy**
 - each non-leaf node holds at least d and at most $2d$ keys
(exception: root node can hold at least 1 key)
 - each leaf node holds between d and $2d$ index entries
- **Fan-out:** each non-leaf node holding m keys has $m + 1$ children
- **Sorted order**
 - all nodes contain entries in ascending key-order
 - child pointer p_i ($1 \leq i < m$) if an internal node with m keys k_1, \dots, k_m leads to a sub-tree where all keys k are $k_i \leq k < k_{i+1}$
 - p_0 points to a sub-tree with keys $k < k_1$ and p_m to a sub-tree with keys $k \geq k_m$
- **Balance:** all leaf nodes are on the same level
- **Height:** $\log_F N$
 - N is the total number of index entries/record and F is the average fan-out
 - because of high fan-out, B+ trees generally have a low height



Database System Architecture and Implementation

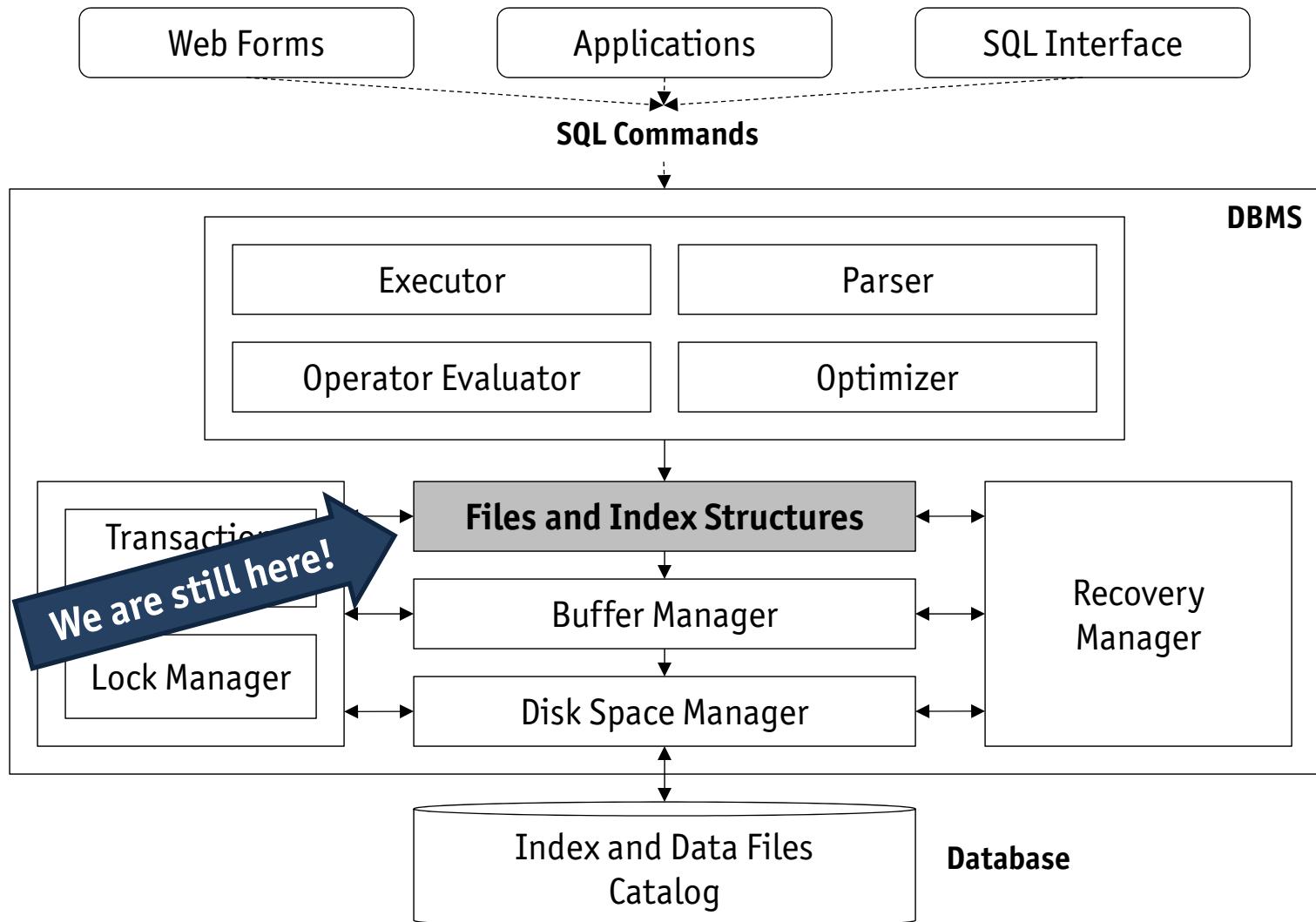
TO BE CONTINUED...



Database System Architecture and Implementation

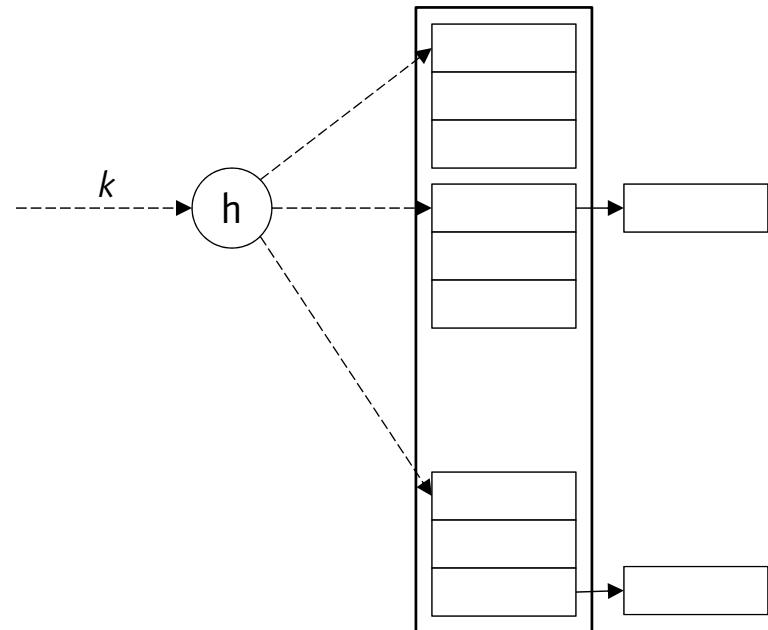
Module 4
Hash-Based Indexes
November 25, 2013

Orientation



Module Overview

- Overview of hash-based indexing
- Static hashing
- Extendible hashing
- Linear hashing



Hash-Based Indexing

Equality selection

```
SELECT *
FROM   R
WHERE  A = k
```

- In addition to tree-structured indexes (B+ trees), typical DBMS also provide support for **hash-based index structures**
 - “unbeatable” when it comes to support **equality selections**
 - can answer equality such queries using a **single I/O operation** (more precisely 1.2 operations), if the hash index is carefully maintained while the underlying data file for relation **R** grows and shrinks
 - other query operations, like (equality joins) internally require a **large number of equality tests**
 - presence (or absence) of support for hash indexes can make a real difference in such scenarios

Hash Indexes vs. B+ Tree Indexes

- Locating a record with key k
 - B+ tree search **compares k** to other keys k' organized in a (tree-shaped) search data structure
 - hash indexes **use the bits of k itself** (independent of all other stored records and their keys) to find (i.e., **compute the address of**) the record
- Range queries
 - B+ trees handle range queries efficiently by leveraging the **sequence set**
 - hash indexes provide **no support for range queries** (hash indexes are also known as **scatter storage**)

Overview of Hash-Based Indexing

- Static hashing
 - used to illustrate **basic concepts** of hashing
 - much like ISAM, static hashing does **not** handle updates well
- Dynamic hashing
 - **extendible hashing** and **linear hashing**
 - refine the hashing principle and adapt well to record insertions and deletions
- Hashing granularity
 - in contrast to in-memory applications where record-oriented hashing prevails, DBMS typically use **bucket-oriented hashing**
 - a bucket **can contain several records** and may have an **overflow chain**
 - a bucket is a **(set of) page(s)** on secondary memory

Static Hashing

Build a static hash index on attribute A

1. Allocate a fixed area of N (successive) disk pages, the so-called **primary buckets**
2. In each bucket, install a pointer to a chain of **overflow pages**
initially, set this pointer to **null**
3. Define a **hash function** h with *range* $[0, \dots, N - 1]$, the *domain* of h is the type of **A**,
e.g.,

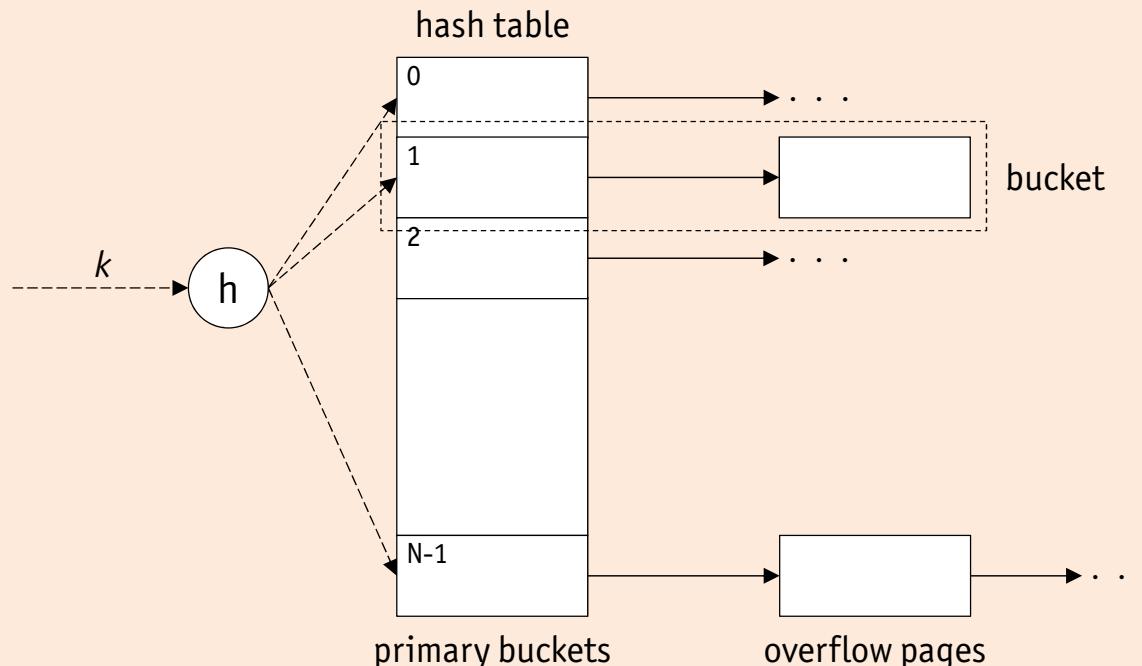
$$h : \text{INTEGER} \rightarrow [0, \dots, N - 1]$$

if A has the SQL type **INTEGER**

- Evaluating the hash function h on a given data value is **cheap**: it only involves a few CPU instructions

Static Hashing

Static hash table



- A primary bucket and its chain of overflow pages is referred to as a **bucket**
- Each bucket contains index entries k^* , which can be implemented using any of the variants ①, ②, and ③

Static Hashing

- Operations **hsearch**(k) , **hinsert**(k) , and **hdelete**(k) for a record with key $\mathbf{A} = k$ depend on the **hashing scheme**

Static hashing scheme

1. apply hash function h to key value, i.e., compute $h(k)$
2. access primary bucket page with number $h(k)$
3. search, insert, or delete the record with key k on that page or, if necessary, access the overflow chain of bucket $h(k)$

- If the hashing scheme works well and overflow chain access can be avoided altogether
 - **hsearch**(k) requires a **single I/O operation**
 - **hinsert**(k) and **hdelete**(k) require **two I/O operations**

Collisions and Overflow Chains

- At least for static hashing, **overflow chain management** is important
 - generally, we do **not** want hash function h to avoid collisions, i.e.,

$$h(k) = h(k') \text{ even if } k \neq k'$$

(otherwise as many primary buckets as different keys in the data file or even in **A**'s domain would be required)

- however, it is important that h **scatters** the domain of **A** **evenly across** $[1, \dots, N - 1]$ in order to avoid long overflow chains for few buckets
- otherwise, the I/O behavior of the hash table becomes non-uniform and unpredictable for a query optimizer
- unfortunately, such “good” hash functions are hard to discover

Probability of Collisions

The birthday paradox

Consider the people in a group as the **domain** and use their birthday as **hash function h** (i.e., $h : \text{Person} \rightarrow [0, \dots, 364]$)

*If the group has 23 or more members, chances are 50% that two people share the same birthday (**collision**)*

Check for yourself

1. Compute the probability that n people all have different birthdays

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ P(n - 1) \times \frac{365 - (n - 1)}{365} & \text{if } n > 1 \end{cases}$$

2. Try to find “birthday mates” at the next larger party

Hash Functions

- If key values were purely **random**, a “good” hash function could simply extract a few bits and use them as a hash value
 - key value distributions found in databases are **not** random
 - it is **impossible** to generate truly random hash values from non-random key values
- But is it possible to define hash functions that scatter even better than a random function?
- Fairly good hash functions can be found rather easily by
 - **division** of the key value
 - **multiplication** of the key value

Hash Functions

Design of a hash function

1. By division: simply define

$$h(k) = k \bmod N$$

- this guarantees that range of $h(k)$ to be $[0, \dots, N - 1]$
- **prime numbers** work best for N
- choosing $N = 2^d$ for some d effectively considers the least d bits of k only

2. By multiplication: extract the fractional part of $Z \cdot k$ (for a specific Z) and multiply by hash table size N

$$h(k) = \lfloor N \cdot (Z \cdot k - \lfloor Z \cdot k \rfloor) \rfloor$$

- the (inverse) **golden ratio** $Z = \frac{2}{\sqrt{5}+1} \approx 0.6180339887$ is a good choice (according to D. E. Knuth, “*Sorting and Searching*”)
- for $Z = \frac{1}{2^w}$ and $N = 2^d$ (w is the number of bits in a CPU word), we simply have $h(k) = msb_d(\frac{1}{2^w} \cdot k)$, where $msb_d(x)$ denotes the d **most significant bits** of x (e.g., $msb_3(42) = 5$)

Static Hashing and Dynamic Files

- Effects of dynamic files on static hashing
 - if the underlying **data file grows**, developing overflow chains spoil the otherwise predictable I/O behavior (1-2 I/O operations)
 - if the underlying **data file shrinks**, a significant fraction of primary hash buckets may be (almost) empty and waste space
 - in the worst case, a hash table can **degrade into a linear list** (one long chain of overflow buckets)
- As in the case of ISAM case, static hashing has **advantages** when it comes to concurrent access
 - allocating a hash table of size 125% of the expected data capacity, i.e., only 80% full, will typically give good results
 - data file could be rehashed periodically to restore this ideal situation (**expensive** operation and the index **cannot be used** during rehashing)

Dynamic Hashing

- Dynamic hashing scheme have been devised to overcome these limitations of static hashing by
 - **combining** the use of hash functions with directories that guide the way to the data records (e.g., extendible hashing)
 - **adapting** the hash function (e.g., linear hashing)

Curb your enthusiasm!

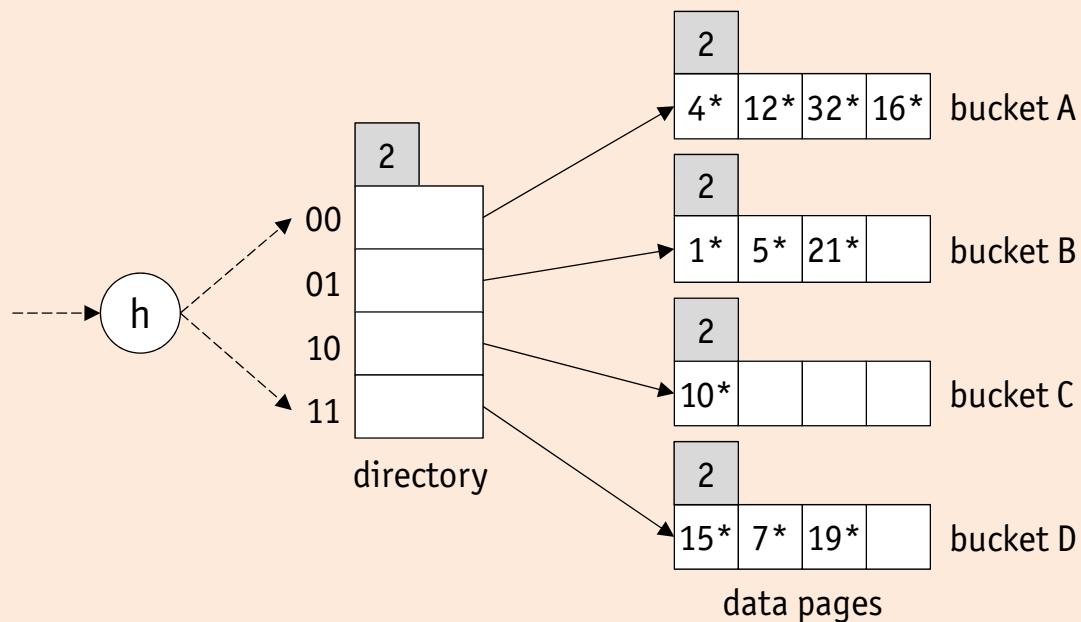
Stand-alone hash indexes are **very rare**!

- **Microsoft SQL Server, Oracle, and DB2:** support for B+ tree indexes only
 - **PostgreSQL:** support for both B+ tree and hash indexes (linear hashing)
 - **MySQL:** depending on storage engine, both B+ tree and hash indexes are supported
 - **Berkeley DB:** support for both B+ tree and hash indexes (linear hashing)
- ↳ However, almost all of these systems implement the **Hybrid Hash Join** (physical) operator that uses hashing to compute the equijoin of two relations (see L. D. Shapiro: “**Join Processing in Database Systems with Large Main Memories**”, 1986)

Extendible Hashing

- **Extendible hashing** adapts to growing (or shrinking) data files
- To keep track of the actual primary buckets that are part of the current hash table, an **in-memory bucket directory** is used

☞ Example: Extendible hash table setup (ignore the 2 fields for now)



Note: This figure depicts the entries as $h(k)^*$, not k^*

Extendible Hashing Search

☛ Search for a record with key k

1. Apply h , i.e., compute $h(k)$
2. Consider the last $\boxed{2}$ bits of $h(k)$ and follow the corresponding directory pointer to find the bucket

- The meaning of the \square fields might become clear now

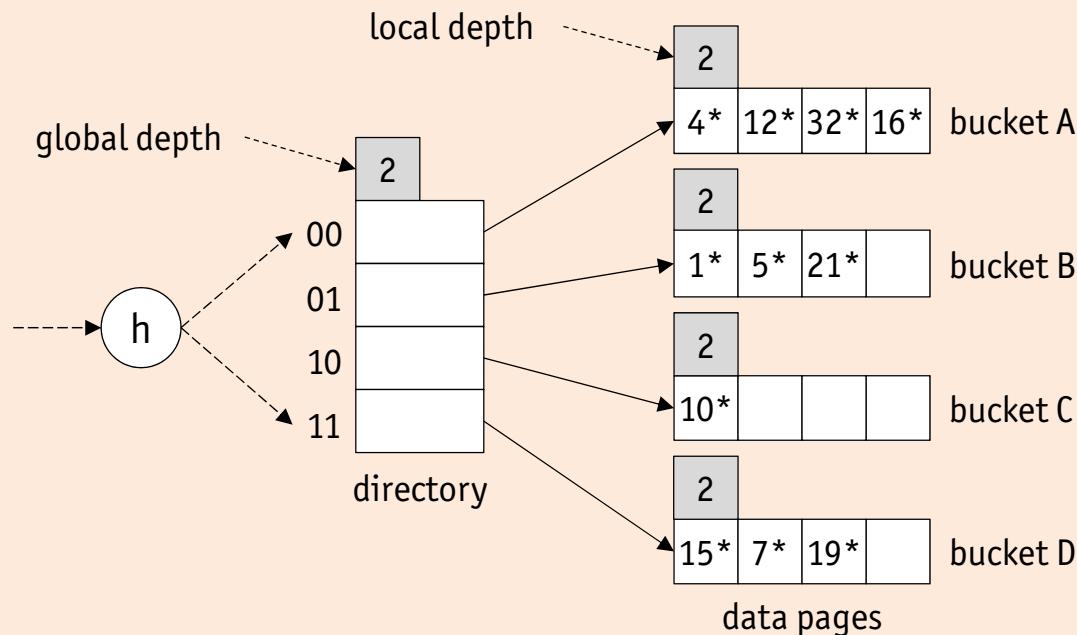
☛ Global and local depth annotations

- **Global depth** (\boxed{n} at hash directory)
*Use the last n bits of $h(k)$ to lookup a bucket pointer in the directory
(the directory size is 2^n)*
- **Local depth** (\boxed{d} at individual buckets)
The hash values $h(k)$ of all entries in this bucket agree on their last d bits

Extendible Hashing Search

Example: Find a record with key k such that $h(k) = 5$

Example: To find a record with key k such that $h(k) = 5 = 101_2$, follow the second directory pointer ($101_2 \wedge 11_2 = 01_2$) to bucket B, then use entry 5^* to access the record



Extendible Hashing Search

Searching in extendible hashing

```
function hsearch (k) : ↑bucket
    n ← n;
    b ← h(k) & (2n - 1);
    ↑bucket ← bucket[b];
end
```

(global depth of hash directory)
(mask all but the low n bits)

- Remarks
 - $bucket[0, \dots, 2^n - 1]$ is an **in-memory array** whose entries point to the hash buckets
 - search returns a pointer to hash bucket containing **potential** hit(s)
 - **&** and **|** denote **bit-wise and** and **bit-wise or** (like in C, C++, Java, etc.)

Extendible Hashing

☞ Insert a record with key k

1. Apply h , i.e., compute $h(k)$
2. Use the last bits of $h(k)$ to lookup the bucket pointer in the directory
3. If the **primary bucket** still has capacity, store $h(k)^*$ in it

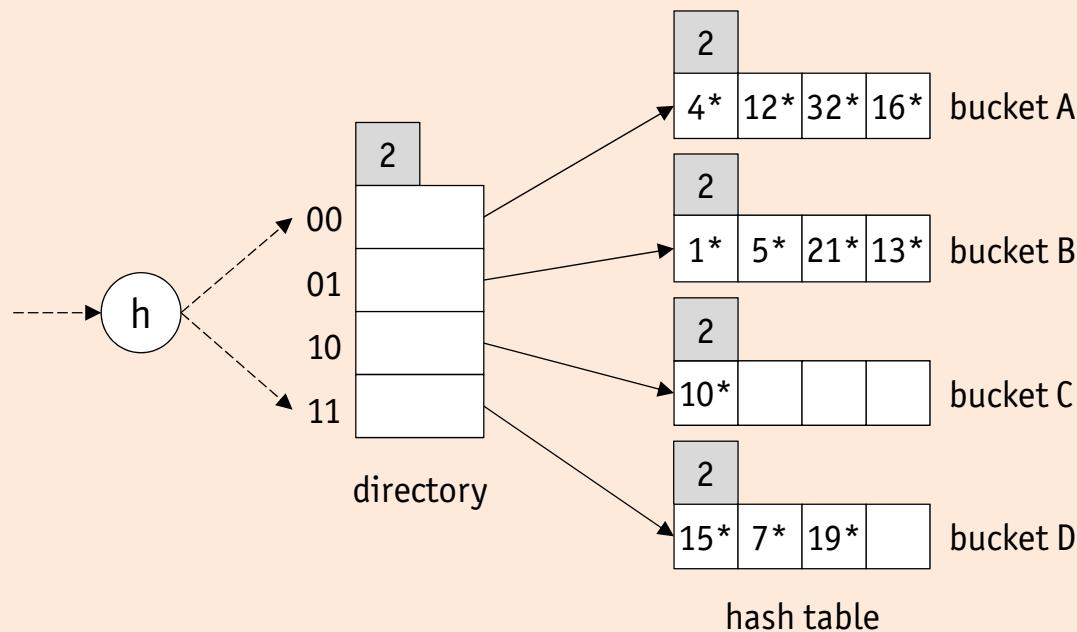
Otherwise...?

- We *cannot* start an overflow chain hanging off the primary bucket as that would compromise uniform I/O behavior
- We *cannot* place $h(k)^*$ in another primary bucket since that would invalidate the hashing principle

Extendible Hashing Insert

Example: Insert a record with $h(k) = 13$

To insert a record with key k such that $h(k) = 13 = 1101_2$, follow the second directory pointer (entry 01) to bucket B (which still has empty slots) and place 13^* there



Extendible Hashing Insert

Example: Insert a record with $h(k) = 20$

Inserting a record with key k such that $h(k) = 20 = 101\underline{00}_2$ causes an **overflow in primary bucket A** and therefore a **bucket split** for A

1. Split bucket A by creating a new bucket A2 and use bit position $d + 1$ to redistribute the entries

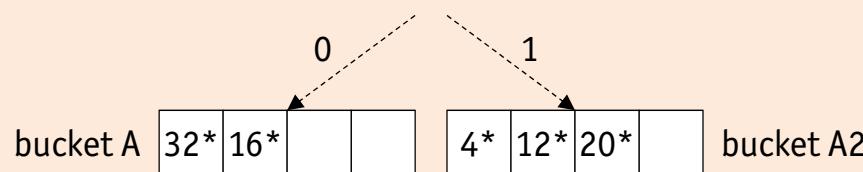
$$4 = \underline{100}_2$$

$$12 = \underline{1100}_2$$

$$32 = \underline{100000}_2$$

$$16 = \underline{10000}_2$$

$$20 = \underline{10100}_2$$

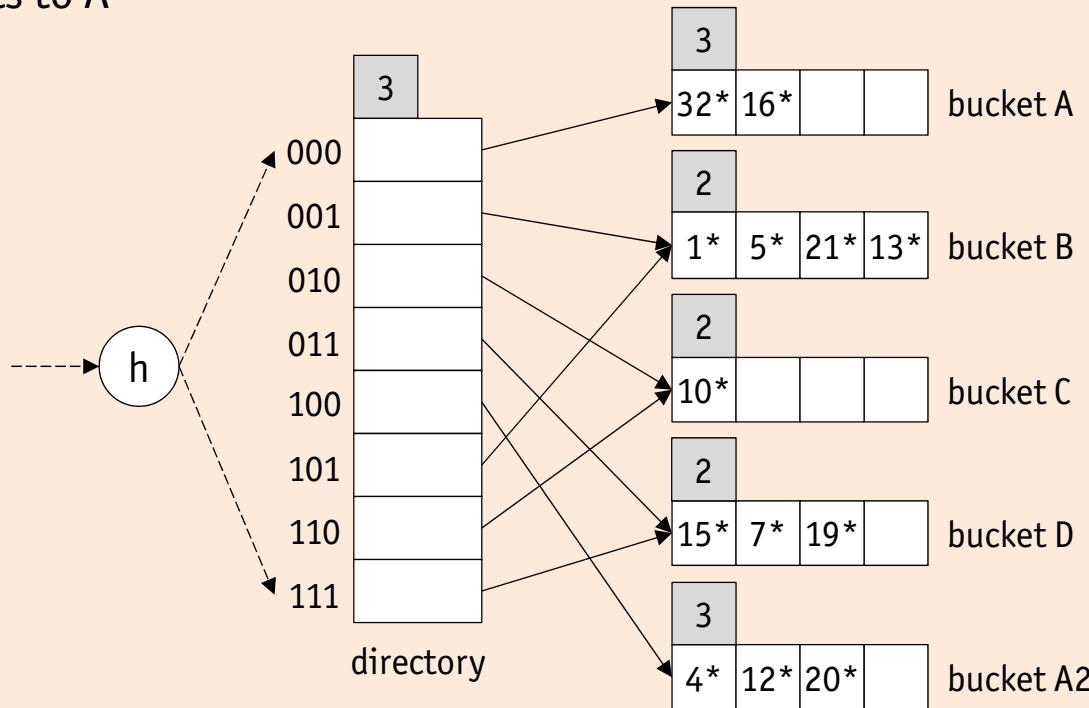


Note that now $\boxed{3}$ bits are used to discriminate between the old bucket A and the new split bucket A2

Extendible Hashing Insert

Example: Insert a record with $h(k) = 20$

2. To address the new bucket, the directory needs to be **doubled** by simply copying its original pages (bucket pointer lookups now use $2 + 1 = 3$ bits)
3. Let bucket pointer for 100_2 point to A2, whereas the directory pointer for 000_2 still points to A



Extendible Hashing Insert

Doubling the directory

In the previous example, the directory had to be double to address the new split bucket. Is doubling the directory always necessary when a bucket is split? Or, how could you tell whether directory doubling is required or not?

Extendible Hashing Insert

Doubling the directory

In the previous example, the directory had to be double to address the new split bucket. Is doubling the directory always necessary when a bucket is split? Or, how could you tell whether directory doubling is required or not?

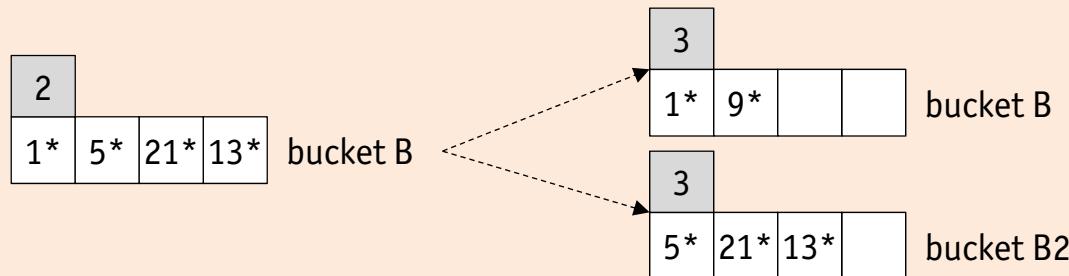
- ⇒ If the local depth of the split bucket is smaller than then global depth, i.e., $d < n$, directory doubling is **not** necessary

Extendible Hashing Insert

- If the local depth of the split bucket is smaller than the global depth, i.e., $d < n$, directory doubling is **not** necessary

Example: Insert a record with $h(k) = 9$

- Insert record with key k such that $h(k) = 9 = \underline{1001}_2$
- The associated bucket B is split by creating a new bucket B2 and redistributing the entries

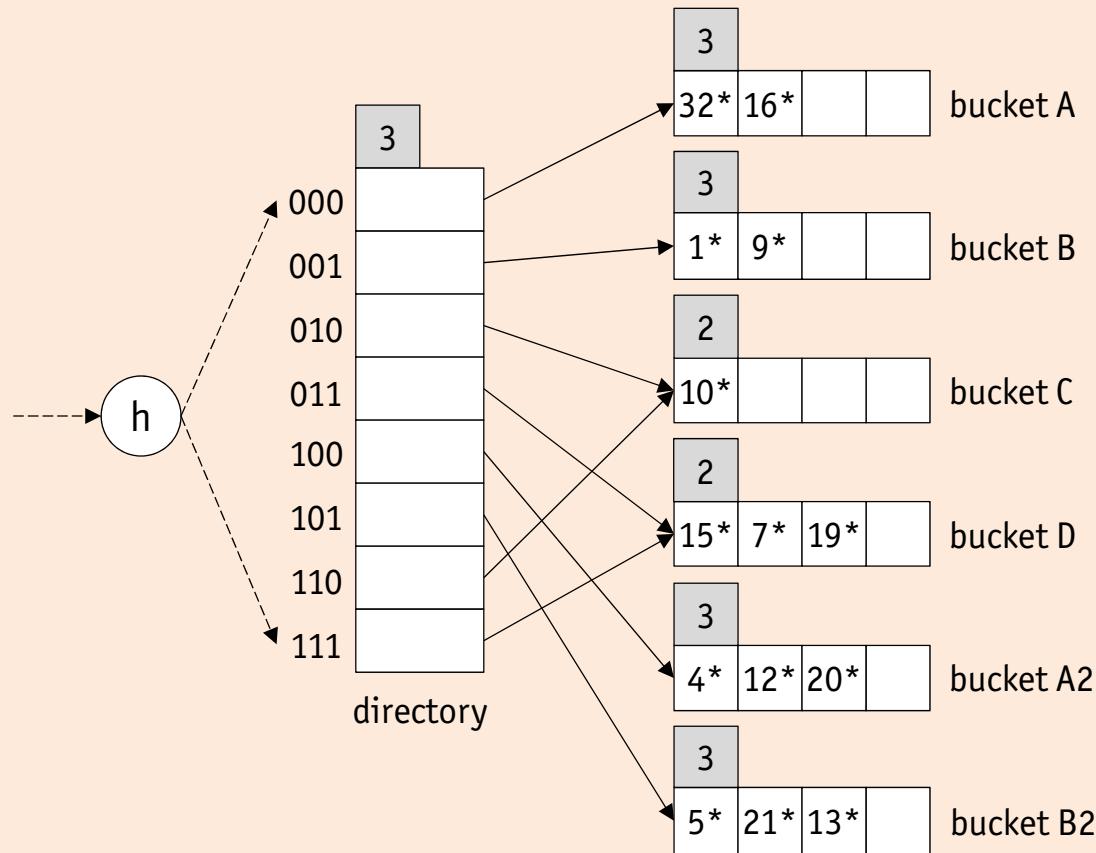


- The new local depth of B and B2 is 3 and thus does **not** exceed the global depth of 3

↳ Modifying the directory's bucket pointer for $\underline{101}_2$ is sufficient (see next slide)

Extendible Hashing Insert

Example: Insert a record with $h(k) = 9$ (cont'd)



Extendible Hashing Insert

💻 Insert in extendible hashing

```
function hinsert ( $k^*$ )
     $n \leftarrow$  n;                                (global depth of hash directory)
     $b \leftarrow \text{hsearch}(k)$  ;
    if  $b$  has capacity then
        place  $k^*$  in bucket  $b$  ;
    else ...
end
```

Extendible Hashing Insert

💻 Insert in extendible hashing (cont'd)

```
function hinsert (k*)
    n ← n;                                (global depth of hash directory)
    b ← hsearch (k) ;
    if b has capacity then ...
    else
        d ← d;                                (local depth of bucket b)
        create a new empty bucket b2 ;
        foreach k'* in bucket b do          (redistribute entries of bucket b including k*)
            if  $h(k') \& 2^d \neq 0$  then move k'* to bucket b2 ;
            d ← d + 1;                            (new local depths for buckets b and b2)
            if n < d + 1 then                  (directory has to be doubled)
                allocate  $2^n$  directory entries bucket[ $2^n, \dots, 2^{n+1} - 1$ ] ;
                copy bucket[0, ...,  $2^n - 1$ ] into bucket[ $2^n, \dots, 2^{n+1} - 1$ ] ;
                n ← n + 1 ;
                bucket[( $h(k) \& (2^n - 1)$ ) |  $2^n$ ] ← @ (b2) ;
    end
```

Overflow Chains in Extendible Hashing

Overflow chains

Extendible hashing uses overflow chains hanging off a bucket only as a last resort. Under which circumstances will extendible hashing create an overflow chain?

Overflow Chains in Extendible Hashing

Overflow chains

Extendible hashing uses overflow chains hanging off a bucket only as a last resort.
Under which circumstances will extendible hashing create an overflow chain?

- ⇒ If considering $d + 1$ bits does **not** lead to a satisfying record distribution in procedure **hinsert**(k^*) (skewed data, hash collisions)

Extendible Hashing Delete

- Routine **hdelete** (k^*) locates and removes entry k^*
 - deleting an entry k^* from a bucket may leave this bucket **empty**
 - an empty buckets can be **merged** with its split bucket
 - however, this step is often **omitted** in practice

Delete in extendible hashing

When is the **local depth** decreased?

When is the **global depth** decreased?

Extendible Hashing Delete

- Routine **hdelete** (k^*) locates and removes entry k^*
 - deleting an entry k^* from a bucket may leave this bucket **empty**
 - an empty buckets can be **merged** with its split bucket
 - however, this step is often **omitted** in practice

Delete in extendible hashing

When is the **local depth** decreased?

↳ when a bucket is merged with its split bucket, the local depth of the merged bucket is decreased

When is the **global depth** decreased?

↳ if every directory entry points to the same bucket as its split directory entry (i.e., 0 and 2^n point to bucket A, 1 and $2^n + 1$ point to bucket B, etc.), the directory can be halved and the global depth decreased

Linear Hashing

- Similar to extendible hashing, **linear hashing** can adapt its underlying data structure to record insertions and deletions
 - linear hashing **does not need a hash directory** in addition to the actual hash table buckets
 - linear hashing can define **flexible criteria** that determine when a bucket is to be split
 - linear hashing may perform bad if the **key distribution is skewed**

Linear Hashing

- Linear hashing uses an **ordered family of hash functions**
 - sequence of hash functions h_0, h_1, h_2, \dots (subscript is often called *level*)
 - range of $h_{level+1}$ is **twice as large** as range of h_{level} (for $level = 0, 1, 2, \dots$)

Hash Function Family

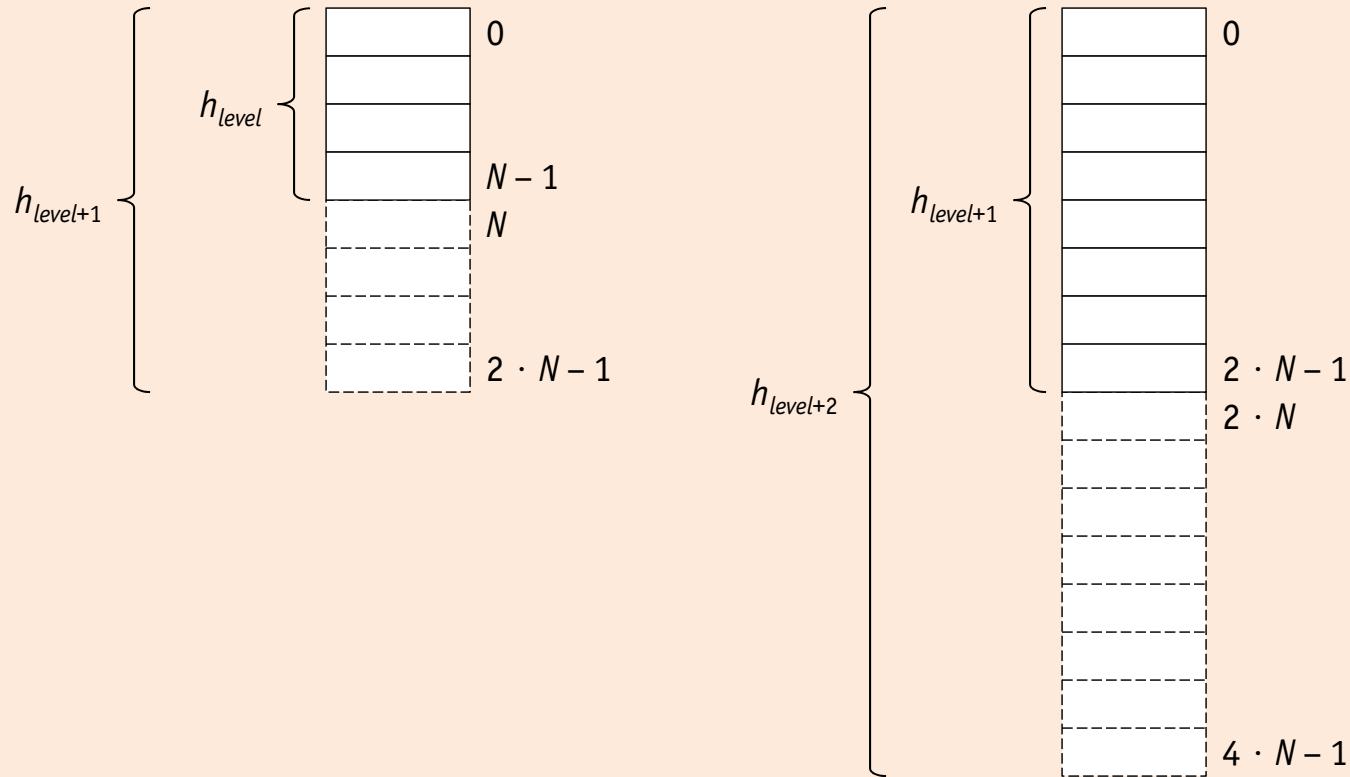
Given an initial hash function h and an initial hash table size N , one approach to define such a family of hash functions h_0, h_1, h_2, \dots would be

$$h_{level}(k) = h(k) \bmod (2^{level} \cdot N)$$

where $level = 0, 1, 2, \dots$

Linear Hashing

Example: h_{level} with range $[0, \dots, N - 1]$



Linear Hashing

↷ Basic linear hashing scheme

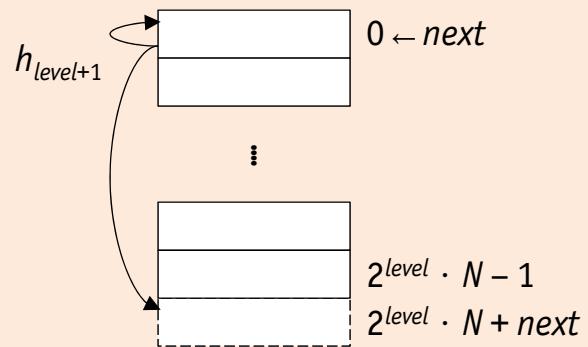
1. Initialize $level \leftarrow 0$ and $next \leftarrow 0$
2. The **current hash function** in use for searches (insertions/deletions) is h_{level} ,
active hash buckets are those in the range of h_{level} , i.e., $[0, \dots, 2^{level} \cdot N - 1]$
3. Whenever the current **hash table overflows**
 - insertions filled a primary bucket beyond c% occupancy
 - overflow chain of a bucket grew longer than p pages
 - or (*insert your criterion here*)the bucket **at hash table position $next$ is split**

Note: In general the bucket that is split is **not** the bucket that triggered the split!

Linear Hashing

↷ Bucket split

1. **Allocate a new bucket and append** it to the hash table at position $2^{level} \cdot N = next$
2. **Redistribute** the entries in bucket $next$ by **rehashing** them via $h_{level+1}$ (some entries will remain in bucket $next$, some will move to bucket $2^{level} \cdot N + next$)



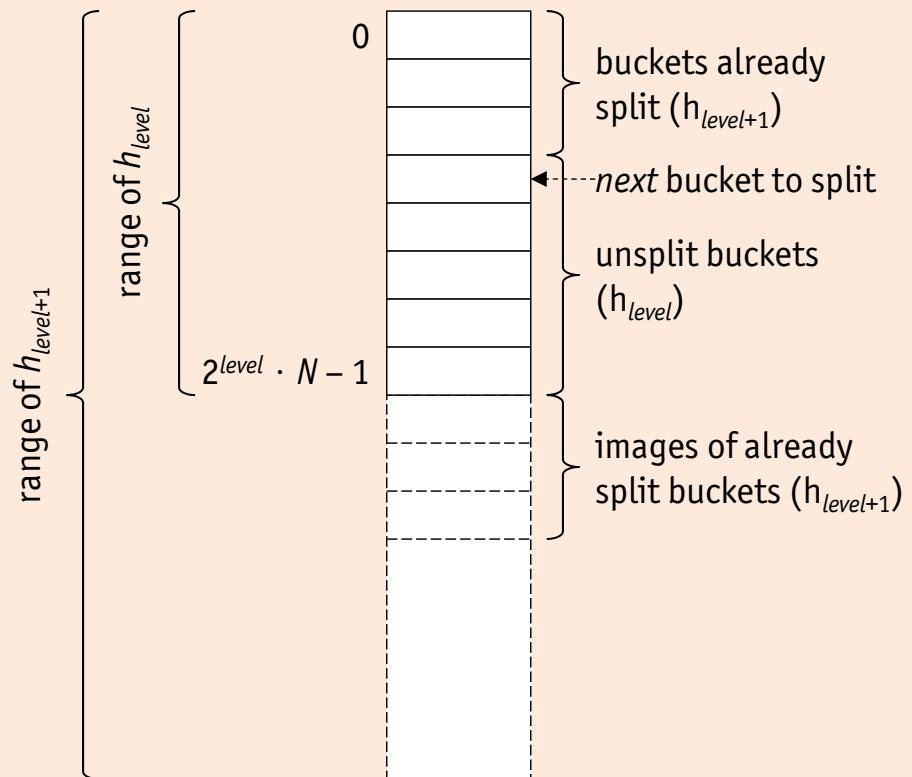
3. **Increment** $next$ by 1

⇒ All buckets with positions $< next$ have been rehashed

Linear Hashing

↷ Rehashing

With every bucket split, next walks down the hash table. Therefore, hashing via h_{level} (search, insert, and delete) needs to take **current next position** into account.



$h_{level}(k)$ {
 < *next*: bucket already split, **rehash**: find record in bucket $h_{level+1}(k)$
 \geq *next*: bucket not yet split, i.e., **bucket found**

Linear Hashing

✍ Split rounds: what happens if $next$ is incremented beyond the hash table size?

A bucket split increments $next$ by 1 to mark the next bucket to be split. How would you propose to handle the situation when $next$ is incremented **beyond** the currently last hash table position, i.e.,

$$next > 2^{level} \cdot N - 1?$$

Linear Hashing

✍ Split rounds: what happens if $next$ is incremented beyond the hash table size?

A bucket split increments $next$ by 1 to mark the next bucket to be split. How would you propose to handle the situation when $next$ is incremented **beyond** the currently last hash table position, i.e.,

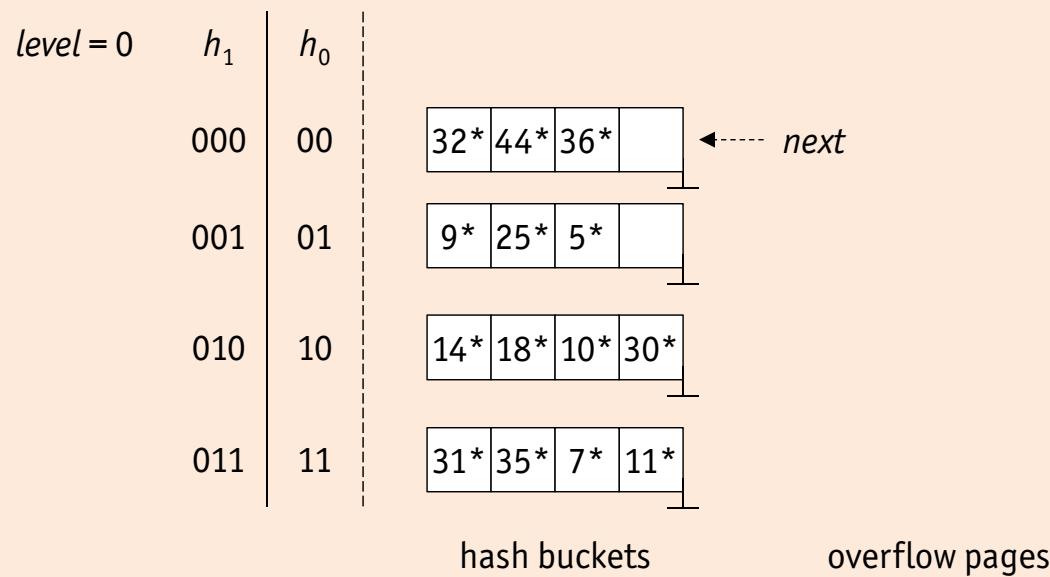
$$next > 2^{level} \cdot N - 1?$$

- If $next > 2^{level} \cdot N - 1$, **all buckets** in the current hash table are hashed via function $h_{level+1}$
 - Linear hashing proceeds in a **round-robin** fashion: if $next > 2^{level} \cdot N - 1$
 1. **increment** $level \leftarrow level + 1$
 2. **reset** $next \leftarrow 0$ (start splitting from top of hash table again)
- ↳ In general, an overflowing bucket is **not split immediately**, but—due to round-robin splitting—no later than in the following round

Linear Hashing

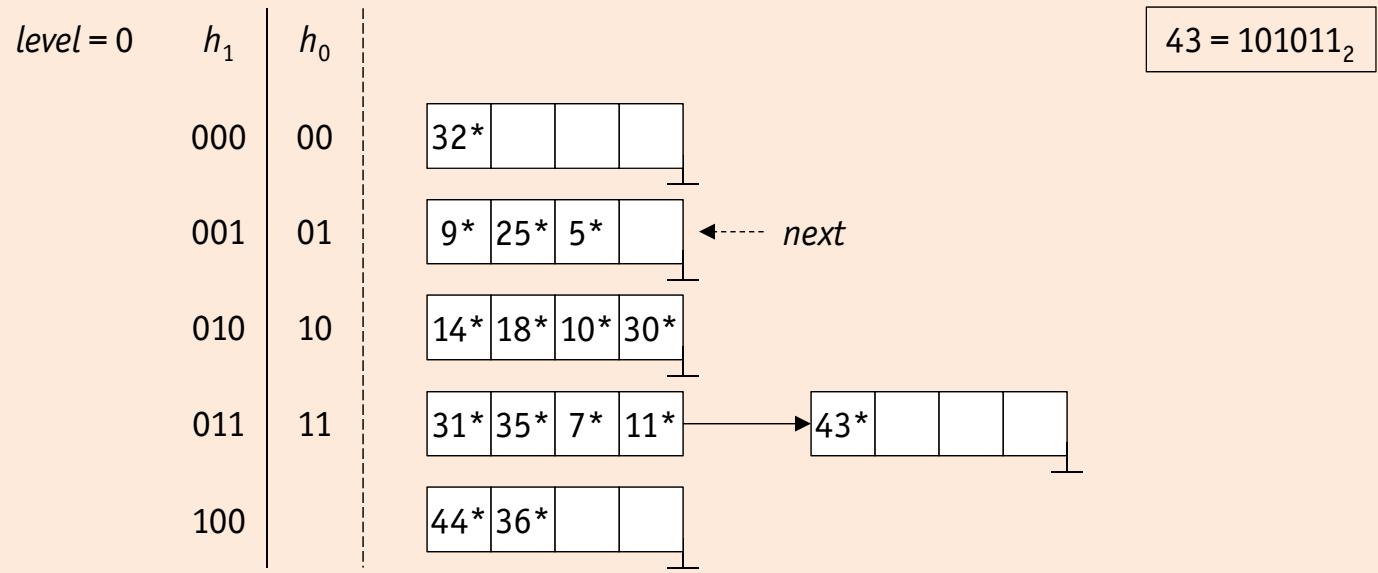
- Setup of linear hash table used in running example
 - bucket capacity of 4, initial hash table size $N = 4$, $level = 0$, $next = 0$
 - split criterion: allocation of a page in an overflow chain

Example: linear hash table ($h_{level}(k)^*$ shown)



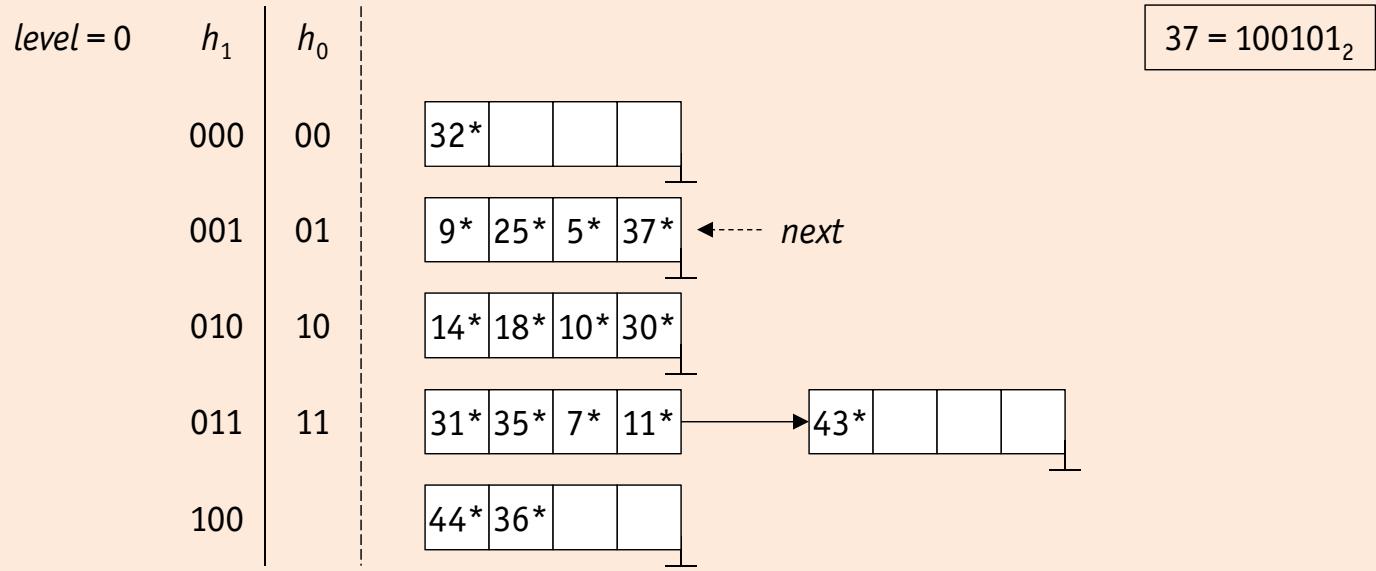
Linear Hashing

↷ Example: insert record with key k such that $h_0(k) = 43$



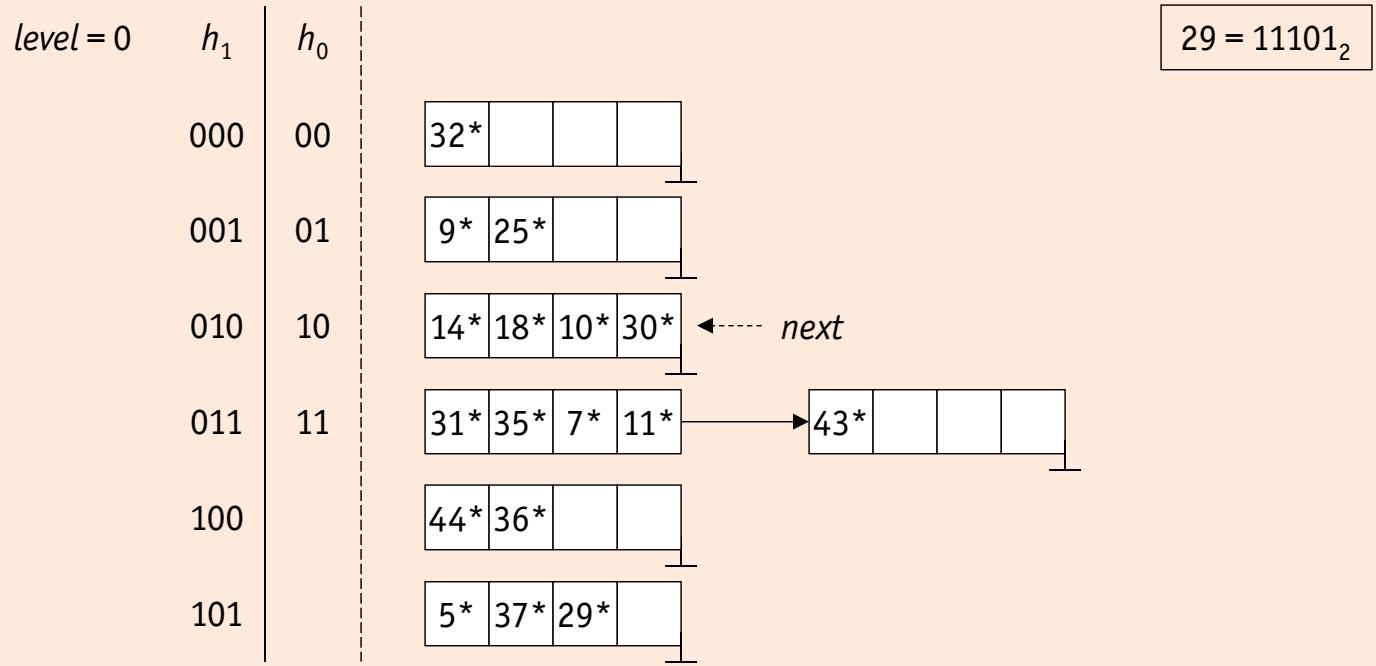
Linear Hashing

↷ Example: insert record with key k such that $h_0(k) = 37$



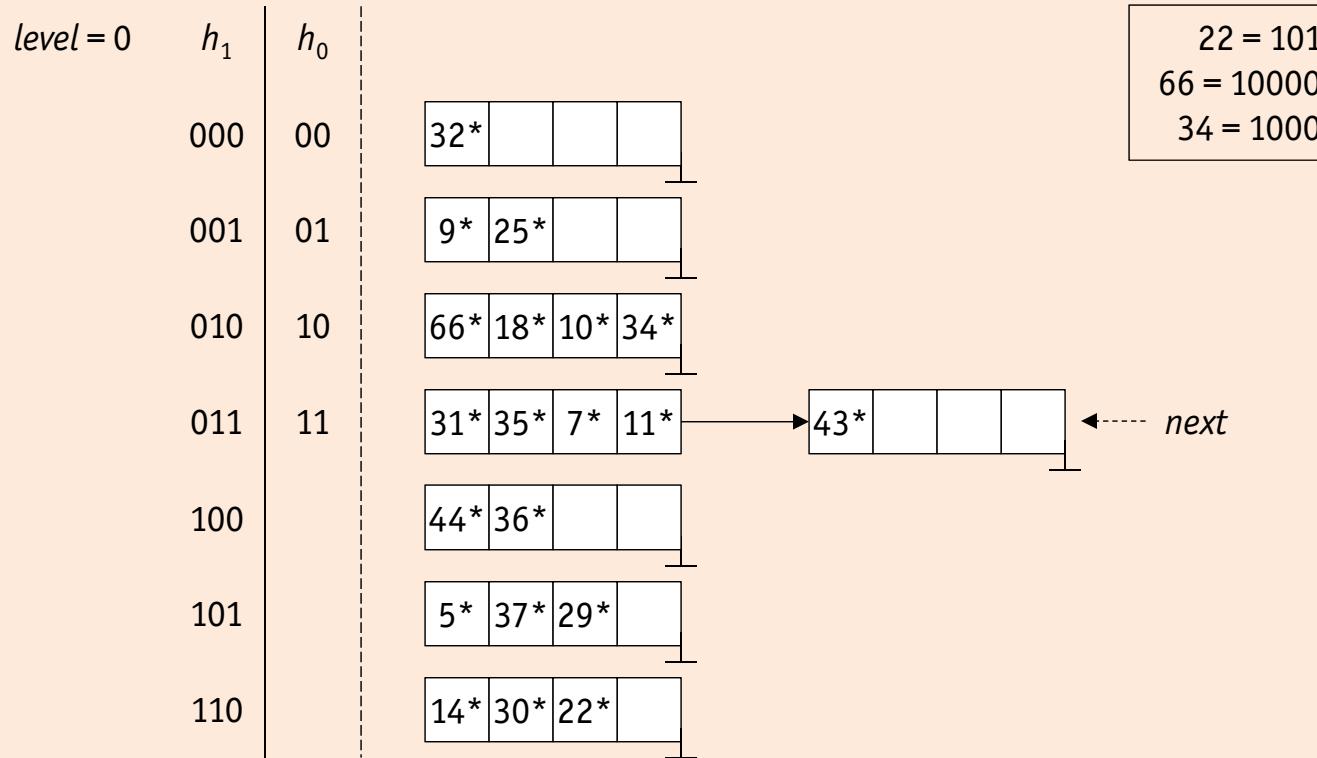
Linear Hashing

↷ Example: insert record with key k such that $h_0(k) = 29$



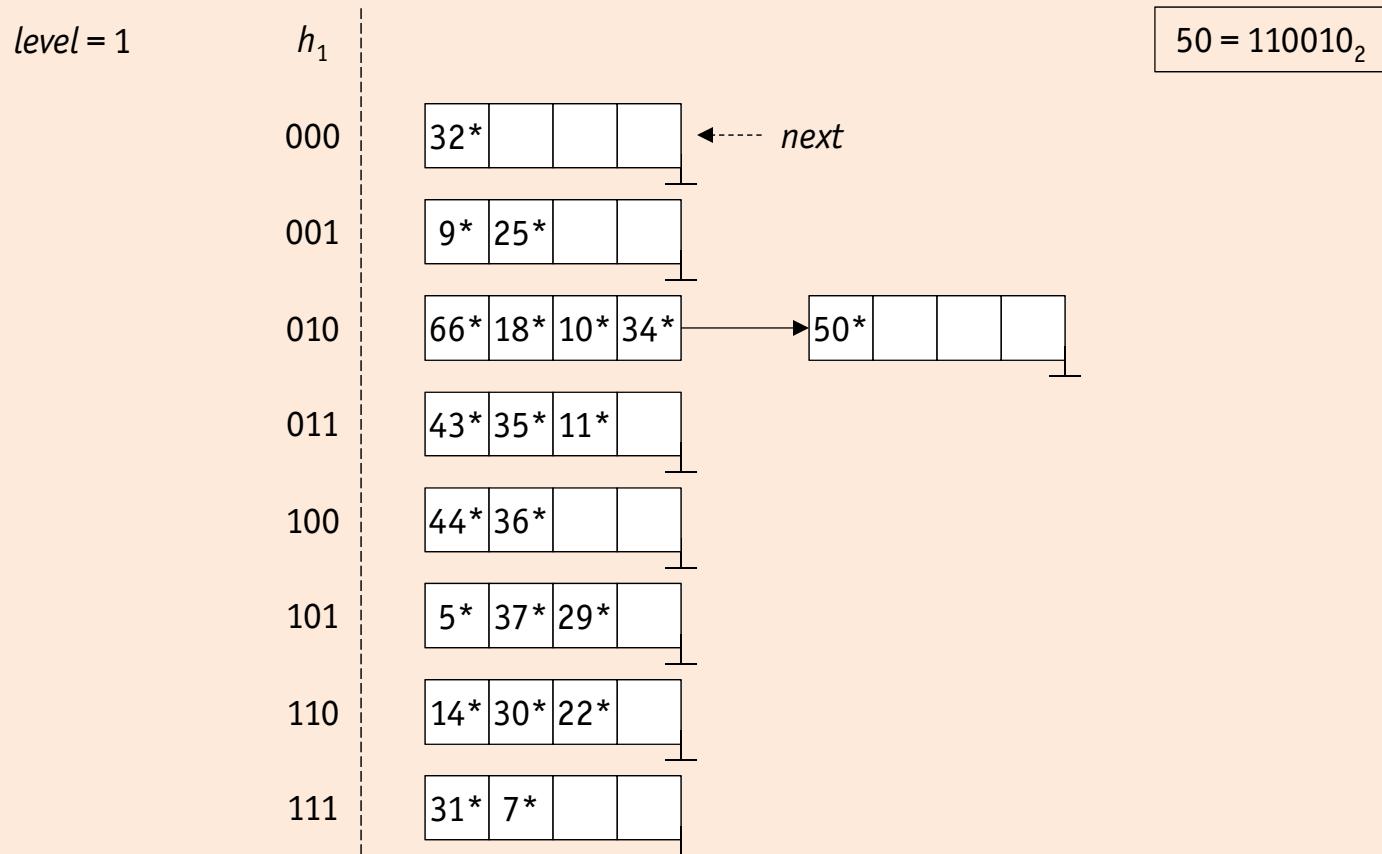
Linear Hashing

Example: insert record with key k such that $h_0(k) = 22, 66$, and 34



Linear Hashing

Example: insert record with key k such that $h_0(k) = 50$



Note: Rehashing a bucket means to **rehash its overflow chain** as well.

