

Introduction

Torsten Grust



Architecture of a  
DBMS

Organizational  
Matters

# Architecture and Implementation of Database Systems

Summer 2014

Torsten Grust  
Wilhelm-Schickard-Institut für Informatik  
Universität Tübingen



Introduction

Torsten Grust



Architecture of a  
DBMS

Organizational  
Matters

# Chapter 1

## Introduction

Preliminaries and Organizational Matters

*Architecture and Implementation of Database Systems*  
Summer 2014

Torsten Grust  
Wilhelm-Schickard-Institut für Informatik  
Universität Tübingen



## Welcome all ...

...to this course whose lectures are primarily about digging in the mud of database system internals.

- While others talk about SQL and graphical query interfaces, we will
  - ① learn how DBMSs can **access files on hard disks without paying too much for I/O traffic,**
  - ② see how to **organize data on disk** and which kind of **“maps” for huge amounts of data** we can use to avoid to get lost,
  - ③ assess what it means to **sort/combine/filter data volumes that exceed main memory size** by far, and
  - ④ learn **how user queries are represented and executed** inside the database kernel.

Introduction

Torsten Grust



Architecture of a DBMS

Organizational Matters

# Architecture of a DBMS / Course Outline

Introduction

Torsten Grust



Architecture of a DBMS

Organizational Matters

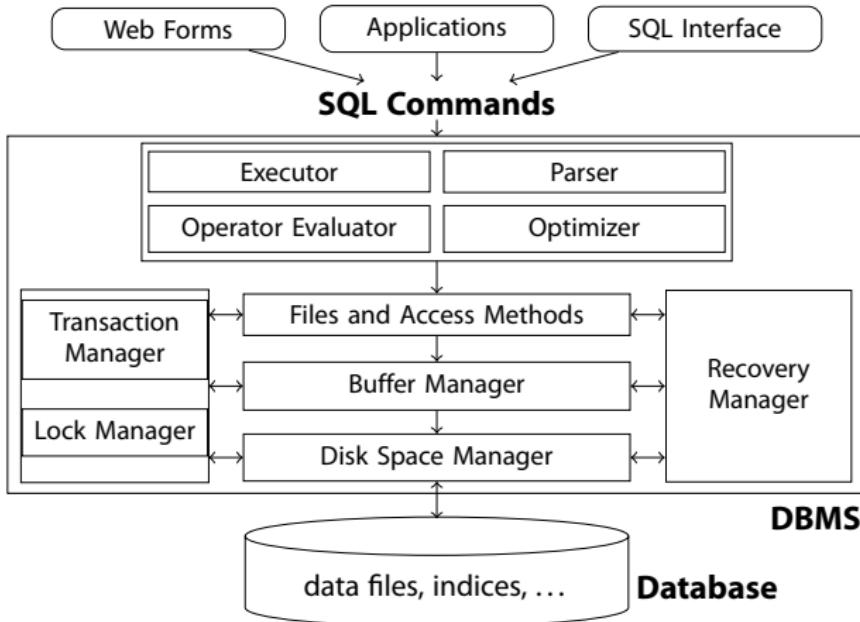


Figure inspired by Ramakrishnan/Gehrke: "Database Management Systems", McGraw-Hill 2003.

# Architecture of a DBMS / Course Outline

Introduction

Torsten Grust



Architecture of a DBMS

Organizational Matters

this course

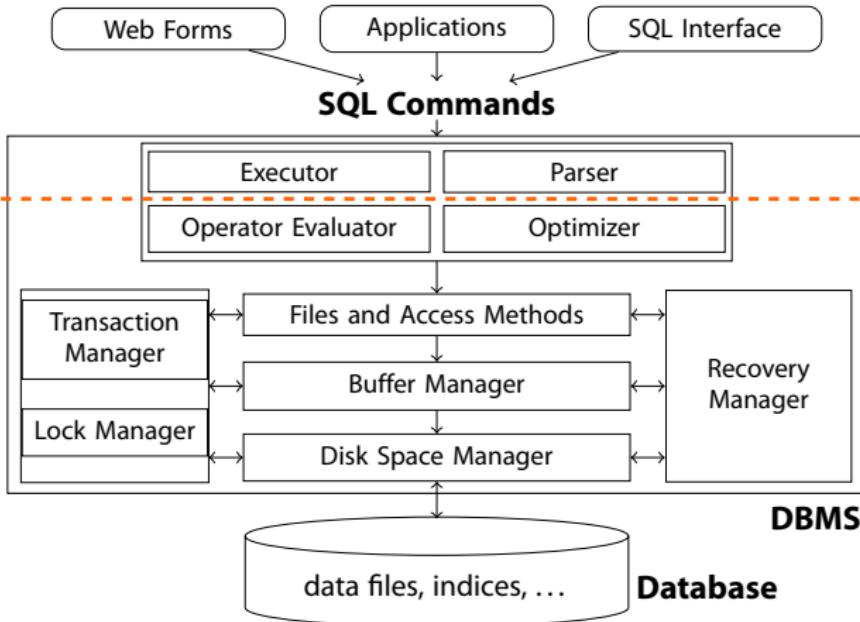


Figure inspired by Ramakrishnan/Gehrke: "Database Management Systems", McGraw-Hill 2003.

## A Few Words About Myself

### Torsten Grust

Originally from Hannover  
1989–1994 Student of Computer Science @ TU Clausthal  
1994–2004 Database Research @ U Konstanz  
1999 Promotion  
2000 Visiting Scientist @ IBM, Silicon Valley Lab,  
*DB2 Everyplace* Development  
2004 Habilitation  
2004–2005 Professor @ TU Clausthal  
2005–2008 Professor @ TU München  
since 9/2008 Professor @ U Tübingen

Web home [db.inf.uni-tuebingen.de](http://db.inf.uni-tuebingen.de)  
Coordinates B318, Sand 13  
+49 7071 29-78952 (Monika Weber)

Introduction

Torsten Grust



Architecture of a  
DBMS

Organizational  
Matters

# Organizational Matters

## Lectures

### When

Mondays, 10:15–11:45      Tuesdays, 10:15–11:45

### Where

Sand 6/7, gr. Hörsaal

[db.inf.uni-tuebingen.de/teaching/  
DatenbanksystemeIISS2014.html](http://db.inf.uni-tuebingen.de/teaching/DatenbanksystemeIISS2014.html)

Please visit regularly — we will post slides and course updates.

Introduction

Torsten Grust



Architecture of a  
DBMS

Organizational  
Matters

# Organizational Matters

## Lectures

### When

Mondays, 10:15–11:45  
Tuesdays, 10:15–11:45

### Where

Sand 6/7, gr. Hörsaal  
Sand 6/7, gr. Hörsaal

[db.inf.uni-tuebingen.de/teaching/  
DatenbanksystemeIISS2014.html](http://db.inf.uni-tuebingen.de/teaching/DatenbanksystemeIISS2014.html)

Please visit regularly — we will post slides and course updates.

## Exercises (Benjamin Dietrich)

### When

Thursdays, 14:15–15:45  
(starts April 17, 2014)

### Where

Sand 6/7, gr. Hörsaal

In-depth discussion of course topics, exercise sheets, plus occasional additional material. **Please register with ILIAS.**

Introduction

Torsten Grust



Architecture of a DBMS

Organizational Matters

## Examination

- **Written exam** to be held on **Monday, July 14, 2014** (this is the regular lecture slot).
- You will be allowed to bring **1 (one) hand-written double-sided piece of A4 paper** with notes.
- Further details will be posted.

Introduction

Torsten Grust



Architecture of a DBMS

Organizational Matters

## Reading Material

- Raghu Ramakrishnan and Johannes Gehrke.  
*Database Management Systems*. McGraw-Hill.
- Alfons Kemper and André Eickler.  
*Datenbanksysteme: Eine Einführung*. Oldenbourg Verlag.
- Dennis Shasha and Philippe Bonet.  
*Database Tuning*. Morgan Kaufmann Publishers.
- ...in fact, any book about advanced database topics and internals will do — pick your favorite.

Here and there, pointers () to specific research papers will be given and you are welcome to search for additional background reading. Use *Google Scholar* or similar search engines.

Introduction

Torsten Grust



Architecture of a DBMS

Organizational Matters

## These Slides...

- ...prepared/updated throughout the semester — **watch out for bugs** and please let me know. Thanks.
- Posted to course web home on the day before the lecture — **bring a printout and take notes.**

### Example

#### 📎 Open Issues/Questions

Take notes.

### Code Snippets, Algorithms

#### DB2. IBM DB2 Specifics

If possible and insightful, discuss how IBM DB2 does things.



#### PostgreSQL Specifics

Ditto, but related to the glorious PostgreSQL (v9.x) system.

Introduction

Torsten Grust



Architecture of a DBMS

Organizational Matters

## Before We Begin

**Questions?**

**Comments?**

**Suggestions?**

Introduction

Torsten Grust



Architecture of a  
DBMS

Organizational  
Matters

# Chapter 2

## Storage

Disks, Buffer Manager, Files...

*Architecture and Implementation of Database Systems*  
Summer 2014

Storage

Torsten Grust



Magnetic Disks

Access Time

Sequential vs. Random Access

I/O Parallelism

RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks

Network-Based Storage

Managing Space

Free Space Management

Buffer Manager

Pinning and Unpinning

Replacement Policies

Databases vs. Operating Systems

Files and Records

Heap Files

Free Space Management

Inside a Page

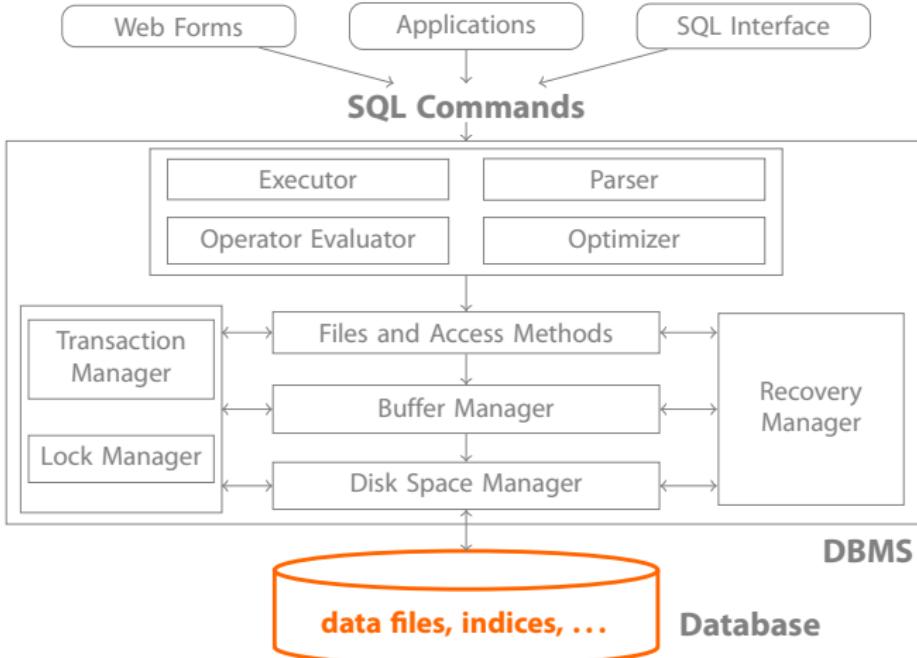
Alternative Page Layouts

Recap

Torsten Grust  
Wilhelm-Schickard-Institut für Informatik  
Universität Tübingen



# Database Architecture



Storage

Torsten Grust



Magnetic Disks

Access Time  
Sequential vs. Random Access

I/O Parallelism

RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space

Free Space Management

Buffer Manager

Pinning and Unpinning  
Replacement Policies

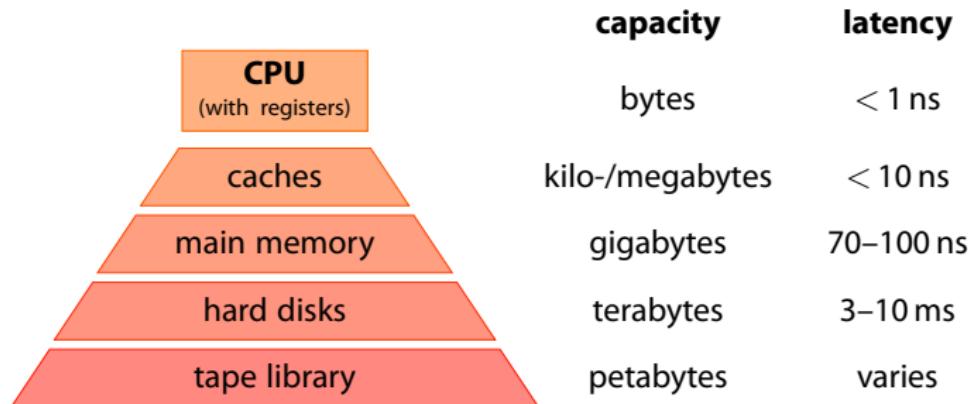
Databases vs.  
Operating Systems

Files and Records

Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

# The Memory Hierarchy



- Fast—but expensive and small—memory close to CPU
- Larger, slower memory at the periphery
- DBMSs try to **hide latency** by using the fast memory as a **cache**.



**Magnetic Disks**  
Access Time  
Sequential vs. Random Access

**I/O Parallelism**  
RAID Levels 1, 0, and 5

**Alternative Storage Techniques**  
Solid-State Disks  
Network-Based Storage

**Managing Space**  
Free Space Management

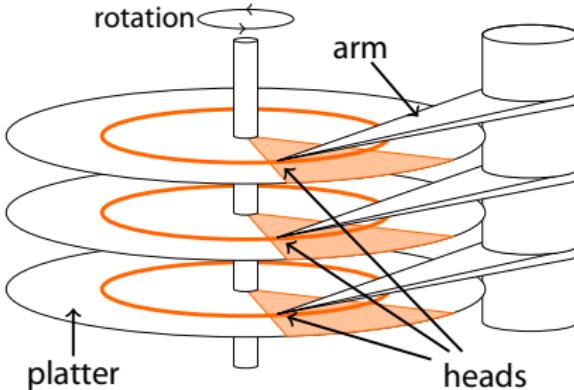
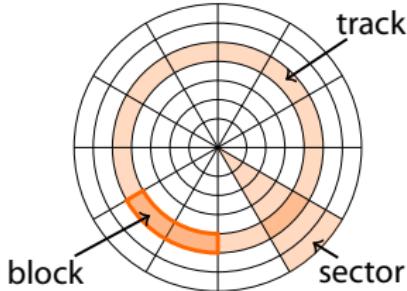
**Buffer Manager**  
Pinning and Unpinning  
Replacement Policies

**Databases vs. Operating Systems**

**Files and Records**  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

**Recap**

# Magnetic Disks



- A stepper motor positions an array of disk heads on the requested track
- Platters (disks) steadily rotate
- Disks are managed in blocks: the system reads/writes data one block at a time



Photo: <http://www.metallurgy.utah.edu/>

Storage  
Torsten Grust



**Magnetic Disks**  
Access Time  
Sequential vs. Random Access  
  
**I/O Parallelism**  
RAID Levels 1, 0, and 5  
  
**Alternative Storage Techniques**  
Solid-State Disks  
Network-Based Storage

**Managing Space**  
Free Space Management

**Buffer Manager**  
Pinning and Unpinning  
Replacement Policies

**Databases vs. Operating Systems**

**Files and Records**  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

**Recap**

## 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** to read/write a given block:

### Definition (Access Time)

- ① Move disk arms to desired track (**seek time**  $t_s$ )
- ② Disk controller waits for desired block to rotate under disk head (**rotational delay**  $t_r$ )
- ③ Read/write data (**transfer time**  $t_{tr}$ )

$$\Rightarrow \text{access time: } t = t_s + t_r + t_{tr}$$



Storage  
Torsten Grust

### Magnetic Disks

- Access Time
  - Sequential vs. Random Access
- I/O Parallelism
- RAID Levels 1, 0, and 5

### Alternative Storage Techniques

- Solid-State Disks
- Network-Based Storage

### Managing Space

- Free Space Management

### Buffer Manager

- Pinning and Unpinning
- Replacement Policies

### Databases vs. Operating Systems

### Files and Records

- Heap Files
- Free Space Management
- Inside a Page
- Alternative Page Layouts

### Recap

## Example: Seagate Cheetah 15K.7 (600 GB, server-class drive)

- Seagate Cheetah 15K.7 performance characteristics:
  - 4 disks, 8 heads, avg. 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 data block?

Storage	Torsten Grust
	Magnetic Disks
Access Time	Sequential vs. Random Access
I/O Parallelism	RAID Levels 1, 0, and 5
Alternative Storage Techniques	Solid-State Disks Network-Based Storage
Managing Space	Free Space Management
Buffer Manager	Pinning and Unpinning Replacement Policies
Databases vs. Operating Systems	
Files and Records	Heap Files Free Space Management Inside a Page Alternative Page Layouts
Recap	

## Example: Seagate Cheetah 15K.7 (600 GB, server-class drive)

- Seagate Cheetah 15K.7 performance characteristics:
  - 4 disks, 8 heads, avg. 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 data block?

average seek time

$$t_s = 3.40 \text{ ms}$$

average rotational delay:  $\frac{1}{2} \cdot \frac{1}{15\,000 \text{ min}^{-1}}$

$$t_r = 2.00 \text{ ms}$$

transfer time for 8 kB:  $\frac{8 \text{ kB}}{163 \text{ MB/s}}$

$$t_{tr} = 0.05 \text{ ms}$$

**access time** for an 8 kB data block

$$t = 5.45 \text{ ms}$$



### Magnetic Disks

Access Time

Sequential vs. Random Access

### I/O Parallelism

RAID Levels 1, 0, and 5

### Alternative Storage Techniques

Solid-State Disks

Network-Based Storage

### Managing Space

Free Space Management

### Buffer Manager

Pinning and Unpinning

Replacement Policies

### Databases vs. Operating Systems

### Files and Records

Heap Files

Free Space Management

Inside a Page

Alternative Page Layouts

### Recap

## Sequential vs. Random Access

### Example (Read 1 000 blocks of size 8 kB)

- **random access:**

$$t_{\text{rnd}} = 1000 \cdot 5.45 \text{ ms} = 5.45 \text{ s}$$

- **sequential read of adjacent blocks:**

$$\begin{aligned} t_{\text{seq}} &= t_s + t_r + 1000 \cdot t_{tr} + 16 \cdot t_{s,\text{track-to-track}} \\ &= 3.40 \text{ ms} + 2.00 \text{ ms} + 50 \text{ ms} + 3.2 \text{ ms} \approx 58.6 \text{ ms} \end{aligned}$$

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

- ⇒ Sequential I/O is **much** faster than random I/O
- ⇒ **Avoid random I/O** 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



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

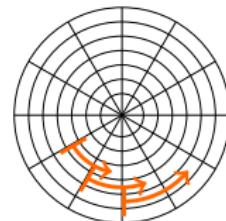
Recap

## Performance Tricks

- Disk manufacturers play a number of tricks to improve performance:

### track skewing

Align sector 0 of each track to avoid rotational delay during longer sequential scans



### request scheduling

If **multiple requests** have to be served, choose the one that requires the smallest arm movement (SPTF: shortest positioning time first, elevator algorithms)

### zoning

Outer tracks are longer than the inner ones. Therefore, divide outer tracks into more sectors than inner tracks

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Evolution of Hard Disk Technology

Disk seek and rotational latencies have only marginally improved over the last years ( $\approx 10\%$  per year)

**But:**

- Throughput (i.e., transfer rates) improve by  $\approx 50\%$  per year
- Hard disk capacity grows by  $\approx 50\%$  every year

**Therefore:**

- Random access cost hurts even more as time progresses

### Example (5 Years Ago: Seagate Barracuda 7200.7)

Read 1K blocks of 8 kB sequentially/randomly: 397 ms / 12 800 ms



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Ways to Improve I/O Performance

The latency penalty is hard to avoid

But:

- Throughput can be increased rather easily by exploiting **parallelism**
- **Idea:** Use multiple disks and access them in parallel, try to hide latency

### DB2. TPC-C: An industry benchmark for OLTP

A recent #1 system (IBM DB2 9.5 on AIX) uses

- 10,992 disk drives (73.4 GB each, 15,000 rpm) (!)  
plus 8 146.8 GB internal SCSI drives,
- connected with 68 4 Gbit fibre channel adapters,
- yielding 6 mio transactions per minute

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

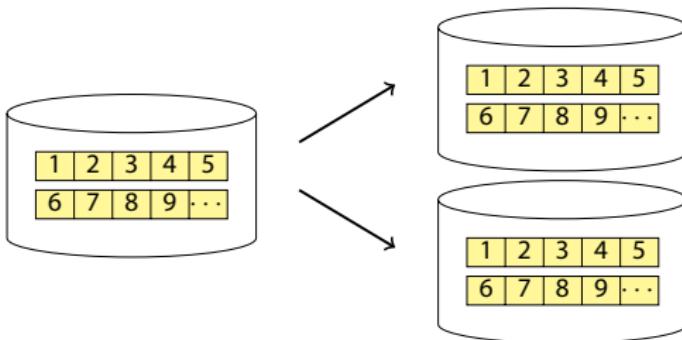
Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Disk Mirroring

- Replicate data onto multiple disks:



- Achieves I/O parallelism only for **reads**
- Improved failure tolerance—can survive one disk failure
- This is also known as **RAID 1** (mirroring without parity)  
**(RAID: Redundant Array of Inexpensive Disks)**

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

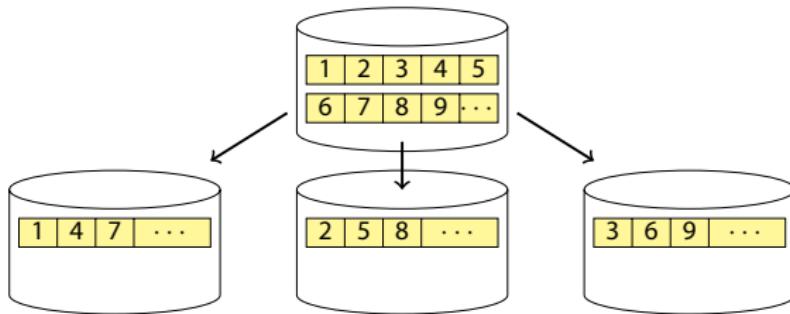
Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

# Disk Striping

- **Distribute** data over disks:



- Full I/O parallelism for **read and write** operations
- High failure risk (here: 3 times risk of single disk failure)!
- Also known as **RAID 0** (striping without parity)



## Magnetic Disks

Access Time  
Sequential vs. Random Access

## I/O Parallelism

RAID Levels 1, 0, and 5

## Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

## Managing Space

Free Space Management

## Buffer Manager

Pinning and Unpinning  
Replacement Policies

## Databases vs. Operating Systems

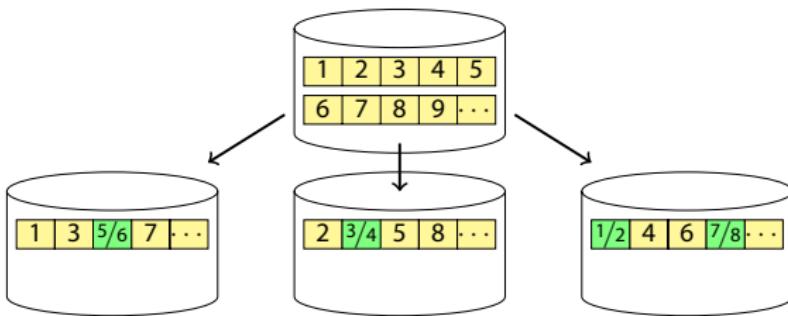
## Files and Records

Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

## Recap

## Disk Striping with Parity

- Distribute data and **parity** information over  $\geq 3$  disks:



- High I/O parallelism
- Fault tolerance: any **one disk may fail** without data loss (with dual parity/RAID 6: two disks may fail)
- Distribute parity (e.g., XOR) information over disks, separating data and associated parity
- Also known as **RAID 5** (striping with distributed parity)

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

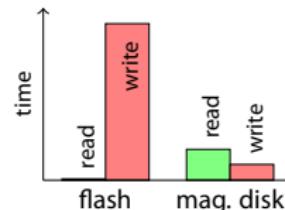
Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Solid-State Disks

Solid state disks (SSDs) have emerged as an alternative to conventional hard disks

- SSDs provide **very low-latency random read access** ( $< 0.01$  ms)
- **Random writes**, however, are significantly **slower** than on traditional magnetic drives:
  - ① (Blocks of) Pages have to be **erased** before they can be updated
  - ② Once pages have been erased, sequentially writing them is almost as fast as reading



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

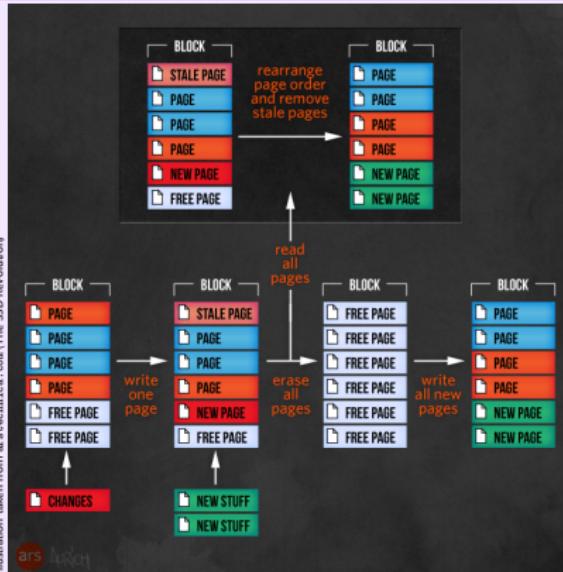
Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

# SSDs: Page-Level Writes, Block-Level Deletes

- Typical **page size**: 128 kB
- SSDs erase **blocks of pages**: block  $\approx$  64 pages (8 MB)

## Example (Perform block-level delete to accomodate new data pages)



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Example: Seagate Pulsar.2 (800 GB, server-class solid state drive)

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

 What is the access time to read an 8 KB data block?

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Example: Seagate Pulsar.2 (800 GB, server-class solid state drive)

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

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

no seek time  $t_s = 0.00 \text{ ms}$

no rotational delay:  $t_r = 0.00 \text{ ms}$

transfer time for 8 kB:  $\frac{128 \text{ kB}}{370 \text{ MB/s}}$   $t_{tr} = 0.30 \text{ ms}$

**access time** for an 8 kB data block  $t = 0.30 \text{ ms}$



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Sequential vs. Random Access with SSDs

### Example (Read 1 000 blocks of size 8 kB)

- **random access:**

$$t_{\text{rnd}} = 1000 \cdot 0.30 \text{ ms} = 0.3 \text{ s}$$

- **sequential read of adjacent pages:**

$$t_{\text{seq}} = \left\lceil \frac{1000 \cdot 8 \text{ kB}}{128 \text{ kB}} \right\rceil \cdot t_{\text{tr}} \approx 18.9 \text{ ms}$$

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

- ⇒ Sequential I/O still beats random I/O  
(but random I/O is more feasible again)
- Adapting database technology to these characteristics is a current research topic



## Network-Based Storage

Today the network is **not** a bottleneck any more:

Storage media/interface	Transfer rate
Hard disk	100–200 MB/s
Serial ATA	375 MB/s
Ultra-640 SCSI	640 MB/s
10-Gbit Ethernet	1,250 MB/s
Infiniband QDR	12,000 MB/s
For comparison (RAM):	
PC2-5300 DDR2-SDRAM	10.6 GB/s
PC3-12800 DDR3-SDRAM	25.6 GB/s

⇒ Why not use the network for database storage?



## Storage Area Network (SAN)

- **Block-based** network access to storage:
  - SAN emulate interface of block-structured disks (“*read block #42 of disk 10*”)
  - This is unlike network file systems (e.g., NFS, CIFS)
- SAN storage devices typically abstract from RAID or physical disks and present logical drives to the DBMS
  - Hardware acceleration and simplified maintainability
- Typical setup: local area network with multiple participating servers and storage resources
  - Failure tolerance and increased flexibility

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Grid or Cloud Storage

Internet-scale enterprises employ clusters with 1000s commodity PCs (e.g., Amazon, Google, Yahoo!):

- **system cost**  $\leftrightarrow$  **reliability** and **performance**,
- use **massive replication** for data storage

Spare CPU cycles and disk space are sold as a **service**:

### Amazon's Elastic Computing Cloud (EC2)

Use Linux-based compute cluster by the hour ( $\sim 10 \text{ \textcent/h}$ ).

### Amazon's Simple Storage System (S3)

"Infinite" store for objects between 1 B and 5 TB in size, organized in a map data structure (key  $\mapsto$  object)

- Latency: 100 ms to 1 s (not impacted by load)
- pricing  $\approx$  disk drives (but addl. cost for access)

⇒ Building a database on S3?  
(↗ Brantner *et al.*, SIGMOD 2008)



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

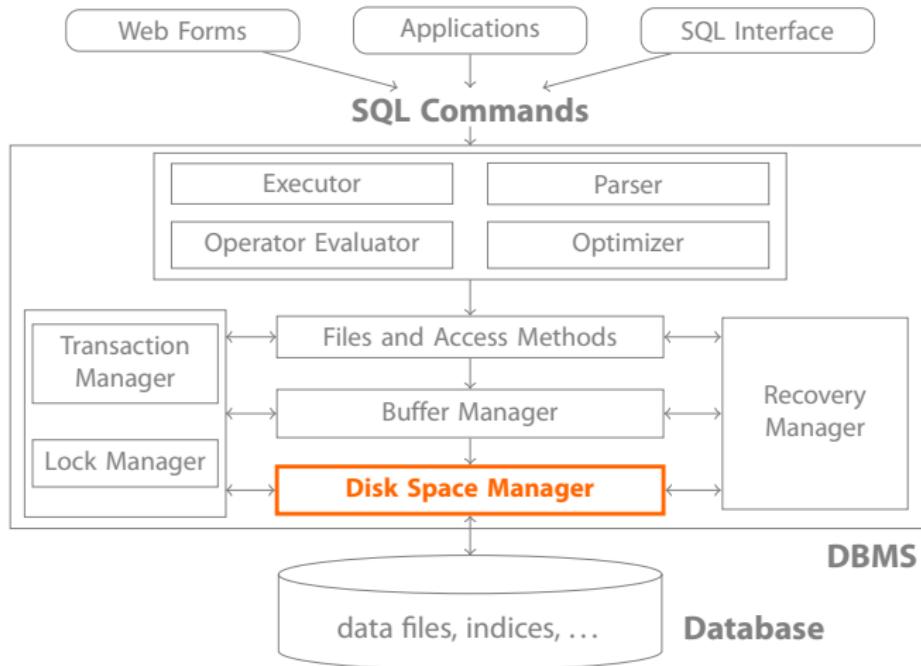
Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

# Managing Space



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

# Managing Space

## Definition (Disk Space Manager)

- Abstracts from the gory details of the underlying storage (disk space manager talks to I/O controller and initiates I/O)
- DBMS issues allocate/deallocate and read/write commands
- Provides the concept of a **page** (typically 4–64 KB) as a unit of storage to the remaining system components
- Maintains a locality-preserving mapping

page number  $\mapsto$  physical location ,

where a physical location could be, e.g.,

- an OS file name and an offset within that file,
- head, sector, and track of a hard drive, or
- tape number and offset for data stored in a tape library

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Empty Pages

The disk space manager also keeps track of **used/free blocks** (deallocation and subsequent allocation may create **holes**):

- ① Maintain a **linked list** of free pages
  - When a page is no longer needed, add it to the list
- ② Maintain a **bitmap** reserving one bit for each page
  - Toggle bit  $n$  when page  $n$  is (de-)allocated

### 📎 Allocation of contiguous pages

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

Which of the techniques (1 or 2) would you choose to support this?

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Empty Pages

The disk space manager also keeps track of **used/free blocks** (deallocation and subsequent allocation may create **holes**):

- ① Maintain a **linked list** of free pages
  - When a page is no longer needed, add it to the list
- ② Maintain a **bitmap** reserving one bit for each page
  - Toggle bit  $n$  when page  $n$  is (de-)allocated

### 📎 Allocation of contiguous pages

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

Which of the techniques (1 or 2) would you choose to support this?

This is a lot easier to do with a free page bitmap (option 2).

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

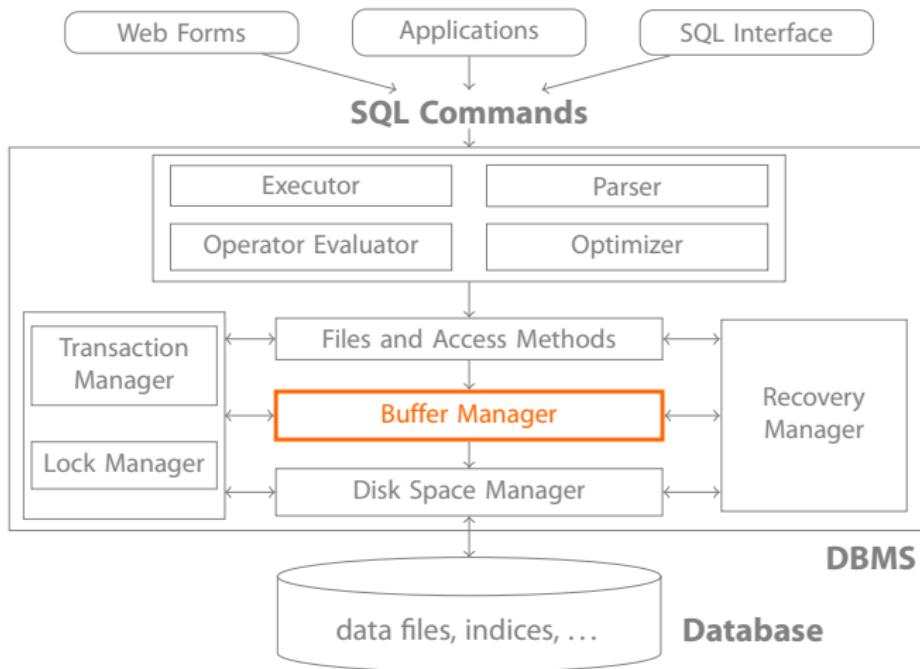
Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

# Buffer Manager



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

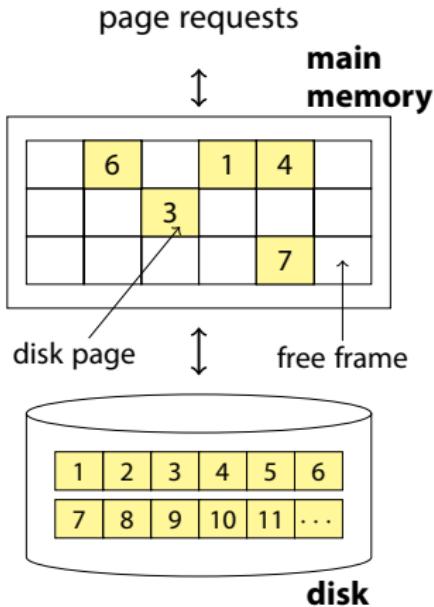
**Buffer Manager**  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

# Buffer Manager



## Definition (Buffer Manager)

- Mediates between external storage and main memory,
- Manages a designated main memory area, the **buffer pool**, for this task

Disk pages are brought into memory as needed and loaded into memory **frames**

A **replacement policy** decides which page to evict when the buffer is full

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Interface to the Buffer Manager

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, load it into memory if necessary and mark page as clean ( $\neg\text{dirty}$ ). Returns a reference to the frame containing *pageno*.

### unpin (*pageno*, *dirty*)

Release page number *pageno*, making it a candidate for eviction. Must set *dirty* = true if page was modified.

### 🔗 Why do we need the *dirty* bit?



Torsten Grust



#### Magnetic Disks

Access Time

Sequential vs. Random Access

#### I/O Parallelism

RAID Levels 1, 0, and 5

#### Alternative Storage Techniques

Solid-State Disks

Network-Based Storage

#### Managing Space

Free Space Management

#### Buffer Manager

Pinning and Unpinning

Replacement Policies

#### Databases vs. Operating Systems

#### Files and Records

Heap Files

Free Space Management

Inside a Page

Alternative Page Layouts

#### Recap

## Interface to the Buffer Manager

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, load it into memory if necessary and mark page as clean ( $\neg\text{dirty}$ ). Returns a reference to the frame containing *pageno*.

### unpin (*pageno*, *dirty*)

Release page number *pageno*, making it a candidate for eviction. Must set *dirty* = true if page was modified.

### 🔗 Why do we need the *dirty* bit?

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

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Proper pin ()/unpin () Nesting

- Any database transaction is required to properly “bracket” any page operation using pin () and unpin () calls:

### A read-only transaction

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

- Proper bracketing enables the systems to keep a count of active users (e.g., transactions) of a page



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Implementation of pin()

### Function pin(*pageno*)

```
1 if buffer pool already contains pageno then
2   pinCount (pageno)  $\leftarrow$  pinCount (pageno) + 1 ;
3   return address of frame holding pageno ;
4 else
5   select a victim frame v using the replacement policy ;
6   if dirty (page in v) then
7     write page in v to disk ;
8   read page pageno from disk into frame v ;
9   pinCount (pageno)  $\leftarrow$  1 ;
10  dirty (pageno)  $\leftarrow$  false ;
11  return address of frame v ;
```

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Implementation of `unpin()`

### Function `unpin(pageno, dirty)`

```
1 pinCount (pageno) ← pinCount (pageno) – 1 ;  
2 dirty (pageno) ← dirty (pageno) ∨ dirty ;
```

☞ Why don't we write pages back to disk during `unpin()`?

Storage

Torsten Grust



Magnetic Disks

Access Time

Sequential vs. Random Access

I/O Parallelism

RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks

Network-Based Storage

Managing Space

Free Space Management

Buffer Manager

Pinning and Unpinning

Replacement Policies

Databases vs. Operating Systems

Files and Records

Heap Files

Free Space Management

Inside a Page

Alternative Page Layouts

Recap

## Implementation of `unpin ()`

### Function `unpin(pageno, dirty)`

```
1 pinCount (pageno) ← pinCount (pageno) – 1 ;  
2 dirty (pageno) ← dirty (pageno) ∨ dirty ;
```

### ☞ Why don't we write pages back to disk during `unpin ()`?

Well, we could ...

- + 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.

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager

Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Concurrent Writes?

### Conflicting changes to a block

Assume the following:

- ① The same page  $p$  is requested by more than one transaction (i.e.,  $\text{pinCount}(p) > 1$ ), and
- 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 (see "Introduction to Database Systems", transaction management, locks).

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



Storage

Torsten Grust

#### Magnetic Disks

Access Time

Sequential vs. Random Access

#### I/O Parallelism

RAID Levels 1, 0, and 5

#### Alternative Storage Techniques

Solid-State Disks

Network-Based Storage

#### Managing Space

Free Space Management

#### Buffer Manager

Pinning and Unpinning

Replacement Policies

#### Databases vs. Operating Systems

#### Files and Records

Heap Files

Free Space Management

Inside a Page

Alternative Page Layouts

#### Recap

## Replacement Policies

The effectiveness of the buffer manager's **caching** functionality can depend on the **replacement policy** it uses, e.g.,

### Least Recently Used (LRU)

Evict the page whose latest unpin () is longest ago

### LRU- $k$

Like LRU, but considers the  $k$  latest unpin () calls, not just the latest

### Most Recently Used (MRU)

Evict the page that has been unpinned most recently

### Random

Pick a victim randomly

📎 Rationales behind each of these policies?



Storage

Torsten Grust



Magnetic Disks

Access Time

Sequential vs. Random Access

I/O Parallelism

RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks

Network-Based Storage

Managing Space

Free Space Management

Buffer Manager

Pinning and Unpinning

Replacement Policies

Databases vs. Operating Systems

Files and Records

Heap Files

Free Space Management

Inside a Page

Alternative Page Layouts

Recap

## Example Policy: Clock (“Second Chance”)

- Simulate an LRU policy with less overhead (no LRU queue reorganization on every frame reference):

### Clock (“Second Chance”)

- ① Number the  $N$  frames in the buffer pool  $0, \dots, N - 1$ ; initialize  $\text{current} \leftarrow 0$ , maintain a bit array  $\text{referenced}[0, \dots, N - 1]$  initialized to all 0
- ② In  $\text{pin}(p)$ : load  $p$  into buffer pool (if needed), assign  $\text{referenced}[\text{frame-of}(p)] \leftarrow 1$
- ③ In  $\text{pin}(p)$ : if we need to find a victim, consider page  $\text{current}$ ; if  $\text{referenced}[\text{current}] = 0$ ,  $\text{current}$  is the victim; otherwise,  $\text{referenced}[\text{current}] \leftarrow 0$ ,  $\text{current} \leftarrow \text{current} + 1 \bmod N$ , repeat ③

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Heuristic Policies Can Fail

The mentioned policies, including LRU, are **heuristics** only and may fail miserably in certain scenarios.

### Example (A Challenge for LRU)

A number of transactions want to scan the same sequence of pages (consider a repeated `SELECT * FROM R`).

Assume a buffer pool capacity of 10 pages.

- ① Let the size of relation `R` be 10 or less pages.  
How many I/O (actual disk page reads) do you expect?
- ② Now grow `R` by one page.  
How about the number of I/O operations in this case?

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## More Challenges for LRU

- ① Transaction  $T_1$  repeatedly accesses a fixed set of pages; transaction  $T_2$  performs a sequential scan of the database pages.
- ② Assume a  $B^+$ -tree-indexed relation  $R$ .  $R$  occupies 10,000 data pages  $R_i$ , the  $B^+$ -tree occupies one root node and 100 index leaf nodes  $I_k$ .

Transactions perform repeated random index key lookups on  $R \Rightarrow$  **page access pattern** (ignores  $B^+$ -tree root node):

$I_1, R_1, I_2, R_2, I_3, R_3, \dots$

💡 How will LRU perform in this case?



Storage

Torsten Grust



Magnetic Disks

Access Time

Sequential vs. Random Access

I/O Parallelism

RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks

Network-Based Storage

Managing Space

Free Space Management

Buffer Manager

Pinning and Unpinning

Replacement Policies

Databases vs. Operating Systems

Files and Records

Heap Files

Free Space Management

Inside a Page

Alternative Page Layouts

Recap

## More Challenges for LRU

- ① Transaction  $T_1$  repeatedly accesses a fixed set of pages; transaction  $T_2$  performs a sequential scan of the database pages.
- ② Assume a  $B^+$ -tree-indexed relation  $R$ .  $R$  occupies 10,000 data pages  $R_i$ , the  $B^+$ -tree occupies one root node and 100 index leaf nodes  $I_k$ .

Transactions perform repeated random index key lookups on  $R \Rightarrow$  **page access pattern** (ignores  $B^+$ -tree root node):

$I_1, R_1, I_2, R_2, I_3, R_3, \dots$

### 📎 How will LRU perform in this case?

With LRU, 50 % of the buffered pages will be pages of  $R$ . However, the probability of re-accessing page  $R_i$  only is  $1/10,000$ .



Storage

Torsten Grust



Magnetic Disks

Access Time

Sequential vs. Random Access

I/O Parallelism

RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks

Network-Based Storage

Managing Space

Free Space Management

Buffer Manager

Pinning and Unpinning

Replacement Policies

Databases vs. Operating Systems

Files and Records

Heap Files

Free Space Management

Inside a Page

Alternative Page Layouts

Recap

## Buffer Management in Practice

### Prefetching

Buffer managers try to anticipate page requests to overlap CPU and I/O operations:

- **Speculative prefetching:** Assume sequential scan and automatically read ahead.
- **Prefetch lists:** Some database algorithms can instruct the buffer manager with a list of pages to prefetch.

### Page fixing/hating

Higher-level code may request to **fix** a page if it may be useful in the near future (e.g., nested-loop join).

Likewise, an operator that **hates** a page will not access it any time soon (e.g., table pages in a sequential scan).

### Partitioned buffer pools

E.g., maintain separate pools for indexes and tables.

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Databases vs. Operating Systems

**Wait!** Didn't we just re-invent the operating system?



**Yes,**

- disk space management and buffer management very much look like **file management** and **virtual memory** in OSs.

**But,**

- a DBMS may be much more aware of the **access patterns** of certain operators (prefetching, page fixing/hating),
- concurrency control often calls for a **prescribed order** of write operations,
- technical reasons may make OS tools unsuitable for a database (*e.g.*, file size limitation, platform independence).

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

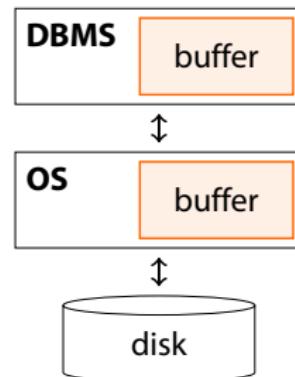
Recap

## Databases vs. Operating Systems

In fact, databases and operating systems sometimes interfere:

- Operating system and buffer manager effectively buffer the same data twice.
- Things get really bad if parts of the DBMS buffer get swapped out to disk by OS VM manager.
- Therefore, database systems try to **turn off** certain OS features or services.