Linear Hashing Search

💻 Search in linear hashing

```
function hsearch (k)
    b ←  $h_{level}(k)$ ;
    if  $b < next$  then
        b ←  $h_{level+1}(k)$ ;
    return bucket[b];
end
```

(b has already been split, record for key may be
in bucket b or bucket $2^{level} \cdot N + b \rightarrow \text{rehash}$)

- Remarks
 - $bucket[0, \dots, 2^{level} \cdot N - 1]$ is an **in-memory array** containing hash table bucket (page) addresses
 - variables $level$ and $next$ are **global variables** of the linear hash table, N is constant

Linear Hashing Search

💻 Insert in linear hashing

```
function hinsert( $k^*$ )
     $b \leftarrow h_{level}(k)$ ;
    if  $b < next$  then
         $b \leftarrow h_{level+1}(k)$ ;
    place  $k^*$  in  $bucket[b]$ ;
    if overflow( $bucket[b]$ ) then      (last insertion triggered a split of bucket  $next$ )
        allocate a new bucket  $b'$ ;
         $bucket[2^{level} \cdot N + next] \leftarrow @b'$ ;          (grow hash table by one page)
        foreach entry  $k'^*$  in  $bucket[next]$  do          (rehash to redistribute entries)
            place entry  $k'^*$  in  $bucket[h_{level+1}(k')]$ ;
         $next \leftarrow next + 1$ ;
        if  $next > 2^{level} \cdot N - 1$  then          (every bucket of the hash table been split)
             $level \leftarrow level + 1$ ;
             $next \leftarrow 0$ ;                      (hash table size has doubled, start a new round)
    end
```

Note: Predicate **overflow**(\cdot) is a tunable parameter to control triggering of splits.

Linear Hashing Delete (Sketch)

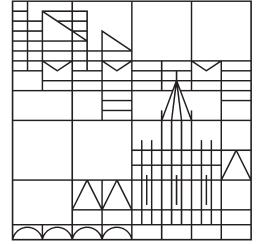
💻 Insert in linear hashing

```
function hdelete (k)
  :
  if empty (bucket[2level · N + next]) then      (deletion left last bucket empty)
    remove page pointed to by bucket[2level · N + next] from hash table ;
    next ← next - 1;
    if next < 0 then                                (round-robin scheme for deletion)
      level ← level - 1;
      next ← 2level · N + next;
  end
```

- Remarks
 - linear hashing deletion is essentially the “inverse” of `hinsert(·)`
 - possible to replace `empty(·)` with a suitable `underflow(·)` predicate

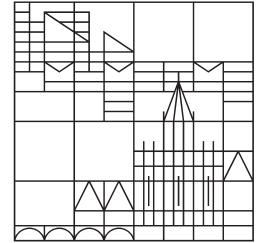
Extendible vs. Linear Hashing

- Directory vs. no directory
 - suppose linear hashing also used a directory with elements $[0, \dots, N - 1]$
 - since first split is at bucket 0, element N is added to the directory
 - imagine the directory is actually doubled at this point
 - since element 1 is the same as element $N + 1$, element 2 is the same as element $N + 2$, and so on, copying these elements can be avoided
 - at end of the round, all N buckets are split and directory doubled in size
- Directory vs. hash function family
 - choice of hashing functions is very similar to effect of directories
 - moving from h_i to h_{i+1} corresponds to doubling the directory: both operations double effective range into which key values are hashed
 - doubling range in a single step vs. doubling range gradually
- New idea behind linear hashing is that directory **can be avoided** by a clever choice of the bucket to split



Database System Architecture and Implementation

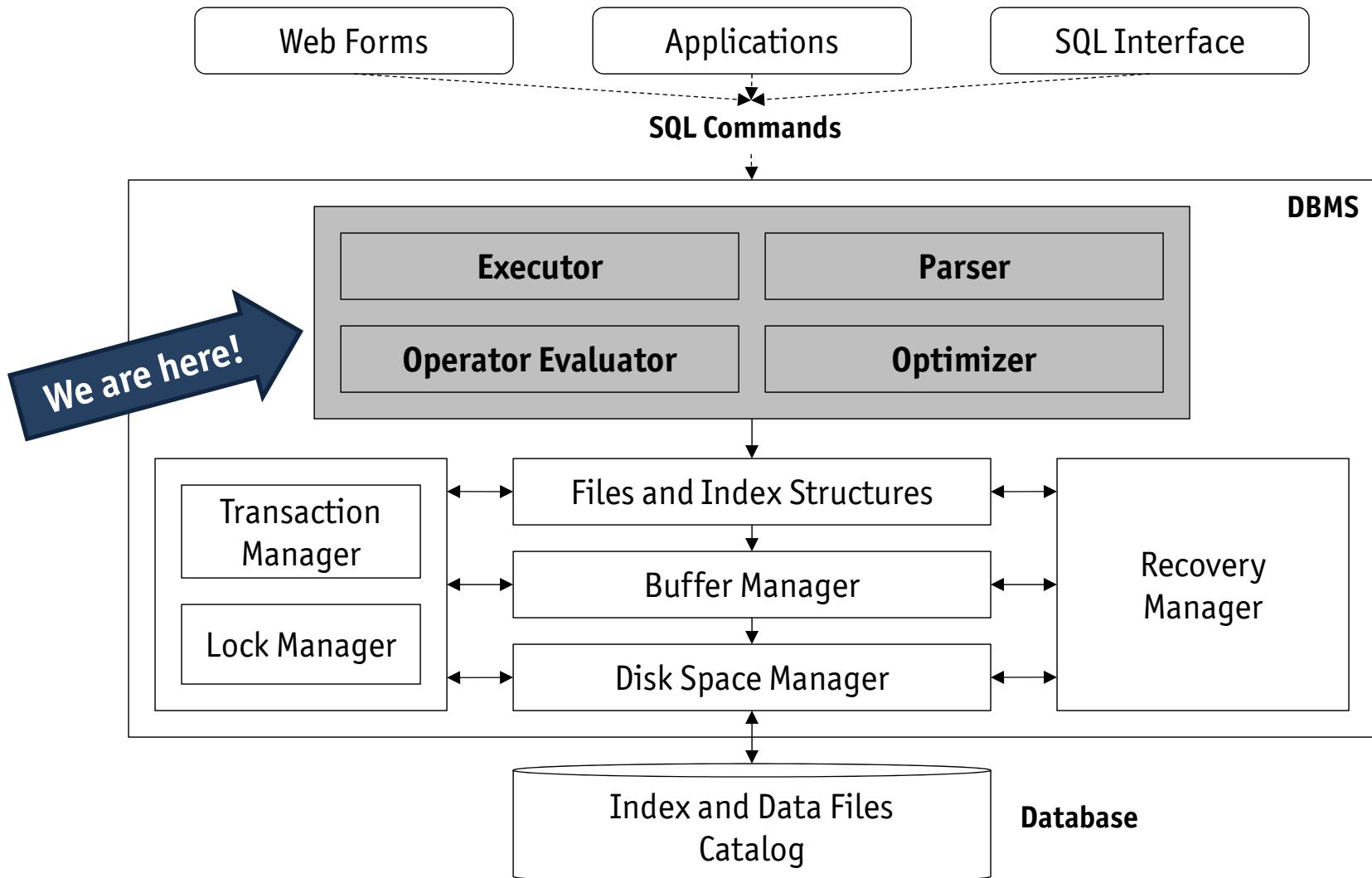
TO BE CONTINUED...



Database System Architecture and Implementation

Module 5
Query Evaluation Overview
December 2, 2013

Orientation



Module Overview

- System catalog
- Relational operator evaluation
 - algorithms for σ , π , and \bowtie
 - operator scheduling
- Query optimization
 - query evaluation plans
 - cost estimation of a plan

Running Example

💻 A simple schema

```
CREATE TABLE Sailors (
    sid      INTEGER,
    sname    STRING,
    rating   INTEGER,
    age      REAL,
    PRIMARY KEY (sid)
)

CREATE TABLE Reserves (
    sid      INTEGER,
    bid      INTEGER,
    day     DATE,
    rname   STRING,
    PRIMARY KEY (sid, bid, day),
    FOREIGN KEY sid REFERENCES
        Sailors(sid)
)
```

- Assumptions
 - relation **Sailors**: 50 bytes/tuple, 80 tuples/page, and 500 pages
 - relation **Reserves**: 40 bytes/tuple, 100 tuples/page, and 1000 pages
- ↳ 4 kB page size, 2 MB **Sailors** data, and 4 MB **Reserves** data

System Catalog

- A DBMS manages **two types of information**
- Data
 - DBMS uses several alternative file structures to store **tables** and **indexes**
 - conversely, files contains either **tuples of table** or **index entries**
 - collection of user tables and indexes represents that **data** of the database
- Metadata
 - DBMS also maintains information that **describes** every table and index
 - descriptive information itself stored in a collection of **special tables**
 - this **metadata** is called catalog tables, data dictionary, or system catalog

System Catalog

- System catalog stores system-wide information
 - **size of buffer pool**, the **page size**, etc.
 - information about **tables**, **views**, and **indexes**.

☞ Information stored in the system catalog

- **Table metadata**
 - *table name, file name* (or some identifier), *file structure* (e.g., heap file)
 - *attribute name* and *type* of each attribute of the table
 - *index name* of each index on the table
 - *integrity constraints* (e.g., primary and foreign key constraints) on the table
- **Index metadata**
 - *index name* and *structure* (e.g., B+ tree)
 - *search key* attributes
- **View metadata**
 - *view name* and *definition*

System Catalog

- Additionally, the system catalog stores
 - **statistics** about tables and indexes
 - statistics are updated **periodically, not every time** tables are modified

Statistics

- **Table statistics**
 - *cardinality*: number of tuples $NTuples(R)$ for each table R
 - *size*: number of pages $NPages(R)$ for each table R
- **Index statistics**
 - *cardinality*: number of distinct key values $NKeys(I)$ for each index I
 - *size*: number of pages $INPages(I)$ for each index (for a tree index I , $INPages(I)$ denotes the number of leaf pages)
 - *height*: number of non-leaf levels for each tree index I
 - *range*: minimum present key value $ILow(I)$ and the maximum present key value $IHigh(I)$ for each index I

System Catalog

What else?

Can you think of other information that would be useful to keep in the system catalog?

System Catalog

What else?

Can you think of other information that would be useful to keep in the system catalog?

- ↳ **user metadata:** user accounts in terms of *user name* and *password*
- ↳ **authorization metadata:** *access control lists* that define which user can create, update, read, or delete what tables, views, or indexes

System Catalog

- DBMS stores system catalog **itself** as a collection of tables
 - **existing techniques** for implementing and managing tables are reused
 - **same query language** can be used for catalog tables as for other tables

Metamodeling

The technique to model a data model in itself is known as **metamodeling** and the resulting data model is referred to as a **metamodel**. Since the choice of catalog tables and their schemas is not unique, real DBMS vary in terms of the actual metamodel.

As the designer of a DBMS, how would you design the schema of the system catalog?

System Catalog

- DBMS stores system catalog **itself** as a collection of tables
 - **existing techniques** for implementing and managing tables are reused
 - **same query language** can be used for catalog tables as for other tables

Metamodeling

The technique to model a data model in itself is known as **metamodeling** and the resulting data model is referred to as a **metamodel**. Since the choice of catalog tables and their schemas is not unique, real DBMS vary in terms of the actual metamodel.

As the designer of a DBMS, how would you design the schema of the system catalog?

- ↳ **Tables**(*name: string, file: string, num_tuples: integer, size: integer*)
- ↳ **Attributes**(*name: string, table: string, type: string, position: integer*)
- ↳ **Views**(*name: string, text: string*)
- ↳ **Indexes**(*name: string, file: string, type: string, num_keys: integer, size: integer*)

System Catalog

Example

Tables

name	file	#tuples	size
Tables	...	6	1
Attributes	...	23	1
Views	...	1	1
Indexes	...	0	1
Sailors	...	40,000	500
Reserves	...	100,000	1000

Views

name	text
Captains	SELECT * FROM Sailors WHERE...

Indexes

name	file	type	#keys	size
Boats	...	B+Tree	100	1

Attributes

name	table	type	pos
name	Tables	string	1
file	Tables	string	2
#tuples	Tables	integer	3
size	Tables	integer	4
name	Attributes	string	1
table	Attributes	string	2
type	Attributes	string	3
pos	Attributes	integer	4
:	:	:	:
sid	Sailors	integer	1
sname	Sailors	string	2
rating	Sailors	integer	3
age	Sailors	real	4
sid	Reserves	integer	1
bid	Reserves	integer	2
day	Reserves	date	3
rname	Reserves	string	4

System Catalog

The Real World

⇒ Oracle 10g defines a total of 8 system tables, including...

- **ALL_TABLES**: *owner, table name*, and about 50 other attributes
- **ALL_VIEWS**: *owner, view name, view text* and about 10 other attributes
- **ALL_TAB_COLUMNS**: *owner, table name, column name, data type, length, nullable, default, low, high, density, histogram* and about 25 other attributes

⇒ Microsoft SQL Server

- **sys.objects**: all user-defined, schema-scoped objects in a database in terms of *name, id, type* (e.g., U = user-defined table) and about 10 other attributes
- **OBJECT_ID (name, type)** : meta data function that returns *id* of an object

⇒ ANSI SQL-92

- **INFORMATION_SCHEMA**: provides information about *tables, views, columns* and *procedures*
- many system implement the standard by defining **views** over their proprietary system catalog tables

System Catalog

Microsoft SQL Server: schema definition in Transact-SQL

```
IF OBJECT_ID ('dbo.Employees', 'U') IS NOT NULL
    DROP TABLE dbo.Employees
IF OBJECT_ID ('dbo.Managers', 'V') IS NOT NULL
    DROP VIEW dbo.Managers
IF OBJECT_ID ('dbo.udf_DistributeBonus', 'P') IS NOT NULL
    DROP PROCEDURE dbo.udf_DistributeBonus
IF OBJECT_ID ('dbo.udf_CalculateBonus', 'FN') IS NOT NULL
    DROP FUNCTION dbo.udf_CalculateBonus
GO

CREATE TABLE dbo.Employees ( . . . )
CREATE VIEW dbo.Managers AS . .
CREATE PROCEDURE dbo.DistributeBonus AS . .
CREATE FUNCTION dbo.udf_CalculateBonus (@empId INT)
    RETURNS FLOAT BEGIN . . . END
```

Relational Operator Evaluation

- Several **alternative algorithms** can be used to implement each relational operators
 - for most operators no algorithm has universally superior performance
 - performance depends on sizes of involved tables, existing indexes and sort orders, size of buffer pool and buffer replacement policy
- Alternative algorithms for relational operators often use **common implementation techniques**
 - **indexing**: an index is used to only process tuples that satisfy a selection or join condition
 - **iteration**: process all tuples of an input table, one after the other, or scan all index data entries of an index that contains all required attributes (not using the search structure of the index)
 - **partitioning**: decompose an operation into a less expensive collection of operations by partitioning tuples on a sort key (e.g., sorting and hashing)

Access Paths

- An **access path** is a way of retrieving tuples from a table, e.g.,
 - file scan
 - index plus a matching selection condition
- Every relational operator accepts **one or more tables** as input
- Access paths chosen to retrieve tuples contribute significantly to the **cost of the operator**

Selection conditions

A selection is in **conjunctive normal form (CNF)**, if it is a conjunction (\wedge) of conditions of the form

attribute op value,

where **op** is one of the comparison operators $<$, \leq , $=$, \neq , \geq , or $>$, and each of these conditions is called a **conjunct**.

Access Paths

- An index **matches** a selection condition, if it can be used to retrieve just the tuples that satisfy the condition
 - a **hash index** matches a CNF selection, if there is a conjunct of the form $attribute = value$ for each attribute in the index's search key
 - a **tree index** matches a CNF selection, if there is a conjunct of the form $attribute \text{ op } value$ for each attribute in a prefix of the index's search key

Example: search key prefixes

$\langle a \rangle$, $\langle a, b \rangle$ are prefixes of key $\langle a, b, c \rangle$, whereas $\langle a, c \rangle$ and $\langle b, c \rangle$ are not.

- An index can match some **subset** of the conjuncts of a CNF selection, even though it does not match the entire selection
 - **primary conjuncts**: the conjuncts that the index matches

Access Paths

Exercise: hash index vs. B+ tree index

Assume a **hash index** H on the search key $\langle rname, bid, sid \rangle$ on table **Reserves**. Which of the following CNF conditions does H match?

- $rname = 'Joe' \wedge bid = 5 \wedge sid = 3$
- $rname = 'Joe' \wedge bid = 5$
- some condition on day

Assume a **B+ tree index** I on the search key $\langle rname, bid, sid \rangle$ on table **Reserves**. Which of the following CNF conditions does I match?

- $rname = 'Joe' \wedge bid = 5 \wedge sid = 3$
- $rname = 'Joe' \wedge bid = 5$
- $bid = 5 \wedge sid = 3$

Access Paths

P Example: primary conjuncts

Assume an index (hash or B+ tree) on the search key $\langle bid, sid \rangle$ on table **Reserves** and the selection condition $rname = 'Joe' \wedge bid = 5 \wedge sid = 3$

1. the index can be **scanned** to retrieve the tuples that satisfy the condition $bid = 5 \wedge sid = 3$, i.e., the primary conjuncts
 2. the **additional condition** on $rname$ must then be applied to each retrieved tuple and will eliminate some of the retrieved tuples from the result.
- ↳ the **number of pages** retrieved depends on the fraction of tuples that satisfy these conjuncts and whether the index is clustered or not

Access Paths

Example: two (partially) matching indexes

Assume an index (hash or B+ tree) on the search key $\langle bid, sid \rangle$, a B+ tree index on day , and the selection condition $day < 8/1/2013 \wedge bid = 5 \wedge sid = 3$

- both indexes match (part of) the selection condition and either one can be used to retrieve **Reserves** tuples
 - regardless of whichever is used, the conjuncts in the selection condition that are not matched by the index must be checked for each retrieved tuple
- ↳ if the B+ tree index on day is used, the condition $bid = 5 \wedge sid = 3$ must be checked
↳ if the hash index on $\langle bid, sid \rangle$ is used, the condition $day < 8/1/2013$ must be checked

Access Path Selectivity

- Recall that several alternative access paths may exist to retrieve all desired tuples in a table
 - **scan** of the data file
 - **index**, if table has an index that matches selection given in the query
 - **index scan**, if index contains all attributes required by the query

R Definition: selectivity of an access path

- the **selectivity** of an access path is the number of index and data pages it retrieves
 - the **most selective** access path is the one that retrieves the fewest pages
- ↳ choosing the most selective path minimizes cost of data retrieval, i.e., number of I/O operations

Access Path Selectivity

- Apart from data file organizations, the selection condition (with respect to the index involved) affects access path selectivity
 - selectivity depends on the **primary conjuncts** in the selection condition
 - each conjunct acts as a **filter** on the table

☞ Definition: reduction factor

- the fraction of tuples in the table that satisfy a given conjunct is called the **reduction factor**.
 - if there are several primary conjuncts, the fraction that satisfies all of them can be **approximated** by the product of their reduction factors.
- ☞ this approach effectively treats all conjuncts as **independent** filters
- ☞ while they **may not be independent**, this approximation is widely used in practice

Access Path Selectivity

Example

Assume a **hash index** H on table **Reserves** with search key $\langle rname, bid, sid \rangle$ and the selection condition $rname = 'Joe' \wedge bid = 5 \wedge sid = 3$

- H can be used to retrieve tuples that satisfy **all** conjuncts
 - the catalog contains the **number of distinct keys** $NKeys(H)$ in the hash index
 - the catalog also contains total number of pages $NPages$ in the **Reserves** table
- ⇒ the **reduction factor**, i.e., the fraction of tuples satisfying the primary conjuncts is

$$NPages(Reserves) \cdot \frac{1}{NKeys(H)}$$

Access Path Selectivity

Exercise: partially matching indexes

How would you estimate the fraction of tuples that satisfy a selection condition, if no index matches the entire condition?

Access Path Selectivity

Exercise: partially matching indexes

How would you estimate the fraction of tuples that satisfy a selection condition, if no index matches the entire condition?

- ↳ use catalog information on partially matching indexes and **approximate** fraction of satisfying tuples by multiplying these reduction factors
- ↳ assume a **default value** for the reduction factor, typically 1/10
- ↳ if available use **statistics** (e.g., histograms) about distribution of attribute values

Access Path Selectivity

- Reduction factor estimation for range selection conditions
 - assume that range attribute values are **uniformly** distributed
 - catalog information $ILow$ and $IHeigh$ can be used
- If there is a tree index T , the estimated reduction factor is
 - $\frac{IHeigh(T) - value}{IHeigh(T) - ILow(T)}$ for selection condition $attr < value$.

Relational Operator Evaluation

- Focus on σ , π , and \bowtie operator as used in so-called **SPJ queries**
 - **this week:** brief overview of evaluation algorithms for these operators
 - **next two weeks:** in-depth look at sorting and operator evaluation
- Cost analysis
 - as before, **only I/O costs** are considered
 - I/O costs are measured in terms of **number of page I/O operations**
 - **examples**, rather than rigorous cost formulas

Selection

- Selection given as $\sigma_{R.attr \text{ op } value}(R)$
 - if there is **no index** on $R.attr$, R has to be scanned
 - if **one or more indexes** match the selection, an index can be used (possibly followed by application of remaining selection conditions)

Example: selection `rname < 'C%`' on **Reserves** table

- assuming that names are **uniformly distributed** with respect to the initial letter, roughly 10% of **Reserves** tuples are estimated to be in the result (for simplicity)
- since **Reserves** has 100,000 tuples, this is a total of 10,000 tuples or 100 pages
- if there is a **clustered** B+ tree index on **Reserves.rname**, the qualifying tuples can be retrieved with 100 I/O operations (plus a few I/O operations to traverse the tree)
- if the index is **unclustered**, retrieving the tuples could require 10,000 I/O operations

A rule of thumb

If over 5% of the tuples of a table are to be retrieved, it is **likely to be cheaper** to simply scan the entire table (instead of using an unclustered index)

Projection

- Projection given as $\pi_{\{R.attr1, R.attr2, \dots\}}(R)$
 - dropping attributes from the input is the easy part
 - the expensive part is **duplicate elimination** from the result
- **Without** duplicate elimination (no **DISTINCT** in **SELECT**)
 - **scan of the table** to retrieve the projected subset of attributes
 - **scan of an index** whose key contains all necessary attributes (clustered vs. unclustered does not matter as only key values are retrieved)
- **With** duplicate elimination (by partitioning)
 1. as above, retrieve subset of projected attributes using a **scan**
 2. using the projected attributes as sort key, **sort** all retrieved tuples
 3. scan tuples and **discard** adjacent duplicates

Projection

- Sorting **disk-resident** data sets is next week's topic
 - typically requires two or three passes of reading and writing entire table
 - projection can be **optimized** by combining it with sorting

Optimized projection (with duplicate elimination)

- **first pass of sorting** scans entire table, but only writes out the subsets of projected attributes
- there might be an **intermediate pass** in which all subsets of projected attributes are read from and written to disk
- **final pass of sorting** scans all tuples, but only one copy of each subset of projected attributes is written out

Projection

- Availability of appropriate indexes can lead to **less expensive** plans than sorting for duplicate elimination
 - if an index exists whose **search key contains all attributes** retained by the projection, we can sort the index entries, rather than the data records
 - if all retained attributes appear in a **prefix of the search key** of a tree index, duplicates are easily detected since they are adjacent

↳ These are example of **index-only** evaluation strategies

Join

- Join given as $R \bowtie_{R.attr = S.attr} S$
 - as joins are both **common** and **expensive**, they have been widely studied
 - DBMS typically supports **several algorithms** to compute joins

Index nested loops join

Consider a join of R and S with the condition $R.attr = S.attr$. If one of the tables has an index on column $attr$, an **index nested loops join** can be used.

Index nested loops join (sketch)

```
function join(R, I, attr)
  for each tuple  $t_R$  in  $R$  do
    do
       $t_S$  = probe  $I$  with key  $t_R.sid$  ;
      if  $t_S \neq \text{null}$  then emit  $t_R \bowtie t_S$  ;
    until  $t_S == \text{null}$  ;
  end;
```

Join

- Assume that I is a hash-based index and it takes about **1.2 I/O operations** on average to retrieve the appropriate page
 - suppose index uses variant **②**, i.e., contains $\langle k, rid \rangle$ entries
 - since sid is a key for **Sailors**, there is **at most** one matching tuple
 - since sid is a foreign key in **Reserves**, there is exactly one match

☛ Cost analysis of index nested loops join plan

- cost of scanning **Reserves** is 1000 (pages)
 - there are $100 \cdot 1000$ tuples in **Reserves**
 - for each of these tuples, retrieving the index page containing the corresponding rid of the matching **Sailors** tuple costs 1.2 I/O operations on average
 - additionally, the **Sailors** page containing the qualifying tuple must be retrieved
- ☛ **total cost** is $1000 + (1.2 + 1) \cdot 100,000 = \underline{221,000}$ I/O operations

Join

- If there is no index that matches the join condition on either table, the index nested loop join cannot be used

☛ Sort-merge join

If no index matches the join condition, a **sort-merge join** can be used

1. **sort** both tables on the join column
2. **scan** sorted tables to find matches

☛ Index nested loop join (very high-level sketch)

```
function join(R, S, attr)
    sort R on attr;  $\uparrow t_R$  = first tuple in R;
    sort S on attr;  $\uparrow t_S$  = first tuple in S;
    do
        if  $t_R.attr \neq t_S.attr$  then advance  $t_R$  else emit  $t_R \bowtie t_S$  and advance  $t_S$ ;
        until  $t_R$  and  $t_S$  reach the end of the table R and S, respectively;
    end;
```

Join

- Assume that both tables can each be sorted in **two** (!) passes

☛ Cost analysis of sort-merge join plan

- sort of **Reserves** reads and writes all of its pages two times: $2 \cdot 2 \cdot 1000 = 4000$
 - sort of **Sailors** reads and writes all of its pages two times: $2 \cdot 2 \cdot 500 = 2000$
 - second phase of sort merge join requires an additional scan of both tables, i.e., $1000 + 500 = 1500$
- ⇒ **total cost** is $4000 + 2000 + 1500 = \underline{7500}$ I/O operations

- Remarks
 - cost of sort-merge join (which **does not require** a pre-existing index) is lower than cost of index nested loops join
 - additionally, the result of the sort-merge join is sorted on join column
 - other join algorithms that do not rely on existing indexes and are often cheaper than index nested loops joins are also known

Join

Exercise: why consider index nested loops joins at all?

What nice property does an index nested loops join have that a sort-merge join does not have with respect to cost?

Join

Exercise: why consider index nested loops joins at all?

What nice property does an index nested loops join have that a sort-merge join does not have with respect to cost?

- ⇒ the cost of an index nested loops join is **incremental** in the number of R tuples (outer loop)
- ⇒ if some additional selection in the query restricts the R tuples that need to be considered to a small subset, computing the join of R and S **can be avoided entirely**

Example: Given a query computing the $\text{Reserves} \bowtie_{\text{Reserves.sid} = \text{Sailors.sid}} \text{Sailors}$ join, assume that this query also has a selection $\sigma_{bid=101}(\text{Reserves})$ and that there are very few such **Reserves** tuples

- **indexed nested loops join:** for each **Reserves** tuple, **Sailors** is probed
- **sort merge join:** the entire **Sailors** table has to be scanned at least once, which is likely to be more expensive than the entire cost of the index nested loops join

Other Operators

- In addition to the basic relational operators (σ , π , and \bowtie), SQL queries can also contain
 - **group by**
 - **aggregation**
 - **set operations**: union, difference, and intersection
- Set operations
 - as in projection, expensive aspect is **duplicate elimination**
 - **partitioning approach** for projection can be adapted for these operations
- Group by
 - typically implemented through **sorting**
 - sorting step can be avoided, if a **tree index** with the grouping attributes as search key exists
- Aggregate operators are carried out using **temporary counters** in main memory as tuples are retrieved

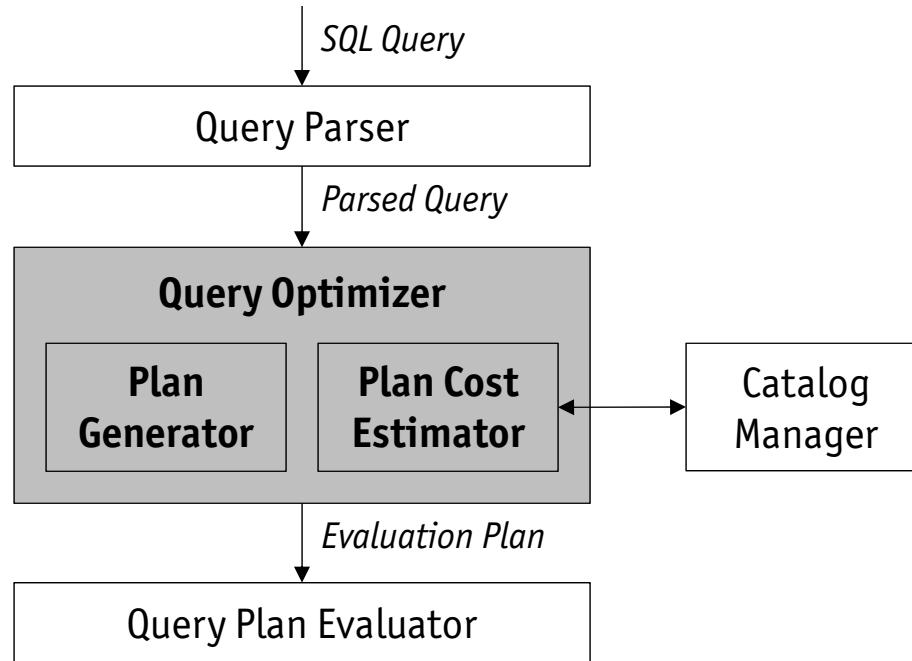
Query Optimization

- “*With great power comes great responsibility.*” (Ben Parker)
 - SQL gives users **great flexibility** how to (declaratively) express a query
 - DBMS can chose from **many different strategies** to evaluate a query
- Quality of the **query optimizer** greatly influences performance
 - difference in cost between the best and worst evaluation strategy may be **several orders of magnitudes**
 - query optimizer cannot be expected to always find the best strategy, but it should consistently find a strategy that is **quite good**

Commercial optimizers

Current relational DBMS optimizers are **very complex** pieces of software with many **closely guarded details**, and they typically represent **40 to 50 man-years** of development effort.

Query Optimization



- Query optimizer identifies an efficient execution plan
 - **plan generator** enumerates alternative plans
 - **plan cost estimator** assigns a cost of each alternative plan
 - plan with the least estimated cost is chosen

Query Optimization

- To enumerate alternative plans, a query optimizer explores the **search space** of possible plans
 - queries are essentially treated as **σ - π - \bowtie algebra expressions**
 - **remaining operations** are applied to the result of the σ - π - \bowtie expression
- Optimizing a σ - π - \bowtie algebra expression involves three steps
 1. **enumerating** alternative plans for evaluating the expression
 2. **estimating** the cost of each enumerated plan
 3. **choosing** the plan with the lowest estimated cost

Restricting the search space

Typically, a query optimizer considers a **subset of all possible plans** because the number of possible plans is very large.

Query Evaluation Plans

- A **query evaluation plan** (or simply plan) consists of an relational algebra tree extended with annotations at each node
 - access paths to use for each table
 - algorithm to use for each relational operator

Example: a simple SPJ query

```
SELECT S.sname
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid AND R.bid = 100 AND S.rating > 5
```

Exercise: translate from SQL to relational algebra

Express the SQL given above in relational algebra using σ , π , and \bowtie operators.

Query Evaluation Plans

- A **query evaluation plan** (or simply plan) consists of an relational algebra tree extended with annotations at each node
 - access paths to use for each table
 - algorithm to use for each relational operator

⌨ Example: a simple SPJ query

```
SELECT S.sname
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid AND R.bid = 100 AND S.rating > 5
```

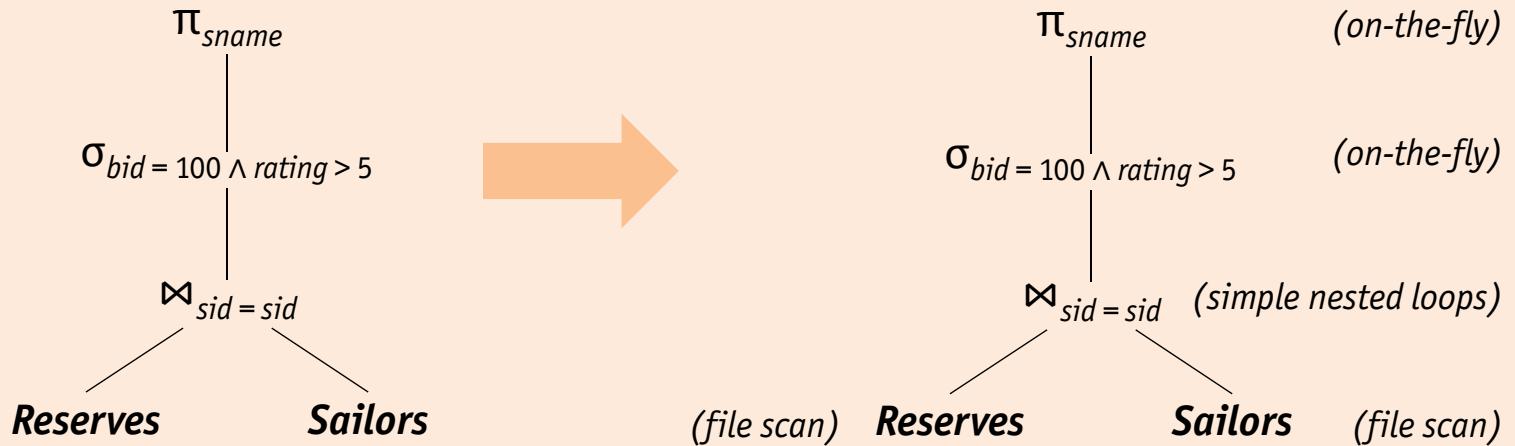
📝 Exercise: translate from SQL to relational algebra

Express the SQL given above in relational algebra using σ , π , and \bowtie operators.

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$

Query Evaluation Plans

Example: from relational algebra tree to query evaluation plan



- Remarks
 - join operator uses the page-oriented **simple nested loops join** algorithm
 - by convention the **outer table is the left child** of the join operator
 - selections and projections are applied **on-the-fly** to each tuple in the result of the join as it is produced

Multi-Operator Queries

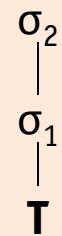
- If a query is composed of **several operators**, the query evaluator needs to decide how these operators
 - **communicate** with each other to pass results of one operator to the next
 - are **scheduled** and **interleaved** (i.e., inter-operator parallelism)
- Communication
 - **materialized**: operator writes its output to a temporary table, next operator reads its input from this temporary table
 - **pipelined**: results are directly “streamed” (typically, one page at a time) from one operator to the next as they are produced
- Scheduling
 - **bracket model**: explicit scheduling
 - **iterator model**: implicit scheduling

Materialized vs. Pipelined Evaluation

- Pipelining has **lower overhead costs** than materialization
 - avoids cost of writing out intermediate results and reading them back in
 - chosen whenever algorithm for operator evaluation permits it
- There are **many opportunities** for pipelining in typical query plans, even simple plans that only involve selections

Example: selection-only query

Consider a selection-only query in which only part of the selection condition matches an index: we can think of such a query as containing **two** instances of the selection operator (σ)



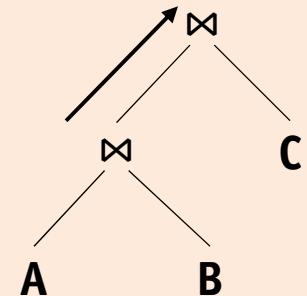
- the first selection (σ_1) contains the **primary** (or matching) **part** of the original selection condition
 - the second selection (σ_2) contains the **rest** of the selection condition
- ↳ **materialized**: apply σ_1 and write results to a temporary table, apply σ_2 to this table
- ↳ **pipelined**: apply σ_2 to each tuple in the result of σ_1 as it is produced

Materialized vs. Pipelined Evaluation

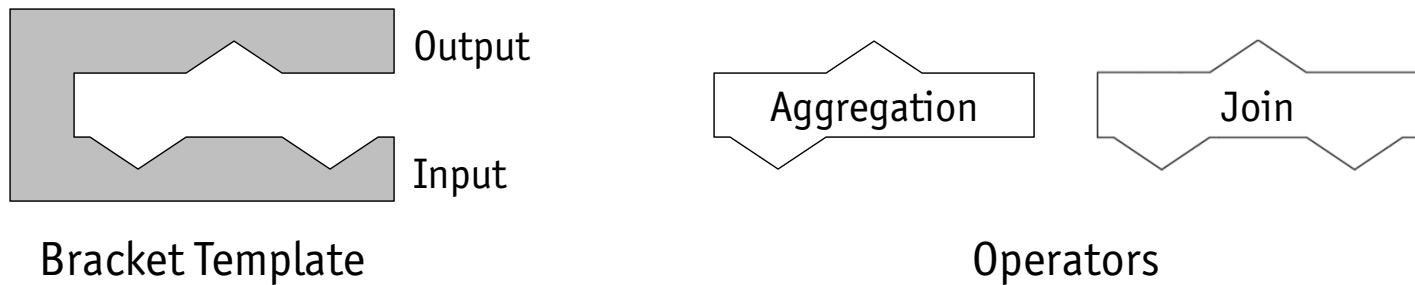
Example: joins

Consider a join of the form $(A \bowtie B) \bowtie C$: both joins can be evaluated in pipelined fashion using some form of **nested loops join**

1. conceptually, the evaluation is initiated from the root, i.e., node $A \bowtie B$ produces tuples as and when they are requested by its parent
 2. when the root node gets a page of tuples from its left child (outer table), all matching inner tuples are retrieved (using either an index or a scan), and joined with matching outer tuples
 3. the current page of outer tuples is then discarded, the next page is requested from the left child, and the process is repeated
- ↳ pipelined evaluation is thus a **control strategy** governing the rate at which different joins in the plan proceed
- ↳ results are produced, consumed, and discarded **one page at a time**, rather than writing intermediate result of joins to a temporary table



Bracket Model



- Bracket Template
 - operating system and communication support
 - **central scheduler** can fit a single operator into the bracket on demand
- Operators
 - each operator **must** run its own thread, assumes all control within its thread, sends and receives data via networking procedures
 - essentially relies on networking procedures for pacing and flow control
 - each operator defines its own phases, synchronization points, and communication needs that must be known to the bracket implementation

Iterator Model

- To simplify the code to coordinate plan execution, all operators implement a **uniform** iterator interface
 - hides **internal implementation details** of each operators
 - assumes **pipelined evaluation** of the query plan (see next slide)

☞ Iterator interface

open()

- initializes iterator by allocating input and output buffers
- used to pass parameters that modify operator behavior, e.g., a selection condition

next()

- calls **next** on each input operator
- executes operator-specific code to process input tuples
- places results in output buffer
- updates iterator state to keep track of how much input has been consumed

close()

- deallocates state information

Iterator Model

Exercise: pipelined vs. materialized evaluation in the iterator model

The iterator interface supports pipelining naturally, but what happens if the evaluation of an operator does not permit pipelining, e.g., blocking operators such as sort, aggregation, and certain joins?

- Iterators and indexes
 - iterator interface is also used to encapsulate access methods (indexes)
 - access methods are viewed as operators that produce a stream of tuples
 - **open()** is used to pass selection conditions that match the access path

Iterator Model

Exercise: pipelined vs. materialized evaluation in the iterator model

The iterator interface supports pipelining naturally, but what happens if the evaluation of an operator does not permit pipelining, e.g., blocking operators such as sort, aggregation, and certain joins?

The decision whether to materialize or pipeline input tuples is **encapsulated** (hidden by the interface) in the operator-specific code that processes input tuples

- if the algorithm used to implement an operator can process input tuples **completely** when they arrive, they are not materialized and the evaluation is pipelined
- if the algorithm used to implement an operator has to examine the input tuples **several times**, they are materialized

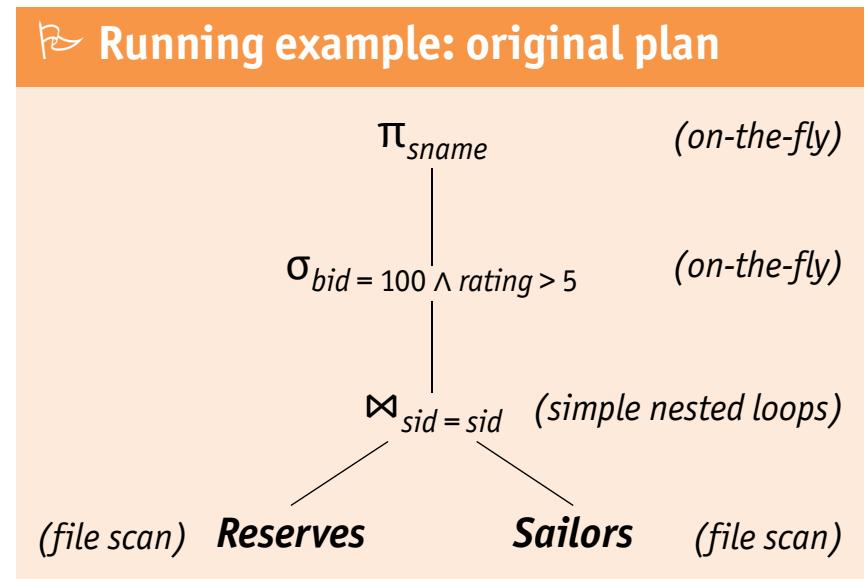
- Iterators and indexes
 - iterator interface is also used to encapsulate access methods (indexes)
 - access methods are viewed as operators that produce a stream of tuples
 - **open()** is used to pass selection conditions that match the access path

Iterator Model

Iterator	open()	next()	close()	State
print	open input	call next() on input, format item on screen	close input	
scan	open file	read next item	close file	open file descriptor
select	open input	call next() on input, until an item qualified	close input	
hash join without overflow resolution	allocate hash directory, open left <i>build</i> input, build hash table calling next() on build input, close build input, open right <i>probe</i> input	call next() in <i>probe</i> input until a match is found	close <i>probe</i> input, deallocate hash directory	hash directory
merge join without duplicates	open both inputs	get next() item from input with smaller key until a match is found	close both inputs	
sort	open input, build all initial run files calling next() on input, close input, merge run files until only one step is left	determine next output item, read new item from the correct run file	destroy remaining run files	merge heap, open file descriptors for run files

Motivating Alternative Plans

- Consider the cost of evaluating the plan shown on the right
 - ignore** cost of writing out the result can as it is the same for all plans
 - cost of (simple nested loops) **join** is $1000 + 1000 \cdot 500$ page I/O operations
 - selections** and **projection** are done on the fly and do not incur additional I/O operations



- Total cost** of this plan is 501,000 page I/O operations
- Now, consider several alternative plans for evaluation this query
 - each plan improves on the original plan in a different way
 - cost benefit of each of these optimization is examined in detail

Pushing Selections

- Recall that joins are relatively expensive operations
 - sizes of the two inputs of the join determine its cost
 - a good **heuristic** is to reduce these inputs as much as possible

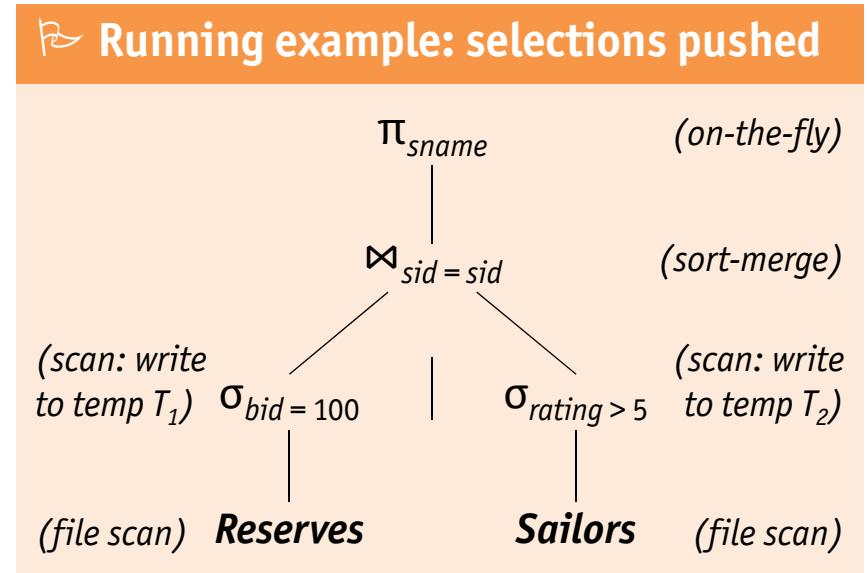
Pushing selections

An example of this heuristic is to apply selections (σ) early: if a selection appears after a join, it is worth examining whether the selection can be **pushed** ahead of the join.

- Running example
 - selection $bid = 100$ only involves attributes of the ***Reserves*** table
 - selection $rating > 5$ only involves attributes of the ***Sailors*** table
 - both selections can be pushed and applied **before** the join

Pushing Selections

- Assumptions
 - selections performed using file scan and result written to temporary tables
 - sort-merge join used to join the temporary tables
 - five buffer pages are available to evaluate this plan
- Cost of selection $\sigma_{bid=100}$
 - cost of scanning **Reserves** (1000 pages)
 - cost of writing result to table T_1 (this cost cannot be ignored)
- Additional information is needed to estimate size of T_1
 - **1 tuple**, if we assume a maximum of one reservation per boat
 - **10 pages**, if we know that there are 100 boats and we assume that reservations are uniformly distributed over boats (let's assume this!)



Pushing Selections

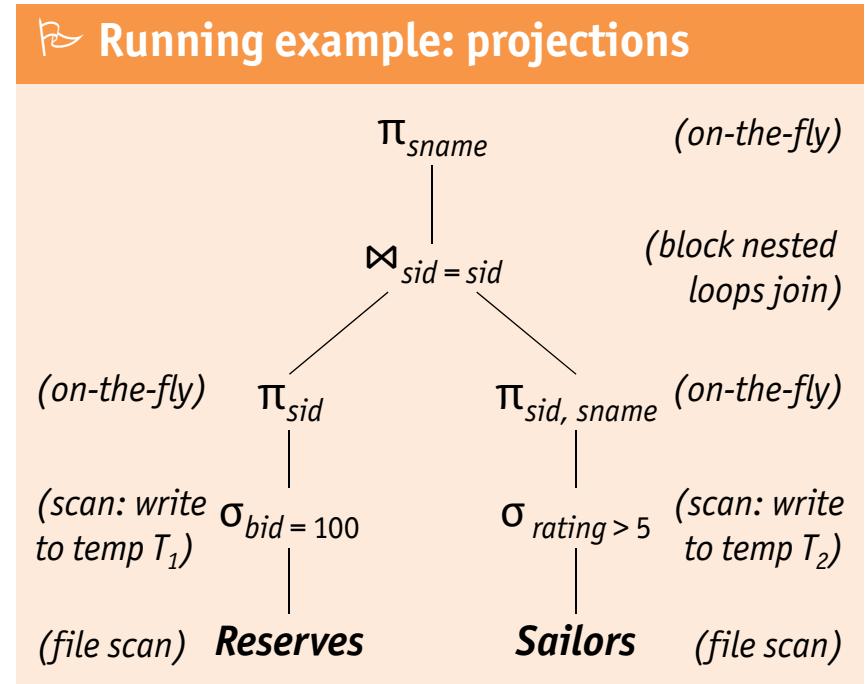
- Cost of selection $\sigma_{rating > 5}$
 - cost of scanning ***Sailors*** (500 pages)
 - cost of writing result to table T_2 (again, this cost cannot be ignored)
- Estimating the size of T_2
 - **250 pages**, if we assume ratings to be uniformly distributed over the range 1 to 10
- Cost of computing $\bowtie_{sid = sid}$ using a sort-merge join
 - use straightforward implementation: sort tables completely, then merge
 - using five available buffer pages, T_1 (10 pages) can be sorted in 2 passes (each pass reads and writes 10 pages), i.e., $2 \cdot 2 \cdot 10 = \mathbf{40 \text{ page I/Os}}$
 - 4 passes needed to sort T_2 (250 pages), i.e., $2 \cdot 4 \cdot 250 = \mathbf{2000 \text{ page I/Os}}$
 - to merge T_1 and T_2 , both tables need to be scanned, i.e., the cost of this step is $10 + 250 = \mathbf{260 \text{ page I/Os}}$
- Final projection π_{sname} is done on-the-fly and does not incur cost

Pushing Selections

- **Total cost** of the plan is therefore 4060 page I/O operations
 - cost of **selections** is $1000 + 10 + 500 + 250 = 1760$
 - cost of the **sort-merge join** is $40 + 2000 + 260 = 2300$
- Assume that a **block nested loops join** is used instead
 - using T_1 as the outer table, for every three-page block of T_1 , all of T_2 is scanned, i.e., T_2 is scanned four times (recall: five buffer pages available)
 - cost of **block nested loops join** is $10 + (4 \cdot 250) = 1010$ page I/Os
 - **total cost** of the plan is now $1760 + 1010 = 2770$ page I/O operations
- Just like selections, pushing projections can also reduce the size of tables in a join
 - only sid of T_1 and $\langle sid, rname \rangle$ of T_2 are required to evaluate the query
 - as **Reserves** and **Sailors** are scanned, unnecessary attributes can be eliminated

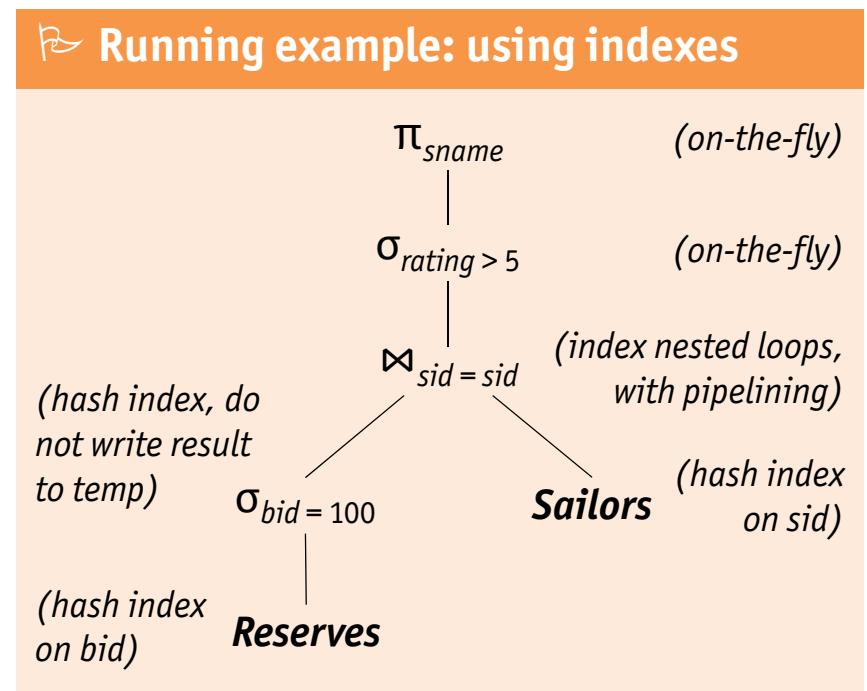
Pushing and Introducing Projections

- **Pushing and introducing projections** is another heuristic to reduce the sizes of table in a join
 - only sid of T_1 and $\langle sid, sname \rangle$ of T_2 are required to evaluate the query
 - as T_1 and T_2 are written, unnecessary attributes can be eliminated
- Cost estimation
 - on-the-fly projections reduce the sizes of tables T_1 and T_2
 - reduction of T_1 substantial, since only integer attributes retained
 - T_1 now fits into three buffer pages
 - block nested loop join only needs to scan T_2 once (250 page I/Os)
- **Total cost** is $1760 + 240 = \underline{2010}$ page I/O operations



Using Indexes

- If indexes are available on *Reserves* and *Sailors*, even better query evaluation plans may be available
- Assumptions
 - clustered static hash index on *Reserves.bid*
 - hash index on *Sailors.sid*
- Cost of selection $\sigma_{bid} = 100$
 - **10 page I/Os**, if we make the same assumption as before
 - $100,000/100 = 1000$ tuples are estimated to be selected
 - since index is clustered, the 1000 tuples are in 10 consecutive pages



Using Indexes

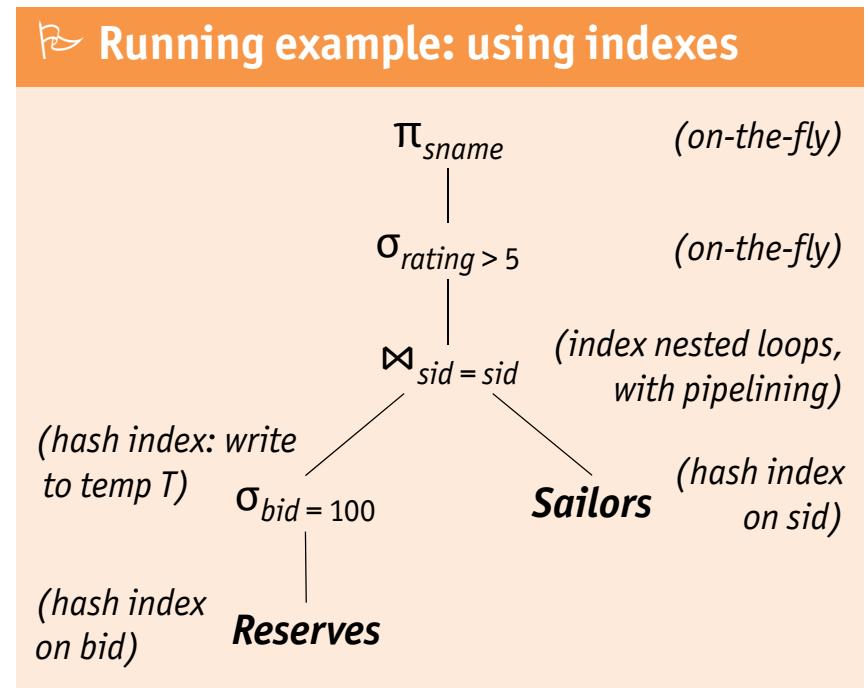
- Computing $\bowtie_{sid = sid}$ using a index nested loop join
 - for each selected **Reserves** tuple, matching **Sailors** tuples are retrieved using the hash index on *sid*
 - selected **Reserves** tuples are not materialized and the join is pipelined
 - selection $\sigma_{rating > 5}$ and projection π_{sname} are performed on-the-fly
- Important points to note
 - since plan is fully pipelined, introduction projections is not needed
 - since *sid* is a key of **Sailors**, at most one **Sailors** tuples matches a given **Reserves** tuple, therefore the cost of retrieving the matching tuple does not depend on whether the index on **Sailors** is clustered or not
 - selection $\sigma_{rating > 5}$ has not been pushed because it would require a scan of **Sailors** (assuming there is no index on *rating*) and because there is no index available on *sid* **after** the selection is applied

Using Indexes

- Cost of computing $\bowtie_{sid = sid}$ using an index nested loops join
 - for each of the 1000 *Reserves* tuples, the *Sailors* index on *sid* is probed
 - cost of probing *Sailors* index depends on whether the hash directory fits in memory and on the presence of overflow pages
 - assuming that the index uses variant ① for entries, **1.2 page I/Os** are a good estimate (if variant ② or ③ is used the cost would be 2.2 page I/Os)
 - cost of **index nested loops join** is $1.2 \cdot 1000 = 1200$ page I/O operations
- **Total cost** is $10 + 1200 = \underline{1210}$ page I/O operations

Using Indexes

- If *Sailors* index on sid is clustered, plan can be further refined
- Assumption
 - materialize result of $\sigma_{bid=100}$ to temporary table T and sort T
- Cost of selection $\sigma_{bid=100}$
 - T has again 10 pages, selecting the tuples costs **10 page I/Os**
 - writing out the result costs another **10 page I/Os**
 - sorting with five buffer pages costs $2 \cdot 2 \cdot 10 = \mathbf{40 \text{ page I/Os}}$
- Pipelining is **not always best**
 - Selected *Reserve* tuples can now be retrieved in order by sid
 - If a sailor has reserved the same boat many times, the matching *Sailors* tuple will be found in the buffer pool (on all but the first request)



Using Indexes

- Combining pushing of selections with index use is very powerful
 - join operation may become trivial, if selected tuples from outer table join with a single inner tuple
 - performance gains with respect to naïve plan may be dramatic

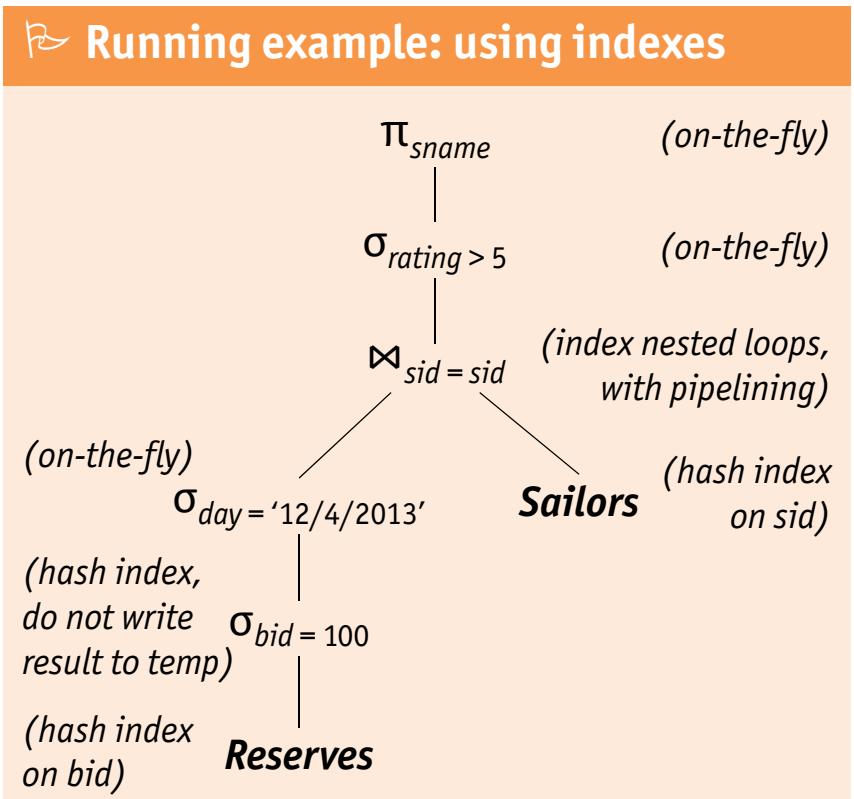
Example: extended SPJ query

```
SELECT S.sname  
FROM   Reserves R, Sailors S  
WHERE  R.sid = S.sid AND R.bid = 100  
       AND S.rating > 5 AND R.day = '12/04/2013'
```

- Remarks
 - extended query introduces additional selection $\sigma_{day='12/4/2013'}$
 - assume that *bid* and *day* form a key on *Reserves*

Using Indexes

- Selection $\sigma_{\text{day} = '12/4/2013'}$ is applied on-the-fly to the result of the selection $\sigma_{\text{bid} = 100}$ on the **Reserves** table
- Cost of selections
 - as before, cost of $\sigma_{\text{bid} = 100}$ is **10 page I/Os**
 - $\sigma_{\text{day} = '12/4/2013'}$ does not incur any cost as it is applied on-the-fly
- Cost of join
 - since $\langle \text{bid}, \text{day} \rangle$ is a key, there is only one outer tuple
 - cost of retrieving inner tuple is **1.2 page I/Os**
- **Total cost is ~ 11 page I/Os**
 - even with this selection, cost of naïve plan is still 501,000 I/Os!



A Typical Query Optimizer

- Recall that a typical query optimizer performs **two main** tasks
 1. enumeration of plans
 2. cost estimation for each plan
- Plan enumeration
 - **logical level:** algebraic equivalences are used to identify equivalent expression for a given query
 - **physical level:** for each such equivalent expression, all available implementation techniques are considered for the operators involved
- Cost estimation
 - cost of each query evaluation plan is estimated (see previous slides)
 - plan with lowest estimated cost is chosen and executed

Plan Enumeration

R Definition: equivalent expressions

Two relational algebra expressions over the same set of input tables are said to be **equivalent** if they produce the same result on all instances of the input tables.