⇒ **Raw disk access instead of OS files.**



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

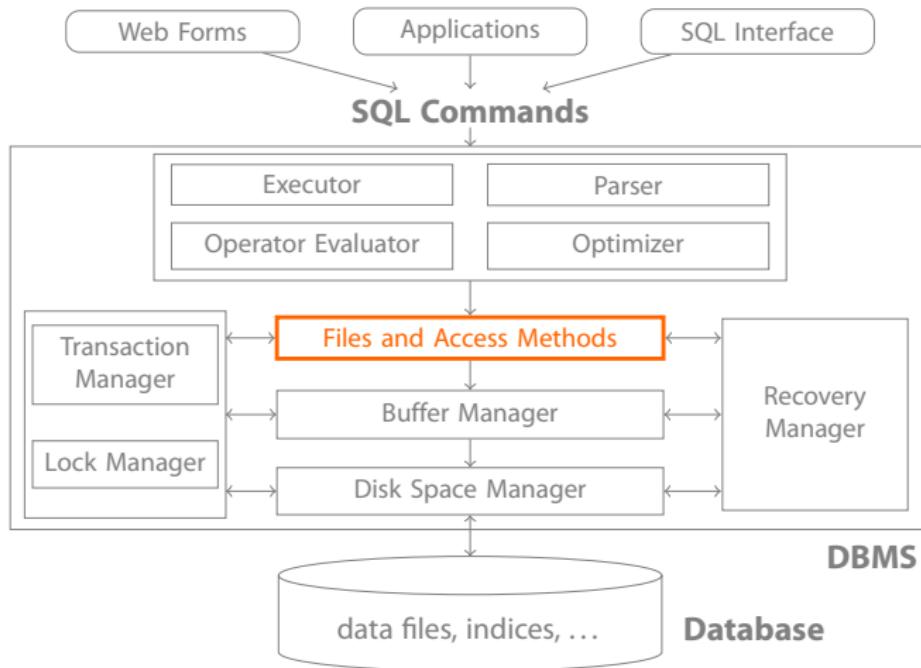
Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

# Files and Records



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

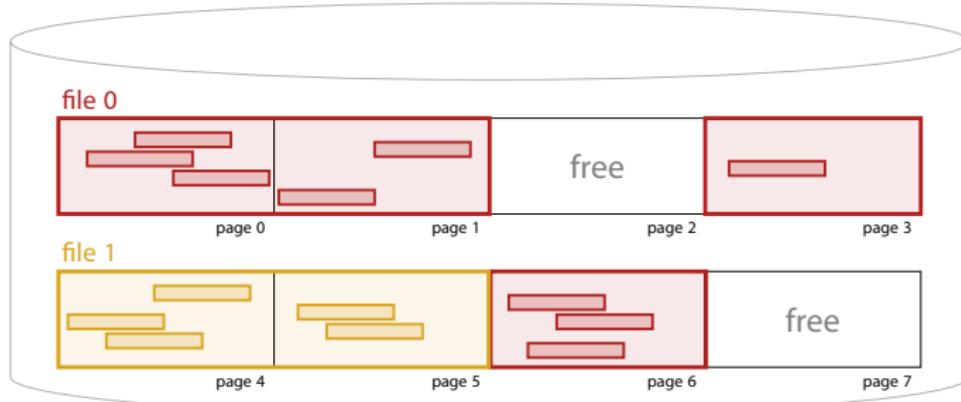
Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Database Files

- So far we have talked about **pages**. Their management is oblivious with respect to their actual content.
- On the conceptual level, a DBMS primarily manages **tables of tuples** and **indexes**.
- Such tables are implemented as **files of records**:
  - A **file** consists of **one or more pages**,
  - each **page** contains **one or more records**,
  - each **record** corresponds to **one tuple**:



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Database Heap Files

The most important type of files in a database is the **heap file**. It stores records in **no particular order** (in line with, e.g., the SQL semantics):

### Typical heap file interface

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

**N.B.** Record ids ( $rid$ ) are used like **record addresses** (or pointers). The heap file structure maps a given  $rid$  to the page containing the record.



Storage

Torsten Grust



### Magnetic Disks

Access Time

Sequential vs. Random Access

### I/O Parallelism

RAID Levels 1, 0, and 5

### Alternative Storage Techniques

Solid-State Disks

Network-Based Storage

### Managing Space

Free Space Management

### Buffer Manager

Pinning and Unpinning

Replacement Policies

### Databases vs. Operating Systems

### Files and Records

Heap Files

Free Space Management

Inside a Page

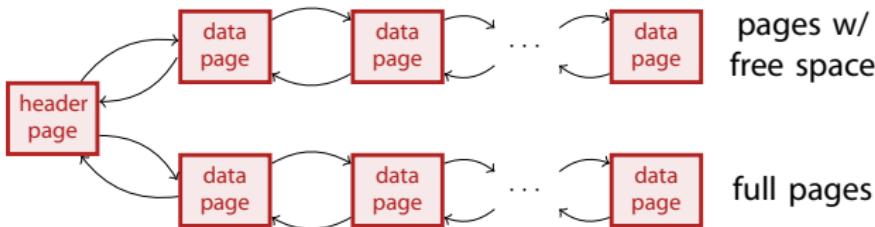
Alternative Page Layouts

### Recap

## Heap Files

### (Doubly) Linked list of pages:

Header page allocated when `createFile( $n$ )` is called—initially both page lists are empty:



- + easy to implement
- most pages will end up in free page list
- might have to search many pages to place a (large) record



#### Magnetic Disks

Access Time  
Sequential vs. Random Access

#### I/O Parallelism

RAID Levels 1, 0, and 5

#### Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

#### Managing Space

Free Space Management

#### Buffer Manager

Pinning and Unpinning  
Replacement Policies

#### Databases vs. Operating Systems

#### Files and Records

Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

#### Recap

## Heap Files

### Operation `insertRecord(f, r)` for linked list of pages

- ① Try to find a page  $p$  in the free list with free space  $> |r|$ ; should this fail, ask the disk space manager to allocate a new page  $p$
- ② Write record  $r$  to page  $p$
- ③ Since, generally,  $|r| \ll |p|$ ,  $p$  will belong to the list of pages with free space
- ④ A unique  $rid$  for  $r$  is generated and returned to the caller

#### 📎 Generating sensible record ids ( $rid$ )

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

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Heap Files

### Operation `insertRecord(f, r)` for linked list of pages

- ① Try to find a page  $p$  in the free list with free space  $> |r|$ ; should this fail, ask the disk space manager to allocate a new page  $p$
- ② Write record  $r$  to page  $p$
- ③ Since, generally,  $|r| \ll |p|$ ,  $p$  will belong to the list of pages with free space
- ④ A unique  $rid$  for  $r$  is generated and returned to the caller

#### 📎 Generating sensible record ids ( $rid$ )

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  $p$ :

$$\langle \text{pageno } p, \text{slotno } r \rangle$$

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

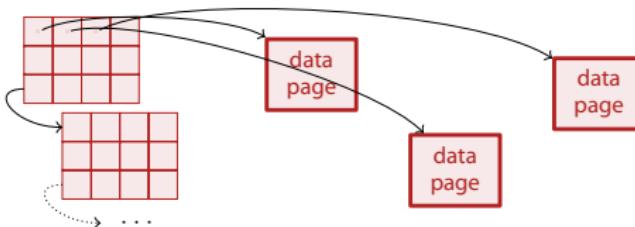
Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Heap Files

### Directory of pages:



- Use as **space map** with information about free space on each page
  - granularity as trade-off space ↔ accuracy  
(may range from *open/closed* bit to exact information)
- + free space search more efficient
- memory overhead to host the page directory



#### Magnetic Disks

Access Time  
Sequential vs. Random Access

#### I/O Parallelism

RAID Levels 1, 0, and 5

#### Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

#### Managing Space

Free Space Management

#### Buffer Manager

Pinning and Unpinning  
Replacement Policies

#### Databases vs. Operating Systems

#### Files and Records

Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

#### Recap

# Free Space Management

Which page to pick for the insertion of a new record?

## Append Only

Always insert into last page. Otherwise, create a new page.

## Best Fit

Reduces fragmentation, but requires searching the entire free list/space map for each insert.

## First Fit

Search from beginning, take first page with sufficient space.  
(⇒ These pages quickly fill up, system may waste a lot of search effort in these first pages later on.)

## Next Fit

Maintain **cursor** and continue searching where search stopped last time.

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files

Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Free Space Witnesses

We can accelerate the search by remembering **witnesses**:

- Classify pages into **buckets**, e.g., “75 %–100 % full”, “50 %–75 % full”, “25 %–50 % full”, and “0 %–25 % full”.
- For each bucket, remember some **witness pages**.
- Do a regular best/first/next fit search only if no witness is recorded for the specific bucket.
- Populate witness information, e.g., as a side effect when searching for a best/first/next fit page.

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management

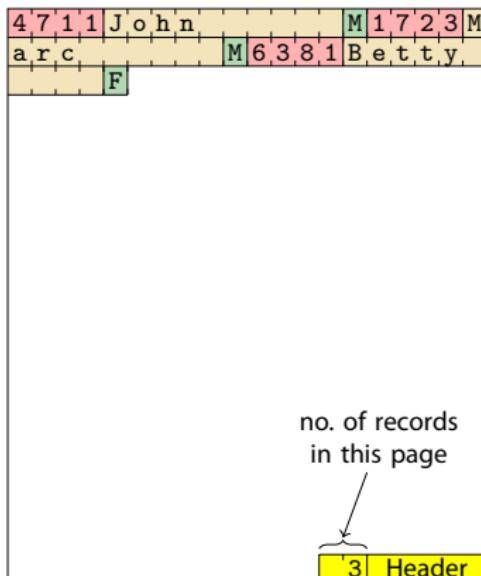
Inside a Page  
Alternative Page Layouts

Recap

## Inside a Page — Fixed-Length Records

Now turn to the **internal page structure**:

ID	NAME	SEX
4711	John	M
1723	Marc	M
6381	Betty	F



- **Record identifier (*rid*):**  
 $\langle \text{pageno}, \text{slotno} \rangle$
- Record position (within page):  
 $\text{slotno} \times \text{bytes per slot}$
- Record **deletion?**
  - record id should **not** change
  - ⇒ **slot directory** (bitmap)

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

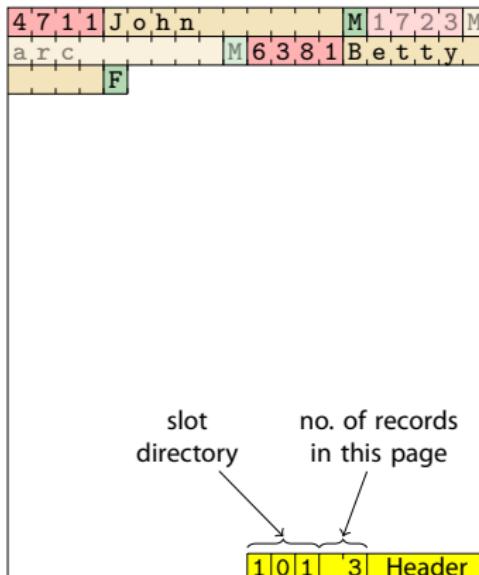
Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Inside a Page — Fixed-Length Records

Now turn to the **internal page structure**:

ID	NAME	SEX
4711	John	M
1723	Marc	M
6381	Betty	F



- **Record identifier (*rid*):**  
 $\langle \text{pageno}, \text{slotno} \rangle$
- Record position (within page):  
 $\text{slotno} \times \text{bytes per slot}$
- Record **deletion?**
  - record id should **not** change
  - ⇒ **slot directory** (bitmap)

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

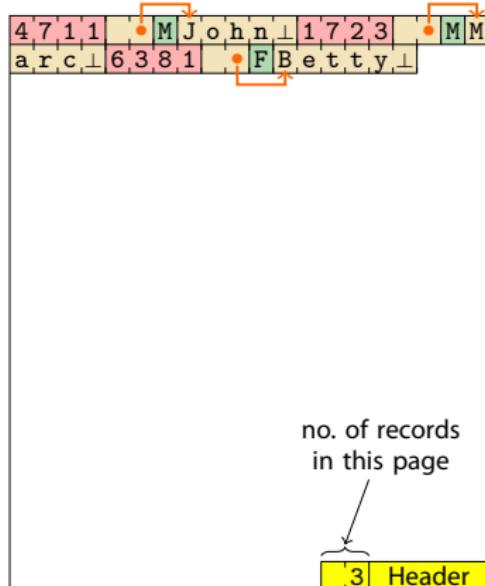
Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Inside a Page—Variable-Sized Fields

- Variable-sized fields moved to **end** of each record.
  - Placeholder points to location.
  - **Why?**



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

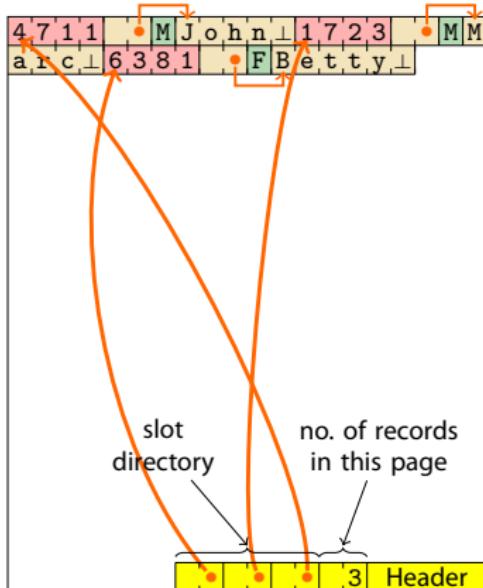
Databases vs. Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Inside a Page—Variable-Sized Fields

- Variable-sized fields moved to **end** of each record.
  - Placeholder points to location.
  -  **Why?**
- Slot directory points to start of each record.



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

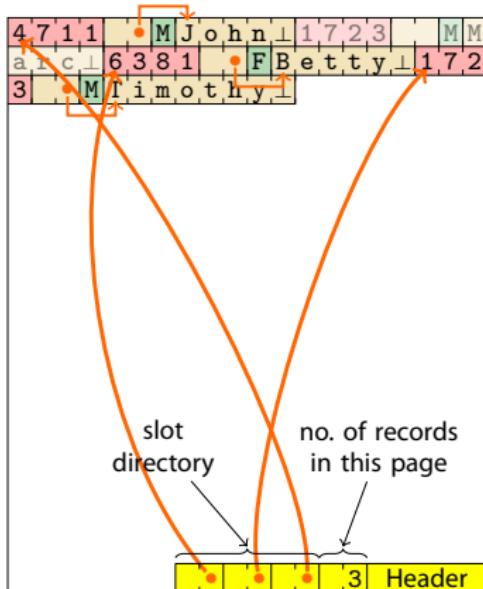
Databases vs. Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Inside a Page—Variable-Sized Fields

- Variable-sized fields moved to **end** of each record.
  - Placeholder points to location.
  - **Why?**
- Slot directory points to start of each record.
- Records **can move** on page.
  - E.g., if field size changes or page is compacted.



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

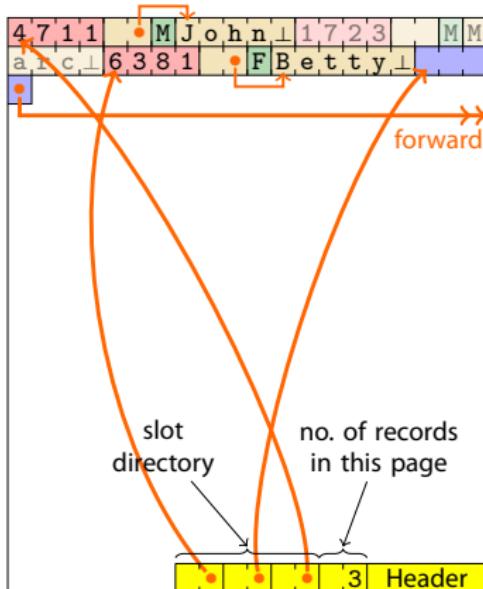
Databases vs.  
Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Inside a Page—Variable-Sized Fields

- Variable-sized fields moved to **end** of each record.
  - Placeholder points to location.
  - **Why?**
- Slot directory points to start of each record.
- Records **can move** on page.
  - E.g., if field size changes or page is compacted.
- Create “**forward address**” if record won’t fit on page.
  - **Future updates?**



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

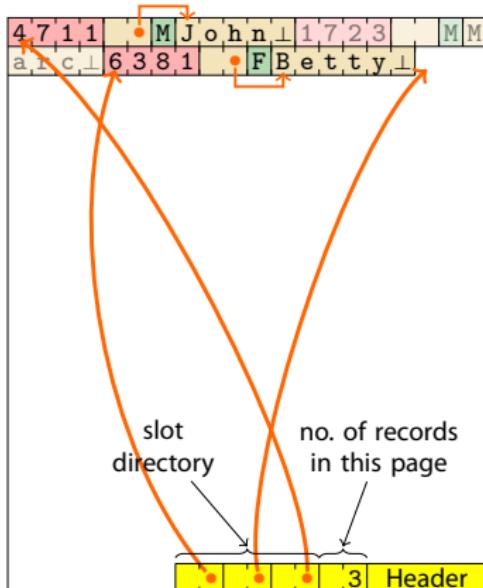
Databases vs. Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Inside a Page—Variable-Sized Fields

- Variable-sized fields moved to **end** of each record.
  - Placeholder points to location.
  - **Why?**
- Slot directory points to start of each record.
- Records **can move** on page.
  - E.g., if field size changes or page is compacted.
- Create “**forward address**” if record won’t fit on page.
  - **Future updates?**
- Related issue: space-efficient representation of NULL values.



Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs. Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

# IBM DB2 Data Pages

## DB2. Data page and layout details

- Support for **4 K, 8 K, 16 K, 32 K data pages** in separate table spaces. Buffer manager pages match in size.
- 68 bytes of database manager overhead per page. On a 4 K page: maximum of 4,028 bytes of user data (maximum record size: 4,005 bytes). Records do *not* span pages.
- **Maximum table size:** 512 GB (with 32 K pages). Maximum number of columns: 1,012 (4 K page: 500), maximum number of rows/page: 255.  **IBM DB2 RID format?**
- Columns of type LONG VARCHAR, CLOB, etc. maintained outside regular data pages (pages contain descriptors only).
- **Free space management:** first-fit order. Free space map distributed on every 500th page in FSCR (free space control records). Records updated in-place if possible, otherwise uses forward records.

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

Databases vs.  
Operating Systems

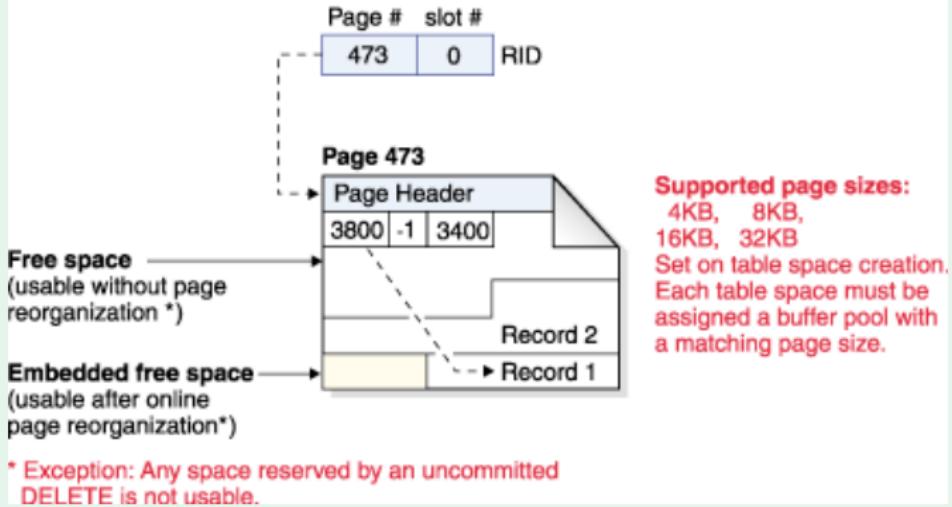
Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

# IBM DB2 Data Pages

DB2. Taken directly from the DB2 V9.5 Information Center

## Data page and RID format



<http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/>

Storage  
Torsten Grust



Magnetic Disks  
Access Time  
Sequential vs. Random Access

I/O Parallelism  
RAID Levels 1, 0, and 5

Alternative Storage Techniques  
Solid-State Disks  
Network-Based Storage

Managing Space  
Free Space Management

Buffer Manager  
Pinning and Unpinning  
Replacement Policies

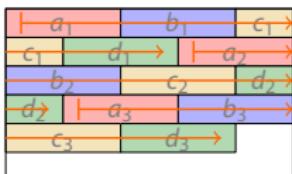
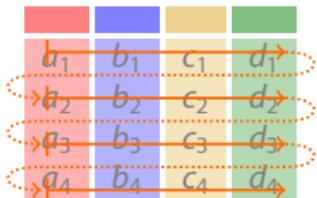
Databases vs. Operating Systems

Files and Records  
Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

Recap

## Alternative Page Layouts

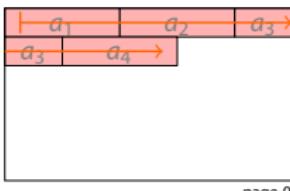
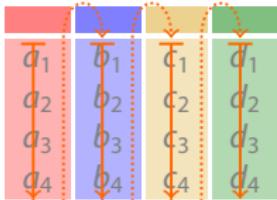
We have just populated data pages in a **row-wise** fashion:



page 0

page 1

We could as well do that **column-wise**:



page 0

page 1



Storage

Torsten Grust



Magnetic Disks

Access Time

Sequential vs. Random Access

I/O Parallelism

RAID Levels 1, 0, and 5

Alternative Storage Techniques

Solid-State Disks

Network-Based Storage

Managing Space

Free Space Management

Buffer Manager

Pinning and Unpinning

Replacement Policies

Databases vs. Operating Systems

Files and Records

Heap Files

Free Space Management

Inside a Page

Alternative Page Layouts

Recap

## Alternative Page Layouts

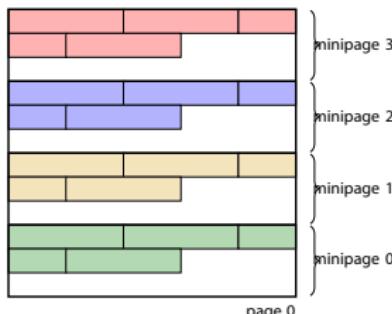
These two approaches are also known as **NSM (n-ary storage model)** and **DSM (decomposition storage model)**.<sup>1</sup>

- Tuning knob for certain workload types (e.g., OLAP)
- Suitable for narrow projections and in-memory database systems  
(↗ Database Systems and Modern CPU Architecture)
- Different behavior with respect to **compression**.

A hybrid approach is the **PAX (Partition Attributes Across)** layout:

- Divide each page into **minipages**.
- Group attributes into them.

↗ Ailamaki et al. Weaving Relations for Cache Performance. VLDB 2001.



<sup>1</sup> Recently, the terms **row-store** and **column-store** have become popular, too.



## Recap

### Magnetic Disks

Random access **orders of magnitude** slower than sequential.

### Disk Space Manager

Abstracts from hardware details and maps page number  $\mapsto$  physical location.

### Buffer Manager

Page **caching** in main memory; pin ()/unpin () interface; **replacement policy** crucial for effectiveness.

### File Organization

Stable **record identifiers (rids)**; maintenance with fixed-sized records and variable-sized fields; NSM vs. DSM.



Storage

Torsten Grust

#### Magnetic Disks

Access Time  
Sequential vs. Random Access

#### I/O Parallelism

RAID Levels 1, 0, and 5

#### Alternative Storage Techniques

Solid-State Disks  
Network-Based Storage

#### Managing Space

Free Space Management

#### Buffer Manager

Pinning and Unpinning  
Replacement Policies

#### Databases vs. Operating Systems

#### Files and Records

Heap Files  
Free Space Management  
Inside a Page  
Alternative Page Layouts

#### Recap

# Chapter 3

## Indexing

Navigate and Search Large Data Volumes

*Architecture and Implementation of Database Systems*  
Summer 2014

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

Torsten Grust  
Wilhelm-Schickard-Institut für Informatik  
Universität Tübingen



## File Organization and Indexes

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model  
Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

Indexes

Index Entries  
Clustered / Unclustered  
Dense / Sparse

- A **heap file** provides just enough structure to maintain a collection of records (of a table).
- The heap file supports **sequential scans** (`openScan(·)`) over the collection, e.g.

### SQL query leading to a sequential scan

```
1 SELECT A,B  
2 FROM   R
```

**But:** No further operations receive specific support from the heap file.

## File Organization and Indexes

- For queries of the following type, 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:

### Queries calling for systematic file organization

```
1 SELECT A,B  
2 FROM   R  
3 WHERE   C > 42
```

```
1 SELECT A,B  
2 FROM   R  
3 ORDER BY C ASC
```

### 📎 File organization for table R

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



Indexing

Torsten Grust



### File Organization

#### File Organization Competition

Cost Model  
Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

#### Indexes

Index Entries  
Clustered / Unclustered  
Dense / Sparse

## File Organization and Indexes

- For queries of the following type, 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:

### Queries calling for systematic file organization

```
1 SELECT A,B  
2 FROM   R  
3 WHERE   C > 42
```

```
1 SELECT A,B  
2 FROM   R  
3 ORDER BY C ASC
```

### 📎 File organization for table R

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 C attribute values**, place records on neighboring pages. (Only include columns A, B, C in records.)



Indexing

Torsten Grust

### File Organization

#### File Organization Competition

Cost Model  
Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

#### Indexes

Index Entries  
Clustered / Unclustered  
Dense / Sparse

## A Competition of File Organizations

- This chapter presents a comparison of three **file organizations**:
  - ① Files of **randomly ordered** records (heap files)
  - ② Files **sorted** on some record field(s)
  - ③ Files **hashed** on some record field(s)
- A file organization is tuned to make a certain query (class) efficient, but if we have to support **more than one query class**, we may be in trouble. Consider:

### Query Q calling for organization sorted on column A

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

- If the file for table R is sorted on C, this does not buy us anything for query Q.
- If Q is an important query but is *not* supported by R's file organization, we can build a support data structure, an **index**, to speed up (queries similar to) Q.



Indexing  
Torsten Grust

### File Organization

#### File Organization Competition

Cost Model  
Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

#### Indexes

Index Entries  
Clustered / Unclustered  
Dense / Sparse

## A Competition of File Organizations

- This competition assesses 3 file organizations in 5 disciplines:
  - ① **Scan**: read all records in a given file.
  - ② **Search with equality test**

### Query calling for equality test support

```
1 SELECT *
2 FROM   R
3 WHERE   C = 42
```

- ③ **Search with range selection** (upper or lower bound might be unspecified)

### Query calling for range selection support

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

- 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.



Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Simple Cost Model

- Performing these 5 database operations clearly involves **block I/O**, the major cost factor.
- However, we have to additionally pay for CPU time used to **search inside a page, compare a record field to a selection constant**, etc.
- To analyze cost more accurately, we introduce the following parameters:

### Simple cost model parameters

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

Indexing  
Torsten Grust



File Organization  
File Organization Competition

#### Cost Model

Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

#### Indexes

Index Entries  
Clustered / Unclustered  
Dense / Sparse

# Simple Cost Model

## Simple cost model parameters

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

## Remarks:

- $D \approx 15 \text{ ms}$
- $C \approx H \approx 0.1 \mu\text{s}$
- This is a coarse model to **estimate** the actual execution time (this does *not* model network access, cache effects, burst I/O, ...).

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

Indexes

Index Entries  
Clustered / Unclustered  
Dense / Sparse

## Aside: Hashing

### 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:

$$\begin{array}{lll} h(\langle 42, \text{true}, \text{'foo'} \rangle) & \rightarrow & 2 \\ h(\langle 14, \text{true}, \text{'bar'} \rangle) & \rightarrow & 6 \\ h(\langle 26, \text{false}, \text{'baz'} \rangle) & \rightarrow & 2 \end{array} \quad (42 = 101010_2) \quad (14 = 1110_2) \quad (26 = 11010_2)$$

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Aside: Hashing

### 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:

$$\begin{array}{lll} h(\langle 42, \text{true}, \text{'foo'} \rangle) & \rightarrow & 2 \\ h(\langle 14, \text{true}, \text{'bar'} \rangle) & \rightarrow & 6 \\ h(\langle 26, \text{false}, \text{'baz'} \rangle) & \rightarrow & 2 \end{array} \quad (42 = 101010_2) \quad (14 = 1110_2) \quad (26 = 11010_2)$$

- 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  $h(\dots) = p$ .
- 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 as a hashed file.

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Scan Cost

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

### 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**:

$$\text{Scan}_{\text{heap}} = b \cdot (D + r \cdot C)$$

### 2 Sorted file

The sort order does not help much here. However, the scan retrieves the records in sorted order (which can be big plus later on):

$$\text{Scan}_{\text{sort}} = b \cdot (D + r \cdot C)$$

### 3 Hashed file

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

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

## Cost for Search With Equality Test ( $A = const$ )

### 1 Heap file

Consider (a) the equality test is on a *primary key*, (b) the equality test is *not* on a *primary key*:

- (a)  $\text{Search}_{\text{heap}} = 1/2 \cdot b \cdot (D + r \cdot (C + H))$
- (b)  $\text{Search}_{\text{heap}} = b \cdot (D + r \cdot (C + H))$

### 2 Sorted file (sorted on A)

We assume the equality test to be on the field determining the sort order. The sort order enables us to use **binary search**:

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Cost for Search With Equality Test ( $A = \text{const}$ )

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

### 1 Heap file

Consider (a) the equality test is on a *primary key*, (b) the equality test is *not* on a *primary key*:

- (a)  $\text{Search}_{\text{heap}} = 1/2 \cdot b \cdot (D + r \cdot (C + H))$
- (b)  $\text{Search}_{\text{heap}} = b \cdot (D + r \cdot (C + H))$

### 2 Sorted file (sorted on A)

We assume the equality test to be on the field determining the sort order. The sort order enables us to use **binary search**:

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

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.)



## Cost for Search With Equality Test ( $A = \text{const}$ )

### ③ Hashed file (hashed on A)

**Hashed files support equality searching best.** The hash function directly leads us to the page containing the hit (overflow chains ignored here).

Consider (a) the equality test is on a *primary key*, (b) the equality test is *not* on a *primary key*:

- (a)  $\text{Search}_{\text{hash}} = H + D + \frac{1}{2} \cdot r \cdot C$
- (b)  $\text{Search}_{\text{hash}} = H + D + r \cdot C$

No dependence on file size  $b$  here. (All qualifying records live on the same page or, if present, in its overflow chain.)

Indexing  
Torsten Grust



File Organization

File Organization  
Competition

Cost Model  
Scan

Equality Test

Range Selection  
Insertion  
Deletion

Indexes

Index Entries  
Clustered / Unclustered  
Dense / Sparse

## Cost for Range Selection ( $A \geq lower$ AND $A \leq upper$ )

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

### ① Heap file

Qualifying records can appear anywhere in the file:

$$\text{Range}_{\text{heap}} = b \cdot (D + r \cdot (C + 2 \cdot H))$$

### ② Sorted file (sorted on A)

**Use equality search** (with  $A = lower$ ), **then sequentially scan** the file until a record with  $A > upper$  is encountered:

$$\text{Range}_{\text{sort}} = \log_2 b \cdot (D + \log_2 r \cdot (C + H)) + \lceil n/r \rceil \cdot D + n \cdot (C + H)$$

( $n + 1$  overall hits in the range,  $n \geq 0$ )

## Cost for Range Selection ( $A \geq lower$ AND $A \leq upper$ )

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

### ③ Hashed file (hashed on A)

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

$$\text{Range}_{\text{hash}} = (100/80) \cdot b \cdot (D + r \cdot (C + H))$$

# Insertion Cost

## ① Heap file

We can add the record to some **arbitrary** free page (finding that page is not accounted for here).

This involves reading and writing that page:

$$\text{Insert}_{\text{heap}} = 2 \cdot D + C$$

## ② Sorted file

On average, the new record will belong in the middle of the file. After insertion, we have to **shift all subsequent records** (in the second half of the file):

$$\text{Insert}_{\text{sort}} = \log_2 b \cdot (D + \log_2 r \cdot (C + H)) + \frac{1}{2} \cdot b \cdot (\underbrace{D + r \cdot C + D}_{\text{read + shift + write}})$$

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Insertion Cost

Indexing

Torsten Grust



File Organization

File Organization

Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

### ③ Hashed file

We pretend to search for the record, then read and write the page determined by the hash function (here we assume the spare 20 % space on the page is sufficient to hold the new record):

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

## Deletion Cost (Record Specified by its *rid*)

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

### ① Heap file

If we do not try to compact the file after we have found and removed the record (because the file uses free space management), the cost is:

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

### ② Sorted file

Again, we access the record's page and then (on average) shift the second half the file to compact the file:

$$\text{Delete}_{\text{sort}} = D + 1/2 \cdot b \cdot (D + r \cdot C + D)$$

### ③ Hashed file

Accessing the page using the *rid* is even faster than the hash function, so the hashed file behaves like the heap file:

$$\text{Delete}_{\text{hash}} = D + C + D$$

## The “Best” File Organization?

- There is **no single file organization** that responds equally fast to all 5 operations.
- This is a dilemma, because more advanced file organizations can really make a difference in speed.

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

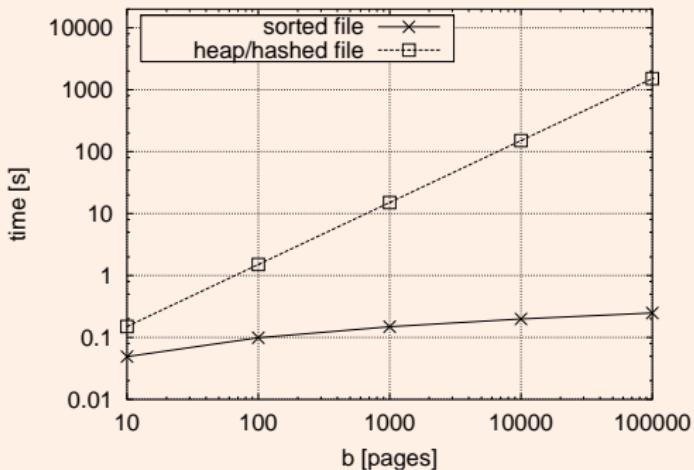
Clustered / Unclustered

Dense / Sparse

## Range selection performance

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

### Range Selection Performance



Indexing  
Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

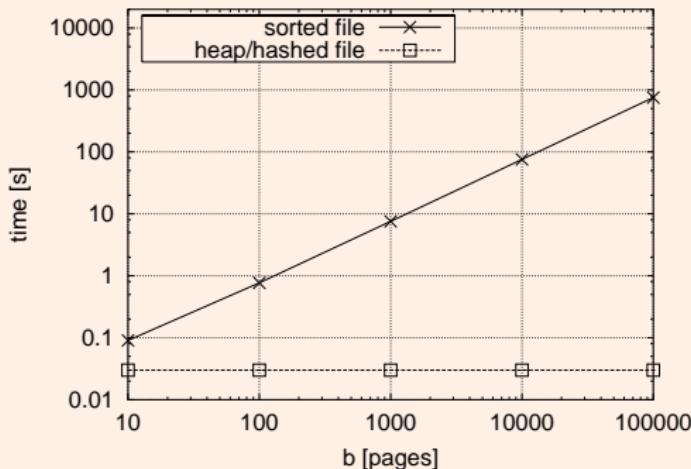
Clustered / Unclustered

Dense / Sparse

## Deletion Performance Comparison

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

### Deletion Performance



Indexing  
Torsten Grust



### File Organization

#### File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

### Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

# Indexes

- There exist **index structures** which offer all the advantages of a sorted file *and* support insertions/deletions efficiently<sup>1</sup>: **B<sup>+</sup>-trees**.
- Before we turn to B<sup>+</sup>-trees in detail, the following sheds some light on indexes in general.



Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model  
Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

Indexes

Index Entries  
Clustered / Unclustered  
Dense / Sparse

---

<sup>1</sup>At the cost of a modest space overhead.

## Indexes

- If the basic organization of a file does not support a particular operation, we can **additionally maintain** an auxiliary structure, an **index**, which adds the needed support.
- We will use indexes like **guides**. Each guide is specialized to accelerate searches on a specific attribute A (or a combination of attributes) of the records in its associated file:

### Index usage

- ① Query the 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 sufficient information to access the actual record in the file).

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Indexes

- If the basic organization of a file does not support a particular operation, we can **additionally maintain** an auxiliary structure, an **index**, which adds the needed support.
- We will use indexes like **guides**. Each guide is specialized to accelerate searches on a specific attribute A (or a combination of attributes) of the records in its associated file:

### Index usage

- ① Query the 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 sufficient 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  $k$ .



Indexing  
Torsten Grust



File Organization  
File Organization Competition

Cost Model  
Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

Indexes  
Index Entries  
Clustered / Unclustered  
Dense / Sparse

## Index entries

- We can design the **index entries**, i.e., the  $k_*$ , in various ways:

### Index entry designs

Variant    Index entry  $k_*$

A         $\langle k, \langle \dots, A = k, \dots \rangle \rangle$

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Index entries

- We can design the **index entries**, i.e., the  $k_*$ , in various ways:

### Index entry designs

Variant	Index entry $k_*$
A	$\langle k, \langle \dots, A = k, \dots \rangle \rangle$
B	$\langle k, rid \rangle$

**Indexing**  
Torsten Grust



### File Organization

#### File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

### Indexes

#### Index Entries

Clustered / Unclustered

Dense / Sparse

## Index entries

- We can design the **index entries**, i.e., the  $k_*$ , in various ways:

### Index entry designs

Variant	Index entry $k_*$
A	$\langle k, \langle \dots, A = k, \dots \rangle \rangle$
B	$\langle k, rid \rangle$
C	$\langle k, [rid_1, rid_2, \dots] \rangle$

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Index entries

- We can design the **index entries**, i.e., the  $k_*$ , in various ways:

### Index entry designs

Variant	Index entry $k_*$
A	$\langle k, \langle \dots, A = k, \dots \rangle \rangle$
B	$\langle k, rid \rangle$
C	$\langle k, [rid_1, rid_2, \dots] \rangle$

### Remarks:

- With variant A, there is *no* need to store the data records in addition to the index—the index itself is a special file organization.

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Index entries

- We can design the **index entries**, i.e., the  $k_*$ , in various ways:

### Index entry designs

Variant	Index entry $k_*$
A	$\langle k, \langle \dots, A = k, \dots \rangle \rangle$
B	$\langle k, rid \rangle$
C	$\langle k, [rid_1, rid_2, \dots] \rangle$

### Remarks:

- With variant A, 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 A

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Index entries

- We can design the **index entries**, i.e., the  $k_*$ , in various ways:

### Index entry designs

Variant	Index entry $k_*$
A	$\langle k, \langle \dots, A = k, \dots \rangle \rangle$
B	$\langle k, rid \rangle$
C	$\langle k, [rid_1, rid_2, \dots] \rangle$

### Remarks:

- With variant A, 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 A to avoid redundant storage of records.

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Index entries

- We can design the **index entries**, i.e., the  $k_*$ , in various ways:

### Index entry designs

Variant	Index entry $k_*$
A	$\langle k, \langle \dots, A = k, \dots \rangle \rangle$
B	$\langle k, rid \rangle$
C	$\langle k, [rid_1, rid_2, \dots] \rangle$

### Remarks:

- With variant A, 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 A to avoid redundant storage of records.
- Variants B and C use  $rid$ (s) to point into the actual data file.
- Variant C leads to less index entries if multiple records match a search key  $k$ , but index entries are of variable length.

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Indexing Example

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

### Example (Indexing example, see following slide)

- The data file contains  $\langle \text{name}, \text{age}, \text{sal} \rangle$  records of table employees, the file itself (index entry variant A) is hashed on field age (hash function  $h_1$ ).
- The index file contains  $\langle \text{sal}, \text{rid} \rangle$  index entries (variant B), pointing into the data file.
- This file organization + index efficiently supports equality searches on the age **and** sal keys.

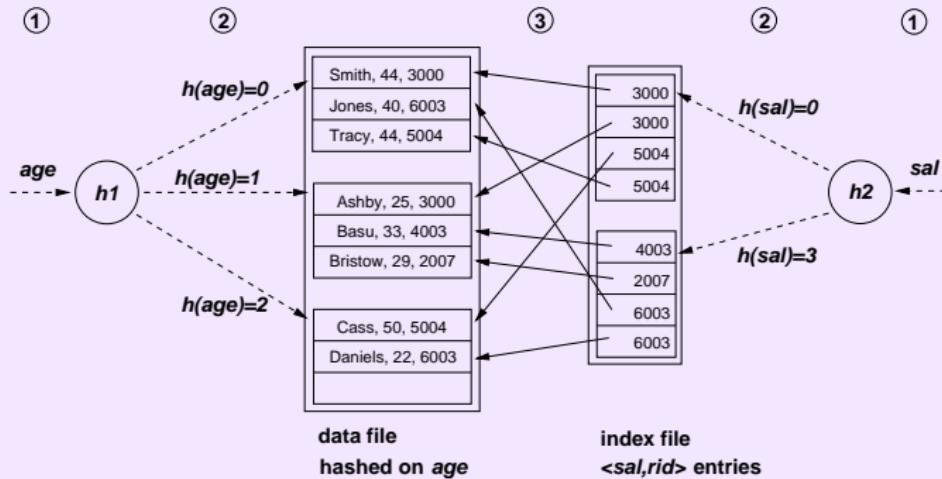
# Indexing Example

Indexing

Torsten Grust



## Example (Indexing example)



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Index Properties: Clustered vs. Unclustered

- Suppose, we have to support **range selections** on records such that  $A \geq lower$  AND  $A \leq upper$ .
- If we maintain an index on the A-field, we can
  - ➊ **query the index** once for a record with  $A = lower$ , and then
  - ➋ **sequentially scan the data file** from there until we encounter a record with field  $A > upper$ .
- **However:** This switch from index to data file will only work provided that the **data file itself is sorted on the field A**.

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

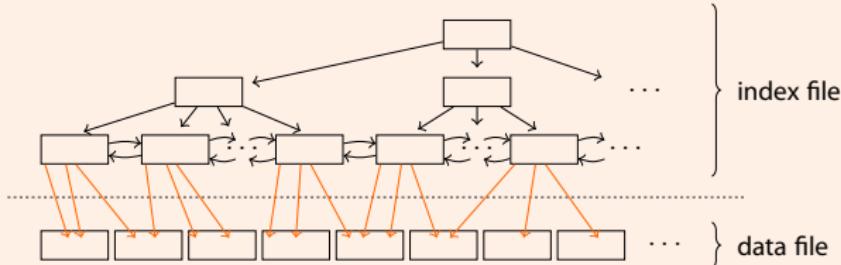
Index Entries

Clustered / Unclustered

Dense / Sparse

## Index Properties: Clustered vs. Unclustered

### Index over a data file with matching sort order



#### Remark:

- In a B<sup>+</sup>-tree, for example, the index entries  $k_*$  stored in the leaves are sorted by key  $k$ .



Indexing

Torsten Grust

File Organization

File Organization  
Competition

Cost Model  
Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

Indexes

Index Entries  
Clustered / Unclustered  
Dense / Sparse

## Index Properties: Clustered vs. Unclustered

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

### 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**. In this case, proximity of index entries does *not* imply proximity in the data file:

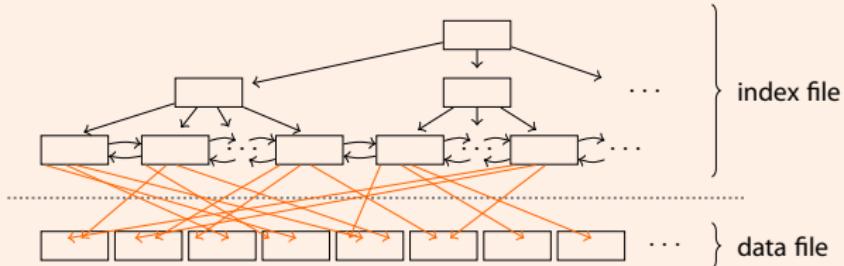


- As before, we can query the index for a record with  $A = lower$ .

To continue the scan, however, we have to **revisit the index entries** which point us to **data pages scattered** all over the data file.

## Index Properties: Clustered vs. Unclustered

### Unclustered index



### Remarks:

- If an index uses entries  $k*$  of variant A, the index is obviously clustered by definition.

### Variant A in Oracle 8i

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

- A data file can have at most one clustered index (but any number of unclustered indexes).

Indexing  
Torsten Grust



File Organization

File Organization Competition

Cost Model  
Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

Indexes

Index Entries  
Clustered / Unclustered  
Dense / Sparse

## Index Properties: Clustered vs. Unclustered

### DB2. Clustered indexes

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

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

The DB2 V9.5 manual says:

*"[ 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)."*



Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model  
Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

Indexes

Index Entries  
Clustered / Unclustered  
Dense / Sparse

## Index Properties: Clustered vs. Unclustered



### Cluster a table based on an existing index

Reorganize 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.
- The evaluation of 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 =  $f$ ),  $f \in 10 \dots 100$ , to reserve page space for subsequent insertions.

Indexing

Torsten Grust



File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Index Properties: Clustered vs. Unclustered

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model  
Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

Indexes

Index Entries  
Clustered / Unclustered  
Dense / Sparse

### DB2. Inspect clustered index details

```
1 db2 => reorgchk update statistics on table grust.accel
2
3 Doing RUNSTATS ...
4
5 Table statistics:
6 .
7 Index statistics:
8
9 F4: CLUSTERRATIO or normalized CLUSTERFACTOR > 80
10
11 SCHEMA.NAME          INDCARD  LEAF  LVLS    KEYS   F4  REORG
12 -----
13 Table: GRUST.ACCEL
14 Index: GRUST.IKIND      235052   529     3       3   99  -----
15 Index: GRUST.IPAR       235052   675     3   67988   99  -----
16 Index: GRUST.IPOST      235052   980     3  235052  100  -----
17 Index: GRUST.ITAG       235052   535     3       75   80  *-----
18 Index: SYSIBM.SQL080119120245000
19                               235052   980     3  235052  100  -----
20 -----
```

## Index Properties: Dense vs. Sparse

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

A **clustered index** comes with more advantages than the improved performance for range selections. We can additionally design the index 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). Key  $k$  is the **smallest key** on that page.

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

## Index Properties: Dense vs. Sparse

Indexing

Torsten Grust



File Organization

File Organization  
Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

To search a record with field  $A = k$  in a sparse A-index,

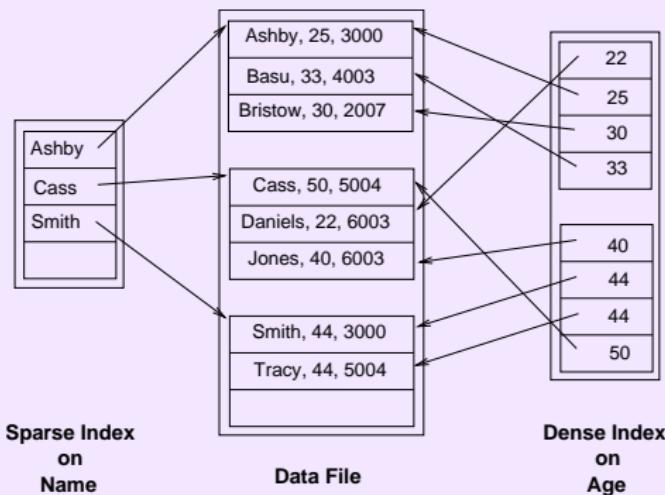
- ① Locate the largest index entry  $k'^*$  such that  $k' \leq k$ , then
- ② access the page pointed to by  $k'^*$ , and
- ③ scan this page (and the following pages, if needed) to find records with  $\langle \dots, A = k, \dots \rangle$ .

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

## Index example (Dense vs. Sparse)

### Example (Sparse index example)

- Again, the data file contains  $\langle \text{name}, \text{age}, \text{sal} \rangle$  records. We maintain a **clustered sparse index** on field name and an **unclustered dense index** on field age. Both use index entry variant B to point into the data file.



Indexing

Torsten Grust

File Organization

File Organization Competition

Cost Model

Scan

Equality Test

Range Selection

Insertion

Deletion

Indexes

Index Entries

Clustered / Unclustered

Dense / Sparse

## Index Properties: Dense vs. Sparse

### Note:

- Sparse indexes need 2–3 orders of magnitude less space than dense indexes (consider # records/page).
- We *cannot* 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?

```
1 SELECT MAX(age)
2 FROM   employees
```

```
1 SELECT MAX(name)
2 FROM   employees
```

Indexing  
Torsten Grust



File Organization  
File Organization Competition

Cost Model  
Scan  
Equality Test  
Range Selection  
Insertion  
Deletion

Indexes  
Index Entries  
Clustered / Unclustered  
Dense / Sparse



### Binary Search

#### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

#### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

# Chapter 4

## Tree-Structured Indexing

### ISAM and B<sup>+</sup>-trees

*Architecture and Implementation of Database Systems*  
Summer 2014



## Ordered Files and Binary Search

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

```
1 SELECT *
2 FROM   CUSTOMERS
3 WHERE  ZIPCODE BETWEEN 8880 AND 8999
```

We could

- ① **sort** the table on disk (in ZIPCODE-order)
- ② To answer queries, use **binary search** to find the first qualifying tuple, then **scan** as long as ZIPCODE < 8999.

Tree-Structured  
Indexing

Torsten Grust



### Binary Search

#### ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

#### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## Ordered Files and Binary Search

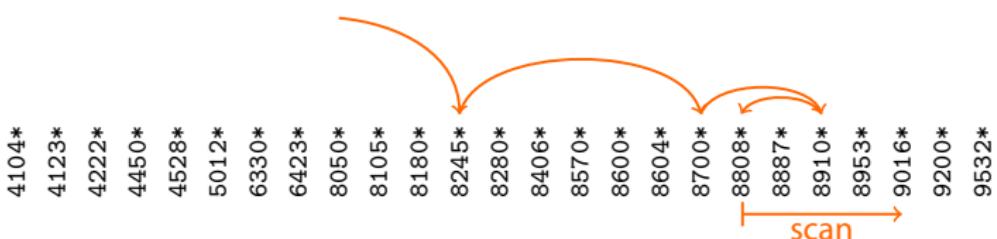
How could we prepare for such queries and evaluate them efficiently?

```
1 SELECT *
2 FROM   CUSTOMERS
3 WHERE  ZIPCODE BETWEEN 8880 AND 8999
```

We could

- ① **sort** the table on disk (in ZIPCODE-order)
- ② To answer queries, use **binary search** to find the first qualifying tuple, then **scan** as long as  $\text{ZIPCODE} < 8999$ .

Here, let  $k*$  denote the full record with key  $k$ :



Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

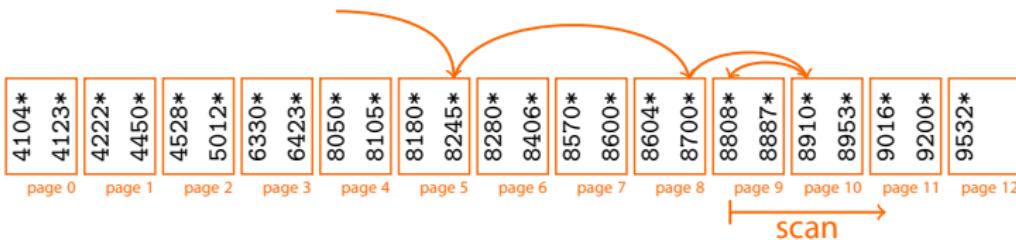
Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## Ordered Files and Binary Search



✓ We get **sequential access** during the **scan phase**.

We need to read  $\log_2(\# \text{ tuples})$  tuples during the **search phase**.

Tree-Structured  
Indexing

Torsten Grust



### Binary Search

#### ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

#### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

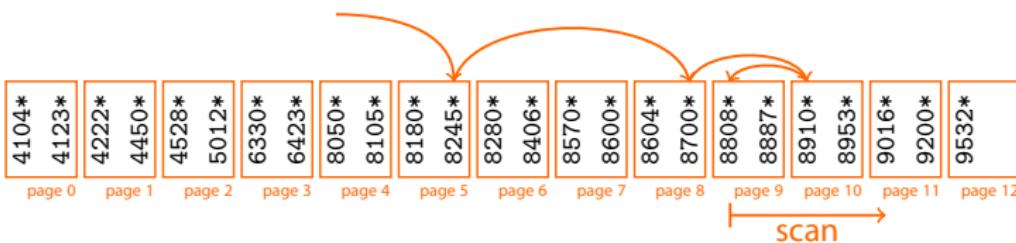
Bulk Loading

Partitioned B<sup>+</sup>-trees

## Ordered Files and Binary Search

Tree-Structured  
Indexing

Torsten Grust



✓ We get **sequential access** during the **scan phase**.

We need to read  $\log_2(\# \text{ tuples})$  tuples during the **search phase**.

✗ We need to read about **as many pages** for this.

The whole point of binary search is that we make **far, unpredictable jumps**. This largely defeats page prefetching.



ISAM  
Multi-Level ISAM  
Too Static?  
Search Efficiency

B<sup>+</sup>-trees  
Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

## Tree-Structured Indexing

Tree-Structured  
Indexing

Torsten Grust



- This chapter discusses two **index structures** which especially shine if we need to support **range selections** (and thus sorted file scans): **ISAM** files and **B<sup>+</sup>-trees**.
- Both indexes are based on the same simple idea which naturally leads to a **tree-structured** organization of the indexes. (Hash indexes are covered in a subsequent chapter.)
- B<sup>+</sup>-trees refine the idea underlying the rather static ISAM scheme and add efficient support for **insertions** and **deletions**.

### Binary Search

#### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

#### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

## Indexed Sequential Access Method (ISAM)

Tree-Structured  
Indexing

Torsten Grust



### Binary Search

#### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

#### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

**Remember:** range selections on ordered files may use **binary search** to locate the lower range limit as a starting point for a sequential scan of the file (until the upper limit is reached).

### ISAM ...

- ...acts as a replacement for the binary search phase, and
- touches considerably fewer pages than binary search.

## Indexed Sequential Access Method (ISAM)

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

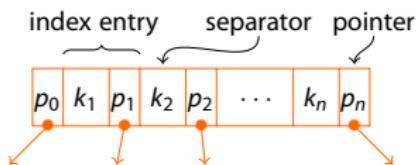
Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

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. In an index entry

$$\langle k_i, \text{pointer to } p_i \rangle ,$$

key  $k_i$  is the first (*i.e.*, the minimal) A value on the data file page pointed to by  $p_i$  ( $p_i$ : page no).

## Indexed Sequential Access Method (ISAM)

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

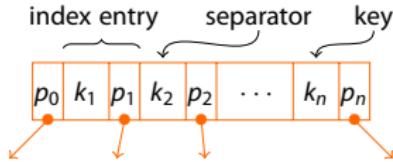
Delete

Duplicates

Key Compression

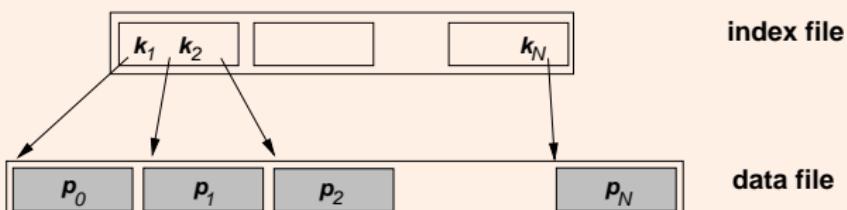
Bulk Loading

Partitioned B<sup>+</sup>-trees



- In the index file, the  $k_i$  serve as **separators** between the contents of pages  $p_{i-1}$  and  $p_i$ .
- 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 of $N + 1$ pages



# Searching ISAM

Tree-Structured  
Indexing

Torsten Grust



## SQL range selection on field A

```
1 SELECT *  
2 FROM   R  
3 WHERE  A BETWEEN lower AND upper
```

To support **range selections**:

- 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 (scan until field A exceeds *upper*).

Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## Indexed Sequential Access Method ISAM

- The size of the index file is likely to be **much smaller** than the data file size. Searching the index is far more efficient than searching the data file.
- For large data files, however, even the index file might be too large to allow for fast searches.

### Main idea behind ISAM indexing

**Recursively** apply the index creation step: treat the topmost index level like the data file and add an additional index layer on top.

Repeat, until the the top-most index layer fits on a single page (the **root page**).

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# Multi-Level ISAM Structure

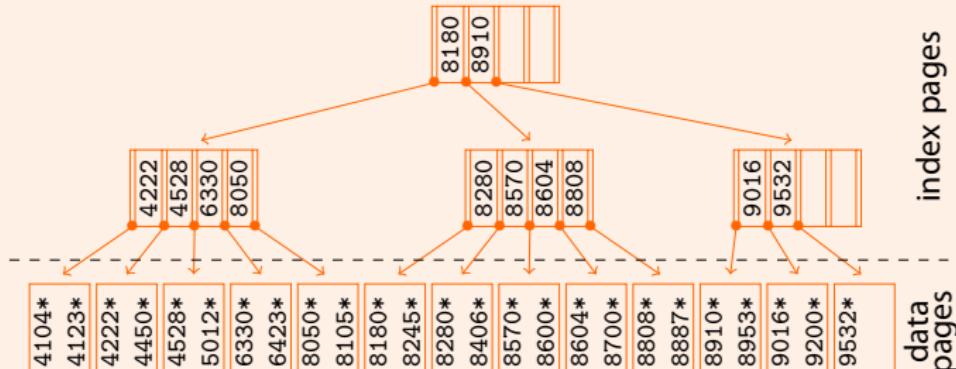
Tree-Structured  
Indexing

Torsten Grust



This recursive index creation scheme leads to a **tree-structured** hierarchy of index levels:

## Multi-level ISAM structure



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

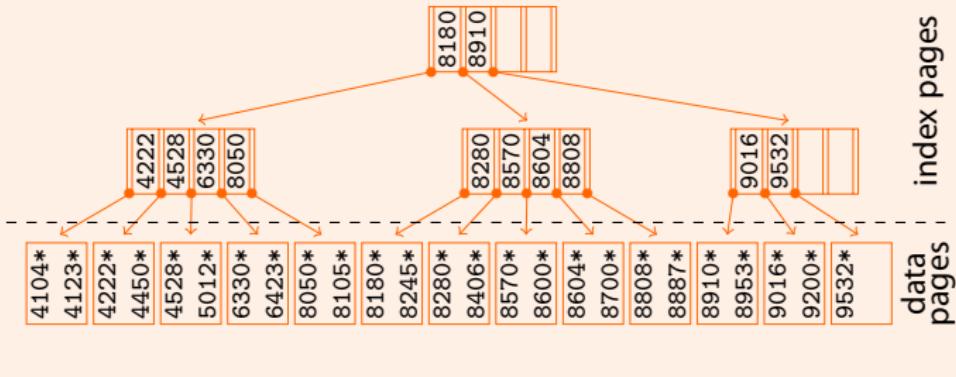
Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# Multi-Level ISAM Structure

## Multi-level ISAM structure



- Each ISAM tree node corresponds to one page (disk block).
- To create the ISAM structure for a given data file, proceed **bottom-up**:
  - ① Sort the data file on the search key field.
  - ② Create the index leaf level.
  - ③ If the top-most index level contains more than one page, repeat.

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## Multi-Level ISAM Structure: Overflow Pages

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

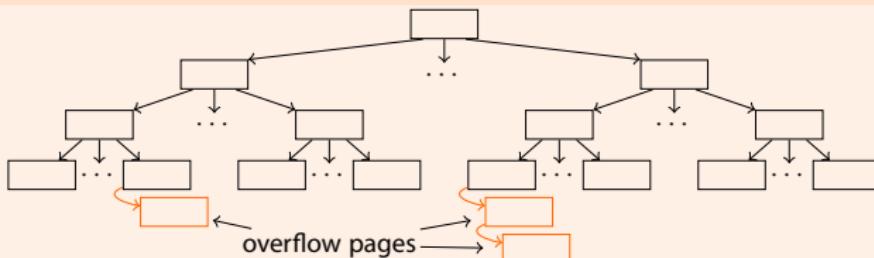
Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

- The upper index levels of the ISAM tree remain **static**: insertions and deletions in the data file do *not* affect the upper tree layers.
- Insertion of record into data file: if space is left on the associated leaf page, insert record there.
- Otherwise create and maintain a chain of **overflow pages** hanging off the full primary leaf page. **Note:** the records on the overflow pages are **not ordered** in general.  
⇒ Over time, **search performance in ISAM can degrade**.

### Multi-level ISAM structure with overflow pages



## Multi-Level ISAM Structure: Example

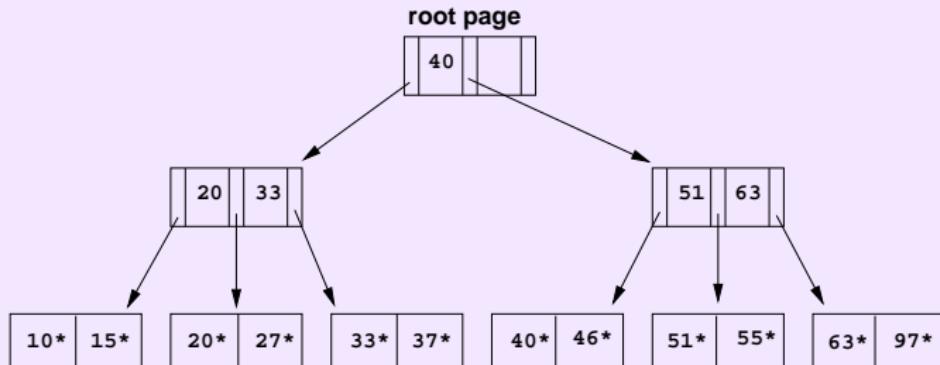
Tree-Structured  
Indexing

Torsten Grust



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

### Example (Initial state of ISAM structure)



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# Multi-Level ISAM Structure: Insertions

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

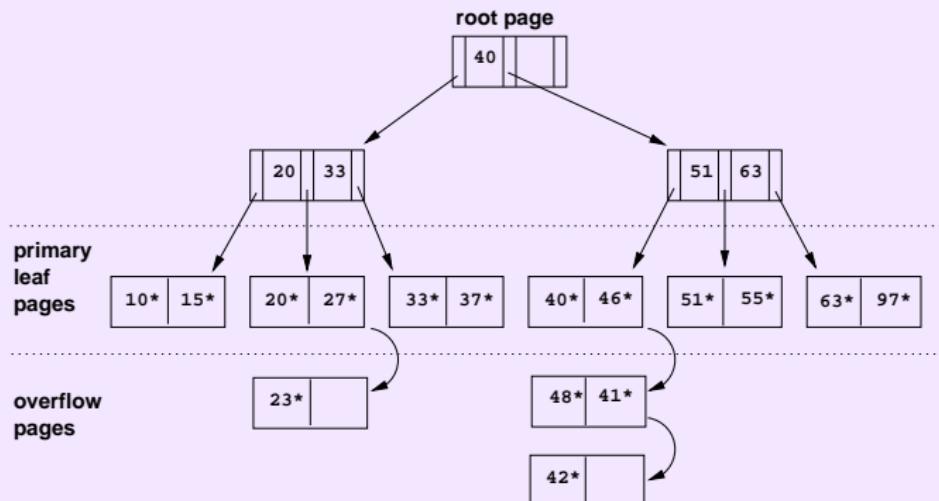
Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

Example (After insertion of data records with keys 23, 48, 41, 42)



# Multi-Level ISAM Structure: Deletions

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

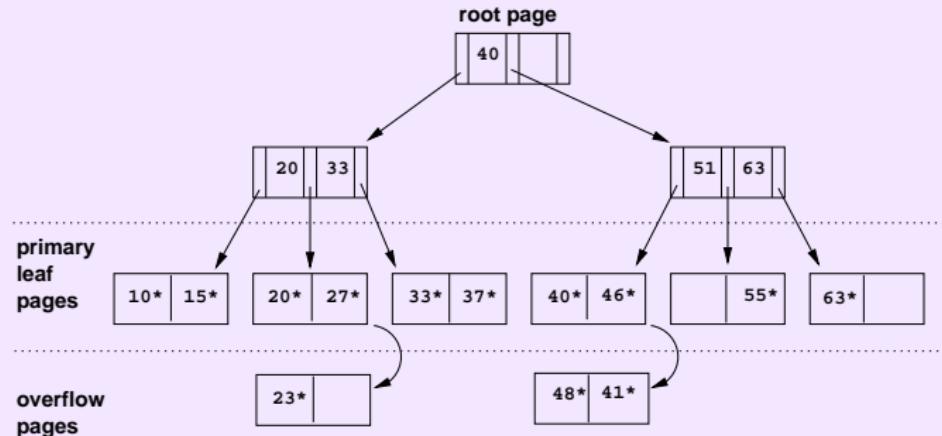
Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

Example (After deletion of data records with keys 42, 51, 97)



## ISAM: Too Static?

- The non-leaf levels of the ISAM structure have not been touched at all by the data file updates.
- This may lead to index key entries which do not appear in the index leaf level (e.g., key value 51 on the previous slide).

### 📎 Orphaned index entries

Does an index key entry like 51 above lead to problems during index key searches?

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## ISAM: Too Static?

- The non-leaf levels of the ISAM structure have not been touched at all by the data file updates.
- This may lead to index key entries which do not appear in the index leaf level (e.g., key value 51 on the previous slide).

### 📎 Orphaned index entries

Does an index key entry like 51 above lead to problems during index key searches?

No, since the index keys maintain their separator property.

- To preserve the **separator property** of the index key entries, maintenance of overflow chains is required.
- ⇒ ISAM may **lose balance** after heavy updating. This complicates life for the query optimizer.

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## ISAM: Being 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 need not be locked** during concurrent index access.
    - Locking can be a serious bottleneck in dynamic tree indexes (particularly near the index root node).
- ⇒ ISAM may be the index of choice for relatively static data.

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees



## ISAM: Efficiency of Searches

- Regardless of these deficiencies, ISAM-based searching is the most efficient **order-aware** index structure discussed so far:

### Definition (ISAM fanout)

- Let  $N$  be the number of pages in the data file, and let  $F$  denote the **fanout** of the ISAM tree, *i.e.*, the maximum number of children per index node
- The fanout in the previous example is  $F = 3$ , typical realistic fanouts are  $F \approx 1,000$ .
- When index searching starts, the search space is of size  $N$ . With the help of the root page we are guided into an index subtree of size

$$N \cdot 1/F$$

Tree-Structured  
Indexing

Torsten Grust



### Binary Search

#### ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

#### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## ISAM: Efficiency of Searches

- As we search down the tree, the search space is repeatedly reduced by a factor of  $F$ :

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

- Index searching ends after  $s$  steps when the search space has been reduced to size 1 (i.e., we have reached the index leaf level and found the data page that contains the wanted record):

$$N \cdot (1/F)^s = 1 \Leftrightarrow s = \log_F N .$$

- Since  $F \gg 2$ , this is significantly more efficient than access via binary search ( $\log_2 N$ ).

### Example (Required I/O operations during ISAM search)

Assume  $F = 1,000$ . An ISAM tree of **height 3** can index a file of one billion ( $10^9$ ) pages, i.e., 3 I/O operations are sufficient to locate the wanted data file page.

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-trees: A Dynamic Index Structure

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

The **B<sup>+</sup>-tree index** structure is derived from the ISAM idea, but is fully dynamic with respect to updates:

- Search performance is only dependent on the **height** of the B<sup>+</sup>-tree (because of high fan-out  $F$ , the height rarely exceeds 3).
- **No overflow chains** develop, a B<sup>+</sup>-tree remains **balanced**.
- B<sup>+</sup>-trees offer efficient **insert/delete procedures**, the underlying data file can grow/shrink **dynamically**.
- B<sup>+</sup>-tree nodes (despite the root page) are **guaranteed to have a minimum occupancy of 50 %** (typically  $2/3$ ).

↗ Original publication (B-tree): R. Bayer and E.M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, vol. 1, no. 3, September 1972.

## B<sup>+</sup>-trees: Basics

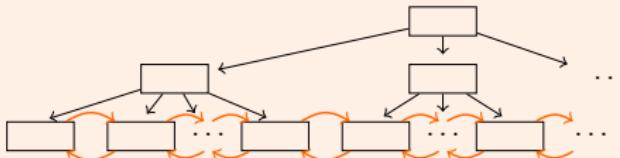
B<sup>+</sup>-trees resemble ISAM indexes, where

- leaf nodes are connected to form a **doubly-linked list**, the so-called **sequence set**,<sup>1</sup>
- leaves may contain **actual data records** or just **references to records** on data pages (*i.e.*, index entry variants **B** or **C**).

*Here we assume the latter since this is the common case.*

Remember: ISAM leaves were the **data pages** themselves, instead.

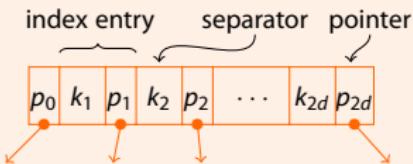
### Sketch of B<sup>+</sup>-tree structure (data pages not shown)



<sup>1</sup>This is not a strict B<sup>+</sup>-tree requirement, although most systems implement it.

## B<sup>+</sup>-trees: Non-Leaf Nodes

### B<sup>+</sup>-tree inner (non-leaf) node



B<sup>+</sup>-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<sup>+</sup>-tree:

$$d \leq n \leq 2 \cdot d \quad (\text{root node: } 1 \leq n \leq 2 \cdot d) .$$

- A node contains  $n + 1$  pointers. Pointer  $p_i$  ( $1 \leq i \leq n - 1$ ) points to a subtree in which all key values  $k$  are such that

$$k_i \leq k < k_{i+1} .$$

( $p_0$  points to a subtree with key values  $< k_1$ ,  $p_n$  points to a subtree with key values  $\geq k_n$ ).



## B<sup>+</sup>-tree: Leaf Nodes

- B<sup>+</sup>-tree leaf nodes contain pointers to data **records** (not **pages**). A leaf node entry with key value  $k$  is denoted as  $k*$  as before.
- Note that we can use all index entry variants **A**, **B**, **C** to implement the leaf entries:
  - For variant **A**, the B<sup>+</sup>-tree represents the index as well as the data file itself. Leaf node entries thus look like

$$k_i* = \langle k_i, \langle \dots \rangle \rangle .$$

- For variants **B** and **C**, the B<sup>+</sup>-tree lives in a file distinct from the actual data file. Leaf node entries look like

$$k_i* = \langle k_i, rid \rangle .$$

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Search

Below, we assume that key values are **unique** (we defer the treatment of **duplicate key values**).

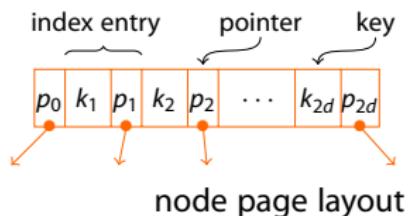
### B<sup>+</sup>-tree search

```
1 Function: search ( $k$ )
2   return tree_search ( $k$ , root);


---


1 Function: tree_search ( $k$ , node)
2 if node is a leaf then
3   return node;
4 switch  $k$  do
5   case  $k < k_1$ 
6     return tree_search ( $k$ ,  $p_0$ );
7   case  $k_i \leq k < k_{i+1}$ 
8     return tree_search ( $k$ ,  $p_i$ );
9   case  $k_{2d} \leq k$ 
10    return tree_search ( $k$ ,  $p_{2d}$ );
```

- Function  $\text{search}(k)$  returns a pointer to the leaf node page that contains potential hits for search key  $k$ .



## B<sup>+</sup>-tree: Insert

- Remember that B<sup>+</sup>-trees remain **balanced**<sup>2</sup> no matter which updates we perform. Insertions and deletions have to preserve this invariant.

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

---

<sup>2</sup>All paths from the B<sup>+</sup>-tree root to any leaf are of equal length.

## B<sup>+</sup>-tree: Insert

- Remember that B<sup>+</sup>-trees remain **balanced**<sup>2</sup> no matter which updates we perform. Insertions and deletions have to preserve this invariant.
- The basic principle of B<sup>+</sup>-tree insertion is simple:
  - ① To insert a record with key  $k$ , call  $\text{search}(k)$  to find the page  $p$  to hold the new record.  
Let  $m$  denote the number of entries on  $p$ .
  - ② If  $m < 2 \cdot d$  (i.e., there is capacity left on  $p$ ), store  $k_*$  in page  $p$ . **Otherwise ...?**

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

<sup>2</sup>All paths from the B<sup>+</sup>-tree root to any leaf are of equal length.

## B<sup>+</sup>-tree: Insert

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

- Remember that B<sup>+</sup>-trees remain **balanced**<sup>2</sup> no matter which updates we perform. Insertions and deletions have to preserve this invariant.
- The basic principle of B<sup>+</sup>-tree insertion is simple:
  - To insert a record with key  $k$ , call  $\text{search}(k)$  to find the page  $p$  to hold the new record.  
Let  $m$  denote the number of entries on  $p$ .
  - If  $m < 2 \cdot d$  (i.e., there is capacity left on  $p$ ), store  $k_*$  in page  $p$ . **Otherwise ...?**
- We must *not* start an overflow chain hanging off  $p$ : this would violate the balancing property.
- We want the cost for  $\text{search}(k)$  to be dependent on tree height only, so placing  $k_*$  somewhere else (even near  $p$ ) is *no* option either.

<sup>2</sup>All paths from the B<sup>+</sup>-tree root to any leaf are of equal length.

## B<sup>+</sup>-tree: Insert

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

- Sketch of the insertion procedure for entry  $\langle k, q \rangle$  (key value  $k$  pointing to  $rid\ q$ ):

- 1 **Find leaf page**  $p$  where we would expect the entry for  $k$ .
- 2 If  $p$  has **enough space** to hold the new entry (i.e., at most  $2d - 1$  entries in  $p$ ), **simply insert**  $\langle k, q \rangle$  into  $p$ .
- 3 Otherwise node  $p$  must be **split** into  $p$  and  $p'$  and a new **separator** has to be inserted into the parent of  $p$ .

Splitting happens recursively and may eventually lead to a split of the root node (increasing the tree height).

- 4 Distribute the entries of  $p$  and the new entry  $\langle k, q \rangle$  onto pages  $p$  and  $p'$ .

## B<sup>+</sup>-tree: Insert

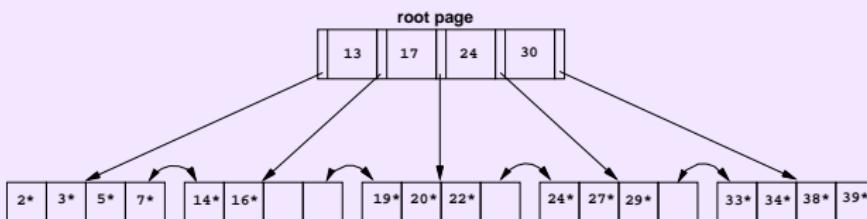
Tree-Structured  
Indexing

Torsten Grust



### Example (B<sup>+</sup>-tree insertion procedure)

- 1 Insert record with key  $k = 8$  into the following B<sup>+</sup>-tree:



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Insert

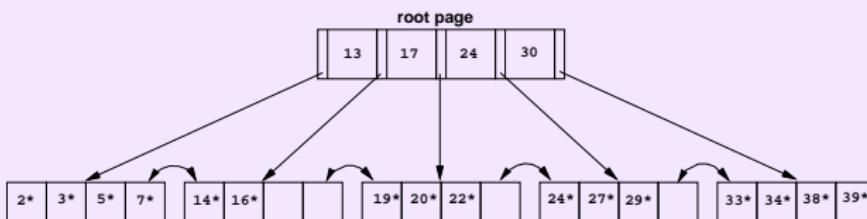
Tree-Structured  
Indexing

Torsten Grust



### Example (B<sup>+</sup>-tree insertion procedure)

- 1 Insert record with key  $k = 8$  into the following B<sup>+</sup>-tree:



- 2 The left-most leaf page  $p$  has to be split. Entries 2\*, 3\* remain on  $p$ , entries 5\*, 7\*, and 8\* (new) go on new page  $p'$ .

Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Insert and Leaf Node Split

Tree-Structured  
Indexing

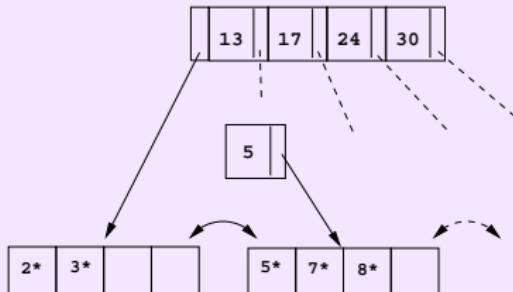
Torsten Grust



### Example (B<sup>+</sup>-tree insertion procedure)

- ③ Pages  $p$  and  $p'$  are shown below.

Key  $k' = 5$ , the **new separator** between pages  $p$  and  $p'$ , has to be **inserted into the parent** of  $p$  and  $p'$  **recursively**:



- Note that, after such a **leaf node split**, the new separator key  $k' = 5$  is **copied up** the tree: the entry  $5*$  itself has to remain in its leaf page.

Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Insert and Non-Leaf Node Split

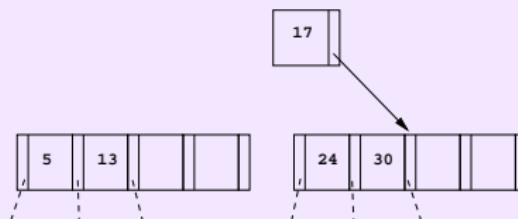
Tree-Structured  
Indexing

Torsten Grust



### Example (B<sup>+</sup>-tree insertion procedure)

- The insertion process is propagated upwards the tree:  
inserting key  $k' = 5$  into the parent leads to a **non-leaf node split** (the  $2 \cdot d + 1$  keys and  $2 \cdot d + 2$  pointers make for two new non-leaf nodes and a **middle key** which we propagate further up for insertion):



- Note that, for a **non-leaf node** split, we can simply **push up** the middle key (17). Contrast this with a leaf node split.

Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Insert and Root Node Split

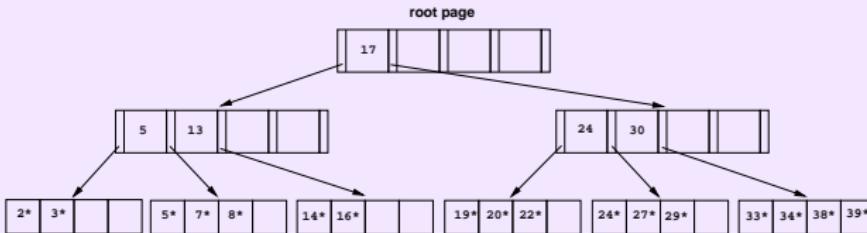
Tree-Structured  
Indexing

Torsten Grust



### Example

- 5 Since the split node was the root node, we create a **new root node** which holds the pushed up middle key only:



- Splitting the old root and creating a new root node is the *only* situation in which the **B<sup>+</sup>-tree height increases**. The B<sup>+</sup>-tree thus remains balanced.

Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Insert

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

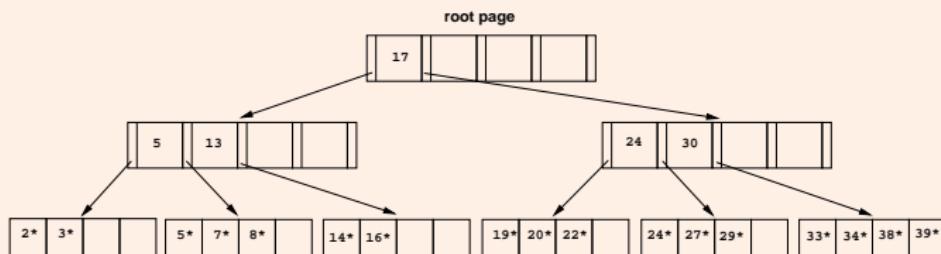
Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## Further key insertions

How does the insertion of records with keys  $k = 23$  and  $k = 40$  alter the B<sup>+</sup>-tree?



## B<sup>+</sup>-tree Insert: Further Examples

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

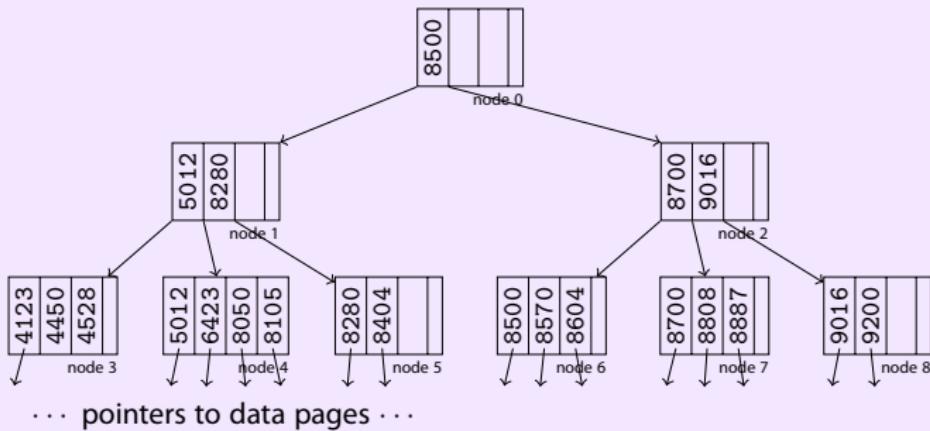
Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees



Insert new entry with key 4222.

## B<sup>+</sup>-tree Insert: Further Examples

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

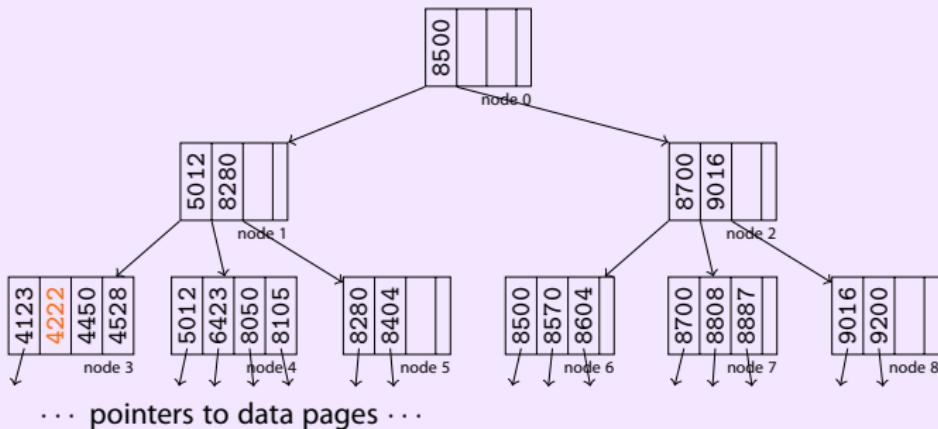
Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees



Insert new entry with key 4222.

- ⇒ Enough space in node 3, simply insert.
- ⇒ Keep entries **sorted within nodes**.

## B<sup>+</sup>-tree Insert: Further Examples

## Tree-Structured Indexing

Torsten Grust



## Binary Search

ISAM

Multi-Level ISAM

## Too Static?

## Search Efficiency

## B<sup>+</sup>-trees

Search

Insert

## Redistribution

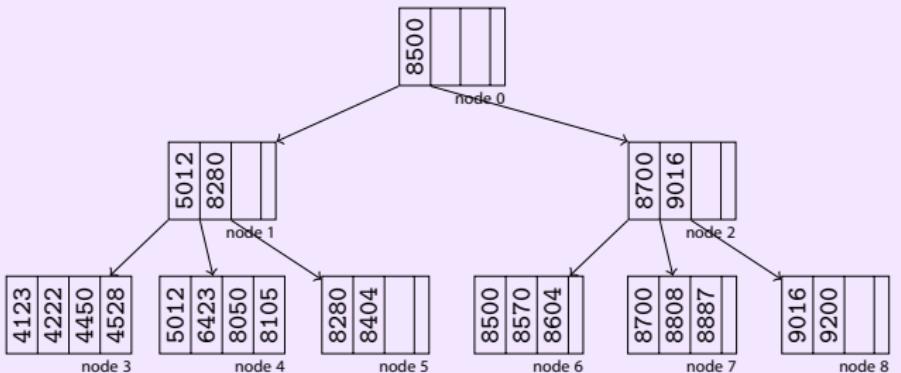
Delete

## Duplicates

### Key Compress

## Bulk Loading

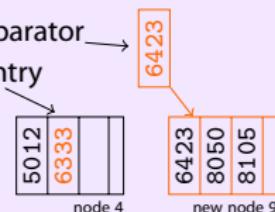
## Partitioned B<sup>+</sup>-tree



Insert key 6333.

⇒ Must **split** node 4.

⇒ **New separator** goes into node 1  
(including pointer to new page).



## B<sup>+</sup>-tree Insert: Further Examples

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

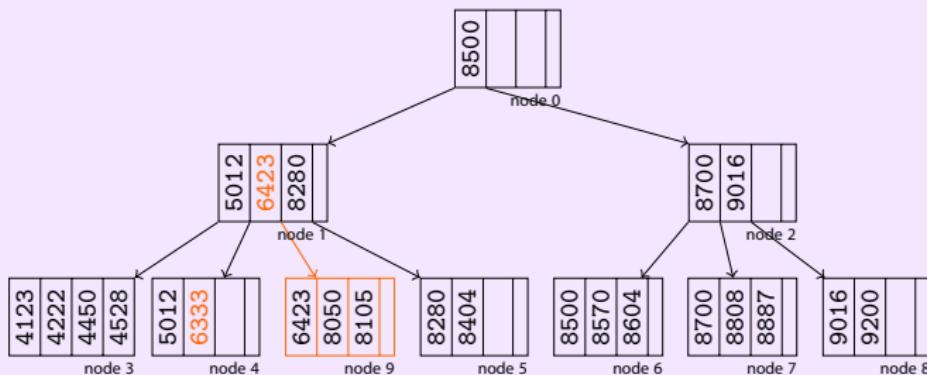
Delete

Duplicates

Key Compression

Bulk Loading

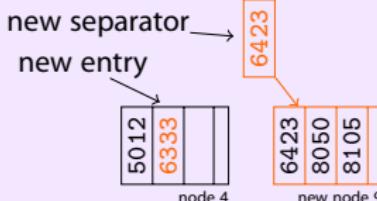
Partitioned B<sup>+</sup>-trees



Insert key 6333.

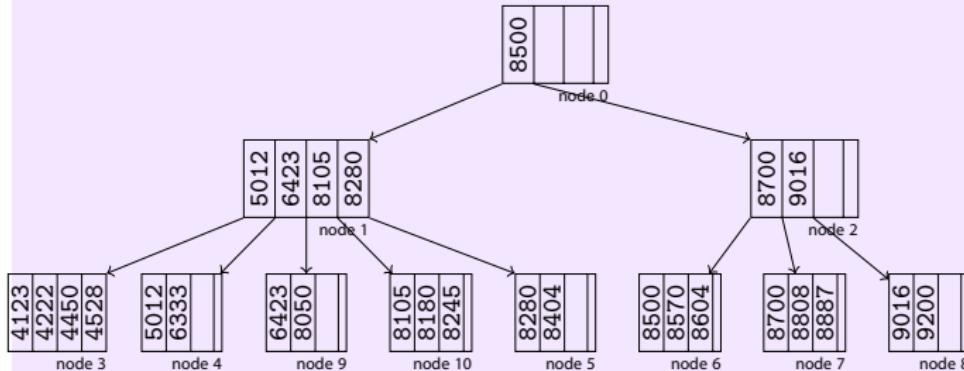
⇒ Must **split** node 4.

⇒ **New separator** goes into node 1  
(including pointer to new page).



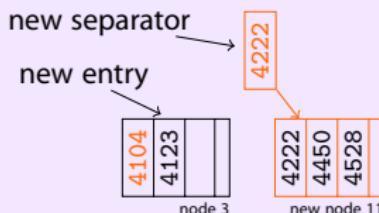
## B<sup>+</sup>-tree Insert: Further Examples

### Example (B<sup>+</sup>-tree insertion procedure)



After 8180, 8245, insert key 4104.

⇒ Must **split** node 3.



Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree Insert: Further Examples

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

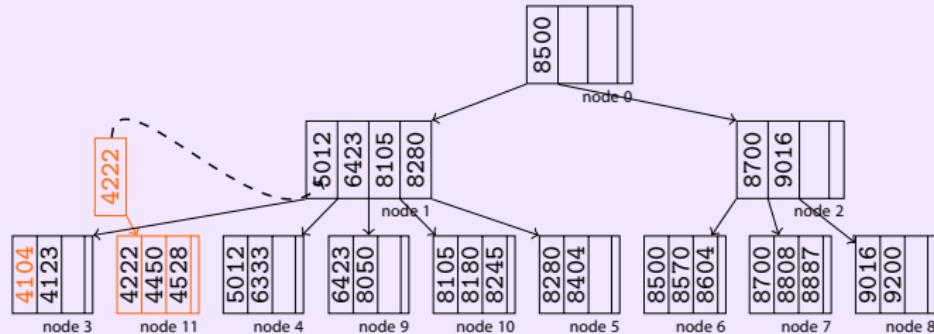
Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

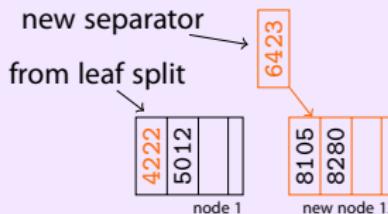


After 8180, 8245, insert key 4104.

- ⇒ Must **split** node 3.
- ⇒ Node 1 overflows ⇒ split it
- ⇒ **New separator** goes into root

Unlike during leaf split, separator key does **not** remain in inner node.

☞ Why?



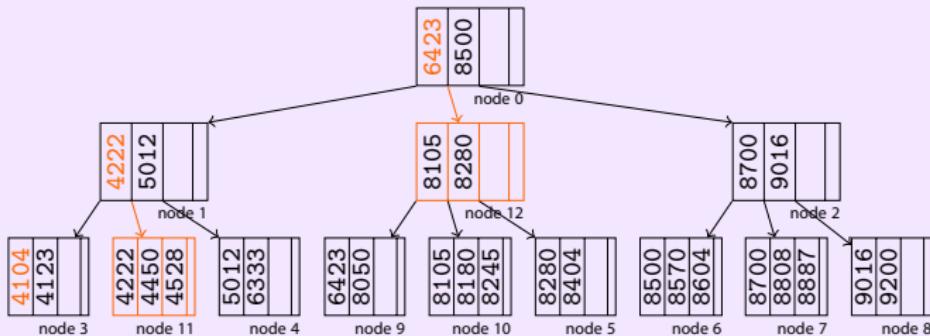
## B<sup>+</sup>-tree Insert: Further Examples

Tree-Structured  
Indexing

Torsten Grust



### Example (B<sup>+</sup>-tree insertion procedure)

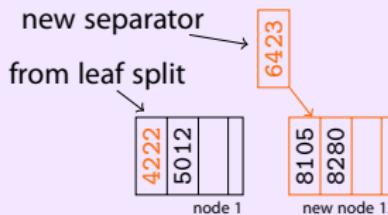


After 8180, 8245, insert key 4104.

- ⇒ Must **split** node 3.
- ⇒ Node 1 overflows ⇒ split it
- ⇒ **New separator** goes into root

Unlike during leaf split, separator key does **not** remain in inner node.

☞ Why?



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Root Node Split

- Splitting starts at the leaf level and continues upward as long as index nodes are fully occupied.
- Eventually, this can lead to a split of the **root node**:
  - Split like any other inner node.
  - Use the separator to create a **new root**.
- The root node is the **only** node that may have an occupancy of less than 50 %.
- This is the **only** situation where the tree height increases.

💡 How often do you expect a root split to happen?

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Root Node Split

- Splitting starts at the leaf level and continues upward as long as index nodes are fully occupied.
- Eventually, this can lead to a split of the **root node**:
  - Split like any other inner node.
  - Use the separator to create a **new root**.
- The root node is the **only** node that may have an occupancy of less than 50 %.
- This is the **only** situation where the tree height increases.

### 💡 How often do you expect a root split to happen?

E.g., B<sup>+</sup>-tree over 8 byte integers, 4 KB pages;  
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

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Insertion Algorithm

```
1 Function: tree_insert (k, rid, node)
2 if node is a leaf then
3   return leaf_insert (k, rid, node);
4 else
5   switch k do
6     case k < k1
7       ⟨sep, ptr⟩ ← tree_insert (k, rid, p0);
8     case ki ≤ k < ki+1
9       ⟨sep, ptr⟩ ← tree_insert (k, rid, pi);
10    case k2d ≤ k
11      ⟨sep, ptr⟩ ← tree_insert (k, rid, p2d);
12    if sep is null then
13      return ⟨null, null⟩;
14    else
15      return non_leaf_insert (sep, ptr, node);
```

} see tree\_search ()

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

```

1 Function: leaf_insert ( $k$ ,  $rid$ ,  $node$ )
2 if another entry fits into  $node$  then
3   insert  $\langle k, rid \rangle$  into  $node$  ;
4   return  $\langle \text{null}, \text{null} \rangle$ ;
5 else
6   allocate new leaf page  $p$  ;
7   let  $\{ \langle k_1^+, rid_1^+ \rangle, \dots, \langle k_{2d+1}^+, rid_{2d+1}^+ \rangle \} :=$  entries from  $node \cup \{ \langle k, rid \rangle \}$ 
8   leave entries  $\langle k_1^+, rid_1^+ \rangle, \dots, \langle k_d^+, rid_d^+ \rangle$  in  $node$  ;
9   move entries  $\langle k_{d+1}^+, rid_{d+1}^+ \rangle, \dots, \langle k_{2d+1}^+, rid_{2d+1}^+ \rangle$  to  $p$  ;
10  return  $\langle k_{d+1}^+, p \rangle$ ;

```

```

1 Function: non_leaf_insert ( $k$ ,  $ptr$ ,  $node$ )
2 if another entry fits into  $node$  then
3   insert  $\langle k, ptr \rangle$  into  $node$  ;
4   return  $\langle \text{null}, \text{null} \rangle$ ;
5 else
6   allocate new non-leaf page  $p$  ;
7   let  $\{ \langle k_1^+, p_1^+ \rangle, \dots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle \} :=$  entries from  $node \cup \{ \langle k, ptr \rangle \}$ 
8   leave entries  $\langle k_1^+, p_1^+ \rangle, \dots, \langle k_d^+, p_d^+ \rangle$  in  $node$  ;
9   move entries  $\langle k_{d+2}^+, p_{d+2}^+ \rangle, \dots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle$  to  $p$  ;
10  set  $p_0 \leftarrow p_{d+1}^+$  in  $p$  ;
11  return  $\langle k_{d+1}^+, p \rangle$ ;

```

# B<sup>+</sup>-tree: Insertion Algorithm

## B<sup>+</sup>-tree insertion algorithm

```
1 Function: insert (k, rid)
2    $\langle \text{key}, \text{ptr} \rangle \leftarrow \text{tree\_insert} (\text{k}, \text{rid}, \text{root});$ 
3   if key is not null then
4     allocate new root page r;
5     populate r with
6        $p_0 \leftarrow \text{root};$ 
7        $k_1 \leftarrow \text{key};$ 
8        $p_1 \leftarrow \text{ptr};$ 
9      $\text{root} \leftarrow r;$ 
```

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

- *insert (k, rid)* is called from outside.
- Variable *root* contains a pointer to the B<sup>+</sup>-tree root page.
- Note how leaf node entries point to *rids*, while inner nodes contain pointers to other B<sup>+</sup>-tree nodes.