1. Transform SQL query into relational algebra

- **FROM** t_1, t_2, \dots, t_n \rightarrow $((t_1 \times t_2) \times \dots) \times t_n$
- **WHERE** $a = v \text{ AND } \dots$ \rightarrow $\sigma_{a=v \wedge \dots}(\cdot)$
- **SELECT** a, b, c \rightarrow $\pi_{\{a,b,c\}}(\cdot)$

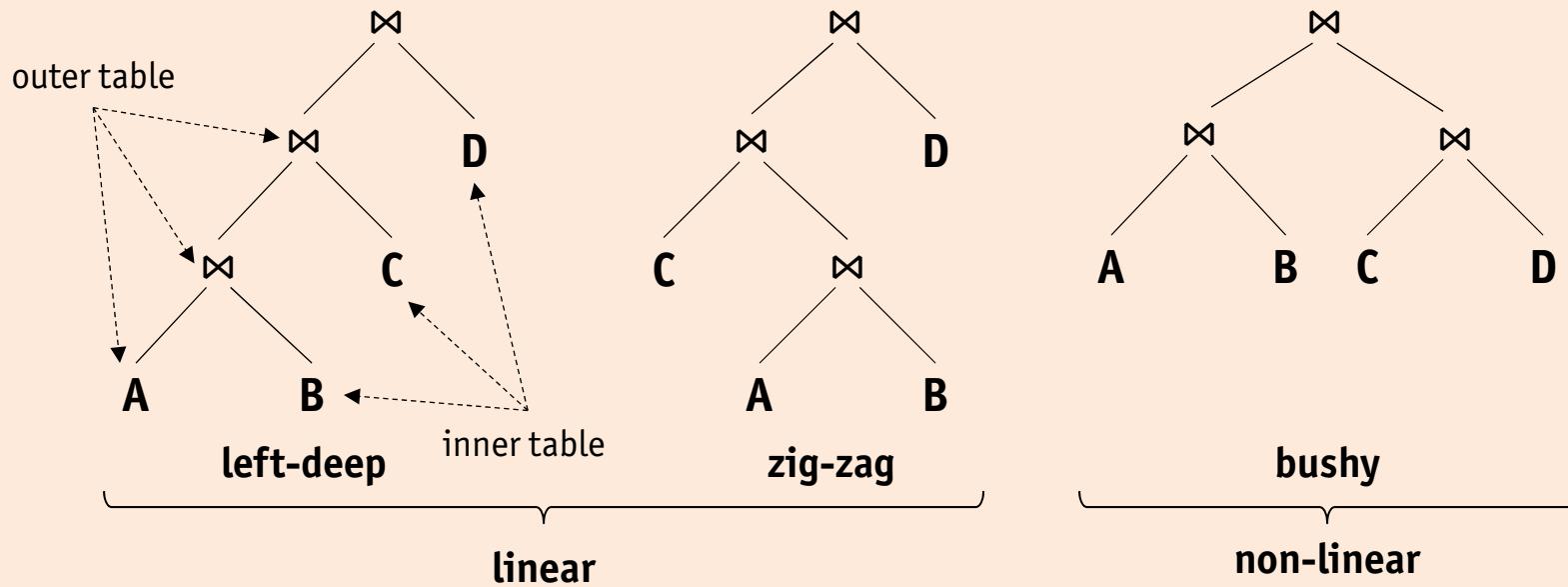
2. Apply relational algebra equivalences to convert initial expression into equivalent expression

- **selections** (σ) and **cross-products** (\times) can be combined into **joins** (\bowtie)
- **selections** (σ) and **projections** (π) can be “pushed” (or introduced) ahead of **joins** (\bowtie) to reduce the size of their inputs
- **joins** (\bowtie) can be extensively reordered

Join Reordering

Example: three join trees

Consider the natural join of four tables, i.e., $A \bowtie B \bowtie C \bowtie D$. Based on relational algebra equivalences (joins commute), the three trees below are equivalent.



Exercise: enumerating join orders

How many left-deep trees for a natural join of four tables are there?

Left-Deep Plans

- Left-deep plans are typically the only query plans considered
 - **dynamic programming** used to efficiently search this class of plans
 - given a specific join order, selection and projection conditions are applied **as early as possible**
- This decision implies that the query optimizer will **not** find the best plan, if the best plan is not a left-deep plan!
- Reasons to concentrate on left-deep plans
 - it is necessary to **prune** the search space since the number of alternative plans increases rapidly with the number of joins in a query
 - left-deep plans can be translated into **fully pipelined** plans, in which all joins are evaluated using pipelining (a join result cannot be used as inner table in a pipelined plan, since inner tables must always be materialized)

Cost Estimation of a Plan

Definition: cost of a plan

The cost of a plan is the **sum** of the cost for the operators it contains. The cost of individual operators is estimated using information obtained from the system catalog.

- Cost of a plan in terms of I/O costs can be broken down
 1. **reading input tables** (for some join and sort algorithms, multiple times)
 2. **writing intermediate tables**
 3. **sorting the final result** (for duplicate elimination or output order)
- Remarks
 - unless one of the plans happens to produce output in the required order, the third part is common to all plans
 - in the common case that a fully pipelined plan is chosen, no intermediate tables are written (and read)

Cost Estimation of a Plan

- Cost of fully pipelined plans is dominated by reading input tables
 - greatly depends on the access paths used to read input tables
 - access paths that are repeatedly used in joins are especially important
- Not fully pipelined plans
 - cost of materializing results as temporary tables can be significant
 - estimation of materializing intermediate results based on its size
 - result size also influences the cost of the operator for which it is an input
- Operator result size
 - σ estimated by multiplying with reduction factor (of selection condition)
 - π is equal to input size (no duplicate elimination)
 - \bowtie estimated by multiplying maximum result size (product of sizes of input tables) with reduction factor (of join condition)

Cost Estimation of a Plan

Exercise: approximating the reduction factor of a join

Suppose a query optimizer needs to estimate the result size of $A \bowtie_{aid=bid} B$. Further assume that there are indexes I_1 and I_2 on aid and bid , respectively. How would you approximate the reduction factor of the join condition?

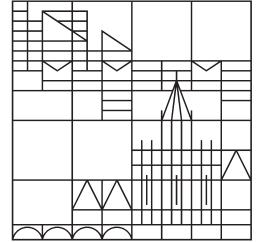
Cost Estimation of a Plan

Exercise: approximating the reduction factor of a join

Suppose a query optimizer needs to estimate the result size of $A \bowtie_{aid=bid} B$. Further assume that there are indexes I_1 and I_2 on aid and bid , respectively. How would you approximate the reduction factor of the join condition?

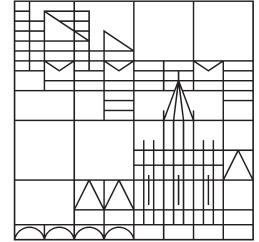
$$\frac{1}{\max(NKeys(I_1), NKeys(I_2))}$$

This formula assumes that each key in the smaller index, say I_1 , has a matching value in the other index. Given a value for aid , we assume that each of the $NKeys(I_2)$ values of bid is equally likely. Thus, the number of tuples that have the same value in bid as a given value in aid is $\frac{1}{NKeys(I_2)}$.



Database System Architecture and Implementation

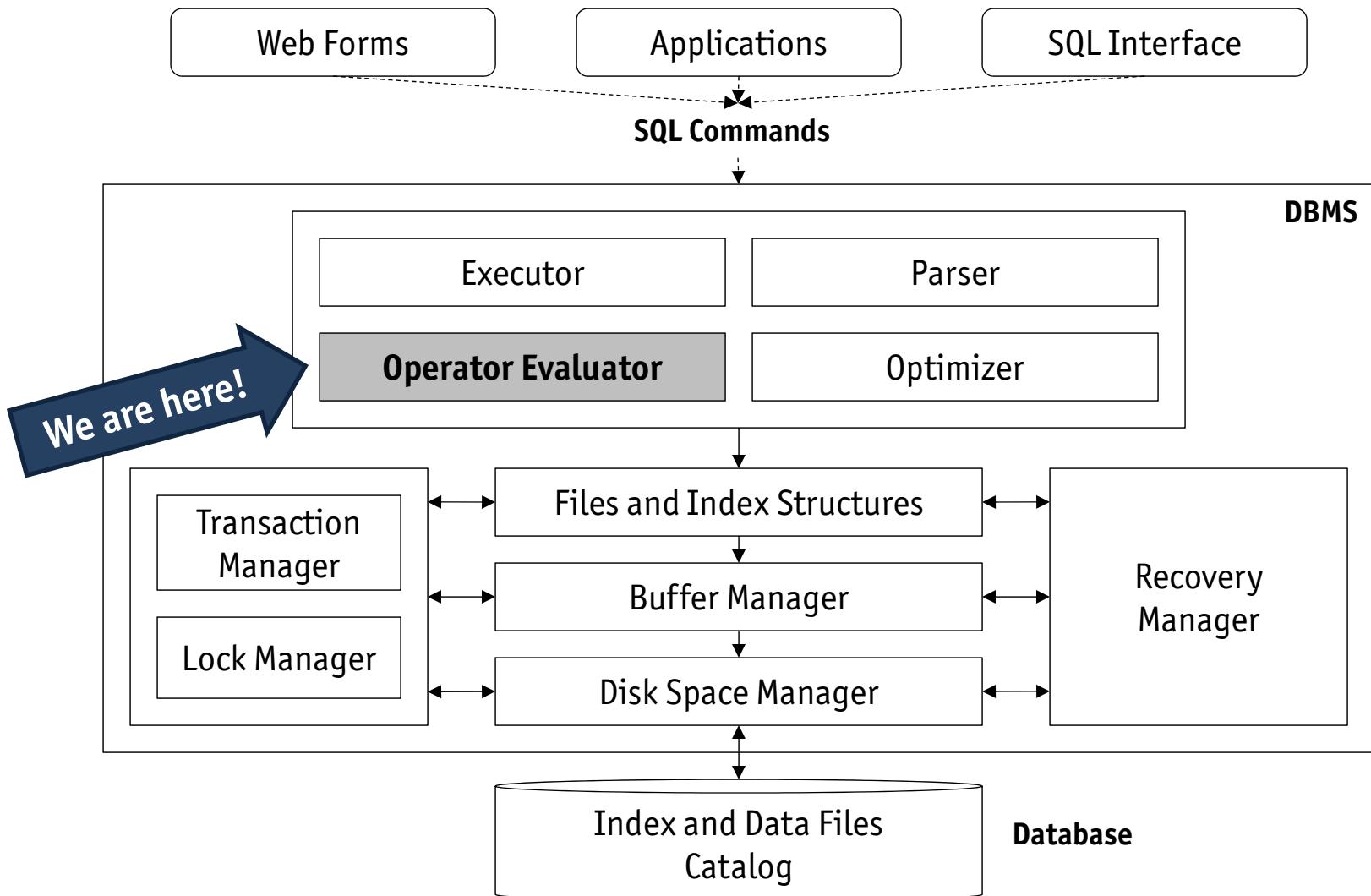
TO BE CONTINUED...



Database System Architecture and Implementation

Module 6
External Sorting
December 9, 2013

Orientation



Module Overview

- Overview of sorting
- Two-way merge sort
- External merge sort
 - longer initial runs using selection sort
 - better CPU usage using double buffering
- Using B+ trees for sorting

Sorting

💻 A (not so) simple SQL query

```
SELECT DISTINCT S.sid, S.sname, S.rating  
FROM Reserves AS R, Sailors AS S  
WHERE R.sid = S.sid  
GROUP BY S.sid, S.sname, S.rating, R.day  
HAVING COUNT(R.bid) > 1  
ORDER BY S.rating DESC
```

📝 Possible reasons to sort

Which operations in the above query might require sorting?

Sorting

💻 A (not so) simple SQL query

```
SELECT DISTINCT S.sid, S.sname, S.rating  
  FROM Reserves AS R, Sailors AS S  
 WHERE R.sid = S.sid  
 GROUP BY S.sid, S.sname, S.rating, R.day  
 HAVING COUNT(R.bid) > 1  
 ORDER BY S.rating DESC
```

📝 Possible reasons to sort

Which operations in the above query might require sorting?

- ↳ **duplicate elimination** and grouping are typically implemented using sorting
- ↳ some **join** algorithms (e.g., sort-merge join) have a sort step
- ↳ **ordering** results in ascending or descending order requires sorting
- ↳ **bulk loading** of a tree index performs sorting as a first step

Sorting

R Definitions

- A file is **sorted** with respect to **sort key** k and **ordering** θ , if for any two records r_1, r_2 in the file, their corresponding keys are in θ -order

$$r_1 \theta r_2 \quad \Leftrightarrow \quad r_1.k \theta r_2.k$$

- A key may be a single attribute as well as an ordered list of attributes. In the latter case, order is defined **lexicographically**. Consider $k = (\mathbf{A}, \mathbf{B})$, $\theta = <$

$$\begin{aligned} r_1 < r_2 \quad &\Leftrightarrow \quad r_1.\mathbf{A} < r_2.\mathbf{A} \vee \\ &\quad (r_1.\mathbf{A} = r_2.\mathbf{A} \wedge r_1.\mathbf{B} < r_2.\mathbf{B}) \end{aligned}$$

Sorting

- If the data to be sorted is too large to fit into available main memory (buffer pool), an **external sorting** algorithm is required
- Stepwise design of an external sorting algorithm
 1. **simple algorithm:** only three pages of buffer space are sufficient to sort a file of arbitrary size
 2. **refined algorithm:** simple algorithm can be adapted to make effective use of larger (and more realistic) buffer sizes
 3. **optimized algorithm:** a number of further optimizations can be applied to reduce the number and duration of required page I/O operations

Two-Way Merge Sort

- Two-way merge sort can sort files of arbitrary size with only **three pages** of available buffer space

Two-way merge sort

Two-way merge sort sorts a file with $N = 2^k$ pages in multiple **passes**, each of which produces a certain number of sorted sub-files, so-called **runs**.

- **Pass 0** sorts each of the 2^k input pages individually and in **main memory**, resulting in 2^k sorted runs.
- **Subsequent passes** merge pairs of runs into larger runs. Pass n produces 2^{k-n} runs.
- **Pass k** produces only one final run, the overall sorted result.

During each pass, every page of the file is read and written. Therefore, a total of **$2 \cdot k \cdot N$ page I/O operations** are required to sort the file.

Exercise: forwards in Pass 0?

What happens if the pages that are sorted in Pass 0 contain **forwards** to moved records?

Two-Way Merge Sort

Pass 0

(**input:** $N = 2^k$ unsorted pages, **output:** 2^k sorted pages)

1. **Read** N pages, one page at a time
2. **Sort** records, page-wise, in main memory
3. **Write** sorted pages to disk (each page results in a **run**)

This pass requires **one page** of buffer space

Pass 1

(**input:** $N = 2^k$ sorted pages, **output:** 2^{k-1} sorted runs)

1. **Open** two runs r_1 and r_2 from Pass 0 for reading
2. **Merge** records from r_1 and r_2 , reading input page by page
3. **Write** new two-page run to disk, page by page

This pass requires **three pages** of buffer space

⋮

Pass n

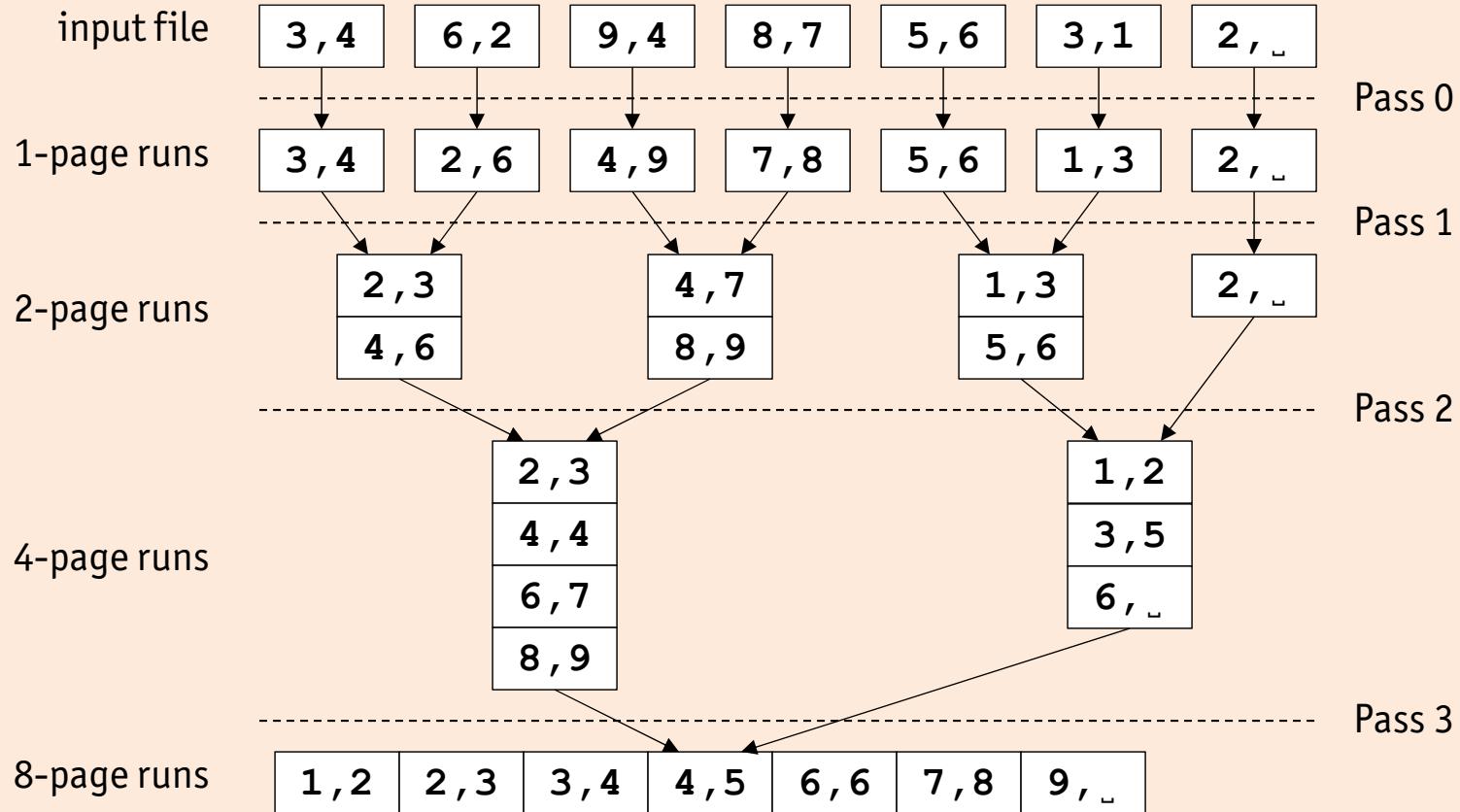
(**input:** $N = 2^{k-n+1}$ sorted runs, **output:** 2^{k-n} sorted runs)

1. **Open** two runs r_1 and r_2 from Pass $n - 1$ for reading
2. **Merge** records from r_1 and r_2 , reading input page by page
3. **Write** new 2^n -page run to disk, page by page

This pass requires **three pages** of buffer space

Two-Way Merge Sort

Example: 7-page file, 2 records/page, keys k shown, $\theta = <$



Two-Way Merge Sort

💻 Two-way merge sort ($N = 2^k$, ordering θ , output in file “run $_k_\theta$ ”)

```
function 2-way-merge-sort(file, N)
    for page number p in 0... $2^k - 1$  do          (Pass 0: write  $2^k$  sorted single-page runs)
         $\uparrow p \leftarrow \text{pin}(file, p)$  ;
         $f_0 \leftarrow \text{createFile}(\text{"run}_0\_\theta")$  ;   ("run $_n\_\theta$ " contains the  $r^{\text{th}}$  run of Pass  $n$ )
        sort the records on page pointed to by  $\uparrow p$  according to  $\theta$  ;
        write page pointed to by  $\uparrow p$  into file  $f_0$  ;
        closeFile( $f_0$ ) ;
        unpin(file, p, false) ;
    for n in 1...k do                      (Passes 1...k)
    end
```

- Remark
 - the in-memory sort and merge steps can be implemented using **standard sorting techniques**, e.g., quick-sort

Two-Way Merge Sort

💻 Two-way merge sort ($N = 2^k$, ordering θ , output in file “run $_k_\theta$ ”)

```
function 2-way-merge-sort(file, N)
    for each page number  $p$  in  $0 \dots 2^k - 1$  do    (Pass 0: write  $2^k$  sorted single-page runs)
        for  $n$  in  $1 \dots k$  do
            for  $r$  in  $0 \dots 2^{k-n}$  do          (pair-wise merge all runs written in Pass  $n - 1$ )
                 $f_1 \leftarrow \text{openFile}(\text{"run}\_(n-1)\_(2\cdot r)");$ 
                 $f_2 \leftarrow \text{openFile}(\text{"run}\_(n-1)\_(2\cdot r+1)");$ 
                 $f_0 \leftarrow \text{createFile}(\text{"run}\_n\_\theta");$ 
                for page number  $p$  in  $0 \dots 2^{n-1} - 1$  do
                     $\uparrow p_1 \leftarrow \text{pin}(f_1, p); \uparrow p_2 \leftarrow \text{pin}(f_2, p);$ 
                    merge the records on pages pointed to by  $\uparrow p_1, \uparrow p_2$  according to  $\theta$ ;
                    append resulting two pages to file  $f_0$ ;           (size( $f_0$ ) = size( $f_1$ ) + size( $f_2$ ))
                     $\text{unpin}(f_1, p, \text{false}); \text{unpin}(f_2, p, \text{false});$ 
                closeFile( $f_0$ );
                deleteFile( $f_1$ );
                deleteFile( $f_2$ );
            end
        end
    end
```

Two-Way Merge Sort

- Each pass in a two-way merge of a file of N pages

- pass **reads** N pages in
 - sorts/merges in memory
 - **writes** N pages out again

$\left. \begin{array}{l} \\ \\ \end{array} \right\} 2 \cdot N \text{ page I/O operations per pass}$

- Number of passes

$$\underbrace{1}_{\text{Pass 0}} + \underbrace{\lceil \log_2 N \rceil}_{\text{Passes } 1, \dots, k}$$

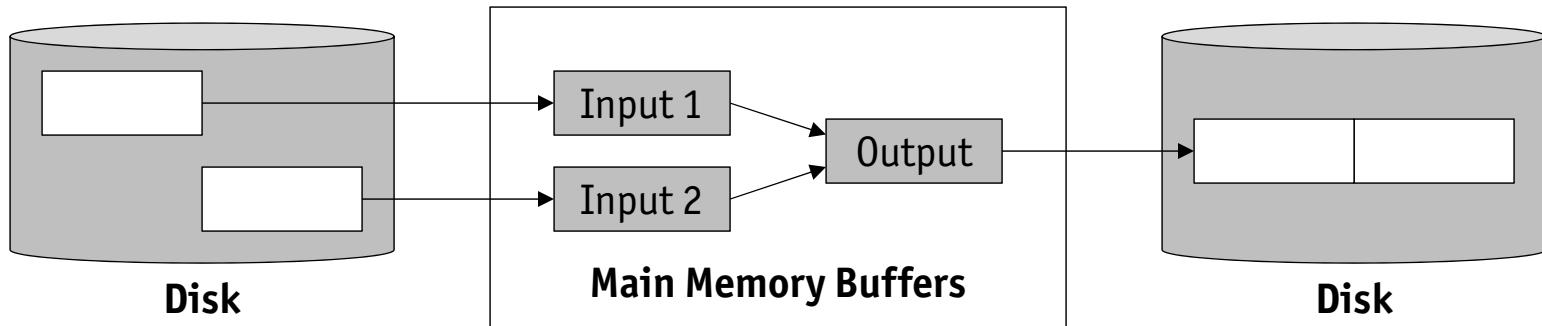
- Total number of I/O operations

$$2 \cdot N \cdot (1 + \lceil \log_2 N \rceil)$$

 **Exercise:** how many page I/O operations does it take to sort an 8 GB file?

Assume a page size of 8 kB (with 1000 records each).

Two-Way Merge Sort

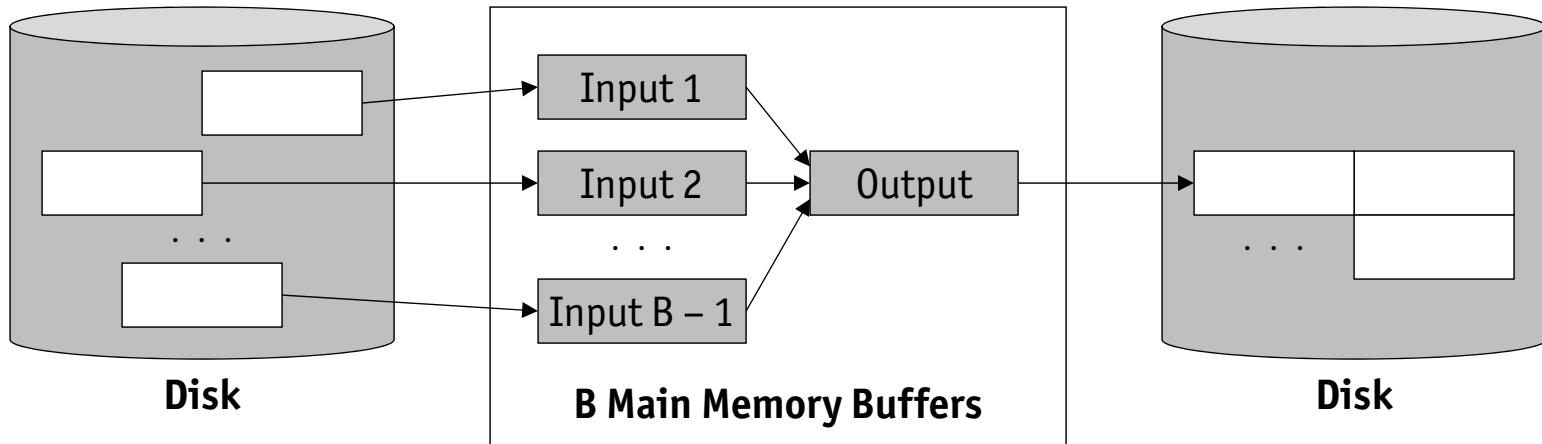


- At any point in time, two-way merge sort uses **no more than three pages** of buffer space
 - consider the `pin(·)` calls in the pseudo-code of the algorithm
 - this restriction is “voluntarily”
- In reality, **many** more free buffer pages will be available and this sort algorithm can be refined to make efficient use of them

External Merge Sort

- **External merge sort** introduces two improvements over simple two-way merge sort
 - **reduce the number of runs** by using the full buffer space during to avoid creating one-page runs in Pass 0
 - **reduce the number of passes** by merging more than two runs at a time

External Merge Sort



- Suppose B pages are available in the buffer pool
 - **B pages can be read at a time** during Pass 0 and sorted in memory
 - **$B - 1$ pages can be merged at a time** (leaving one page as a write buffer)

External Merge Sort

Pass 0

(**input:** N = unsorted pages, **output:** $\lceil \frac{N}{B} \rceil$ sorted pages)

1. **Read** N pages, B pages at a time
2. **Sort** records, page-wise, in main memory
3. **Write** sorted pages to disk (resulting in $\lceil \frac{N}{B} \rceil$ runs)

This pass uses B pages of buffer space

⋮

Pass n

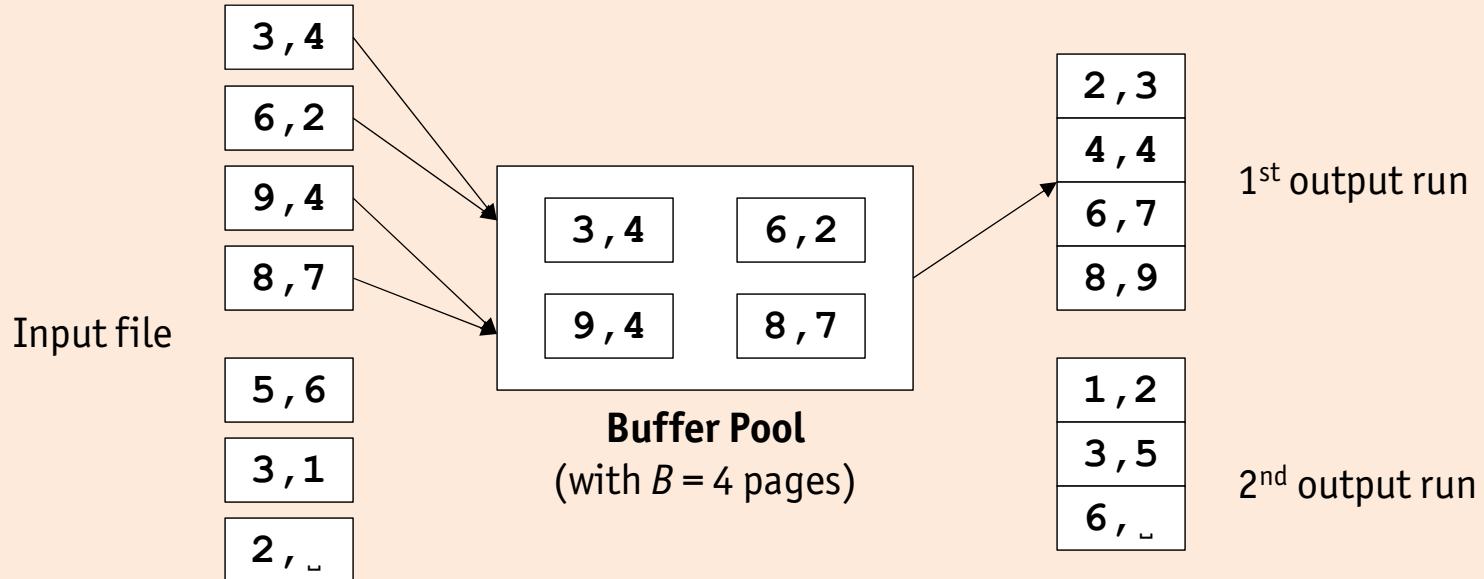
(**input:** $\frac{\lceil N/B \rceil}{(B-1)^{n-1}}$ sorted runs, **output:** $\frac{\lceil N/B \rceil}{(B-1)^n}$ sorted runs)

1. **Open** $B - 1$ runs r_1, \dots, r_{B-1} from Pass $n - 1$ for reading
2. **Merge** records from r_1, \dots, r_{B-1} , reading input page by page
3. **Write** new $B \cdot (B - 1)^n$ -page run to disk, page by page

This pass requires B pages of buffer space

External Merge Sort

Example: 7-page file, 2 records/page, keys k shown, $\theta = <$, 4 buffer pages, pass 0



External Merge Sort

- As in two-way merge sort, each pass reads, processes, and then writes **all N** pages
- The number of initial runs **determines** the number of passes
 - in Pass 0, $\lceil N/B \rceil$ runs are written
 - number of additional passes is thus $\lceil \log_2 \lceil N/B \rceil \rceil$
- With B pages of buffer space, we can do a **$(B - 1)$ -way merge**
 - number of additional passes is thus $\lceil \log_{B-1} \lceil N/B \rceil \rceil$
- Total number of page I/O operations
$$2 \cdot N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$$

 **Exercise:** how many page I/O operations does it take to sort an 8 GB file now?

Assume a page size of 8 kB. Available buffer space is $B = 1000$.

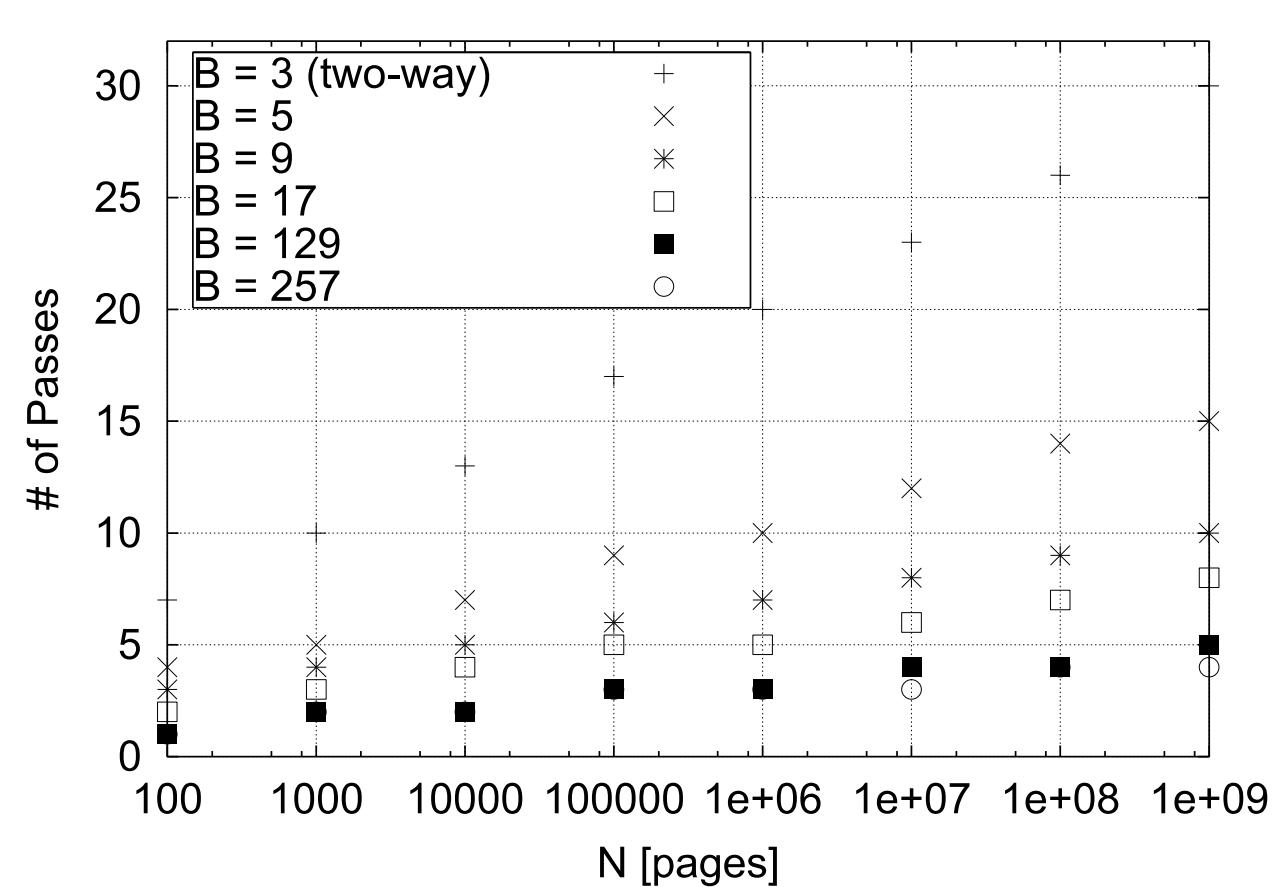
External Merge Sort

N	$B = 3$	$B = 5$	$B = 9$	$B = 17$	$B = 129$	$B = 257$
100	7	4	3	2	1	1
1000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Number of Passes of External Merge Sort

External Merge Sort

Number of passes for buffers of size $B = 3, 5, \dots, 257$



External Merge Sort

Exercise: I/O access pattern

Sorting N pages with B buffer pages requires

$$2 \cdot N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$$

page I/O operations. What is the access pattern of these I/O operations?

External Merge Sort

Exercise: I/O access pattern

Sorting N pages with B buffer pages requires

$$2 \cdot N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$$

page I/O operations. What is the access pattern of these I/O operations?

- ↳ in Pass 0, disk blocks of size B are read **sequentially**
- ↳ in all other passes, disk blocks are read with **random access**
(which of the $B - 1$ runs contains the next record to be merged?)

Blocked I/O

- I/O pattern can be improved by using **blocked I/O**
 - allocate b pages for each input (instead of just one)
 - read **blocks of b pages** at once during the **merge** phases
- Trade-off between I/O cost and number of I/O operations
 - **reduces** I/O cost per page by a factor of $\approx b$
 - reading blocks of pages **decreases the fan-in**, which increases the number of passes and therefore the number of I/O operations
- Total number of page I/O operations
$$2 \cdot N \cdot (1 + \lceil \log_{\lfloor B/b \rfloor - 1} \lceil N/B \rceil \rceil)$$
- In practice, main memory sizes are typically large enough to sort files with **just one merge pass**, even with blocked I/O

Blocked I/O

N	$B = 1000$	$B = 5000$	$B = 10,000$	$B = 50,000$
100	1	1	1	1
1000	1	1	1	1
10,000	2	2	1	1
100,000	3	2	2	2
1,000,000	3	2	2	2
10,000,000	4	3	3	2
100,000,000	5	3	3	2
1,000,000,000	5	4	3	3

Number of Passes of External Merge Sort with Block Size $b = 32$

Blocked I/O

 **Exercise:** How long does it take to sort 8 GB (counting I/O cost only)?

Assume 1000 buffer pages of 8 kB each, 8.5 ms average seek time, 60 MB/s transfer rate, and blocks of $b = 32$ pages.

Without blocked I/O

With blocked I/O

Blocked I/O

Exercise: How long does it take to sort 8 GB (counting I/O cost only)?

Assume 1000 buffer pages of 8 kB each, 8.5 ms average seek time, 60 MB/s transfer rate, and blocks of $b = 32$ pages.

Without blocked I/O

- ↳ recall that sorting in this case takes $\approx 6 \cdot 10^6$ page I/O operations
- ↳ $\approx 4 \cdot 10^6$ disk seeks (9.4 h)
- ↳ $\approx 6 \cdot 10^6$ disk pages transfers (13.6 min)

With blocked I/O

- ↳ sorting in this case with blocked I/O takes $\approx 8 \cdot 10^6$ page I/O operations
- ↳ $\approx 6 \cdot 32,800$ ($1,048,576/32$) disk seeks (28.1 min)
- ↳ $\approx 8 \cdot 10^6$ disk pages transfers (18.1 min)

CPU Load

- External merge sort reduces the I/O load, but is considerably **more CPU intensive**
 - since I/O cost dominates the overall cost, this price is acceptable
- Consider the **$(B - 1)$ -way merge** during passes 1, 2, ...
 - to pick the next record to be moved to the output buffer, $B - 2$ comparisons need to be performed

Example: Comparisons for $B - 1 = 4$, $\theta = <$

$\left\{ \begin{array}{l} 087 \ 503 \ 504 \dots \\ 170 \ 908 \ 994 \dots \\ 154 \ 426 \ 653 \dots \\ 612 \ 613 \ 700 \dots \end{array} \right. \rightarrow 087$

$\left\{ \begin{array}{l} 503 \ 504 \dots \\ 170 \ 908 \ 994 \dots \\ 154 \ 426 \ 653 \dots \\ 612 \ 613 \ 700 \dots \end{array} \right. \rightarrow 087 \ 154$

$\left\{ \begin{array}{l} 503 \ 504 \dots \\ 170 \ 908 \ 994 \dots \\ 426 \ 653 \dots \\ 612 \ 613 \ 700 \dots \end{array} \right. \rightarrow 087 \ 154 \ 170$

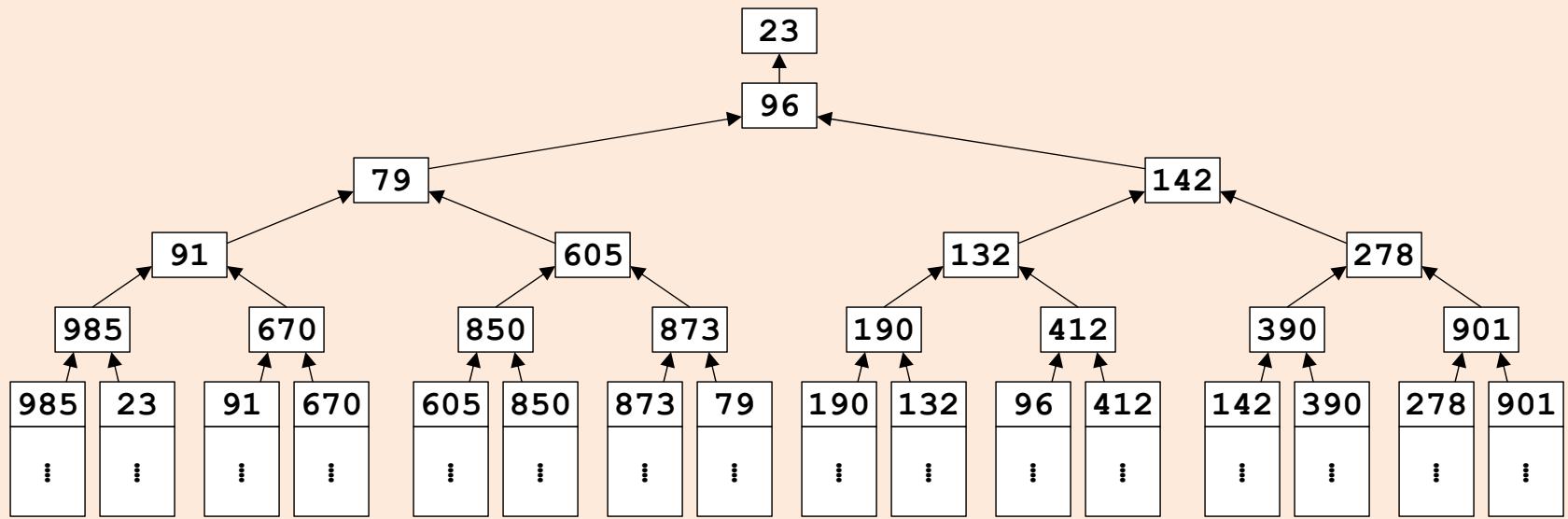
$\left\{ \begin{array}{l} 503 \ 504 \dots \\ 908 \ 994 \dots \\ 426 \ 653 \dots \\ 612 \ 613 \ 700 \dots \end{array} \right. \rightarrow 087 \ 154 \ 170 \ 426$

$\left\{ \begin{array}{l} 503 \ 504 \dots \\ 908 \ 994 \dots \\ 653 \dots \\ 612 \ 613 \ 700 \dots \end{array} \right. \rightarrow 087 \ 154 \ 170 \ 426 \ 503$... etc.

CPU Load

- Number of comparisons can be reduced using a **selection tree**
 - “tree of losers” (D. Knuth: “The Art of Computer Programming”, vol. 3)
 - this optimization cuts the number of comparisons down to $\log_2(B - 1)$
 - for buffer sizes $B \gg 100$ this is a considerable improvement

Example: Selection tree, read bottom-up



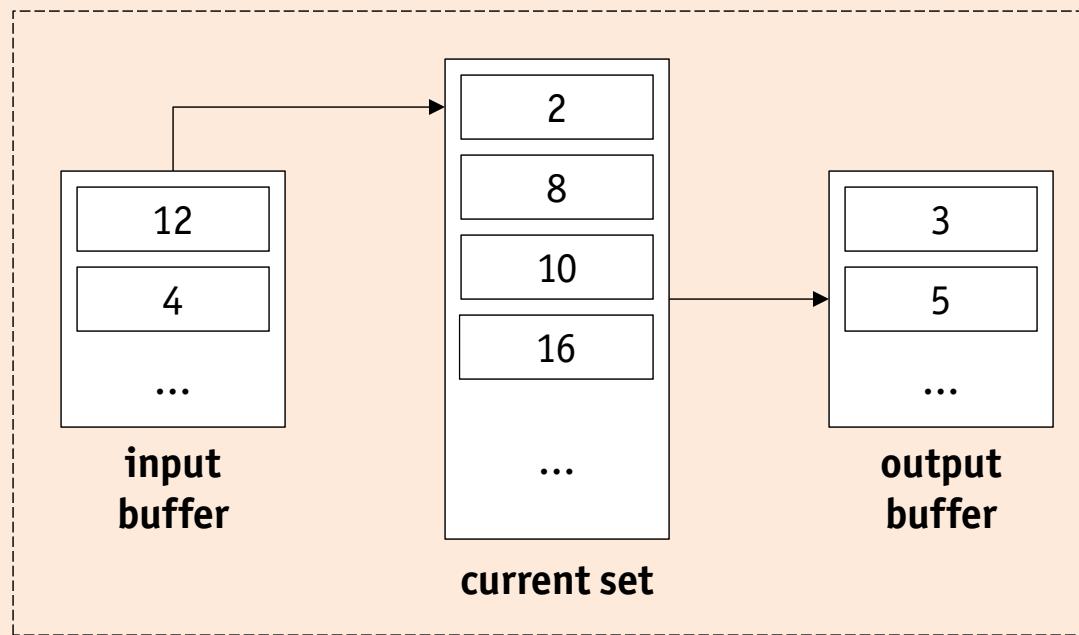
Minimizing the Number of Runs

- Number of initial runs **determines** number of required passes
 - initial runs are the files “run_0_r” written in Pass 0
 - because of $2 \cdot N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$, $r = 0, \dots, \lceil N/B \rceil - 1$
- Reducing the number of initial runs is a **desirable optimization**
- **Replacement sort** is an example of such an optimization
 - **cut down** the number of $\lceil N/B \rceil$ initial runs in Pass 0
 - produce initial runs with **more than B pages**

Minimizing the Number of Runs

Replacement sort

Assume a buffer pool with B pages. Two pages are dedicated **input** and **output buffers**. The remaining $B - 2$ pages are called the **current set**.



Minimizing the Number of Runs

R Replacement sort

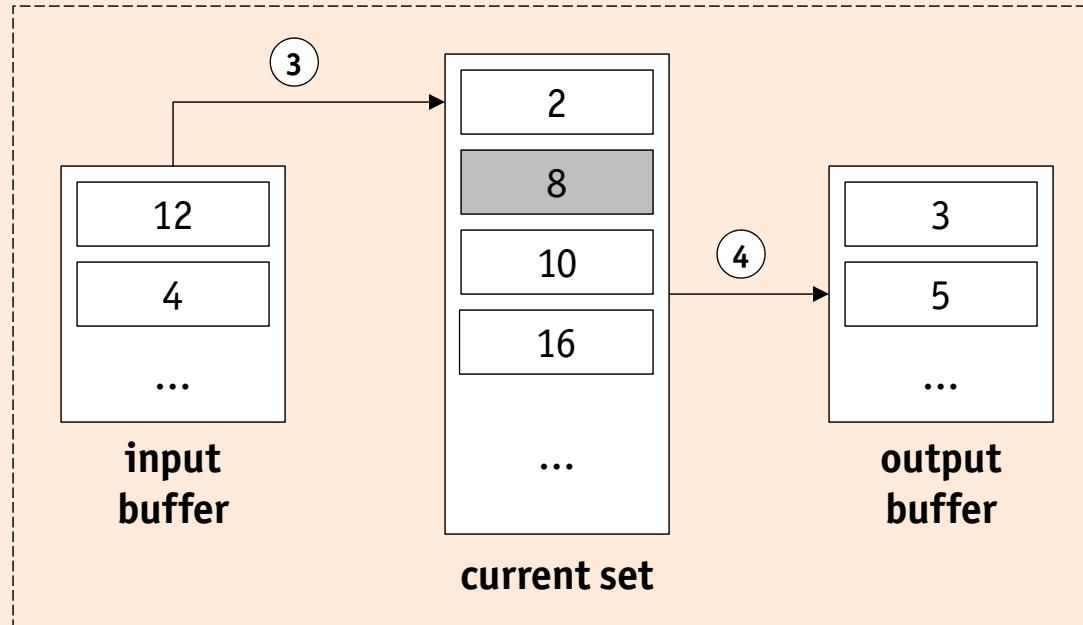
1. Open an empty run file for writing.
2. Load next page of file to be sorted into input buffer. If input file is exhausted, go to 4.
3. While there is space in the current set, move a record from input buffer to current set (if the input buffer is empty, reload it at 2.)
4. In current set, pick record r with smallest key value k such that $k \geq k_{out}$, where k_{out} is the maximum key value in output buffer (if output buffer is empty, define $k_{out} = -\infty$). Move r to output buffer. If output buffer is full, append is to current run.
5. If all k in current set are $< k_{out}$, append output buffer to current run, close current run. Open new empty run file writing.
6. If input file is exhausted, stop. Otherwise go to 3.

- Remark
 - of course, Step 4 of replacement sort will benefit from techniques like the **selection tree**, especially if $B - 2$ (size of current set) is large

Minimizing the Number of Runs

Example

Record with key $k = 8$ will be the next to be moved into the output buffer, current $k_{out} = 5$



The record with $k = 2$ **remains** in the current set and will be written to the **subsequent** run.

Minimizing the Number of Runs

Exercise: tracing replacement sort

Assume $B = 6$, i.e., a current set size of 4. The input files contains records with **INTEGER** key values

503 087 512 061 908 170 897 275 426 154 509 612

Write a trace of replacement sort by filling out the table below, mark the end of the current run by **(EOR)**. The current set has already been populated at Step 3.

current set				output
503	087	512	061	

Minimizing the Number of Runs

Exercise: tracing replacement sort

Assume $B = 6$, i.e., a current set size of 4. The input files contains records with **INTEGER** key values

503 087 512 061 908 170 897 275 426 154 509 612

Write a trace of replacement sort by filling out the table below, mark the end of the current run by **(EOR)**. The current set has already been populated at Step 3.

current set				output
503	087	512	061	061
503	087	512	908	087
503	170	512	908	170
503	897	512	908	503
275	897	512	908	512
275	897	426	908	897
275	154	426	908	908
275	154	426	509	(EOR)
275	154	426	509	154
275	612	426	509	275
	612	426	509	426

Minimizing the Number of Runs

Length of initial runs

The trace of replacement sort suggests that the length of the initial runs indeed increases. In the example, the length of the first run is $8 \approx 2 \cdot B$, i.e., twice the size of the current set.

Exercise: analyzing the length of initial runs

How would you determine the average length of initial runs created by replacement sort in general?

Minimizing the Number of Runs

Length of initial runs

The trace of replacement sort suggests that the length of the initial runs indeed increases. In the example, the length of the first run is $8 \approx 2 \cdot B$, i.e., twice the size of the current set.

Exercise: analyzing the length of initial runs

How would you determine the average length of initial runs created by replacement sort in general?

- ↳ **implement** replacement sort to determine initial run length empirically
- ↳ **check** proper analysis (D. Knuth: “The Art of Computer Programming”, vol. 3, p. 254)

CPU Load

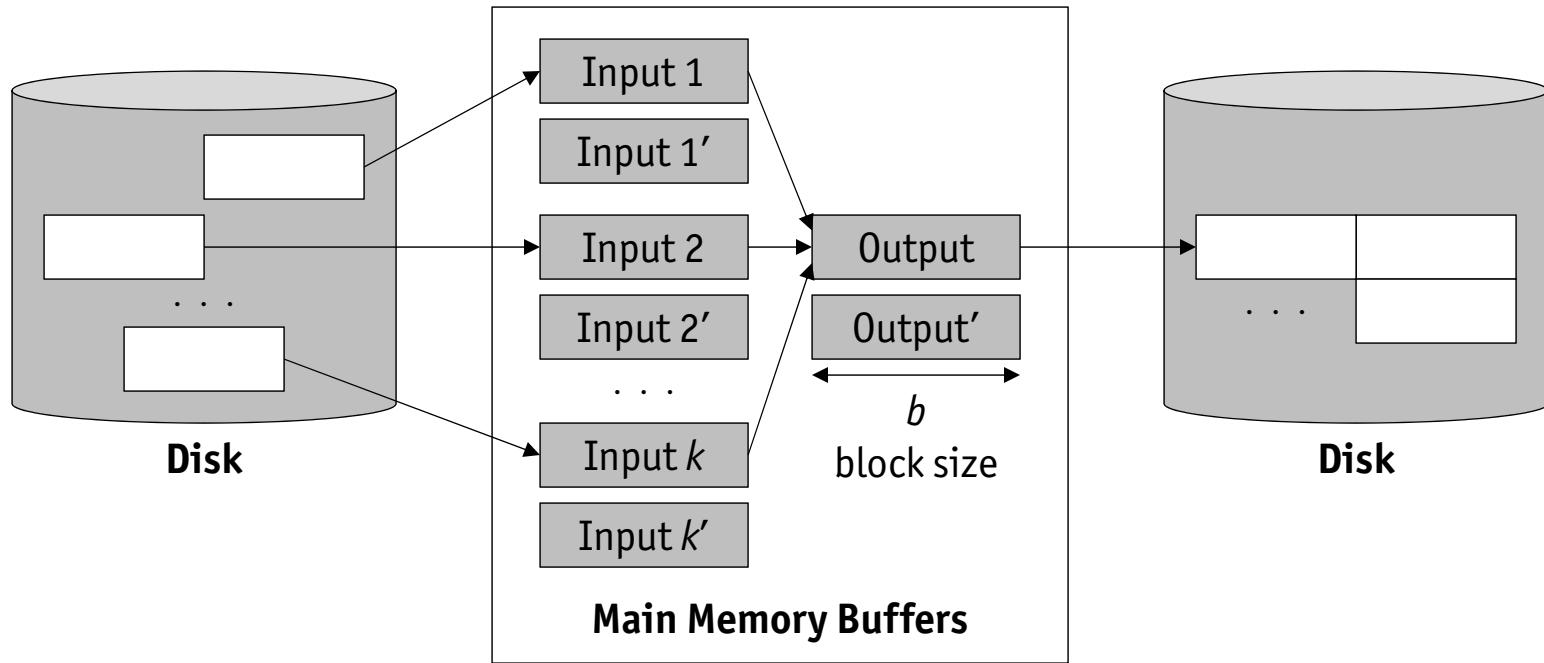
- External merge sort follows a **divide and conquer** principle
 - results in a number of **independent (sub-)tasks**
 - **execute tasks in parallel** in a parallel and distributed DBMS or exploit multi-core parallelism on modern CPUs
- To minimize **query response time**, CPU should never remain idle
 - avoid wait for input buffer to be **reloaded**
 - avoid wait for output buffer to be **appended** to current run

Double buffering

To avoid CPU waits, **double buffering** can be used.

1. create a second **shadow buffers** for each input and output buffer
2. CPU **switches** to the “double” buffer, once original buffers is empty/full
3. original buffer is reloaded by **asynchronously** initiating an I/O operation
4. CPU **switches** back to original buffer, once “double” buffer is empty/full, etc.

CPU Load



Exercise: latency vs. throughput

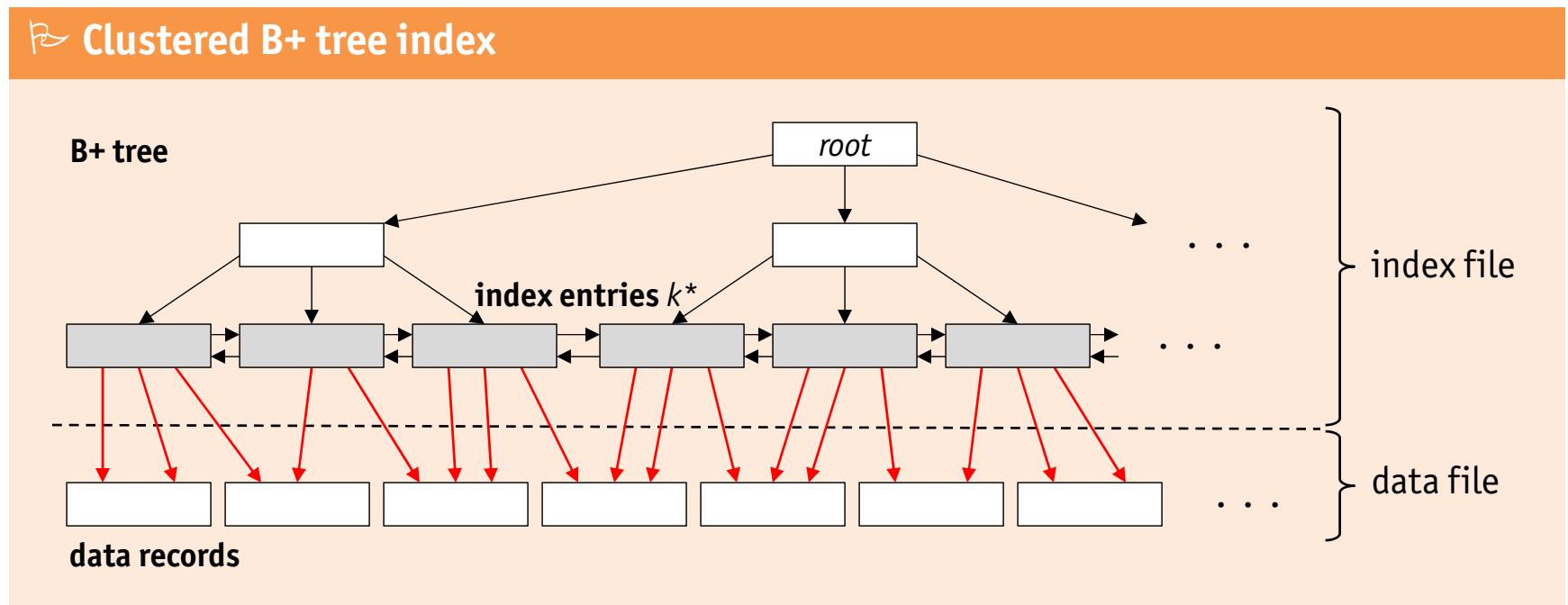
Double buffering minimizes query response time. Does it also impact query throughput?

Using B+ Trees for Sorting

- Suppose that a B+ tree index matches a sorting task
 - B+ tree organized over key k with ordering θ
 - using the index and abandoning external sorting **may** be better
- Decision whether to use the index or not depends on the nature of the index
 - if index is clustered or unclustered
 - index entries used (variants ①, ② and ③)

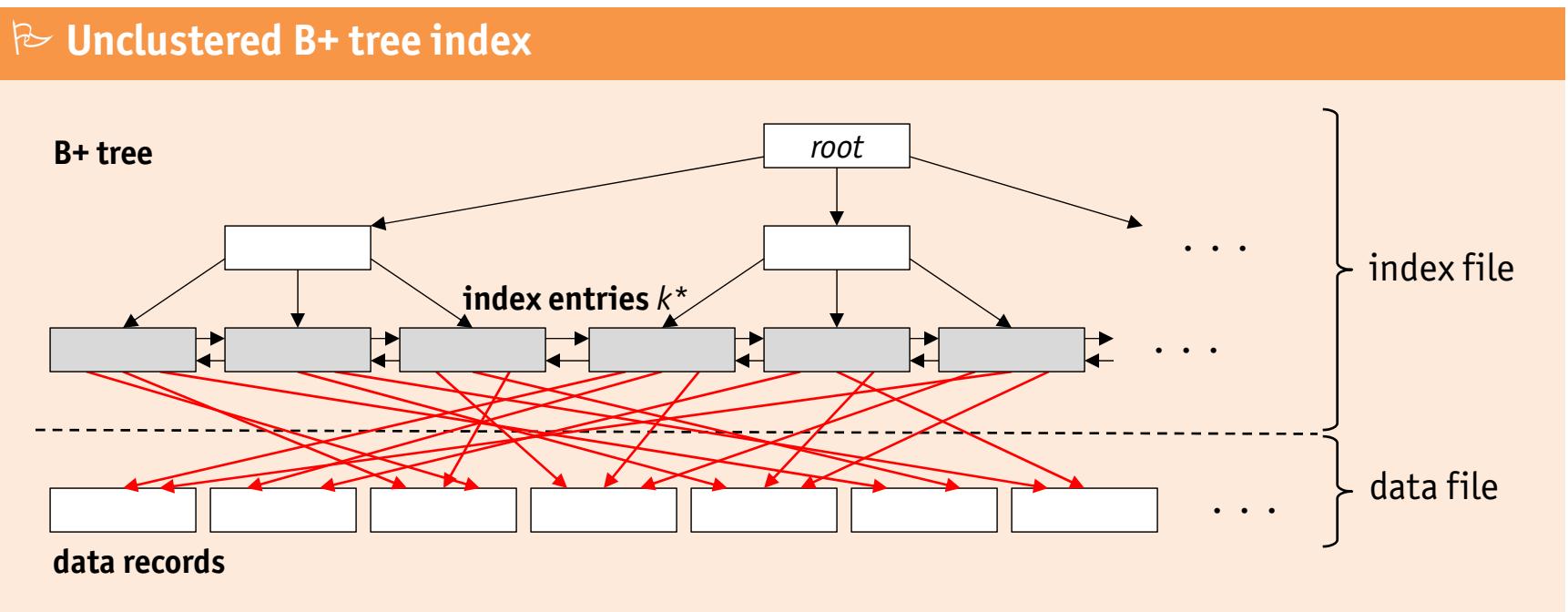
Using B+ Trees for Sorting

- B+ tree index is **clustered**
 - data file itself is already θ -sorted
 - suffices to read all N pages of the file, regardless of the variant of index entries used



Using B+ Trees for Sorting

- B+ tree index is **unclustered**
 - in the worst case, one page I/O operation has to be initiated per record (not per page) in the file!
 - therefore, do not use an unclustered index for sorting



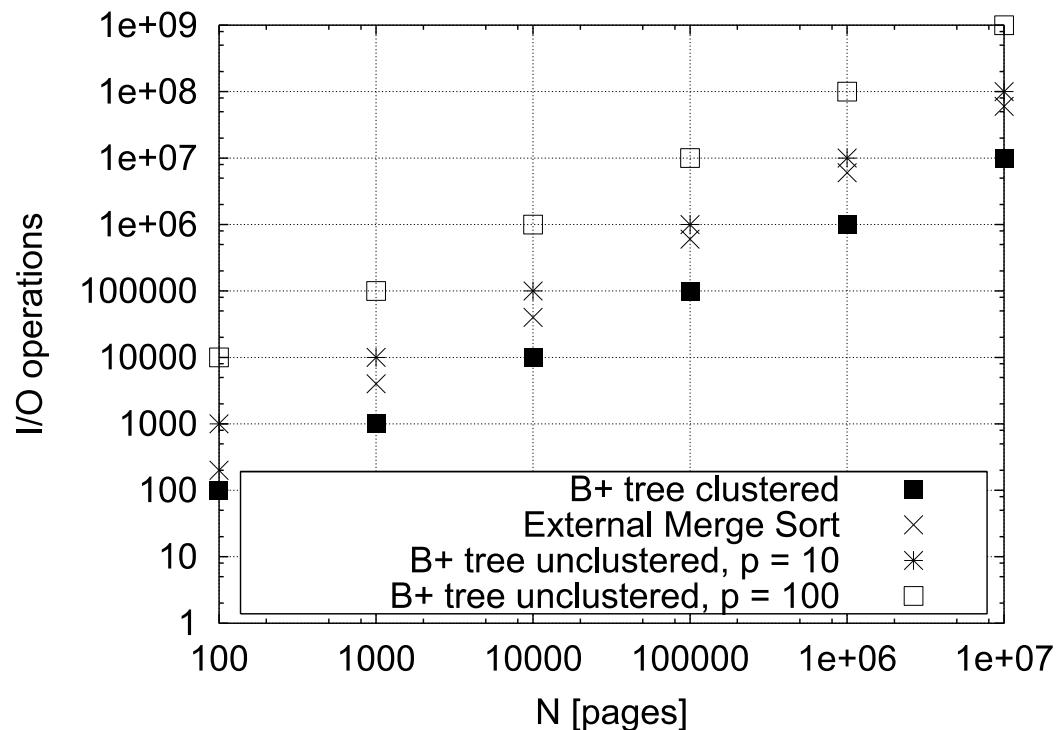
Using B+ Trees for Sorting

Expected page I/O operations for sorting

Let p denote the number of data records per page (typical values are $p = 10, \dots, 1000$). The expected number of page I/O operations to sort using an unclustered B+ tree index is therefore $p \cdot N$ (worst case)

Assumptions in plot

- available buffer space for sorting is $B = 257$ pages
- ignore I/O to traverse B+ tree as well as its sequence set



- Even for modest file sizes, sorting by using an unclustered B+ tree index is **clearly inferior** to external sorting.