## B<sup>+</sup>-tree Insert: Redistribution

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

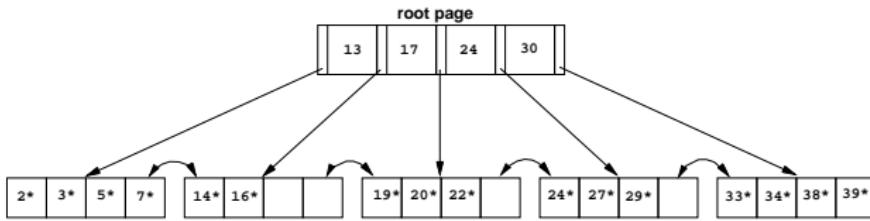
Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

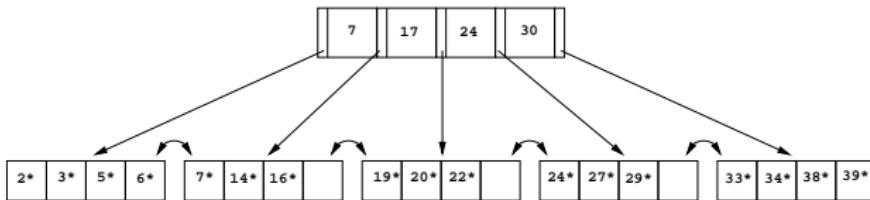
- We can further **improve the average occupancy** of B<sup>+</sup>-tree using a technique called **redistribution**.
- Suppose we are trying to insert a record with key  $k = 6$  into the B<sup>+</sup>-tree below:



- The left-most leaf is full already, its right **sibling** still has capacity, however.

## B<sup>+</sup>-tree Insert: Redistribution

- In this situation, we can avoid growing the tree by **redistributing** entries between siblings (entry 7\* moved into right sibling):



- We have to **update the parent node** (new separator 7) to reflect the redistribution.
  - Inspecting one or both neighbor(s) of a B<sup>+</sup>-tree node involves additional I/O operations.
- ⇒ Actual implementations often use redistribution on the leaf level only (because the sequence set page chaining gives direct access to both sibling pages).

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees



## B<sup>+</sup>-tree Insert: Redistribution

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

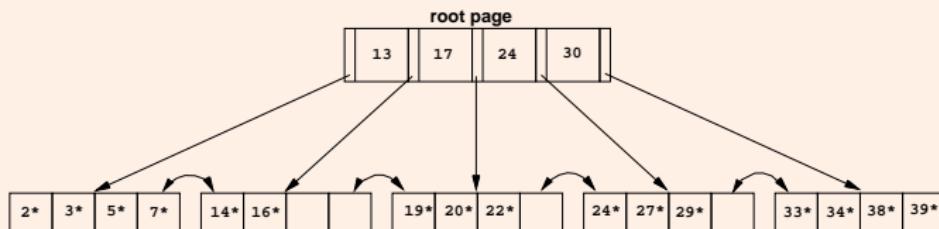
Partitioned B<sup>+</sup>-trees

### 📎 Redistribution makes a difference

Insert a record with key  $k = 30$

- ① without redistribution,
- ② using leaf level redistribution

into the B<sup>+</sup>-tree shown below. How does the tree change?



## B<sup>+</sup>-tree: Delete

- The principal idea to implement B<sup>+</sup>-tree deletion comes as no surprise:
    - ➊ To delete a record with key  $k$ , use  $\text{search}(k)$  to locate the leaf page  $p$  containing the record.  
Let  $m$  denote the number of entries on  $p$ .
    - ➋ If  $m > d$  then  $p$  has sufficient occupancy: simply delete  $k^*$  from  $p$  (if  $k^*$  is present on  $p$  at all).
- Otherwise ...?**

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Delete

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

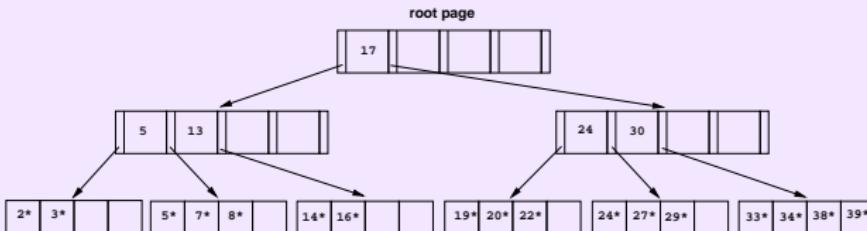
Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

### Example (B<sup>+</sup>-tree deletion procedure)

- 1 Delete record with key  $k = 19$  (i.e., entry  $19*$ ) from the following B<sup>+</sup>-tree:



- 2 A call to search(19) leads us to leaf page  $p$  containing entries  $19*$ ,  $20*$ , and  $22*$ . We can safely remove  $19*$  since  $m = 3 > 2$  (**no page underflow** in  $p$  after removal).

## B<sup>+</sup>-tree: Delete and Leaf Redistribution

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

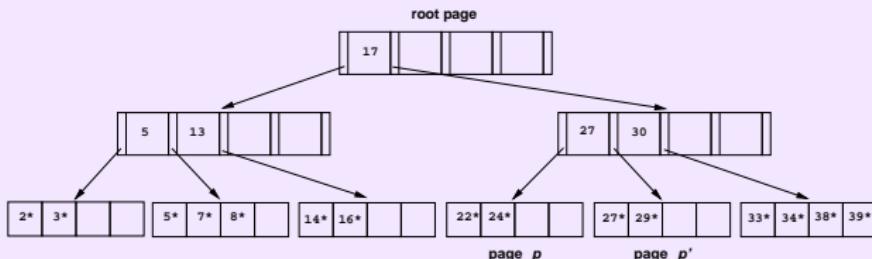
Partitioned B<sup>+</sup>-trees

### Example (B<sup>+</sup>-tree deletion procedure)

- 3 Subsequent deletion of 20\*, however, lets  $p$  **underflow** ( $p$  has minimal occupancy of  $d = 2$  already).

We now use **redistribution** and borrow entry 24\* from the right **sibling**  $p'$  of  $p$  (since  $p'$  hosts  $3 > 2$  entries, redistribution won't let  $p'$  underflow).

The smallest key value on  $p'$  (27) is the **new separator** of  $p$  and  $p'$  in their common parent:



## B<sup>+</sup>-tree: Delete and Leaf Merging

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

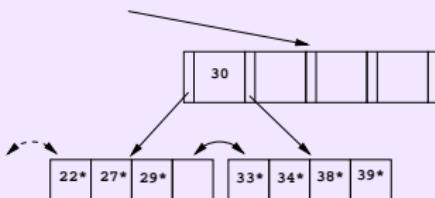
Bulk Loading

Partitioned B<sup>+</sup>-trees

### Example (B<sup>+</sup>-tree deletion procedure)

- 4 We continue and delete entry 24\* from  $p$ . Redistribution is no option now (sibling  $p'$  has minimal occupancy of  $d = 2$ ). We now have  $m_p + m_{p'} = 1 + 2 < 2 \cdot d$  however: B<sup>+</sup>-tree deletion thus **merges leaf nodes**  $p$  and  $p'$ .

Move entries 27\*, 29\* from  $p'$  to  $p$ , then delete page  $p'$ :



- **NB:** the separator 27 between  $p$  and  $p'$  is no longer needed and thus **discarded (recursively deleted)** from the parent.

## B<sup>+</sup>-tree: Delete and Non-Leaf Node Merging

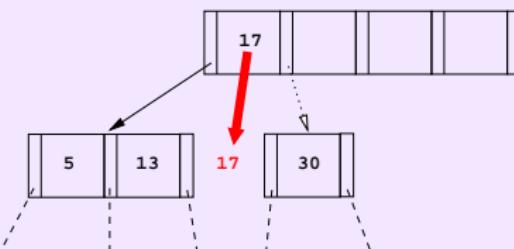
### Example (B<sup>+</sup>-tree deletion procedure)

- 5 The parent of  $p$  experiences underflow. Redistribution is no option, so we **merge with left non-leaf sibling**.

After merging we have

$\underbrace{d}_{\text{left}} + \underbrace{(d - 1)}_{\text{right}}$  keys and  $\underbrace{d + 1}_{\text{left}} + \underbrace{d}_{\text{right}}$  pointers

on the merged page:



The missing key value, namely the separator of the two nodes (17), **is pulled down** (and thus deleted) from the parent to form the complete merged node.

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Root Deletion

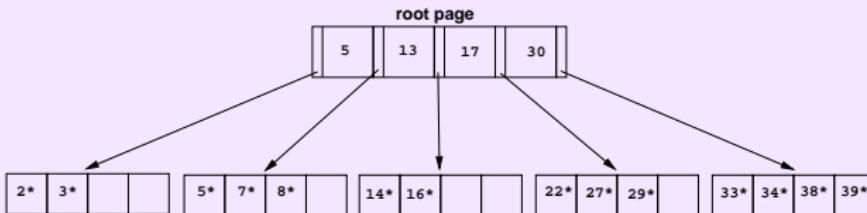
Tree-Structured  
Indexing

Torsten Grust



### Example (B<sup>+</sup>-tree deletion procedure)

- 6 Since we have now deleted the last remaining entry in the root, we discard the root (and make the merged node the new root):



- This is the *only* situation in which the **B<sup>+</sup>-tree height decreases**. The B<sup>+</sup>-tree thus remains balanced.

Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Delete and Non-Leaf Node Redistribution

Tree-Structured  
Indexing

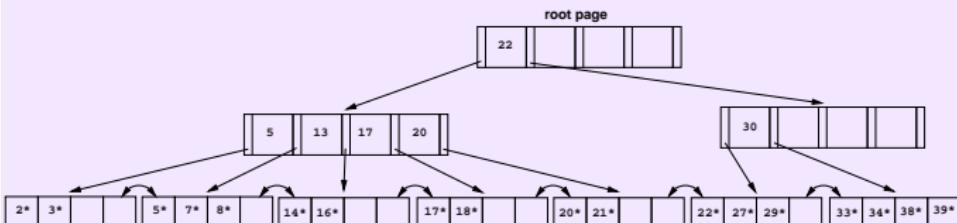
Torsten Grust



### Example (B<sup>+</sup>-tree deletion procedure)

- 7 We have now seen **leaf node merging** and **redistribution** as well as **non-leaf node merging**. The remaining case of **non-leaf node redistribution** is straightforward:

- Suppose *during* deletion we encounter the following B<sup>+</sup>-tree:



- The non-leaf node with entry 30 underflowed. Its left sibling has two entries (17 and 20) to spare.

Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Delete and Non-Leaf Node Redistribution

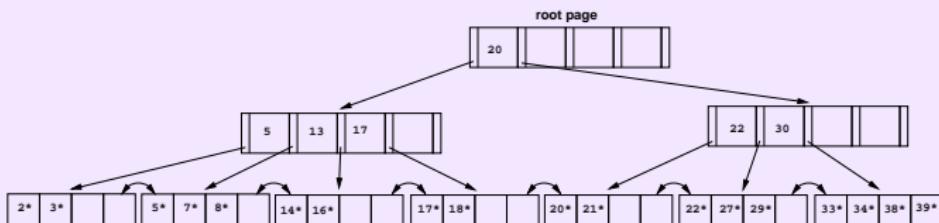
Tree-Structured  
Indexing

Torsten Grust



### Example (B<sup>+</sup>-tree deletion procedure)

- 8 We redistribute entry 20 by “**rotating it through**” the parent. The former parent entry 22 is pushed down:



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## Merge and Redistribution Effort

- Actual DBMS implementations often avoid the cost of merging and/or redistribution, but relax the minimum occupancy rule.

### DB2. B<sup>+</sup>-tree deletion

- System parameter MINPCTUSED (*minimum percent used*) controls when the kernel should try a **leaf node merge** (“online index reorg”).  
(This is particularly simple because of the sequence set pointers connecting adjacent leaves, see slide 40.)
- Non-leaf nodes are never merged** (a “full index reorg” is required to achieve this).
- To improve concurrency, deleted index entries are merely **marked as deleted** and only removed later (IBM DB2 UDB type-2 indexes).

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Duplicates

- As discussed here, the B<sup>+</sup>-tree search, insert (and delete) procedures ignore the presence of **duplicate** key values.
- Often this is a reasonable assumption:
  - 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.

### DB2. Treatment of duplicate keys

Since duplicate keys add to the B<sup>+</sup>-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 system-maintained unique identifiers for each tuple in a table, and
- are *not* dependent on tuple order and *never* rise again.

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Duplicates (IBM DB2 Tuple Identities)

Tree-Structured  
Indexing

Torsten Grust



## DB2. Expose IBM DB2 tuple identity

```
1 $ db2
2 (c) Copyright IBM Corporation 1993,2007
3 Command Line Processor for DB2 Client 9.5.0
4 [...]
5 db2 => CREATE TABLE FOO(ROWID INT GENERATED ALWAYS AS IDENTITY,
6                           text varchar(10))
7 db2 => INSERT INTO FOO VALUES (DEFAULT, 'This\u00f8 is'), ...
8 db2 => SELECT * FROM FOO
9
10 ROWID      TEXT
11 -----
12          1 This is
13          2 nothing
14          3 but a
15          4 silly
16          5 example!
```

Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Duplicates (IBM DB2 Tuple Identities)

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

```
1 db2 => DELETE FROM FOO WHERE TEXT = 'silly'
2 db2 => SELECT * FROM FOO
3
4 ROWID      TEXT
5 -----
6          1 This is
7          2 nothing
8          3 but a
9          5 example!
10
11 db2 => INSERT INTO FOO VALUES (DEFAULT, 'I am new.')
12 db2 => SELECT * FROM FOO
13
14 ROWID      TEXT
15 -----
16          1 This is
17          2 nothing
18          3 but a
19          6 I am new.
20          5 example!
```

## B<sup>+</sup>-tree: Duplicates

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

Other approaches alter the B<sup>+</sup>-tree implementation to add real awareness for duplicates:

- ① Use variant C (see slide 3.22) 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 repeatedly stored (space savings).
- B<sup>+</sup>-tree search and maintenance routines largely unaffected. Index data entry size varies, however (this affects the B<sup>+</sup>-tree **order** concept).
- Implemented in IBM Informix Dynamic Server, for example.

## B<sup>+</sup>-tree: Duplicates

Tree-Structured  
Indexing

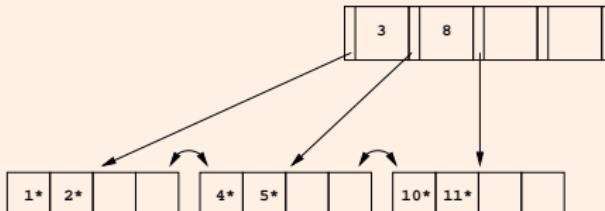
Torsten Grust



- ② Treat duplicate key values like any other value in insert and delete. This affects the search procedure.

### 📎 Impact of duplicate insertion on search

Given the following B<sup>+</sup>-tree of order  $d = 2$ , perform insertions (do not use redistribution):  $\text{insert}(2, \cdot)$ ,  $\text{insert}(2, \cdot)$ ,  $\text{insert}(2, \cdot)$ :



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

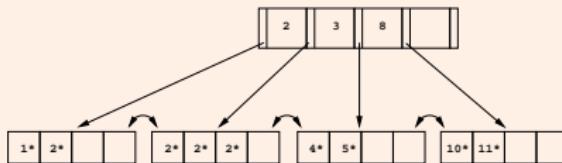
Bulk Loading

Partitioned B<sup>+</sup>-trees

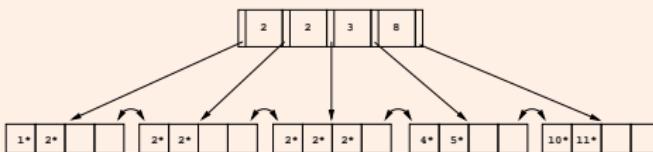
## B<sup>+</sup>-tree: Duplicates

### Impact of duplicate insertion on search

The resulting B<sup>+</sup>-tree is shown here. Now apply  $\text{insert}(2, \cdot)$ ,  $\text{insert}(2, \cdot)$  to this B<sup>+</sup>-tree:



We get the tree depicted below:



- ⇒ In search: in a non-leaf node, follow the **rightmost** page pointer  $p_i$  such that  $k_i < k$  — assume a (non-existent)  $k_0 = -\infty$ .

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

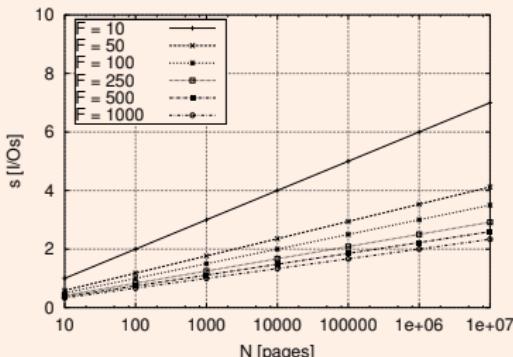
Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Key Compression

- Recall the search I/O effort  $s$  in an ISAM or B<sup>+</sup>-tree for a file of  $N$  pages. The **fan-out**  $F$  has been the deciding factor:

$$s = \log_F N .$$

### 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<sup>+</sup>-tree implementation.

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Key Compression

- Index entries in inner (i.e., non-leaf) B<sup>+</sup>-tree nodes are pairs

$$\langle k_i, \text{pointer to } p_i \rangle .$$

- The representation of page pointers is prescribed by the DBMS's pointer representation, and especially for key field types like CHAR(·) or VARCHAR(·), we will have

$$|\text{pointer}| \ll |k_i| .$$

- To **minimize the size of keys**, observe that key values in inner index nodes are used only to direct traffic to the appropriate leaf page:

### Excerpt of search( $k$ )

```
1 switch  $k$  do
2   case  $k < k_1$ 
3     ...
4   case  $k_i \leq k < k_{i+1}$ 
5     ...
6   case  $k_{2d} \leq k$ 
7     ...
```

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Key Compression

- Index entries in inner (i.e., non-leaf) B<sup>+</sup>-tree nodes are pairs

$$\langle k_i, \text{pointer to } p_i \rangle .$$

- The representation of page pointers is prescribed by the DBMS's pointer representation, and especially for key field types like CHAR(·) or VARCHAR(·), we will have

$$|\text{pointer}| \ll |k_i| .$$

- To **minimize the size of keys**, observe that key values in inner index nodes are used only to direct traffic to the appropriate leaf page:

### Excerpt of search( $k$ )

```
1 switch  $k$  do
2   case  $k < k_1$ 
3     ...
4   case  $k_i \leq k < k_{i+1}$ 
5     ...
6   case  $k_{2d} \leq k$ 
7     ...
```

⇒ The actual key values are *not* needed as long as we maintain their separator property.

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Key Compression

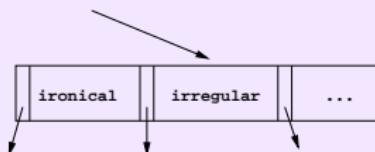
Tree-Structured  
Indexing

Torsten Grust



### Example (Searching a B<sup>+</sup>-tree node with VARCHAR(·) keys)

To guide the search across this B<sup>+</sup>-tree node



it is sufficient to store the **prefixes** iro and irr.



We must preserve the B<sup>+</sup>-tree semantics, though:

*All index entries stored in the subtree left of iro have keys  $k < \text{iro}$  and index entries stored in the subtree right of iro have keys  $k \geq \text{iro}$  (and  $k < \text{irr}$ ).*

Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Key Suffix Truncation

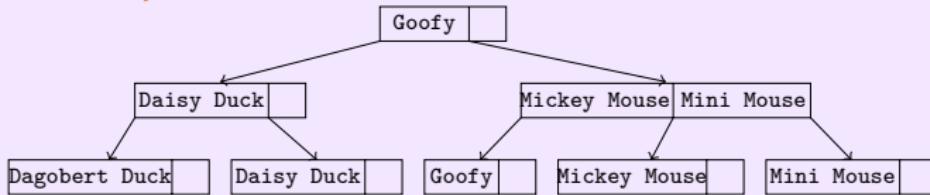
Tree-Structured  
Indexing

Torsten Grust



Example (Key suffix truncation, B<sup>+</sup>-tree with order  $d = 1$ )

Before key suffix truncation:



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Key Suffix Truncation

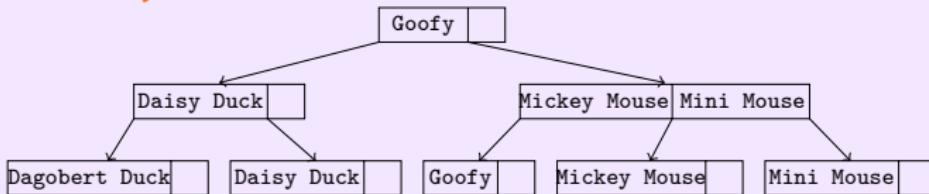
Tree-Structured  
Indexing

Torsten Grust

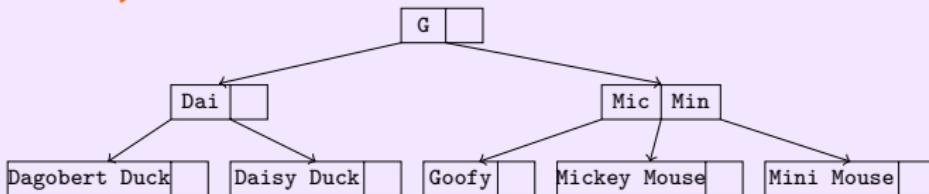


Example (Key suffix truncation, B<sup>+</sup>-tree with order  $d = 1$ )

Before key suffix truncation:



After key suffix truncation:



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Key Suffix Truncation

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

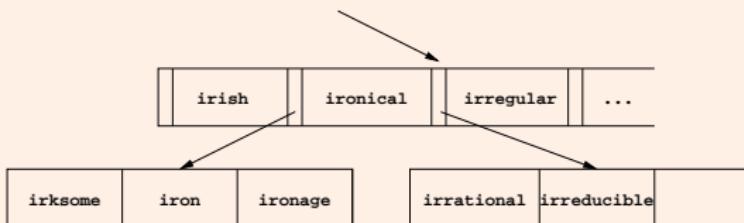
Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

### Key suffix truncation

How would a B<sup>+</sup>-tree **key compressor** alter the key entries in the inner node of this B<sup>+</sup>-tree snippet?



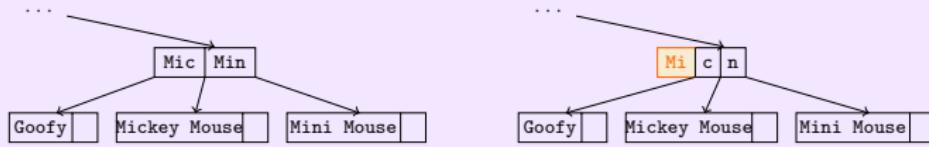
## B<sup>+</sup>-tree: Key Prefix Compression

Tree-Structured  
Indexing

Torsten Grust



### Example (Shared key prefixes in inner B<sup>+</sup>-tree nodes)



### Key prefix compression:

- Store common prefix only once (e.g., as " $k_0$ ")
- Keys have become highly discriminative now.

Violating the 50% occupancy rule can help to improve the effectiveness of prefix compression.

↗ Rudolf Bayer, Karl Unterauer: Prefix B-Trees. *ACM TODS* 2(1), March 1977.

Binary Search

ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Bulk Loading

- Consider the following database session (this might as well be commands executed on behalf of a database transaction):

### Table and index creation

```
1 CREATE TABLE foo (id INT, text VARCHAR(10));  
2  
3 [... insert 1,000,000 rows into table foo ...]  
4  
5 CREATE INDEX foo_idx ON foo (id ASC)
```

- The last SQL command initiates 1,000,000 calls to the B<sup>+</sup>-tree `insert()` procedure—a so-called index **bulk load**.
- ⇒ The DBMS will traverse the growing B<sup>+</sup>-tree index from its root down to the leaf pages 1,000,000 times.

 This is bad ...

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

Tree-Structured Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Bulk Loading

- Most DBMS installations ship with a **bulk loading utility** to reduce the cost of operations like the above.

### B<sup>+</sup>-tree bulk loading algorithm

- Create a sequence of pages that contains a **sorted list** of index entries  $k*$  for each key  $k$  in the data file.

**Note:** For index variants B or C, this does *not* imply to sort the data file itself. (For variant A, we effectively create a clustered index.)

- Allocate an empty index root page and let its  $p_0$  page pointer point to the first page of sorted  $k*$  entries.

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

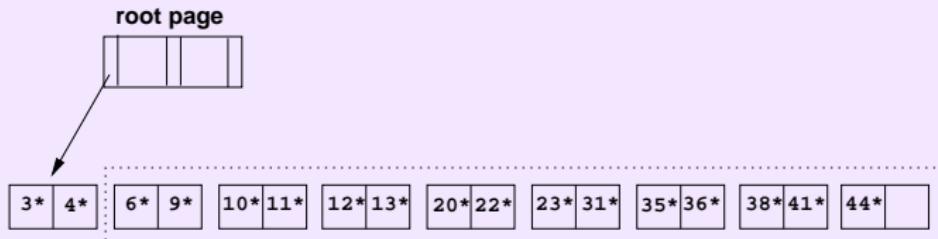
## B<sup>+</sup>-tree: Bulk Loading

Tree-Structured  
Indexing

Torsten Grust



Example (State of bulk load after step ②, order of B<sup>+</sup>-tree  $d = 1$ )



(Index leaf pages not yet in B<sup>+</sup>-tree are framed.)

### 📎 Bulk loading continued

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

Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-tree: Bulk Loading

- We now use the fact that the  $k*$  are **sorted**. Any insertion will thus hit the **right-most index node** (just above the leaf level).
- Use a specialized `bulk_insert(·)` procedure that avoids B<sup>+</sup>-tree root-to-leaf traversals altogether:

### B<sup>+</sup>-tree bulk loading algorithm (continued)

- ③ For each leaf level page  $p$ , insert the index entry

$\langle \text{minimum key on } p, \text{pointer to } p \rangle$

into the right-most index node just above the leaf level.

The right-most node is filled **left-to-right**. Splits occur only on the **right-most path** from the leaf level up to the root.

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

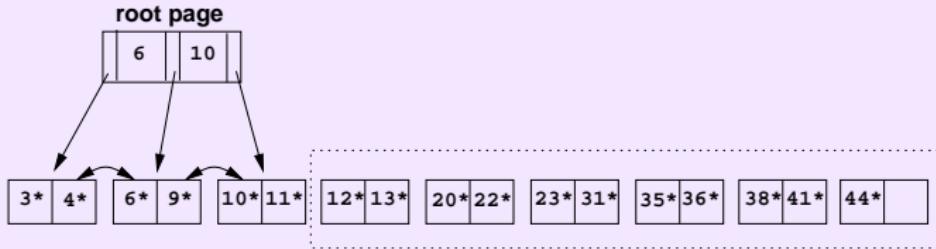
Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Bulk Loading

## Example (Bulk load continued)



Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

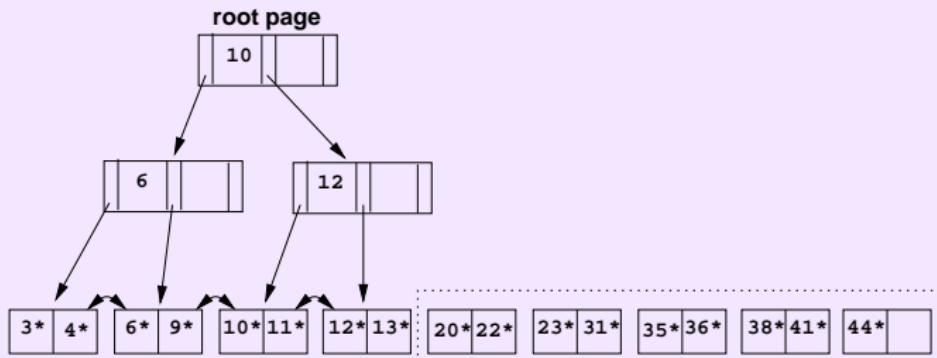
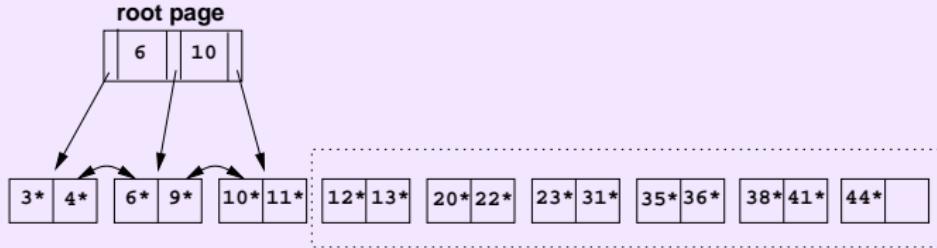
Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Bulk Loading

## Example (Bulk load continued)



Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

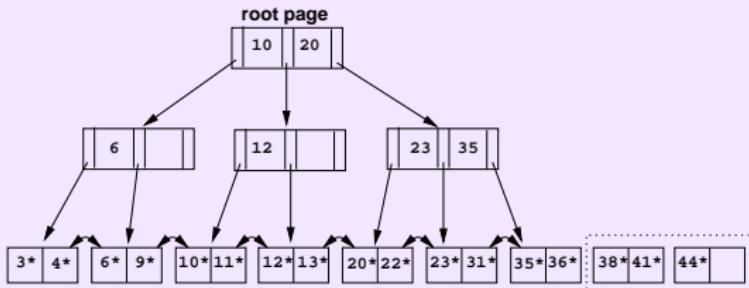
Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Bulk Loading

## Example (Bulk load continued)



Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

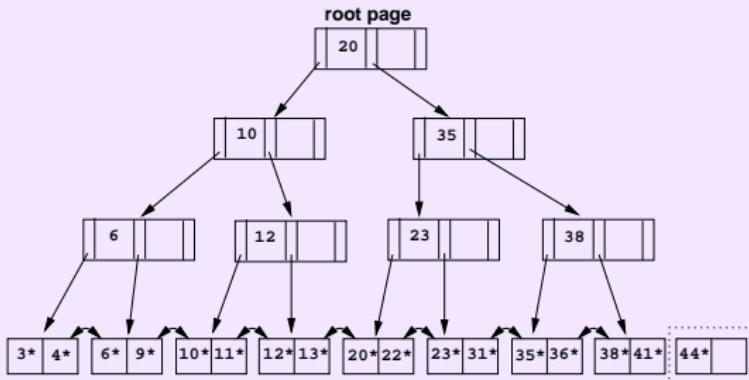
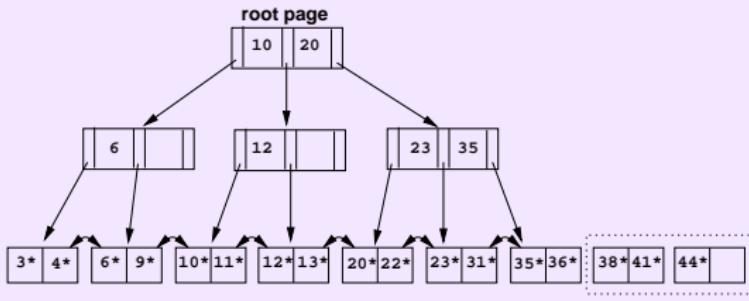
Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Bulk Loading

## Example (Bulk load continued)



Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## Composite Keys

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

B<sup>+</sup>-trees can (in theory<sup>3</sup>) be used to index everything with a defined **total order**, e.g.:

- integers, strings, dates, ..., and
- **concatenations** thereof (based on **lexicographical order**).

Possible in most SQL DDL dialects:

### Example (Create an index using a composite (concatenated) key)

```
CREATE INDEX ON TABLE CUSTOMERS (LASTNAME,  
FIRSTNAME);
```

A useful application are, e.g., **partitioned B-trees**:

- Leading index attributes effectively **partition** the resulting B<sup>+</sup>-tree.

↗ G. Graefe: Sorting And Indexing With Partitioned B-Trees. CIDR 2003.

<sup>3</sup>Some implementations won't allow you to index, e.g., large character fields.

## Partitioned B-trees

Tree-Structured  
Indexing

Torsten Grust



### Example (Index with composite key, low-selectivity key prefix)

```
CREATE INDEX ON TABLE STUDENTS (SEMESTER, ZIPCODE);
```



**What types of queries could this index support?**

Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

## Partitioned B-trees

Tree-Structured  
Indexing

Torsten Grust



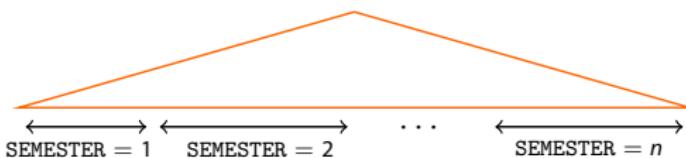
### Example (Index with composite key, low-selectivity key prefix)

```
CREATE INDEX ON TABLE STUDENTS (SEMESTER, ZIPCODE);
```



#### What types of queries could this index support?

The resulting B<sup>+</sup>-tree is going to look like this:



It can efficiently answer queries with, e.g.,

- **equality predicates** on SEMESTER **and** ZIPCODE,
- **equality** on SEMESTER and **range predicate** on ZIPCODE, or
- a **range predicate** on SEMESTER **only**.

Binary Search

ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading

Partitioned B<sup>+</sup>-trees

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

# Chapter 5

## Multi-Dimensional Indexing

### Indexing Points and Regions in $k$ Dimensions

*Architecture and Implementation of Database Systems*  
Summer 2014

Torsten Grust  
Wilhelm-Schickard-Institut für Informatik  
Universität Tübingen



## More Dimensions...

### One SQL query, many range predicates

```
1 SELECT *
2 FROM   CUSTOMERS
3 WHERE  ZIPCODE BETWEEN 8880 AND 8999
4 AND    REVENUE BETWEEN 3500 AND 6000
```

This query involves a **range predicate** in **two dimensions**.

Typical use cases for **multi-dimensional data** include

- **geographical data** (GIS: Geographical Information Systems),
- **multimedia retrieval** (e.g., image or video search),
- **OLAP** (Online Analytical Processing),
- queries against **encoded data** (e.g., XML)

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

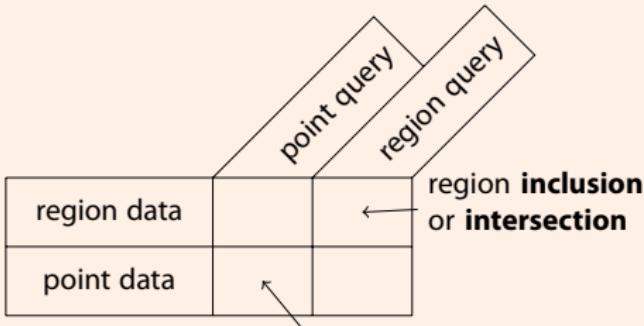
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## ...More Challenges...

Queries and data can be points or regions



... and you can come up with many more meaningful types of queries over multi-dimensional data.

Note: All equality searches can be reduced to one-dimensional queries.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

# Points, Lines, and Regions

Multi-Dimensional  
Indexing

Torsten Grust



## B+-trees...

... over composite keys

## Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

## k-d Trees

Balanced Construction

## K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

## R-Trees

Searching and Inserting

Splitting R-Tree Nodes

## UB-Trees

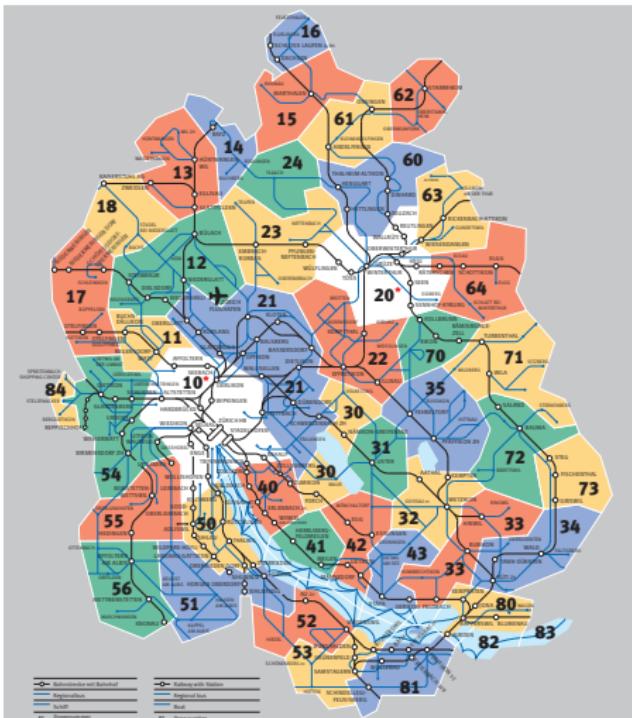
Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

## Spaces with High Dimensionality

## Wrap-Up



## ... More Solutions

### Recent proposals for multi-dimension index structures

- |                                    |                                       |
|------------------------------------|---------------------------------------|
| Quad Tree [Finkel 1974]            | K-D-B-Tree [Robinson 1981]            |
| R-tree [Guttman 1984]              | Grid File [Nievergelt 1984]           |
| R <sup>+</sup> -tree [Sellis 1987] | LSD-tree [Henrich 1989]               |
| R*-tree [Geckmann 1990]            | hB-tree [Lomet 1990]                  |
| Vp-tree [Chiueh 1994]              | TV-tree [Lin 1994]                    |
| UB-tree [Bayer 1996]               | hB- $\nabla$ -tree [Evangelidis 1995] |
| SS-tree [White 1996]               | X-tree [Berchtold 1996]               |
| M-tree [Ciaccia 1996]              | SR-tree [Katayama 1997]               |
| Pyramid [Berchtold 1998]           | Hybrid-tree [Chakrabarti 1999]        |
| DABS-tree [Böhm 1999]              | IQ-tree [Böhm 2000]                   |
| Slim-tree [Faloutsos 2000]         | Landmark file [Böhm 2000]             |
| P-Sphere-tree [Goldstein 2000]     | A-tree [Sakurai 2000]                 |

None of these provides a “fits all” solution.

### Multi-Dimensional Indexing

Torsten Grust



#### B+-trees...

... over composite keys

#### Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

#### k-d Trees

Balanced Construction

#### K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

#### R-Trees

Searching and Inserting

Splitting R-Tree Nodes

#### UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

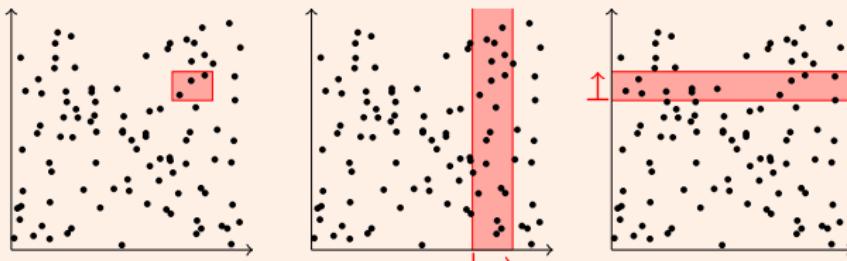
#### Spaces with High Dimensionality

#### Wrap-Up

## Can We Just Use a B<sup>+</sup>-tree?

- Can two B<sup>+</sup>-trees, one over ZIPCODE and over REVENUE, adequately support a two-dimensional range query?

### Querying a rectangle



- Can only scan along **either** index at once, and both of them produce many **false hits**.
- If all you have are these two indices, you can do **index intersection**: perform both scans in separation to obtain the *rids* of candidate tuples. Then compute the intersection between the two *rid* lists (DB2: IXAND).

Multi-Dimensional  
Indexing

Torsten Grust



#### B+-trees...

... over composite keys

#### Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

#### k-d Trees

Balanced Construction

#### K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

#### R-Trees

Searching and Inserting

Splitting R-Tree Nodes

#### UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

#### Spaces with High Dimensionality

#### Wrap-Up

# IBM DB2: Index Intersection

Multi-Dimensional  
Indexing

Torsten Grust



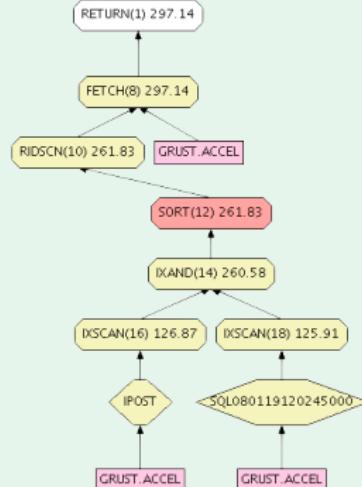
## DB2. IBM DB2 uses index intersection (operator IXAND)

```
1 SELECT *
2 FROM   ACCEL
3 WHERE  PRE BETWEEN 0 AND 10000
4 AND    POST BETWEEN 10000 AND 20000
```

Relevant indexes defined over table ACCEL:

- ① IPOST over column POST
- ② SQL0801192024500 over column PRE (primary key)

## DB2. Plan selected by the query optimizer



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search  
Inserting Data  
Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations  
Splitting a Point Page  
Splitting a Region Page

R-Trees

Searching and Inserting  
Splitting R-Tree Nodes

UB-Trees

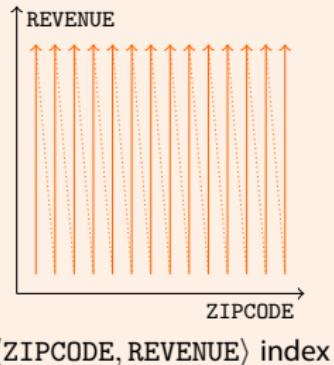
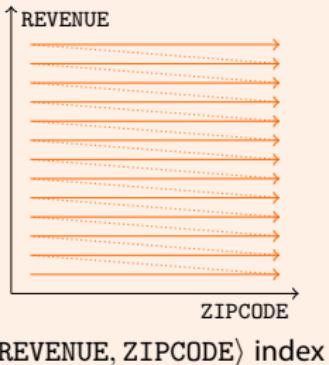
Bit Interleaving / Z-Ordering  
B+-Trees over Z-Codes  
Range Queries

Spaces with High Dimensionality

Wrap-Up

# Can Composite Keys Help?

## Indexes with composite keys



Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## Multi-Dimensional Indices

- B<sup>+</sup>-trees can answer **one-dimensional** queries **only**.<sup>1</sup>
- We would like to have a **multi-dimensional index structure** that
  - is **symmetric** in its dimensions,
  - **clusters** data in a space-aware fashion,
  - is **dynamic** with respect to updates, and
  - provides good support for useful queries.
- We will start with data structures that have been designed for **in-memory** use, then tweak them into **disk-aware** database indices (e.g., organize data in a page-wise fashion).

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

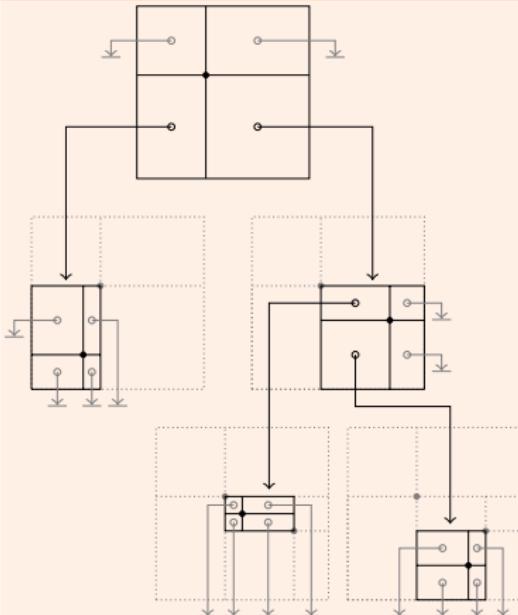
---

<sup>1</sup>Toward the end of this chapter, we will see UB-trees, a nifty trick that uses B<sup>+</sup>-trees to support some multi-dimensional queries.

## "Binary" Search Tree

In  $k$  dimensions, a "**binary tree**" becomes a  $2^k$ -**ary tree**.

### Point quad tree ( $k = 2$ )



- Each data point **partitions** the data space into  $2^k$  **disjoint regions**.
- In each node, a region points to another node (representing a refined partitioning) or to a special **null** value.
- This data structure is a **point quad tree**.