Sorting in Commercial DBMS

The Real World

↳ **IBM DB2, Microsoft SQL Server, and Oracle**

- all systems use external merge sort
- all systems use asynchronous I/O and prefetching
- none of these systems uses optimization that produces runs larger than available memory, in part because it is difficult to implement it efficiently in the presence of variable-length records

↳ **IBM DB2**

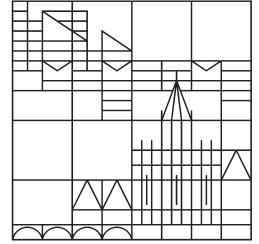
- uses a separate area of memory to do sorting
- uses radix sort as in-memory sorting algorithm

↳ **Oracle**

- uses a separate area of memory to do sorting
- uses insertion sort as in-memory sorting algorithm

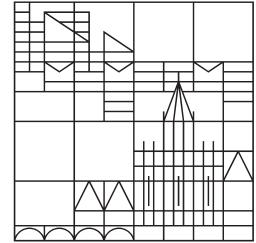
↳ **Microsoft SQL Server**

- uses buffer pool frames for sorting
- uses merge sort as in-memory sorting algorithm



Database System Architecture and Implementation

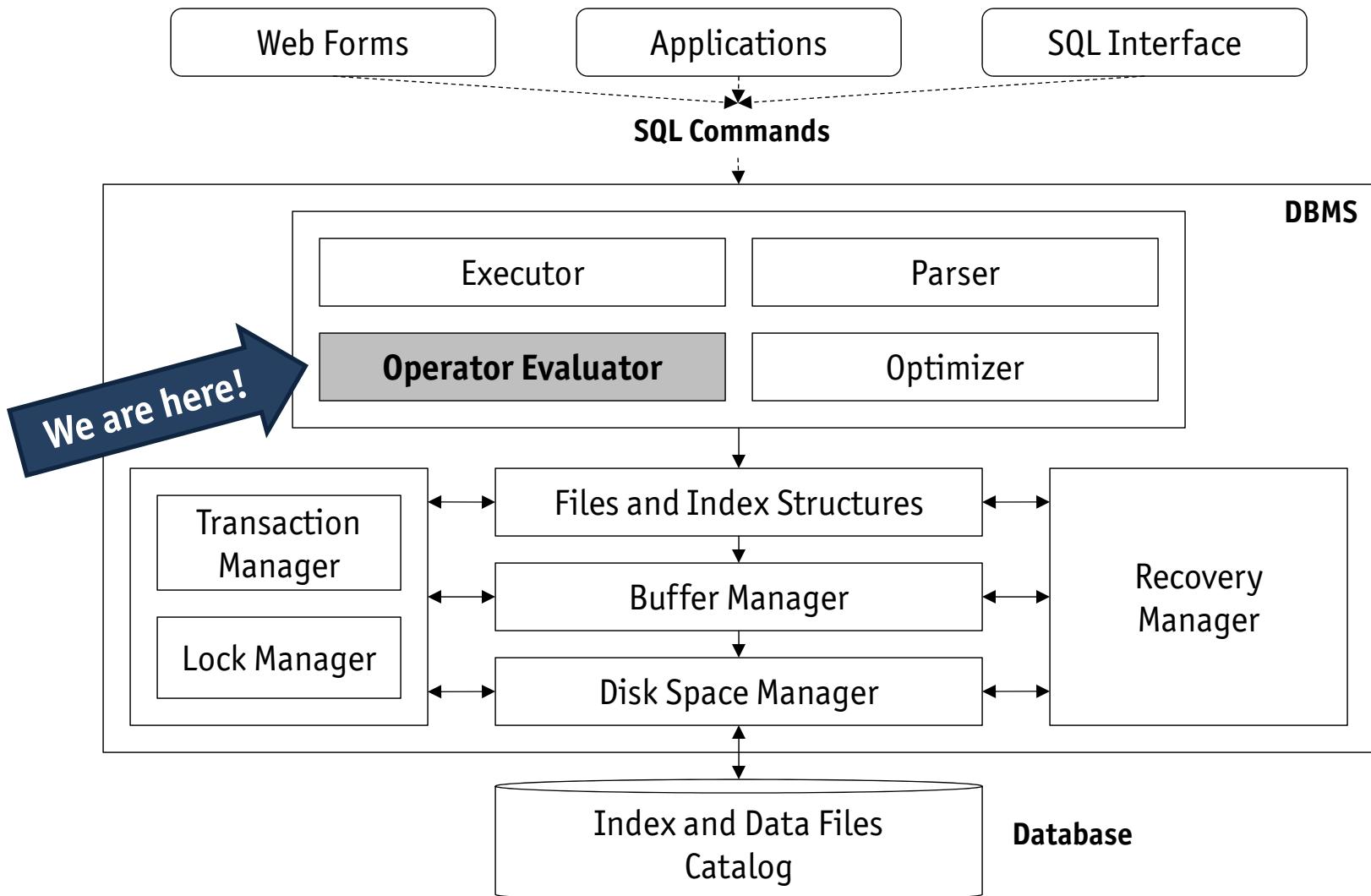
TO BE CONTINUED...



Database System Architecture and Implementation

Module 7
Evaluating Relational Operators
December 16, 2013

Orientation



Module Overview

- Unary operators
 - selection
 - projection
- Binary operators
 - join
 - set operations
- Extended relational algebra operators
 - grouping
 - aggregation
- Impact of buffering

Running Example

💻 A simple schema

```
CREATE TABLE Sailors (
    sid      INTEGER,
    sname    STRING,
    rating   INTEGER,
    age      REAL,
    PRIMARY KEY (sid)
)

CREATE TABLE Reserves (
    sid      INTEGER,
    bid      INTEGER,
    day     DATE,
    rname   STRING,
    PRIMARY KEY (sid, bid, day),
    FOREIGN KEY sid REFERENCES
        Sailors(sid)
)
```

- Assumptions
 - relation **Sailors**: 50 bytes/tuple, 80 tuples/page, and 500 pages
 - relation **Reserves**: 40 bytes/tuple, 100 tuples/page, and 1000 pages
- ↳ 4 kB page size, 2 MB **Sailors** data, and 4 MB **Reserves** data

Operator Variants

- Operator variants can be implemented using specific algorithms that are tailored to exploit **physical properties** of the system
 - **presence or absence of indexes** on the input file(s)
 - **sortedness** of the input file(s)
 - **size** of the input file(s)
 - available **buffer space in the buffer pool** (as with external sorting)
 - **buffer replacement policy**, ...

Logical and physical operators

Such a specific operator variant is referred to as a **physical operator**. In general, physical operators implement the **logical operators** of the relational algebra.

During query processing, the **query optimizer** replaces logical operators by physical operators based on its knowledge of system internals, statistics, and ongoing book-keeping. It selects the best (or most reasonable) variant for each operator.

Selection

Simple selection query

```
SELECT *
FROM   Reserves R
WHERE  R.name = 'Joe'
```

$\sigma_{\text{name}=\text{'Joe'}}$ (Reserves)

- Selection can be implemented using **iteration** and **indexing**
- Choice of physical selection operator depends on
 - sortedness of the input file
 - presence or absence of indexes on the input file
- The following cases can be distinguished
 - no index, unsorted data
 - no index, sorted data
 - B+ tree index
 - hash index, equality selection

Selection: No Index, Unsorted Data

Selection

```
function  $\sigma$  ( $p, R_{in}, R_{out}$ )
     $in \leftarrow \text{openScan}(R_{in})$  ;
     $out \leftarrow \text{createFile}(R_{out})$  ;
    while ( $r \leftarrow \text{nextRecord}(in)$ )  $\neq \langle EOF \rangle$  do
        if  $p(r)$  then  $\text{appendRecord}(out, r)$  ;
    closeFile( $out$ ) ;
end
```

- Selection (σ_p) reads an input file R_{in} of records and writes those records that satisfy predicate p into the output file R_{out}
- Remarks
 - special “record” $\langle EOF \rangle$ indicates the end of the input file
 - simple algorithm **does not require** input file to have special physical properties (algorithm is defined exclusively in terms of heap files)
 - in particular, predicate p may be **arbitrary**

Selection: No Index, Unsorted Data

↷ Cost of $\sigma_p^{scan}(R_{in})$ using a sequential scan

access path	file scan (<code>openScan</code>) of R_{in}
prerequisites	none (p arbitrary, R_{in} may be a heap file)
I/O cost	$\underbrace{\ R_{in}\ }_{\text{input cost}} + \underbrace{sel(p) \cdot \ R_{in}\ }_{\text{output cost}}$

- Notation
 - $\|R_{in}\|$ denotes the **number of pages** in file R_{in}
 - $|R_{in}|$ denotes the **number of records** in file R_{in}
 - if b records fit on one page, we have $\|R_{in}\| = [|R_{in}|/b]$
 - $sel(p)$ denotes the **selectivity** (or **reduction factor**) of predicate p
- Note that in a fully **pipelined** plan, selection can be applied on-the-fly without incurring any cost!

Selectivity

Definition

The **selectivity** (or **reduction factor**) or a predicate p , denoted by $sel(p)$, is the fraction of records in a relation R that satisfy the predicate p .

$$0 \leq sel(p) = \frac{|\sigma_p(R)|}{|R|} \leq 1$$

Examples

What can you say about the following selectivities?

1. $sel(true)$
2. $sel(false)$
3. $sel(A = 0)$

Selection: No Index, Sorted Data

- If input file R_{in} is **sorted** with respect to a sort key k
 - **binary search** could be used to find the first record
 - to find more hits, scan the **sorted** file
- Sort key k must **match** selection predicate p

When does a predicate match a (sort) key?

Assume R_{in} is sorted (or indexed) on attribute A in ascending order. Which of the selections below can benefit from the sortedness of (or index on) R_{in} ?

1. $\sigma_{A=27}(R_{in})$
2. $\sigma_{A>27}(R_{in})$
3. $\sigma_{A>27 \wedge A<100}(R_{in})$
4. $\sigma_{A>27 \vee A>100}(R_{in})$
5. $\sigma_{A>39 \wedge A<27}(R_{in})$
6. $\sigma_{A>27 \wedge B=10}(R_{in})$
7. $\sigma_{A>27 \vee B=10}(R_{in})$

Selection: No Index, Sorted Data

R Cost of $\sigma_p(R_{in})$ using binary search

access path	binary search, then sorted file scan of R_{in}
prerequisites	R_{in} sorted on sort key k that matches p
I/O cost	$\underbrace{\log_2 \ R_{in}\ }_{\text{input cost}} + \underbrace{sel(p) \cdot \ R_{in}\ }_{\text{sorted scan}} + \underbrace{sel(p) \cdot \ R_{in}\ }_{\text{output cost}}$

- Remarks
 - **disjunctive predicates** (e.g., $A < 27 \vee A > 100$) are examined later
 - even if input is sorted, binary search **cannot be used** without incurring I/O cost in a fully pipelined plan!

The Real World

It is unlikely that a relation will be kept sorted if the DBMS supports variant ① for index entries. Most systems therefore do not implement binary search for value lookups.

Selection: B+ Tree Index

- The cost of using a B+ tree index on R_{in} whose sort key **matches** the selection predicate p to evaluate $\sigma_p(R_{in})$ depends on
 - number of qualifying tuples
 - whether the index is clustered or unclustered

☞ Implementing $\sigma_p(R_{in})$ using a B+ tree

- Descend the B+ tree to retrieve the first index entry that satisfies p
- If the index is **clustered**, access that record on its page in R_{in} and continue to scan inside R_{in}
- If the index is **unclustered** and $sel(p)$ indicates a **large number of qualifying records**, it pays off to
 1. read all matching index entries $k^* = \langle k, rid \rangle$ in the sequence set
 2. sort those entries on the rid field
 3. access the pages of R_{in} in sorted rid order

☞ **Note** that the lack of clustering is a minor issue if $sel(p)$ is close to 0

Selection: B+ Tree Index

↷ Cost of $\sigma_p(R_{in})$ using a clustered B+ tree index

access path	access of B+ tree on R_{in} , then sequence set scan
prerequisites	clustered B+ tree on R_{in} with key k that matches p
I/O cost	$\approx \underbrace{3}_{B+ \text{tree access}} + \underbrace{sel(p) \cdot \ R_{in}\ }_{\text{sorted scan}} + \underbrace{sel(p) \cdot \ R_{in}\ }_{\text{output cost}}$

IBM DB2

IBM DB2 uses the physical operator quadruple **IXSCAN/SORT/RIDSCN/FETCH** to implement $\sigma_p(R_{in})$ using a B+ tree index.

Selection: Hash Index, Equality Selection

- A selection predicate p **matches a hash index** on R_{in} . A only if it contains a term of the form $A = c$, where c is a constant
- Application of hash function $h(c)$ returns the address of the bucket containing qualifying records
 - additional cost may incur due to presence of **overflow chains**
 - **selectivity** $sel(p)$ is likely to be close to 0 for equality predicates

Cost of $\sigma_p(R_{in})$ using a hash index

access path	hash index on R_{in}
prerequisites	R_{in} hashed on key k , p has a term $k = c$
I/O cost	$\underbrace{\approx 1.2}_{B+ tree\ access} + \underbrace{sel(p) \cdot \ R_{in}\ }_{output\ cost}$

General Selection Conditions

- Selection operations with simple predicates like $\sigma_{A \theta c}(R_{in})$ are only a special case
 - in practice, selection operators need to support **complex predicates**
 - complex predicates use **Boolean connectives** \wedge (**AND**) and \vee (**OR**) to combine **simple comparisons** of the form $A \theta c$, where $\theta \in \{=, <, >, \leq, \geq\}$
- So far, the **conjunctive normal form (CNF)** has been defined as a number of **conjuncts** of the form $A \theta c$, connected by \wedge (**AND**)

$$\underbrace{A_1 \theta_1 c_1 \wedge A_2 \theta_2 c_2 \wedge \dots \wedge A_n \theta_n c_n}_{conjunction}$$

- Recall that a conjunctive predicate p **matches** a (multi-attribute)
 - hash index, if p **covers** the key
 - B+ tree index, if p is a **prefix** of the key

General Selection Conditions

- In general, each conjunct consists of one or more **terms** of the form $A \theta c$ that are connected by \vee (**OR**)
- Conjuncts that contain \vee (**OR**) are said to be **disjunctive** or to **contain disjunction**

Exercise: converting selection predicates

The selection predicate

$$(day < 12/15/2013 \wedge rname = 'Joe') \vee bid = 5 \vee sid = 3$$

is **not** in conjunctive normal form. Actually, it is in **disjunctive normal form (DNF)**. Can you convert it from DNF to CNF?

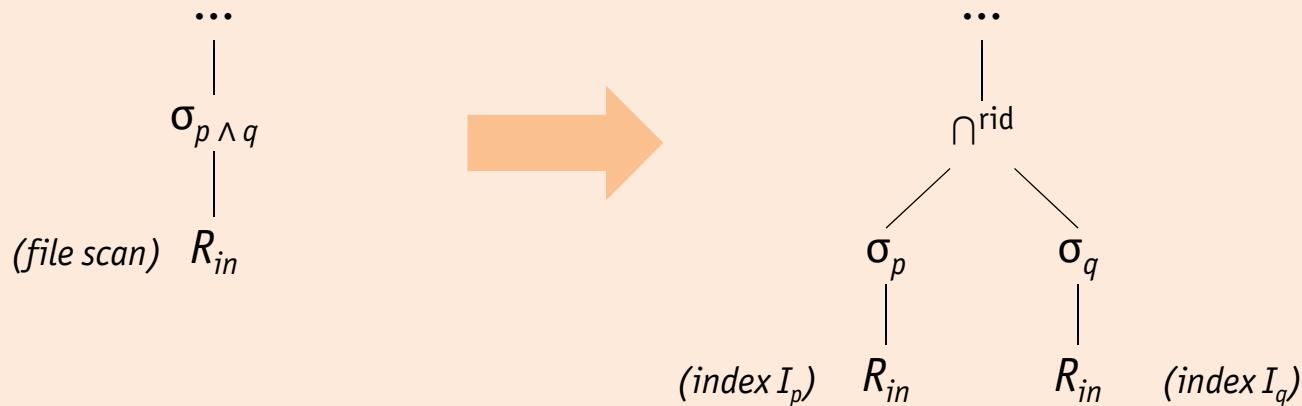
Selections without Disjunction

- If the selection predicate is a **conjunction of terms**, i.e., does not contain disjunction, there are three evaluation options
 - **single file scan**
 - **single index** that matches a subset of (primary) conjuncts and apply all non-primary conjuncts to each retrieved tuple (on-the-fly)
 - **multiple indexes** that each match a subset of conjuncts
- Recall that we have already discussed the first two options and will now focus on the third

Selection without Disjunction

↷ Intersecting *rid* sets

The conjunctive predicate in $\sigma_{p \wedge q}(R_{in})$ does **not** match a single index, but both conjuncts p and q match indexes I_p and I_q , respectively. Assume that I_p and I_q contain index entries using variant ② or ③. A typical query optimizer might decide to transform the query as follows.



Here, \cap^{rid} denotes a **set intersection operator defined by *rid* equality**.

Selection without Disjunction

The Real World

↳ IBM DB2

- physical operator **IXAND** does intersection of *rids* sets from indexes
- implemented using Bloom filters

↳ Oracle

- uses several techniques to do *rid* set intersection
- bitwise and of bitmaps
- hash join of indexes

↳ Microsoft SQL Server

- implements *rid* set intersection through index joins

Selections with Disjunction

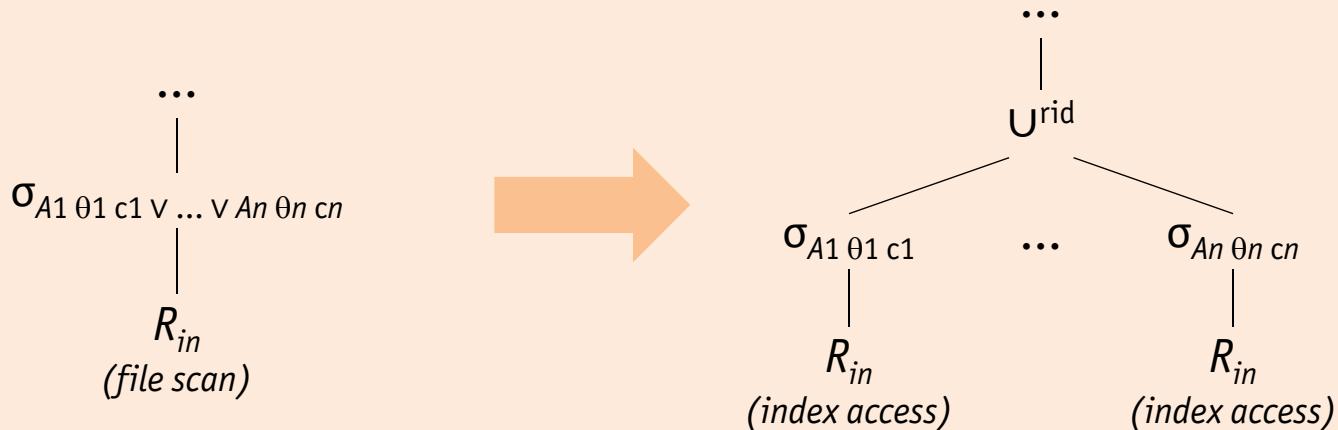
- Choosing a reasonable execution plan for **disjunctive selection predicates** of the following general form is **much** harder

$$A_1 \theta_1 c_1 \vee A_2 \theta_2 c_2 \vee \dots \vee A_n \theta_n c_n$$

- if only **one single** term does **not** match any index, evaluation is forced to use a naïve file scan
- if **all** terms are supported by indexes, **rid-based set union** \cup^{rid} can be exploited

Selections with Disjunction

Unioning *rid* sets



Exercise: selectivity of disjunctive predicates

What can you say about the selectivity of the disjunctive predicate $p \vee q$, $\text{sel}(p \vee q)$?

Bypass Selections

- Parts of a selection predicate may be **expensive** to check or be **very unselective**
 - so far, we assumed that checking a predicate was cheap (low CPU cost)
 - it is a good strategy to evaluate cheap and selective predicates first
- Boolean laws used for this optimization include

$$\begin{aligned}\text{true} \vee P &\equiv \text{true} && (\text{evaluating } P \text{ is not necessary}) \\ \text{false} \vee P &\equiv P && (\text{only evaluate } P \text{ now})\end{aligned}$$

Example

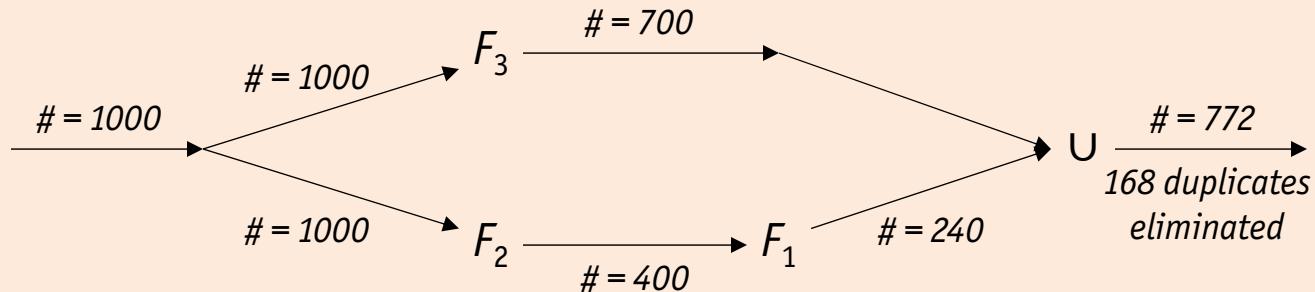
$Q := \sigma_{(F_1 \wedge F_2) \vee F_3}(R)$, where the selectivities and cost of each part of the selection condition are given in the table on the right.

formula	selectivity	cost
F_1	$s_1 = 0.6$	$C_1 = 18$
F_2	$s_2 = 0.4$	$C_2 = 3$
F_3	$s_3 = 0.7$	$C_3 = 40$

Bypass Selections

- First evaluation alternative
 - convert selection condition to **disjunctive normal form** (example is already in DNF)
 - push each tuple from input through each disjunct **in parallel**
 - **collect** matching tuples from each disjunct (duplicate elimination)

Example

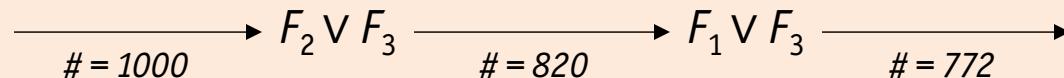


Mean cost per tuple (ignoring duplicate elimination): $C_3 + C_2 + s_2 \cdot C_1 = 50.2$

Bypass Selections

- Second evaluation alternative
 - convert selection condition to **conjunctive normal form**
$$\text{CNF}[(F_1 \wedge F_2) \vee F_3] = (F_1 \vee F_3) \wedge (F_2 \vee F_3)$$
 - push each tuple from input through each conjunct **sequentially**
 - matching tuples “survive” conjuncts (**no** duplicate elimination)

Example



Mean cost per tuple: $\begin{aligned} & C_2 + (1 - s_2) \cdot C_3 \\ & + (s_2 + (1 - s_2) \cdot s_3) \cdot C_1 + (1 - s_1) \cdot (s_2 + (1 - s_2) \cdot s_3) \cdot C_3 = 54.88 \end{aligned}$

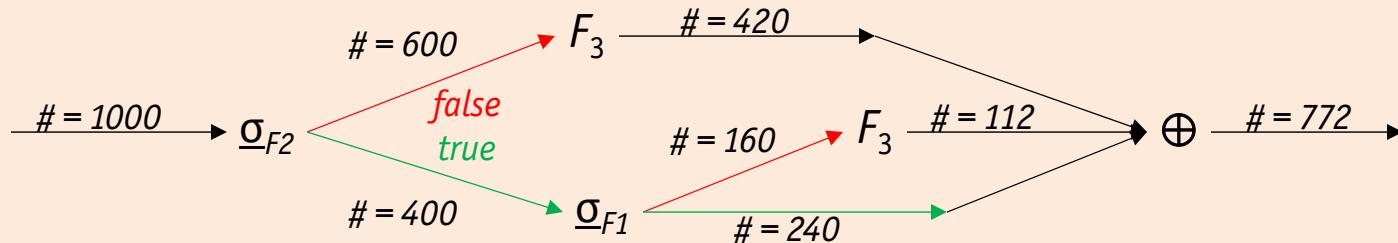
Problem: F_3 is evaluated multiple times, result could be cached

Mean cost per tuple with caching: $C_2 + C_3 + s_2 \cdot (1 - s_3) \cdot C_1 = 45.16$

Bypass Selections

- **Goal:** eliminate tuples early, avoid duplicate
- Bypass selection operator $\underline{\sigma}_F$
 - produces two disjoint outputs: true and false results
 - bypass plans are derived from the conjunctive normal form
 - Boolean factors and disjuncts in factors are sorted by cost

Example



Mean cost per tuple (\oplus disjoint union): $C_2 + (1 - s_2) \cdot C_3 + s_2 \cdot (C_1 + (1 - s_1) \cdot C_3) = 40.6$

Note that many variations are possible, e.g., for tuning in parallel environments

Selection with Disjunction

The Real World

Most systems do not handle selection conditions with disjunction efficiently and concentrate on optimizing selections without disjunction.

⇒ Oracle

- convert the query into a **union query without OR**
- if the conditions involve the same attribute (e.g., $sal < 5 \vee sal > 30$), use a **nested query with an IN list** and an index on the attribute to retrieve tuples matching a value in the list
- use **bitmap operations**, e.g., evaluate $sal < 5 \vee sal > 30$ by generating bitmaps for the values 5 and 30 and then bitwise or the bitmaps to find tuples that satisfy one of the conditions
- simply apply the disjunctive condition as a **filter** on the set of retrieved tuples

⇒ Microsoft SQL Server

- considers the use of **union queries** and **bitmaps** to deal with disjunctive conditions

Projection

Simple projection query

```
SELECT DISTINCT R.sid, R.bid  
FROM Reserves R }  $\pi_{\{sid,bid\}}(\text{Reserves})$ 
```

- **Projection** (π_l) modifies each record in its input file
 1. **removes** unwanted attributes, i.e., those not specified in attribute list l
 2. **eliminate** any duplicate tuples produced (SQL keyword **DISTINCT**)
- In general, the size of the resulting file will therefore only be a fraction of the original input file size
- Projection is implemented using **iteration** and **partitioning**
- Projection
 - **without** duplicate elimination can be fully **pipelined**
 - **with** duplicate elimination has to **materialize** intermediate results

Projection

Example

$$\pi_{\{sid, bid\}}$$

sid	bid	day	rname
1	1	07/04/2013	George
1	2	07/05/2013	Thomas
1	1	08/01/2013	George
1	2	08/02/2013	Thomas
2	1	07/05/2013	Joe
2	1	08/02/2013	Joe

= ①

sid	bid
1	1
1	2
1	1
1	2
2	1
2	1

= ②

sid	bid
1	1
1	2
2	1

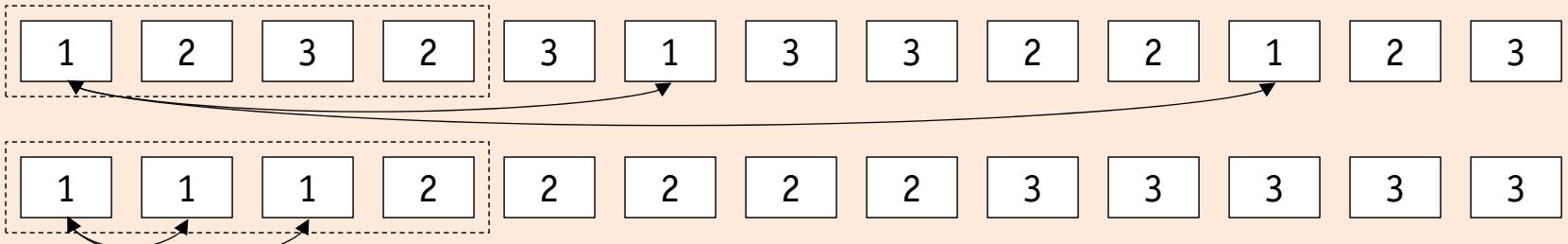
- While **step ①** calls for a rather straightforward file scan (cannot really use indexes), it is **step ②** which makes projection costly
- Partitioning is used to implement duplicate elimination
 - sorting
 - hashing

Partitioning

- Duplicate elimination
 - compare each tuple to **all other tuples** to check whether it is a duplicate
 - in general, it is **not possible** to fit all other tuples in memory
- Partitioning
 - given B buffer pages, group “similar” tuples into partitions
 - load partitions into memory one after another
 - only compare a tuple to other tuples in that partition

Illustration

Available buffer pages



Projection Based on Sorting

- Records with all fields equal will be adjacent after sorting
 1. scan R_{in} and output set of tuples that contain **only** the desired attributes
 2. sort this set of tuples using **all** of its attributes as sort key
 3. scan the sorted result, comparing adjacent tuples, and discard duplicates
- A benefit of sort-based projection is that the π_l^{sort} operator outputs a sorted result

$$R_{in} \xrightarrow{?} \pi_l^{sort} \xrightarrow{s}$$

Exercise: sort ordering

What would be the correct sort ordering θ to apply in the case of duplicate elimination?

Projection Based on Sorting

Projection based on sorting

```
function  $\pi^{\text{sort}}$  ( $l, R_{in}, R_{out}$ )
   $in \leftarrow \text{openScan}(R_{in})$  ;
   $out \leftarrow \text{createFile}(R_{tmp})$  ;
  while ( $r \leftarrow \text{nextRecord}(in)$ )  $\neq \langle EOF \rangle$  do
     $r' \leftarrow r$  with any field not listed in  $l$  cut off;
    appendRecord( $out, r'$ ) ;
  closeFile( $out$ ) ;
  external-merge-sort( $R_{tmp}, \|R_{tmp}\|, \theta$ ) ;
   $in \leftarrow \text{openScan}("run_*_0")$  ;
   $out \leftarrow \text{createFile}(R_{out})$  ;
   $lastr \leftarrow \langle \rangle$  ;
  while ( $r \leftarrow \text{nextRecord}(in)$ )  $\neq \langle EOF \rangle$  do
    if  $r \neq lastr$  then
      appendRecord( $out, r$ ) ;
       $lastr \leftarrow r$  ;
  closeFile( $out$ ) ;
end
```

Projection Based on Sorting

- Approach can be improved by modifying the sorting algorithm to do projection with duplicate elimination

Marriage of sorting and projection with duplicate elimination

Step ① (**projection**) and Step ② (**duplicate elimination**) can be **integrated** into the passes performed by the external merge sort algorithm.

Pass 0

1. read B pages at a time, **projecting unwanted attributes out**
2. use in-memory sort to sort the records of these B pages
3. write a sorted run of B internally sorted pages out to disk

Pass 1, ..., n

1. select $B - 1$ runs from previous pass, read a page from each run
2. perform a $(B - 1)$ -way merge, **eliminating duplicates**
3. use the B -th page as an output buffer

Projection Based on Sorting

Cost of $\pi_l^{sort}(R_{in})$ using sorting for duplicate elimination

access path	file scan (<code>openScan</code>) of R_{in}
prerequisites	none (B available buffer pages)
I/O cost	$\underbrace{\ R_{in}\ + \ R_{tmp}\ }_{projection} + \underbrace{2 \cdot \ R_{tmp}\ \cdot \left(\lceil \log_{B-1} \lceil \frac{\ R_{tmp}\ }{B} \rceil \rceil \right)}_{duplicate\ elimination}$

- Remarks
 - depending on the fraction of duplicate tuples, $\|R_{tmp}\|$ will be smaller in the duplicate elimination phase

Projection Based on Hashing

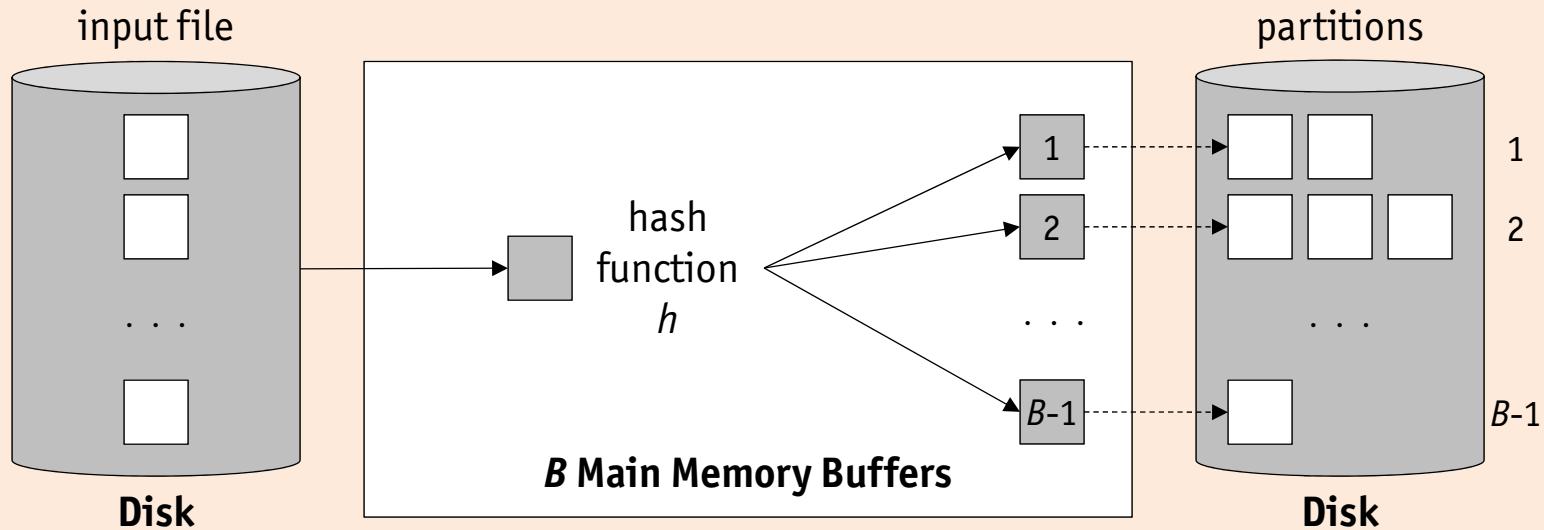
- Hash-based projection $\pi_l^{hash}(R_{in})$
 - worth considering if there is a fairly large number of buffer pages (say, B) relative to the total number of pages of R_{in}
 - two phases: **partitioning** and **duplicate elimination**

Partitioning phase

1. Allocate all B buffer pages: one page will be the **input buffer**, the remaining $B - 1$ pages will be used as **hash buckets**.
2. Read the file R_{in} page by page: for each record r , **project out** the attributes not listed in list l .
3. For each such record, apply hash function $h_1(r) = h(r) \text{ mod } (B - 1)$, which depends on **all remaining fields of r** , and store r in hash bucket $h_1(r)$.
4. If the hash bucket is **full**, write it to disk, i.e., the overflow chain of a bucket resides on disk

Projection Based on Hashing

Partitioning phase



- Two **identical** records r, r' will be mapped to the **same partition**
 - $h_1(r) = h_1(r') \Leftarrow r = r'$
 - duplicate elimination is an intra-partition problem only
 - **but**, due to hash collisions, the records in a partition are not guaranteed to be all equal, i.e., $h_1(r) = h_1(r') \nLeftrightarrow r = r'$

Projection Based on Hashing

Duplicate elimination phase

1. For each partition, read each partition page by page (possibly in parallel), using the same buffer layout as before.
2. To each record, apply a hash function h_2 ($h_2 \neq h_1$) to all record fields.
3. Only if two records collide with respect to h_2 , check if $r = r'$ and, if so, discard r' .
4. After the entire partition has been read in, append all hash buckets to the result file, which will be free of duplicates.

Exercise: large partitions

Projection based on hashing only works efficiently, if duplicate elimination can be **performed in the buffer** (main memory). What can be done if the partition size exceeds the buffer size?

Projection Based on Hashing

Cost of $\pi_l^{hash}(R_{in})$ using hashing for duplicate elimination

access path	file scan (<code>openScan</code>) of R_{in}
prerequisites	none (B available buffer pages)
I/O cost	$\underbrace{\ R_{in}\ + \ R_{tmp}\ }_{\text{projection}} + \underbrace{\ R_{tmp}\ + \ R_{tmp}\ }_{\text{duplicate elimination}}$

- Remarks
 - depending on the fraction of duplicate tuples, $\|R_{tmp}\|$ will be smaller in the duplicate elimination phase

Sorting Versus Hashing

- Sorting is the standard approach for duplicate elimination
 - superior to hashing if there are many duplicates or if the distribution of (hash) values is very non-uniform
 - sorting is required for a number of other reasons and therefore already implemented in most systems
- If there are $B > \sqrt{\|R_{tmp}\|}$ buffer pages, both approaches have the same cost
 - sorting takes two passes: $\|R_{in}\| + \|R_{tmp}\| + \|R_{tmp}\| + \|R_{tmp}\|$
 - hashing is the same: $\|R_{in}\| + \|R_{tmp}\| + \|R_{tmp}\| + \|R_{tmp}\|$
- Choice of projection algorithm therefore depends on **CPU cost**, desirability of **sorted order**, **skew** in value distribution, etc.

Using Indexes for Projection

- Index key contains **all attributes** retained in the projection
 - use an index-only plan to retrieve all values from index without accessing actual data records
 - apply hashing or sorting to eliminate duplicates from this (much smaller) set of pages
- Index key contains projected attributes as a **prefix** and is **sorted**
 - use an index only plan both to retrieve projected attribute values **and** to eliminate duplicates

The Real World

- ↳ **IBM DB2 and Oracle**
 - sorting is used for duplicate elimination
- ↳ **Microsoft SQL Server**
 - implements both hash-based and sort-based algorithms for duplicate elimination

Join

Join vs. Cartesian product



- Semantics of **join operator** \bowtie_p is equivalent to combination of **cross product** \times and **selection** σ_p
- One way to implement \bowtie_p is to use this relational equivalence
 1. enumerate all record pairs in the cross product $R_1 \times R_2$
 2. pick those record pairs that satisfy the predicate p
- More advance algorithms try to avoid the obvious inefficiency in Step 1 (the size of the intermediate result is $|R_1| \cdot |R_2|$)

Nested Loops Join

- The **nested loops join** is the straightforward implementation of the cross product (\times) and selection (σ) combination
- For obvious reasons, R_1 is referred to as the **outer** (relation), whereas R_2 is called the **inner** (relation)

💻 Nested loops join

```
function  $\bowtie^{nl}$  ( $R_1, R_2, R_{out}, p$ )
     $in_1 \leftarrow \text{openScan}(R_1)$  ;
     $out \leftarrow \text{createFile}(R_{out})$  ;
    while ( $r_1 \leftarrow \text{nextRecord}(in_1)$ )  $\neq \langle EOF \rangle$  do
         $in_2 \leftarrow \text{openScan}(R_2)$  ;
        while ( $r_2 \leftarrow \text{nextRecord}(in_2)$ )  $\neq \langle EOF \rangle$  do
            if  $p(r_1, r_2)$  then  $\text{appendRecord}(out, \langle r_1, r_2 \rangle)$  ;
        closeFile(out) ;
    end
```

Nested Loops Join

Cost of $R_1 \bowtie_p^{nl} R_2$

access path	file scan (<code>openScan</code>) of R_1 and R_2
prerequisites	none (p arbitrary, R_1 and R_2 may be heap files)
I/O cost	$\underbrace{\ R_1\ }_{\text{outer loop}} + \underbrace{ R_1 \cdot \ R_2\ }_{\text{inner loop}}$

- Remarks
 - algorithm can easily be refined such that one scan of the inner relation is initiated for each **page** of the outer relation (instead of each **record**)
 - in this case, the cost of the inner loop is $\|R_1\| \cdot \|R_2\|$

Nested Loops Join

- The **good news** is that \bowtie^{nl} only needs **three pages** of buffer space (two to read R_1 and R_2 , one to write the result)
- The **bad news** is its **staggering I/O cost**, since it effectively enumerate all records in the cross product $R_1 \times R_2$

Exercise: execution time of \bowtie_p^{nl}

Assume that $\|R_1\| = 1000$ pages and $\|R_2\| = 500$ pages. Both relations store 100 tuples per page. If the access time is 10 ms per page, how long will the join $R_1 \bowtie_p^{nl} R_2$ take?

Nested Loops Join

- The **good news** is that \bowtie^{nl} only needs **three pages** of buffer space (two to read R_1 and R_2 , one to write the result)
- The **bad news** is its **staggering I/O cost**, since it effectively enumerate all records in the cross product $R_1 \times R_2$

Exercise: execution time of \bowtie_p^{nl}

Assume that $\|R_1\| = 1000$ pages and $\|R_2\| = 500$ pages. Both relations store 100 tuples per page. If the access time is 10 ms per page, how long will the join $R_1 \bowtie_p^{nl} R_2$ take?

- ↳ $1000 + (100,000 \cdot 500) \approx 140$ hours
- ↳ $1000 + (1000 \cdot 500) \approx 83$ minutes

Note that switching the roles of R_1 and R_2 to make R_2 (the smaller one) the **outer relation** does not bring a significant advantage.

Block Nested Loops Join

- The **block nested loops join** saves random access cost by reading R_1 and R_2 in blocks of, say b_1 and b_2 pages
 - R_1 is still read once, with only $\lceil \|R_1\|/b_1 \rceil$ disk seeks
 - R_2 is scanned $\lceil \|R_1\|/b_1 \rceil$ times, with $\lceil \|R_1\|/b_1 \rceil \cdot \lceil \|R_2\|/b_2 \rceil$ disk seeks

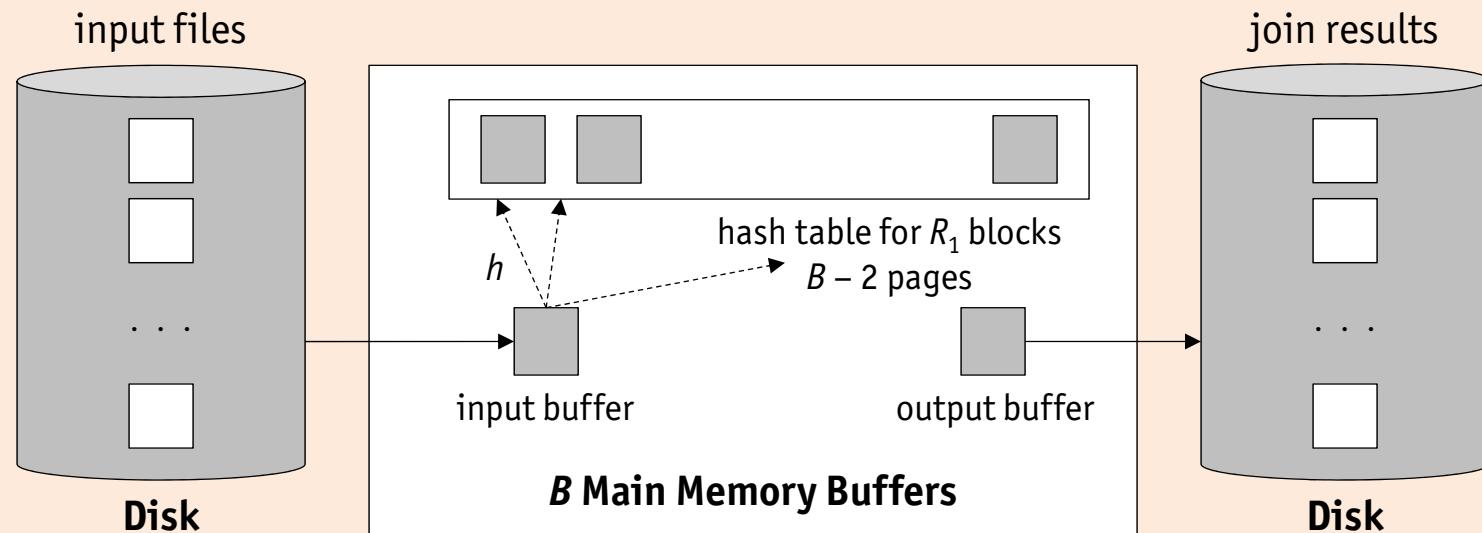
💻 General block nested loops join

```
function  $\bowtie^{block-nl}$  ( $R_1, R_2, R_{out}, p$ )
     $in_1 \leftarrow \text{openScan}(R_1)$  ;
     $out \leftarrow \text{createFile}(R_{out})$  ;
    foreach  $b_1$ -sized block in  $in_1$  do
         $in_2 \leftarrow \text{openScan}(R_2)$  ;
        foreach  $b_2$ -sized block in  $in_2$  do
            for all matching in-memory tuples  $r_1 \in R_1$  block and  $r_2 \in R_2$  blocks,
            add  $\langle r_1, r_2 \rangle$  to the result  $out$ 
        closeFile( $out$ ) ;
    end
```

Block Nested Loops Join

- Building a hash table over the R_1 block can speed up the **in-memory join** between the R_1 and R_2 blocks considerably
 - read outer relation R_1 in **blocks of $B - 2$ pages**
 - this optimization only works for **equi-joins**

Block nested loops join with hash table



Block Nested Loops Join

Block nested loops join with hash table

```
function  $\bowtie^{block-nl}$  ( $R_1, R_2, R_{out}, p$ )
     $in_1 \leftarrow \text{openScan}(R_1)$  ;
     $out \leftarrow \text{createFile}(R_{out})$  ;
    repeat
         $B' \leftarrow \min(B - 2, \# \text{remaining blocks in } R_1)$  ;           (do not read beyond <EOF> of  $R_1$ )
        if  $B' > 0$  then
            read  $B'$  blocks of  $R_1$  into buffer, hash record  $r \in R_1$  to buffer page  $h(r.A_1) \bmod B'$  ;
             $in_2 \leftarrow \text{openScan}(R_2)$  ;
            while ( $r_2 \leftarrow \text{nextRecord}(in_2)$ )  $\neq \langle EOF \rangle$  do
                compare record  $r_2$  with records  $r_1$  stored in buffer page  $h(r_2.A_2) \bmod B'$  ;
                if  $r_1.A_1 = r_2.A_2$  then  $\text{appendRecord}(out, \langle r_1, r_2 \rangle)$  ;
        until  $B' < B - 2$  ;
        closeFile( $out$ ) ;
end
```

Block Nested Loops Join

 Exercise: execution time of $\bowtie_p^{block-nl}$

As before, assume that $\|R_1\| = 1000$ pages and $\|R_2\| = 500$ pages. If the access time is 10 ms per page, how long will the join $R_1 \bowtie_p^{block-nl} R_2$ take, using $B = 100$ buffer pages?

Block Nested Loops Join

✍ Exercise: execution time of $\bowtie_p^{block-nl}$

As before, assume that $\|R_1\| = 1000$ pages and $\|R_2\| = 500$ pages. If the access time is 10 ms per page, how long will the join $R_1 \bowtie_p^{block-nl} R_2$ take, using $B = 100$ buffer pages?

$$\Rightarrow \left(1000 + \left\lceil \frac{1000}{100-2} \right\rceil \cdot 500 \right) \cdot 10 \text{ ms} = 65 \text{ sec}$$

Index Nested Loops Join

- The **index nested loops** join takes advantage of an index on the inner relation (swap *outer* \leftrightarrow *inner*, if necessary)
 - index must **match** join condition p
 - index nested loops **avoids** enumeration of the cross product

⌨ Index nested loops join

```
function  $\bowtie^{index-nl}$  ( $R_1, R_2, R_{out}, p$ )
     $in_1 \leftarrow \text{openScan}(R_1)$  ;
     $out \leftarrow \text{createFile}(R_{out})$  ;
    while ( $r_1 \leftarrow \text{nextRecord}(in_1)$ )  $\neq \langle \text{EOF} \rangle$  do
        probe index on  $R_2$  using (key value in)  $r_1$  to find matching tuples  $r_2 \in R_2$  ;
         $\text{appendRecord}(out, \langle r_1, r_2 \rangle)$  ;
    closeFile( $out$ ) ;
end
```

Index Nested Loops Join

- For each tuple in R_1 , the index is probed for matching R_2 tuples
- Breakdown of costs
 1. **access** index to find its first matching entry costs
 2. **scan** the index to retrieve **all** n matching rids (I/O cost for this step is typically negligible due to locality in the index)
 3. **fetching** the n matching R_2 tuples
- Note that the cost of an index nested loops joins **depends on the size of the join result**, due to points 2 and 3 above



Exercise: quantifying these costs

You have seen a few cost analyses so far and should be able to quantify these costs for clustered or unclustered hash and B+ tree indexes!

Index Nested Loops Join

Cost of $R_1 \bowtie_p^{index-nl} R_2$

access path	file scan (openScan) of R_1 , index access to R_2
prerequisites	index on R_2 that matches join predicate p
I/O cost	$\underbrace{\ R_1\ }_{outer\ loop} + \underbrace{ R_1 \cdot (\text{cost of } \mathbf{one} \text{ index access to } R_2)}_{inner\ loop}$

- Remarks
 - $\bowtie_p^{index-nl}$ is particularly useful if the index is a **clustered index**
 - even with **unclustered indexes** and few matches per outer tuple, $\bowtie_p^{index-nl}$ outperform simple nested loops
 - overall, the use of $\bowtie_p^{index-nl}$ pays off if the join is **selective** (picks out a few tuples only from a big table)

Index Nested Loops Join

 Exercise: execution time of $\bowtie_p^{index-nl}$

If the access time is 10 ms per page, how long will the join

$Reserves \bowtie_{sid=sid}^{block-nl} Sailors$

take, assuming there is a hash-based index on $Reserves.sid$ that uses index entries of alternative ②?

Index Nested Loops Join

Exercise: execution time of $\bowtie_p^{index-nl}$

If the access time is 10 ms per page, how long will the join

$Reserves \bowtie_{sid=sid}^{block-nl} Sailors$

take, assuming there is a hash-based index on $Reserves.sid$ that uses index entries of alternative ②?

- ↳ *Sailors* will be the outer table (500 I/O operations)
- ↳ for each of the $80 \cdot 500 = 40,000$ tuples in *Sailors*, matching *Reserves* tuples are retrieved (1.2 I/O operations per tuple)
- ↳ total $500 + 40,000 \cdot 1.2 = 48,500$ I/O Operations
- ↳ additionally, there is the cost of retrieving the *Reserves* tuples
- ↳ if we assume uniform distribution, each *Sailors* tuple matches with 2.5 *Reserves* tuple, since there are 100,000 reservations for 40,000 sailors
- ↳ clustered index: 1 I/O operation per Sailor tuple, i.e., 40,000 I/O operations
- ↳ unclustered index: 2.5 I/O operations per Sailor tuple, i.e., 100,000 I/O operations
- ↳ total cost varies from $48,500 + 40,000 = 88,500$ to $48,500 + 100,000 = 148,500$ I/O operations, or 15 to 25 minutes

The Road So Far

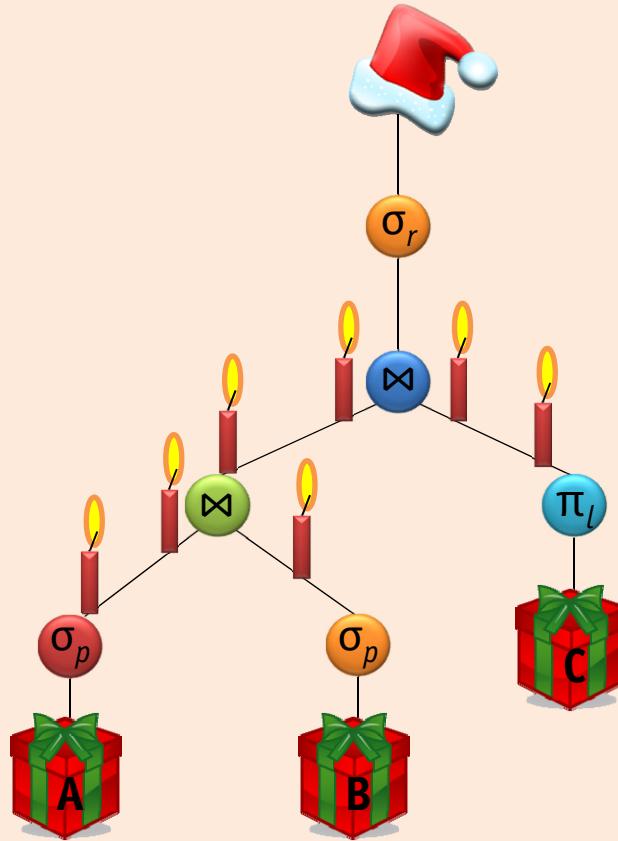
Common implementation techniques

Before we take a well-deserved vacation, let us revisit the algorithms for relational operators that we examined so far in terms of the three **common implementation techniques**.

Algorithm	Iteration	Indexing	Partitioning
$\sigma_p^{scan}(R)$			
$\sigma_p^{binsearch}(R)$			
$\sigma_p^{B+tree}(R)$			
$\sigma_p^{hash}(R)$			
$\pi_l^{sort}(R)$			
$\pi_l^{hash}(R)$			
$R_1 \bowtie_p^{nl} R_2$			
$R_1 \bowtie_p^{block-nl} R_2$			
$R_1 \bowtie_p^{index-nl} R_2$			

Happy Holidays!

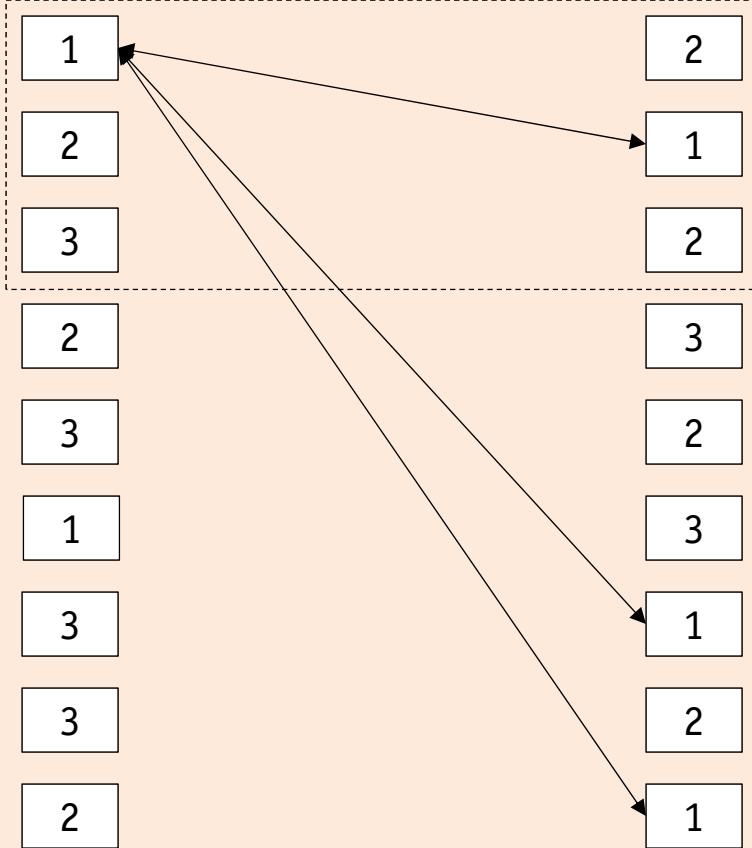
↻ Festive query tree



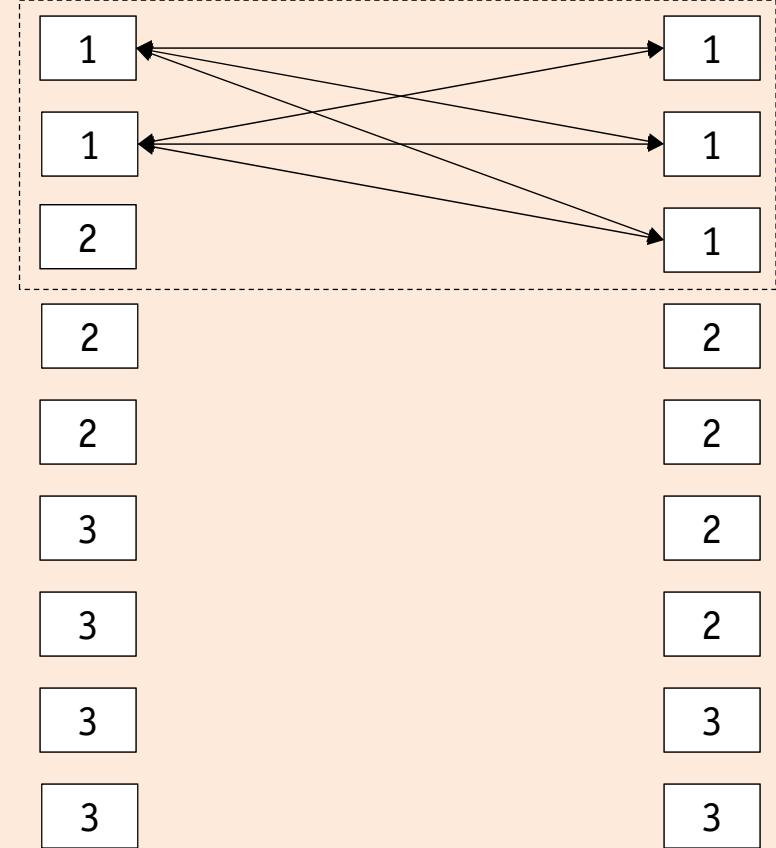
Partitioning for Joins

Illustration

Available buffer pages



Available buffer pages



Sort-Merge Join

- The **sort-merge join** uses sorting to partition both inputs
 - sort both relations on the join attribute
 - look for qualifying tuples by merging the two relations
- Sort-merge join is typically used for equi-joins only
- Merge phase similar to merge passes of external sort
 - simply consider input relations as two runs that have to be merged
 - slight adaptation required to deal duplicates on both sides

☞ Multiple matches per tuple (disrupts sequential access)

$$\left(\begin{array}{|c|c|} \hline \textbf{A} & \textbf{B} \\ \hline \text{Chuck} & 1 \\ \hline \text{Sarah} & 2 \\ \hline \text{Casey} & 2 \\ \hline \text{Morgan} & 2 \\ \hline \text{Ellie} & 4 \\ \hline \end{array} \right) \bowtie_{\text{B}=\text{C}} \left(\begin{array}{|c|c|} \hline \textbf{C} & \textbf{D} \\ \hline 1 & \text{true} \\ \hline 2 & \text{false} \\ \hline 2 & \text{true} \\ \hline 3 & \text{false} \\ \hline \end{array} \right)$$

Sort-Merge Join

Sort-merge join

```
function  $\bowtie^{sort-merge}(R_1, R_2, R_{out}, R_1.A = R_2.B)$ 
    if  $R_1$  not sorted on  $A$  then sort it;      if  $R_2$  not sorted on  $B$  then sort it;
    out  $\leftarrow$  createFile( $R_{out}$ );
    in1  $\leftarrow$  openScan( $R_1$ );
    r1  $\leftarrow$  nextRecord(in1);
    in2  $\leftarrow$  openScan( $R_2$ );
    r2  $\leftarrow$  nextRecord(in2);
    while  $r_1 \neq \langle EOF \rangle \wedge r_2 \neq \langle EOF \rangle$  do
        while  $r_1.A < r_2.B$  do  $r_1 \leftarrow$  nextRecord(in1);
        while  $r_1.A > r_2.B$  do  $r_2 \leftarrow$  nextRecord(in2);
         $r'_2 \leftarrow r_2$ ;                      (remember current position in  $R_2$ )
        while  $r_1.A = r'_2.B$  do
             $r_2 \leftarrow r'_2$ ;                  (all  $R_1$  tuples with the same  $A$  value)
            while  $r_1.A = r_2.B$  do
                appendRecord(out,  $\langle r_1, r_2 \rangle$ );
                 $r_2 \leftarrow$  nextRecord(in2);
                 $r_1 \leftarrow$  nextRecord(in1);
        closeFile(out);
    end
```

Sort-Merge Join

✍ Exercise: best case and worst case of $\bowtie^{sort-merge}$

What is the **best case** and what is the **worst case scenario** for $R_1 \bowtie_{A=B}^{sort-merge} R_2$?

⇒ **best case**

⇒ **worst case**

Sort-Merge Join

Exercise: best case and worst case of $\bowtie^{sort-merge}$

What is the **best case** and what is the **worst case scenario** for $R_1 \bowtie_{A=B}^{sort-merge} R_2$?

⇒ **best case**

if A is a key in R_1 , i.e., all values of A are different

⇒ **worst case**

if all (r_1, r_2) -pairs match, i.e., if the result is a Cartesian product because all values of A and B are constant and carry the same value

Sort-Merge Join

Cost of $R_1 \bowtie_{A=B}^{\text{sort-merge}} R_2$

access path	sorted file scan of R_1 and R_2
prerequisites	p equality predicate $R_1.A = R_2.B$
I/O cost	cost of sorting R_1 and/or R_2 , if not sorted already, plus best case: $\ R_1\ + \ R_2\ $ worst case: $\ R_1\ \cdot \ R_2\ $

- Remarks
 - best case of sort-merge join is optimal
 - sort-merge join operator can be integrated into the final sort pass of the external sort operator
 - blocked I/O, double buffering, and replacement sort can also be applied to speed up the sorting and merging of the input relations
 - output of sort-merge join is also sorted on join attribute

Sort-Merge Join

✍ Exercise: execution time of $\bowtie_p^{\text{sort-merge}}$

If the access time is 10 ms per page, how long will the join

Reserves $\bowtie_{\text{sid}=\text{sid}}^{\text{sort-merge}}$ *Sailors*

take in the best and the worst case, assuming both relations are sorted on *sid*?

↳ **best case**

↳ **worst case**

Sort-Merge Join

Exercise: execution time of $\bowtie_p^{\text{sort-merge}}$

If the access time is 10 ms per page, how long will the join

Reserves $\bowtie_{sid=sid}^{\text{sort-merge}} \text{Sailors}$

take in the best and the worst case, assuming both relations are sorted on *sid*?

↳ **best case**

$$(1000 + 500) \cdot 10 \text{ ms} = 15,000 \text{ ms} = 15 \text{ s}$$

↳ **worst case**

$$(1000 \cdot 500) \cdot 10 \text{ ms} = 5,000,000 \text{ ms} \approx 83 \text{ minutes}$$

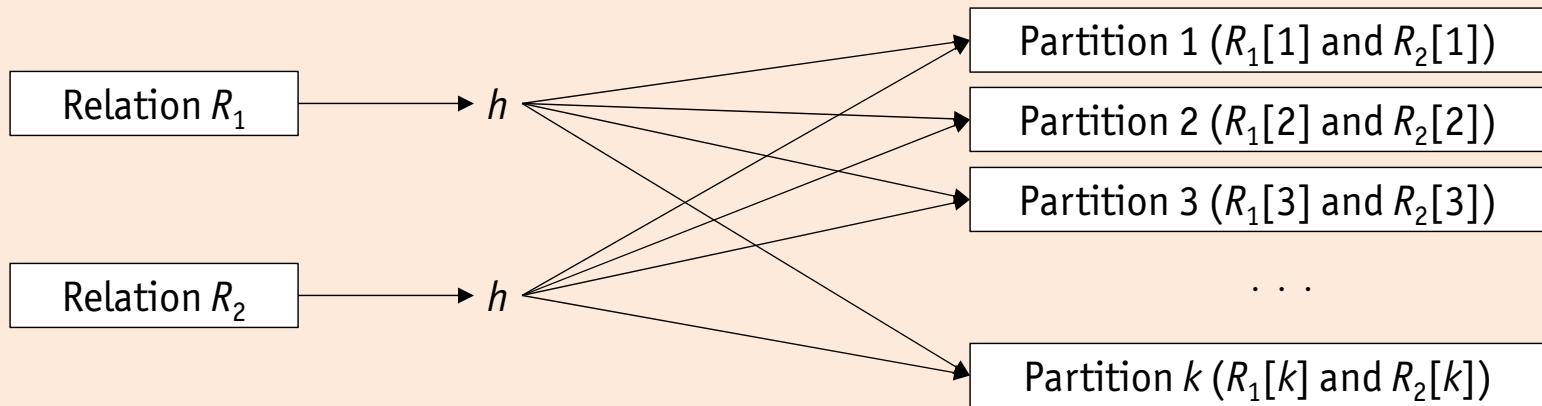
Note that the worst case I/O performance of the **sort-merge join** is (almost) the same as the I/O performance of the **nested loops join**!

Hash Joins

- Like the sort-merge join algorithm, join algorithms in the **family of hash joins** work in two phases
 - **partitioning (or building) phase** uses hashing to divide R_1 and R_2 into k partitions $R_1[k]$ and $R_2[k]$
 - **probing (or matching) phase** only compares tuples in partition $R_1[i]$ to tuples in the corresponding partition $R_2[i]$
- Algorithm follows **divide and conquer** principle
 - instead of one big disk-based join, do many small in-memory joins
 - observe that $R_1[i] \bowtie R_2[j] = \emptyset$ for all $i \neq j$
- Examples
 - Grace hash join (named after GRACE database management system)
 - hybrid hash join
 - ... there are many more!

Grace Hash Join

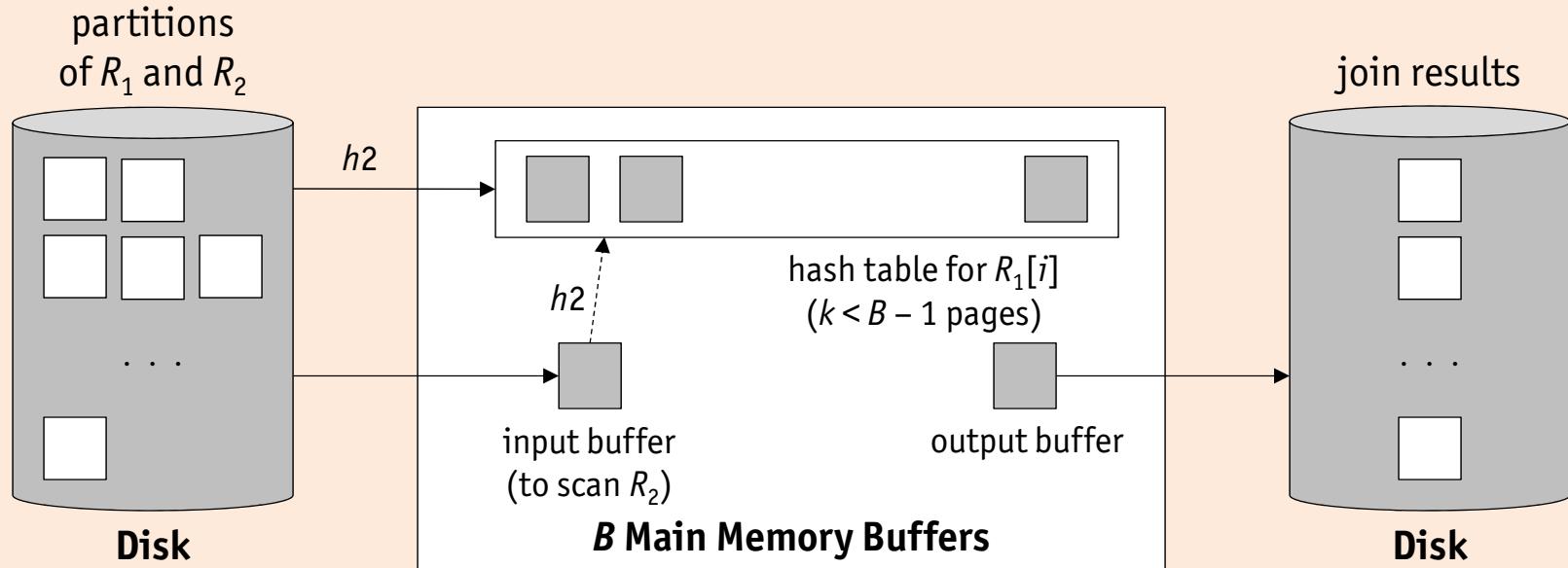
Partitioning phase of hash join



- Partitioning phase scans each input relation in turn, applies hash function h , and writes out k hash buckets
- Matching tuples are guaranteed to end up together in same partition (works for **equality predicates** only)

Grace Hash Join

↷ Probing phase of a hash join



- Probing phase scans each of the k buckets one, and computes an intra-partition **in-memory join** (hopefully)
- As with the block nested loops join, the in-memory join accelerated using a hash table with hash function $h2$

Grace Hash Join

Grace hash join

```
function  $\bowtie^{hash-join}$  ( $R_1, R_2, R_{out}, R_1.A = R_2.B$ )
   $in_1 \leftarrow \text{openScan}(R_1)$  ; (partitioning phase)
  while ( $r_1 \leftarrow \text{nextRecord}(in_1)$ )  $\neq \langle EOF \rangle$  do
    add  $r_1$  to partition  $R_1[h(r_1.A)]$  ;
    add  $r_1$  to partition  $R_1[h(r_1.A)]$  ; (flushed as page fills)
   $in_2 \leftarrow \text{openScan}(R_2)$  ;
  while ( $r_2 \leftarrow \text{nextRecord}(in_2)$ )  $\neq \langle EOF \rangle$  do
    add  $r_2$  to partition  $R_2[h(r_2.B)]$  ;
    add  $r_2$  to partition  $R_2[h(r_2.B)]$  ; (flushed as page fills)
   $out \leftarrow \text{createFile}(R_{out})$  ;
  foreach  $i \in 1, \dots, k$  do (probing phase)
    foreach tuple  $r_1 \in R_1[i]$  do (build in-memory hash table for  $R_1[i]$ , using  $h_2$ )
      insert  $r_1$  into hash table  $H$ , using  $h_2(r_1.A)$  ;
    foreach tuple  $r_2 \in R_2[i]$  do (scan  $R_2[i]$  and probe for matching  $R_1[i]$  tuples)
      probe  $H$  using  $h_2(r_2.B)$  and append matching tuples  $\langle r_1, r_2 \rangle$  to  $out$  ;
      clear  $H$  to prepare for next partition ;
    closeFile( $out$ ) ;
  end
```

Grace Hash Join

Cost of $R_1 \bowtie_{A=B}^{\text{hash-join}} R_2$

access path	file scan (<code>openScan</code>) of R_1 and R_2		
prerequisites	equi-join, i.e., p equality predicate $R_1.A = R_2.B$		
I/O cost	$\underbrace{\ R_1\ + \ R_2\ }_{\text{read}} + \underbrace{\ R_1\ + \ R_2\ }_{\text{write}} + \underbrace{\ R_1\ + \ R_2\ }_{\text{probing phase}} = 3 \cdot (\ R_1\ + \ R_2\)$		
	<i>partitioning phase</i>		

- Remarks
 - the partitioning phase reads each page of R_1 and R_2 exactly **once** and writes **about the same** amount of pages out for the partitions
 - the probing phase reads each partition **once**
 - note that this cost estimation ignores **memory bottlenecks**

Grace Hash Join

 **Exercise:** execution time of $\bowtie_p^{\text{hash-join}}$

If the access time is 10 ms per page, how long will the join

Reserves $\bowtie_{sid=sid}^{\text{hash-join}} \text{Sailors}$

take, assuming there are no memory bottlenecks?

Grace Hash Join

 **Exercise:** execution time of $\bowtie_p^{\text{hash-join}}$

If the access time is 10 ms per page, how long will the join

Reserves $\bowtie_{sid=sid}^{\text{hash-join}} \text{Sailors}$

take, assuming there are no memory bottlenecks?

$$\Rightarrow 3 \cdot (1000 + 500) \cdot 10 \text{ ms} = 45,000 \text{ ms} = 45 \text{ s}$$

Grace Hash Join

- For the probing phase, partitions should fit into memory
 - to minimize partition size, number of partitions has to be maximized
 - with B buffers, $B - 1$ partitions can be created in the partitioning phase
- Memory requirements for Grace hash join
 - assuming equal distribution, each R partition has size $\frac{\|R\|}{B-1}$
 - size of (in-memory) hash table built during the probing phase for an R partition is $\frac{f \cdot \|R\|}{B-1}$, where $f > 1$ is a **fudge factor** that captures the (small) increase in size between the partition and a hash table for the partition
 - probing phase needs to keep one such in-memory hash table plus an input and an output buffer in memory, i.e., $B > \frac{f \cdot \|R\|}{B-1} + 2$
 - algorithm needs **approximately** $B > \sqrt{f \cdot \|R\|}$ buffers to perform well
- Some cases require multiple passes (recursive partitioning)
 - if input table R has more than $(B - 1)^2$ pages
 - if values of R are not uniformly distributed over the partitions

Hybrid Hash Join

- If more than $B > \sqrt{f \cdot \|R\|}$ memory is available, **hybrid hash join** achieves better performance
- Suppose that $B > f \cdot (\|R\|/k)$, for some integer k
 - if R is divided into k partitions of size $\|R\|/k$, an in-memory hash table can be created for each partition
 - k output buffers and one input buffer are needed to partition R (and S), which leaves $B - (k + 1)$ extra buffer pages
- Suppose $B - (k + 1) > f \cdot (\|R\|/k)$, i.e., there is enough space to hold an in-memory hash table for a partition of R
 - **partitioning of R** : build hash table for first partition of R
 - **partitioning of S** : probe hash table with tuples of first partition of S
 - after the partitioning phase, first partitions of R and S are already joined
- Saves I/O cost for writing and reading first partitions of R and S

Hybrid Hash Join

R Example: cost of $\bowtie_p^{\text{hybrid-hash-join}}$

Consider the example with 500 pages in *Reserves*, 1000 pages in *Sailors*, and 300 buffer pages.

We can split *Reserves* (and *Sailors*) into 2 partitions and build an in-memory hash table for first partition of *Reserves*

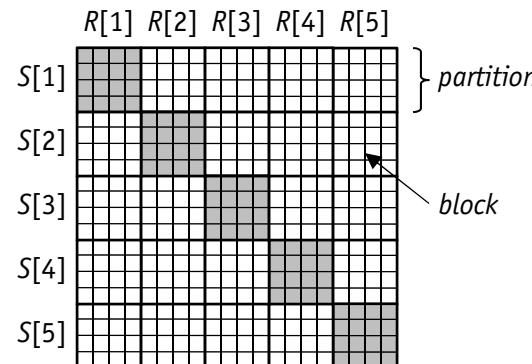
- **partitioning** of *Reserves*: scanning table and writing one partition ($500 + 250$ pages)
- **partitioning** of *Sailors*: scanning table and writing one partition ($1000 + 500$ pages)
- **probing phase**: read second partition of *Reserves* and *Sailors* ($250 + 500$ pages)
- **total cost**: $750 + 1500 + 750 = \underline{3000}$ page I/O operations

Recall that the cost of the Grace hash join was $3 \cdot (1000 + 500) = \underline{4500}$ page I/O operations, but it only requires $\lceil \sqrt{f \cdot 500} \rceil \approx 25$ buffer pages.

Note that if there are $B > f \cdot \|R\| + 2$ buffer pages available, i.e., the hash table for **all of R** fits into memory, the cost of the hybrid hash join is $500 + 1000 = \underline{1500}$ page I/O operations.

Hash Join vs. Block Nested Loops Join

- Recall the variant of block nested loops joins that builds an in-memory hash table for the inner relation
- If a hash table for the entire smaller relation fits into memory, block nested loops join and hash join are the same
- If both relations are large relative to the available buffer size
 - block nested loops join needs several passes over one of the relations
 - hash join is a more effective application of hashing in this case



Hash Join vs. Sort-Merge Join

- $3 \cdot (N + M)$ page I/O operations
 - cost of **hash join**, if there are $B > \sqrt{\|M\|}$ buffer pages, where M is the **smaller** relation and we assume uniform partitioning
 - cost of **sort-merge join**, if there are $B > \sqrt{\|N\|}$ buffer pages, where N is the **larger** relation
- Choosing between hash join and sort-merge join
 - sort-merge join is less sensitive to **skew**, which would lead to partitions in hash join that are not uniformly sized
 - hash join costs less if the number of available buffer pages falls between $\sqrt{\|M\|}$ and $\sqrt{\|N\|}$ (the larger the difference in size between the two relations, the more important this factor becomes)
 - sort-merge join produces a **sorted result**

General Join Conditions

- Equalities over a combination of several attributes
 - **index nested loops join** can build a new index on combination of all attributes or use an existing index on combination of all attribute as well as of a subset of the attributes
 - **hash join** and **sort-merge join** partition over combination of attributes
- Inequalities
 - **index nested loops join** requires a B+ tree index
 - **hash join** and **sort-merge join** are not applicable
- Other join algorithms essentially remain unaffected

Summary of Join Algorithms

- No single join algorithm performs best under all circumstances
- Choice of algorithm affected by
 - size of relations joined
 - size of available buffer space
 - availability of indexes
 - form of join condition
 - selectivity of join predicate
 - available physical properties of inputs (e.g., sort order)
 - desirable physical properties of outputs (e.g., sort order)
- Performance differences between “good” and “bad” algorithm for any given join can be enormous
- Join algorithms have been subject to intensive research efforts

A (Foo) Bar Joke

A SQL statement walks into a bar.
He walks up to two tables and asks: “Mind if I join you?”
The tables reply: “Normally, we would say yes,
but today we are just here for the view.”
So, he walks up to the bartender and orders a drink.
Says the bartender: “Not before I get your keys!”

Set Operations

- **Intersection** and **cross-product** are implemented as special cases of join
 - **equality on all attributes** as join condition computes intersection
 - **true** as join condition computes the cross-product
- **Union** and **difference** can be thought of as a selection with a complex selection condition
 - main point to address in **union** implementation is duplicate elimination
 - **difference** can be implemented using a variation of duplicate elimination
 - again, both **sorting** and **hashing** can be used for duplicate elimination

Set Operations

Sorting for union and difference

Implementation of $R \cup S$

1. sort both R and S using the combination of **all** fields
2. scan the sorted R and S in parallel and merge them, eliminating duplicates

As with projection, the implementation of difference can be integrated with the external sort operator.

Implementation of $R - S$ is similar. During the merging pass tuples of R are only written to the result after checking that they **do not appear** in S .

Set Operations

↷ Hashing for union and difference

Implementation of $R \cup S$

1. partition both R and S using a hash function $h(\cdot)$ over the combination of **all** fields
2. process each partition i as follows
 - build an in-memory hash table, using hash function $h_2(\cdot) \neq h(\cdot)$, for $S[i]$
 - scan $R[i]$; for each tuple probe the hash table for $S[i]$; if the tuple is in the hash table, discard it; otherwise, add it to the table
 - write out the hash table; clear it to prepare for the next partition

Implementation of $R - S$ is similar, but processes the partition differently. After building an in-memory hash table for $S[i]$, $R[i]$ is scanned and $S[i]$ is probed for every tuple in $R[i]$. If the tuple is **not in the table**, it is written to the result.

Aggregate Operations

- Several techniques exist to implement the aggregate operations (**SUM**, **AVG**, **COUNT**, **MIN**, and **MAX**) supported since SQL-92
- **Basic algorithm**
 - scan whole relation (possibly on-the-fly) and maintain some **running information** during that scan
 - upon completion of the scan, compute aggregate value from running information

Aggregate	Running Information
SUM	Total of values read
AVG	$\langle Total, Count \rangle$ of values read
COUNT	Count of values read
MIN	Smallest value read
MAX	Largest value read

Aggregate Operations

- For **aggregation combined with grouping**, there are two evaluation algorithms based on sorting or hashing, respectively
- **Sorting approach** (cost equal to I/O cost of sorting)
 - sort relation on grouping attribute(s)
 - scan again to compute the result of the aggregate operation (using the basic algorithm) for each group
 - as a refinement, aggregation can be done as part of the sorting step
- **Hashing approach** (if hash table fits into memory, cost is $\|R\|$)
 - build a (in-memory) hash table on the grouping attribute(s) with entries \langle grouping value, running information \rangle
 - for each tuple of the relation, probe hash table to find entry of the group to which the tuple belongs and update running information
 - when hash table is complete, its entries for grouping value can be used to compute the corresponding result tuples in a straightforward way

Aggregate Operations

- Using an **index** to select a subset of tuples is not applicable
- Under certain conditions, index entries instead of data records can be used to evaluate aggregate operations efficiently
 - if index search key includes **all attributes** needed for the aggregation query, fetching data records can be avoided by working with index entries
 - if **GROUP BY** clause attribute list forms a prefix of index search key and index is a tree index, sorting can be avoided by retrieving index entries (and data records, if necessary) in order required by grouping operation
- An index may support one or both of these techniques
- Both techniques are examples of **index-only** plans

Impact of Buffering

- Effective use of the buffer pool is very important
 - crucial for efficient implementation of a relation query engine
 - several operators use size of available buffer space as parameter
- Points to note
 1. operators that execute concurrently have to **share** the buffer pool
 2. if tuples are accessed using an index, the likelihood of finding a page in the buffer pool becomes (unpredictably) dependent on its **size** and **replacement policy**
 3. if tuples are accessed using an unclustered index, the buffer pool **fills up quickly** as each retrieved tuple is likely to require a new page to be brought into the buffer pool
 4. if an operation has a **repeated pattern of page accesses**, a clever replacement policy and/or sufficient number of buffer pages can speed up an operation significantly (*see next slide*)

Impact of Buffering

R Examples of patterns of repeated access

Simple Nested Loops Join: “*for each tuple of the outer relation, scan all pages of the inner relation*”

If there is enough buffer space to hold entire inner relation, the replacement policy is irrelevant, otherwise it is critical

- LRU will **never find** a requested page in the buffer pool (“sequential flooding”)
- MRU achieves best buffer utilization: the first $B - 2$ pages will **always stay** in the buffer pool

Block Nested Loops Join: “*for each block of the outer relation, scan all pages of the inner relation*”

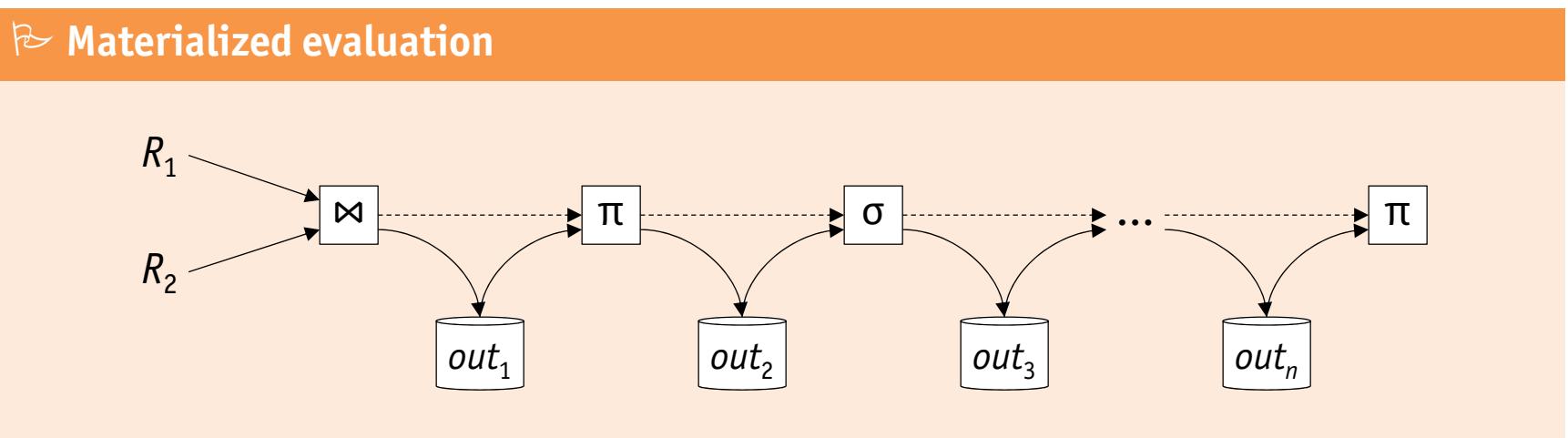
Since only one unpinned page is available for the scan of the inner relation, the replacement policy make no difference

Index Nested Loops Join: “*for each tuple of the outer relation, probe the index to find matching tuples of the inner relation*”

For duplicate values in the join attributes of the outer relation, there is a repeated access pattern on the inner relation, which can be maximized by sorting the outer relation on the join attributes

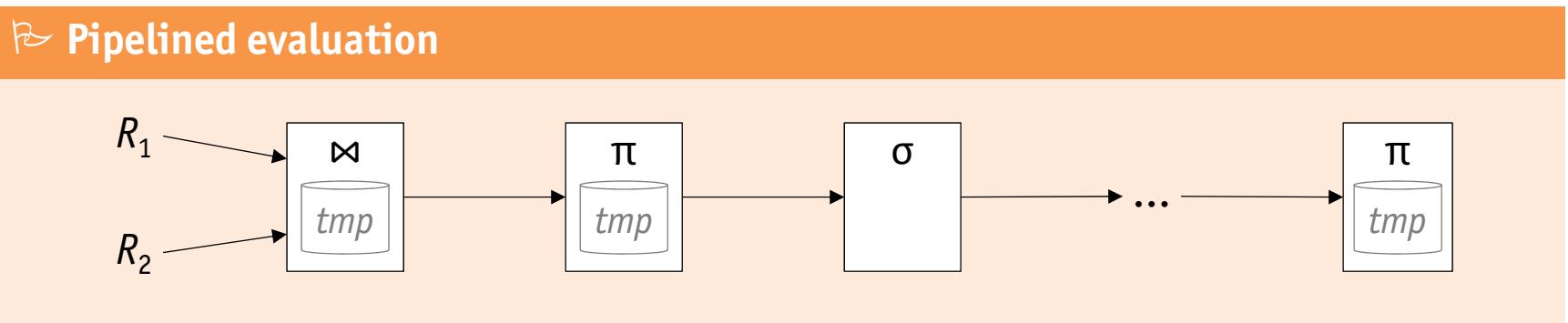
Materialized vs. Pipelined Evaluation

- Pseudo-code implementation of all relational operators discussed so far assumes **materialized evaluation**
 - operators communicate through secondary storage by **reading** (R_{in}) and **writing** (R_{out}) files, which causes **a lot of disk I/O**
 - operator **cannot start** processing until all its inputs are fully materialized
 - all operators are executed **in sequence**, first result tuple is only available after the last operator has executed



Materialized vs. Pipelined Evaluation

- Alternatively, **pipelined evaluation** can be used to avoid writing temporary files to disk whenever possible
- Each operator passes its results **directly** to the next operator
 - as soon as results are available, propagate output **immediately**
 - as soon as input data is available, start computing **as early as possible**
 - all operators can execute in **parallel**



Pipelined Evaluation

- To support pipelined evaluation, the current implementation of the relational operators discussed so far has to be adapted
- The **granularity** at which data is passed is an important design decision that may influence performance
 - **communication**: fine granularity reduces response time as results are produced as early as possible
 - **scheduling/control**: coarse granularity may improve the effectiveness of (instruction) caches and buffer pool as there are fewer context switches

The Real World

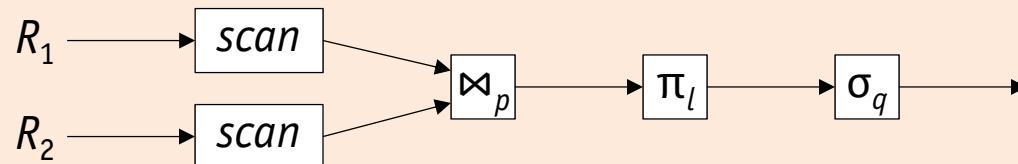
Actual systems typically operate at one **tuple at a time**.

Pipelined Evaluation

- Pipelined evaluation is typically implemented based on the **open-next-close interface** or **Volcano iterator model**
- Each relational operator implements the functions
 - open ()** **initialize** the operator's internal state
 - next ()** produce and return the **next result tuple** or **(EOF)**
 - close ()** **free** any operator-internal resources, typically after all tuples have been processed
- All **state** is kept within each operator instance
 1. upon call to **next ()**, operator produces a tuple, updates its internal state and then **pauses**
 2. upon subsequent call to **next ()**, operator uses its internal state to **resume** and produce another tuple

Pipelined Evaluation

Example



- Query plans like the one shown above are evaluated by the query processor as follows
 1. reset query plan by calling **open()** on root operator, i.e., $\sigma_q.\text{open}()$
 2. operators forward **open()** call through the query plan
 3. control returns to query processor
 4. next (first) tuple is produced by calling **next()** on the root operator, i.e., $\sigma_q.\text{next}()$
 5. operators forward **next()** call through the query plan as needed
 6. as soon as the next (first) record is produced, control returns to query processor again

Pipelined Evaluation

💻 Volcano-style query evaluator

```
function eval(q)
    q.open();
    r ← q.next();
    while r ≠ <EOF> do
        emit(r);                                (deliver record r, i.e., print or send to DB client)
        r ← q.next();
    q.close();                                 (deallocate all resources)
end
```

- Iterator interface above implements a **demand-driven** query processing infrastructure
 - **consumers** (later operators) request input on-demand from **producers** (earlier operators) whenever they are ready to process it
 - demand-driven pipelining **minimizes** resource requirements and “wasted” effort in case a user/client does not request the whole result

Pipelined Evaluation

⌚ Volcano-style implementation of $\sigma_p(R_{in})$

```
function open()
    Rin.open();
end

function next()
    while (r ← Rin.next()) ≠ ⟨EOF⟩ do
        if p(r) then return r;
    return ⟨EOF⟩;
end

function close()
    Rin.close();
end
```

- Remarks
 - R_{in} is the input operator (sub-plan root) of $\sigma_p(R_{in})$
 - $p(\cdot)$ is the predicate of $\sigma_p(R_{in})$

Pipelined Evaluation

✍ Exercise: Volcano-style implementation of $\bowtie_p^{nl}(R_1, R_2)$

Pipelined Evaluation

✍ Exercise: Volcano-style implementation of $\bowtie_p^{nl}(R_1, R_2)$

```
function open ()  
     $R_1.\text{open}();$   
     $R_2.\text{open}();$   
     $r_1 \leftarrow R_1.\text{next}();$   
end
```

```
function next ()  
    while  $r_1 \neq \langle \text{EOF} \rangle$  do  
        while ( $r_2 \leftarrow R_2.\text{next}()$ )  $\neq \langle \text{EOF} \rangle$  do  
            if  $p(r_1, r_2)$  then return  $\langle r_1, r_2 \rangle$ ;  
         $R_2.\text{close}();$   
         $R_2.\text{open}();$   
         $r_1 \leftarrow R_1.\text{next}();$   
    return  $\langle \text{EOF} \rangle$ ;  
end
```

```
function close ()  
     $R_1.\text{close}();$   
     $R_2.\text{close}();$   
end
```

*(emit concatenated result)
(reset inner join input)*

Pipelined Evaluation

- Pipelining **avoids materialization** and **reduces response time** because data is processed one tuple (chunk of data) at a time
- So-called **blocking operators** cannot be implemented in this way
 - entire input needs to be consumed before output can be produced
 - operator has to internally buffer (materialize) the data on disk

Exercise: examples of blocking operators

Which operators discussed so far do not permit pipelining without internal materialization?

Pipelined Evaluation

- Pipelining **avoids materialization** and **reduces response time** because data is processed one tuple (chunk of data) at a time
- So-called **blocking operators** cannot be implemented in this way
 - entire input needs to be consumed before output can be produced
 - operator has to internally buffer (materialize) the data on disk

Exercise: examples of blocking operators

Which operators discussed so far do not permit pipelining without internal materialization?

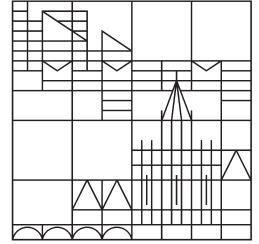
- ↳ (external) **sorting**
- ↳ **projection** with duplicate elimination over unsorted input
- ↳ **sort-merge join** over unsorted input
- ↳ **hash join**
- ↳ **grouping** over unsorted input

Pipelined Evaluation

Iterator	open()	next()	close()	State
print	open input	call next() on input, format item on screen	close input	
scan	open file	read next item	close file	open file descriptor
select	open input	call next() on input, until an item qualified	close input	
hash join without overflow resolution	allocate hash directory, open left <i>build</i> input, build hash table calling next() on build input, close build input, open right <i>probe</i> input	call next() in <i>probe</i> input until a match is found	close <i>probe</i> input, deallocate hash directory	hash directory
merge join without duplicates	open both inputs	get next() item from input with smaller key until a match is found	close both inputs	
sort	open input, build all initial run files calling next() on input, close input, merge run files until only one step is left	determine next output item, read new item from the correct run file	destroy remaining run files	merge heap, open file descriptors for run files

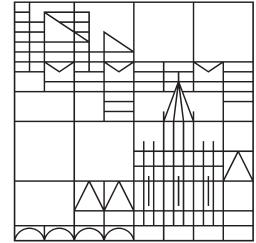
Pipelined Evaluation

- An alternative query processing infrastructure to the demand-driven pipelining of the iterator model is **data-driven** pipelining
 - producer and consumer operators are connected by a **queue**
 - all operators execute **asynchronously** and in **parallel**
 - operators process all their input data available from incoming queue(s), produce results as fast as possible, and enqueue them to outgoing queue
- This model relies on queues for communication and scheduling
 - queues **buffer** data that is pipelined between operators
 - queues **suspend** operators using blocking enqueue/dequeue calls
- Data-driven pipelining is able **exploit more parallelism** than demand-driven pipelining
 - operators only need to wait, if input queue is empty or output queue is full
 - trades off higher resources requirements for more parallelism



Database System Architecture and Implementation

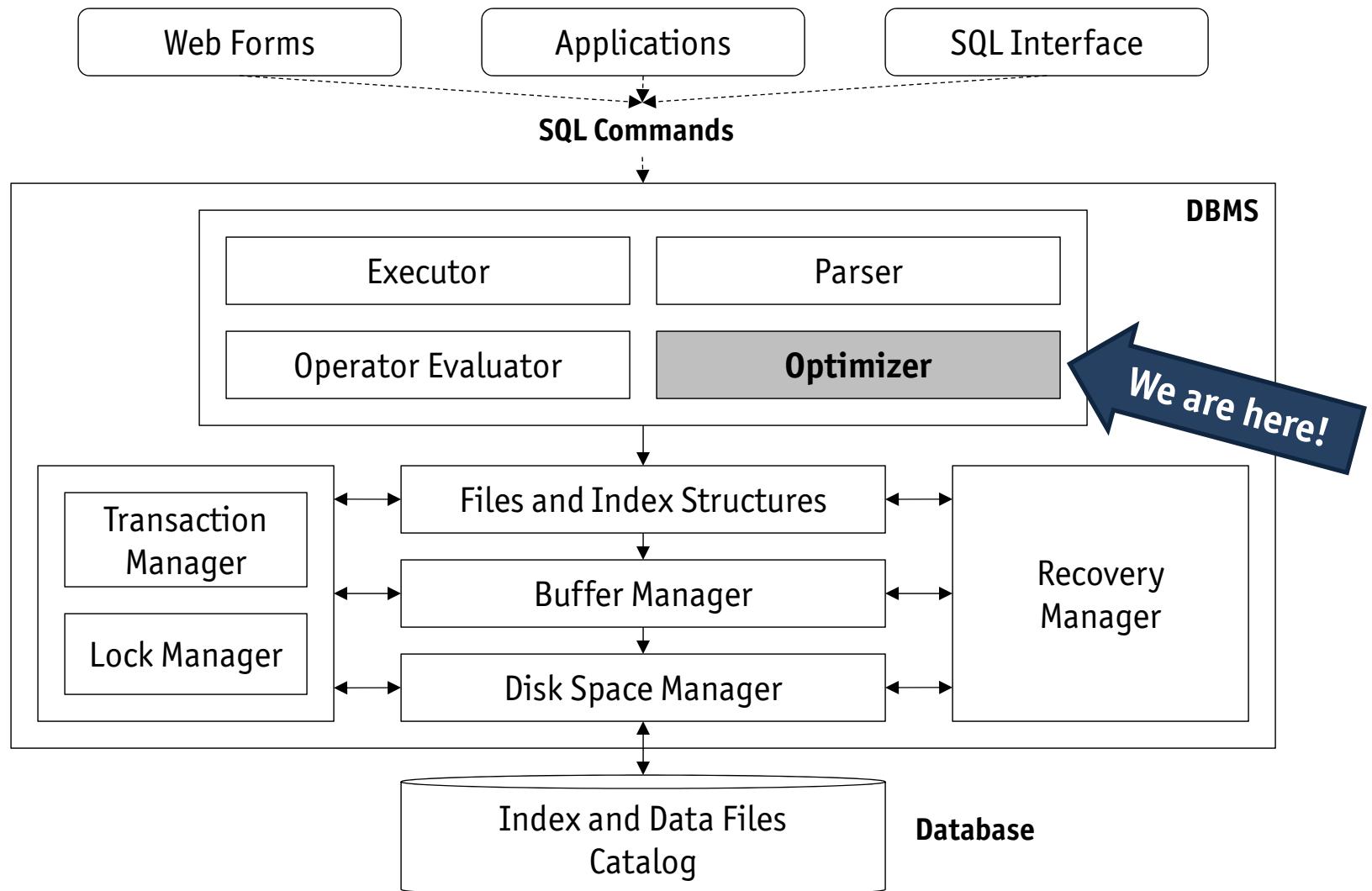
TO BE CONTINUED...



Database System Architecture and Implementation

Module 8
Query Optimization
January 15, 2014

Orientation



Module Overview

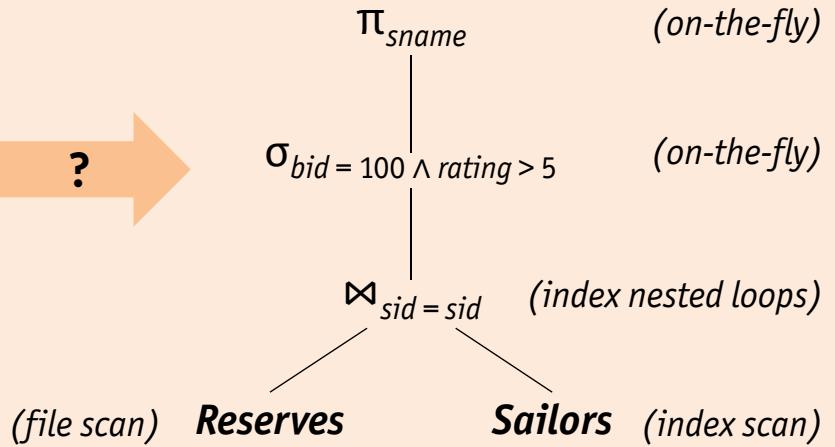
- Translating queries into algebra
- Relational algebra equivalences
- Plan enumeration
- Cost estimation and histograms
- Nested sub-queries
- Example query optimizers

Outline

From SQL to a query execution plan

```
SELECT sname  
FROM   Reserves R, Sailors S  
WHERE  R.sid = S.sid AND  
       R.bid = 100 AND  
       S.rating > 5
```

?



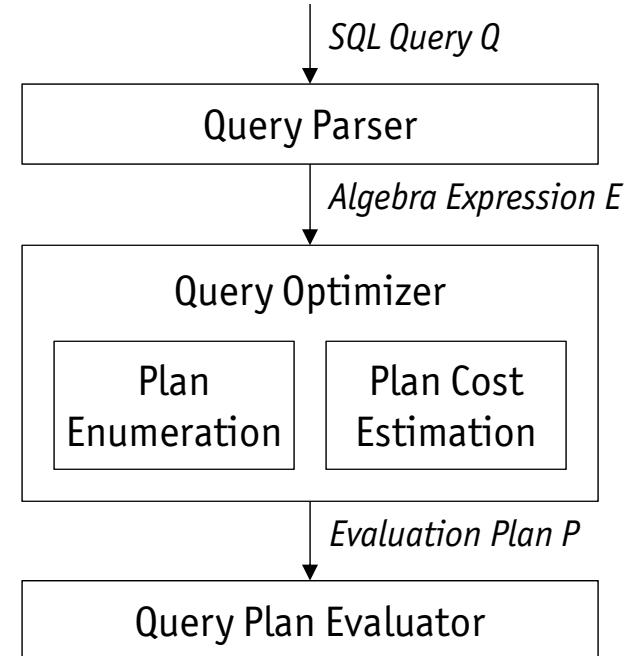
- Recall that there may be **many ways** to answer a given SQL query
 - different query trees (pushing selections and projections, join orders)
 - different physical operators for the same logical operator
 - different parameters (block size, buffer allocation, ...)
- Task of finding the **best** execution plan is crucial to any database implementation

Outline

- To evaluate a given (SQL) query Q , a database management system performs the following steps
 1. parse and analyze Q
 2. derive a **relational algebra** expression E that computes Q
 3. generate a set of **logical plans** L by transforming and simplifying E
 4. generate a set of **physical plans** P by annotating each plan in L with access methods and operator algorithms
 5. for each plan in P , **estimate the quality** (cost) of the plan and choose the best plan as the final evaluation plan
- Query optimizer encompasses the last three steps
 - **plan enumeration** in Steps 3 and 4
 - **algebraic** (or rewrite) query optimization in Step 3
 - **non-algebraic** (or cost-based) query optimization in Steps 4 and 5

Outline

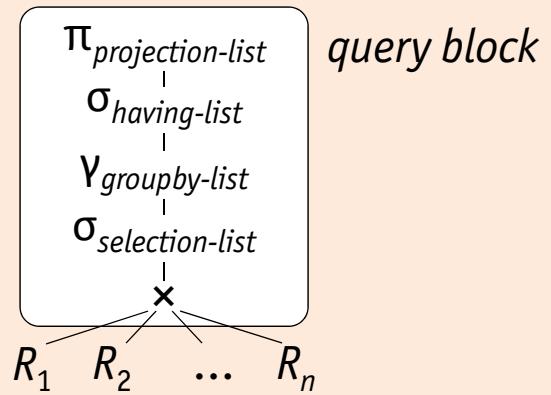
- **Query parser** performs syntactical and semantical analysis and checks
- **Plan enumeration**
 - **rewrite optimization** uses heuristics independent of the current database state
 - **cost-based optimization** relies on plan cost estimation to annotate plan with access path and operator algorithms
- **Plan cost estimation** assigns a cost to a plan based on
 - cost model (I/O cost, CPU cost, etc.)
 - current database state (table sizes, index availability, etc.)
- **Evaluator** executes the resulting plan using the relational operators presented in the previous two modules



Parser

Deriving a query block from an SQL statement

```
SELECT projection-list
      FROM  $R_1, R_2, \dots, R_n$ 
     WHERE selection-list
    GROUP BY groupby-list
   HAVING having-list
```



- After checking its syntactical and semantical correctness, parser creates **internal representation** of the input query
 - internal representation resembles the original query
 - each SELECT-FROM-WHERE clause is translated to a **query block**
 - each R_i can be a base relation or another query block

Query Optimizer

- Query optimizer considers each query block and chooses an evaluation plan for this block
 - for time being the focus is on **single-block queries**
 - optimization of **nested queries** will be discussed separately
- In order to find the best evaluation plan, query optimizer explores so-called **search space** of possible alternative plans
 - **logical level:** relational algebra equivalences
 - **physical level:** access methods and operator algorithms

Relational Algebra Equivalences

- Recall that each query block is a relation algebra expression consisting of cross-product, selection, and projection
- Rewrite optimization applies **relational algebra equivalences** to transform a query block into an equivalent expression
 - convert cross-product to join
 - choose different join orders
 - push selections and projections ahead of joins
- Two relational algebra expressions E_1, E_2 are **equivalent** if they generate the same set of tuples on every legal database instance
- Such equivalences imply **equivalence rules** of the form $E_1 \equiv E_2$
 - optimizer may apply equivalence rules in both directions (\rightarrow, \leftarrow)
 - these equivalence rules are first introduced in the course “Database Systems” (INF-12040)

Relational Algebra Equivalences

- Selections
 1. conjunctive selections can be deconstructed into a sequence of individual selections (**cascading selections**)

$$\sigma_{c1 \wedge c2 \wedge \dots \wedge cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\dots \sigma_{cn}(R))\dots)$$

2. selections are **commutative**

$$\sigma_p(\sigma_q(R)) \equiv \sigma_q(\sigma_p(R))$$

- Projections
 3. only the last projection in a sequence of projections is needed, the others can be omitted (**cascading projections**)

$$\pi_{a1}(R) \equiv \pi_{a1}(\pi_{a2}(\dots (\pi_{an}(R))\dots))$$

where $a_i \subseteq a_{i+1}$ for $i = 1, \dots, n - 1$

Relational Algebra Equivalences

- Cross-products and natural joins

4. are both **commutative**

$$R \times S \equiv S \times R$$

$$R \bowtie S \equiv S \bowtie R$$

5. are both **associative**

$$R \times (S \times T) \equiv (R \times S) \times T$$

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$$

- General joins

6. are **associative**

$$(R \bowtie_p S) \bowtie_{q \wedge r} T \equiv R \bowtie_{p \wedge q} (S \bowtie_r T)$$

where predicate r involves attributes of S and T only

Relational Algebra Equivalences

Importance of these equivalences

What is the relevance of these two properties of cross-products and joins with respect to query optimization?

- ⇒ optimizer can choose which relation is the inner and which is the outer relation (**commutative property**)
- ⇒ optimizer can choose any join order (**both properties**)

Relational Algebra Equivalences

- Selections, cross-product, and join

7. selections can be **combined with cross-product** to form a join

$$\sigma_p(R \times S) \equiv R \bowtie_p S$$

8. selections can be **combined with join**

$$\sigma_p(R \bowtie_q S) \equiv R \bowtie_{p \wedge q} S$$

9. selections **commute** with cross-products and joins, if predicate p involves attributes of R only

$$\sigma_p(R \times S) \equiv \sigma_p(R) \times S$$

$$\sigma_p(R \bowtie_q S) \equiv \sigma_p(R) \bowtie_q S$$

10. selections **distribute** over cross-product and join, if predicate p only involves attributes of R and predicate q only involves attributes of

$$\sigma_{p \wedge q}(R \times S) \equiv \sigma_p(R) \times \sigma_p(S)$$

$$\sigma_{p \wedge q}(R \bowtie_r S) \equiv \sigma_p(R) \bowtie_r \sigma_p(S)$$

Relational Algebra Equivalences

- Projections, selections, cross-product, and join
 11. projection and selection **commute** if the selection predicate p only involves attributes retained by the projection list a
$$\pi_a(\sigma_p(R)) \equiv \sigma_p(\pi_a(R))$$
 12. projection **distributes** over cross-product
$$\pi_a(R \times S) \equiv \pi_{a1}(R) \times \pi_{a2}(S)$$
where a_1 is the subset of attributes in a that appear in R and a_2 is the subset of attributes that appear in S
 13. projection **distributes** over join
$$\pi_a(R \bowtie_p S) \equiv \pi_{a1}(R) \bowtie_p \pi_{a2}(S)$$
where a_1 is the subset of attributes in a that appear in R , a_2 is the subset of attributes that appear in S , and predicate p only involves attributes in $a_1 \cup a_2$

Relational Algebra Equivalences

- Union and intersection

14. are both **commutative**

$$R \cup S \equiv S \cup R$$

$$R \cap S \equiv S \cap R$$

15. are both **associative**

$$(R \cup S) \cup T \equiv R \cup (S \cup T)$$

$$(R \cap S) \cap T \equiv R \cap (S \cap T)$$

- Projection and union

16. projection **distributes** over union

$$\pi_a(R \cup S) \equiv \pi_a(R) \cup \pi_a(S)$$

Relational Algebra Equivalences

- Selection, union, intersection, and difference

17. are **distributive**

$$\sigma_p(R \cup S) \equiv \sigma_p(S) \cup \sigma_p(R)$$

$$\sigma_p(R \cap S) \equiv \sigma_p(S) \cap \sigma_p(R)$$

$$\sigma_p(R \setminus S) \equiv \sigma_p(S) \setminus \sigma_p(R)$$

18. intersect and differences are **commutative** (does **not** apply for union)

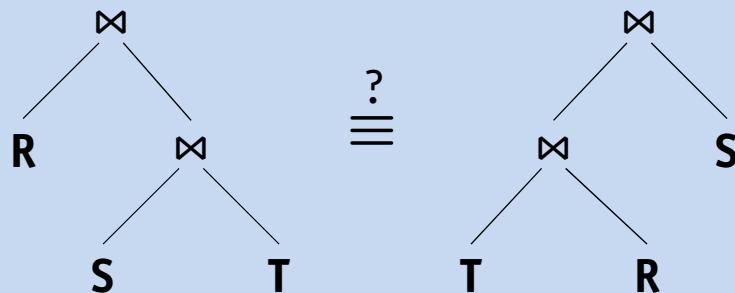
$$\sigma_p(R \cap S) \equiv \sigma_p(S) \cap R$$

$$\sigma_p(R \setminus S) \equiv \sigma_p(S) \setminus R$$

Relational Algebra Equivalences

Verification of equivalence

Using relational algebra equivalences, show that the following two query trees are equivalent.



- Relational algebra expression

$$R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$$

- From (natural) join commutativity [4]

$$R \bowtie (S \bowtie T) \equiv R \bowtie (T \bowtie S)$$

- From (natural) join associativity [5]

$$R \bowtie (T \bowtie S) \equiv (R \bowtie T) \bowtie S$$

- From (natural) join commutativity [4]

$$(R \bowtie T) \bowtie S \equiv (T \bowtie R) \bowtie S$$

Rewrite Optimization

- Query optimizers use relational algebra equivalence rules to improve expected performance of a query in **most cases**
- Optimization is guided by the following **heuristics**
 1. **Break apart conjunctive selections** into a sequence of simpler selections (rule [1], preparatory step for second step)
 2. **Move selections down the query tree** to reduce the cardinality of intermediate results as early as possible (rules [2], [9], and [17])
 3. **Replace selection–cross-product pairs with joins** to avoid large intermediate results (rule [7])
 4. **Break lists of projection attributes apart, move them down the query tree, and create new projections where possible** to reduce tuple widths as early as possible (rules [3], [13], and [16])
 5. **Perform joins with the smallest expected result first** (cost-based heuristic)

Rewrite Optimization

Example query

```
SELECT *
  FROM Sailors S
 WHERE S.rating * 100 > 50
```

Example query after predicate simplification

```
SELECT *
  FROM Sailors S
 WHERE S.rating > 0.5
```

Importance of predicate simplification

What is the main reason why the query optimizer rewrites the selection predicate as shown above?

Rewrite Optimization

Example query

```
SELECT *
  FROM A, B, C
 WHERE A.a = B.b AND B.b = C.c
```

Example query after implicit join predicate expansion

```
SELECT *
  FROM A, B, C
 WHERE A.a = B.b AND B.b = C.c AND A.a = C.c
```

- Implicit join predicates can be turned into explicit one to enable more join orders
- In the example, the rewrite makes the join tree $(A \bowtie C) \bowtie B$ feasible, which otherwise would have needed a cross-product

Plan Enumeration

- Given a query, an optimizer **enumerates** a certain set of plans and chooses the plan with the least estimated cost
- The **search space** of alternative plans is given by
 - relational algebra equivalences
 - choice of implementation technique for relational operators
 - choice of access method based on presence of indexes
 - other available resources, e.g., number of buffer pages, etc.
- In general, it is impossible to enumerate all plans as it would be prohibitively expensive for all but the simplest queries
- Two important cases can be distinguished
 - single-relation queries
 - multiple relation queries

Single-Relation Queries

- Optimizer enumerates **all possible plans** and assesses their cost
- Main decision in a single-relation plan is the access method used to retrieve the tuples of the relation
 - plans without index
 - plans utilizing an index
- Observe that different single-relation plans might have different **physical properties**, i.e., sort order to the produced tuples
- For each set of physical properties, the plan with the least estimated cost is retained

Single-Relation Queries

- Plans without index
 1. heap file scan on (single) relation
 2. apply selection and projection (without duplicate elimination) on-the-fly
 3. sort according to **GROUP BY** clause
 4. apply aggregation and **HAVING** clause on-the-fly to each group
- Single-index access path
 1. choose index that is estimated to retrieve the fewest pages
 2. apply projections and non-primary selections
 3. proceed to compute grouping and aggregation operations by sorting
- Multiple-index access path (for index entry variants ② and ③)
 1. retrieve and intersect rid sets and sort results by page id
 2. retrieve tuples that satisfy primary condition of all indexes
 3. apply projections and non-primary selection terms, followed by grouping and aggregation operations

Single-Relation Queries

- Sorted-index access path
 1. retrieve tuples in order required by **GROUP BY** clause
 2. apply selection and projection to each retrieved tuple on-the-fly
 3. compute aggregate operations for each group on-the-fly
- Index-only access path
 1. retrieve matching tuples or perform an index-only scan
 2. apply (non-primary) selection conditions and projections to each retrieved tuple on-the-fly
 3. possibly, sort the result to achieve grouping
 4. compute aggregate operations for each group on-the-fly

Single-Relation Queries

The Real World

All major RDBMS recognize the importance of **index-only plans** and look for such plans whenever possible.

↳ IBM DB2

- users can specify a set of **include columns** that are kept in the index, but are not part of the index key
- this feature enables a richer set of index-only queries to be handled

↳ Microsoft SQL Server

- exploits indexes for partially matching selection predicates by **joining index entries** on the rid of the data record

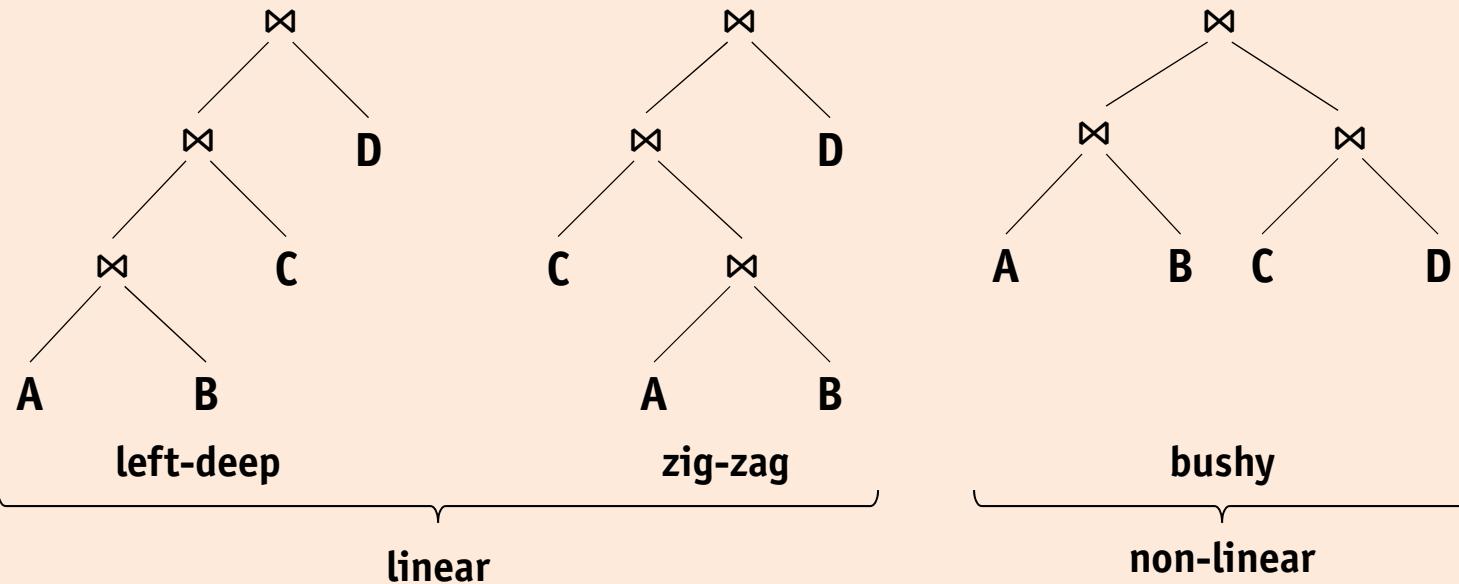
Multiple-Relation Queries

- Queries over multiple relations require joins (or cross-products)
- Finding a good plan for such queries is very important
 - these queries can be quite expensive (since joins are expensive)
 - queries involving joins are the normal case in a normalized database
- Size of **final result** can be estimated by taking product of
 - sizes of all relations
 - reduction factors of all selection predicates
- Size of **intermediate relations** can vary substantially depending on order in which relations are joined
- Therefore multiple-relation plans can have **very** different costs

Multiple-Relation Queries

Enumerating multiple-relation plans

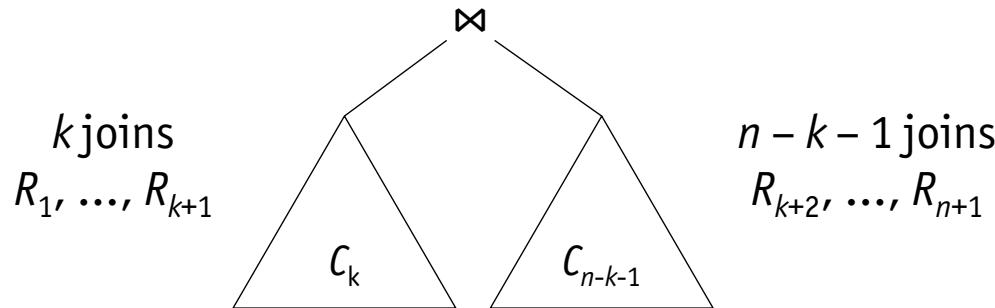
Using relational algebra equivalences (joins commute), the following equivalent plans can be enumerated for the query $A \bowtie B \bowtie C \bowtie D$.



Of course, these are **not all** possible join orders for a four-way join. In fact, there are many more...

Multiple-Relation Queries

- A join over $n + 1$ relations R_1, \dots, R_{n+1} requires n **binary joins**
- Its **root-level operator** joins sub-plans of k and $n - k - 1$ join operators ($0 \leq k \leq n - 1$)



- Let C_i be the **number of possibilities** to construct a binary tree of i inner nodes (join operators)

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1}$$

Multiple-Relation Queries

- It turns out that this recurrence is satisfied by **Catalan numbers** describing the number of ordered binary trees with $n + 1$ leaves

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1} = \frac{(2n)!}{(n+1)! \cdot n!}$$

- For each of these trees, the input relations R_1, \dots, R_{n+1} can be permuted, which adds a factor $(n + 1)!$

 Number of possible join trees for an $(n + 1)$ -way relational join

$$\frac{(2n)!}{(n+1)! \cdot n!} \cdot (n+1)! = \frac{(2n)!}{n!}$$

Multiple-Relation Queries

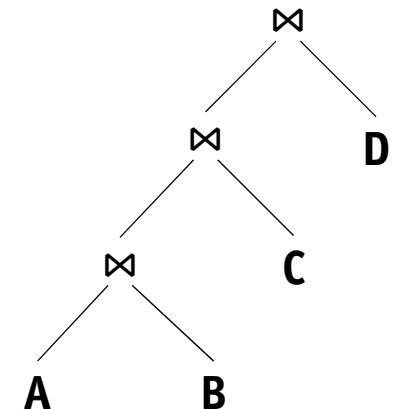
☞ Explosion of the search space

number of relations n	C_{n-1}	join trees
2	1	2
3	2	12
4	5	120
5	14	1,680
6	42	30,240
7	132	665,280
8	429	17,297,280
10	4,862	17,643,225,600

- Remarks
 - formula only include **logical** query evaluation plans
 - considering the use of k **different join algorithms** adds a factor k^{n-1}

Multiple-Relation Queries

- It is impossible to enumerate all plan and that is a fact
 - optimizer will **not** be able to find the overall best plan
 - optimizer can do damage control by trying to avoid **really bad** plans
- Restrict the search space by only considering **left-deep plans**
 - left-deep plans can be translated to **fully pipelined** plans because inner relation is already materialized
 - some join algorithms may benefit from index on inner relation
- Number of possible left-deep joins orders for an n -way join is “only” $n!$



Multiple-Relation Queries

- If the optimizer cannot find the overall best plan, at least it needs to ensure that it does not miss the best left-deep plan
- There are still some degrees of freedom left
 - for each base relation in the query, consider all **access methods**
 - for each join operation in the left-deep tree, select a **join algorithm**

⤵ How many possible query plans are left now?