↗ Finkel and Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, vol. 4, 1974.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

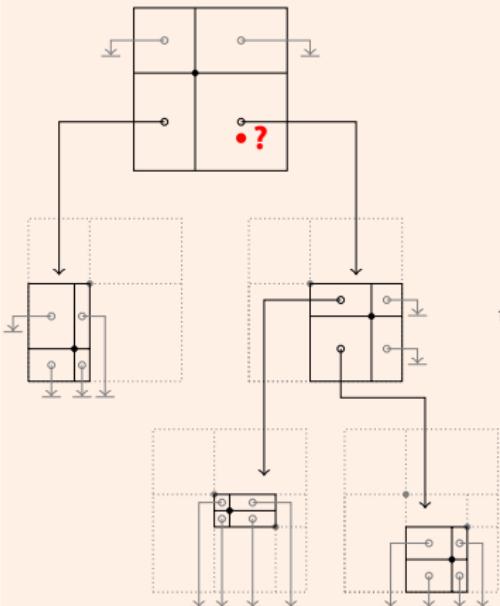
Range Queries

Spaces with High Dimensionality

Wrap-Up

# Searching a Point Quad Tree

## Point quad tree ( $k = 2$ )



## Point quad tree search

```
1 Function: p_search ( $q$ ,  $node$ )
2   if  $q$  matches data point in  $node$  then
3     return data point;
4   else
5      $P \leftarrow$  partition containing  $q$ ;
6     if  $P$  points to null then
7       return not found;
8     else
9        $node' \leftarrow$  node pointed to by  $P$ ;
10      return p_search ( $q$ ,  $node'$ );
```

```
1 Function: pointsearch ( $q$ )
2   return p_search ( $q$ ,  $root$ );
```

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

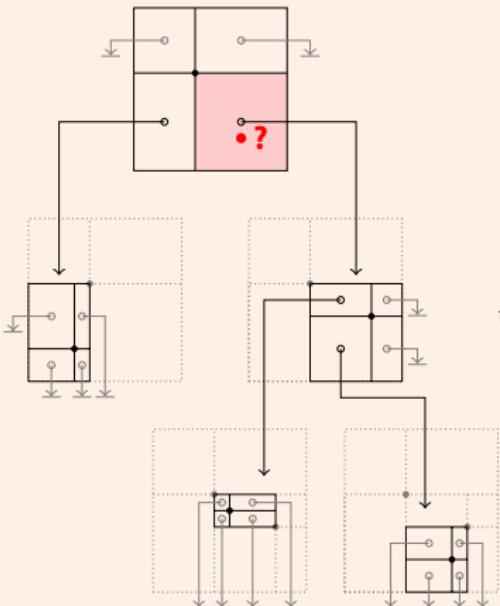
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

# Searching a Point Quad Tree

## Point quad tree ( $k = 2$ )



## Point quad tree search

```
1 Function: p_search ( $q$ ,  $node$ )
2   if  $q$  matches data point in  $node$  then
3     return data point;
4   else
5      $P \leftarrow$  partition containing  $q$ ;
6     if  $P$  points to null then
7       return not found;
8     else
9        $node' \leftarrow$  node pointed to by  $P$ ;
10      return p_search ( $q$ ,  $node'$ );
```

```
1 Function: pointsearch ( $q$ )
2   return p_search ( $q$ ,  $root$ );
```

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

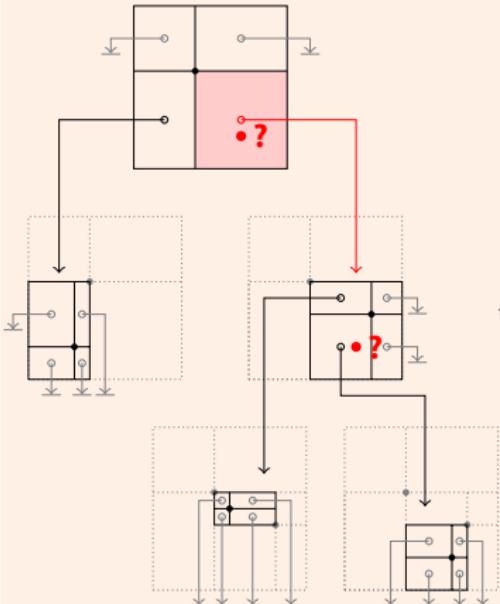
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

# Searching a Point Quad Tree

## Point quad tree ( $k = 2$ )



## Point quad tree search

```
1 Function: p_search ( $q$ ,  $node$ )
2   if  $q$  matches data point in  $node$  then
3     return data point;
4   else
5      $P \leftarrow$  partition containing  $q$ ;
6     if  $P$  points to null then
7       return not found;
8     else
9        $node' \leftarrow$  node pointed to by  $P$ ;
10      return p_search ( $q$ ,  $node'$ );
```

```
1 Function: pointsearch ( $q$ )
2   return p_search ( $q$ ,  $root$ );
```

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

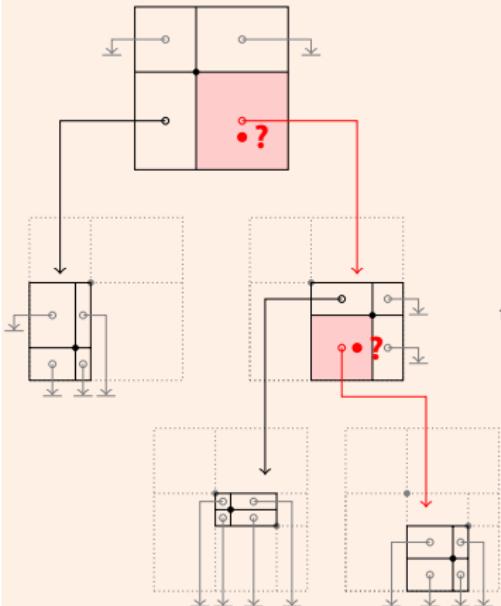
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

# Searching a Point Quad Tree

## Point quad tree ( $k = 2$ )



## Point quad tree search

```
1 Function: p_search( $q, node$ )
2   if  $q$  matches data point in  $node$  then
3     return data point;
4   else
5      $P \leftarrow$  partition containing  $q$ ;
6     if  $P$  points to null then
7       return not found;
8     else
9        $node' \leftarrow$  node pointed to by  $P$ ;
10      return p_search( $q, node'$ );
```

```
1 Function: pointsearch( $q$ )
2   return p_search( $q, root$ );
```

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

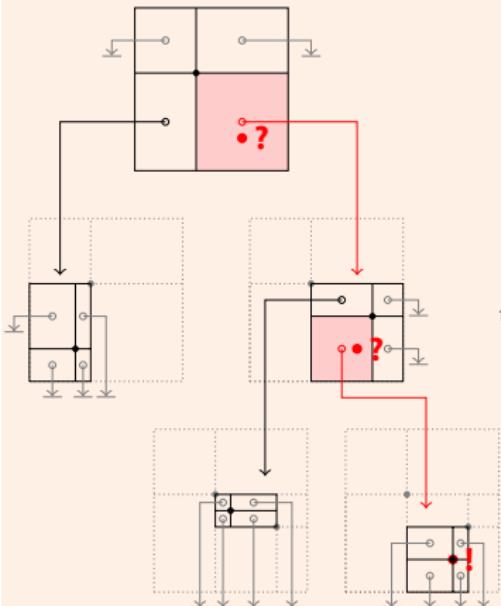
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

# Searching a Point Quad Tree

## Point quad tree ( $k = 2$ )



## Point quad tree search

```
1 Function: p_search ( $q, node$ )
2   if  $q$  matches data point in  $node$  then
3     return data point;
4   else
5      $P \leftarrow$  partition containing  $q$ ;
6     if  $P$  points to null then
7       return not found;
8     else
9        $node' \leftarrow$  node pointed to by  $P$ ;
10      return p_search ( $q, node'$ );
```

```
1 Function: pointsearch ( $q$ )
2   return p_search ( $q, root$ );
```

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## Inserting into a Point Quad Tree

Inserting a point  $q_{\text{new}}$  into a quad tree happens analogously to an insertion into a binary tree:

- ① Traverse the tree just like during a search for  $q_{\text{new}}$  until you encounter a partition  $P$  with a **null** pointer.
- ② Create a **new node**  $n'$  that spans the same area as  $P$  and is partitioned by  $q_{\text{new}}$ , with all partitions pointing to **null**.
- ③ Let  $P$  point to  $n'$ .

Note that this procedure does **not** keep the tree **balanced**.

### Example (Insertions into an empty point quad tree)



Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

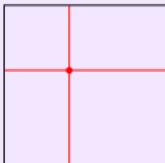
## Inserting into a Point Quad Tree

Inserting a point  $q_{\text{new}}$  into a quad tree happens analogously to an insertion into a binary tree:

- ① Traverse the tree just like during a search for  $q_{\text{new}}$  until you encounter a partition  $P$  with a **null** pointer.
- ② Create a **new node**  $n'$  that spans the same area as  $P$  and is partitioned by  $q_{\text{new}}$ , with all partitions pointing to **null**.
- ③ Let  $P$  point to  $n'$ .

Note that this procedure does **not** keep the tree **balanced**.

### Example (Insertions into an empty point quad tree)



Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

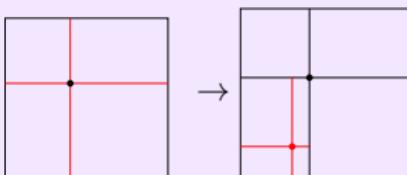
## Inserting into a Point Quad Tree

Inserting a point  $q_{\text{new}}$  into a quad tree happens analogously to an insertion into a binary tree:

- ① Traverse the tree just like during a search for  $q_{\text{new}}$  until you encounter a partition  $P$  with a **null** pointer.
- ② Create a **new node**  $n'$  that spans the same area as  $P$  and is partitioned by  $q_{\text{new}}$ , with all partitions pointing to **null**.
- ③ Let  $P$  point to  $n'$ .

Note that this procedure does **not** keep the tree **balanced**.

### Example (Insertions into an empty point quad tree)



Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

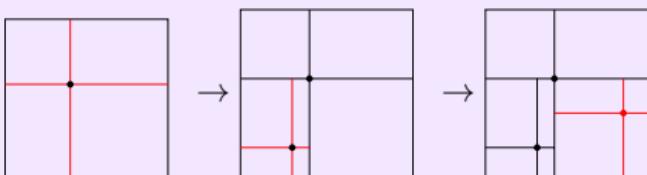
## Inserting into a Point Quad Tree

Inserting a point  $q_{\text{new}}$  into a quad tree happens analogously to an insertion into a binary tree:

- ① Traverse the tree just like during a search for  $q_{\text{new}}$  until you encounter a partition  $P$  with a **null** pointer.
- ② Create a **new node**  $n'$  that spans the same area as  $P$  and is partitioned by  $q_{\text{new}}$ , with all partitions pointing to **null**.
- ③ Let  $P$  point to  $n'$ .

Note that this procedure does **not** keep the tree **balanced**.

### Example (Insertions into an empty point quad tree)



Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

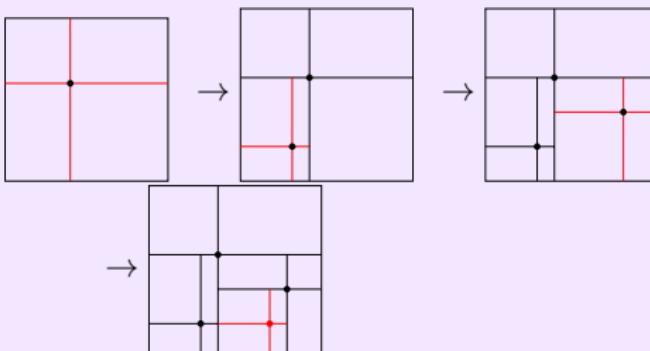
## Inserting into a Point Quad Tree

Inserting a point  $q_{\text{new}}$  into a quad tree happens analogously to an insertion into a binary tree:

- ① Traverse the tree just like during a search for  $q_{\text{new}}$  until you encounter a partition  $P$  with a **null** pointer.
- ② Create a **new node**  $n'$  that spans the same area as  $P$  and is partitioned by  $q_{\text{new}}$ , with all partitions pointing to **null**.
- ③ Let  $P$  point to  $n'$ .

Note that this procedure does **not** keep the tree **balanced**.

### Example (Insertions into an empty point quad tree)



Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

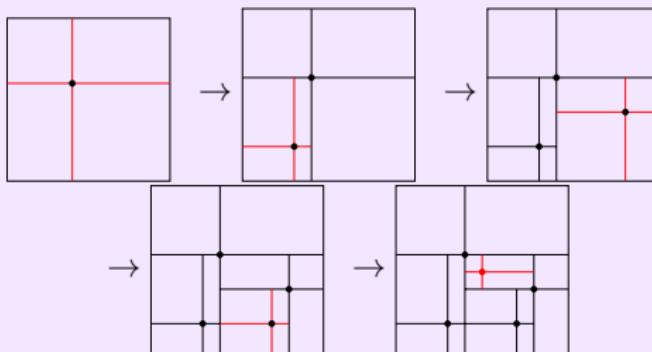
## Inserting into a Point Quad Tree

Inserting a point  $q_{\text{new}}$  into a quad tree happens analogously to an insertion into a binary tree:

- ① Traverse the tree just like during a search for  $q_{\text{new}}$  until you encounter a partition  $P$  with a **null** pointer.
- ② Create a **new node**  $n'$  that spans the same area as  $P$  and is partitioned by  $q_{\text{new}}$ , with all partitions pointing to **null**.
- ③ Let  $P$  point to  $n'$ .

Note that this procedure does **not** keep the tree **balanced**.

### Example (Insertions into an empty point quad tree)



Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## Range Queries

To evaluate a **range query**<sup>2</sup>, we may need to follow **several** children of a quad tree node *node*:

### Point quad tree range search

```
1 Function: r_search (Q, node)
2   if data point in node is in Q then
3     append data point to result ;
4   foreach partition P in node that intersects with Q do
5     node'  $\leftarrow$  node pointed to by P ;
6     r_search (Q, node') ;
```

---

```
1 Function: regionsearch (Q)
2   return r_search (Q, root) ;
```

---

<sup>2</sup>We consider **rectangular** regions only; other shapes may be answered by querying for the **bounding rectangle** and post-processing the output.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## Point Quad Trees—Discussion

### Point quad trees

- ✓ are **symmetric** with respect to all dimensions and
- ✓ can answer **point queries** and **region queries**.

But

- ✗ the shape of a quad tree depends on the **insertion order** of its content, in the worst case **degenerates** into a **linked list**,
- ✗ **null** pointers are **space inefficient** (particularly for large  $k$ ).

In addition,

- they can only store **point data**.

Also remember that quad trees were designed for **main memory** operation.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

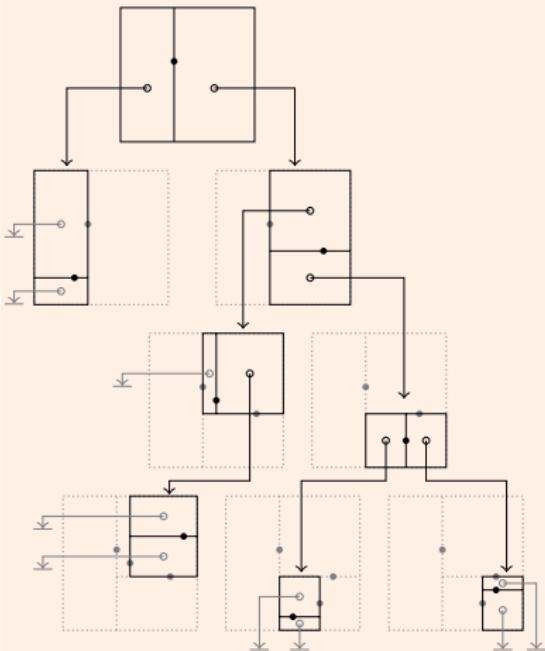
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

# k-d Trees

## Sample k-d tree ( $k = 2$ )



- Index  $k$ -dimensional data, but keep the tree **binary**.
- For each **tree level** / use a different **discriminator dimension**  $d_i$  along which to **partition**.
  - Typically: **round robin**
- This is a  **$k$ -d tree**.
  - ↗ Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Comm. ACM*, vol. 18, no. 9, Sept. 1975.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High Dimensionality

Wrap-Up

## k-d Trees

k-d trees inherit the positive properties of the point quad trees, but improve on **space efficiency**.

For a given point set, we can also construct a **balanced k-d tree**:<sup>3</sup>

### k-d tree construction (Bentley's *OPTIMIZE* algorithm)

```
1 Function: kdTree (pointset, level)
2   if pointset is empty then
3     return null ;
4   else
5     p  $\leftarrow$  median from pointset (along  $d_{level}$ ) ;
6     pointsleft  $\leftarrow \{v \in \text{pointset} \text{ where } v_{d_{level}} < p_{d_{level}}\}$ ;
7     pointsright  $\leftarrow \{v \in \text{pointset} \text{ where } v_{d_{level}} \geq p_{d_{level}}\} \setminus \{p\}$ ;
8     n  $\leftarrow$  new k-d tree node, with data point p ;
9     n.left  $\leftarrow$  kdTree (pointsleft, level + 1) ;
10    n.right  $\leftarrow$  kdTree (pointsright, level + 1) ;
11    return n ;
```

<sup>3</sup> $v_i$ : coordinate *i* of point *v*.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

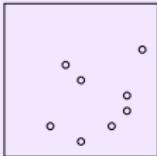
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

# Balanced k-d Tree Construction

## Example (Step-by-step balanced k-d tree construction)



Multi-Dimensional  
Indexing

Torsten Grust



### B+-trees...

... over composite keys

### Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

### k-d Trees

Balanced Construction

### K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

### R-Trees

Searching and Inserting

Splitting R-Tree Nodes

### UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

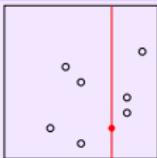
Range Queries

### Spaces with High Dimensionality

### Wrap-Up

# Balanced k-d Tree Construction

## Example (Step-by-step balanced k-d tree construction)



Multi-Dimensional  
Indexing

Torsten Grust



### B+-trees...

... over composite keys

### Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

### k-d Trees

Balanced Construction

### K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

### R-Trees

Searching and Inserting

Splitting R-Tree Nodes

### UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

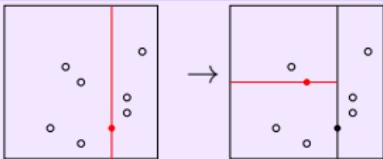
Range Queries

### Spaces with High Dimensionality

### Wrap-Up

# Balanced k-d Tree Construction

## Example (Step-by-step balanced k-d tree construction)



Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

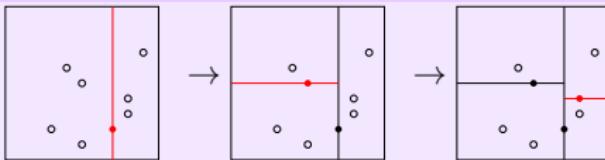
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

# Balanced k-d Tree Construction

## Example (Step-by-step balanced k-d tree construction)



Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

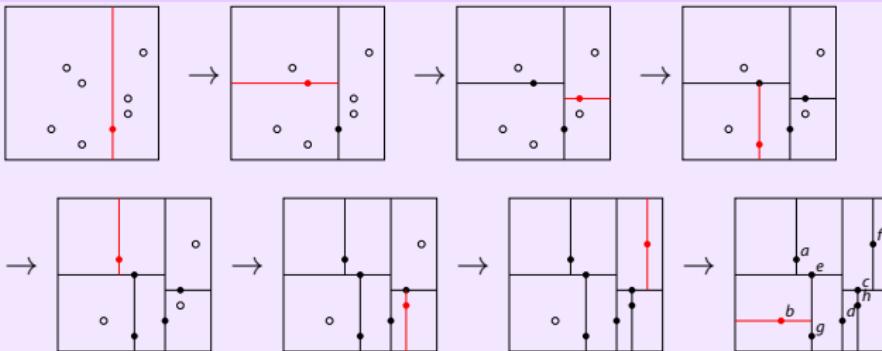
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

# Balanced k-d Tree Construction

## Example (Step-by-step balanced k-d tree construction)



Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

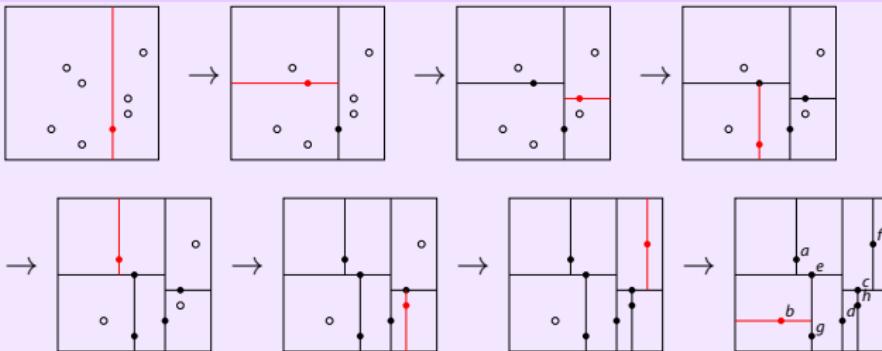
Range Queries

Spaces with High  
Dimensionality

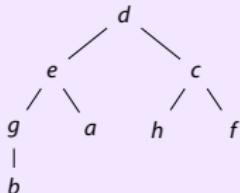
Wrap-Up

# Balanced k-d Tree Construction

## Example (Step-by-step balanced k-d tree construction)



Resulting tree shape:



Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## K-D-B-Trees

k-d trees improve on some of the deficiencies of point quad trees:

- ✓ We can **balance** a k-d tree by **re-building** it.  
(For a limited number of points and in-memory processing, this may be sufficient.)
- ✓ We're no longer wasting big amounts of **space**.

It's time to bring k-d trees to the disk. The **K-D-B-Tree**

- uses **pages** as an organizational unit  
(e.g., each node in the K-D-B-tree fills a page) and
- uses a **k-d tree-like layout** to organize each page.

↗ John T. Robinson. The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. *SIGMOD 1981*.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

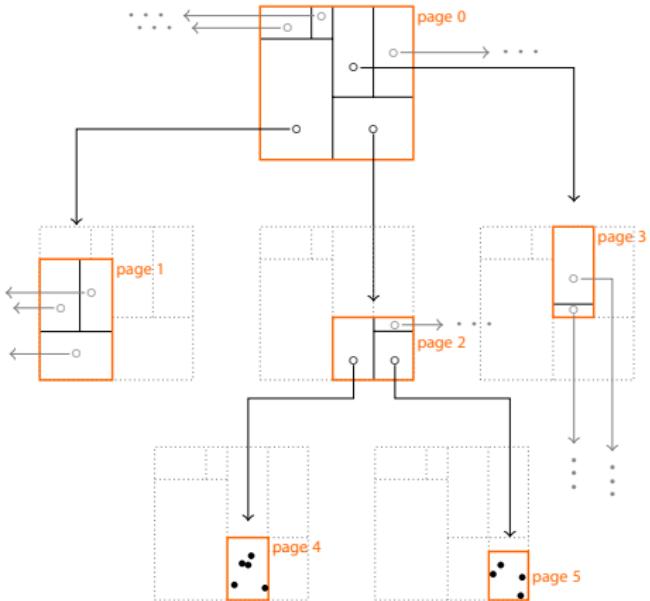
B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## K-D-B-Tree Idea



### Region pages:

- contain entries  $\langle \text{region}, \text{pageID} \rangle$
- no **null** pointers
- form a **balanced** tree
- all regions **disjoint** and **rectangular**

### Point pages:

- contain entries  $\langle \text{point}, \text{rid} \rangle$
- $\leadsto$  B<sup>+</sup>-tree leaf nodes

Multi-Dimensional  
Indexing

Torsten Grust



B-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## K-D-B-Tree Operations

- **Searching** a K-D-B-Tree works straightforwardly:
  - Within each page determine all regions  $R_i$  that contain the query point  $q$  (intersect with the query region  $Q$ ).
  - For each of the  $R_i$ , consult the page it points to and recurse.
  - On point pages, fetch and return the corresponding record for each matching data point  $p_i$ .
- When **inserting** data, we keep the K-D-B-Tree **balanced**, much like we did in the **B<sup>+</sup>-tree**:
  - Simply insert a  $\langle \text{region}, \text{pageID} \rangle$  ( $\langle \text{point}, \text{rid} \rangle$ ) entry into a region page (point page) if there's **sufficient space**.
  - **Otherwise, split** the page.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

# K-D-B-Tree: Splitting a Point Page

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...  
... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## Splitting a point page $p$

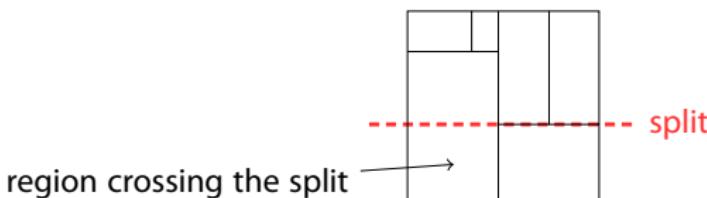
- ① **Choose a dimension**  $i$  and an  $i$ -coordinate  $x_i$  along which to split, such that the split will result in two pages that are not overfull.
- ② **Move** data points  $p$  with  $p_i < x_i$  and  $p_i \geq x_i$  to new pages  $p_{\text{left}}$  and  $p_{\text{right}}$  (respectively).
- ③ Replace  $\langle \text{region}, p \rangle$  in the **parent** of  $p$  with  $\langle \text{left region}, p_{\text{left}} \rangle$   $\langle \text{right region}, p_{\text{right}} \rangle$ .

Step ③ may cause an **overflow** in  $p$ 's parent and, hence, lead to a **split** of a **region page**.

## K-D-B-Tree: Splitting a Region Page

- Splitting a **point page** and moving its data **points** to the resulting pages is straightforward.
- In case of a **region page split**, by contrast, some **regions** may intersect with **both** sides of the split.

Consider, e.g., page 0 on slide 19:



- Such regions need to be **split**, too.
- This can cause a **recursive splitting downward (!) the tree**.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

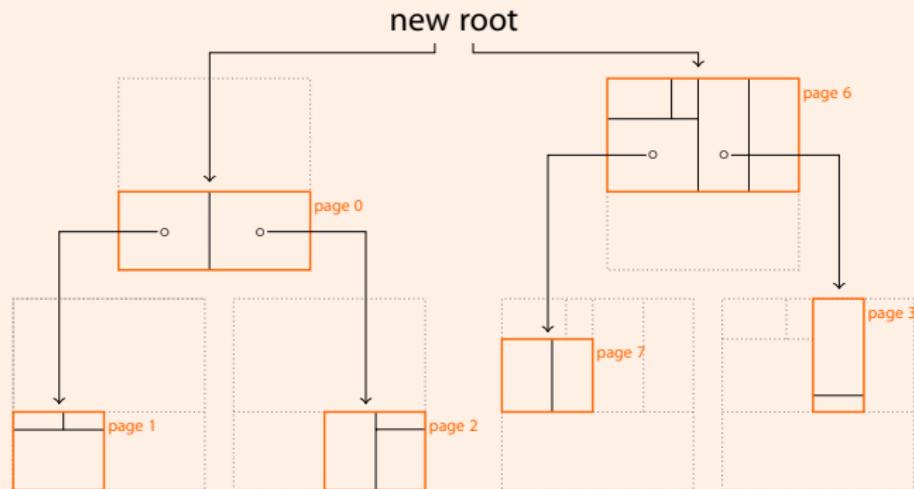
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## Example: Page 0 Split in K-D-B-Tree of slide 19

### Split of region page 0



- Root page 0  $\Rightarrow$  pages 0 and 6 ( $\leadsto$  creation of new root).
- Region page 1  $\Rightarrow$  pages 1 and 7 (point pages not shown).

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## K-D-B-Trees: Discussion

### K-D-B-Trees

- ✓ are **symmetric** with respect to all dimensions,
- ✓ **cluster** data in a space-aware and page-oriented fashion,
- ✓ are **dynamic** with respect to updates, and
- ✓ can answer **point queries** and **region queries**.

However,

- we still don't have support for **region data** and
- K-D-B-Trees (like  $k$ -d trees) will not handle **deletes** dynamically.

This is because we always partitioned the data space such that

- every region is **rectangular** and
- regions never **intersect**.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

$k$ -d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## R-Trees

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

R-trees do not have the disjointness requirement:

- R-tree inner or leaf nodes contain  $\langle \text{region}, \text{pageID} \rangle$  or  $\langle \text{region}, \text{rid} \rangle$  entries (respectively).  
*region* is the **minimum bounding rectangle** that spans all data items reachable by the respective pointer.
- Every node contains between  $d$  and  $2d$  entries ( $\rightsquigarrow$  B<sup>+</sup>-tree).<sup>4</sup>
- **Insertion** and **deletion** algorithms keep an R-tree **balanced** at all times.

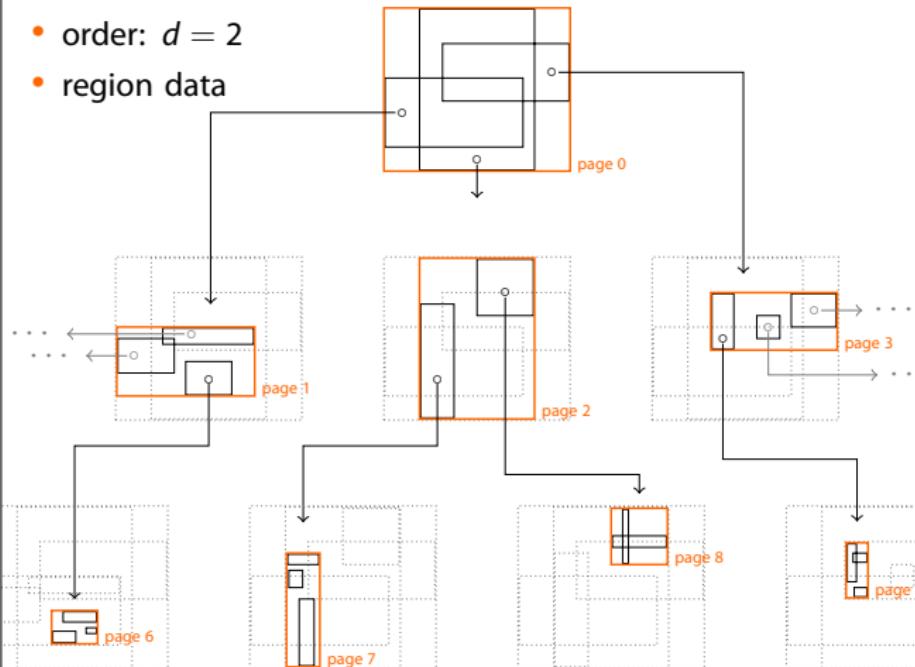
R-trees allow the storage of **point and region data**.

↗ Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD 1984*.

<sup>4</sup>except the root node

# R-Tree: Example

- order:  $d = 2$
- region data



inner nodes

leaf nodes

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## R-Tree: Searching and Inserting

While **searching** an R-tree, we may have to descend into more than one child node for point **and** region queries ( $\nearrow$  range search in point quad trees, slide 13).

### Inserting into an R-tree (cf. B<sup>+</sup>-tree insertion)

- ① **Choose** a leaf node  $n$  to insert the new entry.
  - Try to minimize the necessary region enlargement(s).
- ② If  $n$  is **full**, **split** it (resulting in  $n$  and  $n'$ ) and distribute old and new entries evenly across  $n$  and  $n'$ .
  - Splits may propagate bottom-up and eventually reach the root ( $\nearrow$  B<sup>+</sup>-tree).
- ③ After the insertion, some regions in the ancestor nodes of  $n$  may need to be **adjusted** to cover the new entry.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

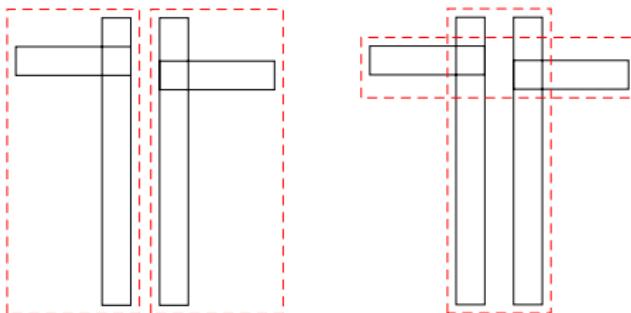
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## Splitting an R-Tree Node

To **split** an R-tree node, we generally have more than one alternative:



**Heuristic: Minimize the totally covered area.** But:

- **Exhaustive** search for the best split is infeasible.
- Guttman proposes two ways to **approximate** the search.
- Follow-up papers (e.g., the R\*-tree) aim at improving the quality of node splits.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

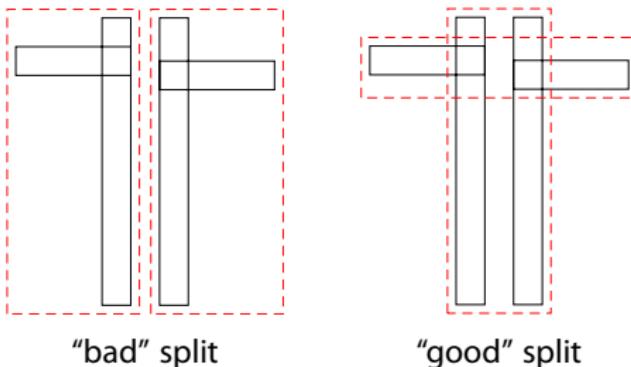
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## Splitting an R-Tree Node

To **split** an R-tree node, we generally have more than one alternative:



**Heuristic: Minimize the totally covered area.** But:

- **Exhaustive** search for the best split is infeasible.
- Guttman proposes two ways to **approximate** the search.
- Follow-up papers (e.g., the R\*-tree) aim at improving the quality of node splits.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## R-Tree: Deletes

All R-tree invariants (see 25) are maintained during **deletions**.

- ① If an R-tree node  $n$  **underflows** (i.e., less than  $d$  entries are left after a deletion), the whole node is **deleted**.
- ② Then, all entries that existed in  $n$  are **re-inserted** into the R-tree.

Note that Step ① may lead to a recursive deletion of  $n$ 's parent.

- Deletion, therefore, is a rather **expensive** task in an R-tree.

## Spatial indexing in mainstream database implementations

- Indexing in commercial database systems is typically based on **R-trees**.
- Yet, only few systems implement them out of the box (e.g., PostgreSQL). Most require the licensing/use of specific extensions.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

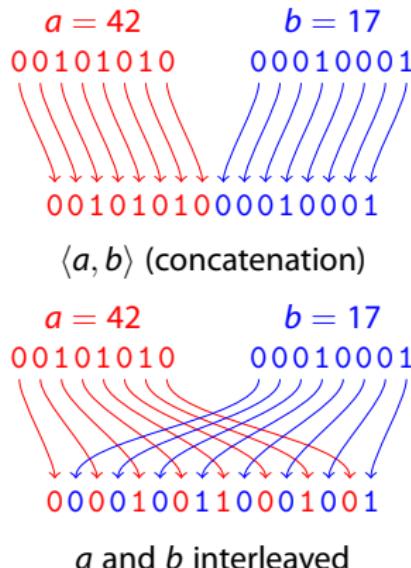
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## Bit Interleaving

- We saw earlier that a B<sup>+</sup>-tree over **concatenated** fields  $\langle a, b \rangle$  does not help our case, because of the **asymmetry** between the role of  $a$  and  $b$  in the index key.
- What happens if we **interleave** the bits of  $a$  and  $b$  (hence, make the B<sup>+</sup>-tree “more symmetric”)?



Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

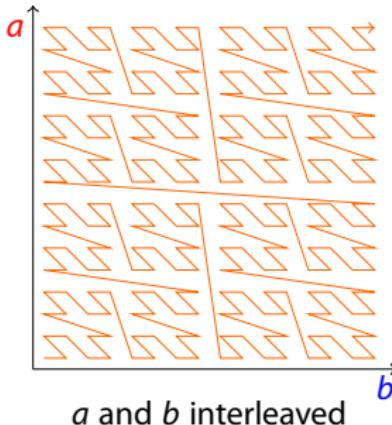
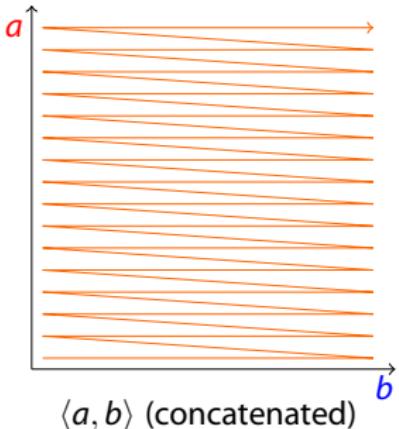
B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## Z-Ordering



- Both approaches **linearize** all coordinates in the value space according to some **order**.  
↗ see also slide 8
- Bit interleaving leads to what is called the **Z-order**.
- The Z-order (largely) preserves spatial **clustering**.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

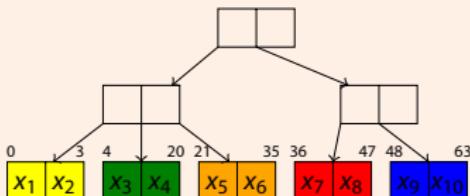
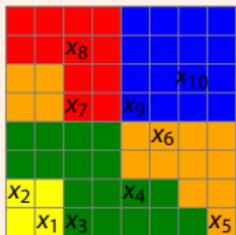
Spaces with High  
Dimensionality

Wrap-Up

## UB-Tree: B<sup>+</sup>-trees Over Z-Codes

- Use a **B<sup>+</sup>-tree** to index Z-codes of multi-dimensional data.
- Each leaf in the B<sup>+</sup>-tree describes an **interval** in the **Z-space**.
- Each interval in the Z-space describes a **region** in the multi-dimensional data space:

### UB-Tree: B<sup>+</sup>-tree over Z-codes



- To retrieve all data points in a query region  $Q$ , try to touch only those leave pages that **intersect** with  $Q$ .

Multi-Dimensional  
Indexing

Torsten Grust



B<sup>+</sup>-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## UB-Tree: Range Queries

After each page processed, perform an **index re-scan** (↗) to fetch the next leaf page that intersects with  $Q$ .



Example by Volker Markl and Rudolf Bayer taken from <http://matrikel.in.tum.de/>

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

### UB-tree range search (Function `ub_range( $Q$ )`)

```
1  cur ← z( $Q_{\text{bottom}, \text{left}}$ );
2  while true do
3      // search B+-tree page containing  $cur$  (↗ slide 0.0)
4      page ← search( $cur$ );
5      foreach data point  $p$  on  $page$  do
6          if  $p$  is in  $Q$  then
7              append  $p$  to result;
8
9      if region in  $page$  reaches beyond  $Q_{\text{top}, \text{right}}$  then
10         break;
11
12     // compute next Z-address using  $Q$  and data on current
13     // page
14     cur ← get_next_z_address( $Q$ ,  $page$ );
```

## UB-Trees: Discussion

- Routine `get_next_z_address()` (return next Z-address lying in the query region) is non-trivial and depends on the shape of the Z-region.
  - UB-trees are **fully dynamic**, a property inherited from the underlying B<sup>+</sup>-trees.
  - The use of other **space-filling curves** to linearize the data space is discussed in the literature, too. *E.g., Hilbert curves.*
- UB-trees have been commercialized in the Transbase® database system.

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...

... over composite keys

Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

R-Trees

Searching and Inserting

Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

Spaces with High  
Dimensionality

Wrap-Up

## Spaces with High Dimensionality

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...  
... over composite keys

Point Quad Trees

Point (Equality) Search  
Inserting Data  
Region Queries

k-d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations  
Splitting a Point Page  
Splitting a Region Page

R-Trees

Searching and Inserting  
Splitting R-Tree Nodes

UB-Trees

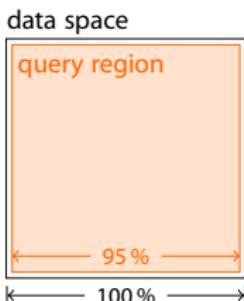
Bit Interleaving / Z-Ordering  
B+-Trees over Z-Codes  
Range Queries

Spaces with High  
Dimensionality

Wrap-Up

For large  $k$ , all techniques that have been discussed become **ineffective**:

- For example, for  $k = 100$ , we get  $2^{100} \approx 10^{30}$  partitions per node in a **point quad tree**. Even with billions of data points, **almost all** of these are empty.
- Consider a **really big** search region, cube-sized covering 95 % of the range along **each** dimension:



For  $k = 100$ , the probability of a point being in this region is still only  $0.95^{100} \approx 0.59\%$ .

- We experience the **curse of dimensionality** here.

## Page Selectivity for $k$ -NN Search

Multi-Dimensional  
Indexing

Torsten Grust



B+-trees...  
... over composite keys

Point Quad Trees

Point (Equality) Search  
Inserting Data  
Region Queries

$k$ -d Trees

Balanced Construction

K-D-B-Trees

K-D-B-Tree Operations  
Splitting a Point Page  
Splitting a Region Page

R-Trees

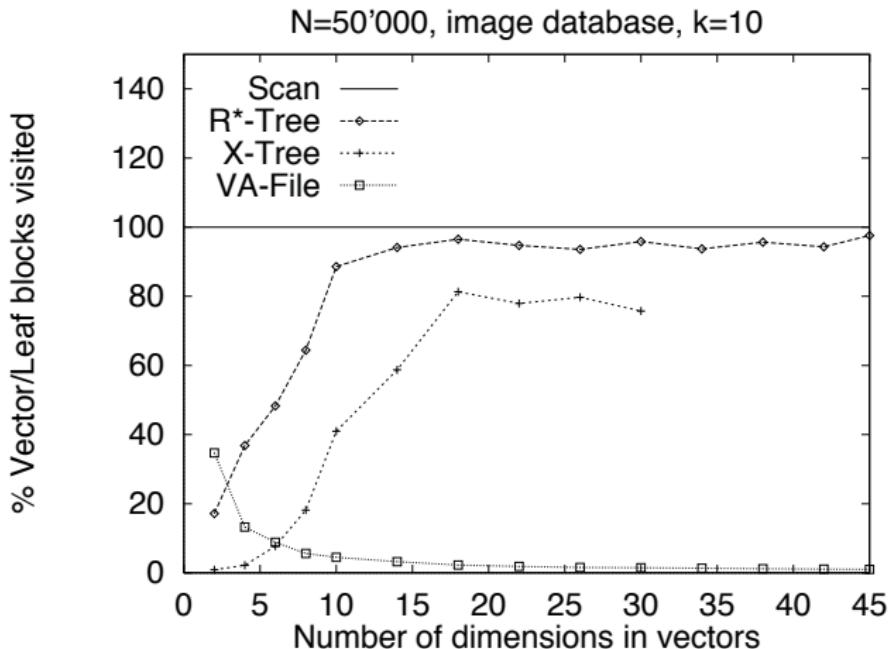
Searching and Inserting  
Splitting R-Tree Nodes

UB-Trees

Bit Interleaving / Z-Ordering  
B+-Trees over Z-Codes  
Range Queries

Spaces with High  
Dimensionality

Wrap-Up



Data: Stephen Bloch. What's Wrong with High-Dimensionality Search. VLDB 2008.

## Wrap-Up

### Point Quad Tree

$k$ -dimensional analogy to binary trees; main memory only.

### $k$ -d Tree, K-D-B-Tree

$k$ -d tree: partition space one dimension at a time

(round-robin); K-D-B-Tree:  $B^+$ -tree-like organization with pages as nodes, nodes use a  $k$ -d-like structure internally.

### R-Tree

regions within a node may overlap; fully dynamic; for point and region data.

### UB-Tree

use space-filling curve (Z-order) to linearize  $k$ -dimensional data, then use  $B^+$ -tree.

### Curse Of Dimensionality

most indexing structures become ineffective for large  $k$ ; fall back to sequential scanning and approximation/compression.

Multi-Dimensional  
Indexing

Torsten Grust



#### B-trees...

... over composite keys

#### Point Quad Trees

Point (Equality) Search

Inserting Data

Region Queries

#### $k$ -d Trees

Balanced Construction

#### K-D-B-Trees

K-D-B-Tree Operations

Splitting a Point Page

Splitting a Region Page

#### R-Trees

Searching and Inserting

Splitting R-Tree Nodes

#### UB-Trees

Bit Interleaving / Z-Ordering

B+-Trees over Z-Codes

Range Queries

#### Spaces with High Dimensionality

#### Wrap-Up

# Chapter 6

## Hash-Based Indexing

### Efficient Support for Equality Search

*Architecture and Implementation of Database Systems*  
Summer 2014

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Reloading)

Running Example

Procedures



# Hash-Based Indexing

- We now turn to a different family of index structures: **hash indexes**.
- Hash indexes are “unbeatable” when it comes to support for **equality selections**:

## Equality selection

```
1 SELECT *
2 FROM   R
3 WHERE  A = k
```

- Further, other query operations internally generate a flood of equality tests (e.g., nested-loop join).  
(Non-)presence of hash index support can make a real difference in such scenarios.

## Hash-Based Indexing

Torsten Grust



## Hash-Based Indexing

### Static Hashing

Hash Functions

### Extendible Hashing

Search

Insertion

Procedures

### Linear Hashing

Insertion (Split, Reloading)

Running Example

Procedures

## Hashing vs. B<sup>+</sup>-trees

- Hash indexes provide **no support for range queries**, however (hash indexes are also known as **scatter storage**).
- In a B<sup>+</sup>-tree-world, to locate a record with key  $k$  means to **compare  $k$  with 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) to find the location of the associated record.
- We will now briefly look into **static hashing** to illustrate the basics.
  - Static hashing does *not* handle updates well (much like ISAM).
  - Later, we introduce **extendible hashing** and **linear hashing** which refine the hashing principle and adapt well to record insertions and deletions.

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Reloading)

Running Example

Procedures

## Static Hashing

- To build a **static hash index** on attribute A:

### Build static hash index on column 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 the 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 is of SQL type INTEGER.

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

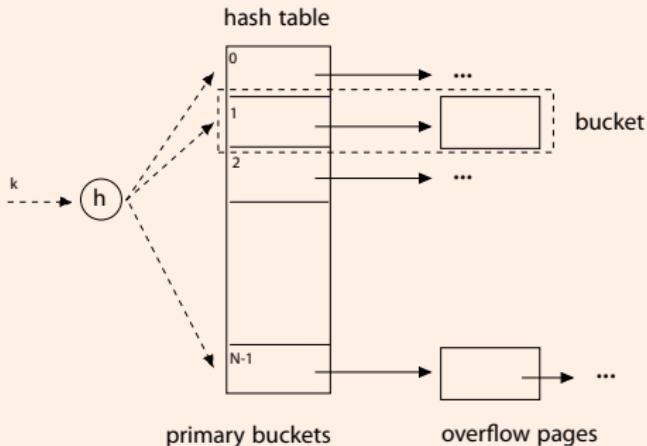
Insertion (Split, Reloading)

Running Example

Procedures

# Static Hashing

## Static hash table



- A primary bucket and its associated chain of overflow pages is referred to as a **bucket** (█████ above).
- Each bucket contains **index entries**  $k*$  (implemented using any of the variants **A**, **B**, **C**, see slide 2.22).