Back-of-envelope calculation for a query with n relations, assuming j available join algorithms and i indexes per relation

$$\# \text{plans} \approx n! \cdot j^{n-1} \cdot (i+1)^n$$

⤵ Example with $n = 3$ relations, $j = 3$, and $i = 2$

$$\# \text{plans} \approx 3! \cdot 3^2 \cdot 3^3 = 1458$$

Multiple-Relation Queries

↷ Step-by-step example: Setup and assumptions

```
SELECT S.sname, R.rname, B.bname  
      FROM Sailors S, Reserves R, Boats B  
     WHERE S.sid = R.sid AND R.bid = B.bid
```

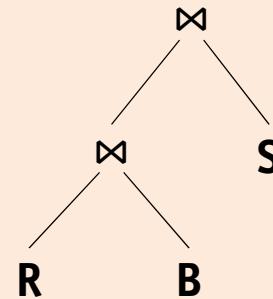
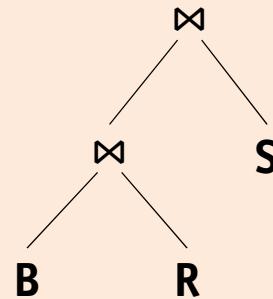
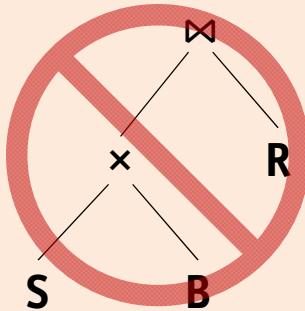
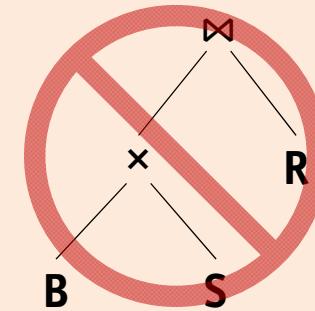
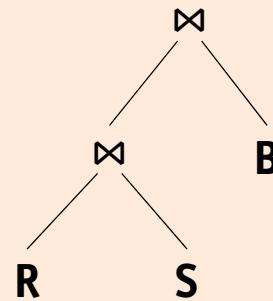
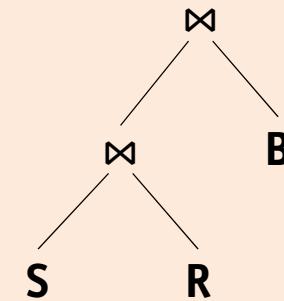
Assumptions

- Available join algorithms
 - hash join
 - block nested loops join
 - index nested loops join
- Available indexes
 - clustered B+ tree index **I** on **R.sid**, $INPages(\mathbf{I}) = 50$
- Relation cardinality
 - $NPages(\mathbf{S}) = 500$ pages, 80 tuples/page
 - $NPages(\mathbf{R}) = 1000$ pages, 100 tuples/page
 - $NPages(\mathbf{B}) = 10$ pages
- 100 ($\mathbf{R} \bowtie \mathbf{S}$)-tuples fit on a page

Multiple-Relation Queries

Step-by-step example: Enumerate candidate plans

- Enumerate $n!$ left-deep join trees ($3! = 6$)

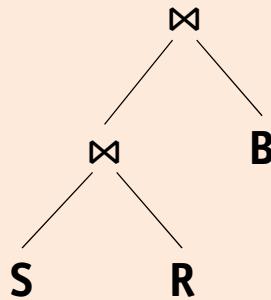


- Prune** plans that need a cross-product immediately (no join predicate between S, B)
- Four **candidate plans** remain

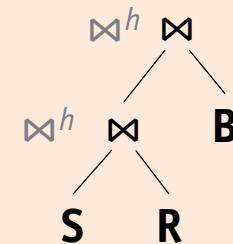
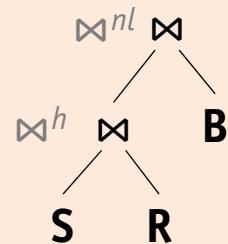
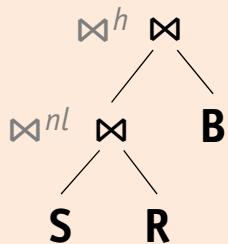
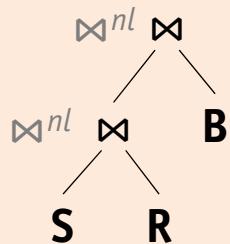
Multiple-Relation Queries

↷ Step-by-step example: Choose join algorithms

Candidate plan



Possible join algorithm choices for this candidate plan

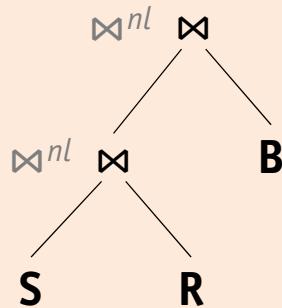


↷ Repeat for remaining three candidate plans

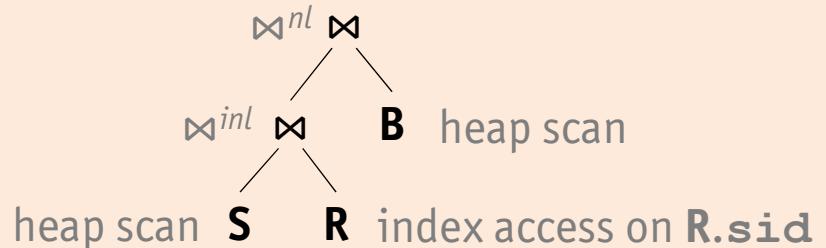
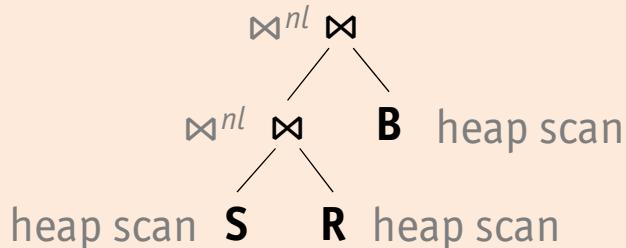
Multiple-Relation Queries

↷ Step-by-step example: Choose access methods

Candidate plan



Possible access method choices for this candidate plan

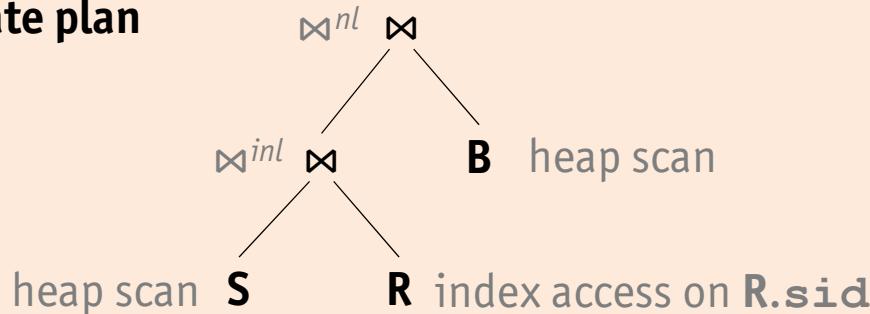


↷ Repeat for remaining candidate plans

Multiple-Relation Queries

↷ Step-by-step example: Cost estimation

Candidate plan



Cost estimation for this candidate plan

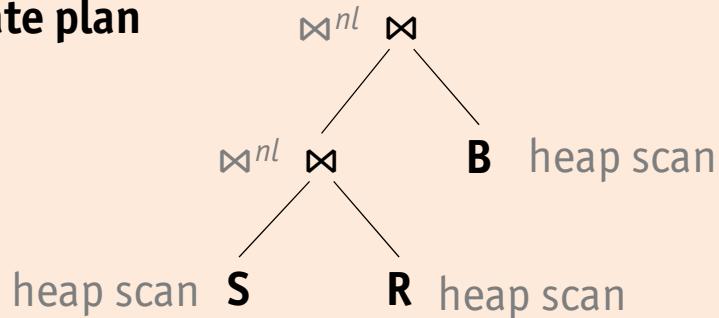
- cost of heap scan on **S**: 500 pages
- cost of **S** \bowtie **R** (**R.sid** is a foreign key)
 - $NTuples(S) \cdot sel(S.sid = R.sid) \cdot (NPages(R) + NPages(I))$
 - $40,000 \cdot 1/40,000 \cdot (1000 + 50) = 1050$ pages
- $NPages(S \bowtie R) = NTuples(S \bowtie R)/100 = NTuples(R)/100 = 100,000/100 = 1000$ pages
- cost of $(S \bowtie R) \bowtie B$: $NPages(S \bowtie R) \cdot NPages(B) = 1000 \cdot 10 = 10,000$ pages

Total estimated cost: $500 + 1050 + 10,000 = \underline{11,500}$ pages

Multiple-Relation Queries

↷ Step-by-step example: Cost estimation

Candidate plan



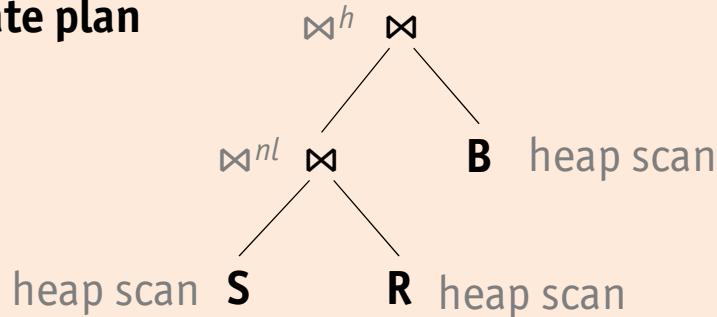
Total estimated cost for candidate plan

- $NPages(S) + NPages(S) \cdot NPages(R) + NPages(S \bowtie R) \cdot NPages(B)$
- $500 + 500 \cdot 1000 + 1000 \cdot 10 = \underline{510,500 \text{ pages}}$

Multiple-Relation Queries

↷ Step-by-step example: Cost estimation

Candidate plan



Assumption

- \bowtie^h requires two passes

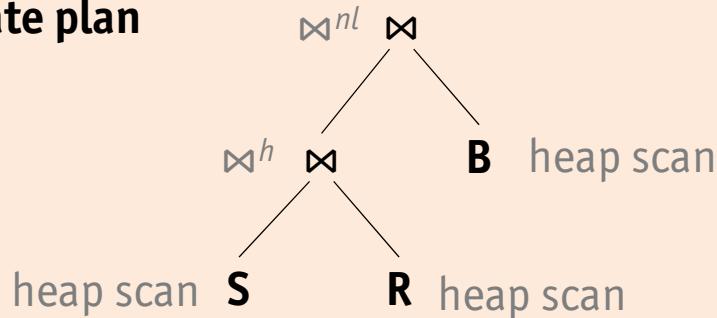
Total estimated cost for candidate plan

- $NPages(S) + NPages(S) \cdot NPages(R) + 2 \cdot (NPages(S \bowtie R) + NPages(B))$
- $500 + 500 \cdot 1000 + 2 \cdot (1000 + 10) = \underline{502,520 \text{ pages}}$

Multiple-Relation Queries

↷ Step-by-step example: Cost estimation

Candidate plan



Assumption

- \bowtie^h requires two passes

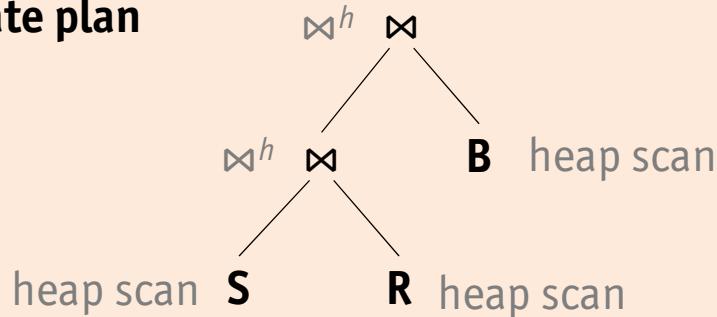
Total estimated cost for candidate plan

- $2 \cdot (NPages(S) + NPages(R)) + NPages(S \bowtie R) \cdot NPages(B)$
- $2 \cdot (500 + 1000) + 1000 \cdot 10 = \underline{13,000 \text{ pages}}$

Multiple-Relation Queries

↷ Step-by-step example: Cost estimation

Candidate plan



Assumption

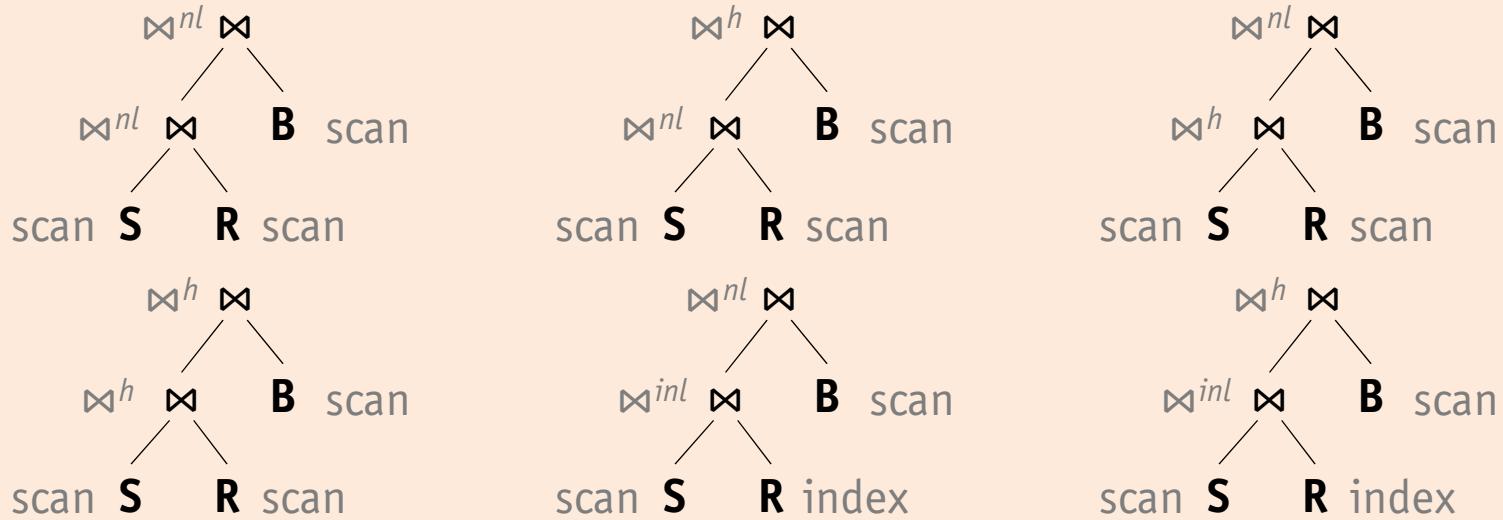
- \bowtie^h requires two passes

Total estimated cost for candidate plan

- $2 \cdot (NPages(S) + NPages(R)) + 2 \cdot (NPages(S \bowtie^h R) + NPages(B))$
- $2 \cdot (500 + 1000) + 2 \cdot (1000 + 10) = \underline{5020 \text{ pages}}$

Multiple-Relation Queries

⟳ Repeated enumeration of identical sub-plans



- Plan enumeration reconsiders the same sub-plans over and over
 - cost and result size of sub-plan are **independent** of embedding plan
 - optimizer needs to avoid **regenerating** and **reassessing** such sub-plans
- Dynamic programming **memoizes** already considered sub-plans

Dynamic Programming

- Dynamic programming approach was pioneered by System R
 - find the cheapest plan for an n -way join in n **passes**
 - in each pass k , find the best plan for all **k -relation sub-plans**
 - **construct** the plans in pass k by joining another relation to the best $(k - 1)$ -relation sub-plans found in earlier passes

Principle of Optimality

Assumption

↳ To find the optimal **global plan**, it is sufficient to only consider the optimal plans for all its possible **sub-plans**

Dynamic Programming

Pass 1 (all 1-relation plans)

- find best 1-relation plan for each relation
- this pass mainly consists of selecting access method
- also see discussion on single-relation queries

Keep the best 1-relation plans for each set of physical properties

Pass 2 (all 2-relation plans)

- find best way to join sub-plans from Pass 1 to another relation
 - generate left-deep trees with sub-plans from Pass 1 as outer relation in these joins
- Again, keep the best 2-relation plans for each set of physical properties

⋮

Pass n (all n -relation plans)

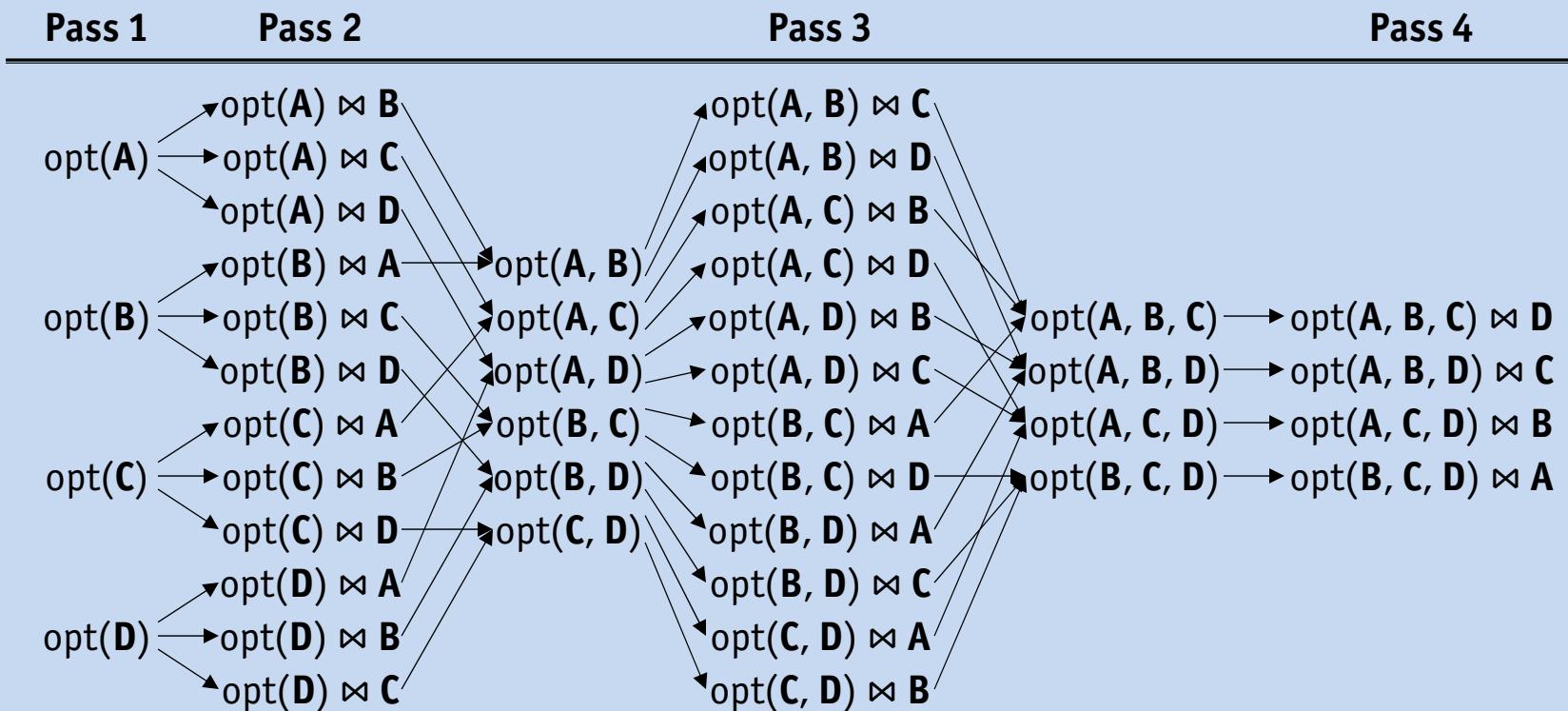
- find best way to join sub-plans of Pass $n - 1$ to the n th relation
- sub-plans of Pass $n - 1$ appear as outer relation in this join

Return the **overall best** plan

Dynamic Programming

Number of join orderings enumerated

Ignoring the physical properties of a plan, the presence of different join algorithms, and the availability of indexes, how many plans does the dynamic programming strategy enumerate to find the best plan for $A \bowtie B \bowtie C \bowtie D$?



Dynamic Programming

- Dynamic programming records **cost** and **result size estimates** for each retained plan
- **Pruning** for each subset of joined relations
 - keep cheapest sub-plan **overall**
 - keep cheapest sub-plans that generate an intermediate result with an **interesting order** of tuples
 - discard sub-plans that involve **cross-products** due to lack of a join condition between two relations
- **Interesting order** determined by
 - presence of SQL **ORDER BY** clause in the query
 - presence of SQL **GROUP BY** clause in the query
 - join attributes of subsequent equi-joins (prepare for merge join)

Dynamic Programming

↷ Step-by-step example: Setup and assumptions

```
SELECT S.sname, R.rname, B.bname  
      FROM Sailors S, Reserves R, Boats B  
     WHERE S.sid = R.sid AND R.bid = B.bid
```

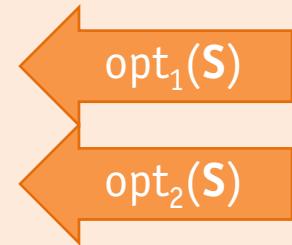
Assumptions

- Available join algorithms
 - merge join
 - block nested loops join
 - index nested loops join
- Available indexes
 - clustered B+ tree index **I** on **S.sid**, $height(\mathbf{I}) = 3$, $INPages(\mathbf{I}) = 500$
- Relation cardinality
 - $NPages(\mathbf{S}) = 10,000$ pages, 5 tuples/page
 - $NPages(\mathbf{R}) = 10$ pages, 10 tuples/page
 - $NPages(\mathbf{B}) = 10$ pages, 20 tuples/page
- 10 (**R** \bowtie **S**)-tuples fit on a page, 10 (**B** \bowtie **R**)-tuples fit on a page

Dynamic Programming

↷ Step-by-step example: Pass 1 (1-relation plans)

- access methods for **S**
 1. heap scan
 $\text{cost} = N\text{Pages}(S) = 10,000$
 2. index scan on **S.sid**, index **I**
 $\text{cost} = IN\text{Pages}(I) + N\text{Pages}(S) = 10,500$
- access method for **R**
 1. heap scan
 $\text{cost} = N\text{Pages}(R) = 10$
- access method for **B**
 1. heap scan
 $\text{cost} = N\text{Pages}(B) = 10$



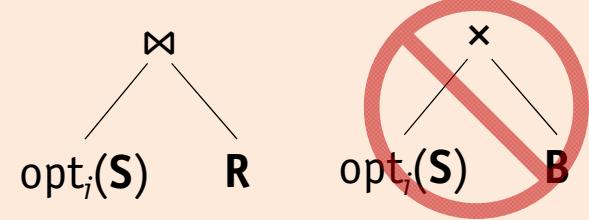
Note that both $\text{opt}_1(S)$ and $\text{opt}_2(S)$ are kept, since $\text{opt}_2(S)$ has an **interesting order** on attribute **sid**, which is a join attribute

Dynamic Programming

↷ Step-by-step example: Pass 2 (2-relation plans)

Plans with $\text{opt}_i(\mathbf{S})$ as outer relation

- $\text{opt}_i(\mathbf{S})$ with \mathbf{R} as inner relation
 1. **block nested loops join**, $\text{opt}_1(\mathbf{S})$ using **heap scan**
 $\text{cost} = 10,000 + 10,000 \cdot \text{NPages}(\mathbf{R}) = \underline{110,000}$
 2. **merge join** (with two-way sort), $\text{opt}_1(\mathbf{S})$ using **heap scan**
 $\text{cost} = 10,000 + 2 \cdot 10,000 + 2 \cdot \text{NPages}(\mathbf{R}) + \text{NPages}(\mathbf{R}) = \underline{30,030}$
 3. **block nested loops join**, $\text{opt}_2(\mathbf{S})$ using **index scan**
 $\text{cost} = 10,500 + \text{NPages}(\mathbf{S}) \cdot \text{NPages}(\mathbf{R}) = \underline{110,500}$
 4. **merge join** (with two-way sort), $\text{opt}_2(\mathbf{S})$ using **index scan**
 $\text{cost} = 10,500 + 2 \cdot \text{NPages}(\mathbf{R}) + \text{NPages}(\mathbf{R}) = \underline{10,530}$
- $\text{opt}_i(\mathbf{S})$ with \mathbf{B} as inner relation needs **cross-product** and gets **pruned**



Note that $\text{opt}(\mathbf{R}, \mathbf{S})$ exploits an interesting order of a non-optimal sub-plan

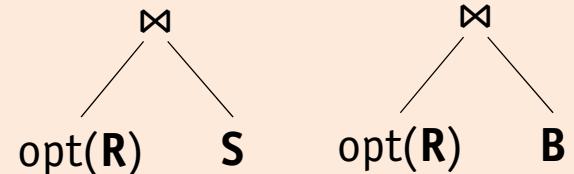
$\text{opt}_1(\mathbf{R}, \mathbf{S})$

Dynamic Programming

↷ Step-by-step example: Pass 2 (2-relation plans, cont'd)

Plans with $\text{opt}(R)$ as outer relation

- $\text{opt}(R)$ with S as inner relation
 1. **block nested loops join, heap scan on S**
 $\text{cost} = 10 + 10 \cdot NPages(S) = \underline{100,010}$
 2. **index nested loops join, index access on S**
 $\text{cost} = 10 + NTuples(R) \cdot (\text{height}(I) + 1) = \underline{410}$
 3. **merge join (with two-way sort), heap scan on S**
 $\text{cost} = 10 + NPages(S) + 2 \cdot (10 + NPages(S)) = \underline{30,300}$
 4. **merge join (with two-way sort), index scan on S**
 $\text{cost} = 10 + 2 \cdot 10 + 10,500 = \underline{10,530}$
- $\text{opt}(R)$ with B as inner relation
 5. **block nested loops join**
 $\text{cost} = 10 + 10 \cdot NPages(B) = \underline{110}$
 6. **merge join (with two-way sort)**
 $\text{cost} = 10 + NPages(B) + 2 \cdot (10 + NPages(B)) = \underline{60}$



$\leftarrow \text{opt}_2(R, S)$

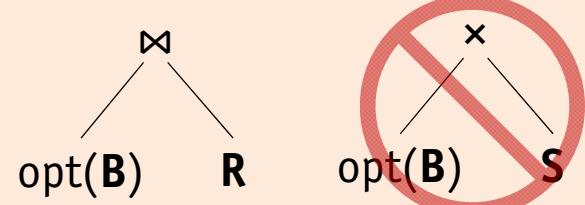
$\leftarrow \text{opt}_1(B, R)$

Dynamic Programming

↷ Step-by-step example: Pass 2 (2-relation plans, cont'd)

Plans with $\text{opt}(\mathbf{B})$ as outer relation

- $\text{opt}(\mathbf{B})$ with \mathbf{R} as inner relation
 1. **block nested loops join**
 $\text{cost} = 10 + 10 \cdot \text{NPages}(\mathbf{R}) = \underline{110}$
 2. **merge join** (with two-way sort)
 $\text{cost} = 10 + \text{NPages}(\mathbf{R}) + 2 \cdot (10 + \text{NPages}(\mathbf{R})) = \underline{110}$
- $\text{opt}(\mathbf{B})$ with \mathbf{S} as inner relation needs **cross-product** and gets **pruned**

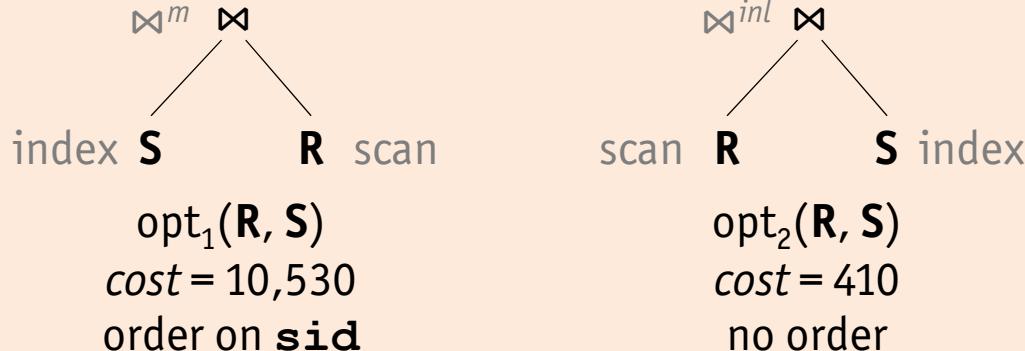


$\text{opt}_2(\mathbf{B}, \mathbf{R})$

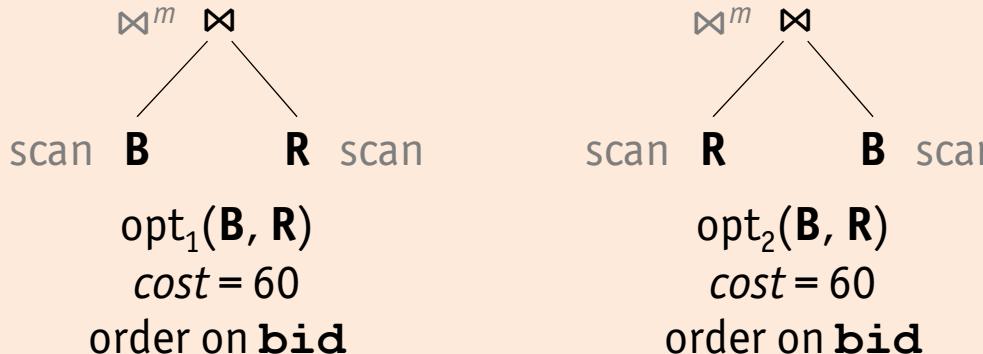
Dynamic Programming

↷ Step-by-step example: Summary of Pass 2

- $R \bowtie S$



- $B \bowtie R$



Plans $\text{opt}_2(R, S)$ and $\text{opt}_1(B, R)$ or $\text{opt}_2(B, R)$ are kept for the next pass. Note that the order in $\text{opt}_1(R, S)$ is **not** interesting for subsequent join(s).

Dynamic Programming

↷ Step-by-step example: Pass 3 (3-relation plans)

Plans with $\text{opt}(\mathbf{R}, \mathbf{S})$ as outer relation

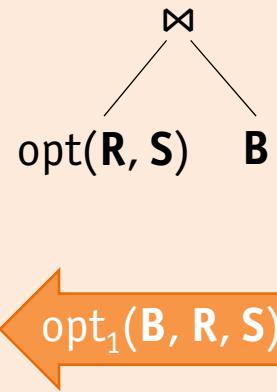
- $\text{opt}(\mathbf{R}, \mathbf{S})$ with \mathbf{B} as inner relation

1. block nested loops join

$$\text{cost} = 410 + \text{NPages}(\mathbf{R} \bowtie \mathbf{S}) \cdot \text{NPages}(\mathbf{B}) = \underline{510}$$

2. merge join (with two-way sort)

$$\text{cost} = 410 + \text{NPages}(\mathbf{B}) + 2 \cdot (\text{NPages}(\mathbf{R} \bowtie \mathbf{S}) + \text{NPages}(\mathbf{B})) = \underline{460}$$



Observe that $\mathbf{R}.\mathbf{s}\mathbf{i}\mathbf{d}$ is a foreign key pointing to $\mathbf{S}.\mathbf{s}\mathbf{i}\mathbf{d}$. Therefore, every tuple in \mathbf{R} matches **exactly one** tuple in \mathbf{S} . Since there are 100 \mathbf{R} -tuples, $NTuples(\mathbf{R} \bowtie \mathbf{S}) = 100$. As 10 $(\mathbf{R} \bowtie \mathbf{S})$ -tuples fit in one page, $\text{NPages}(\mathbf{R} \bowtie \mathbf{S}) = 10$.

Dynamic Programming

↷ Step-by-step example: Pass 3 (3-relation plans)

Plans with $\text{opt}(\mathbf{B}, \mathbf{R})$ as outer relation

- $\text{opt}(\mathbf{B}, \mathbf{R})$ with \mathbf{S} as inner relation

1. **block nested loops join, heap scan on \mathbf{S}**

$$\text{cost} = 60 + N\text{Pages}(\mathbf{B} \bowtie \mathbf{R}) \cdot N\text{Pages}(\mathbf{S}) = \underline{100,060}$$

2. **index nested loops join, index access on \mathbf{S}**

$$\text{cost} = 60 + N\text{Tuples}(\mathbf{B} \bowtie \mathbf{R}) \cdot (\text{height}(\mathbf{I}) + 1) = \underline{460}$$

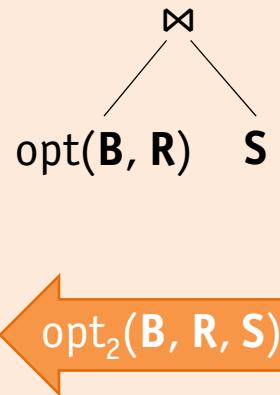
3. **merge join (with two-way sort), heap scan on \mathbf{S}**

$$\text{cost} = 60 + N\text{Pages}(\mathbf{S}) + 2 \cdot (N\text{Pages}(\mathbf{B} \bowtie \mathbf{R}) + N\text{Pages}(\mathbf{S})) = \underline{30,080}$$

4. **merge join (with two-way sort), index scan on \mathbf{S}**

$$\text{cost} = 60 + 2 \cdot N\text{Pages}(\mathbf{B} \bowtie \mathbf{R}) + 10,500 = \underline{10,580}$$

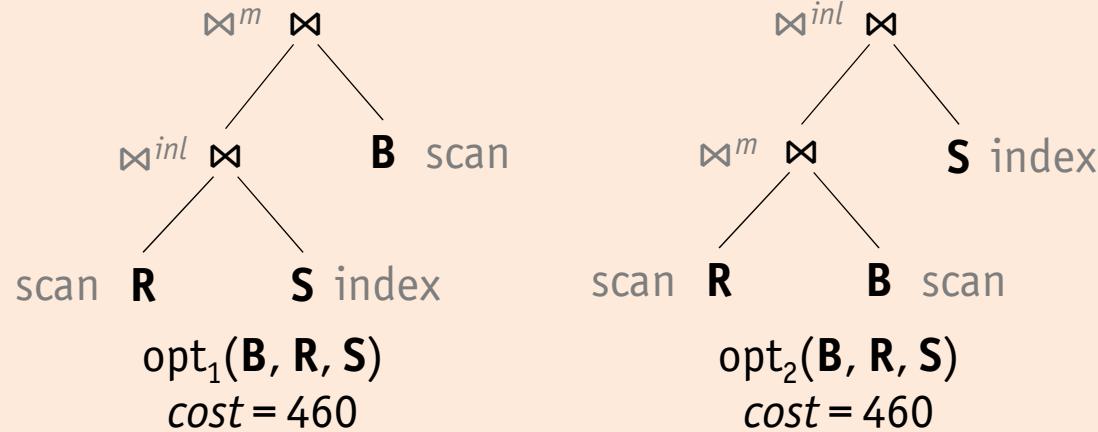
Observe that $\mathbf{R}.\mathbf{bid}$ is a foreign key pointing to $\mathbf{B}.\mathbf{bid}$. Therefore, every tuple in \mathbf{R} matches **exactly one** tuple in \mathbf{B} . Since there are 100 \mathbf{R} -tuples, $N\text{Tuples}(\mathbf{B} \bowtie \mathbf{R}) = 100$. As 10 ($\mathbf{B} \bowtie \mathbf{R}$)-tuples fit in one page, $N\text{Pages}(\mathbf{B} \bowtie \mathbf{R}) = 10$.



Dynamic Programming

↷ Step-by-step example: Summary of Pass 3

- $B \bowtie R \bowtie S$



Observations

- best plans mix join algorithms and exploits indexes
- cost of worst plan > 100,000 (exact cost unknown due to pruning)
- optimization yielded ≈ 1000 -fold improvement over the worst plan!

Dynamic Programming

Algorithm to find an optimal n -way left-deep join tree

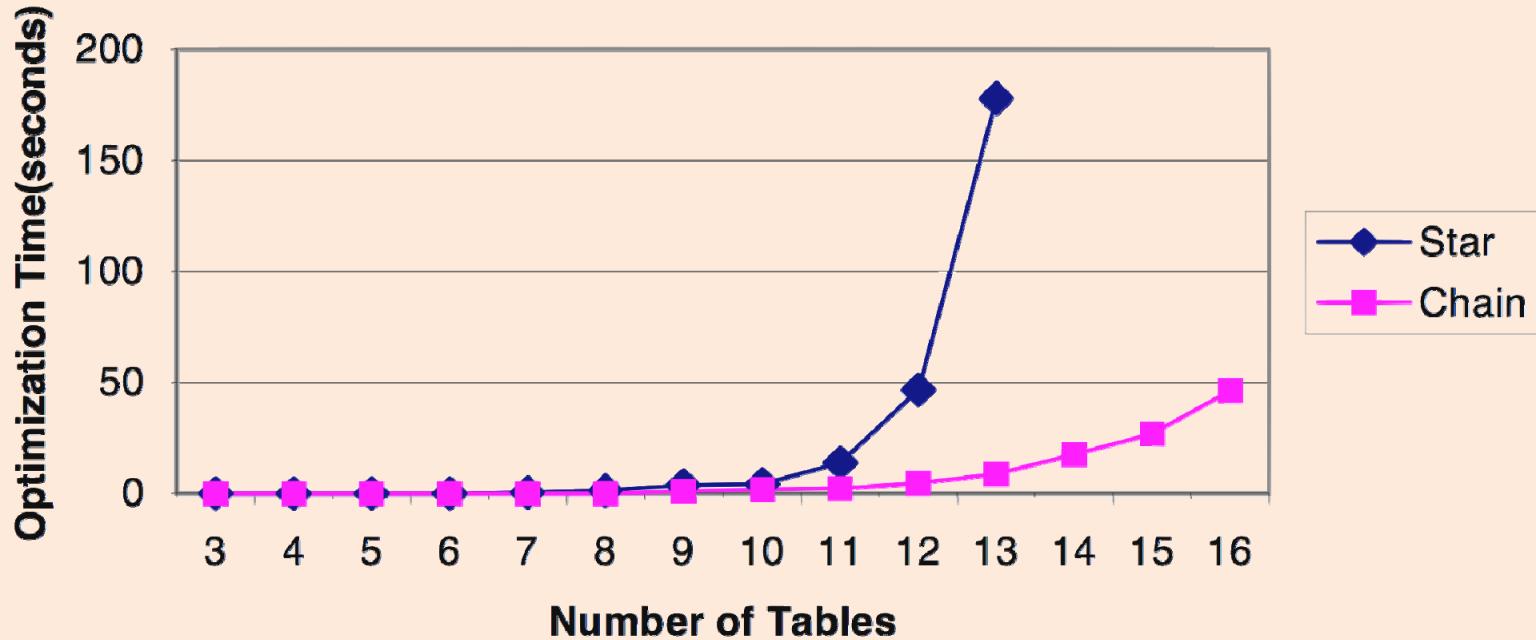
```
function optimize (q( $R_1, \dots, R_n$ ))
    for  $i = 1$  to  $n$  do                                (Pass 1)
         $optPlan(\{R_i\}) \leftarrow accessPlans(R_i)$  ;
        prunePlans ( $optPlan(\{R_i\})$ ) ;
    for  $i = 2$  to  $n$  do                                (Pass 2 to  $n$ )
        foreach  $S \subseteq \{R_1, \dots, R_n\}$  such that  $|S| = i$  do
             $optPlan(S) \leftarrow \emptyset$  ;
            foreach  $R \subseteq \{R_1, \dots, R_n\}$  do
                 $optPlan(S) \leftarrow optPlan(S) \cup possibleJoins \left( \begin{array}{c} \bowtie \\ optPlan(S \setminus R) \quad optPlan(R) \end{array} \right)$  ;
            prunePlans ( $optPlan(S)$ ) ;
    return  $optPlan(\{R_1, \dots, R_n\})$  ;
end
```

Dynamic Programming

- Subroutines in dynamic programming algorithm
 - **accessPlans** (R) enumerates all access plans for a single relation R
 - **possibleJoins** ($R \bowtie S$) enumerates the possible joins between relations R and S , e.g., nested loops join, sort-merge join, hash join, etc.
 - **prunePlans** (set) discards all but the best plans from set
- Function **optimize()** draws its advantage from **filtering** candidate plans early in the process
 - Pass 2 of the example keeps 2 out of 12 plans
 - Pass 3 of the example keeps 1 out of 6 plans
- Heuristics are used to reduce the search space and to balance **plan quality** and **optimizer runtime**

Joining Many Relations

Optimization time for chain and star queries in the Columbia optimizer



Joining Many Relations

- Join enumeration still has **exponential** resource requirements
 - time complexity: $O(3^n)$
 - space complexity: $O(2^n)$
- This approach may still be too expensive
 - for joins involving many relations (~10-20 and more)
 - for simple queries over well-indexed data (where the right plan choice should be easy to make)
- The **greedy join enumeration** algorithm aims to close this gap

Joining Many Relations

(Grid icon) Greedy join enumeration algorithm for an n -way join

```
function optimize-greedy (q( $R_1, \dots, R_n$ ))
    worklist  $\leftarrow \emptyset$ ;
    for  $i = 1$  to  $n$  do
        worklist  $\leftarrow$  worklist  $\cup$  bestAccessPlans ( $R_i$ ) ;
    for  $i = n$  downto 2 do
        find  $P_j, P_k \in$  worklist  $\wedge \bowtie^*$  such that cost ( $P_j \bowtie^* P_k$ ) is minimal;
        worklist  $\leftarrow$  worklist  $\setminus \{P_j, P_k\} \cup \{(P_j \bowtie^* P_k)\}$ ;
    return single plan left in worklist;
end
```

$(worklist = \{P_1, \dots, P_i\})$

$(worklist = \{P_1\})$

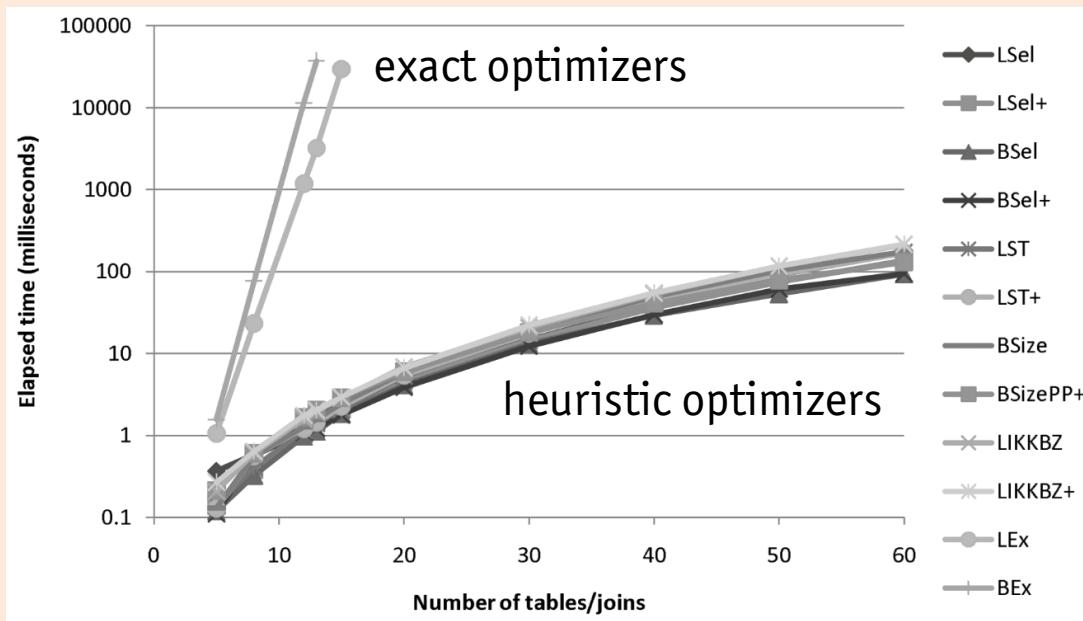
- In each iteration, choose the **cheapest** join that can be made over the remaining sub-plans at that time (greedy part)
- Observe that **optimize-greedy ()** operates similar to finding the optimum binary tree for **Huffman coding**

Joining Many Relations

- Greedy join enumeration algorithm has $O(n^3)$ time complexity
 - each loop has n iterations: $O(n)$
 - each iteration looks at all remaining pairs of plans in *worklist*: $O(n^2)$
- Greedy algorithm considers all types of join trees
 - balances risk of returning a really bad plan
 - opens opportunities for additional (rewrite) heuristics
- There are also other join enumeration techniques
 - **randomized algorithms**: randomly rewrite the join tree on rewrite as a time, using **hill-climbing** or **simulated annealing** strategy to find optimal plan
 - **genetic algorithms**: explore plan space by **combining** plans (“offspring”) and **altering** some plans randomly (“mutations”)

Joining Many Relations

Performance of different optimization alternatives



- Naming scheme
 - Search space: **L** (linear) or **B** (bushy)
 - Ranking: **Sel** (*minSel*), **Size** (*minSize*), or **ST** (*smallestTable*)
 - Merging: **+** (*Switch-HJ* and *Switch-Idx*) or **PP** (*Pull/Push*)

Cardinality Estimation

- Recall the **cost model** for single-relation plans used so far
- Cost estimation of a multi-relation plan additionally involves **cardinality estimation** of intermediate query results
 - cost of a plan is dominated by page I/O operations
 - cost of implementation algorithms is determined by size of inputs
- Cardinality estimates are also used to allocate buffer pages or to determine blocking factor b for blocked I/O

Cost model for access methods on relation R

Access method

access primary index \mathbf{I}

clustered index \mathbf{I} matching predicate p

unclustered index \mathbf{I} matching predicate p

sequential scan

Cost

$height(\mathbf{I}) + 1$	if \mathbf{I} is B+ tree
$1.2 + 1$	if \mathbf{I} is hash index
$(NPages(\mathbf{I}) + NPages(\mathbf{R})) \cdot sel(p)$	
$(NPages(\mathbf{I}) + NTuples(\mathbf{R})) \cdot sel(p)$	
$NPages(\mathbf{R})$	

Cardinality Estimation

- A **database profile** is one of two principal approaches to query result cardinality estimation
 - **base relations**: maintain statistical information, e.g., number and sizes of tuples, distribution of attribute values, etc. in database catalog
 - **intermediate query results**: derive this information based on a simple statistical model during query optimization
- Remarks
 - statistical model typically assumes **uniformity** and **independence**
 - both are typically **not valid**, but they allow for simple calculations
 - system can record **histograms** that approximate value distributions more accurately in order to provide better cardinality estimations

Cardinality Estimation

- Alternatively, **sampling techniques** can be used to estimate query result set cardinality
 - run query on a **small sample** of the database
 - **gather necessary statistics** about query plan
 - **extrapolate** to full input size at query execution time
- Remarks
 - it is crucial to find the right balance between sample size (performance) and the resulting accuracy of estimation

Simple Database Profiles

- Keep profile information in the database catalog
 - update information whenever database updates are issued
 - derive a (**very simple**) statistical model using **primitive assumptions**

Typical database profile for relation R

$NTuples(R)$	number of tuples in relation R
$NPages(R)$	number of disk pages allocated for relation R
$s(R)$	average record size (width) of relation R
b	block size, alternative to $s(R)$ $NPages(N) = NTuples(R) / [^b / s(R)]$
$V(A, R)$	number of distinct values of attribute A in relation R
$High(A, R)/Low(A, R)$	maximum and minimum value of attribute A in relation R
$MCV(A, R)$	most common value(s) of attribute A in relation R
$MVF(A, R)$	frequency of most common value(s) of attribute A in relation R
:	<i>possibly many more</i>

Simple Database Profiles

- In order to obtain a simple and tractable cardinality estimation formulae, assume **one of the following**

Assumptions

1. Uniformity and independence assumption

All values of an attribute uniformly appear with the same probability (or even distribution). Values of different attributes are independent of each other.

↳ *Simple, yet rarely realistic assumption*

2. Worst case assumption

No knowledge about relation contents available at all. In case of a selection σ_p , assume that all records will satisfy predicate p .

↳ *Unrealistic assumption, can only be used for computing upper bounds*

3. Perfect knowledge assumption

Details about the exact distribution of values are known. Requires huge catalog or prior knowledge of incoming queries.

↳ *Unrealistic assumption, can only be used for computing lower bounds*

Simple Database Profiles

- Find formulae describing properties of intermediate query results for all (logical) operators
 - express formulae in terms of profile information
 - database systems typically assume uniformity and independence

↷ Selection query $Q := \sigma_{A=c}(R)$

Selectivity $sel(A = c)$

$$\begin{cases} MCF(A, R)[c] & \text{if } c \in MCF(A, R) \\ 1/V(A, R) & \text{(uniformity assumption)} \end{cases}$$

Cardinality $|Q|$

$$sel(A = c) \cdot NTuples(R)$$

Record size $s(Q)$

$$s(R)$$

Number of attribute values $V(A', Q)$

$$\begin{cases} 1, & \text{for } A' = A \\ c(|R|, V(A, R), |Q|) & \text{otherwise} \end{cases}$$

Simple Database Profiles

- Number $c(|R|, V(A, R), |Q|)$ of distinct values in attribute A' after selection, is estimated using a well-known statistics formula
- Number c of distinct colors obtained by drawing r balls from a bag of n balls in m different colors is $c(n, m, r)$

$$c(n, m, r) = \begin{cases} r, & \text{for } r < \frac{m}{2} \\ \frac{r + m}{3}, & \text{for } \frac{m}{2} \leq r < 2m \\ m, & \text{for } r \geq 2m \end{cases}$$

Simple Database Profiles

↷ Selection query $Q := \sigma_{A=B}(R)$

Equality between attributes, e.g., $\sigma_{A=B}(R)$ can be approximated by

$$sel(A = B) = 1 / \max(V(A, R), V(B, R))$$

This formula assumes that each value of the attribute with fewer distinct values has a corresponding match in the other attribute (**independence assumption**).

↷ Selection query $Q := \sigma_{A>c}(R)$

If $Low(A, R) \leq c \leq High(A, R)$, range selections, e.g., $\sigma_{A>c}(R)$ can be approximated by

$$sel(A > c) = \frac{High(A, R) - c}{High(A, R) - Low(A, R)}$$

This formula uses the **uniformity assumption**.

↷ Selection query $Q := \sigma_{A \text{ IN } (...)}(R)$

Element tests, e.g., $\sigma_{A \text{ IN } (...)}(R)$ can be approximated by multiplying the selectivity for an equality selection $sel(A = c)$ with the number of elements in the list of values.

Simple Database Profiles

- Estimating the number of result tuples of a **projection** query is difficult due to effect of duplicate elimination

Projection query $Q := \pi_L(R)$

Cardinality $|Q|$

$$\begin{cases} V(\mathbf{A}, R), & \text{for } L = \{\mathbf{A}\} \\ |R| & \text{if keys of } R \in L \\ |R| & \text{no duplicate elimination} \\ \min(|R|, \prod_{\mathbf{A}_i \in L} V(\mathbf{A}_i, R)) & \text{otherwise} \end{cases}$$

Record size $s(Q)$

$$\sum_{\mathbf{A}_i \in L} s(\mathbf{A}_i)$$

Number of attribute values $V(\mathbf{A}_i, Q) \quad V(\mathbf{A}_i, R)$

for $\mathbf{A}_i \in L$

Simple Database Profiles

↷ Union query $Q := R \cup S$

$$\begin{aligned}|Q| &\leq |R| + |S| \\ s(Q) &= s(R) = s(S) \\ V(\mathbf{A}, Q) &\leq V(\mathbf{A}, R) + V(\mathbf{A}, S)\end{aligned}$$

$$sch(R) = sch(S)$$

↷ Difference query $Q := R - S$

$$\begin{aligned}\max(0, |R| - |S|) &\leq |Q| \leq |R| \\ s(Q) &= s(R) = s(S) \\ V(\mathbf{A}, Q) &\leq V(\mathbf{A}, R)\end{aligned}$$

↷ Cross-product query $Q := R \times S$

$$\begin{aligned}|Q| &= |R| \cdot |S| \\ s(Q) &= s(R) + s(S) \\ V(\mathbf{A}, Q) &= \begin{cases} V(\mathbf{A}, R), & \text{if } \mathbf{A} \in sch(R) \\ V(\mathbf{A}, S), & \text{if } \mathbf{A} \in sch(S) \end{cases}\end{aligned}$$

Simple Database Profiles

- Cardinality estimation for the general **join** case is challenging
 - there are, however, a few special simple cases
 - a very common simple case is the **foreign key relationship**

☞ Special cases of join queries $Q := R \bowtie_p S$

- no common attributes ($sch(R) \cap sch(S) = \emptyset$) or join predicate $p = \text{true}$
$$R \bowtie_p S = R \times S$$
- join attribute, say **A**, is key in one of the relations, e.g., in **R**, and assuming the inclusion dependency $\pi_A(S) \subseteq \pi_A(R)$

$$|Q| = |R|$$

☞ this inclusion dependency is guaranteed by a foreign key relationship between **R.A** and **S.A** in $R \bowtie_{R.A=S.A} S$.

Simple Database Profiles

General join queries $Q := R \bowtie_{R.A=S.B} S$

Assuming inclusion dependencies, the cardinality of a general join query Q can be estimated as

Cardinality $|Q|$

$$\begin{cases} \frac{|R| \cdot |S|}{V(A, R)}, & \text{for } \pi_B(S) \subseteq \pi_A(R) \\ \frac{|R| \cdot |S|}{V(B, S)}, & \text{for } \pi_A(R) \subseteq \pi_B(S) \end{cases}$$

Typically, the smaller of these two estimates is used

Cardinality $|Q|$

$$\frac{|R| \cdot |S|}{\max(V(A, R), V(B, S))}$$

Record size $s(Q)$ $s(R) + s(S) - \sum s(A_i)$ for all common A_i of a natural join

Number of attribute values $V(A', Q)$ $\leq \begin{cases} \min(V(A', R), V(A', S)), & A' \in sch(R) \cap sch(S) \\ V(A', X), & A' \in sch(X) \end{cases}$

Simple Database Profiles

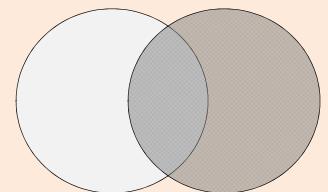
- Estimating selectivity of selections with **composite predicates**
 - compute selectivity of each individual condition separately
 - combine results under the independence assumption

☛ Selections with composite predicates

- **conjunctive** predicates, e.g., $Q := \sigma_{A=c_1 \wedge B=c_2}(R)$

$$sel(A = c_1 \wedge B = c_2) = sel(A = c_1) \cdot sel(B = c_2)$$

which gives $|Q| = \frac{|R|}{V(A, R) \cdot V(B, R)}$



- **disjunctive** predicates, e.g., $Q := \sigma_{A=c_1 \vee B=c_2}(R)$

$$sel(A = c_1 \vee B = c_2) = sel(A = c_1) + sel(B = c_2) - sel(A = c_1) \cdot sel(B = c_2)$$

which gives $|Q| = \frac{|R|}{V(A, R) + V(B, R) - V(A, R) \cdot V(B, R)}$

Histograms

- In realistic database instances, values are **not uniformly distributed** across the active domain of an attribute
- To keep track of non-uniform value distribution of attribute **A**, maintain a **histogram** to approximate the actual distribution
 1. divide the active domain of **A** into **adjacent intervals** (so-called **buckets**) by selecting boundary values $b_i \in \text{dom}(A)$
 2. collect **statistical parameters** for each interval such as the number of tuples $b_{i-1} < t[A] \leq b_i$ or the number of distinct **A**-values in that interval
- Histograms allow for more exact estimates of both **equality** and **range selections**

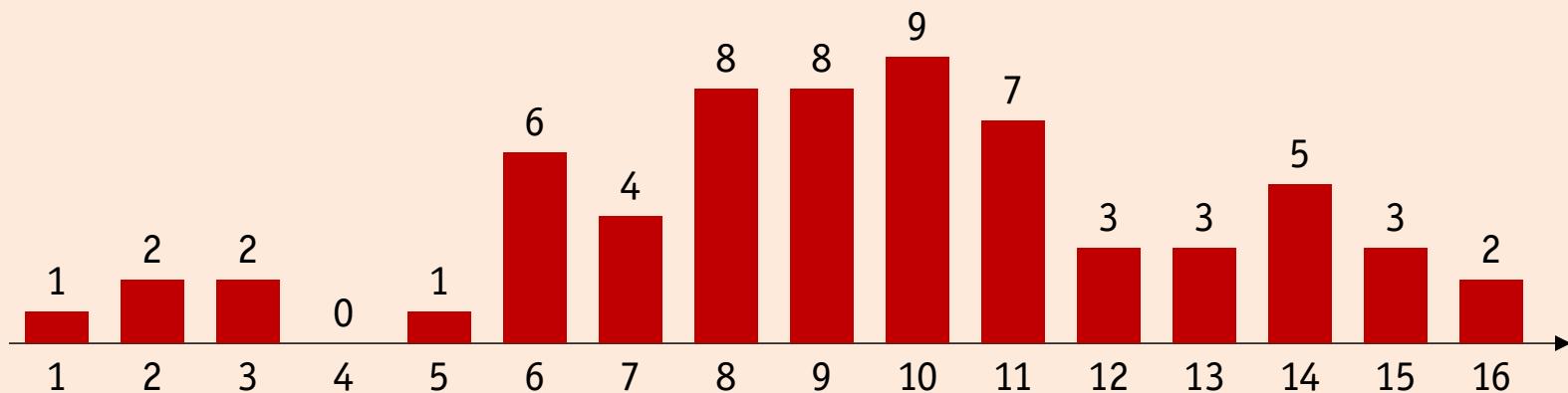
Histograms

- Two types of histograms are widely used
 1. **Equi-Width Histograms**
All histogram buckets have the **same width**, i.e., boundary $b_i = b_i + w$, for some fixed width w
 2. **Equi-Depth Histograms**
All histogram buckets contain the **same number of tuples**, i.e., their width is varying
- Equi-depth histograms are better able to adapt to data skew (high uniformity)
- Number of histogram buckets can be used to control the **trade-off** between
 - **histogram resolution**, which defines estimation quality
 - **histogram size** (space in database catalog is limited)

Histograms

Example: actual value distribution

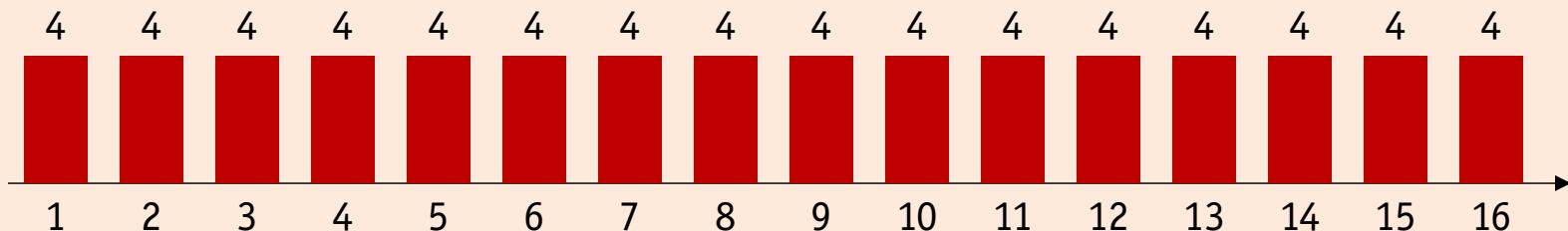
Assume a column **A** of SQL type **INTEGER** (domain $\{\dots, -2, -1, 0, 1, 2, \dots\}$) with the following actual non-uniform value distribution in a relation **R**.



Histograms

Example: uniform distribution assumption

For skewed, i.e., non-uniform, data distributions, working without histograms gives **bad** estimates. Working under the uniformity assumption, the value distribution from the previous slide is approximated as follows.



As a consequence, the selectivity $\text{sel}(\mathbf{A} < 6)$ would be computed as $64/16 \cdot 5 = 20$, which is **far from correct**. The actual number of tuples that qualify is 6.

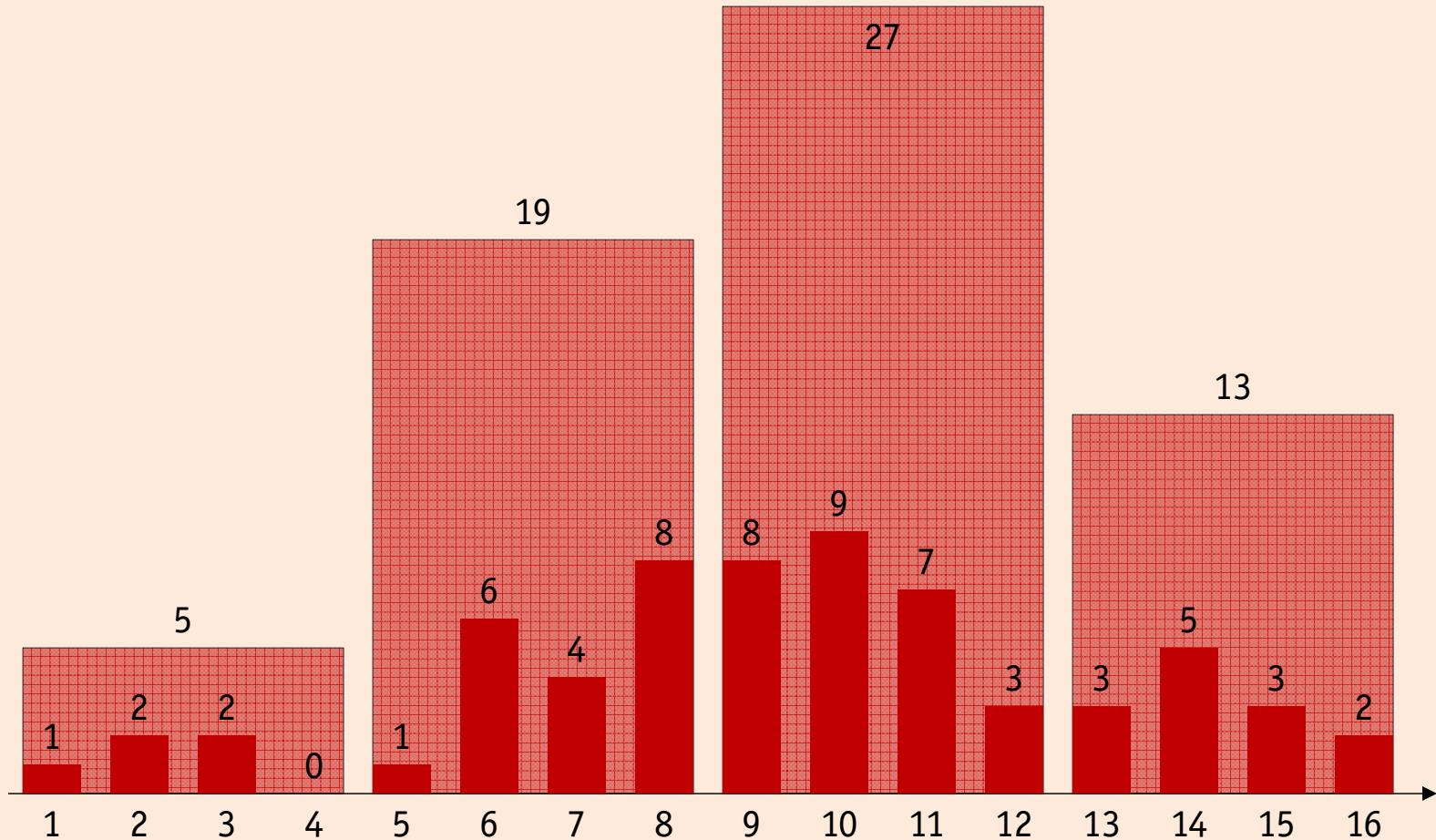
The error in estimation using the uniformity assumption is especially large for those values, which occur often in the database. This situation is particularly bad, since for those values the best estimates are actually required.

Histograms

- Construct an **equi-width histogram** on attribute **A** of relation **R**
 1. divide **active domain** of attribute **A** into B buckets of equal **bucket width** w , such that
$$w = \frac{\text{High}(\mathbf{A}, \mathbf{R}) - \text{Low}(\mathbf{A}, \mathbf{R}) + 1}{B}$$
 2. bucket boundary $b_0 = 0$ and $b_i = b_{i-1} + w$
 3. while scanning **R** once sequentially, maintain B **running sums of value frequencies**, one for each bucket
- If scanning **R** is prohibitive, scan sample $\mathbf{R}_{\text{sample}} \subseteq \mathbf{R}$, then scale counters by $|\mathbf{R}|/|\mathbf{R}_{\text{sample}}|$
- To maintain histogram under insertions and deletions, simply increment or decrement frequency counter in affected bucket
- To estimate result cardinality, use uniformity assumption **within each bucket**

Histograms

R Example: equi-width histogram ($B = 4$)



Histograms

Cardinality estimates with equi-width histograms

What are the cardinality estimates for the following selections based on this **equi-width histogram**? How do they compare to the actual value and the value estimated under the uniformity assumption?