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

## Static Hashing

- To perform  $\text{hsearch}(k)$  (or  $\text{hinsert}(k)$ / $\text{hdelete}(k)$ ) for a record with key  $A = k$ :

### Static hashing scheme

- 1 Apply hash function  $h$  to the key value, i.e., compute  $h(k)$ .**
- 2 Access the primary bucket page with number  $h(k)$ .**
- 3 Search (insert/delete) subject record on this page or, if required, access the overflow chain of bucket  $h(k)$ .**

- If the hashing scheme works well and overflow chain access is avoidable,
  - $\text{hsearch}(k)$  requires a **single I/O operation**,
  - $\text{hinsert}(k)$ / $\text{hdelete}(k)$  require **two I/O operations**.

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relhashing)

Running Example

Procedures

## Static Hashing: Collisions and Overflow Chains

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Reloading)

Running Example

Procedures

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

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

(otherwise we would need as many primary bucket pages as different key values in the data file).

- At the same time, we want  $h$  to **scatter** the key attribute domain **evenly** across  $[0, \dots, N - 1]$  to avoid the development of long overflow chains for few buckets. This makes the hash tables' I/O behavior non-uniform and unpredictable for a query optimizer.
- Such "good" hash functions are hard to discover, unfortunately.

## The Birthday Paradox (Need for Overflow Chain Management)

### Example (The birthday paradox)

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

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

**Check:** Compute the probability that  $n$  people *all have different birthdays*:

```
1 Function: different_birthday ( $n$ )
2   if  $n = 1$  then
3     return 1;
4   else
5     return different_birthday ( $n - 1$ )  $\times$   $\frac{365 - (n - 1)}{365}$  ;
probability that  $n - 1$  persons
have different birthdays                                probability that  $n$ th person
                                                    has birthday different from
                                                    first  $n - 1$  persons
```

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Reloading)

Running Example

Procedures

## Hash Functions

- It is impossible to generate truly random hash values from the non-random key values found in actual table. Can we define hash functions that scatter even better than a random function?

### Hash function

- By division. Simply define

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

This guarantees the range of  $h(k)$  to be  $[0, \dots, N - 1]$ .

**Note:** Choosing  $N = 2^d$  for some  $d$  effectively considers the least  $d$  bits of  $k$  only. **Prime numbers** work best for  $N$ .

- By multiplication. Extract the fractional part of  $Z \cdot k$  (for a specific  $Z^1$ ) and multiply by arbitrary hash table size  $N$ :

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

---

<sup>1</sup>The (inverse) **golden ratio**  $Z = 2/(\sqrt{5}+1) \approx 0.6180339887$  is a good choice.  
See D.E.Knuth, "Sorting and Searching."

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Reloading)

Running Example

Procedures

## Static Hashing and Dynamic Files

- For a static hashing scheme:
  - If the underlying **data file grows**, the development of overflow chains spoils the otherwise predictable behavior hash 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—a waste of page space.
- As in the ISAM case, however, static hashing has advantages when it comes to concurrent access.
- We may periodically **rehash** the data file to restore the ideal situation (20 % free space, no overflow chains).  
⇒ Expensive and the index cannot be used while rehashing is in progress.

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Reloading)

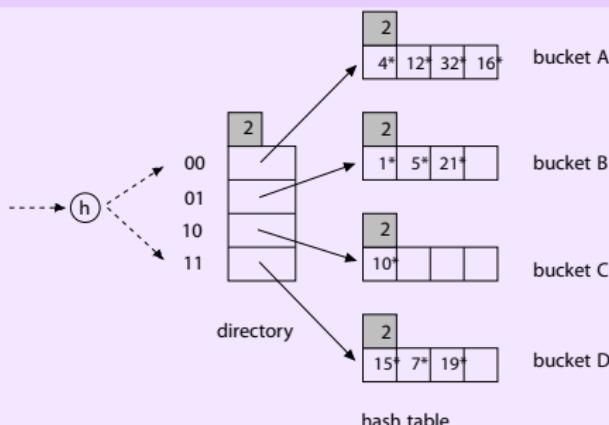
Running Example

Procedures

## Extendible Hashing

- Extendible Hashing can adapt to growing (or shrinking) data files.
- To keep track of the actual primary buckets that are part of the current hash table, we hash via an **in-memory bucket directory**:

Example (Extendible hash table setup; ignore the [2] fields for now<sup>2</sup>)



<sup>2</sup>**Note:** This figure depicts the entries as  $h(k)*$ , not  $k*$ .

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Reloading)

Running Example

Procedures

## Extendible Hashing: Search

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relhashing)

Running Example

Procedures

### 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.

### Example (Search for a record)

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 wanted record.

# Extendible Hashing: Global and Local Depth

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

## Global and local depth annotations

- **Global depth** ( $n$  at hash directory):

*Use the last  $n$  bits of  $h(k)$  to lookup a bucket pointer in the directory (the directory size is  $2^n$ ).*

# Extendible Hashing: Global and Local Depth

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

## Global and local depth annotations

- **Global depth** ( $n$  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** ( $d$  at individual buckets):

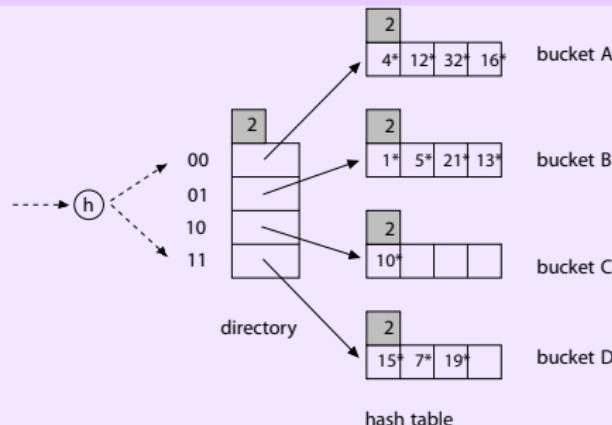
*The hash values  $h(k)$  of all entries in this bucket agree on their last  $d$  bits.*

## Extendible Hashing: Insert

Insert record with key  $k$

- ① Apply  $h$ , i.e., compute  $h(k)$ .
- ② Use the last  $n$  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 ...?)

Example (Insert record with  $h(k) = 13 = 1101_2$ )



Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Reloading)

Running Example

Procedures

## Extendible Hashing: Insert, Bucket Split

Example (Insert record with  $h(k) = 20 = 101\underline{00}_2$ )

Insertion of a record with  $h(k) = 20 = 101\underline{00}_2$  leads to **overflow in primary bucket A**. Initiate a **bucket split** for A.

- 1 **Split** bucket A (creating a new bucket A2) and use bit position  $d + 1$  to redistribute the entries:

$$\begin{array}{ll} 4 & 100_2 \\ 12 & 11\underline{00}_2 \\ 32 & 100\underline{000}_2 \\ 16 & 10\underline{000}_2 \\ 20 & 101\underline{00}_2 \end{array}$$



**Note:** We now need **3** bits to discriminate between the old bucket A and the new split bucket A2.

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

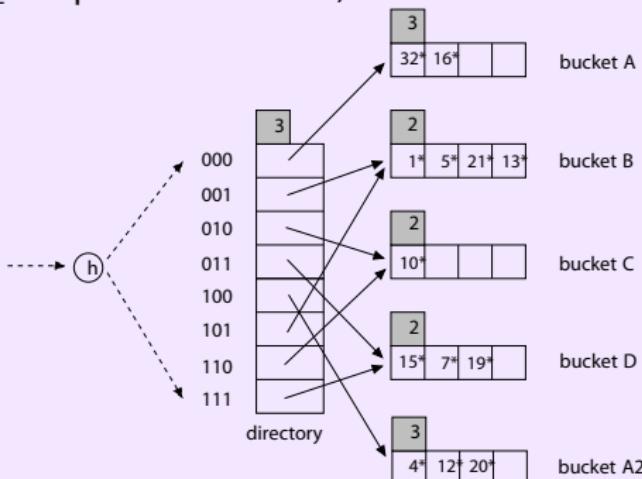
Running Example

Procedures

## Extendible Hashing: Insert, Directory Doubling

Example (Insert record with  $h(k) = 20 = 10100_2$ )

- 2 In the present case, we need to **double the directory** by simply copying its original pages (we now use  $2 + 1 = 3$  bits to lookup a bucket pointer).
- 3 Let bucket pointer for  $100_2$  point to A2 (the directory pointer for  $000_2$  still points to bucket A):



Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

## Extendible Hashing: Insert

If we split a bucket with local depth  $d < n$  (global depth), directory doubling is *not* necessary:

### Example (Insert record with $h(k) = 9 = \underline{1001}_2$ )

- Insert record with key  $k$  such that  $h(k) = 9 = \underline{1001}_2$ .
- The associated bucket B is split, creating a new bucket B2. Entries are redistributed. New local depth of B and B2 is  $\boxed{3}$  and thus does *not* exceed the global depth of  $\boxed{3}$ .
- ⇒ Modifying the directory's bucket pointer for  $\underline{101}_2$  is sufficient (see following slide).

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

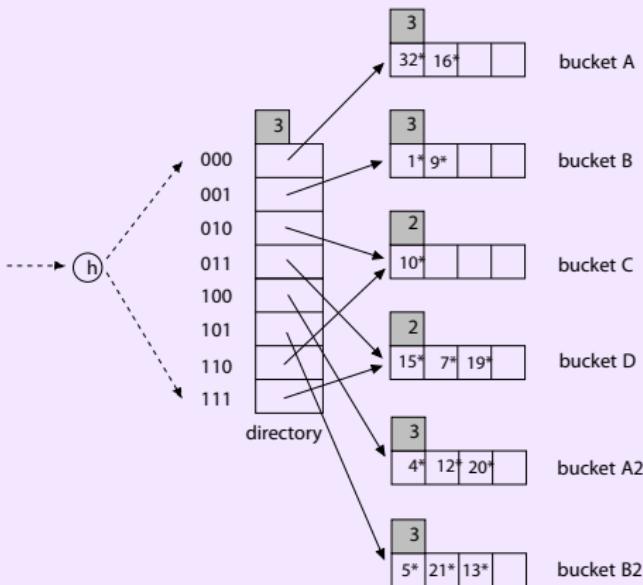
Insertion (Split, Relooking)

Running Example

Procedures

## Extendible Hashing: Insert

Example (After insertion of record with  $h(k) = 9 = 1001_2$ )



Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

## Extendible Hashing: Search Procedure

- The following `hsearch(·)` and `hinsert(·)` procedures operate over an in-memory array representation of the bucket directory  $bucket[0, \dots, 2^{\boxed{n}} - 1]$ .

### Extendible Hashing: Search

```
1 Function: hsearch( $k$ )
2  $n \leftarrow \boxed{n}$ ;                                /* global depth */
3  $b \leftarrow h(k) \& (2^n - 1)$ ;    /* mask all but the low  $n$  bits */
4 return  $bucket[b]$ ;
```

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relabelling)

Running Example

Procedures

## Extendible Hashing: Insert Procedure

Hash-Based Indexing

Torsten Grust



### Extendible Hashing: Insertion

```
1 Function: hinsert( $k*$ )
2  $n \leftarrow \boxed{n}$ ;                                /* global depth */
3  $b \leftarrow \text{hsearch}(k)$ ;
4 if  $b$  has capacity then
5   Place  $k*$  in bucket  $b$ ;
6   return;
7   /* overflow in bucket  $b$ , need to split           */
8    $d \leftarrow \boxed{d}_b$ ;          /* local depth of hash bucket  $b$  */
9   Create a new empty bucket  $b2$ ;
   /* redistribute entries of  $b$  including  $k*$            */
   :
   :
```

Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

## Extendible Hashing: Insert Procedure (continued)

### Extendible Hashing: Insertion (cont'd)

```
1   :
2   /* redistribute entries of  $b$  including  $k$  */          */
3   foreach  $k'$  * in bucket  $b$  do
4       if  $h(k') \& 2^d \neq 0$  then
5           | Move  $k'$  * to bucket  $b2$  ;
6
7   /* new local depths for buckets  $b$  and  $b2$  */          */
8    $d_b \leftarrow d + 1$  ;
9    $d_{b2} \leftarrow d + 1$  ;
10  if  $n < d + 1$  then
11      /* we need to double the directory */                  */
12      Allocate  $2^n$  new directory entries  $bucket[2^n, \dots, 2^{n+1} - 1]$  ;
13      Copy  $bucket[0, \dots, 2^n - 1]$  into  $bucket[2^n, \dots, 2^{n+1} - 1]$  ;
14       $n \leftarrow n + 1$  ;
15      /* update the bucket directory to point to  $b2$  */    */
16       $bucket[(h(k) \& (2^n - 1)) \mid 2^n] \leftarrow addr(b2)$ 
```

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relabeling)

Running Example

Procedures

## Extendible Hashing: Overflow Chains? / Delete

### Overflow chains?

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

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

## Extendible Hashing: Overflow Chains? / Delete

### 🔗 Overflow chains?

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

If considering  $d + 1$  bits does *not* lead to satisfying record redistribution in procedure `hinsert(k)` (skewed data, hash collisions).

- Deleting an entry  $k_*$  from a bucket may leave its bucket completely (or almost) empty.
- Extendible hashing then tries to **merge** the empty bucket and its associated partner bucket.

### 🔗 Extendible hashing: deletion

When is local depth decreased? When is global depth decreased?  
(Try to work out the details on your own.)

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Reloading)

Running Example

Procedures

## Linear Hashing

- Linear hashing can, just like extendible hashing, 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, however, may perform badly if the key distribution in the data file is *skewed*.
- We will now investigate linear hashing in detail and come back to the points above as we go along.
- The core idea behind linear hashing is to use an **ordered family of hash functions**,  $h_0, h_1, h_2, \dots$   
(traditionally the subscript is called the hash function's *level*).

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relabelling)

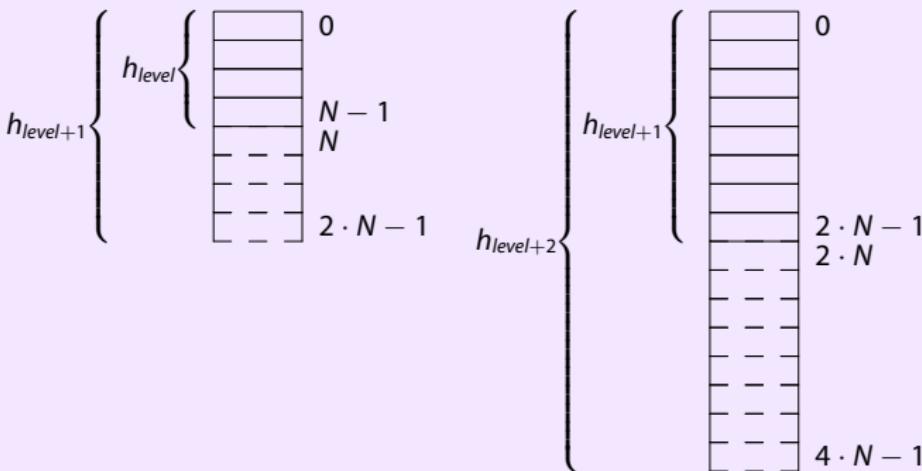
Running Example

Procedures

## Linear Hashing: Hash Function Family

- We design the family so that the **range of  $h_{level+1}$  is twice as large as the range of  $h_{level}$**  (for  $level = 0, 1, 2, \dots$ ).

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



Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

## Linear Hashing: 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:

### Hash function family

$$h_{level}(k) = h(k) \bmod (2^{level} \cdot N) \quad (level = 0, 1, 2, \dots)$$

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

## Linear Hashing: Basic Scheme

### Basic linear hashing scheme

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

we **split the bucket** at hash table position  $next$   
(in general, this is **not the bucket which triggered  
the split!**)

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relabelling)

Running Example

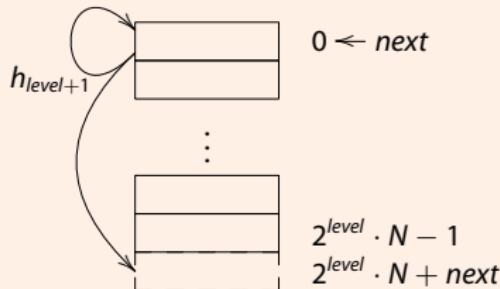
Procedures



# Linear Hashing: Bucket Split

## Linear hashing: bucket split

- ① **Allocate a new bucket, append it to the hash table** (its position will be  $2^{level} \cdot N + next$ ).
- ② **Redistribute** the entries in bucket *next* by **rehashing** them via  $h_{level+1}$  (some entries will remain in bucket *next*, some go to bucket  $2^{level} \cdot N + next$ ). For  $next = 0$ :



- ③ **Increment** *next* by 1.

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

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Rehashing)

Running Example

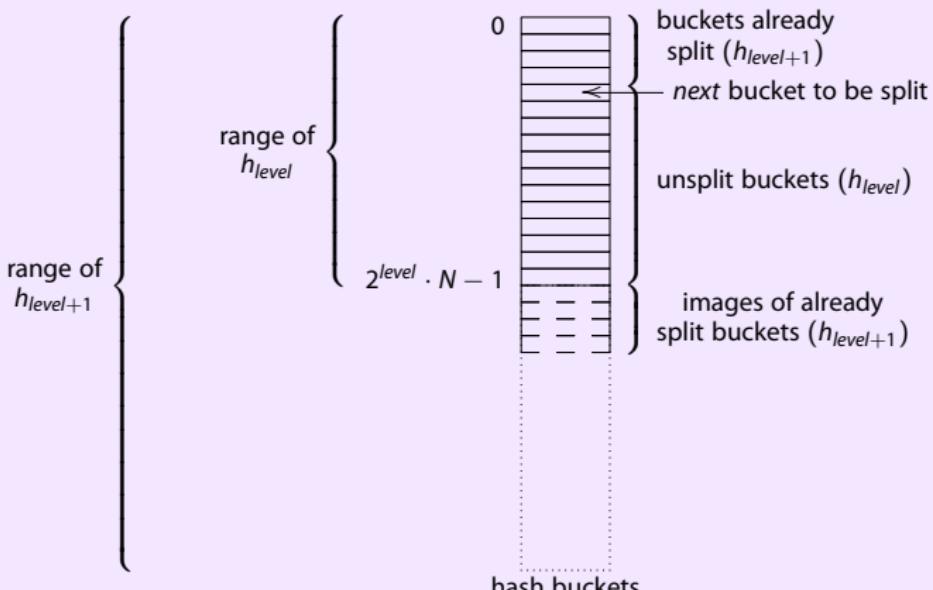
Procedures

## Linear Hashing: Rehashing

Searches need to take current *next* position into account

$$h_{level}(k) \quad \begin{cases} < \text{next} : \text{we hit an already split bucket, rehash} \\ \geq \text{next} : \text{we hit a yet unsplit bucket, bucket found} \end{cases}$$

### Example (Current state of linear hashing scheme)



Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search  
Insertion  
Procedures

Linear Hashing

Insertion (Split, Relooking)  
Running Example  
Procedures

## Linear Hashing: Split Rounds

📎 When *next* is incremented beyond 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 last current hash table position, i.e.

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

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

## Linear Hashing: Split Rounds

### When $next$ is incremented beyond 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 last current hash table position, i.e.

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

### Answer:

- If  $next > 2^{level} \cdot N - 1$ , **all buckets** in the current hash table are hashed via function  $h_{level+1}$ .

⇒ Proceed in a **round-robin fashion**:

If  $next > 2^{level} \cdot N - 1$ , then

- ① increment  $level$  by 1,
- ②  $next \leftarrow 0$  (start splitting from hash table top again).

- In general, an overflowing bucket is *not* split immediately, but—due to round-robin splitting—no later than in the following round.

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relabelling)

Running Example

Procedures

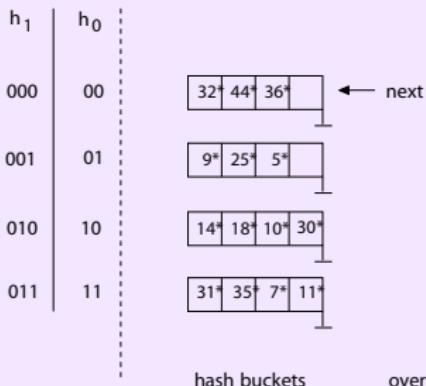
## Linear Hashing: Running Example

### Linear hash table setup:

- Bucket capacity of 4 entries, initial hash table size  $N = 4$ .
- Split criterion: allocation of a page in an overflow chain.

#### Example (Linear hash table, $h_{\text{level}}(k)^*$ shown)

level = 0



Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

# Linear Hashing: Running Example

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

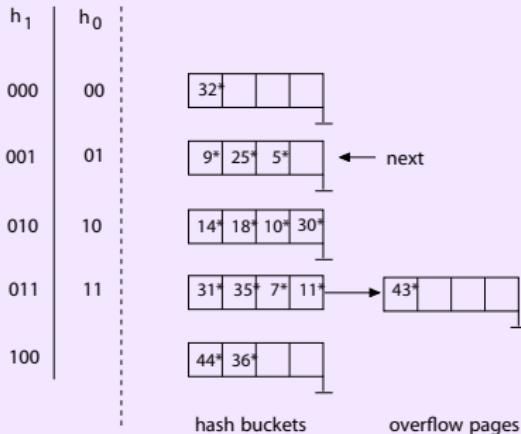
Insertion (Split, Relooking)

Running Example

Procedures

Example (Insert record with key  $k$  such that  $h_0(k) = 43 = 101011_2$ )

level = 0



# Linear Hashing: Running Example

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

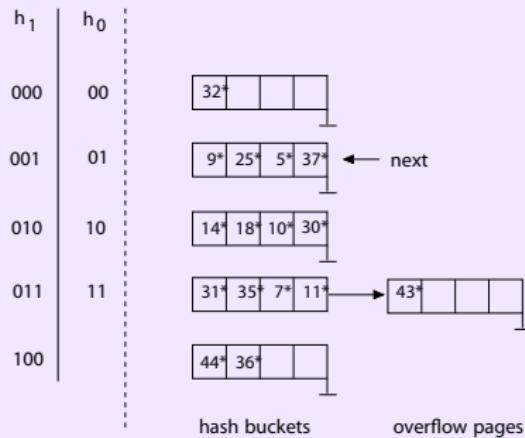
Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

level = 0



# Linear Hashing: Running Example

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

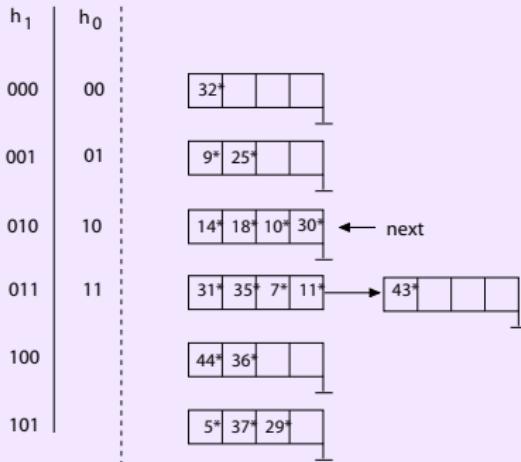
Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

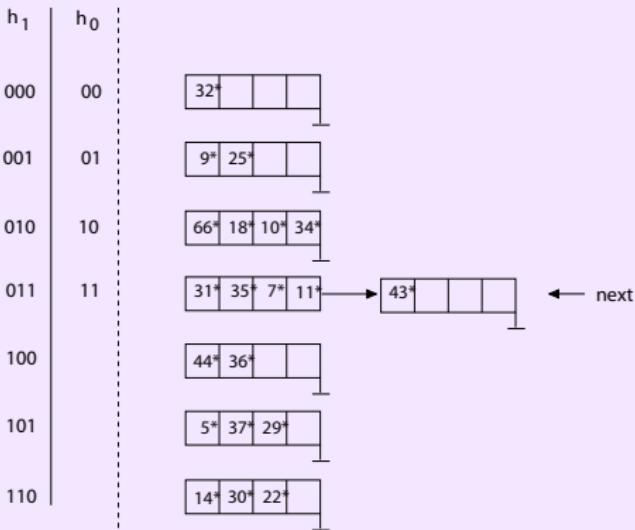
level = 0



# Linear Hashing: Running Example

Example (Insert three records with key  $k$  such that  
 $h_0(k) = 22 = 10110_2 / 66 = 1000010_2 / 34 = 100010_2$ )

level = 0



Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

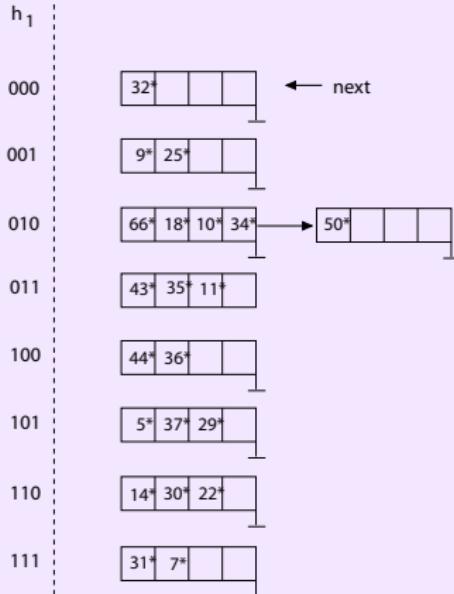
Running Example

Procedures

## Linear Hashing: Running Example

Example (Insert record with key  $k$  such that  $h_0(k) = 50 = 110010_2$ )

level = 1



Rehashing a bucket requires rehashing its overflow chain, too.



Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

## Linear Hashing: Search Procedure

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

### Linear hashing: search

1 **Function:** hsearch( $k$ )

2  $b \leftarrow h_{level}(k)$  ;

3 **if**  $b < next$  **then**

/\*  $b$  has already been split, record for key  $k$  \*/

/\* may be in bucket  $b$  or bucket  $2^{level} \cdot N + b$  \*/

/\*  $\Rightarrow$  rehash \*/

4      $b \leftarrow h_{level+1}(k)$  ;

/\* return address of bucket at position  $b$  \*/

5 **return**  $bucket[b]$  ;

## Linear Hashing: Insert Procedure

### Linear hashing: insert

```
1 Function: hinsert(k*)
2   b ← hlevel(k);
3   if b < next then
4     /* rehash
5     b ← hlevel+1(k);
6   Place k* in bucket[b];
7   if overflow(bucket[b]) then
8     Allocate new page b';
9     /* Grow hash table by one page
10    bucket[2level · N + next] ← addr(b');
11    :
12  */
```

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relooking)

Running Example

Procedures

- Predicate  $overflow(\cdot)$  is a tunable parameter:  
whenever  $overflow(bucket[b])$  returns *true*, trigger a split.

## Linear Hashing: Insert Procedure (continued)

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Reloading)

Running Example

Procedures

```
1   :
2 if overflow( . . . ) then
3   :
4   foreach entry  $k'*$  in  $bucket[next]$  do
5     /* redistribute */  
      Place  $k'*$  in  $bucket[h_{level+1}(k')]$  ;  
6     next  $\leftarrow$  next + 1 ;
7     /* did we split every bucket in the hash? */  
8     if next  $> 2^{level} \cdot N - 1$  then
9       /* hash table size doubled, split from top */  
       level  $\leftarrow$  level + 1 ;
10      next  $\leftarrow$  0 ;
11
12 return;
```

## Linear Hashing: Delete Procedure (Sketch)

- Linear hashing deletion essentially behaves as the “inverse” of  $\text{hinsert}(\cdot)$ :

### Linear hashing: delete (sketch)

```
1 Function: hdelete( $k$ )
2    $b \leftarrow h_{level}(k)$  ;
3   :
4   Remove  $k*$  from  $bucket[b]$  ;
5   if  $\text{empty}(bucket[b])$  then
6     Move entries from page  $bucket[2^{level} \cdot N + next - 1]$ 
7       to page  $bucket[next - 1]$  ;
8      $next \leftarrow next - 1$  ;
9     if  $next < 0$  then
10      /* round-robin scheme for deletion */ *
11       $level \leftarrow level - 1$  ;
12       $next \leftarrow 2^{level} \cdot N - 1$  ;
13
14 return;
```

- Possible: replace  $\text{empty}(\cdot)$  by suitable  $\text{underflow}(\cdot)$  predicate.

Hash-Based Indexing

Torsten Grust



Hash-Based Indexing

Static Hashing

Hash Functions

Extendible Hashing

Search

Insertion

Procedures

Linear Hashing

Insertion (Split, Relabelling)

Running Example

Procedures

# Chapter 7

## External Sorting

Sorting Tables Larger Than Main Memory

*Architecture and Implementation of Database Systems*  
Summer 2014

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort  
External Merge Sort  
Comparisons  
Replacement Sort  
 $B^+$ -trees for Sorting

Torsten Grust  
Wilhelm-Schickard-Institut für Informatik  
Universität Tübingen



# Query Processing

External Sorting

Torsten Grust



## Challenges lurking behind a SQL query

aggregation

```
SELECT C.CUST_ID, C.NAME, SUM(O.TOTAL) AS REVENUE  
  FROM CUSTOMERS AS C, ORDERS AS O  
 WHERE C.ZIPCODE BETWEEN 8000 AND 8999  
   AND C.CUST_ID = O.CUST_ID  
 GROUP BY C.CUST_ID  
 ORDER BY C.CUST_ID, C.NAME
```

selection

join

sorting

A DBMS **query processor** needs to perform a number of tasks

- with **limited memory resources**,
- over **large amounts of data**,
- yet **as fast as possible**.

Query Processing

Sorting

Two-Way Merge Sort

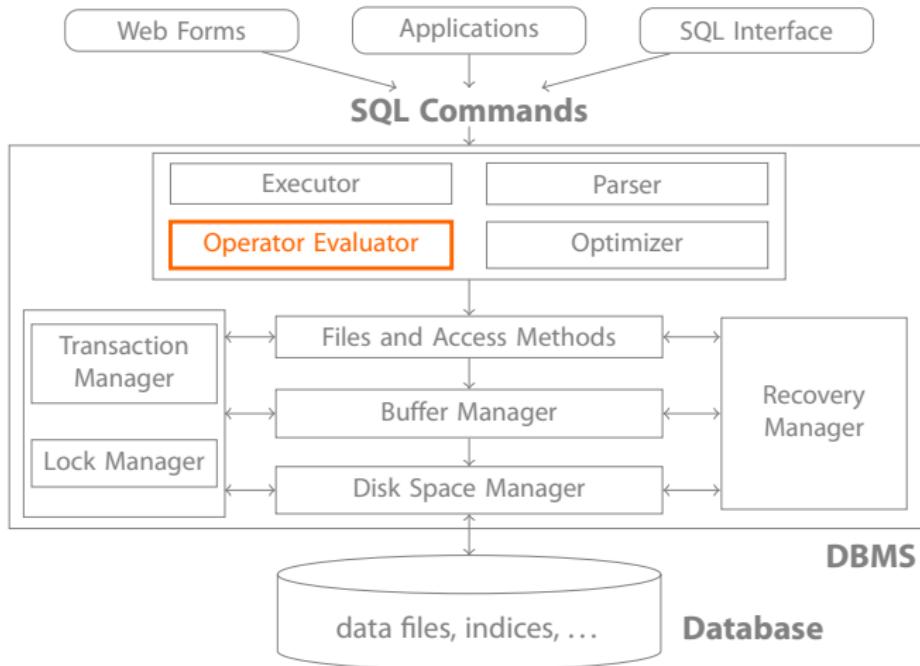
External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

# Query Processing



External Sorting

Torsten Grust



## Query Processing

### Sorting

- Two-Way Merge Sort
- External Merge Sort
- Comparisons
- Replacement Sort
- B<sup>+</sup>-trees for Sorting

# Query Plans and Operators

## Query plans and operators

- DBMS does *not* execute a query as a large monolithic block but rather provides a number of specialized routines, the **query operators**.
  - Operators are “plugged together” to form a network of operators, a **plan**, that is capable of evaluating a given query.
  - Each operator is carefully implemented to perform a specific task well (*i.e.*, time- and space-efficient).
- 
- **Now:** Zoom in on the details of the implementation of one of the most basic and important operators: **sort**.

External Sorting

Torsten Grust



## Query Processing

### Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## Query Processing: Sorting

- **Sorting** stands out as a useful operation, explicit or implicit:

### Explicit sorting via the SQL ORDER BY clause

```
1      SELECT    A,B,C  
2      FROM      R  
3      ORDER BY  A
```

### Implicit sorting, e.g., for duplicate elimination

```
1      SELECT DISTINCT A,B,C  
2      FROM          R
```

### Implicit sorting, e.g., to prepare equi-join

```
1      SELECT  R.A,S.Y  
2      FROM    R,S  
3      WHERE   R.B = S.X
```

- Further:  
Grouping via GROUP BY, B<sup>+</sup>-tree bulk loading, sorted *rid* scans after access to unclustered indexes, ...

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort  
External Merge Sort  
Comparisons  
Replacement Sort  
B<sup>+</sup>-trees for Sorting

## Sorting

External Sorting

Torsten Grust



## Sorting

- A file is **sorted** with respect to **sort key**  $k$  and **ordering**  $\theta$ , if for any two records  $r_{1,2}$  with  $r_1$  preceding  $r_2$  in the file, we have that their corresponding keys are in  $\theta$ -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:  $k = (A, B)$ ,  $\theta = <$ :

$$r_1 < r_2 \quad \Leftrightarrow \quad r_1.A < r_2.A \vee \\ (r_1.A = r_2.A \wedge r_1.B < r_2.B) .$$

Query Processing

### Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## Sorting

- As it is a principal goal not to restrict the file sizes a DBMS can handle, we face a fundamental problem:

*How can we sort a file of records whose **size exceeds the available main memory space** (let alone the available buffer manager space) by far?*

- Approach the task in a two-phase fashion:
  - 1 Sorting a file of arbitrary size is possible even if **three pages** of buffer space is all that is available.
  - 2 Refine this algorithm to make effective use of larger and thus more realistic buffer sizes.
- As we go along, consider a number of further optimizations in order to **reduce the overall number of required page I/O operations.**

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## Two-Way Merge Sort

We start with **two-way merge sort**, which can sort files of arbitrary size with only **three pages** of buffer space.

### Two-way merge sort

Two-way merge sort sorts a file with  $N = 2^k$  pages in multiple **passes**, each of them producing a certain number of sorted sub-files 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  $k$**  leaves only one run left, the sorted overall result.

During each pass, we consult every page in the file. Hence,  $k \cdot N$  page reads and  $k \cdot N$  page writes are required to sort the file.

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## Basic Two-Way Merge Sort Idea

External Sorting

Torsten Grust



**Pass 0**      (**Input:**  $N = 2^k$  unsorted pages; **Output:**  $2^k$  sorted runs)

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 runs; **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:**  $2^{k-n+1}$  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.

⋮

Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

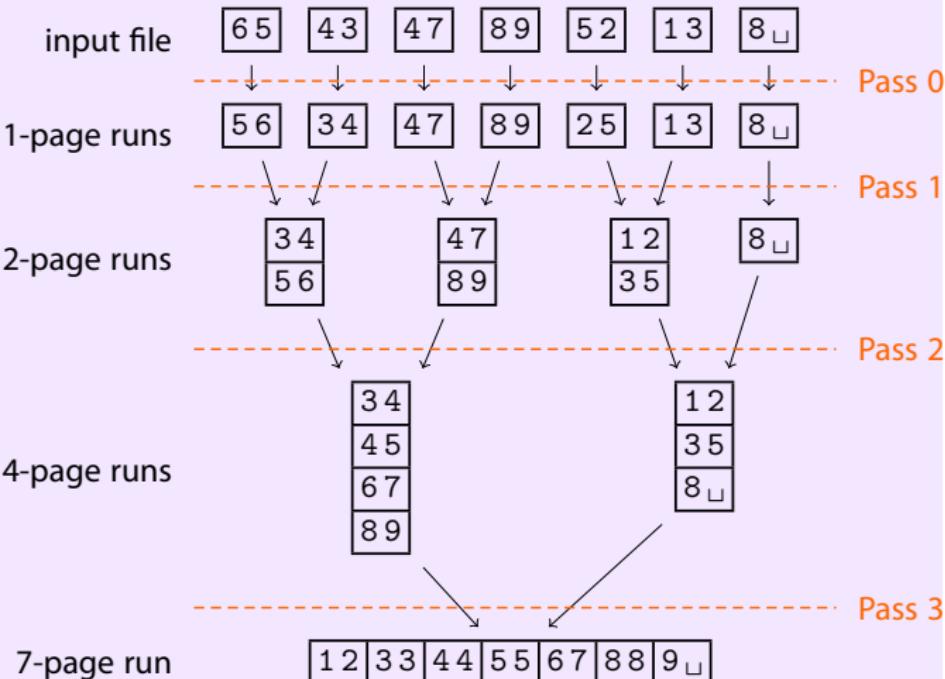
Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## Two-way Merge Sort: Example

Example (7-page file, two records per page, keys  $k$  shown,  $\theta = <$ )



External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## Two-Way Merge Sort: Algorithm

Two-way merge sort,  $N = 2^k$

```
1 Function: two_way_merge_sort (file, N)
  /* Pass 0:  create N sorted single-page runs
   * (in-memory sort) */
  foreach page p in file do
    | read p into memory, sort it, write it out into a new run;
    /* next k passes merge pairs of runs, until only one
     run is left */
    for n in 1...k do
      for r in 0... $2^{k-n} - 1$  do
        | merge runs  $2 \cdot r$  and  $2 \cdot r + 1$  from previous pass into a
        | new run, reading the input runs one page at a time;
        | delete input runs  $2 \cdot r$  and  $2 \cdot r + 1$  ;
    result  $\leftarrow$  last output run;
```

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B+-trees for Sorting

Each merge requires **three buffer frames** (two to read the two input files and one to construct output pages).

## Two-Way Merge Sort: I/O Behavior

- To sort a file of  $N$  pages, in each pass we read  $N$  pages, sort/merge, and write  $N$  pages out again:

$2 \cdot N$  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:

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

📎 How many I/Os does it take to sort an 8 GB file?

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

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B+-trees for Sorting

## External Merge Sort

- So far we have “voluntarily” used only three pages of buffer space.

*How could we **make effective use of a significantly larger buffer page pool** (of, say,  $B$  frames)?*

- Basically, there are two knobs we can turn and tune:
  - ➊ **Reduce the number of initial runs** by using the full buffer space during the in-memory sort.

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

$B^+$ -trees for Sorting

## External Merge Sort

- So far we have “voluntarily” used only three pages of buffer space.

*How could we **make effective use of a significantly larger buffer page pool** (of, say,  $B$  frames)?*

- Basically, there are two knobs we can turn and tune:
  - ➊ **Reduce the number of initial runs** by using the full buffer space during the in-memory sort.
  - ➋ **Reduce the number of passes** by merging more than two runs at a time.

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

$B^+$ -trees for Sorting

## Reducing the Number of Initial Runs

With  $B$  frames available in the buffer pool, **we can read  $B$  pages at a time during Pass 0** and sort them in memory ( $\nearrow$  slide 9):

**Pass 0**      (Input:  $N$  unsorted pages; Output:  $\lceil \frac{N}{B} \rceil$  sorted runs)

1. **Read  $N$  pages,  $B$  pages at a time**
2. **Sort** records 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.

The **number of initial runs** determines the **number of passes** we need to make ( $\nearrow$  slide 12):

⇒ **Total number of I/O operations:**

$$2 \cdot N \cdot (1 + \lceil \log_2 \lceil \frac{N}{B} \rceil \rceil) .$$

📎 **How many I/Os does it take to sort an 8 GB file now?**

Again, assume 8 KB pages. Available buffer space is  $B = 1,000$ .

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## Reducing the Number of Passes

With  $B$  frames available in the buffer pool, we can **merge  $B - 1$  pages at a time** (leaving one frame as a write buffer).

**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 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.

With  $B$  pages of buffer space, we can do a  **$(B - 1)$ -way merge**.

⇒ **Total number of I/O operations:**

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

📎 **How many I/Os does it take to sort an 8 GB file now?**

Again, assume 8 KB pages. Available buffer space is  $B = 1,000$ .

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B+-trees for Sorting

# Reducing the Number of Passes

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

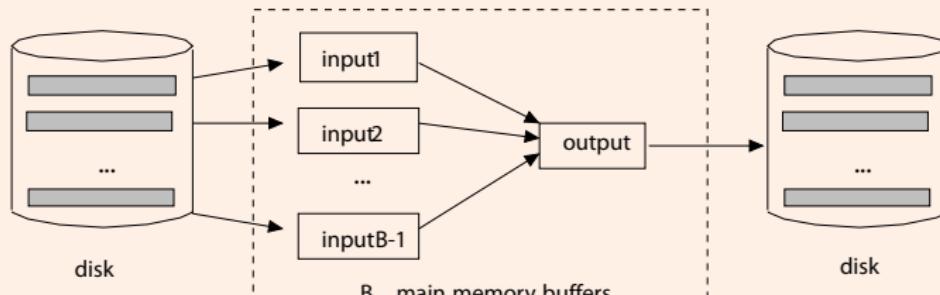
External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

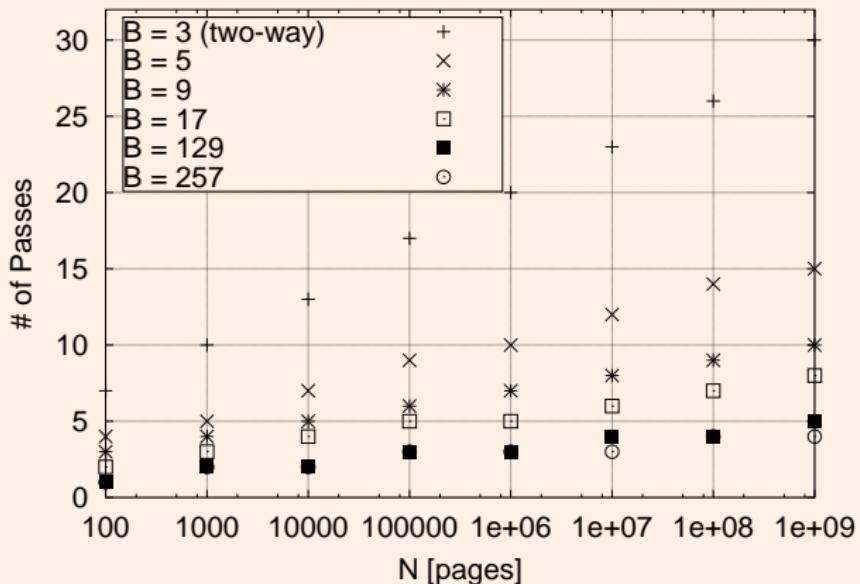
$(B - 1)$ -way merge using a buffer of  $B$  pages



## External Sorting: I/O Behavior

- The I/O savings in comparison to two-way merge sort ( $B = 3$ ) can be substantial:

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



External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## External Sorting: I/O Behavior

- Sorting  $N$  pages with  $B$  buffer frames requires

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

I/O operations.

📎 What is the access pattern of these I/Os?

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## External Sorting: I/O Behavior

- Sorting  $N$  pages with  $B$  buffer frames requires

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

I/O operations.

### ⌚ What is the access pattern of these I/Os?

- In Pass 0, we read chunks of size  $B$  **sequentially**.
- Everything else is **random access** due to the  $B - 1$  way merge.  
*(Which of the  $B - 1$  runs will contribute the next record...?)*

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

$B^+$ -trees for Sorting

## Blocked I/O

We could improve the I/O pattern by reading **blocks** of, say,  $b$  pages at once during the **merge** phases.

- Allocate  $b$  pages for each input (instead of just one).
- **Reduces per-page I/O cost** by a factor of  $\approx b$ .
- The price we pay is a **decreased fan-in** (resulting in an increased number of passes and more I/O operations).
- In practice, main memory sizes are typically large enough to sort files with **just one merge pass**, even with blocked I/O.

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting



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

Assume 1,000 buffer pages of 8 KB each, 8.5 ms average seek time.

## Blocked I/O

We could improve the I/O pattern by reading **blocks** of, say,  $b$  pages at once during the **merge** phases.

- Allocate  $b$  pages for each input (instead of just one).
- **Reduces per-page I/O cost** by a factor of  $\approx b$ .
- The price we pay is a **decreased fan-in** (resulting in an increased number of passes and more I/O operations).
- In practice, main memory sizes are typically large enough to sort files with **just one merge pass**, even with blocked I/O.

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting



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

Assume 1,000 buffer pages of 8 KB each, 8.5 ms average seek time.

- Without blocked I/O:  $\approx 4 \cdot 10^6$  disk seeks (10.5 h) + transfer of  $\approx 6 \cdot 10^6$  disk pages (13.6 min)
- With blocked I/O ( $b = 32$  page blocks):  $\approx 6 \cdot 32,800$  disk seeks (28.1 min) + transfer of  $\approx 8 \cdot 10^6$  disk pages (18.1 min)

## External Merge Sort: CPU Load and Comparisons

- External merge sort reduces the I/O load, but is considerably **CPU intensive**.
- Consider the  **$(B - 1)$ -way merge** during passes  $1, 2, \dots$ : To pick the next record to be moved to the output buffer, we need to perform  $B - 2$  comparisons.

### Example (Comparisons for $B - 1 = 4, \theta = <$ )

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

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## External Merge Sort: CPU Load and Comparisons

- External merge sort reduces the I/O load, but is considerably **CPU intensive**.
- Consider the  **$(B - 1)$ -way merge** during passes  $1, 2, \dots$ : To pick the next record to be moved to the output buffer, we need to perform  $B - 2$  comparisons.

### Example (Comparisons for $B - 1 = 4, \theta = <$ )

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

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## External Merge Sort: CPU Load and Comparisons

- External merge sort reduces the I/O load, but is considerably **CPU intensive**.
- Consider the  **$(B - 1)$ -way merge** during passes  $1, 2, \dots$ : To pick the next record to be moved to the output buffer, we need to perform  $B - 2$  comparisons.

### Example (Comparisons for $B - 1 = 4, \theta = <$ )

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

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## External Merge Sort: CPU Load and Comparisons

- External merge sort reduces the I/O load, but is considerably **CPU intensive**.
- Consider the  **$(B - 1)$ -way merge** during passes  $1, 2, \dots$ : To pick the next record to be moved to the output buffer, we need to perform  $B - 2$  comparisons.