Equality selection query $Q := \sigma_{A=5}(R)$

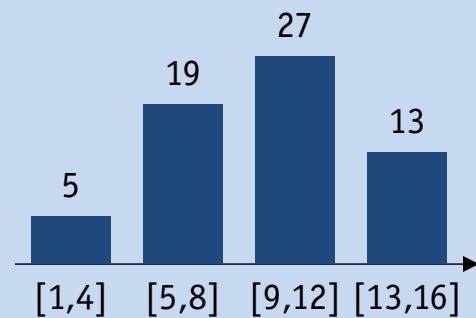
- **equi-width histogram:** $|Q| = 19/4 \approx 5$
- **uniformity assumption:** $|Q| = 4$
- **actual value:** $|Q| = 1$

Range selection query $Q := \sigma_{A < 6}(R)$

- **equi-width histogram:** $|Q| = 5 + 19/4 \approx 10$
- **uniformity assumption:** $|Q| = 5 \cdot 4 = 20$
- **actual value:** $|Q| = 6$

Range selection query $Q := \sigma_{A > 6 \wedge A < 14}(R)$

- **equi-width histogram:** $|Q| = 19/2 + 27 + 13/4 \approx 40$
- **uniformity assumption:** $|Q| = 7 \cdot 4 = 28$
- **actual value:** $|Q| = 42$



Histograms

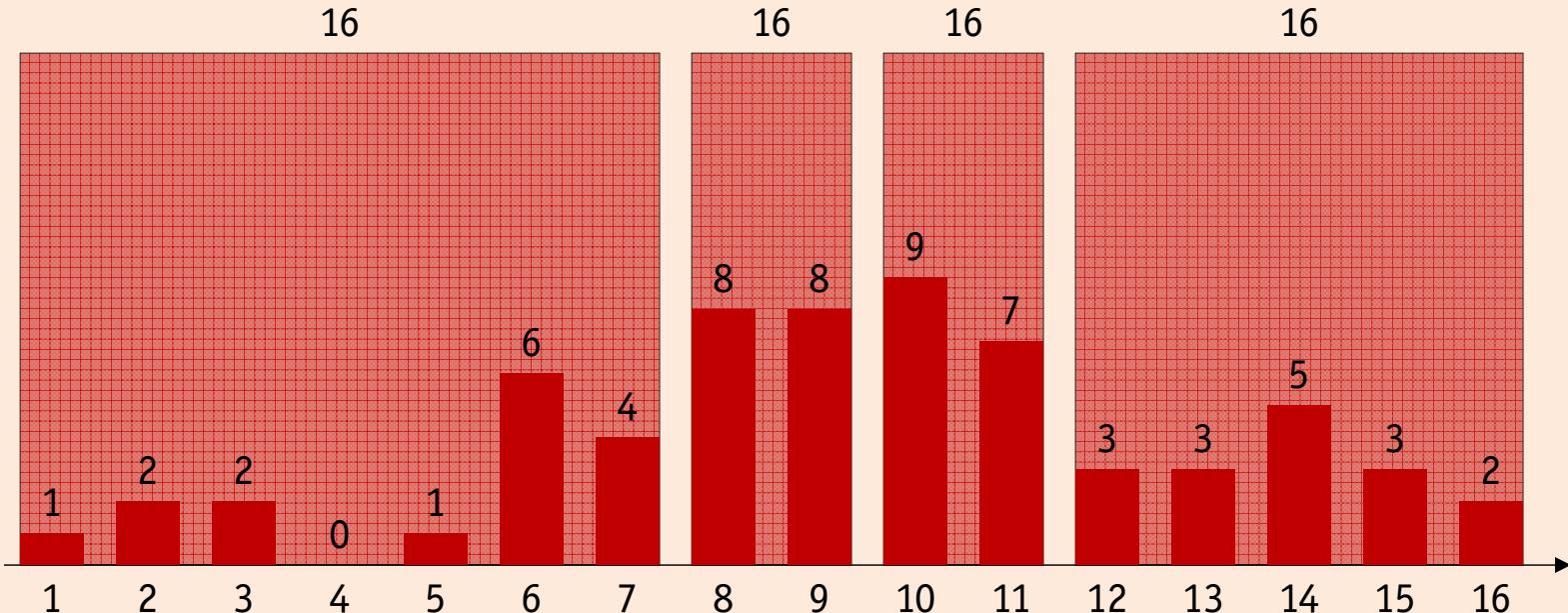
- In order to construct an **equi-depth histogram** on attribute **A** in relation **R**
 1. divide **active domain** of attribute **A** into B buckets of equal **bucket depth** d , such that $d = NTuples(\mathbf{R})/B$
 2. sort **R** by sort criterion on attribute **A**, e.g., natural order
 3. bucket boundary $b_0 = Low(\mathbf{A}, \mathbf{R})$, then determine b_i by dividing the sorted relation **R** into blocks of size $\sim d$

Example: equi-width histogram ($B = 4$, $|R| = 64$)

1. $d = 64/4 = 16$
2. sorted relation **R** (on attribute **A**)
 $\{1,2,2,3,3,5,6,6,6,6,6,6,7,7,7,7,8,8,8,8,8,8,8,9,9,9,9,9,9,9,9,10,10, \dots\}$
3. boundaries of d -sized blocks in sorted relation **R**
$$\underbrace{\{1,2,2,3,3,5,6,6,6,6,6,6,7,7,7,7,8,8,8,8,8,8,8,8,9,9,9,9,9,9,9,9,10,10, \dots\}}_{b_1 = 7} \quad \underbrace{\{8,8,8,8,8,8,8,8,8,8,9,9,9,9,9,9,9,9,10,10, \dots\}}_{b_2 = 9}$$

Histograms

R Example: equi-width histogram ($B = 4, d = 16$)

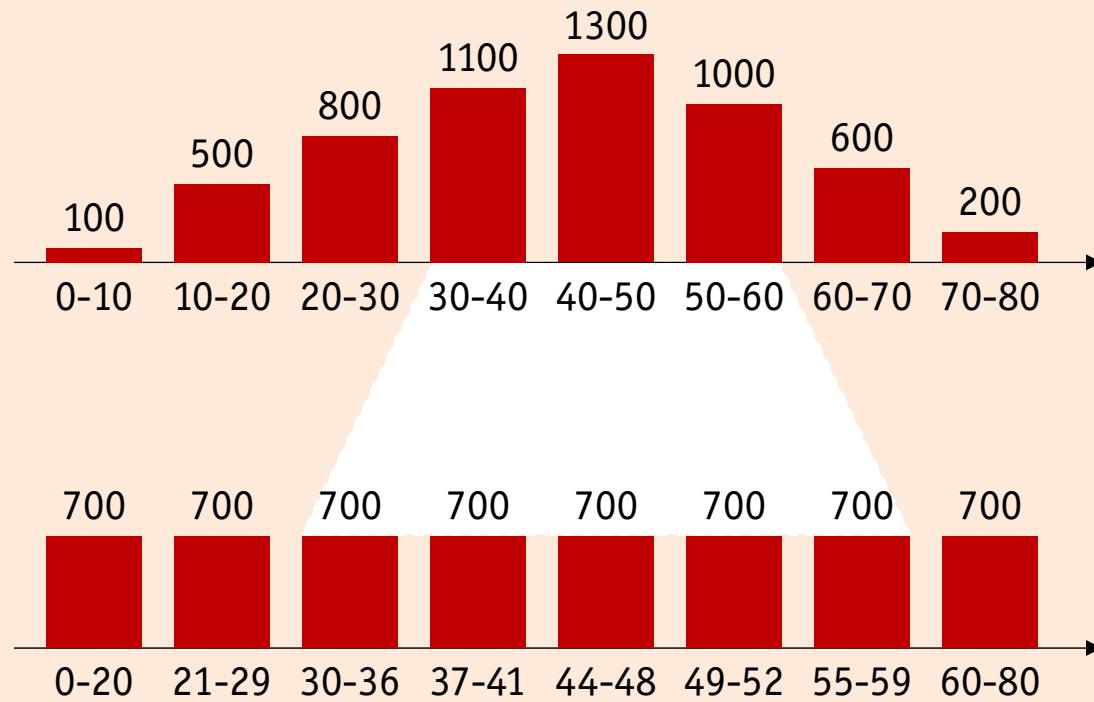


- Intuition
 - high value frequencies are **more important** than low frequencies
 - resolution of histogram **adapts** to skewed value distributions

Histograms

Example: equi-width vs. equi-depth histogram on person age ($B = 8$, $|R| = 5600$)

Equi-depth histogram “invests” bytes in the densely populated person age region between 30 and 59



Histograms

Cardinality estimates with equi-width histograms

What are the cardinality estimates for the following selections based on this **equi-depth histogram**? How do they compare to the actual value and the value estimated using the equi-width histogram?

Equality selection query $Q := \sigma_{A=5}(R)$

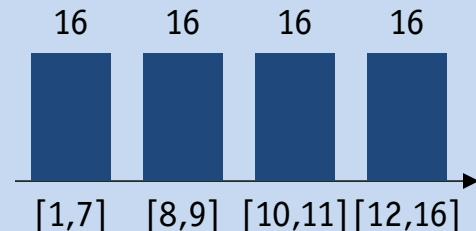
- **equi-depth histogram:** $|Q| = 16/7 \approx 2$
- **equi-width histogram:** $|Q| \approx 5$
- **actual value:** $|Q| = 1$

Range selection query $Q := \sigma_{A < 6}(R)$

- **equi-depth histogram:** $|Q| = 16/7 \cdot 5 \approx 11$
- **equi-width histogram:** $|Q| \approx 10$
- **actual value:** $|Q| = 6$

Range selection query $Q := \sigma_{A > 6 \wedge A < 14}(R)$

- **equi-depth histogram:** $|Q| = 16/7 + 16 + 16 + 16/5 \cdot 2 \approx 41$
- **equi-width histogram:** $|Q| = 43$
- **actual value:** $|Q| = 42$



Sampling

- Maintaining a database profile can be **costly** and **error-prone**
 - frequent database updates introduce overhead
 - provided estimates may be far-off, in particular for complex queries
- In such cases, it might be better to use **sampling** instead
 1. run complex query on a small sample of the data to collect statistics
 2. extrapolate statistics to full data set and optimize query accordingly
- Parameters of sampling
 - **sample size**: small enough to be executed efficiently, large enough to obtain useful characteristics
 - **extrapolation precision**: good predictions require a large (enough) sample
- Select a value for either one of those parameters and derive the other one

Sampling

- Three approaches are typically found in the literature
 - **adaptive sampling** tries to achieve a given precision with minimal sample size
 - **double (two-phase) sampling** obtains a coarse picture from a very small sample first and then computes the necessary sample size for a “useful” sample in a second step
 - **sequential sampling:** uses a sliding (continuous) calculation of characteristics and stops once the estimated precision is high enough

Nested Subqueries

- Recall, **unit of optimization** in typical systems is a query block
 - a query with nested subqueries consists of **multiple** query blocks
 - focus on optimization of nested **subqueries in the WHERE clause**
- Optimizer has several options to deal with such queries
 - evaluate subquery **once**, reuse result for all tuples of top-level query
 - evaluate subquery **multiple times**, once for every tuple of top-level query
 - try to **rewrite** query to an equivalent query without nesting
- Available optimization options depend on
 - form of **WHERE predicate**
 - presence or absence of **aggregates** in subquery
 - whether subquery is **uncorrelated** or **correlated**
 - other characteristics

Nested Subqueries

Canonical form

A SQL query is in **canonical form** if its **FROM** clause only contains table names and its **WHERE** clause only contains simple and join predicates.

Uncorrelated and correlated subqueries

A nested subquery is said to **correlated** if it references a table or tuple variable of the top-level query. Otherwise, the nested subquery is said to be **uncorrelated**.

Uncorrelated subquery

```
SELECT *  
  FROM A  
 WHERE A.a IN  
       {  
         nested or  
         inner query  
         {  
           (SELECT B.a  
            FROM B  
            WHERE B.b = 1)  
         }  
       }  
     } top-level or  
     } outer query
```

Correlated subquery

```
SELECT *  
  FROM A  
 WHERE A.a IN  
       {  
         correlation  
         (SELECT B.a  
          FROM B  
          WHERE B.b = A.b)  
       }
```

Nested Subqueries

- Types of nesting
 - **Type A:** uncorrelated, aggregated subquery
 - **Type N:** uncorrelated, not aggregated subquery
 - **Type J:** correlated, not aggregated subquery
 - **Type JA:** correlated, aggregated subquery
 - **Type D:** correlated, division subquery
- Due to the removal of the **CONTAINS** keyword in SQL-2 and higher, Type D nesting is no longer possible

Nested Subqueries

💻 Extended example database schema

```
CREATE TABLE Sailors (
    sid      INTEGER,
    sname    STRING,
    rating   INTEGER,
    city     STRING,
    PRIMARY KEY (sid)
)

CREATE TABLE Boats (
    bid      INTEGER,
    bname    STRING,
    color    STRING,
    qty      INTEGER,
    harbor   STRING,
    PRIMARY KEY (bid)
)

CREATE TABLE Reserves (
    sid      INTEGER,
    bid      INTEGER,
    qty      INTEGER,
    day     DATE,
    harbor   STRING,
    PRIMARY KEY (sid, bid),
    FOREIGN KEY sid REFERENCES Sailors(sid),
    FOREIGN KEY bid REFERENCES Boats(bis)
)
```

Nested Subqueries

R Example: Type A nesting

```
SELECT S.sname
  FROM Sailors S
 WHERE S.rating = (SELECT MAX(S2.rating)
                      FROM Sailors S2)
```

- Using **tuple iteration semantics** to evaluate this simple query is very inefficient
 - nested subquery is evaluated for **every tuple** of the top-level query
 - result of nested subquery is **constant**
- Better evaluation strategy
 - evaluate nested subquery just **once**, yielding a single value
 - rewrite top-level query to incorporate this value into its **WHERE** clause,
e.g., **S.rating = 8**

Nested Subqueries

R Example: Type N nesting

```
SELECT S.sname
  FROM Sailors S
 WHERE S.sid IN (SELECT R.sid
                   FROM Reserves R
                  WHERE R.bid = 103)
```

- Evaluation strategy
 - again, evaluate nested subquery just **once**, yielding a collection of **sids**
 - for each tuple of the top-level query, check if its **sid** value is in computed collection of **sids**, i.e., a join of **Sailors** and the computed collection
 - in principle, the full range of join methods is available, e.g., assuming an index on **Sailors.sid**, an index nested loops join could be used
- Typically, optimizers **cannot** find such strategies and revert to a nested loops join with computed collection as inner relation

Nested Subqueries

☛ Example: Type J nesting

```
SELECT S.sname
  FROM Sailors S
 WHERE EXISTS (SELECT *
                  FROM Reserves R
                 WHERE R.bid = 103 AND
                      S.sid = R.sid)
```

☛ Example: Type JA nesting

```
SELECT B.bname
  FROM Boats B
 WHERE B.bid = (SELECT MAX(R.bid)
                  FROM Reserves R
                 WHERE R.harbor = B.harbor)
```

Nested Subqueries

- Correlated subqueries cannot be evaluated just once
 - typical nested subquery is evaluated for each tuple of top-level query
 - this evaluation strategy is very inefficient
- Optimizer is **likely** to do a poor job using this limited approach to nested query optimization
 - if the same value appears in the correlation field of several tuples of the top-level query, the **same subquery** is evaluated multiple times
 - approach is **not set-oriented**, i.e., join method is effectively index nested loops (with evaluation of nested query as “index access”) and other join methods (sort-merge join, hash join) cannot be considered
 - even if index nested loops join is appropriate, optimizer **cannot leverage indexes** on relations of nested queries as there is no real join predicate
 - nesting imposes an **implicit ordering on query evaluation**, which leads to missed opportunities for finding a good evaluation plan

Nested Subqueries

- Try to rewrite queries with nested subqueries to **canonical form** in order to open up more optimization opportunities
 - instead of computed collections **use database tables directly**, if possible
 - otherwise, create **appropriate temporary tables** from nested subqueries
- Based on types of nested subqueries, two algorithms have been defined that follow this idea
 - algorithm **NEST-N-J**: transforms nested subqueries of Type N and J to a query in canonical form
 - algorithm **NEST-JA**: uses temporary tables to remove one level of nesting for subqueries of Type JA
- Subqueries of Type A are still dealt with as described before

Nested Subqueries

- Algorithm **NEST-N-J** for Type N or Type J nested subqueries
 1. combine **FROM** clauses of all query blocks into one **FROM** clause
 2. combine **WHERE** clauses of all query blocks using **AND**, replacing element test (**IN**) with equality (=)
 3. retain **SELECT** clause of the outermost query block
- Resulting canonical query is **equivalent** to original nested query

Algorithm NEST-N-J

```
SELECT Ri.Ck
  FROM Ri
 WHERE Ri.Cm IN
       (SELECT Rj.Cn
        FROM Rj
       WHERE Ri.Cx = Rj.Cy)
```



```
SELECT Ri.Ck
  FROM Ri, Rj
 WHERE Ri.Cm = Rj.Cn AND
       Ri.Cx = Rj.Cy
```

Nested Query

Rewriting Type N and Type J nested subqueries

Rewrite the following query with nested subqueries to a canonical query using algorithm NEST-N-J.

```
SELECT S.sname
  FROM Sailors S
 WHERE S.sid IN (
   SELECT R.sid
     FROM Reserves R
    WHERE S.city = R.harbor
      AND R.bid IN (
        SELECT B.bid
          FROM Boats B
         WHERE B.bname = "Laser"
      )
    )
```

```
SELECT S.name
  FROM Sailors S,
       Reserves R,
       Boats B
 WHERE S.sid = R.sid
   AND S.city = R.habor
   AND R.bid = B.bid
   AND B.bname = "Laser"
```

Nested Subqueries

- Assume R_1 is the relation in the top-level query and R_2 is the relation in the nested subquery
- Algorithm **NEST-JA** for Type JA nested subqueries
 1. create a temporary relation $R_{tmp}(C_1, \dots, C_n, C_{n+1})$ from relation R_2 , such that $R_{tmp}.C_{n+1}$ is the result of applying aggregate function **AGG** on the C_n columns of R_2 that have matching values for C_1, \dots, C_n in R_1
 2. transform inner query block of the original query by changing all references to R_2 columns in join predicates that also reference R_1 to corresponding R_{tmp} columns
- Result of algorithm NEST-JA is a Type J nested query that can be transformed to its canonical equivalent by algorithm NEST-N-J

Nested Subqueries

Algorithm NEST-JA (Step 1)

```
SELECT R1.Cn+2
  FROM R1
 WHERE R1.Cn+1 = (SELECT AGG(R2.Cn+1)
                           FROM R2
                          WHERE R1.C1 = R2.C1
                            AND R1.C2 = R2.C2
                            ...
                            AND R1.Cn = R2.Cn)
```



```
CREATE TABLE Rtmp(C1, ..., Cn, Cn+1) AS
  SELECT C1, ..., Cn, AGG(Cn+1)
    FROM R2
   GROUP BY C1, ..., Cn
```

Nested Subqueries

↷ Algorithm NEST-JA (Step 2)

```
SELECT R1.Cn+2
  FROM R1
 WHERE R1.Cn+1 = (SELECT Rtmp.Cn+1
                           FROM Rtmp
                          WHERE R1.C1 = Rtmp.C1
                            AND R1.C2 = Rtmp.C2
                            ...
                            AND R1.Cn = Rtmp.Cn)
```

⌚ The Real World

Many real-world RDBMS have functionality to support **temporary tables**.

↳ Microsoft SQL Server

- **local** temporary tables (name prefixed with #) are only visible in current session
- **global** temporary tables (name prefixed with ##) are visible to all sessions
- temporary tables are **automatically dropped** when they go out of scope

Nested Query

Rewriting Type JA nested subqueries

Rewrite the following query with a nested subquery to a canonical query using algorithm NEST-JA.

```
SELECT B.bname
  FROM Boats B
 WHERE B.bid = (SELECT MAX(R.sid)
                  FROM Reserves R
                 WHERE R.harbor = B.harbor)
```

```
CREATE TABLE TMP1 AS
  SELECT harbor,
         MAX(sid) AS maxsid
    FROM Reserves
   GROUP BY harbor
```

```
SELECT B.bname
  FROM Boats B
 WHERE B.bid = (
  SELECT T.maxsid
    FROM TMP1 T
   WHERE T.harbor = B.harbor
 )
```

Nested Subqueries

- Algorithm NEST-JA may give **incorrect results** for nested subqueries that contain the **COUNT** aggregation function

Example: Problem with COUNT

What is the result of the following query?

```
SELECT B.bid  
      FROM Boats B  
     WHERE B.qty =  
           (SELECT COUNT(R.day)  
            FROM Reserves R  
           WHERE B.bid = R.bid AND  
                 R.day <= 31-12-2010)
```

Result

bid
10
8

Boats

bid	bname	harbor	qty
3	Laser	Konstanz	6
10	Finn	Konstanz	1
8	49erFX	Kreuzlingen	0

Reserves

sid	bid	qty	day
2	3	4	07-03-2009
1	3	2	10-01-2008
1	10	1	06-08-2008
3	10	2	08-10-2011
4	8	5	05-07-2013

Nested Subqueries

Example: Problem with COUNT (cont'd)

Step 1 of algorithm NEST-JA creates the following temporary table.

```
CREATE TABLE TMP1 AS
  SELECT bid,
         COUNT(day) AS numday
    FROM Reserves
   WHERE day <= 31-12-2010
  GROUP BY bid
```

Reserves

sid	bid	qty	day
2	3	4	07-03-2009
1	3	2	10-01-2008
1	10	1	06-08-2008
3	10	2	08-10-2011
4	8	5	05-07-2013

Step 2 of algorithm NEST-JA rewrites the query as follows.

```
SELECT B.bid
  FROM Boats B
 WHERE B.qty =
  (SELECT T.numday
    FROM TMP1 T
   WHERE B.bid = T.bid)
```

TMP1	bid	numday
3		2
10		1

Result	bid
10	

Nested Subqueries

- For nested subqueries using the **COUNT** aggregation function, algorithm NEST-JA needs to include an **outer join** in step 1
- Full outer join **A \bowtie B** retains all records from **A** and **B**, even if there is no match, and inserts **NULL** values where necessary

☞ Full outer join (\bowtie)

The diagram illustrates the execution of a full outer join between two tables, A and B. On the left, table A has columns bid and bname, with rows (3, Laser), (10, Finn), (8, 49erFX), and (9, Narca 17). On the right, table B has columns bid, qty, and day, with rows (3, 4, 07-03-2009), (5, 1, 06-22-2008), (10, 2, 08-10-2011), and (8, 5, 05-07-2013). The join operation, indicated by the symbol \bowtie , results in a combined table on the right, which contains all rows from both tables, including matches and non-matches, with NULL values where necessary.

bid	bname	qty	day
3	Laser	4	07-03-2009
10	Finn	1	06-22-2008
8	49erFX	2	08-10-2011
9	Narca 17	5	05-07-2013

Nested Subqueries

☛ Using outer join to fix problem with COUNT

Adapted step 1 of algorithm NEST-JA creates the following temporary table.

```
CREATE TABLE TMP2 AS
  SELECT B.bid, COUNT(R.day) AS numday
    FROM Boats B FULL OUTER JOIN Reserves R
      ON B.bid = R.rid
     WHERE R.day <= 31-12-2010
   GROUP BY B.bid
```

Adapted step 2 of algorithm NEST-JA rewrites the query as follows.

```
SELECT B.bid
  FROM Boats B, TMP2 T
 WHERE B.qty = T.numday
   AND B.bid = T.bid
```

Nested Subqueries

☛ Using outer join to fix problem with COUNT

Boats

bid	bname	harbor	qty
3	Laser	Konstanz	6
10	Finn	Konstanz	1
8	49erFX	Kreuzlingen	0

Reserves

sid	bid	qty	day
2	3	4	07-03-2009
1	3	2	10-01-2008
1	10	1	06-08-2008
3	10	2	08-10-2011
4	8	5	05-07-2013

Boats $\bowtie \sigma_{day \leq 31-12-2010}$ (*Reserves*)

bid	bname	harbor	qty	sid	qty	day
3	Laser	Konstanz	6	2	4	07-03-2009
3	Laser	Konstanz	6	1	2	10-01-2008
10	Finn	Konstanz	1	1	1	06-08-2008
8	49erFX	Kreuzlingen	0	NULL	NULL	NULL

TMP2

bid	numday
3	2
10	1
8	0

Result

bid
10
8

Nested Subqueries

- Points to observe to obtain correct results when using outer join to deal with nested subqueries that contain **COUNT**
- Evaluation order
 - selection predicate, e.g., $day \leq 31-12-2010$, has to be applied **before** outer join
 - if necessary split creation of temporary table into two steps to force evaluation order
- Subqueries with **COUNT (*)**:
 - such subqueries will always return a result > 0 because of the outer join
 - Change ***** to a column name from relation of the nested subquery, typically the join column
- Duplicates in the join column
 - *see next slide*

Nested Subqueries

☛ Problem with duplicates in join column of outer join

Boats

bid	bname	harbor	qty
3	Laser	Konstanz	6
3	Laser	Kreuzlingen	2
10	Finn	Konstanz	1
10	Finn	Kreuzlingen	0
8	49erFX	Kreuzlingen	0

Reserves

sid	bid	qty	day
2	3	4	08-14-2007
1	3	2	11-11-2008
1	10	1	06-22-2006

Boats $\bowtie \sigma_{day \leq 31-12-2010}$ (Reserves)

bid	bname	harbor	qty	sid	qty	day
3	Laser	Konstanz	6	2	4	07-03-2009
3	Laser	Konstanz	6	1	2	10-01-2008
3	Laser	Kreuzlingen	2	2	4	07-03-2009
3	Laser	Kreuzlingen	2	1	2	10-01-2008
10	Finn	Konstanz	1	1	1	06-08-2008
10	Finn	Kreuzlingen	0	1	1	06-08-2008
8	49erFX	Kreuzlingen	0	NULL	NULL	NULL

TMP2

bid	numday
3	4
10	2
8	0

Result

bid
8

Actual Result

bid
3
10
8

Nested Subqueries

☛ Problem with duplicates in join column of outer join

Solution

1. During the creation of the temporary table, remove duplicates **before** computing the outer join.

```
CREATE TABLE TMP3 AS
    SELECT DISTINCT bid
    FROM Boats
```

TMP3
bid
3
10
8

2. In any join required to build the temporary table, use this projection **instead** of the original outer relation

```
CREATE TABLE TMP4 AS
    SELECT T.bid, COUNT(R.day) AS numday
    FROM TMP3 T FULL OUTER JOIN Reserves R
    ON T.bid = R.rid
    WHERE R.day <= 31-12-2010
    GROUP BY T.bid
```

TMP4	
bid	numday
3	2
10	1
8	0

Nested Subqueries

- Another problem arises with algorithm NEST-JA, if join predicate of nested subquery uses **inequality**

Example: Problem with inequality in join predicates of nested subquery

What is the result of the following query?

```
SELECT B.bid
  FROM Boats B
 WHERE B.qty =
    (SELECT MAX(R.qty)
      FROM Reserves R
     WHERE R.bid < B.bid AND
           R.day <= 31-12-2010)
```

Result

bid
8

Boats

bid	bname	harbor	qty
3	Laser	Konstanz	0
10	Finn	Konstanz	4
8	49erFX	Kreuzlingen	4

Reserves

sid	bid	qty	day
2	3	4	07-03-2009
1	3	2	10-01-2008
1	10	1	06-08-2008
3	9	5	03-02-2009

Nested Subqueries

Example: Problem with inequality in join predicates of nested subquery (cont'd)

Step 1 of algorithm NEST-JA creates the following temporary table.

```
CREATE TABLE TMP5 AS
  SELECT bid,
         MAX(qty) AS maxqty
    FROM Reserves
   WHERE day <= 31-12-2010
  GROUP BY bid
```

Step 2 of algorithm NEST-JA followed by application of algorithm NEST-N-J rewrites the query as follows.

```
SELECT B.bid
  FROM Boats B, TMP5 T
 WHERE B.qty = T.maxqty AND
       T.bid < B.bid
```

<i>Boats</i>	bid	bname	harbor	qty
3	Laser	Konstanz	0	
10	Finn	Konstanz	4	
8	49erFX	Kreuzlingen	4	

<i>Reserves</i>	sid	bid	qty	day
2	3	4	07-03-2009	
1	3	2	10-01-2008	
1	10	1	06-08-2008	
3	9	5	03-02-2009	

<i>TMP5</i>	bid	maxqty	<i>Result</i>	bid
3	4		10	
10	1		8	
9	5			

Nested Subqueries

- Reason for this problem
 - **GROUP BY** calculates the **MAX** for each distinct value of the join column of temporary relation (**b1d**)
 - **MAX** should be computed for a **set of join column values** in inner relation that are all smaller than a given join column value in outer relation
- Solution
 - **push** join predicate into computation of temporary relation, i.e., switch the evaluation order of join and group-by/aggregate
 - join is now performed **before** group-by/aggregate, causing temporary table to include aggregate value over proper ranges of join column values
 - as before, join predicate in outer query needs to be **changed to equality**

Nested Subqueries

☛ Problem with inequality in join predicates of nested subquery

Perform inequality join **before** group-by/aggregate.

```
CREATE TABLE TMP6 AS
  SELECT B.bid,
         MAX(R.qty) AS maxqty
    FROM Boats B, Reserves R
   WHERE R.day <= 31-12-2010
     AND R.bid < B.bid
 GROUP BY bid
```

Transform query to use temporary table in **equality** join.

```
SELECT B.bid
  FROM Boats B, TMP6 T
 WHERE B.qty = T.maxqty
   AND B.bid = T.bid
```

Boats

bid	bname	harbor	qty
3	Laser	Konstanz	0
10	Finn	Konstanz	4
8	49erFX	Kreuzlingen	4

Reserves

sid	bid	qty	day
2	3	4	07-03-2009
1	3	2	10-01-2008
1	10	1	06-08-2008
3	9	5	03-02-2009

TMP6

bid	maxqty
10	5
8	4

Result

bid
8

Nested Subqueries

- Assume R_1 is the relation in the top-level query and R_2 is the relation in the nested subquery
- Modified algorithm **NEST-JA2** for Type JA nested subqueries
 1. project join column of R_1 and restrict it with any simple predicates applying to R_1
 2. create temporary relation R_{tmp} by joining R_1 and R_2 using same operator as join predicate in original query
 - if aggregate function is **COUNT**, join must be outer join
 - if aggregate function is **COUNT (*)**, compute aggregate over join column
 - include join column(s) and aggregated column in **SELECT** clause
 - include join column(s) in **GROUP BY** clause
 3. join R_1 to R_{tmp} according to transformed version of original query by changing join predicate to equality (=)
- Result of algorithm NEST-JA2 is again a Type J nested query

Nested Subqueries

- So far, only nested predicates with **scalar** and **set inclusion (IN)** operators where considered
- SQL supports additional **set comparisons**
 - **EXISTS** and **NOT EXISTS**
 - **ANY** and **ALL**
- Extensions to transformation algorithms are required to support these comparisons
 - rewrite predicates using general set comparisons to predicates containing scalar or set inclusion operators only
 - process rewritten query using same algorithms as before

Nested Subqueries

☛ EXISTS and NOT EXISTS

... WHERE EXISTS
(SELECT A
FROM R
WHERE B = C)

... WHERE 0 <
(SELECT COUNT(A)
FROM R
WHERE B = C)

... WHERE NOT EXISTS
(SELECT A
FROM R
WHERE B = C)

... WHERE 0 =
(SELECT COUNT(A)
FROM R
WHERE B = C)



Nested Subqueries

ANY and ALL

```
... WHERE const < ANY  
(SELECT A  
  FROM R  
 WHERE B = C)
```



```
... WHERE const <  
(SELECT MAX(A)  
  FROM R  
 WHERE B = C)
```

And ...> ANY (SELECT A...) is transformed to ...> SELECT MIN(A) ..., for reasons of symmetry.

```
... WHERE const < ALL  
(SELECT A  
  FROM R  
 WHERE B = C)
```



```
... WHERE const <  
(SELECT MIN(A)  
  FEOM R  
 WHERE B = C)
```

And ...> ALL (SELECT A...) is transformed to ...> SELECT MAX(A) ..., for reasons of symmetry.

Finally, ...= ANY... is transformed to ...IN..., whereas ...<> ANY... is transformed to ...NOT IN....

Nested Subqueries

⌚ Recursive algorithm to process a general nested query

```
function nest-g (outer)                                (outer: outer query block)
  foreach p ∈ outer.WHERE do                         (p: predicate)
    if p.inner ≠ Ø then                               (p contains an inner query block, i.e., p is nested)
      nest-g (p.inner) ;
      if p.inner.SELECT contains aggregation function then
        if p.inner.WHERE is correlated then           (Type JA nesting)
          nest-ja2 (p.inner) ;
          nest-n-j (outer, p.inner) ;
        else                                         (Type A nesting)
          nest-a (p.inner) ;
      else                                         (Type N or Type J nesting)
        nest-n-j (outer, p.inner) ;
    end
```

Example Query Optimizers

“Query optimization is not rocket science.

When you flunk out of query optimization,
we make you go build rockets.”

– *Anonymous Quote*

Example Query Optimizers

- **System R Optimizer**
 - bottom-up
 - dynamic programming (plan generation)
 - interesting orders
- **Starburst**
 - bottom-up
 - rule-based (plan transformation and generation)
 - e.g., IBM DB2
- **Cascades**
 - top-down
 - rule-based (plan transformation)
 - e.g., Microsoft SQL Server

Example Query Optimizers

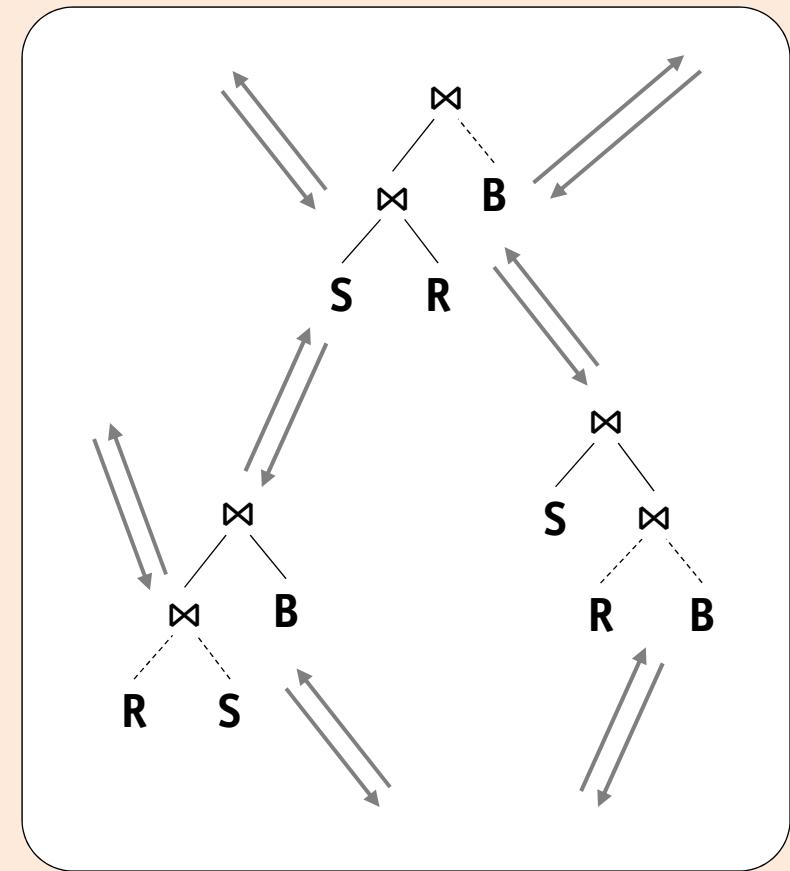
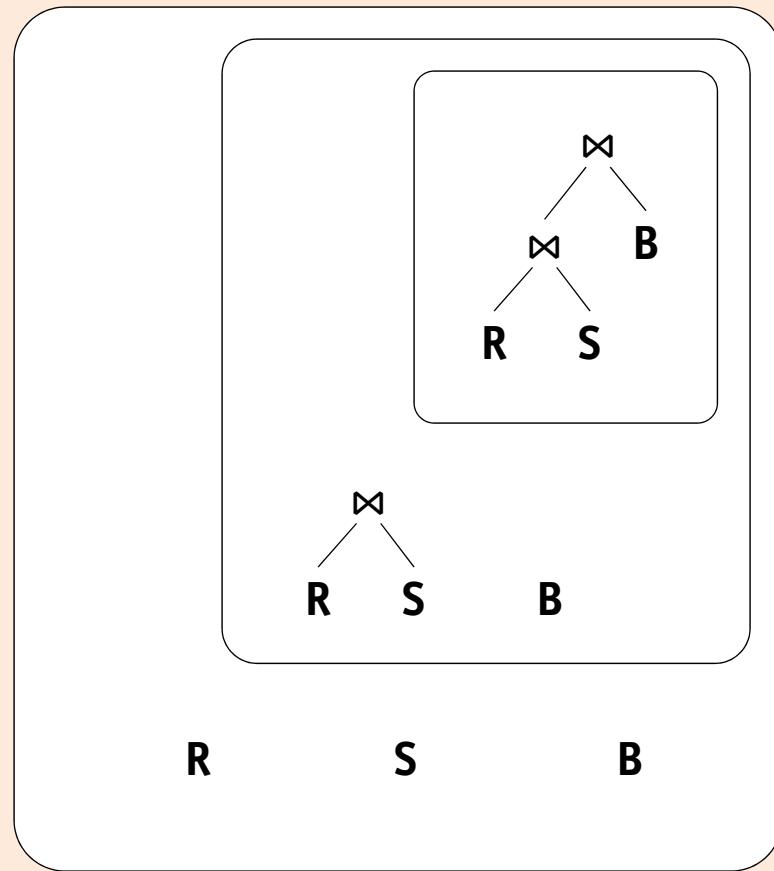
- Starting point of optimization process is one way to classify query optimizers
- **Bottom-up:** start with individual relations of query
 - global cost of a plan **must** be computed bottom-up as the cost of each operator depends on the cost of its input(s)
 - dynamic programming **requires** breadth-first enumeration to pick the best plan
 - impossible to pick best plan until its cost has been computed
- **Top-down:** start with entire expression of query
 - operators may **require** certain properties (e.g., order or partitioning)
 - limit exploration based upon context of use
 - prune based on upper and lower bound

Example Query Optimizers

- Another way to classify query optimizers is what approach is used for plan enumeration
- **Generation**
 - enumerates different plans by assembling building blocks and adding one operator after another, until a complete plan has been produced
 - e.g., dynamic programming
- **Transformation**
 - enumerates different plans by transforming one plan into another equivalent plan
 - e.g., application of algebraic equivalences
- In the generation-based approach, plans can only be executed once all building blocks and operators have been assembled

Example Query Optimizers

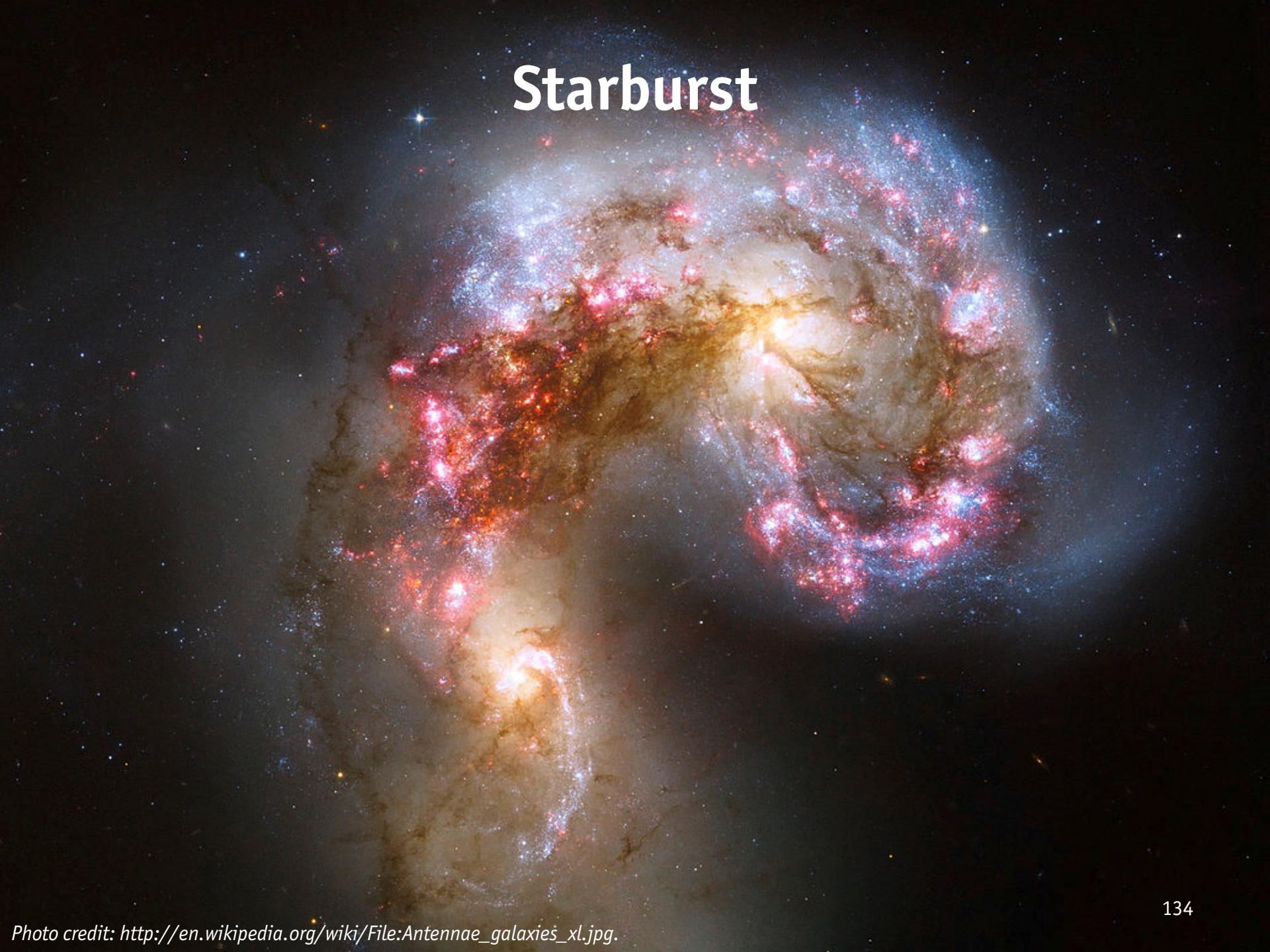
Plan generation vs. transformation



System R Optimizer

- Many ideas pioneered by System R have already been discussed
- Summary of important design choices in optimizer of System R
 1. **Cost Model:** account for both I/O cost and CPU cost
 2. **Use of Statistics:** estimate cost of a query evaluation plan based on statistics about database instance
 3. **Select-Project-Join Queries:** focus on query blocks without nesting, process nested queries in a relatively ad hoc way
 4. **Delay Duplicate Elimination:** only perform duplicate elimination for projections as a final step, if required by a **DISTINCT** clause
 5. **Left-Deep Plans:** to reduce number of alternative plans, consider only plans with binary joins, where inner relation is a base relation
 6. **Dynamic Programming:** build plans bottom up and prune search space after every step
 7. **Interesting Orders:** keep sub-plans that produce tuples in an order required by the query or other operators

Starburst

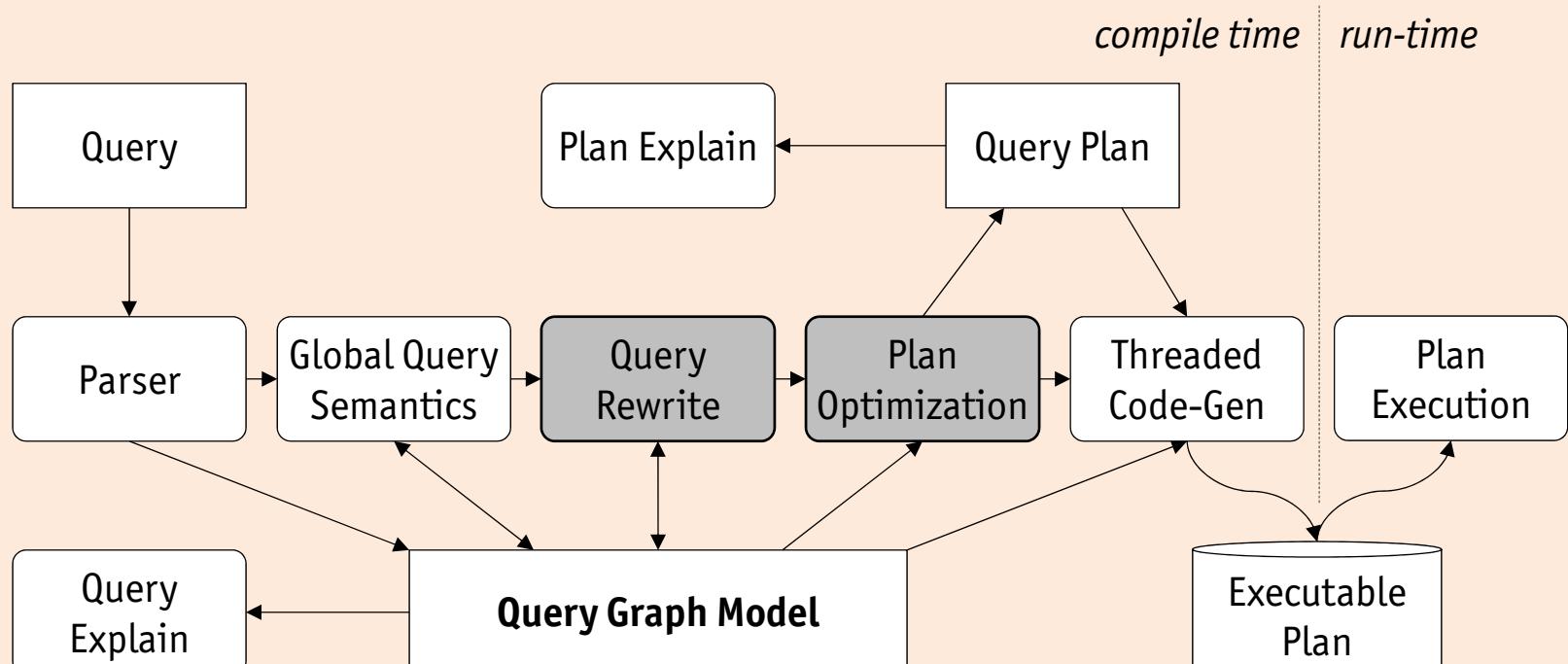


Starburst

- Starburst optimizer extends System R approach to support greater extensibility
- Queries are represented using the **query graph model** (QGM) throughout entire optimization lifecycle
- Query optimization has **two phases**
 1. **Query Rewrite** uses rules to transform a QGM representation of a query into another equivalent QGM representation
 2. **Plan Optimization** derives a query execution plan from QGM representation of a query bottom up using production rules
- First phase performs rewrite (heuristic) optimization, whereas second phase uses cost-based optimization

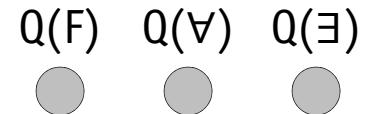
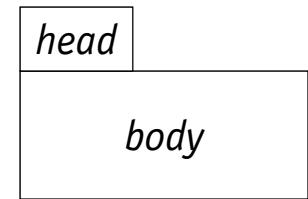
Starburst

Query compiler overview



Starburst

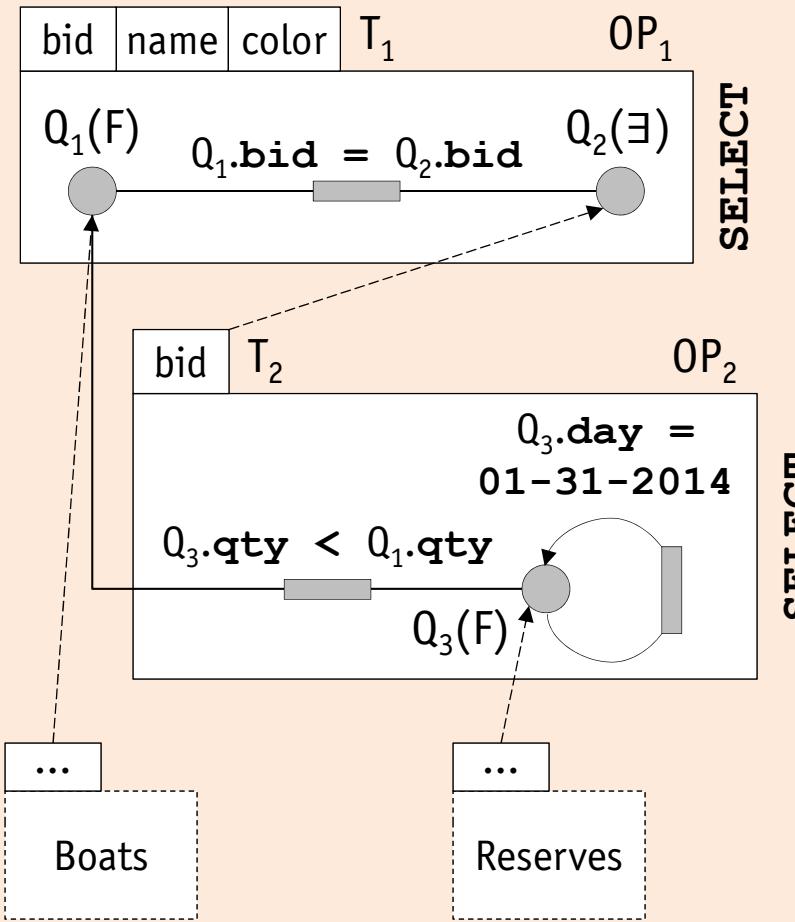
- Query Graph Model (QGM)
 - captures entire semantics of query to be compiled
 - used in all phases of query processing
- Boxes represent operations
 - **head** describes schema of output table
 - **body** contains a graphical representation of operation
 - select, insert, update, delete, union, and intersection
- Vertices represent iterators
 - **set formers** (F) contribute to result generation
 - **quantifiers** (\forall and \exists) are used to restrict a result
- Edges
 - **range edges** connect vertices to table or other operations
 - **qualifier edges** represent conjuncts of a predicate



Starburst

Example: query graph model

```
SELECT bid, name, color
  FROM Boats B
 WHERE B.bid IN
 (SELECT R.bid
    FROM Reserves R
   WHERE R.qty < B.qty
     AND R.day = 01-31-2014)
```



Starburst

- Query rewrite phase
 - uses a rule-based engine to **transform** a QGM into a more efficient QGM
 - terminates when no rules are eligible or (time) budget is exceeded
- Transformation rules
 - consist of a **condition** and an **action**, both implemented as a C function
 - **access** and **manipulate** QGM representation of query directly
 - rules can be grouped into classes, e.g., for **conflict resolution**
- Classes of rules
 - **predicate migration** pushes predicates into lower level operations
 - **projection push-down** avoids retrieval of unused table and view columns
 - **operation merging** combines two QGM operations into a single one
- Starburst avoids need for a general rule interpreter with pattern matching by using C as a rule language

Starburst

Example: query rewrite phase

Rule 1 (Subquery to join)

IF $OP_1.type = \text{SELECT} \wedge Q_2.type = \exists \wedge (\text{at each evaluation of the existential predicate at most one tuple of } T_2 \text{ satisfies the predicate})$

THEN

$Q_2.type := F;$

Rule 2 (Operation merging)

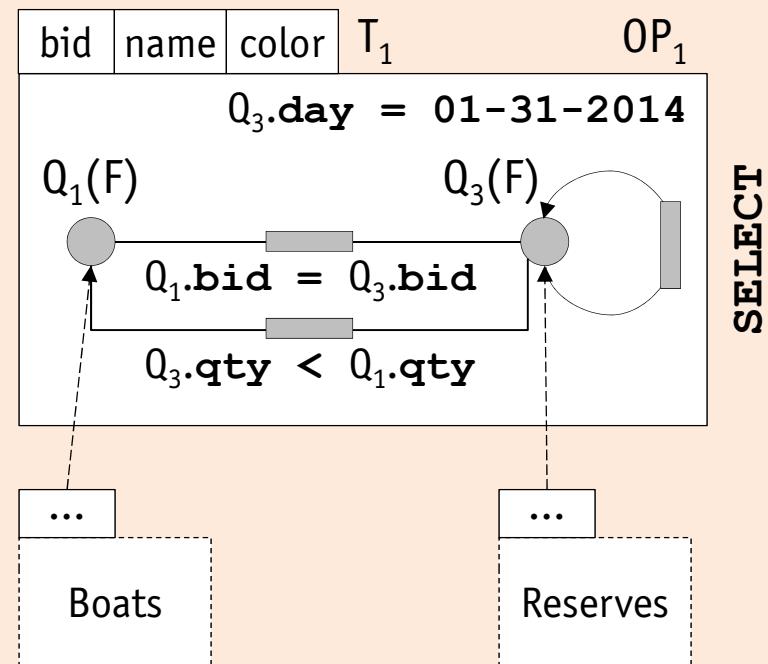
IF $OP_1.type = \text{SELECT} \wedge OP_2.type = \text{SELECT} \wedge Q_2.type = F \wedge (\text{NOT}(T_1.distinct = \text{FALSE}) \wedge OP_2.dupelim = \text{TRUE})$

THEN

merge OP_2 into OP_1 ;

IF $OP_2.dupelim = \text{TRUE}$

THEN $OP_1.dupelim := \text{TRUE};$

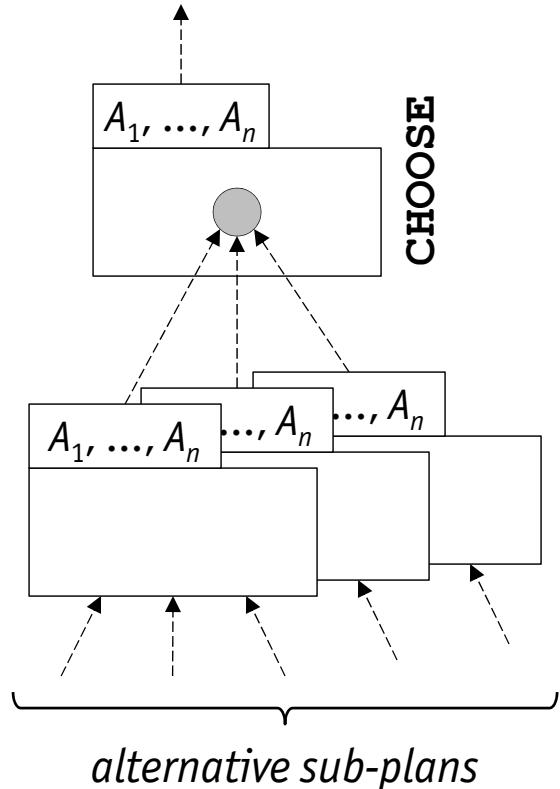


Starburst

- Rule engine applies rules using **forward chaining** method
 - start with available data, i.e., QGM sub-graphs
 - apply transformation rules until goal is reached
- Several **control strategies** are available to manage rules
 - **sequential**: process all rules sequentially
 - **priority**: give a chance to rules with a higher priority first
 - **statistical**: next rule is chosen randomly based on a user-defined probability distribution
- Search strategy
 - identifies QGM **sub-graphs** to which rules can be applied
 - supports **depth-first** (top down) and **breadth-first** search

Starburst

- Cost estimates are **not known** during query rewrite phase
 - both phases are executes strictly sequentially
 - without cost estimates plans cannot be pruned
- New QGM operator **CHOOSE**
 - links together alternative equivalent plans
 - cost-based optimization selects best plan later
- Number of alternatives may be very large
 - predicate migration
 - view merging vs. view materialization
- However, selection of best plan may even be deferred until query execution time



Starburst

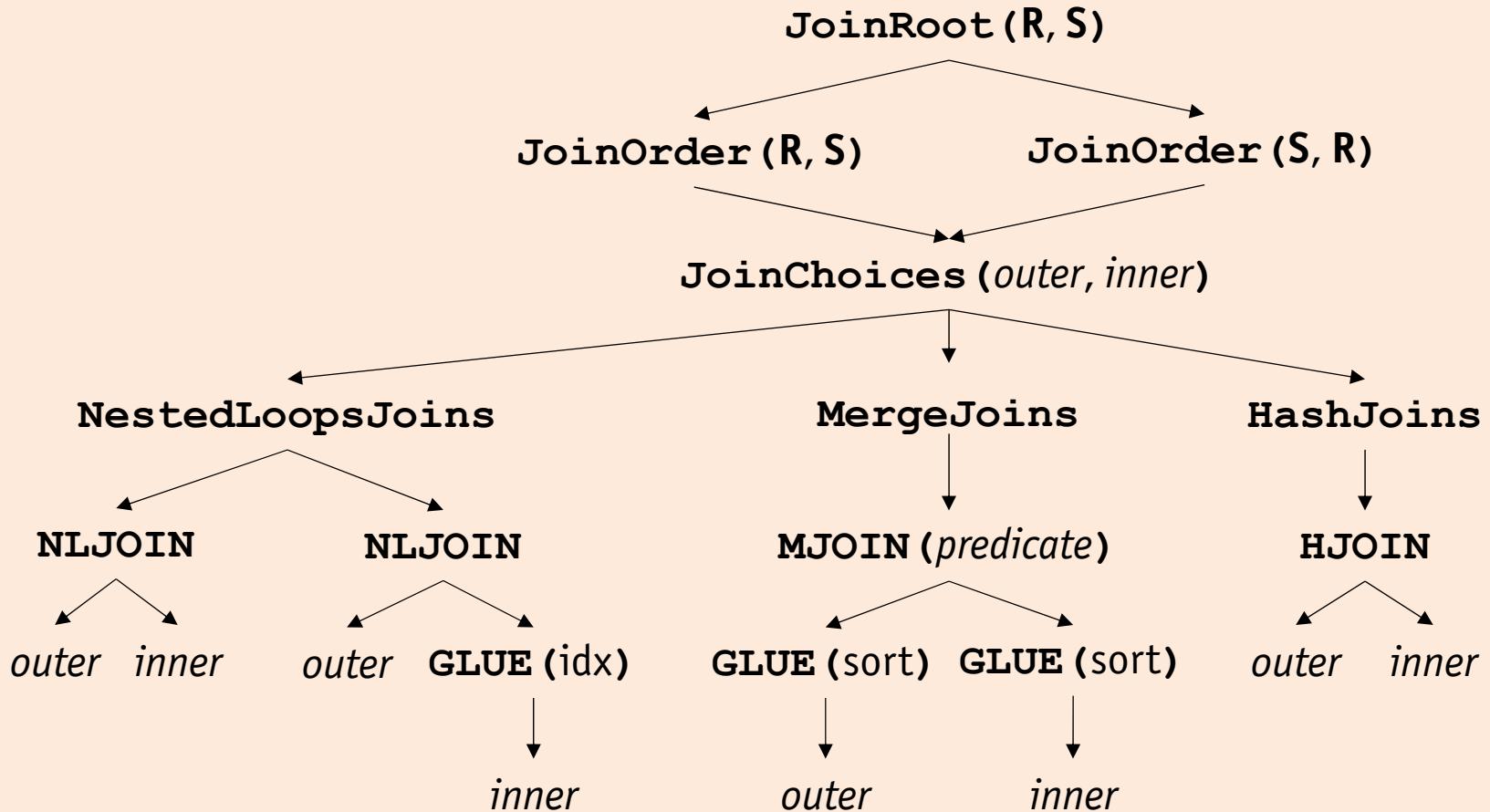
- Plan optimization phase
 - rule-driven generator derives query execution plans bottom-up
 - plan has a **relational** description, estimated **cost**, and **physical** properties
 - prunes plans with same logical and physical properties but higher cost
- Plan optimizer is a **select-project-join** optimizer
 - join enumerator is similar to enumeration scheme of System R
 - plan generator (*see next slide*)
- Join enumerator
 - uses feasibility criteria (mandatory and heuristic) to limit number of joins
 - enumeration algorithm can be replaced due to modular design

Starburst

- Plan generator is rule-based
- Strategy alternative rule (STAR)
 - set of parameterized rules is similar to a language grammar
 - define higher-level **non-terminal** from low-level **terminal** symbols
 - priority queue controls order in which STARs are evaluated
- Low-level plan operator (LOLEPOP)
 - terminal symbol of grammar language
 - concrete relational operators and physical operators (fetch, scan, etc.)
- Glue STAR
 - enforce required physical properties, e.g., sort order for merge join
 - may add LOLEPOP to make existing plan meet requirements

Starburst

Example: generation of join alternatives using STARS



Cascades



Cascades

- Transformation-based, top-down approach
 - **depth-first search**: beginning with original query, consider subqueries, and optimize them
 - **goal-driven**: no need to maintain bottom-up interesting orders
- Fully cost-based
 - no separation into phases, i.e., heuristic and cost-based optimization
- Flexible and extensible concepts
 - **operators**: expressions consisting of logical, physical, and item operators are used to represent query trees and execution plans
 - **rules**: plan search space is defined by a set of transformation, implementation, and enforcer rules
 - **strategy**: search space exploration strategy is guided by sequence of optimization tasks

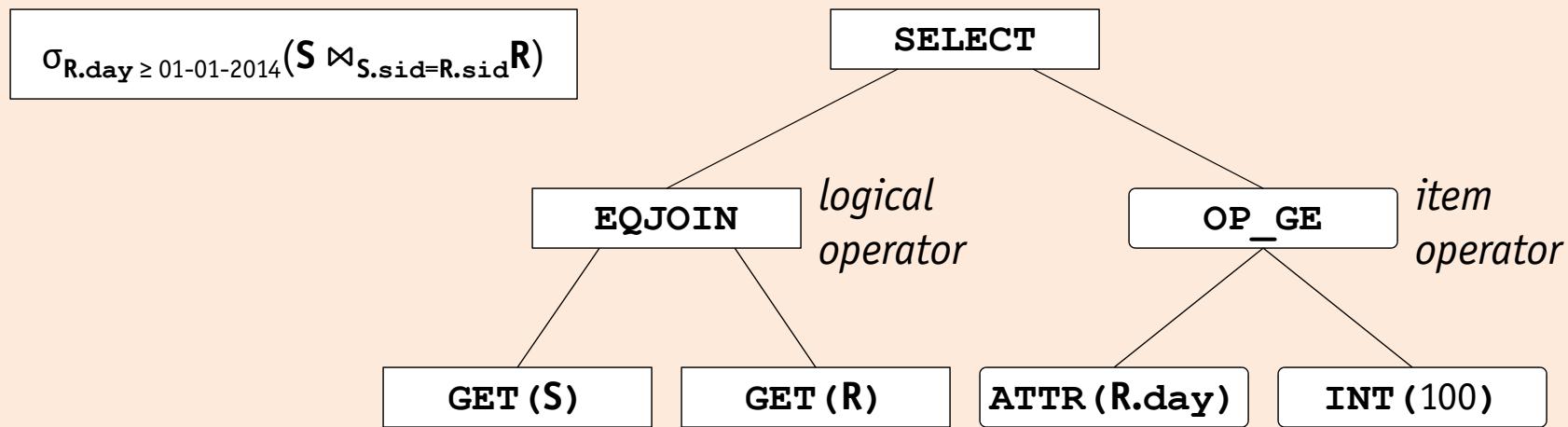
Cascades

- Query trees and execution plans are represented as **expressions**
- Logical operators
 - matching rule patterns to expressions
 - hashing for detecting duplicate expressions
 - deriving logical properties (e.g., schema) from input
- Physical operators
 - computing cost from operator algorithm and input cost
 - checking cost limit between optimization of two inputs
 - translating optimization goals to input operators
- Item operators
 - represent selection predicates for easy manipulation by rules