### Example (Comparisons for $B - 1 = 4, \theta = <$ )

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

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## External Merge Sort: CPU Load and Comparisons

- External merge sort reduces the I/O load, but is considerably **CPU intensive**.
- Consider the  **$(B - 1)$ -way merge** during passes  $1, 2, \dots$ : To pick the next record to be moved to the output buffer, we need to perform  $B - 2$  comparisons.

### Example (Comparisons for $B - 1 = 4, \theta = <$ )

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

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## Selection Trees

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

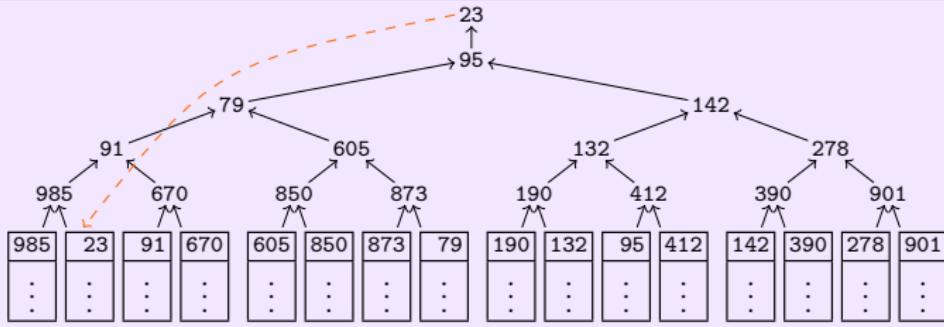
Replacement Sort

B<sup>+</sup>-trees for Sorting

Choosing the next record from  $B - 1$  (or  $B/b - 1$ ) input runs can be quite CPU intensive ( $B - 2$  comparisons).

- Use a **selection tree** to reduce this cost.
- E.g., "tree of losers" (↗ D. Knuth, TAoCP, vol. 3):

### Example (Selection tree, read bottom-up)



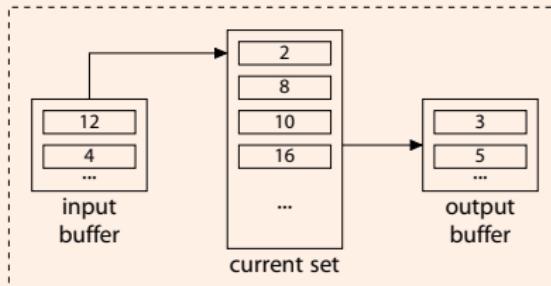
- This cuts the number of comparisons down to  $\log_2 (B - 1)$ .

## Further Reducing the Number of Initial Runs

- **Replacement sort** can help to further cut down the number of initial runs  $[N/B]$ : try to **produce initial runs with more than  $B$  pages**.

### Replacement sort

- Assume a buffer of  $B$  pages. Two pages are dedicated **input** and **output buffers**. The remaining  $B - 2$  pages are called the **current set**:



External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

# Replacement Sort

## 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.<sup>1</sup> Move  $r$  to output buffer. If output buffer is full, append it 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 for writing.
- ⑥ If input file is exhausted, stop. Otherwise go to ③.

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

<sup>1</sup>If output buffer is empty, define  $k_{out} = -\infty$ .

# Replacement Sort

External Sorting

Torsten Grust



Query Processing

Sorting

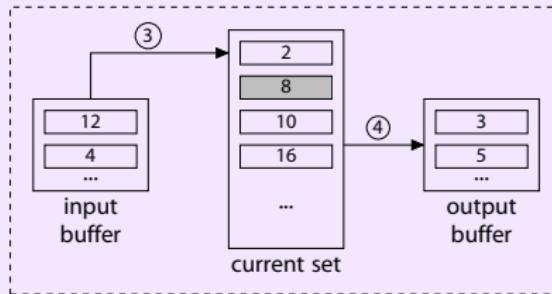
Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting



- The record with key  $k = 2$  remains in the current set and will be written to the subsequent run.

## Replacement Sort

### 🔗 Tracing replacement sort

Assume  $B = 6$ , i.e., a current set size of 4. The input file 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	

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## Replacement Sort

- Step ④ of replacement sort will benefit from techniques like selection tree, esp. if  $B - 2$  is large.
- The replacement sort trace suggests that the length of the initial runs indeed increases. In the example: first run length  $7 \approx$  **twice the size of the current set.**

### 📎 Length of initial runs?

Implement replacement sort to empirically determine initial run length or check the proper analysis ( $\nearrow$  D. Knuth, TAoCP, vol. 3, p. 254).

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

$B^+$ -trees for Sorting

## External Sort: Remarks

- External sorting follows a **divide and conquer** principle.
  - This results in a number of **independent (sub-)tasks**.
  - **Execute tasks in parallel** in a distributed DBMS or exploit multi-core parallelism on modern CPUs.
- To keep the CPU busy while the input buffer is reloaded (or the output buffer appended to the current run), use **double buffering**:

Create **shadow buffers** for the input and output buffers. Let the CPU switch to the “double” input buffer as soon as the input buffer is empty and **asynchronously initiate an I/O operation** to reload the original input buffer.  
Treat the output buffer similarly.

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

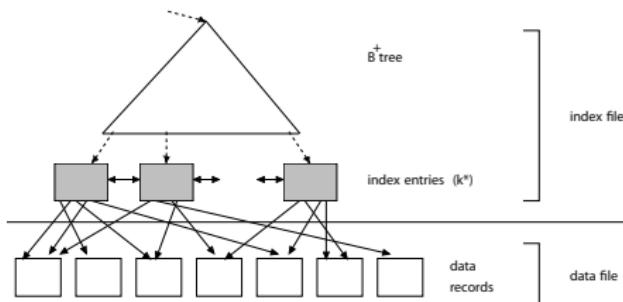
Replacement Sort

B<sup>+</sup>-trees for Sorting

## (Not) Using B<sup>+</sup>-trees for Sorting

- If a B<sup>+</sup>-tree matches a sorting task (i.e., B<sup>+</sup>-tree organized over key  $k$  with ordering  $\theta$ ), we *may* be better off to **access the index and abandon external sorting**.

- 1 If the B<sup>+</sup>-tree is **clustered**, then
  - the data file itself is already  $\theta$ -sorted,  
⇒ simply sequentially read the sequence set (or the pages of the data file).
- 2 If the B<sup>+</sup>-tree is **unclustered**, then
  - in the worst case, we have to initiate one I/O operation per record (not per page)!  
⇒ do not consider the index.



External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

## (Not) Using B<sup>+</sup>-tree for Sorting

External Sorting

Torsten Grust



Query Processing

Sorting

Two-Way Merge Sort

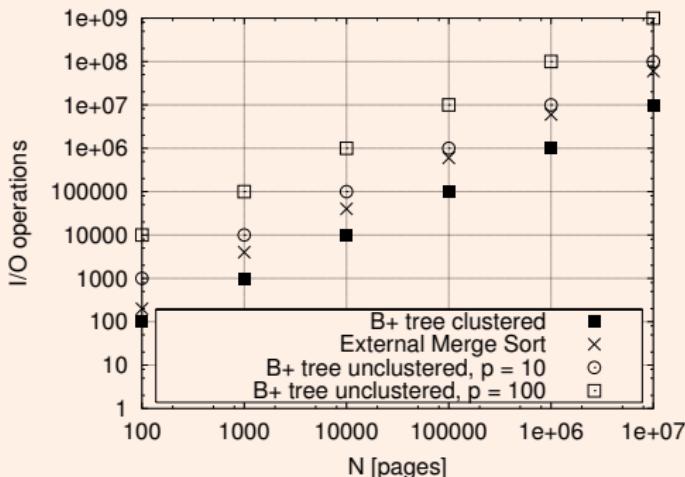
External Merge Sort

Comparisons

Replacement Sort

B<sup>+</sup>-trees for Sorting

Expected sort I/O operations (assume  $B = 257$ )



# Chapter 8

## Evaluation of Relational Operators

### Implementing the Relational Algebra

*Architecture and Implementation of Database Systems*  
Summer 2014

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

Torsten Grust  
Wilhelm-Schickard-Institut für Informatik  
Universität Tübingen



## Relational Query Engines

- In many ways, a DBMS's **query engine** compares to virtual machines (e.g., the Java VM):

Relational Query Engine	Virtual Machine (VM)
Operators of the relational algebra	Primitive VM instructions
Operates over streams of rows	Acts on object representations
Operator network (tree/DAG)	Sequential program (with branches, loops)
Several <b>equivalent variants</b> of an operator	Compact instruction set

### Equivalent operator variants

Instead of a single  $\bowtie$  operator, a typical DBMS query engine features equivalent variants  $\bowtie'$ ,  $\bowtie''$ , ....

What would **equivalent** mean in the context of the relational model?



## Operator Variants

- Specific operator variants may be tailored to exploit **physical properties** of its input or the current system state:
  - ① The **presence or absence of indexes** on the input file(s),
  - ② the **sortedness** of the input file(s),
  - ③ the **size** of the input file(s),
  - ④ the **available space in the buffer pool**,
  - ⑤ the **buffer replacement policy**,
  - ⑥ ...

### Physical operators

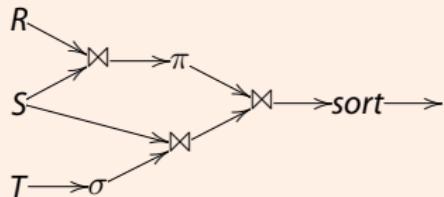
The variants ( $\bowtie'$ ,  $\bowtie''$ ) are thus referred to **physical operators**. They implement the **logical operators** of the relational algebra.

- The **query optimizer** is in charge to perform optimal (or, reasonable) **operator selection** (much like the instruction selection phase in a programming language compiler).



# Operator Selection

## Initial, logical operator network (“plan”)



Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

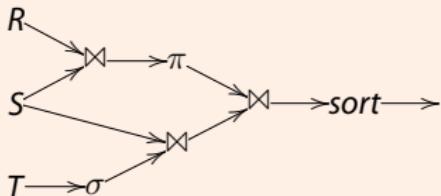
Hash Join

Operator Pipelining

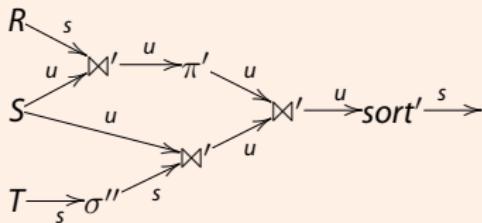
Volcano Iterator Model

# Operator Selection

## Initial, logical operator network (“plan”)



## Physical plan with (un)sortedness annotations ( $u/s$ )



Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

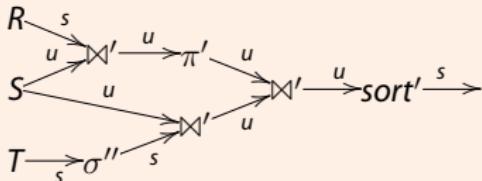
Hash Join

Operator Pipelining

Volcano Iterator Model

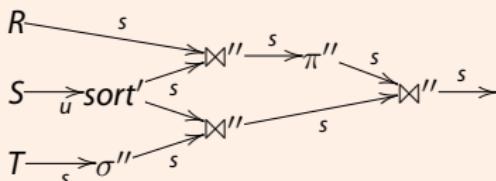
## Plan Rewriting

### Physical plan with (un)sortedness annotations (u/s)



- Rewrite the plan to exploit that the  $\oplus''$  variant of operator  $\oplus$  can benefit from/preserve sortedness of its input(s):

### Rewritten physical plan (preserve equivalence!)



Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )—No Index, Unsorted Data

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

### Selection

```
1 Function:  $\sigma(p, R_{in}, R_{out})$ 
2    $out \leftarrow \text{createFile}(R_{out})$  ;
3    $in \leftarrow \text{openScan}(R_{in})$  ;
4   while ( $r \leftarrow \text{nextRecord}(in)$ )  $\neq \langle EOF \rangle$  do
5     if  $p(r)$  then
6       appendRecord( $out, r$ ) ;
7   closeFile( $out$ ) ;
```

## Selection ( $\sigma$ )—No Index, Unsorted Data

### Remarks:

- Reading the special “record”  $\langle \text{EOF} \rangle$  from a file via `nextRecord()` indicates that all its records have been retrieved (scanned) already.
- This simple procedure does **not require  $r_{in}$  to come with any special physical properties** (the procedure is exclusively defined in terms of heap files).
- In particular, **predicate  $p$  may be arbitrary**.

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )—No Index, Unsorted Data

- We can summarize the characteristics of this implementation of the selection operator as follows:

### Selection ( $\sigma$ )—no index, unsorted data

---

 $\sigma_p(R)$ 

---

**input access**<sup>1</sup> file scan (openScan) of  $R$

**prerequisites** none ( $p$  arbitrary,  $R$  may be a heap file)

**I/O cost**

$$N_R + \underbrace{sel(p)}_{\text{input cost}} \cdot \underbrace{N_R}_{\text{output cost}}$$

---

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

- $N_R$  denotes the **number of pages** in file  $R$ ,  $|R|$  denotes the **number of records**

(if  $p_R$  records fit on one page, we have  $N_R = \lceil |R|/p_R \rceil$ )

---

<sup>1</sup>Also known as **access path** in the literature and text books.

## Aside: Selectivity

- $sel(p)$ , the **selectivity of predicate**  $p$ , is the fraction of records satisfying predicate  $p$ :

$$0 \leqslant sel(p) = \frac{|\sigma_p(R)|}{|R|} \leqslant 1$$

### 📎 Selectivity examples

What can you say about the following selectivities?

- ①  $sel(true)$
- ②  $sel(false)$
- ③  $sel(A = 0)$

### DB2. Estimated selectivities

IBM DB2 reports (estimated) selectivities in the operators details of, e.g., its IXSCAN operator.



## Selection ( $\sigma$ )—Matching Predicates with an Index

- A selection on input file  $R$  can be sped up considerably if an index has been defined and that **index matches predicate  $p$** .
- The matching process depends on  $p$  itself as well as on the index type. If there is no immediate match but  $p$  is **compound**, a sub-expression of  $p$  may still find a **partial match**. Residual predicate evaluation work may then remain.

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )—Matching Predicates with an Index

- A selection on input file  $R$  can be sped up considerably if an index has been defined and that **index matches predicate  $p$** .
- The matching process depends on  $p$  itself as well as on the index type. If there is no immediate match but  $p$  is **compound**, a sub-expression of  $p$  may still find a **partial match**. Residual predicate evaluation work may then remain.

### When does a predicate match a sort key?

Assume  $R$  is tree-indexed on attribute  $A$  in ascending order.  
Which of the selections below can benefit from the index on  $R$ ?

- ①  $\sigma_{A=42} (R)$
- ②  $\sigma_{A < 42} (R)$
- ③  $\sigma_{A > 42 \text{ AND } A < 100} (R)$
- ④  $\sigma_{A > 42 \text{ OR } A > 100} (R)$
- ⑤  $\sigma_{A > 42 \text{ AND } A < 32} (R)$
- ⑥  $\sigma_{A > 42 \text{ AND } B = 10} (R)$
- ⑦  $\sigma_{A > 42 \text{ OR } B = 10} (R)$

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )—B<sup>+</sup>-tree Index

- A **B<sup>+</sup>-tree index** on  $R$  whose key **matches** the selection predicate  $p$  is clearly the superior method to evaluate  $\sigma_p(R)$ :
    - **Descend the B<sup>+</sup>-tree** to retrieve the first index entry to satisfy  $p$ . If the index is **clustered**, access that record on its page in  $R$  and continue to scan inside  $R$ .
    - If the index is **unclustered** and  $sel(p)$  indicates a large number of qualifying records, it pays off to
      - ① read the matching index entries  $k* = \langle k, rid \rangle$  in the sequence set,
      - ② sort those entries on their  $rid$  field,
      - ③ and then access the pages of  $R$  in sorted  $rid$  order.
- Note that lack of clustering is a minor issue if  $sel(p)$  is close to 0.

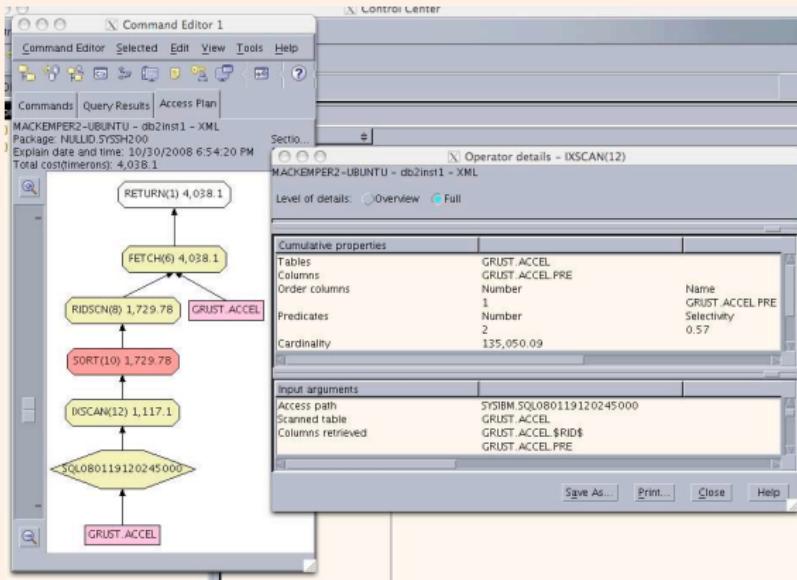
### DB2. Accessing unclustered B<sup>+</sup>-trees

IBM DB2 uses physical operator quadruple  
IXSCAN/SORT/RIDSCN/FETCH to implement the above strategy.

Evaluation of Relational Operators
Torsten Grust
Relational Query Engines
Operator Selection
Selection ( $\sigma$ )
Selectivity
Conjunctive Predicates
Disjunctive Predicates
Projection ( $\pi$ )
Join ( $\bowtie$ )
Nested Loops Join
Block Nested Loops Join
Index Nested Loops Join
Sort-Merge Join
Hash Join
Operator Pipelining
Volcano Iterator Model

# Selection ( $\sigma$ )—B<sup>+</sup>-tree Index

## The IXSCAN/SORT/RIDSCN/FETCH quadruple



- Note: Selectivity of predicate estimated as 57 % (table accel has 235,501 rows).

Evaluation of Relational Operators

Torsten Grust



Relational Query Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )—B<sup>+</sup>-tree Index

### Selection ( $\sigma$ )—clustered B<sup>+</sup>-tree index

$\sigma_p(R)$

**input access** access of B<sup>+</sup>-tree on  $R$ , then sequence set scan

**prerequisites** clustered B<sup>+</sup>-tree on  $R$  with key  $k$ ,  $p$  matches key  $k$

**I/O cost** 
$$\approx \underbrace{3}_{\text{B}^+\text{-tree acc.}} + \underbrace{\text{sel}(p) \cdot N_R}_{\text{sorted scan}} + \underbrace{\text{sel}(p) \cdot N_R}_{\text{output cost}}$$

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Junctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )—Hash Index, Equality Predicate

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

- A selection predicate  $p$  **matches an hash index** only if  $p$  contains a term of the form  $A = c$  ( $c$  constant, assuming the hash index has been built over column  $A$ ).
- We are directly led to the bucket(s) of qualifying records and pay I/O cost only for this direct access<sup>2</sup>. Note that  $sel(p)$  is likely to be close to 0 for many equality predicates.

### Selection ( $\sigma$ )—hash index, equality predicate

---

 $\sigma_p(R)$ 

---

**input access**

hash table on  $R$

**prerequisites**

$r_{in}$  hashed on key  $A$ ,  $p$  has term  $A = c$

**I/O cost**

$$\underbrace{sel(p) \cdot N_R}_{\text{bucket access}} + \underbrace{sel(p) \cdot N_R}_{\text{output cost}}$$

---

<sup>2</sup>Remember that this may include access cost for the pages of an overflow chain hanging off the primary bucket page.

## Selection ( $\sigma$ )—Conjunctive Predicates

- Indeed, selection operations with simple predicates like  $\sigma_{A \theta c}(R)$  are a special case only.
- We somehow need to deal with **complex predicates**, built from **simple comparisons** and the **Boolean connectives** AND and OR.
- Matching a selection predicate with an index can be extended to cover the case where predicate  $p$  has a **conjunctive form**:

$$\underbrace{A_1 \theta_1 c_1}_{\text{conjunct}} \quad \text{AND} \quad A_2 \theta_2 c_2 \quad \text{AND} \quad \dots \quad \text{AND} \quad A_n \theta_n c_n .$$

- Here, each **conjunct** is a simple comparison ( $\theta_i \in \{=, <, >, \leq, \geq\}$ ).

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )—Conjunctive Predicates

- Indeed, selection operations with simple predicates like  $\sigma_{A \theta c}(R)$  are a special case only.
- We somehow need to deal with **complex predicates**, built from **simple comparisons** and the **Boolean connectives** AND and OR.
- Matching a selection predicate with an index can be extended to cover the case where predicate  $p$  has a **conjunctive form**:

$$\underbrace{A_1 \theta_1 c_1}_{\text{conjunct}} \quad \text{AND} \quad A_2 \theta_2 c_2 \quad \text{AND} \quad \dots \quad \text{AND} \quad A_n \theta_n c_n .$$

- Here, each **conjunct** is a simple comparison ( $\theta_i \in \{=, <, >, \leq, \geq\}$ ).
- An index with a multi-attribute key may match the *entire* complex predicate.

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )—Conjunctive Predicates

### 🔗 Matching a multi-attribute hash index

Consider a hash index for the multi-attribute key  $k = (A, B, C)$ , i.e., all three attributes are input to the hash function.

Which conjunctive predicates  $p$  would **match** this type of index?

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )—Conjunctive Predicates

### 🔗 Matching a multi-attribute hash index

Consider a hash index for the multi-attribute key  $k = (A, B, C)$ , i.e., all three attributes are input to the hash function.

Which conjunctive predicates  $p$  would **match** this type of index?

### Conjunctive predicate match rule for hash indexes

A **conjunctive predicate  $p$  matches a (multi-attribute) hash index** with key  $k = (A_1, A_2, \dots, A_n)$ , if  $p$  **covers the key**, i.e.,

$$p \equiv A_1 = c_1 \text{ AND } A_2 = c_2 \text{ AND } \dots \text{ AND } A_n = c_n \text{ AND } \phi .$$

The residual conjunct  $\phi$  is not supported by the index itself and has to be **evaluated after index retrieval**.

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

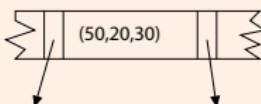
## Selection ( $\sigma$ )—Conjunctive Predicates

### Matching a multi-attribute $B^+$ -tree index

Consider a  $B^+$ -tree index for the multi-attribute key  $k = (A, B, C)$ , i.e., the  $B^+$ -tree nodes are searched/inserted in lexicographic order w.r.t. these three attributes:

$$\begin{aligned} k_1 < k_2 \equiv & A_1 < A_2 \vee \\ & (A_1 = A_2 \wedge B_1 < B_2) \vee \\ & (A_1 = A_2 \wedge B_1 = B_2 \wedge C_1 < C_2) \end{aligned}$$

Excerpt of an inner  $B^+$ -tree node (separator):



Which conjunctive predicates  $p$  would **match** this type of index?



## Selection ( $\sigma$ )—Conjunctive Predicates

Evaluation of  
Relational Operators

Torsten Grust



### Conjunctive predicate match rule for $B^+$ -tree indexes

A **conjunctive predicate  $p$  matches a (multi-attribute)  $B^+$ -tree index** with key  $k = (A_1, A_2, \dots, A_n)$ , if  $p$  is a prefix of the key, i.e.,

$$p \equiv A_1 \theta_1 c_1 \text{ AND } \phi$$

$$p \equiv A_1 \theta_1 c_1 \text{ AND } A_2 \theta_2 c_2 \text{ AND } \phi$$

⋮

$$p \equiv A_1 \theta_1 c_1 \text{ AND } A_2 \theta_2 c_2 \text{ AND } \dots \text{ AND } A_n \theta_n c_n \text{ AND } \phi$$

Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join  
Block Nested Loops Join  
Index Nested Loops Join  
Sort-Merge Join  
Hash Join

Operator Pipelining

Volcano Iterator Model

- **Note:** Whenever a multi-attribute hash index matches a predicate, so does a  $B^+$ -tree over the same key.

## Selection ( $\sigma$ )—Conjunctive Predicates

- If the system finds that a conjunctive predicate does not match a single index, its (smaller) **conjuncts may nevertheless match distinct indexes**.

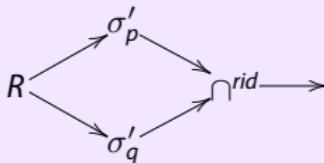
### Example (Partial predicate match)

The conjunctive predicate in  $\sigma_{p \text{ AND } q}(R)$  does not match an index, but both conjuncts  $p, q$  do.

A typical optimizer might thus decide to transform the original query

$$R \longrightarrow \sigma_p \text{ AND } q \longrightarrow$$

into



Here,  $\cap^{rid}$  denotes a **set intersection operator defined by rid equality** (IBM DB2: IXAND).



## Selection ( $\sigma$ )—Conjunctive Predicates

Evaluation of  
Relational Operators

Torsten Grust



### 📎 Selectivity of conjunctive predicates

What can you say about the selectivity of the conjunctive predicate  $p \text{ AND } q$ ?

$$sel(p \text{ AND } q) =$$

Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )—Conjunctive Predicates

Evaluation of  
Relational Operators

Torsten Grust



### 📎 Selectivity of conjunctive predicates

What can you say about the selectivity of the conjunctive predicate  $p \text{ AND } q$ ?

$$sel(p \text{ AND } q) =$$

Now assume  $p \equiv \text{AGE} \leq 16$  and  $q \equiv \text{SALARY} > 5000$ .  
Reconsider your proposal above.

Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

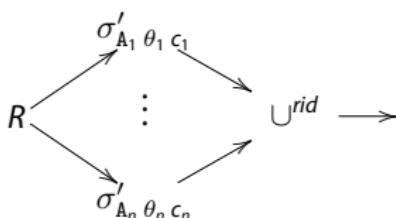
## Selection ( $\sigma$ )—Disjunctive Predicates

- Choosing a reasonable execution plan for **disjunctive selection** of the general form

$$A_1 \theta_1 c_1 \text{ OR } A_2 \theta_2 c_2 \text{ OR } \dots \text{ OR } A_n \theta_n c_n$$

is much harder:

- We are forced to **fall back to a naive file scan based evaluation** as soon only a **single term does not match** an index.
- If **all terms are matched** by indexes, we can exploit a **rid-based set union**  $\cup^{rid}$  to improve the plan:



Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

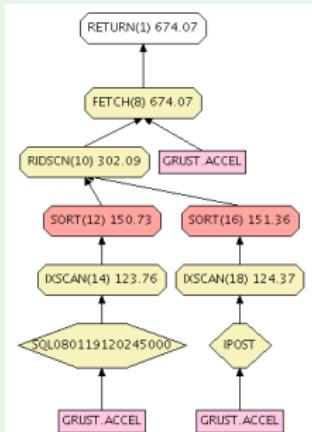
Hash Join

Operator Pipelining

Volcano Iterator Model

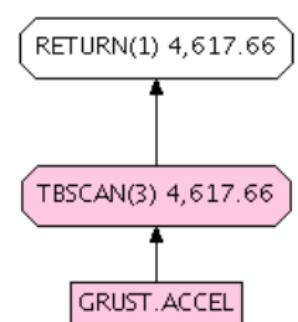
## Selection ( $\sigma$ )—Disjunctive Predicates

### DB2. Selective disjunctive predicate



**Note:** Multi-input RIDSCN operator.

### DB2. Non-selective disjunctive predicate



**Note:** Presence of indexes ignored.



## Selection ( $\sigma$ )—Disjunctive Predicates

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

### 📎 Selectivity of disjunctive predicates

What can you say about the selectivity of the disjunctive predicate  $p \text{ OR } q$ ?

$$\text{sel}(p \text{ OR } q) =$$

## Projection ( $\pi$ )

- **Projection** ( $\pi_\ell$ ) modifies each record in its input file and cuts off any field not listed in the attribute list  $\ell$ :

### Relational projection

$$\pi_{A,B} \left( \begin{array}{ccc} A & B & C \\ \hline 1 & 'foo' & 3 \\ 1 & 'bar' & 2 \\ 1 & 'foo' & 2 \\ 1 & 'bar' & 0 \\ 1 & 'foo' & 0 \end{array} \right) \underset{\textcircled{1}}{=} \begin{array}{cc} A & B \\ \hline 1 & 'foo' \\ 1 & 'bar' \\ 1 & 'foo' \\ 1 & 'bar' \\ 1 & 'foo' \end{array} \underset{\textcircled{2}}{=} \begin{array}{cc} A & B \\ \hline 1 & 'foo' \\ 1 & 'bar' \end{array}$$

- In general, the size of the resulting file will only be a fraction of the original input file:
  - ① any unwanted fields (here: C) have been thrown away, and
  - ② optionally **duplicates removed** (SQL: DISTINCT).



Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

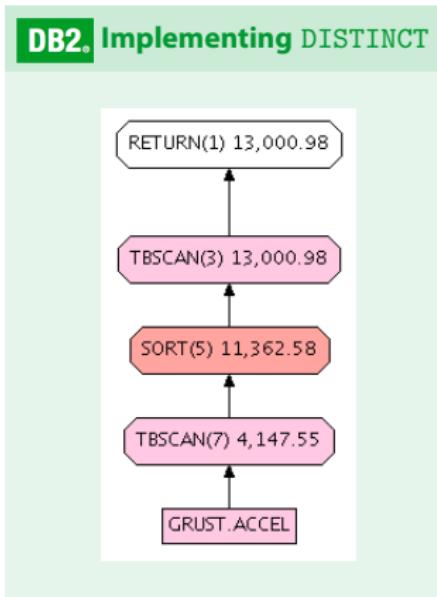
Hash Join

Operator Pipelining

Volcano Iterator Model

## Projection ( $\pi$ )—Duplicate Elimination, Sorting

- **Sorting** is one obvious preparatory step to facilitate duplicate elimination: records with all fields equal will end up adjacent to each other.



- One benefit of sort-based projection is that operator  $\pi_\ell$  will write a sorted output file, i.e.,

$$R \xrightarrow{?} \pi_\ell^{\text{sort}} \xrightarrow{s}$$

### Sort ordering?

What would be the correct ordering  $\theta$  to apply in the case of duplicate elimination?

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Projection ( $\pi$ )—Duplicate Elimination, Hashing

- If the DBMS has a fairly large number of buffer pages ( $B$ , say) to spare for the  $\pi_\ell(R)$  operation, a **hash-based** projection may be an efficient alternative to sorting:

### Hash-based projection $\pi_\ell$ : partitioning phase

- Allocate all  $B$  buffer pages. One page will be the **input buffer**, the remaining  $B - 1$  pages will be used as **hash buckets**.
- Read the file  $R$  page-by-page, for each record  $r$ : cut off fields not listed in  $\ell$ .
- For each such record, apply hash function  $h_1(r) = h(r) \bmod (B - 1)$ —which depends on **all remaining fields of  $r$** —and store  $r$  in hash bucket  $h_1(r)$ . (Write the bucket to disk if full.)

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

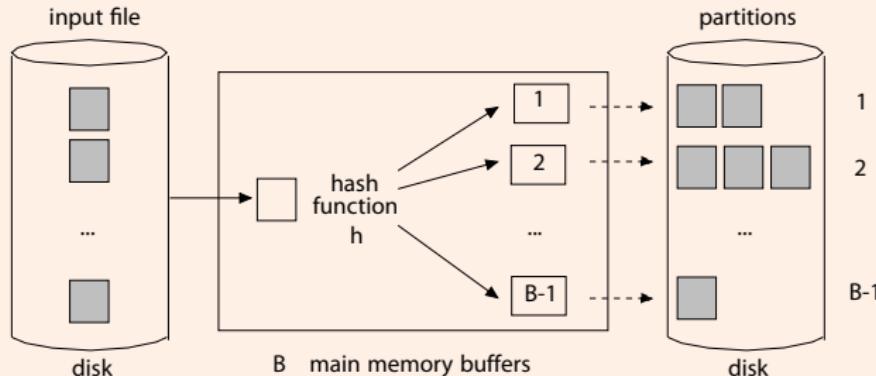
Hash Join

Operator Pipelining

Volcano Iterator Model

## Projection ( $\pi$ )—Hashing

### Hash-based projection $\pi_\ell$ : partitioning phase



- After partitioning, duplicate elimination becomes an **intra-partition** problem only: two identical records have been mapped to the same partition:

$$h_1(r) = h_1(r') \iff r = r' .$$

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Projection ( $\pi$ )—Hashing

### Hash-based projection $\pi_{\ell}$ : duplicate elimination phase

- ① For each partition, read each partition page-by-page (possibly in parallel).
- ② To each record, apply hash function  $h_2 \neq h_1$  to all record fields.
- ③ Only if two records **collide** w.r.t.  $h_2$ , check if  $r = r'$ . 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).

### 💡 Huge partitions?

**Note:** Works efficiently only if duplicate elimination phase can be **performed in the buffer** (main memory).

### What to do if partition size exceeds buffer size?

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## The Join Operator ( $\bowtie_p$ )

The **join operator**  $\bowtie_p$  is actually a short-hand for a combination of **cross product**  $\times$  and **selection**  $\sigma_p$ .

### Join vs. Cartesian product



One way to implement  $\bowtie_p$  is to follow this equivalence:

- ① Enumerate and concatenate all records in the cross product of  $r_1$  and  $r_2$ .
- ② Then pick those that satisfy  $p$ .

More advanced algorithms try to avoid the obvious inefficiency in Step ① (the size of the intermediate result is  $|R| \cdot |S|$ ).



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join  
Block Nested Loops Join  
Index Nested Loops Join  
Sort-Merge Join  
Hash Join

### Operator Pipelining

Volcano Iterator Model

## Nested Loops Join

The **nested loops join** is the straightforward implementation of the  $\sigma \times$  combination:

### Nested loops join

```
1 Function: nljoin ( $R, S, p$ )
  /* outer relation  $R$  */ *
2 foreach record  $r \in R$  do
  /* inner relation  $S$  */ *
3   foreach record  $s \in S$  do
    /*  $\langle r, s \rangle$  denotes record concatenation */ *
4     if  $\langle r, s \rangle$  satisfies  $p$  then
      append  $\langle r, s \rangle$  to result
  
```

Let  $N_R$  and  $N_S$  the number of **pages** in  $R$  and  $S$ ; let  $p_R$  and  $p_S$  be the number of records per page in  $R$  and  $S$ .

The **total number of disk reads** then is

$$N_R + \underbrace{p_R \cdot N_R \cdot N_S}_{\text{\# tuples in } R} .$$

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Nested Loops Join: I/O Behavior



The **good news** about `njoin()` is that it needs only **three pages** of buffer space (two to read  $R$  and  $S$ , one to write the result).

The **bad news** is its enormous I/O cost:

- Assuming  $p_R = p_S = 100$ ,  $N_R = 1000$ ,  $N_S = 500$ , we need to read  $1000 + (5 \cdot 10^7)$  disk pages.
- With an access time of 10 ms for each page, this join would take 140 hours!
- Switching the role of  $R$  and  $S$  to make  $S$  (the smaller one) the **outer relation** does not bring any significant advantage.

Note that reading data page-by-page (even tuple-by-tuple) means that **every** I/O suffers the disk latency penalty, even though we process both relations in sequential order.

### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

## Block Nested Loops Join

Evaluation of  
Relational Operators

Torsten Grust



- Again we can **save random access cost** by reading  $R$  and  $S$  in **blocks** of, say,  $b_R$  and  $b_S$  pages.

### Block nested loops join

```
1 Function: block_nljoin ( $R, S, p$ )
2 foreach  $b_R$ -sized block in  $R$  do
3   foreach  $b_S$ -sized block in  $S$  do
4     /* performed in the buffer */           */
      find matches in current  $R$ - and  $S$ -blocks and append them
      to the result ;
```

Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

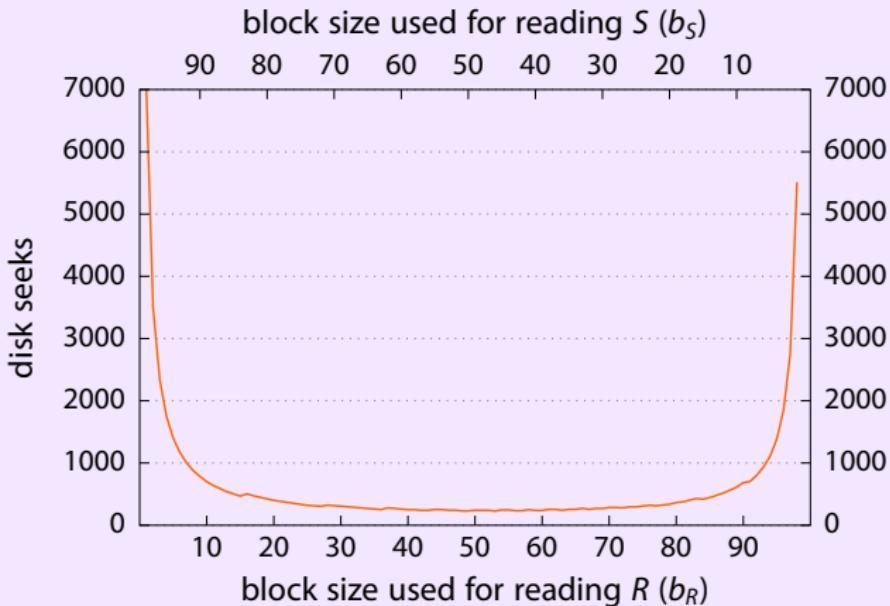
Volcano Iterator Model

- $R$  is still read once, but now with only  $\lceil N_R/b_R \rceil$  disk seeks.
- $S$  is scanned only  $\lceil N_R/b_R \rceil$  times now, and we need to perform  $\lceil N_R/b_R \rceil \cdot \lceil N_S/b_S \rceil$  disk seeks to do this.

## Choosing $b_R$ and $b_S$

**Example:** Buffer pool with  $B = 100$  frames,  $N_R = 1000$ ,  $N_S = 500$ :

### Example (Choosing $b_r$ and $b_s$ )



## In-Memory Join Performance

- Line 4 in `block_nljoin' (R, S, p)` implies an **in-memory join** between the  $R$ - and  $S$ -blocks currently in memory.
- Building a hash table over the  $R$ -block can speed up this join considerably.

### Block nested loops join: build hash table from outer row block

```
1 Function: block_nljoin' (R, S, p)
2   foreach  $b_R$ -sized block in  $R$  do
3     build an in-memory hash table  $H$  for the current  $R$ -block ;
4     foreach  $b_S$ -sized block in  $S$  do
5       foreach record  $s$  in current  $S$ -block do
6         probe  $H$  and append matching  $\langle r, s \rangle$  tuples to result ;
```

- Note that this optimization only helps **equi-joins**.



## 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 nested loops join

```
1 Function: index_nljoin( $R, S, p$ )
2 foreach record  $r \in R$  do
3   scan  $S$ -index using (key value in)  $r$  and concatenate  $r$  with all
4   matching tuples  $s$  ;
    append  $\langle r, s \rangle$  to result ;
```

- The index must **match** the join condition  $p$ .
  - Hash indices, e.g., only support equality predicates.
  - Remember the discussion about composite keys in  $B^+$ -trees.
- Such predicates are also called **sargable** (*sarg*: search argument  $\nearrow$  Selinger *et al.*, SIGMOD 1979)



## Index Nested Loop Join: I/O Behavior

For each record in  $R$ , we use the index to find matching  $S$ -tuples. While searching for matching  $S$ -tuples, we incur the following I/O costs **for each tuple** in  $R$ :

- ① **Access** the index to find its first matching entry:  $N_{\text{idx}}$  I/Os.
- ② **Scan** the index to retrieve **all**  $n$  matching  $rids$ . The I/O cost for this is typically negligible (locality in the index).
- ③ **Fetch** the  $n$  matching  $S$ -tuples from their data pages.
  - For an **unclustered** index, this requires  $n$  I/Os.
  - For a **clustered** index, this only requires  $\lceil n/p_s \rceil$  I/Os.

Note that (due to ② and ③), the cost of an index nested loops join becomes **dependent on the size of the join result**.



## Index Access Cost

If the index is a **B<sup>+</sup>-tree index**:

- A **single** index access requires the inspection of  $h$  pages.<sup>3</sup>
- If we **repeatedly** probe the index, however, most of these are **cached** by the buffer manager.
- The effective value for  $N_{\text{idx}}$  is around 1–3 I/Os.

If the index is a **hash index**:

- Caching will not help here (no locality in accesses to hash table).
- A typical value for  $N_{\text{idx}}$  is 1.2 I/Os ( $> 1$  due to overflow pages).

Overall, the use of an index (over, e.g., a block nested loops join) pays off if the join is **selective** (picks out only few tuples from a big table).

---

<sup>3</sup> $h$ : B<sup>+</sup>-tree height



## Sort-Merge Join

Join computation becomes particularly simple if both inputs are **sorted** with respect to the join attribute(s).

- The **merge join** essentially **merges** both input tables, much like we did for sorting. Both tables are **read once, in parallel**.
- In contrast to sorting, however, we need to be careful whenever a tuple has **multiple** matches in the other relation:

### Multiple matches per tuple (disrupts sequential access)

A	B		C	D
"foo"	1		1	false
"foo"	2		2	true
"bar"	2	$\bowtie$ B=C	2	false
"baz"	2		3	true
"baf"	4			

- Merge join is typically used for **equi-joins only**.



# Merge Join: Algorithm

## Merge join algorithm

```
1 Function: merge_join ( $R, S, \alpha = \beta$ ) //  $\alpha, \beta$ : join cols in  $R, S$ 
2  $r \leftarrow$  position of first tuple in  $R$ ; //  $r, s, s'$ : cursors over  $R, S, S$ 
3  $s \leftarrow$  position of first tuple in  $S$ ;
4 while  $r \neq \langle \text{EOF} \rangle$  and  $s \neq \langle \text{EOF} \rangle$  do //  $\langle \text{EOF} \rangle$ : end of file marker
5   while  $r.\alpha < s.\beta$  do
6      $\downarrow$  advance  $r$ ;
7   while  $r.\alpha > s.\beta$  do
8      $\downarrow$  advance  $s$ ;
9    $s' \leftarrow s$ ; // Remember current position in  $S$ 
10  while  $r.\alpha = s'.\beta$  do // All  $R$ -tuples with same  $\alpha$  value
11     $s \leftarrow s'$ ; // Rewind  $s$  to  $s'$ 
12    while  $r.\alpha = s.\beta$  do // All  $S$ -tuples with same  $\beta$  value
13      append  $\langle r, s \rangle$  to result;
14       $\downarrow$  advance  $s$ ;
15     $\downarrow$  advance  $r$ ;
```



## Merge Join: I/O Behavior

- If both inputs are already sorted **and** there are no exceptionally long sequences of identical key values, the I/O cost of a merge join is  $N_R + N_S$  (which is optimal).
- By using **blocked I/O**, these I/O operations can be done almost entirely as **sequential** reads.
- Sometimes, it pays off to explicitly **sort** a (unsorted) relation first, then apply merge join. This is particularly the case if a sorted **output** is beneficial later in the execution plan.
- The final sort pass can also be combined with merge join, avoiding one round-trip to disk and back.



What is the worst-case behavior of merge join?

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Merge Join: I/O Behavior

- If both inputs are already sorted **and** there are no exceptionally long sequences of identical key values, the I/O cost of a merge join is  $N_R + N_S$  (which is optimal).
- By using **blocked I/O**, these I/O operations can be done almost entirely as **sequential** reads.
- Sometimes, it pays off to explicitly **sort** a (unsorted) relation first, then apply merge join. This is particularly the case if a sorted **output** is beneficial later in the execution plan.
- The final sort pass can also be combined with merge join, avoiding one round-trip to disk and back.

### ☞ What is the worst-case behavior of merge join?

If both join attributes are constants and carry the same value (i.e., the result is the Cartesian product), merge join degenerates into a nested loops join.

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

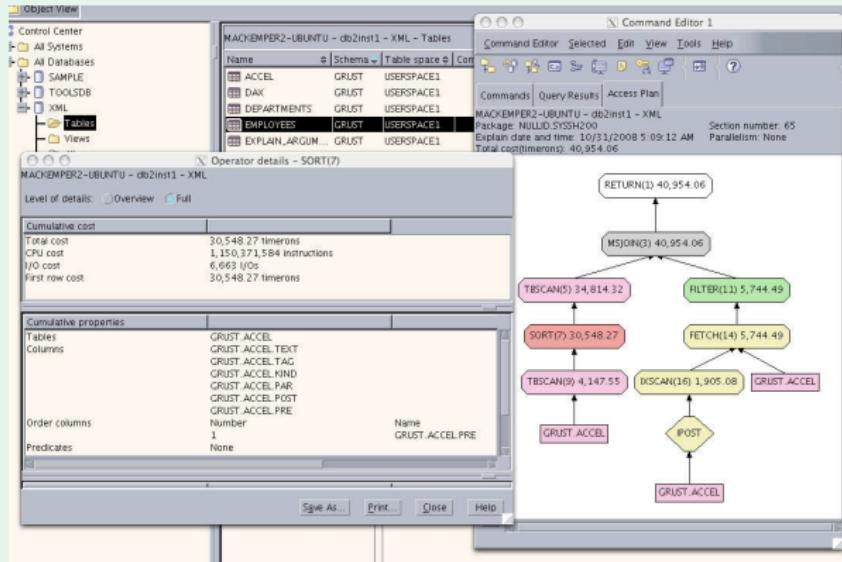
Hash Join

Operator Pipelining

Volcano Iterator Model

# Merge Join: IBM DB2 Plan

## DB2. Merge join (left input: sort, right input: sorted index scan)



- **Note:** The **FILTER(11)** implements the join predicate of the **MSJOIN(3)**.