Cascades

Example: query expression tree

```
SELECT *
  FROM Sailors S, Reserves R
 WHERE S.sid = R.sid
   AND R.day >= 01-01-2014
```



Cascades

- A **group** is a set of equivalent logical and physical expressions
 - records **logical properties** of all expressions
 - maintains **lower bound**, such that every expression in group has a cost greater than this lower bound
 - current best plan, i.e., **winner**, is stored for each group

Example: expression group

[ABC] lower bound: 927

Logical Expressions

$(A \bowtie B) \bowtie C, (B \bowtie C) \bowtie A, (A \bowtie C) \bowtie B, A \bowtie (B \bowtie C), C \bowtie (A \bowtie B), B \bowtie (A \bowtie C),$
 $(B \bowtie A) \bowtie C, (C \bowtie B) \bowtie A, (C \bowtie A) \bowtie B, A \bowtie (C \bowtie B), C \bowtie (B \bowtie A), B \bowtie (C \bowtie A)$

Physical Expressions

$(A_F \bowtie_{NL} B_F) \bowtie_{NL} C_F, (A_F \bowtie_M B_F) \bowtie_M C_F, (B_F \bowtie_{NL} C_F) \bowtie_{NL} A_F, (A_F \bowtie_{NL} C_F) \bowtie_{NL} B_F, \dots$

Winner

$(B_F \bowtie_M C_F) \bowtie_{INL} A_{IDX}$

Cascades

- Use of **multi-expressions** saves space required for a group
 - consist of logical and physical operators, but take groups as input
 - reduce number of expressions by representing all equivalent expressions

Example: multi-expression group

[ABC] lower bound: 927

Logical Expressions

$[AB] \bowtie [C], [BC] \bowtie [A], [AC] \bowtie [B], [A] \bowtie [BC], [C] \bowtie [AB], [B] \bowtie [AC]$

Physical Expressions

$[AB] \bowtie_{NL} [C], [AB] \bowtie_M [C], [BC] \bowtie_{NL} [A], [AC] \bowtie_{NL} [B], \dots$

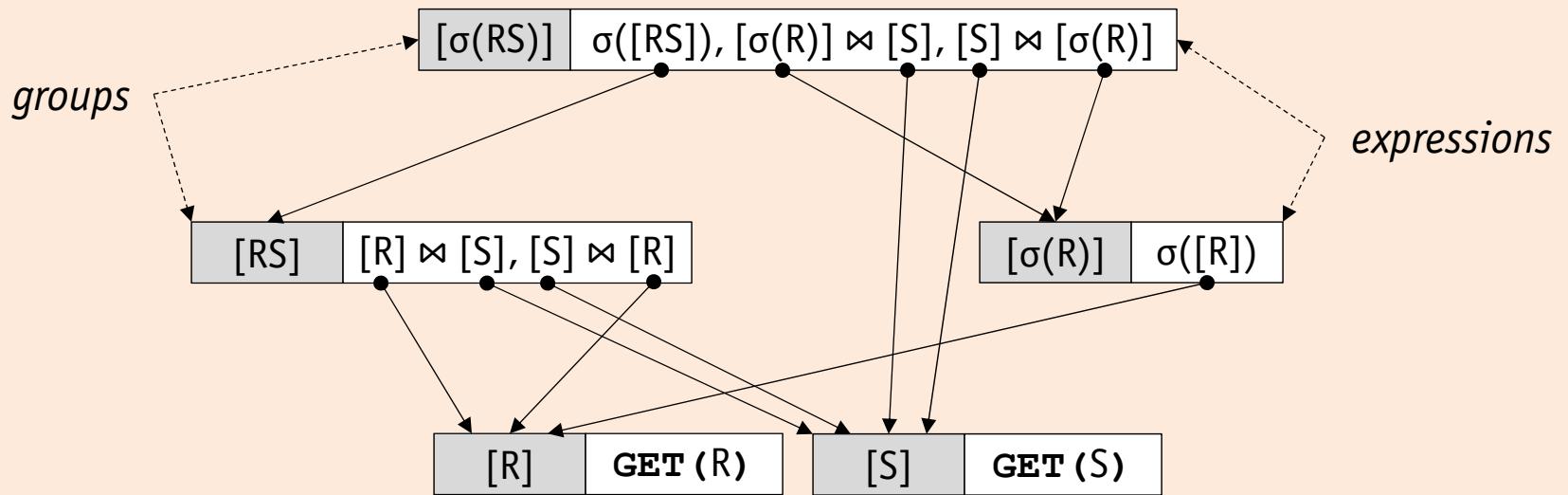
Winner

$(B_F \bowtie_M C_F) \bowtie_{INL} A_{IDX}$

Cascades

Example: query expression tree

```
SELECT *
  FROM Sailors S, Reserves R
 WHERE S.sid = R.sid
   AND R.day >= 01-01-2014
```



Cascades

- Possible search space is defined by **rules**
 - rules transform an expression into a **logically equivalent** expression
 - **pattern** defines structure of logical expression to which applies
 - **condition** checks whether rule can be applied
 - **substitute** defines structure of result after rule application
- Cascades distinguishes three types of rules
 - **transformation rule**: substitute is another logical expression
 - **implementation rule**: substitute is a physical expression
 - **enforcer rule**: substitute contains an enforcer, i.e., an operator that guarantees certain physical properties, e.g., sort
- Rules can have a **promise** value that guides exploration order
 - rules that require a specific physical property typically have promise 0
 - transformation rules typically have promise 1
 - implementation rules typically have promise 2

Cascades

Example: rules

- Transformation rules

- EQJOIN_LTOR

Pattern: $(Leaf(X) \bowtie Leaf(Y)) \bowtie Leaf(Z)$

Substitute: $Leaf(X) \bowtie (Leaf(Y) \bowtie Leaf(Z))$

- Implementation rules

- EQ_TO_SM

Pattern: $Leaf(X) \bowtie Leaf(Y)$

Substitute: $Leaf(X) \bowtie_{SM} Leaf(Y)$

- EQ_TO_INL

Pattern: $Leaf(X) \bowtie Leaf(Y)$

Condition: \exists index on join attribute

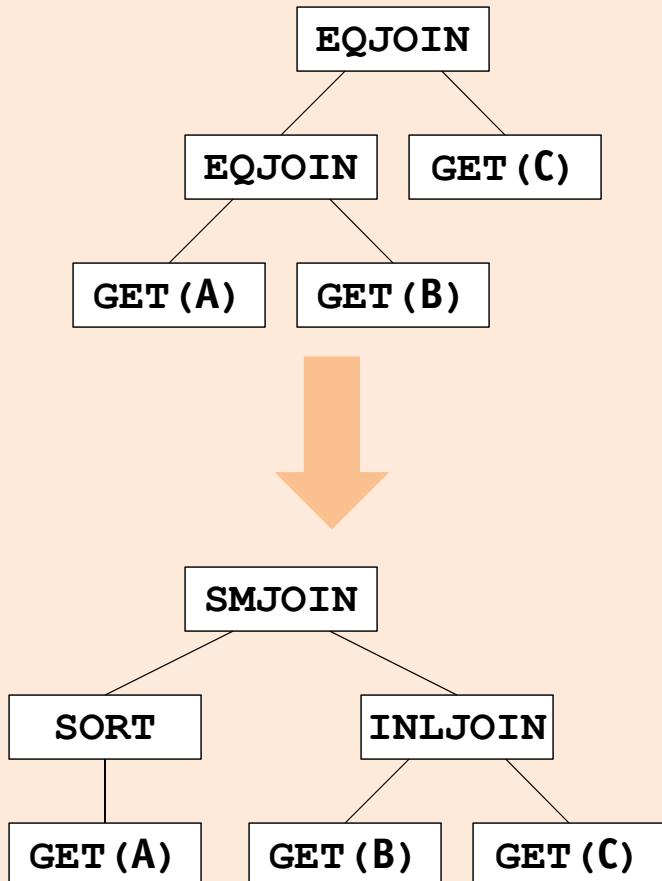
Substitute: $Leaf(X) \bowtie_{INL} Leaf(Y)$

- Enforcer rules

- SORT

Pattern: $Leaf(X)$

Substitute: $SORT Leaf(X)$

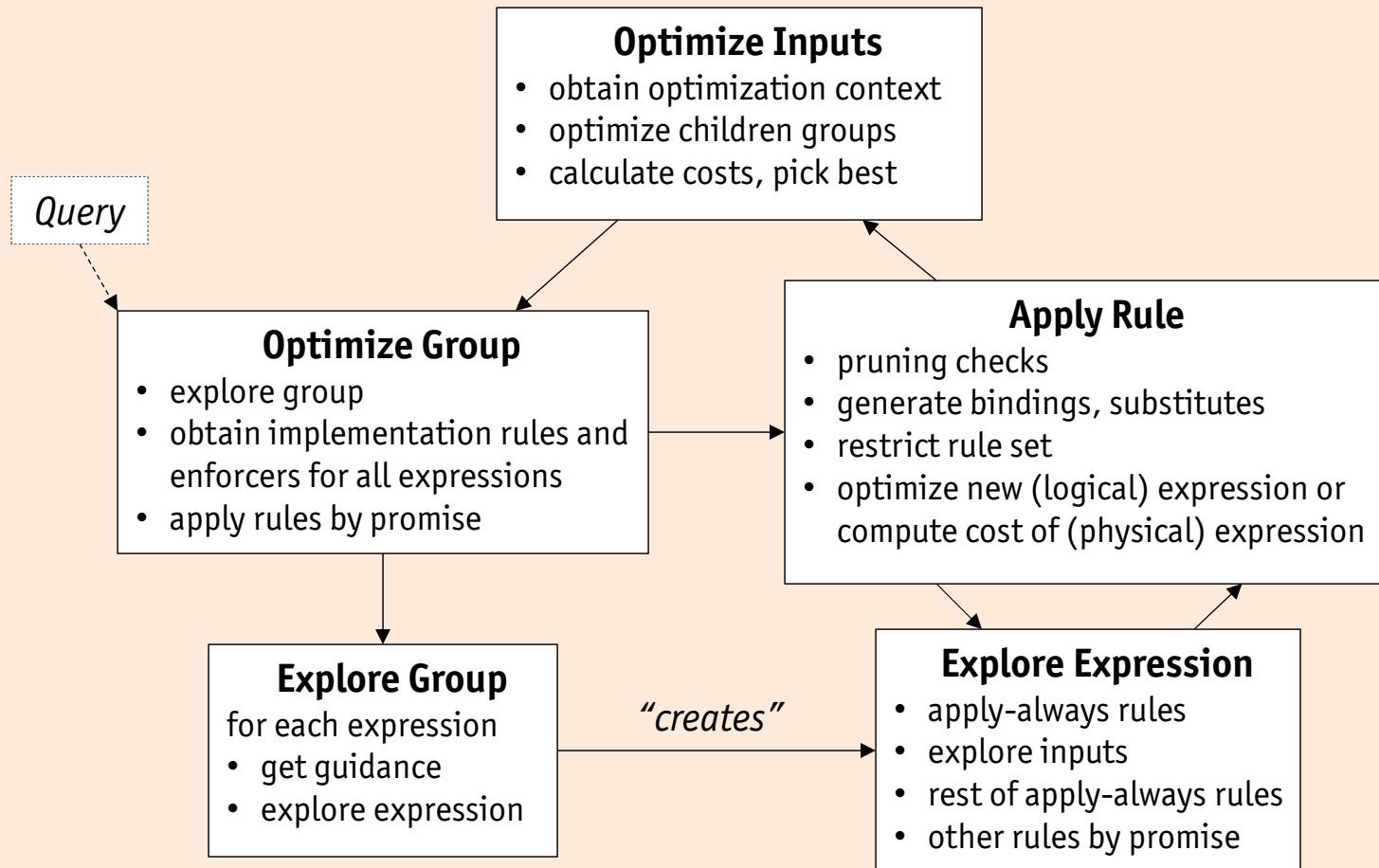


Cascades

- Optimization algorithm is broken down into **tasks**
 - original task is to optimize entire query
 - tasks create and schedule each other
 - when no undone tasks remain, optimization terminates
- Tasks are managed in a **task structure**
 - last-in-first-out stack, dependency graph, etc.
 - tasks can be reordered for heuristic guidance
- An **optimization goal** combines a group or expression and a cost limit with required and excluded physical properties
 - pursuing an optimization goal results either in a plan or in failure
 - dynamic programming and memoization are used to ensure that the same optimization goal is not pursued multiple times

Cascades

Optimization tasks



Cascades

>Main optimization loop

```
function optimize (query)

Stack<OptTask> tasks = new Stack<OptTask> () ;

tasks .push (new OptGroup (query) ) ;           (start optimization at top group)

while tasks is not empty do      (perform undone tasks until there are no more left)
    task ← tasks .pop () ;          (get next task)
    task .perform () ;            (perform next task)

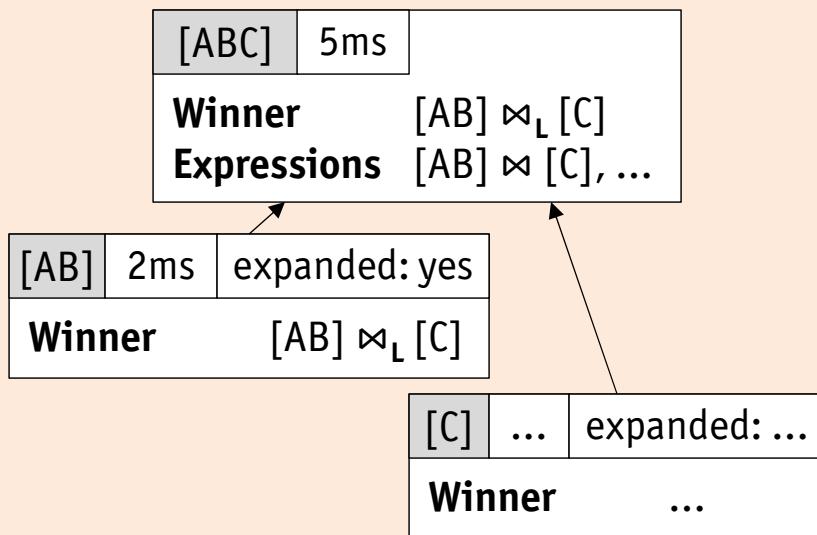
end
```

Cascades

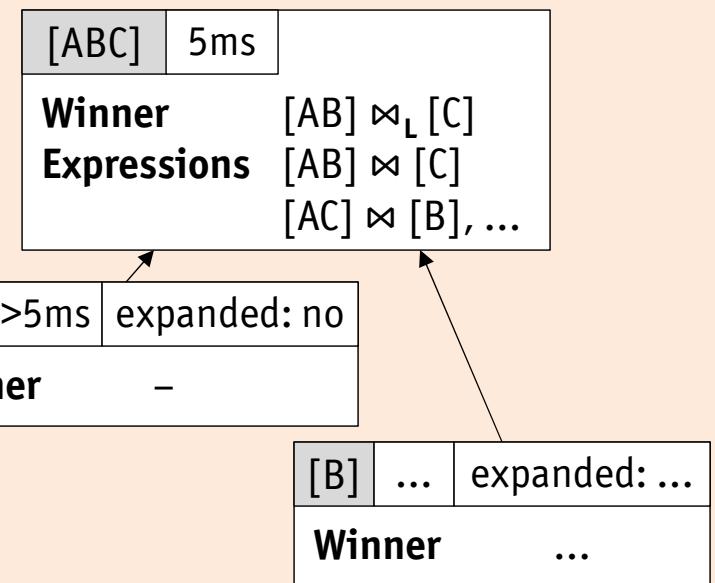
- Plan **pruning** is performed in the optimize input task
 - each time an input has been optimized, lowest cost so far is recorded
 - this cost serves as limit for optimizing next inputs

Example: pruning

Assume that $[AC]$ is a Cartesian product.



After optimizing $[AB] \bowtie [C]$



After optimizing $[AC] \bowtie [B]$

Starburst vs. Cascades

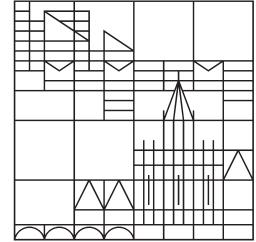
- Optimization phases
 - Starburst optimizes queries in **two phases**, a (heuristic) query rewrite phase and a (cost-based) plan optimization phase
 - Cascades only uses **one phase** since all transformations are algebraic and cost-based
- Mapping logical to physical operators
 - Starburst expands expressions **step-by-step** as grammar-like set of plan production rules implies hierarchy of intermediate levels (non-terminals)
 - Cascades accomplishes this mapping in **one single step**
- Rule application
 - Starburst applies rules by **forward chaining** in query rewrite phase
 - Cascades performs **goal-driven** application of rules

Assignments and Exam Grading

- Same grading scale used for assignments and exam
- Exam will consist of four exercises with 25 points each
 1. disk and file storage
 2. hash and tree-based indexes
 3. query evaluation
 4. query optimization
- Overflow bonus points to exam points conversion

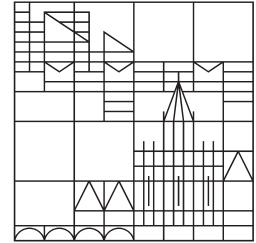
$$P_{exam} = \left\lceil \frac{1}{2} \cdot \frac{100}{260} \cdot P_{bonus} \right\rceil$$

Percentage of Points	German Grade	Letter Grade
100-91	1.0	A
90-86	1.3	A-
85-81	1.7	B+
80-76	2.0	B
75-71	2.3	B-
70-66	2.7	C+
65-61	3.0	C
60-56	3.3	C-
55-51	3.7	D+
50-45	4.0	D
44-0	5.0	F



Database System Architecture and Implementation

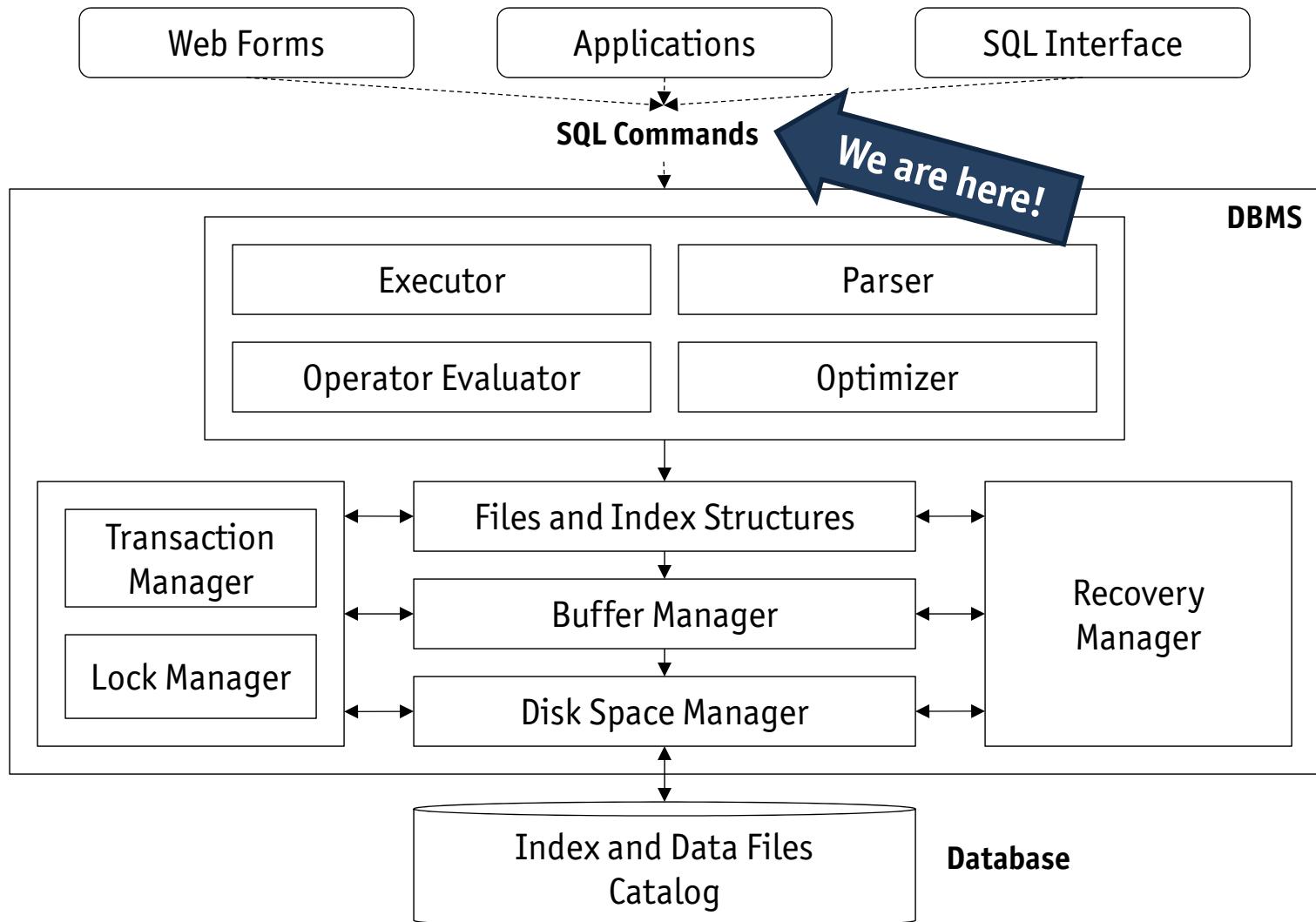
TO BE CONTINUED...



Database System Architecture and Implementation

Module 9
Physical Database Design
February 10, 2014

Orientation

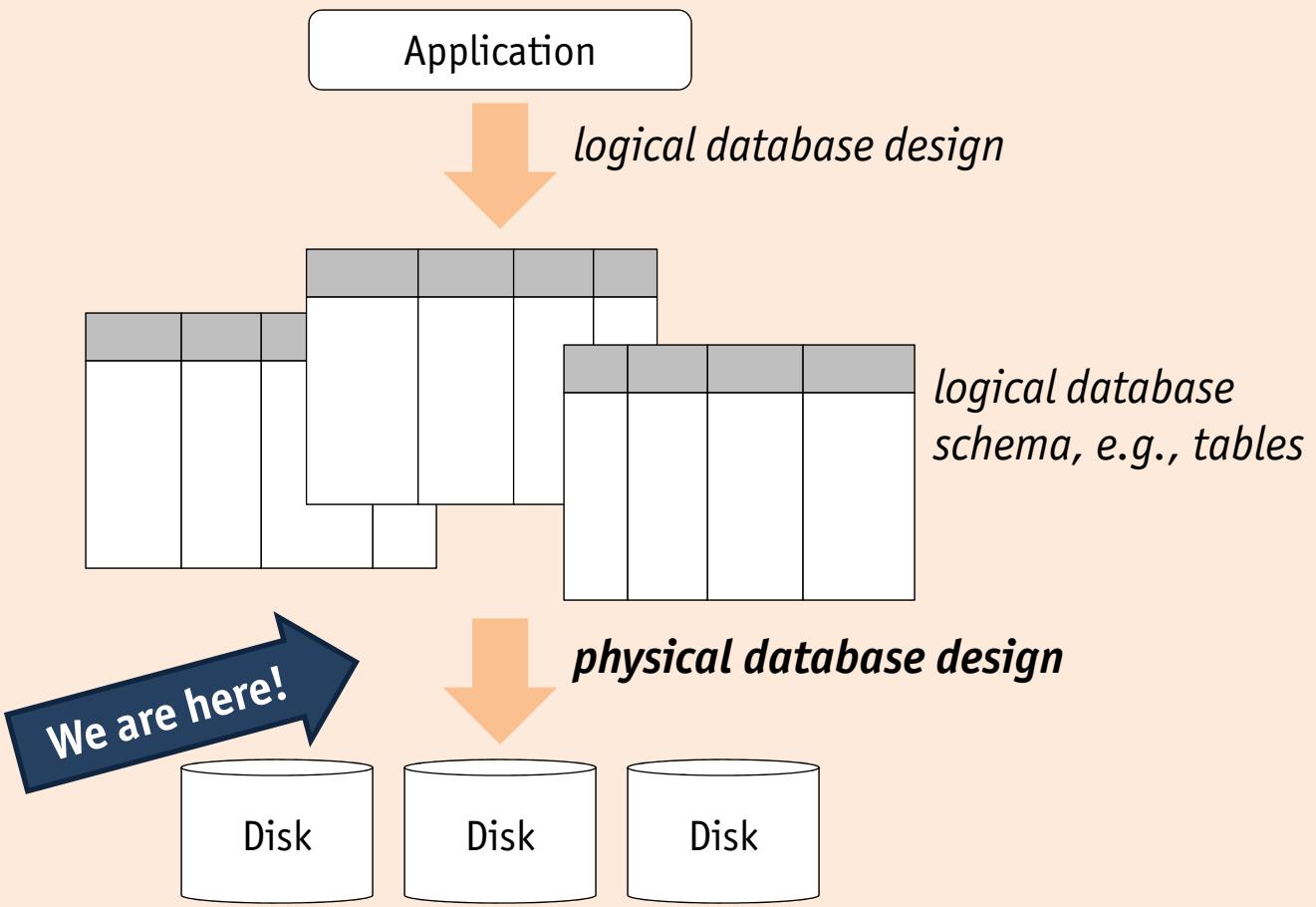


Module Overview

- Physical database design
 - index selection
 - clustering and indexing
- Database tuning
 - index tuning
 - conceptual schema tuning
 - query and view tuning
- Benchmarking

Outline

The physical database design problem



Physical Database Design

Problem statement

Given

- a **logical database schema**, e.g., in the form of relational table definitions
- a description of a **database work load**

Find

- an **internal data representation** that maximizes overall performance, e.g., maximal throughput, minimal response time, etc.
- Different internal representations have different, competing effects on performance of individual operations
- Optimization problem
 - data structure is chosen to **speed up** one particular type of operation
 - same data structure typically **slows down** other operations

Physical Database Design

Exercise

Can you think of a **data structure** that speeds up one type of operation, but slows down another?

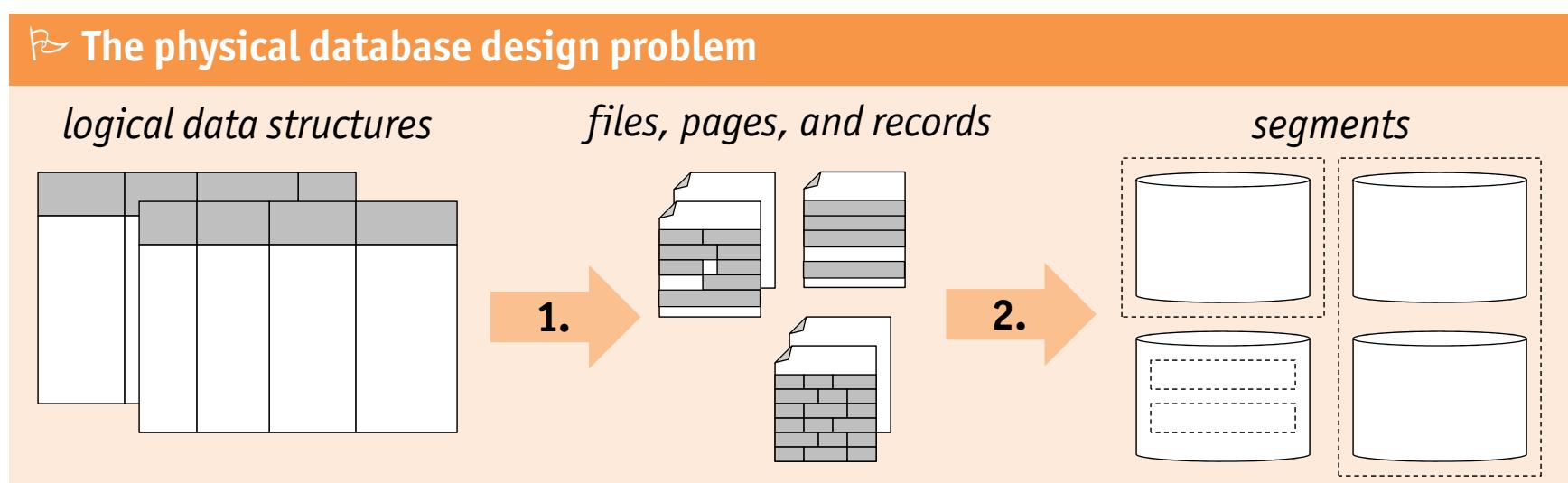
Physical Database Design

- Recall abstractions that take care of low-level data management
 - disk and file management: files of disk blocks (pages)
 - main memory management: (buffer) pages of records
- Higher-level abstraction: a continuous sequences of pages is called a **segment**
 - can think of a segment as an abstraction of a single file
 - typically, a DBMS uses more than one level of mapping segments to operating system files/disks
 - DBMS have variety of names for segments: table space, partition, storage pool, etc.
 - internally, a database comprises a collection of segments to hold data
- Physical database design denotes problem of mapping logical database schema (e.g., tables and their tuples) to segments

Physical Database Design

- Break down physical database design problem into two steps
 1. map logical data structures onto **internal data types** (and indexes)
 2. allocate instances of internal data types to (**pages of**) **segments**
- Focus of this lecture
 - techniques for first step will be discussed in detail
 - overview of second step is given on next slide

The physical database design problem



Internal Data Types to Segments

- When allocating instances of internal data types to segments, **locality of access operations** needs to be considered
 - records within a page can be accessed within same disk I/O operation
 - neighboring pages can be accessed faster than distant pages
 - different segments are mapped to disks individually and independently
- Main decision is whether to map different internal data type into the **same segment or not**
 - in this lecture, we assume **1:1-mapping** in most cases, i.e., separate segments per internal data type
 - in this case, segment free space manager is only DBMS component that makes allocation decisions
 - therefore, any other influence has to be exercised through the mapping of logical to internal data types (Step 1 on previous slide)

Logical Structures to Internal Data Types

R Standard solution

1 logical table/relation \longleftrightarrow 1 internal record type (file)
1 tuple \longleftrightarrow 1 record
1 attribute \longleftrightarrow 1 field
+
indexes to speed up search

- Depending on database workload other options may be useful
 - decomposing tuples of a relation (horizontally/vertically)
 - grouping related tuples from different relations
 - replicating (parts of) tuples
 - materializing results of important/frequent queries
 - combining any of the above options

Database Workload

- A database workload description includes following information
 - list of **queries** (with their frequencies, as a ratio of all queries/updates)
 - list of **updates** and their frequencies
 - **performance goals** for each type of query and update
- For each query, the description identifies
 - which relations are accessed
 - which attributes are retained (in **SELECT** clause)
 - which attributes have selection or join conditions on them (in **WHERE** clause) and how selective these conditions are likely to be
- For each update, the description identifies
 - which attributes have selection or join conditions on them (in **WHERE** clause) and how selective these conditions are likely to be
 - type of update (**INSERT**, **DELETE**, or **UPDATE**) and updated relation
 - fields modified by the update (for **UPDATE** commands)

Physical Design and Tuning Decision

- Choice of indexes to create
 - which **relations to index** and which (combination of) fields to chose as index search keys
 - for each index, whether it should be **clustered** or **unclustered**
- Tuning conceptual schema
 - **alternative normalized schemas**: choose one of multiple possible ways to decompose a schema into a desired normal form (BCNF, 3NF)
 - **denormalization**: partially “undo” normalization to improve performance of queries that involve attributes from several decomposed relations
 - **vertical or horizontal partitioning**: split relations to improve performance of queries that only involve a subset of attributes or tuples
 - **views**: add views to mask changes in conceptual schema from user
- Query and transaction tuning by **rewriting** frequently executed queries to run faster

Guidelines for Index Selection

Guideline #1: Whether to Index

- do not build an index unless some query (including query components of updates) **benefits** from it
- if possible, choose indexes that speed up **more than one** query

Guideline #2: Choice of Search Key

Attributes mentioned in **WHERE** clause are candidates for indexing

- **exact match selection condition:** consider a (hash) index on selected attributes
- **range selection condition:** consider a B+ tree (or ISAM) index on selected attributes

Exercise: B+ tree vs. ISAM index

When is an ISAM index worth considering?

Guidelines for Index Selection

Guideline #3: Multi-Attribute Search Keys

Indexes with multi-attribute search keys should be considered

- if a **WHERE** clause includes conditions on more than one attribute of a relation
- if they enable **index-only evaluation strategies** for important queries

Note that order of attributes in search key is particularly important if range queries are expected.

Guideline #4: Whether to Cluster

Choice of clustered index is important as only one index per relation can be clustered and clustering affects performance greatly

- **rule of thumb:** range queries are likely to benefit most from clustered index
- if **several range queries** involve different sets of attributes, choice of clustered index should be guided by selectivity of conditions and relative frequency of queries
- for indexes that are used to enable an **index-only evaluation strategy**, clustering is not required

Guidelines for Index Selection

☞ Guideline #5: Hash vs. Tree Index

Tree indexes support more types of selection conditions and are usually preferable, but hash indexes are better in the following situations

- index is intended to support **nested loops join** (indexed relation is inner relation, search key includes join columns): slight improvement of hash index over tree index is magnified by iterations of outer loop
- there is a **very important equality query**, but no range queries, involving search key attributes

☞ Guideline #6: Balancing Cost of Index Maintenance

After drawing up a “wishlist” of indexes to create, consider impact of each index on updates in workload

- if maintaining a potential index slows down **frequent update operations**, consider dropping it
- however, adding an index may well **speed up** a given update operation (query component of updates)

Guidelines for Index Selection

Exercise: Which indexes should be created?

```
SELECT E.ename, D.mgr  
  FROM Employees E, Departments D  
 WHERE D.dname = "Toy" AND E.dno = D.dno
```

Index selection guidelines suggest that **D . dname**, **D . dno**, and **E . dno** are candidate attributes to be indexed. What indexes and what types of indexes would you create, under which assumptions?

Index attribute **D . dname**?

Index attribute **D . dno**?

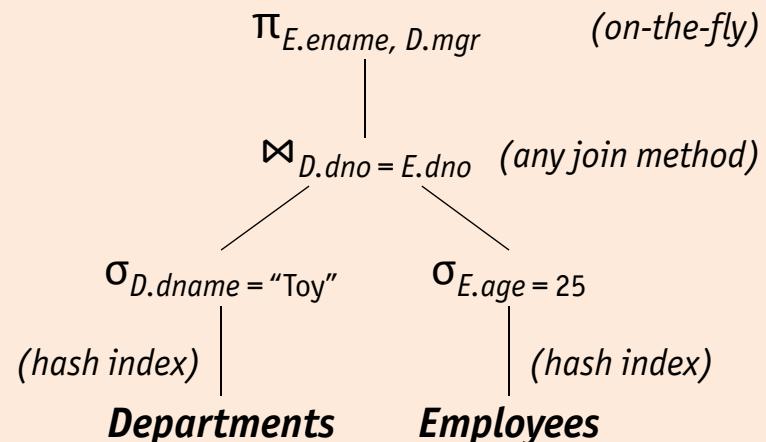
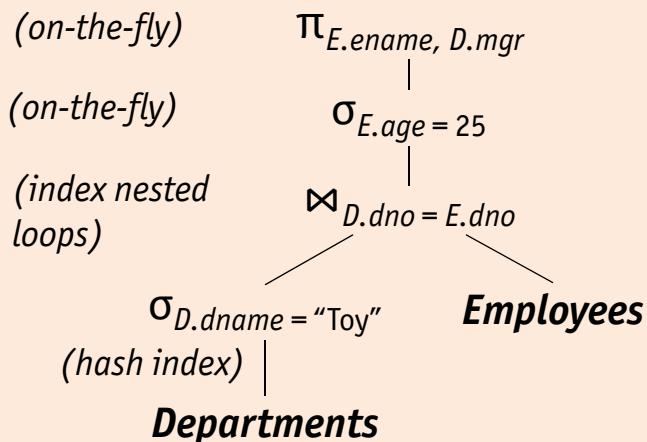
Index attribute **E . dno**?

Guidelines for Index Selection

Example: alternative evaluation plans

Let us consider alternative evaluation plans for this slight variant of the previous query.

```
SELECT E.ename, D.mgr  
      FROM Employees E, Departments D  
     WHERE D.dname = "Toy" AND E.dno = D.dno AND E.age = 25
```



- ☞ If there is already an index on **E . dno**, adding another index on **E . age** is **not justified** by this additional evaluation plan.

Guidelines for Index Selection

Exercise: Which indexes should be created?

```
SELECT E.ename, D.mgr
  FROM Employees E, Departments D
 WHERE E.sal BETWEEN 10000 AND 20000
       AND E.hobby = "Stamps" AND E.dno = D.dno
```

Index selection guidelines suggest that **D.dno**, **E.hobby**, **E.sal**, and **E.dno** are candidate attributes to be indexed. What indexes and what types of indexes would you create, under which assumptions?

Index attribute **E.hobby** or **E.sal**?

Index attribute **E.dno**?

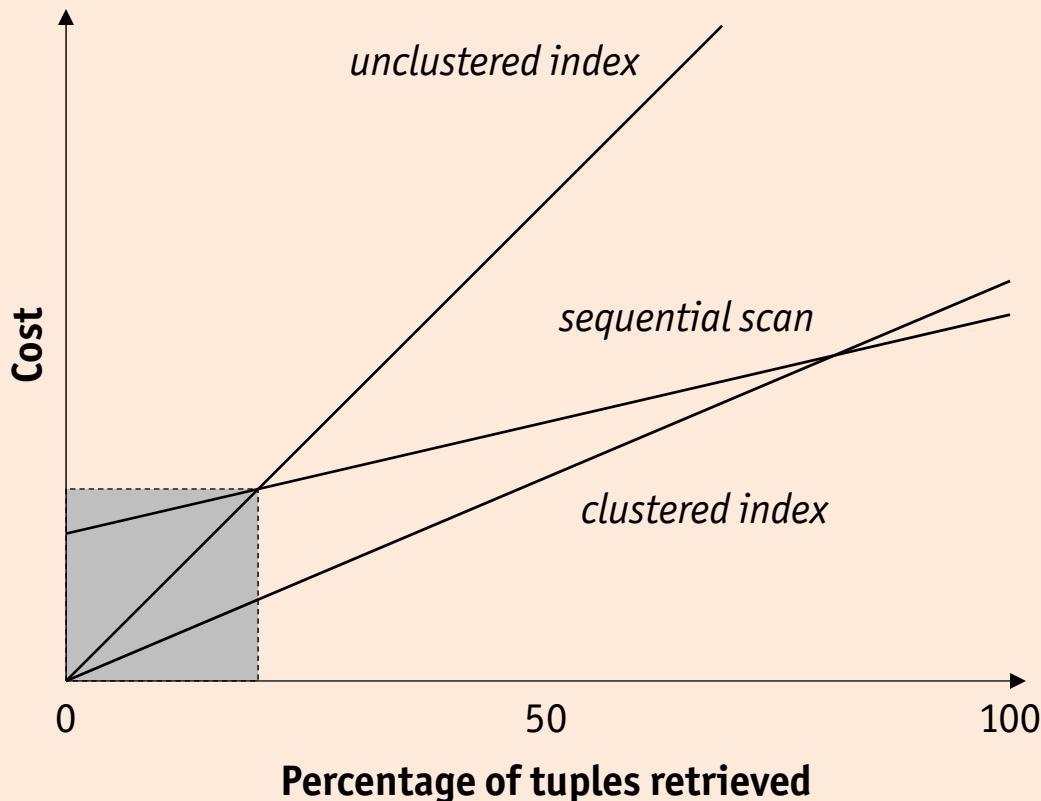
Index attribute **D.dno**?

Clustering and Indexing

- Apart from which indexes to create, whether these indexes are **clustered** or **unclustered** is an important decision
 - clustered indexes: use for selection predicates with **low selectivity**, e.g., range selections or selections attributes that are not candidate keys
 - unclustered indexes: use for selection predicates with **high selectivity**, e.g., equality selections on attributes that are candidate keys
- Impact on clustering depends on number of retrieved tuples
 - to retrieve **one tuple**, unclustered index is just as good as clustered index
 - to retrieve **a few tuples**, unclustered index is better than a sequential scan of entire relation
 - to retrieve **all tuples**, sequential scan of entire relation is better than clustered index
- Use of blocked I/O improves relative advantage of sequential scan over unclustered index

Clustering and Indexing

Impact of clustering



Note that **grey area** denotes range in which unclustered index is better than sequential scan of entire relation.

Clustering and Indexing

Exercise: Clustered or unclustered index?

```
SELECT E.ename, D.mgr  
  FROM Employees E, Departments D  
 WHERE D.dname = "Toy" AND E.dno = D.dno
```

For this example query, we have already decided to create an index on **D . dname** and **E . dno**. Should these indexes be clustered or unclustered?

Clustered or unclustered index on **D . dname**?

Clustered or unclustered index on **E . dno**?

Clustering and Indexing

R Example: Clustered or unclustered index?

```
SELECT E.ename, D.mgr
  FROM Employees E, Departments D
 WHERE E.hobby = "Stamps" AND E.dno = D.dno
```

In contrast to the previous example (`D.dname = "Toy"`), the selection condition on `E.hobby` is not likely to be very selective as many employees might collect stamps.

- ↳ a plan using a **block nested loops** or a **sort-merge join**, rather than a plan using an index nested loops join is likely to be chosen by query optimizer
- ↳ a sort-merge join would benefit from a **clustered tree index** on `D.dno`
- ↳ if there is no index on `E.dno`, query optimizer might still be able to use another index on `Employees`, say a clustered index on `E.hobby`, to retrieve tuples

Checking Query Execution Plans

The Real World

All major RDBMS provide functionality to display the execution plan chosen by the query optimizer for a particular query.

↳ **IBM DB2 and PostgreSQL**

- offer an **EXPLAIN** command that can be used for any explainable statement

↳ **Oracle 10g**

- provides an **EXPLAIN PLAN** statement, which creates a relation **PLAN_TABLE** with a row for each step of the query execution plan

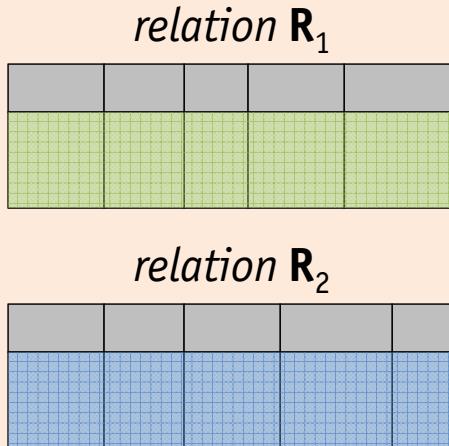
↳ **Microsoft SQL Server**

- SQL Server Management studio can display graphical execution plans (CTRL+M)
- command **SET SHOWPLAN_TEXT ON** will display textual execution plans
- command **SET SHOWPLAN_XML ON** will display execution plans in XML

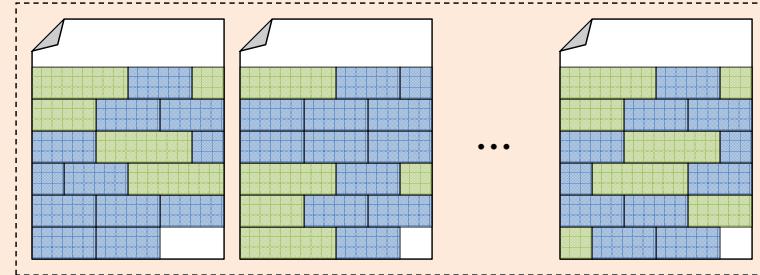
Co-Clustering Two Relations

- Some DBMS can store records from **more than one** relation to be stored in a **single** file
 - physical interleaving can be configured by database administrator
 - this data layout is sometimes referred to as **co-clustering** two relations

Co-clustering two relations



co-clustered internal file



Co-Clustering Two Relations

Example: Co-clustering two relations

Parts (*pid: integer, pname: string, cost: integer, supplierid: integer*)

Assembly (*partid: integer, componentid: integer, quantity: integer*)

Relation **Assembly** represents a 1:N relationship between parts and their sub-parts: a part can have many sub-parts, but each part is the sub-part of at most one part.

```
SELECT P.pid, A.componentid  
FROM Parts P, Assembly A  
WHERE P.pid = A.partid AND P.cost = 10
```

Performance of this query is improved by an index on **P.cost** and one on **A.partid**. Parts with *cost = 10* can be retrieved using first index. However, if many parts have this cost, second index has to be accessed for each of these records.

This second index access can be avoided by co-clustering the two relations: each **Parts** record **P** is followed on disk by all **Assembly** records **A**, such that **P.pid = A.partid**.

This optimization is especially important if the parts hierarchy is traversed **recursively**!

Co-Clustering Two Relations

- **Pros**
 - can speed up joins, in particular key-foreign key joins corresponding to 1:N relationships
- **Cons**
 - sequential scan of either relation becomes slower, since in both cases (possibly multiple) retrieved records have to be skipped, wasting I/O
 - all inserts, deletes, and updates that alter record lengths become slower, due to the overhead involved in maintaining the clustering

Index-Only Plans

- Finding indexes that enable **index-only plans** is an important goal of index selection
 - avoid retrieving tuples from one of the referenced relations
 - scan an associated index instead, which is likely to be much smaller
- An index that is used (only) for index-only plans does **not** have to be clustered
 - no tuples from indexed relation need to be retrieved
 - index contains all data required to evaluate query
- Recall that index search key can consist of many attribute values

Index-Only Plans

Example: Index-only plan

```
SELECT D.mgr, E.eid  
  FROM Departments D, Employees E  
 WHERE D.dno = E.dno
```

Considerations

- performance of this query is improved by an index on **E.dno**, as it enables an index nested loops join with **Departments** as outer relation
- since **E.dno** is not a candidate key, this index needs to be clustered in order for this plan to be efficient
- however, if there is already a clustered index on **Employees**, another one cannot be created
- by creating an unclustered index with search key **(dno, eid)**, an index-only plan can be enabled
- this plan uses an index nested loops join with **Departments** as outer relation and an index-only scan of the inner relation

Database Tuning

- Actual use of database provides detailed information that can be used to **refine** its initial design
 - assumptions about workload can be replaced by observed usage patterns
 - guesses about size of data can be replaced with actual statistics
- Continued database tuning is important to get best performance
 - index tuning
 - conceptual schema tuning
 - query and view tuning

Tuning Indexes

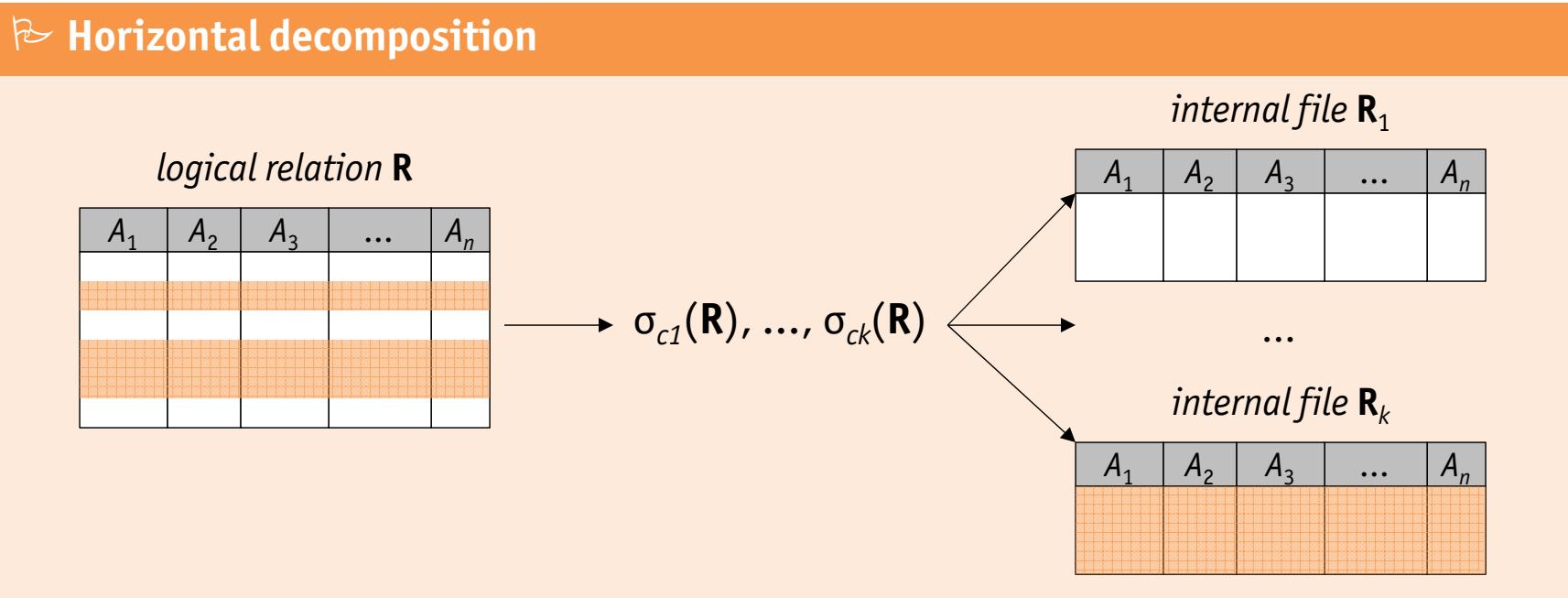
- Reasons to refine choice of indexes
 - queries and updates expected to be important are not very frequent
 - new queries and updates may be identified to be important
 - optimizer does not find the plans that it was expected to
- Index tuning actions
 - drop indexes created for queries and updates that are infrequent
 - add new indexes to support frequent queries and updates
 - alter existing indexes (search key, clustered vs. unclustered)
 - periodically rebuild static indexes (e.g., ISAM) or dynamic B+ trees that do not merge pages on delete to improve space occupancy

Tuning Conceptual Schema

- Redesign conceptual schema (and re-examine physical database design) in order to meet performance objectives
 - once database has been designed and populated with tuples, changing the conceptual schema requires significant effort
 - changes to schema of an operational database sometimes referred to as **schema evolution**
- Choice of conceptual schema should be guided by consideration of queries and updates in workload
 - **decomposition** of logical relations
 - **denormalization** of logical relations

Horizontal Decomposition

- Map tuples of logical relation into several (disjoint or overlapping) subsets that are stored in separate internal files
 - all resulting internal files share the **same structure**
 - internal files can be viewed as materialized results of a set of **selection queries** on the logical relation

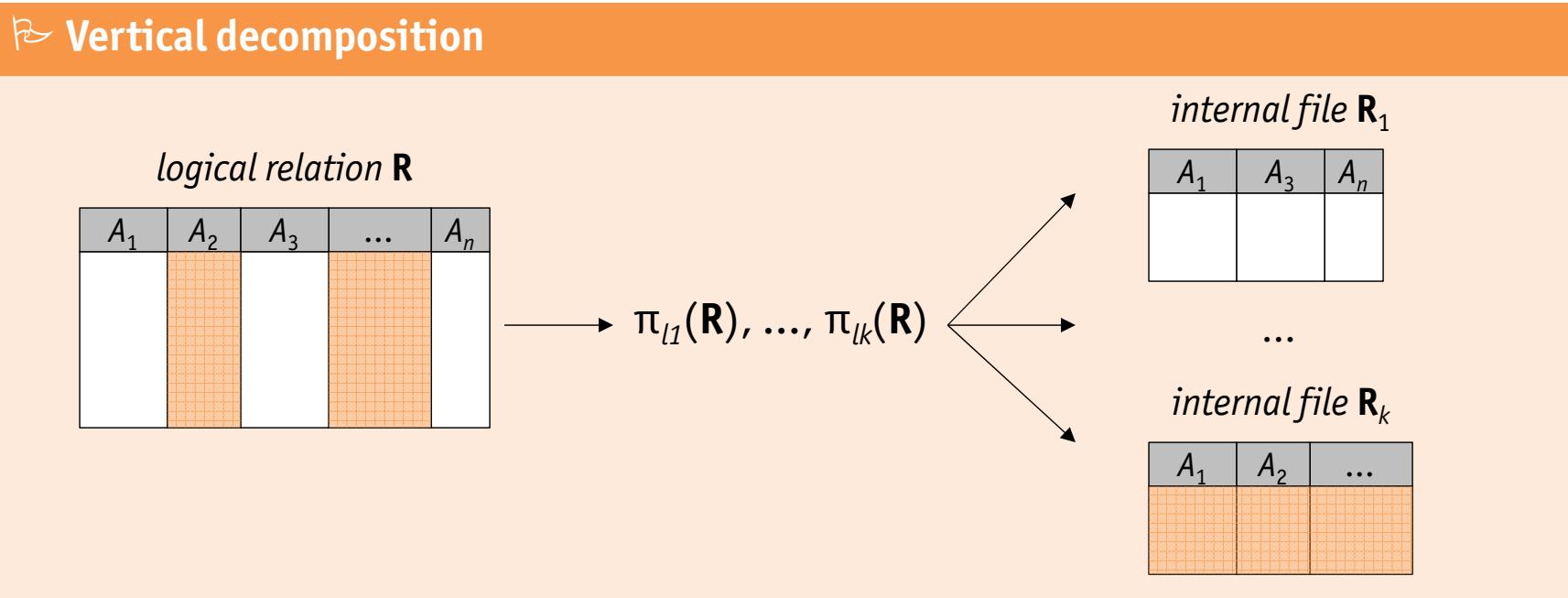


Horizontal Decomposition

- Each internal file may now be treated individually (in a separate segment) in further mapping to the disks
- **Pros**
 - access (read and write) to parts of relation are faster, if the only refer to a subset of the internal files, e.g., suitable selection queries
 - files/segments may be distributed across several servers
 - indexes can be defined independently on some of the internal files
- **Cons**
 - access to whole relation requires more effort (union operation)
 - change of attribute value may result in deletion from one and insertion into another internal file
 - index over all tuples may no longer be possible
- **Caution:** make sure that $c_1 \vee \dots \vee c_k \equiv \text{true!}$

Vertical Decomposition

- Map different attributes of logical tuples into fields of different internal sub-records stored in separate files
 - all resulting internal files share **different structures**
 - internal files can be viewed as materialized results of a set of **projection queries** on the logical relation

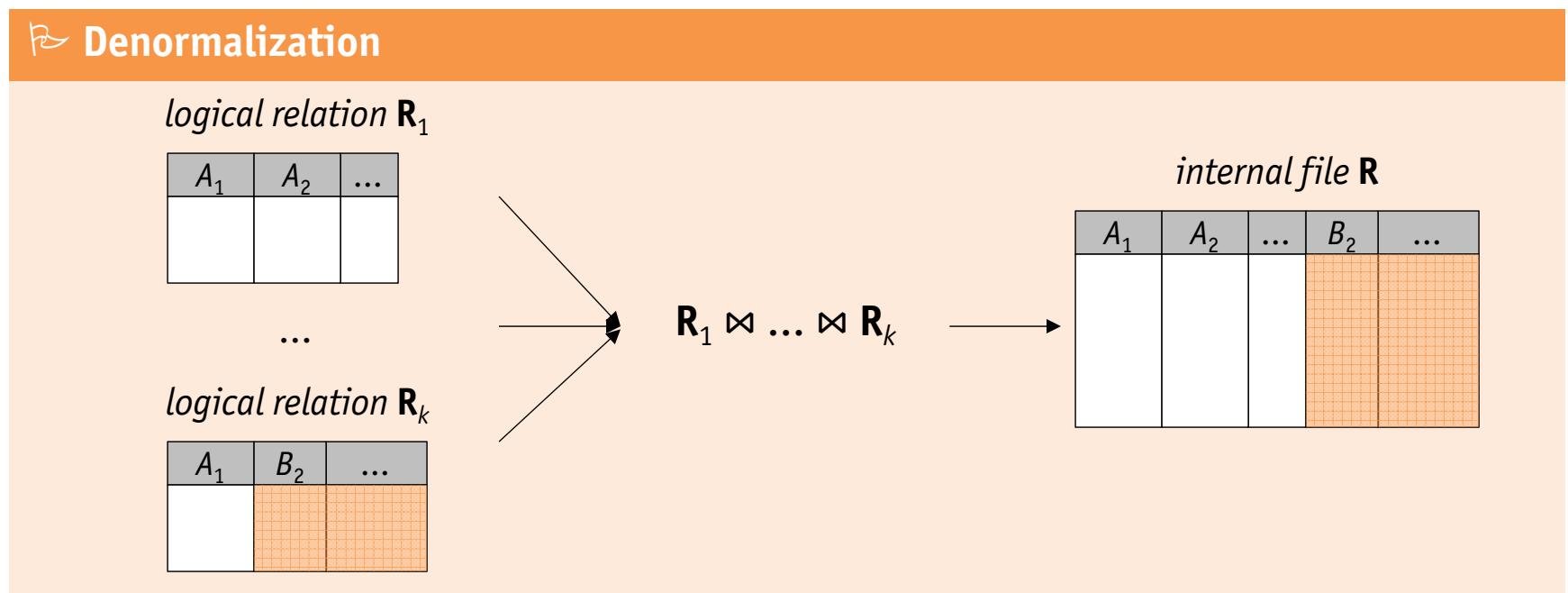


Vertical Decomposition

- Again, each internal file may now be treated individually in the further mapping to the disks
- **Pros**
 - access (read and write) to parts of tuples are faster, if they only refer to a subset of the internal files, e.g., suitable projection queries
 - files/segments may be distributed across several servers
 - indexes can be defined independently on some of the internal files
- **Cons**
 - access to whole tuples requires significant effort (join operation)
 - insert/delete needs to manipulate multiple internal files
 - indexes over multiple attributes are no longer possible, if they are stored in separate files
- **Caution:** make sure that $\pi_{l_1}(R) \bowtie \dots \bowtie \pi_{l_k}(R) \equiv R!$

Denormalization

- Map tuples related via a key-foreign key relationship from different relations into single internal records
 - internal files can be viewed as materialized results of a **join query** between two (or more) logical relations



Denormalization

- Internal file is treated as a whole in subsequent mapping to the disks and all fields of its larger records will stay closely together
- **Pros**
 - access to related tuples is much faster, since no join is required
 - index on join attributes can be exploited for all participating relations
- **Cons**
 - access to tuples of single relations requires extra effort (projection, duplicate elimination, more page I/O operations due to larger records)
 - insert/delete gets more tricky
 - change in join attribute value requires significant overhead
 - change in other attribute value requires more effort for replicated tuples
 - significant space overhead, depending on join selectivity
- **Caution:** make sure that $\forall_i : \pi_{R_i}(\mathbf{R}) = \mathbf{R}_i$, possibly using outer join

Tuning Queries and Views

- Examine queries and views that run much slower than expected to identify problems
 - verify that DBMS uses expected query evaluation plan
 - tune indexes to improve performance of expected plan
 - rewrite query or view to help optimizer find expected plan
- Possible reasons why optimizer is not finding best plan
 - selection conditions involving **NULL** values
 - selection conditions involving arithmetic or string expressions
 - selection conditions using **OR** connective
 - inability to recognize a sophisticated plan (e.g., index-only scan) for aggregation queries involving **GROUP BY** clause
 - nested queries
 - temporary relations

Tuning Queries and Views

 Exercise: How can these queries be rewritten to help the optimizer?

1. There is an index on **E.age**, but the optimizer does not use it to evaluate the condition **D.age = 2 * E.age**.
2. There are indexes on **E.hobby** and **E.age**, but the optimizer does not use them to evaluate the following query.

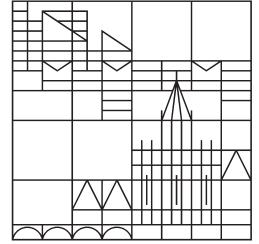
```
SELECT E.dno  
      FROM Employees E  
     WHERE E.hobby = "Stamps" OR E.age = 25
```

DBMS Benchmarks

- Several benchmarks exist to compare performance of different DBMS products with respect to a certain class of applications
 - **Online Transaction Processing (OLTP)**, e.g., TPC-E and TPC-C
 - **Offline Analytical Processing (OLAP)**, e.g., TPC-H and TPC-DS
 - **Object Databases**, e.g., 007 and PolePosition
 - **XML Databases**, e.g., XMark and XBench
 - **RDF Databases**, e.g., LUBM
- Criteria for database performance benchmarks
 - **portable**: can be run on all DBMS products
 - **easy to understand**
 - **scale** naturally to larger problem instances
 - measure **peak performance** (transactions per seconds, i.e., tps)
 - measure **price/performance ratio**, i.e., \$/tps

DBMS Benchmarks

- Benchmarks should be used with good understanding of what is measured and in what application environment DBMS is used
- How meaningful is a given benchmark?
 - performance of a complex system **cannot** be distilled into one number
 - good benchmarks have **suite of tasks** to test several relevant features
- How well does a benchmark reflect application workload?
 - **compare** expected application workload with workload of benchmark
 - select benchmark tasks that are **relevant** to intended application
 - consider **how** performance is measured (e.g., single-user vs. multi-user)
- Create or adapt benchmark
 - vendors often **tune** their DBMS to tasks of important benchmarks
 - **modify** standard benchmark tasks slightly
 - **replace** standard tasks with similar tasks from application workload



Database System Architecture and Implementation

THAT'S ALL FOLKS!