Evaluation of Relational Operators

Torsten Grust



Relational Query Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

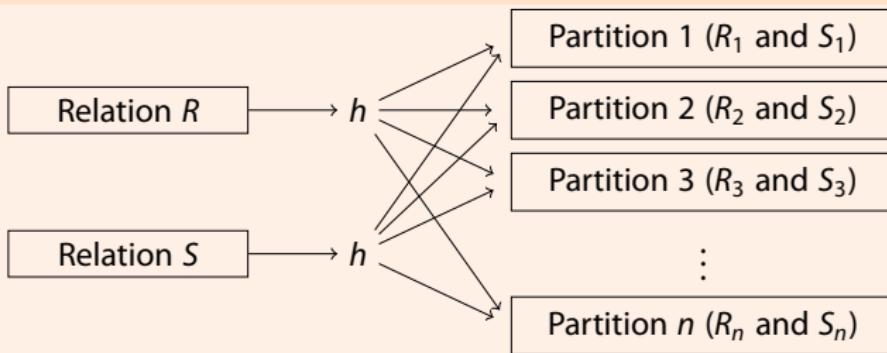
Operator Pipelining

Volcano Iterator Model

## Hash Join

- Sorting effectively brought related tuples into **spatial proximity**, which we exploited in the merge join algorithm.
- We can achieve a similar effect with **hashing**, too.
- Partition  $R$  and  $S$  into partitions  $R_1, \dots, R_n$  and  $S_1, \dots, S_n$  using the **same** hash function (applied to the join attributes).

### Hash partitioning for both inputs



- Observe that  $R_i \bowtie S_j = \emptyset$  for all  $i \neq j$ .

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Hash Join

- By partitioning the data, we reduced the problem of joining to **smaller sub-relations**  $R_i$  and  $S_i$ .
- Matching tuples are guaranteed to end up together in the same partition (again: works for equality predicates only).
- We only need to compute  $R_i \bowtie S_i$  (for all  $i$ ).
- By choosing  $n$  properly (i.e., the hash function  $h$ ), partitions become small enough to implement the  $R_i \bowtie S_i$  as **in-memory joins**.
- The in-memory join is typically accelerated using a hash table, too. We already did this for the block nested loops join (↗ slide 34).

### 📎 Intra-partition join via hash table

Use a **different** hash function  $h' \neq h$  for the intra-partition join.  
**Why?**

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

# Hash Join Algorithm



```
1 Function: hash_join ( $R, S, \alpha = \beta$ )
  /* Partitioning phase */ */
  2 foreach record  $r \in R$  do
    | append  $r$  to partition  $R_{h(r.\alpha)}$ 
  3 foreach record  $s \in S$  do
    | append  $s$  to partition  $S_{h(s.\beta)}$ 
  4 /* Intra-partition join phase */ */
  5 foreach partition  $i \in 1, \dots, n$  do
    | build hash table  $H$  for  $R_i$ , using hash function  $h'$ ;
    | foreach block  $b \in S_i$  do
      |   foreach record  $s \in b$  do
        |     | probe  $H$  via  $h'(s.\beta)$  and append matching tuples to
        |     | result ;
    |
```

## Hash Join—Buffer Requirements

- We assumed that we can create the necessary  $n$  partitions in one pass (note that we want  $N_{R_i} < (B - 1)$ ).
- This works out if  $R$  consists of **at most**  $\approx (B - 1)^2$  pages.

📎 Why  $(B - 1)^2$ ? Why  $\approx$ ?

- Larger input tables require **multiple passes** for partitioning (recursive partitioning).

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Hash Join—Buffer Requirements

- We assumed that we can create the necessary  $n$  partitions in one pass (note that we want  $N_{R_i} < (B - 1)$ ).
- This works out if  $R$  consists of **at most**  $\approx (B - 1)^2$  pages.

### 📎 Why $(B - 1)^2$ ? Why $\approx$ ?

- We can write out at most  $B - 1$  partitions in one pass; the  $R$  part of each partition should be at most  $B - 1$  pages in size.
- Hashing does not guarantee an even distribution. Since the actual size of each partition varies,  $R$  must actually be smaller than  $(B - 1)^2$ .
- Larger input tables require **multiple passes** for partitioning (recursive partitioning).

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

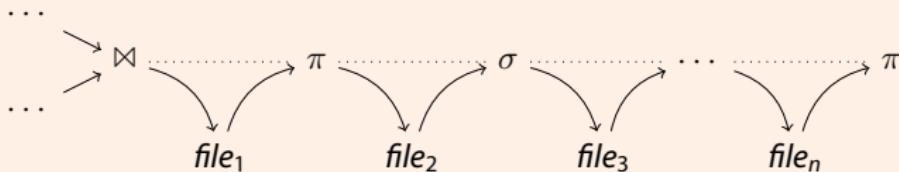
Operator Pipelining

Volcano Iterator Model

## Orchestrating Operator Evaluation

So far we have assumed that all database operators consume and produce **files** (i.e., on-disk items):

### File-based operator input and output



- Obviously, using **secondary storage as the communication channel** causes **a lot of disk I/O**.
- In addition, we suffer from **long response times**:
  - An operator cannot start computing its result before **all** its input files are fully generated ("**materialized**").
  - Effectively, all operators are executed **in sequence**.

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

# Unix: Temporary Result Files

- Architecting the query processor in this fashion bears much resemblance with using the Unix shell like this:

## File-based Unix command sequencing

```
1 # report "large" XML files below current working dir
2 $ find . -size +500k      > tmp1
3 $ xargs file             < tmp1 > tmp2
4 $ grep -i XML            < tmp2 > tmp3
5 $ cut -d: -f1             < tmp3
6 {output generated here}
7
8 # remove temporary files
9 $ rm -f tmp[0-9]*
```

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Pipelined Evaluation

- Alternatively, each operator could pass its result **directly** on to the next operator (without persisting it to disk first).
  - ⇒ Do not wait until entire file is created, but propagate output **immediately**.
  - ⇒ Start computing results **as early as possible**, i.e., as soon as enough input data is available to start producing output.
- This idea is referred to as **pipelining**.
- The **granularity** in which data is passed may influence performance:
  - Smaller chunks reduce the **response time** of the system.
  - Larger chunks may improve the effectiveness of **(instruction) caches**.
  - Actual systems typically operate **tuple at a time**.

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Unix: Pipelines of Processes

Unix provides a similar mechanism to communicate between processes ("operators"):

### Pipeline-based Unix command sequencing

```
1 $ find . -size +500k | xargs file | grep -i XML | cut -d: -f1  
2 <output generated here>
```

Execution of this pipe is driven by the **rightmost** operator—all operators act in **parallel**:



- To produce a line of output, `cut` only needs to see the next line of its input: `grep` is requested to produce this input.
- To produce a line of output, `grep` needs to request as many input lines from the `xargs` process until it receives a line containing the string "XML".
- ⋮
- Each line produced by the `find` process is passed through the pipe until it reaches the `cut` process and eventually is echoed to the terminal.

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## The Volcano Iterator Model

- The **calling interface** used in database execution runtimes is very similar to the one used in Unix process pipelines.
- In databases, this interface is referred to as **open-next-close interface** or **Volcano iterator model**.
- Each operator implements the functions
  - open ()** **Initialize** the operator's internal state.
  - next ()** Produce and return the **next result tuple** or **<EOF>**.
  - close ()** **Clean up** all allocated resources (typically after all tuples have been processed).
- All **state** is kept inside each operator instance:
  - Operators are required to produce a tuple via **next ()**, then **pause**, and later **resume** on a subsequent **next ()** call.

↗ Goetz Graefe. Volcano—An Extensible and Parallel Query Evaluation System. *Trans. Knowl. Data Eng.* vol. 6, no. 1, February 1994.

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

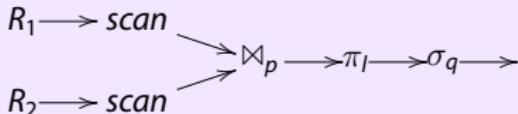
Hash Join

Operator Pipelining

Volcano Iterator Model

## Volcano-Style Pipelined Evaluation

### Example (Pipelined query plan)



- Given a query plan like the one shown above, query evaluation is driven by the query processor like this (just like in the Unix shell):
  - ① The whole plan is initially reset by calling `open()` on the **root operator**, i.e.,  $\sigma_q.open()$ .
  - ② The `open()` call is **forwarded** through the plan by the operators themselves (see  $\sigma.open()$  on slide 53).
  - ③ Control returns to the query processor.
  - ④ The root is requested to produce its next result record, i.e., the call  $\sigma_q.next()$  is made.
  - ⑤ Operators forward the `next()` request as needed. **As soon as the next result record is produced, control returns** to the query processor again.

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

# Volcano-Style Pipelined Evaluation

- In a nutshell, the query processor uses the following routine to evaluate a query plan:

## Query plan evaluation driver

```
1 Function: eval (q)
2   q.open ();
3   r ← q.next ();
4   while r ≠ <EOF> do
5     /* deliver record r (print, ship to DB client) */
6     emit (r);
7     r ← q.next ();
8   /* resource deallocation now */ *
9   q.close ();
```

Evaluation of  
Relational Operators

Torsten Grust



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model

## Volcano-Style Selection ( $\sigma_p$ )

- Input operator (sub-plan root)  $R$ , predicate  $p$ :

### Volcano-style interface for $\sigma_p(R)$

1 **Function:** `open ()`

2  `$R.open () ;$`

---

1 **Function:** `close ()`

2  `$R.close () ;$`

---

1 **Function:** `next ()`

2 **while**  $((r \leftarrow R.next ()) \neq \langle EOF \rangle)$  **do**

3   **if**  $p(r)$  **then**  
4     **return**  $r$ ;

5 **return**  $\langle EOF \rangle$ ;



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

# Volcano-Style Nested Loops Join ( $\bowtie_p$ )

📎 A Volcano-style implementation of nested loops join  $R \bowtie_p S$ ?

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Volcano-Style Nested Loops Join ( $\bowtie_p$ )

📎 A Volcano-style implementation of nested loops join  $R \bowtie_p S$ ?

1 **Function:** open ()

2  $R.\text{open}();$

3  $S.\text{open}();$

4  $r \leftarrow R.\text{next}();$

1 **Function:** close ()

2  $R.\text{close}();$

3  $S.\text{close}();$

---

1 **Function:** next ()

2 **while** ( $r \neq \langle \text{EOF} \rangle$ ) **do**

3     **while** (( $s \leftarrow S.\text{next}()$ )  $\neq \langle \text{EOF} \rangle$ ) **do**

4         **if**  $p(r, s)$  **then**

           /\* emit concatenated result \*/

**return**  $\langle r, s \rangle;$

5         /\* reset inner join input \*/

\*/

6          $S.\text{close}();$

7          $S.\text{open}();$

8          $r \leftarrow R.\text{next}();$

9 **return**  $\langle \text{EOF} \rangle;$

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

# Pipelining (Real DBMS Product)

## Volcano-style pipelined selection operator (C code)

```
1 /* eFLTR -- apply filter predicate pred to stream
2  Filter the in-bound stream, only stream elements that fulfill
3  e->pred contribute to the result. No index support. */
4
5 eRC eOp_FLTR(eOp *ip) {
6     eObj_FLTR *e = (eObj_FLTR *)eObj(ip);
7
8     while (eIntp(e->in) != eEOS) {
9         eIntp(e->pred);
10        if (eT_as_bool(eVal(e->pred))) {
11            eVal(ip) = eVal(e->in);
12            return eOK;
13        }
14    }
15    return eEOS;
16 }
17
18 eRC eOp_FLTR_RST(eOp *ip) {
19     eObj_FLTR *e = (eObj_FLTR *)eObj(ip);
20
21     eReset(e->in);
22     eReset(e->pred);
23     return eOK;
24 }
```

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Blocking Operators

- Pipelining reduces memory requirements and response time since each chunk of input is propagated to the output **immediately**.
- Some operators **cannot** be implemented in such a way.

📎 **Which operators do not permit pipelined evaluation?**

- Such operators are said to be **blocking**.
- Blocking operators **consume their entire input before they can produce any output**.
  - The data is typically buffered (“materialized”) on disk.

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

## Blocking Operators

- Pipelining reduces memory requirements and response time since each chunk of input is propagated to the output **immediately**.
- Some operators **cannot** be implemented in such a way.

### 📎 Which operators do not permit pipelined evaluation?

- (external) sorting (this is also true for Unix sort)
  - hash join
  - grouping and duplicate elimination over unsorted input
- 
- Such operators are said to be **blocking**.
  - Blocking operators **consume their entire input before they can produce any output**.
    - The data is typically buffered (“materialized”) on disk.

Evaluation of  
Relational Operators

Torsten Grust



Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )

Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

Operator Pipelining

Volcano Iterator Model

# Chapter 9

## Cardinality Estimation

How Many Rows Does a Query Yield?

*Architecture and Implementation of Database Systems*  
Summer 2014

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator  
Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

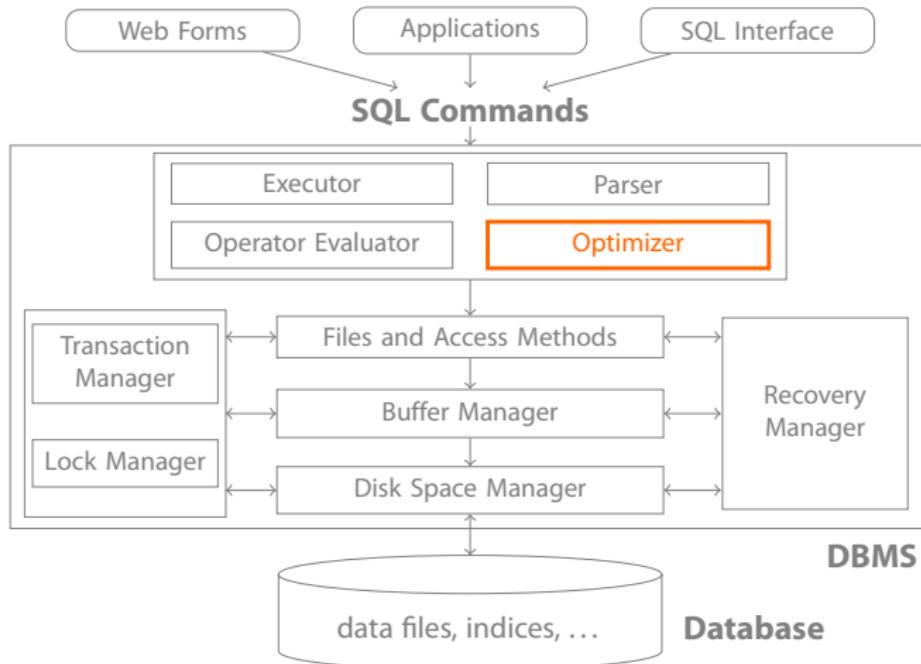
Equi-Depth

Statistical Views

Torsten Grust  
Wilhelm-Schickard-Institut für Informatik  
Universität Tübingen



# Cardinality Estimation



Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## Cardinality Estimation

- A relational query optimizer performs a phase of **cost-based plan search** to identify the—presumably—“cheapest” alternative among a set of equivalent execution plans ( $\nearrow$  Chapter on Query Optimization).
- Since page I/O cost dominates, the **estimated cardinality of a (sub-)query result** is crucial input to this search.
  - Cardinality typically measured in pages or rows.
- **Cardinality estimates** are also valuable when it comes to buffer “right-sizing” **before query evaluation starts** (e.g., allocate  $B$  buffer pages and determine blocking factor  $b$  for external sort).

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## Estimating Query Result Cardinality

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator  
Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

There are two principal approaches to query cardinality estimation:

### ① Database Profile.

Maintain statistical information about *numbers and sizes of tuples, distribution of attribute values* for **base relations**, as part of the **database catalog** (meta information) **during database updates**.

- Calculate these parameters for **intermediate query results** based upon a (simple) statistical model during query optimization.
- Typically, the statistical model is based upon the **uniformity** and **independence assumptions**.
- Both are typically **not valid**, but they allow for simple calculations  $\Rightarrow$  **limited accuracy**.
- In order to improve accuracy, the system can record **histograms** to more closely model the actual value distributions in relations.



# Estimating Query Result Cardinality

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator  
Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## ② Sampling Techniques.

Gather the necessary characteristics of a query plan (base relations and intermediate results) at **query execution time**:

- Run query on a small sample of the input.
- Extrapolate to the full input size.
- It is crucial to find the right balance between sample size and the resulting accuracy.

These slides focus on ① Database Profiles.

## Database Profiles

Keep profile information in the **database catalog**. Update whenever SQL DML commands are issued (database updates):

### Typical database profile for relation $R$

$|R|$

**number of records** in relation  $R$

$N_R$

**number of disk pages** allocated for these records

$s(R)$

average **record size** (width)

$V(A, R)$

**number of distinct values** of attribute A

$MCV(A, R)$

**most common values** of attribute A

$MCF(A, R)$

**frequency** of most common values of attribute A

:

*possibly many more*

### Cardinality Estimation

Torsten Grust



### Cardinality Estimation

#### Database Profiles

Assumptions

#### Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

#### Histograms

Equi-Width

Equi-Depth

#### Statistical Views

## Database Profiles: IBM DB2

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

```
1 db2 => SELECT TABNAME, CARD, NPAGES  
2 db2 (cont.) => FROM SYSCAT.TABLES  
3 db2 (cont.) => WHERE TABSCHEMA = 'TPCH';
```

4 TABNAME	CARD	NPAGES
5 -----		
6 ORDERS	1500000	44331
7 CUSTOMER	150000	6747
8 NATION	25	2
9 REGION	5	1
10 PART	200000	7578
11 SUPPLIER	10000	406
12 PARTSUPP	800000	31679
13 LINEITEM	6001215	207888

```
14 8 record(s) selected.
```

- **Note:** Column CARD  $\equiv |R|$ , column NPAGES  $\equiv N_R$ .

## Database Profile: Assumptions

In order to obtain tractable cardinality estimation formulae, assume *one of the following*:

### Uniformity & independence (simple, yet rarely realistic)

All values of an attribute uniformly appear with the same probability (even distribution). Values of different attributes are independent of each other.

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator  
Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## Database Profile: Assumptions

In order to obtain tractable cardinality estimation formulae, assume *one of the following*:

### Uniformity & independence (simple, yet rarely realistic)

All values of an attribute uniformly appear with the same probability (even distribution). Values of different attributes are independent of each other.

### Worst case (unrealistic)

No knowledge about relation contents at all. In case of a selection  $\sigma_p$ , assume all records will satisfy predicate  $p$ .

(May only be used to compute upper bounds of expected cardinality.)

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator  
Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## Database Profile: Assumptions

In order to obtain tractable cardinality estimation formulae, assume *one of the following*:

### Uniformity & independence (simple, yet rarely realistic)

All values of an attribute uniformly appear with the same probability (even distribution). Values of different attributes are independent of each other.

### Worst case (unrealistic)

No knowledge about relation contents at all. In case of a selection  $\sigma_p$ , assume all records will satisfy predicate  $p$ .

(May only be used to compute upper bounds of expected cardinality.)

### Perfect knowledge (unrealistic)

Details about the exact distribution of values are known. Requires huge catalog or prior knowledge of incoming queries.

(May only be used to compute lower bounds of expected cardinality.)

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## Cardinality Estimation for $\sigma$ (Equality Predicate)

- Database systems typically operate under the **uniformity assumption**. We will come across this assumption multiple times below.

**Query:**  $Q \equiv \sigma_{A=c}(R)$

**Selectivity**  $sel(A = c) = MCF(A, R)[c]$  if  $c \in MCV(A, R)$

**Selectivity**  $sel(A = c) = 1/V(A, R)$

**Uniformity** 

**Cardinality**  $|Q| = sel(A = c) \cdot |R|$

**Record size**  $s(Q) = s(R)$

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## Selectivity Estimation for $\sigma$ (Other Predicates)

- **Equality between attributes** ( $Q \equiv \sigma_{A=B}(R)$ ):

Approximate selectivity by

$$sel(A = B) = 1 / \max(V(A, R), V(B, R)) .$$

(Assumes that each value of the attribute with fewer distinct values has a corresponding match in the other attribute.)

### Independence



- **Range selections** ( $Q = \sigma_{A>c}(R)$ ):

In the database profile, **Maintain the minimum and maximum value** of attribute A in relation R,  $Low(A, R)$  and  $High(A, R)$ .

Approximate selectivity by

### Uniformity



$$sel(A > c) =$$

$$\begin{cases} \frac{High(A, R) - c}{High(A, R) - Low(A, R)}, & Low(A, R) \leq c \leq High(A, R) \\ 0, & \text{otherwise} \end{cases}$$

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup, \setminus, \times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## Cardinality Estimation for $\pi$

- For  $Q \equiv \pi_L(R)$ , estimating the number of result rows is difficult ( $L = \langle A_1, A_2, \dots, A_n \rangle$ : list of projection attributes):

$Q \equiv \pi_L(R)$       (duplicate elimination)

$$\text{Cardinality } |Q| = \begin{cases} V(A, R), & \text{if } L = \langle A \rangle \\ |R|, & \text{if keys of } R \in L \\ |R|, & \text{no dup. elim.} \\ \min(|R|, \prod_{A_i \in L} V(A_i, R)), & \text{otherwise} \end{cases}$$

Independence 

Record size  $s(Q) = \sum_{A_i \in L} s(A_i)$

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup, \setminus, \times$   
Join  $\bowtie$

Histograms

Equi-Width  
Equi-Depth

Statistical Views

## Cardinality Estimation for $\cup, \setminus, \times$

$Q \equiv R \cup S$

$$\begin{aligned} |Q| &\leq |R| + |S| \\ s(Q) &= s(R) = s(S) \quad \text{schemas of } R, S \text{ identical} \end{aligned}$$

$Q \equiv R \setminus S$

$$\begin{aligned} \max(0, |R| - |S|) &\leq |Q| \leq |R| \\ s(Q) &= s(R) = s(S) \end{aligned}$$

$Q \equiv R \times S$

$$\begin{aligned} |Q| &= |R| \cdot |S| \\ s(Q) &= s(R) + s(S) \\ V(A, Q) &= \begin{cases} V(A, R), & \text{for } A \in R \\ V(A, S), & \text{for } A \in S \end{cases} \end{aligned}$$

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup, \setminus, \times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## Cardinality Estimation for $\bowtie$

- Cardinality estimation for the general join case is challenging.
- A special, yet very common case: **foreign-key relationship** between input relations  $R$  and  $S$ :

### Establish a foreign key relationship (SQL)

```
1 CREATE TABLE R (A INTEGER NOT NULL,  
2           ...  
3           PRIMARY KEY (A));  
4 CREATE TABLE S (...,  
5           A INTEGER NOT NULL,  
6           ...  
7           FOREIGN KEY (A) REFERENCES R);
```

$$Q \equiv R \bowtie_{R.A=S.A} S$$

The foreign key constraint guarantees  $\pi_A(S) \subseteq \pi_A(R)$ . Thus:

$$|Q| = |S| .$$

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator  
Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup, \setminus, \times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## Cardinality Estimation for $\bowtie$

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator

Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

$$Q \equiv R \bowtie_{R.A=S.B} S$$

$$|Q| = \begin{cases} \frac{|R| \cdot |S|}{V(A, R)}, & \pi_B(S) \subseteq \pi_A(R) \\ \frac{|R| \cdot |S|}{V(B, S)}, & \pi_A(R) \subseteq \pi_B(S) \end{cases}$$

$$s(Q) = s(R) + s(S)$$

## Histograms

- In realistic database instances, values are **not uniformly distributed** in an attribute's **active domain** (actual values found in a column).
- To keep track of this non-uniformity for an attribute A, maintain a **histogram** to **approximate the actual distribution**:
  - ➊ Divide the active domain of A into adjacent intervals by selecting **boundary values**  $b_i$ .
  - ➋ Collect statistical parameters for each interval between boundaries, e.g.,
    - # of rows  $r$  with  $b_{i-1} < r.A \leq b_i$ , or
    - # of distinct A values in interval  $(b_{i-1}, b_i]$ .
- The histogram intervals are also referred to as **buckets**.

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

(↗ Y. Ioannidis: The History of Histograms (Abridged), Proc. VLDB 2003)

## Histograms in IBM DB2

### DB2. Histogram maintained for a column in a TPC-H database

```
1 SELECT SEQNO, COLVALUE, VALCOUNT  
2   FROM SYSCAT.COLDIST  
3  WHERE TABNAME = 'LINEITEM'  
4    AND COLNAME = 'L_EXTENDEDPRICE'  
5    AND TYPE = 'Q';
```

SEQNO	COLVALUE	VALCOUNT
1	+0000000000996.01	3001
2	+0000000004513.26	315064
3	+0000000007367.60	633128
4	+0000000011861.82	948192
5	+0000000015921.28	1263256
6	+0000000019922.76	1578320
7	+0000000024103.20	1896384
8	+0000000027733.58	2211448
9	+0000000031961.80	2526512
10	+0000000035584.72	2841576
11	+0000000039772.92	3159640
12	+0000000043395.75	3474704
13	+0000000047013.98	3789768
21	⋮	

- Catalog table SYSCAT.COLDIST also contains information like

- the  $n$  most frequent values (and their frequency),
- the number of distinct values in each bucket.

- Histograms may even be manipulated **manually** to tweak optimizer decisions.

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

# Histograms

- Two types of histograms are widely used:

## ① Equi-Width Histograms.

All buckets have the **same width**, i.e., boundary  
 $b_i = b_{i-1} + w$ , for some fixed  $w$ .

## ② Equi-Depth Histograms.

All buckets contain the **same number of rows** (i.e., their width is varying).

- Equi-depth histograms (②) are able to adapt to data skew (high non-uniformity).
- The number of buckets is the *tuning knob* that defines the **tradeoff** between estimation quality (**histogram resolution**) and **histogram size**: catalog space is limited.

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## Equi-Width Histograms

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator  
Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

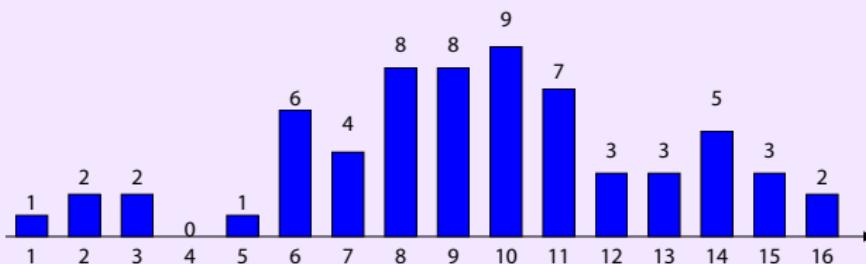
Equi-Width

Equi-Depth

Statistical Views

### Example (Actual value distribution)

Column A of SQL type INTEGER (domain  $\{\dots, -2, -1, 0, 1, 2, \dots\}$ ).  
Actual non-uniform distribution in relation  $R$ :

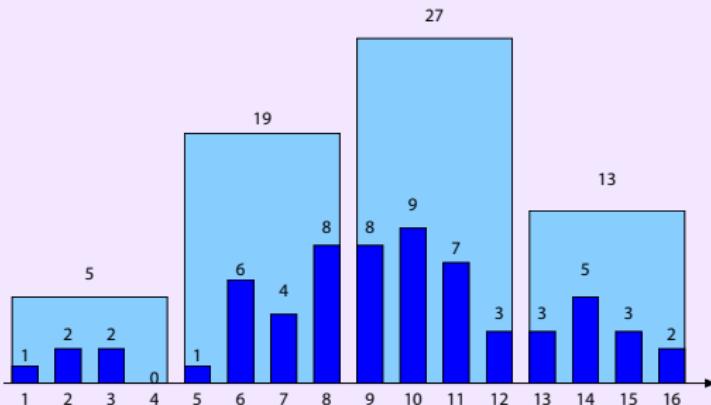


## Equi-Width Histograms

- Divide **active domain** of attribute A into  $B$  buckets of equal width. The **bucket width**  $w$  will be

$$w = \frac{\text{High}(A, R) - \text{Low}(A, R) + 1}{B}$$

### Example (Equi-width histogram ( $B = 4$ ))



- Maintain **sum of value frequencies** in each bucket (in addition to bucket boundaries  $b_i$ ).

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

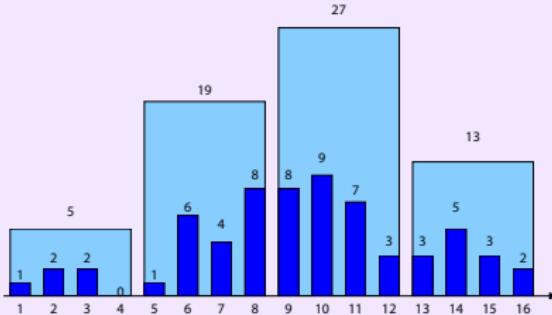
Equi-Width

Equi-Depth

Statistical Views

## Equi-Width Histograms: Equality Selections

Example ( $Q \equiv \sigma_{A=5}(R)$ )



- Value 5 is in bucket [5, 8] (with 19 tuples)
- Assume **uniform distribution within the bucket**:

$$|Q| = 19/w = 19/4 \approx 5 .$$

Actual:  $|Q| = 1$

What would be the cardinality under the uniformity assumption (no histogram)?

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

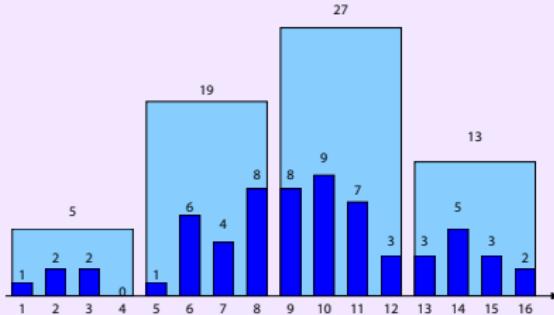
Equi-Width

Equi-Depth

Statistical Views

## Equi-Width Histograms: Range Selections

Example ( $Q \equiv \sigma_{A>7} \text{ AND } A \leq 16(R)$ )



- Query interval  $(7, 16]$  covers buckets  $[9, 12]$  and  $[13, 16]$ .  
Query interval touches  $[5, 8]$ .

$$|Q| = 27 + 13 + 19/4 \approx 45 .$$

Actual:  $|Q| = 48$

What would be the cardinality under the uniformity assumption  
(no histogram)?

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup, \setminus, \times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## Equi-Width Histogram: Construction

- To **construct** an equi-width histogram for relation  $R$ , attribute A:
  - ① Compute boundaries  $b_i$  from  $\text{High}(A, R)$  and  $\text{Low}(A, R)$ .
  - ② Scan  $R$  once sequentially.
  - ③ While scanning, maintain  $B$  running tuple frequency counters, one for each bucket.
- If scanning  $R$  in step ② is prohibitive, scan small sample  $R_{sample} \subset R$ , then scale frequency counters by  $|R|/|R_{sample}|$ .
- To **Maintain** the histogram under insertions (deletions):
  - ① Simply increment (decrement) frequency counter in affected bucket.

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

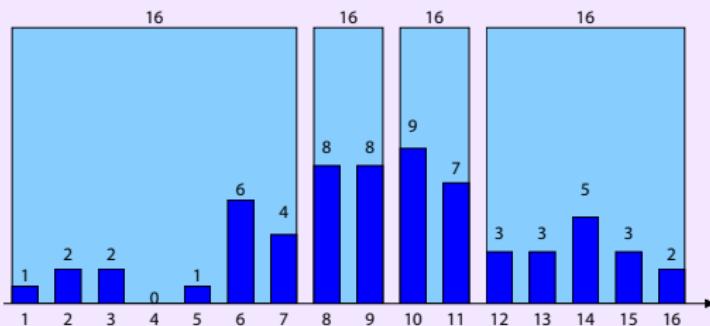
Statistical Views

## Equi-Depth Histograms

- Divide **active domain** of attribute A into  $B$  buckets of roughly the same number of tuples in each bucket, **depth**  $d$  of each bucket will be

$$d = \frac{|R|}{B} .$$

### Example (Equi-depth histogram ( $B = 4, d = 16$ ))



- Maintain **depth** (and bucket boundaries  $b_i$ ).

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup, \setminus, \times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

# Equi-Depth Histograms

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator  
Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

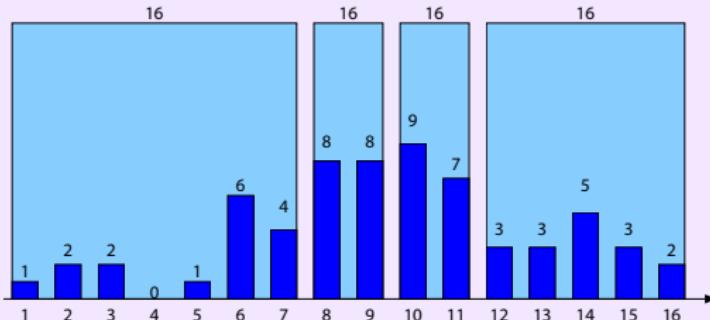
Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

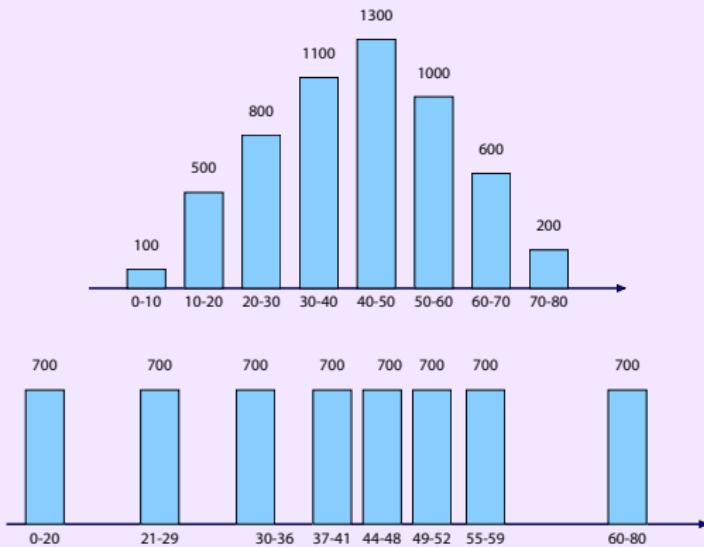


## Intuition:

- High value frequencies are more important than low value frequencies.
- Resolution of histogram adapts to skewed value distributions.

## Equi-Width vs. Equi-Depth Histograms

Example (Histogram on *customer age* attribute ( $B = 8, |R| = 5,600$ ))



- Equi-depth histogram “invests” bytes in the densely populated customer age region between 30 and 59.

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup, \setminus, \times$

Join  $\bowtie$

Histograms

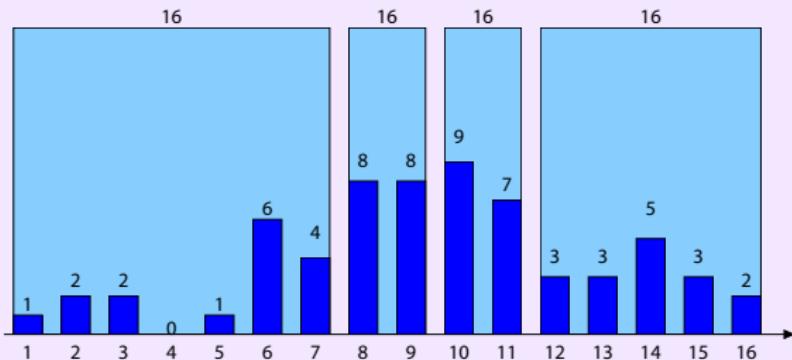
Equi-Width

Equi-Depth

Statistical Views

## Equi-Depth Histograms: Equality Selections

Example ( $Q \equiv \sigma_{A=5}(R)$ )



- Value 5 is in first bucket [1, 7] (with  $d = 16$  tuples)
- Assume **uniform distribution within the bucket**:

$$|Q| = d/7 = 16/7 \approx 2 .$$

(Actual:  $|Q| = 1$ )

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

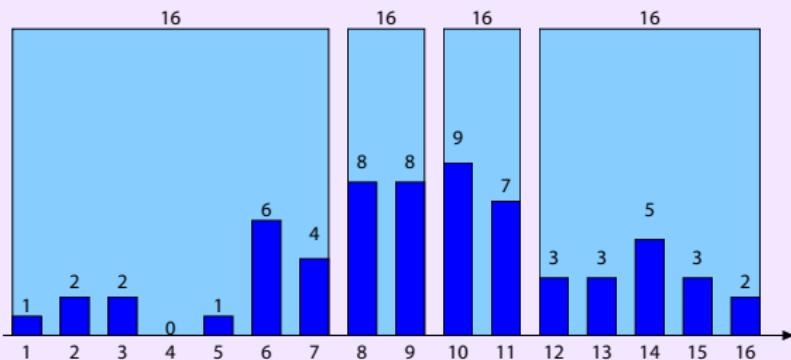
Equi-Width

Equi-Depth

Statistical Views

## Equi-Depth Histograms: Range Selections

Example ( $Q \equiv \sigma_{A>5} \text{ AND } A \leqslant 16(R)$ )



- Query interval  $(5, 16]$  covers buckets  $[8, 9]$ ,  $[10, 11]$  and  $[12, 16]$  (all with  $d = 16$  tuples). Query interval touches  $[1, 7]$ .

$$|Q| = 16 + 16 + 16 + 2/7 \cdot 16 \approx 53 .$$

(Actual:  $|Q| = 59$ )

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup, \setminus, \times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## Equi-Depth Histograms: Construction

- To **construct** an equi-depth histogram for relation  $R$ , attribute A:
  - Compute depth  $d = |R|/B$ .
  - Sort  $R$  by sort criterion A.
  - $b_0 = \text{Low}(A, R)$ , then determine the  $b_i$  by dividing the sorted  $R$  into chunks of size  $d$ .

### Example ( $B = 4, |R| = 64$ )

- $d = 64/4 = 16$ .
- Sorted R.A:  
 $\langle 1, 2, 2, 3, 3, 5, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, \dots \rangle$
- Boundaries of  $d$ -sized chunks in sorted  $R$ :

$\underbrace{\langle 1, 2, 2, 3, 3, 5, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, \dots \rangle}_{b_1=7}$        $\underbrace{\langle 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, \dots \rangle}_{b_2=9}$

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup, \setminus, \times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## A Cardinality (Mis-)Estimation Scenario

- Because exact cardinalities and estimated selectivity information is provided for base tables only, the DBMS relies on **projected cardinalities** for derived tables.
- In the case of foreign key joins, IBM DB2 promotes selectivity factors for one join input to the join result, for example.

### Example (Selectivity promotion; K is key of S, $\pi_A(R) \subseteq \pi_K(S)$ )

$$R \bowtie_{R.A=S.K} (\sigma_{B=10}(S))$$

If  $sel(B = 10) = x$ , then assume that the join will yield  $x \cdot |R|$  rows.

- Whenever the value distribution of A in R does not match the distribution of B in S, the cardinality estimate may be severely off.

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

# A Cardinality (Mis-)Estimation Scenario

## Example (Excerpt of a data warehouse)

Dimension table STORE:

STOREKEY	STORE_NUMBER	CITY	STATE	DISTRICT
...	...	...	...	...
...	...	...	...	...

} 63 rows

Dimension table PROMOTION:

PROMOKEY	PROMOTYPE	PROMODESC	PROMOVOLUME
...	...	...	...
...	...	...	...

} 35 rows

Fact table DAILY\_SALES:

STOREKEY	CUSTKEY	PROMOKEY	SALES_PRICE
...	...	...	...
...	...	...	...

} 754 069 426 rows

Let the tables be arranged in a **star schema**:

- The **fact table** references the **dimension tables**,
- the dimension tables are small/stable, the fact table is large/continuously updated on each sale.
- ⇒ Histograms are maintained for the dimension tables.

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator  
Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

## A Cardinality (Mis-)Estimation Scenario

### DB2. Query against the data warehouse

Find the number of those sales in store '01' (18 of the overall 63 locations) that were the result of the sales promotion of type 'XMAS' (**"star join"**):

```
1  SELECT COUNT(*)
2  FROM   STORE d1, PROMOTION d2, DAILY_SALES f
3  WHERE  d1.STOREKEY = f.STOREKEY
4  AND    d2.PROMOKEY = f.PROMOKEY
5  AND    d1.STORE_NUMBER = '01'
6  AND    d2.PROMOTYPE = 'XMAS'
```

The query yields 12,889,514 rows. The histograms lead to the following selectivity estimates:

$$\begin{aligned} \text{sel}(\text{STORE\_NUMBER} = '01') &= 18/63 \quad (28.57\%) \\ \text{sel}(\text{PROMOTYPE} = 'XMAS') &= 1/35 \quad (2.86\%) \end{aligned}$$

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator  
Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

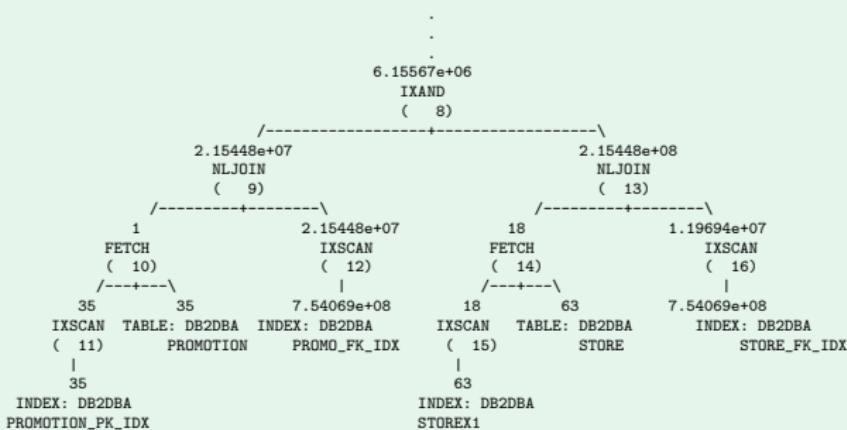
Statistical Views

## A Cardinality (Mis-)Estimation Scenario

**DB2. Estimated cardinalities and selected plan**

```
1  SELECT COUNT(*)
2  FROM   STORE d1, PROMOTION d2, DAILY_SALES f
3  WHERE  d1.STOREKEY = f.STOREKEY
4  AND    d2.PROMOKEY = f.PROMOKEY
5  AND    d1.STORE_NUMBER = '01'
6  AND    d2.PROMOTYPE = 'XMAS'
```

Plan fragment (top numbers indicates estimated cardinality):



## IBM DB2: Statistical Views

- To provide database profile information (estimate cardinalities, value distributions, ...) for **derived tables**:
  - Define a **view** that precomputes the derived table (or possibly a small sample of it, IBM DB2: 10 %),
  - use the view result to gather and keep **statistics**, then delete the result.

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views

```
1 CREATE VIEW sv_store_dailysales AS
2   (SELECT s.* 
3     FROM STORE s, DAILY_SALES ds
4    WHERE s.STOREKEY = ds.STOREKEY);
5
6 CREATE VIEW sv_promotion_dailysales AS
7   (SELECT p.* 
8     FROM PROMOTION p, DAILY_SALES ds
9    WHERE p.PROMOKEY = ds.PROMOKEY);
10
11 ALTER VIEW sv_store_dailysales ENABLE QUERY OPTIMIZATION;
12 ALTER VIEW sv_promotion_dailysales ENABLE QUERY OPTIMIZATION;
13
14 RUNSTATS ON TABLE sv_store_dailysales WITH DISTRIBUTION;
15 RUNSTATS ON TABLE sv_promotion_dailysales WITH DISTRIBUTION;
```

# Cardinality Estimation with Statistical Views

## DB2. Estimated cardinalities and selected plan after reoptimization

```
1.04627e+07
IXAND
( 8)
-----
6.99152e+07          1.2845e+08
NLJOIN                NLJOIN
( 9)                  ( 13)
-----
18                   3.88418e+06          1           1.12845e+08
FETCH                IXSCAN              FETCH        IXSCAN
( 10)                 ( 12)               ( 14)        ( 16)
-----
18       63           7.54069e+08          35          35           7.54069e+08
IXSCAN   TABLE:DB2DBA INDEX: DB2DBA  IXSCAN   TABLE: DB2DBA INDEX: DB2DBA DB2DBA
( 11)      STORE      STORE_FK_IDX    ( 15)     PROMOTION  PROMO_FK_IDX
|                               |
63
INDEX: DB2DBA
STOREX1
INDEX: DB2DBA
PROMOTION_PK_IDX
```

### Note new estimated selectivities *after join*:

- Selectivity of PROMOTYPE = 'XMAS' now only 14.96 %  
(was: 2.86 %)
- Selectivity of STORE\_NUMBER = '01' now 9.27 %  
(was: 28.57 %)

Cardinality Estimation

Torsten Grust



Cardinality Estimation

Database Profiles

Assumptions

Estimating Operator  
Cardinality

Selection  $\sigma$

Projection  $\pi$

Set Operations  $\cup$ ,  $\setminus$ ,  $\times$

Join  $\bowtie$

Histograms

Equi-Width

Equi-Depth

Statistical Views



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

# Chapter 10

## Query Optimization

Exploring the Search Space of Alternative Query Plans

*Architecture and Implementation of Database Systems*  
Summer 2014

Torsten Grust  
Wilhelm-Schickard-Institut für Informatik  
Universität Tübingen

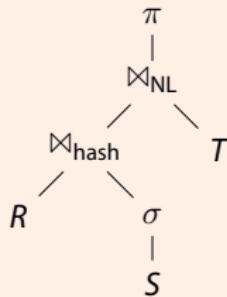


## Finding the “Best” Query Plan

Throttle or break?

SELECT ...  
FROM ...  
WHERE ...

?



- We already saw that there may be more than one way to answer a given query.
  - Which one of the join operators should we pick? With which parameters (block size, buffer allocation, ...)?
- The task of **finding the best execution plan** is, in fact, the “holy grail” of any database implementation.

Query Optimization

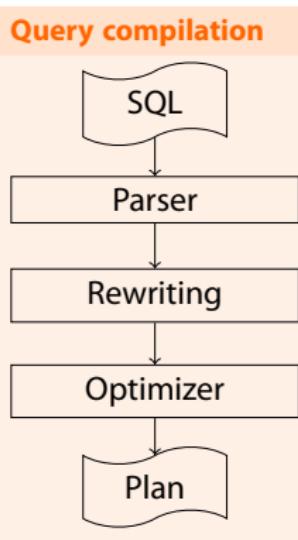
Torsten Grust



Query Optimization

Search Space Illustration  
Dynamic Programming  
Example: Four-Way Join  
Algorithm  
Discussion  
Left/Right-Deep vs. Bushy  
Greedy join enumeration

## Plan Generation Process



- **Parser:** syntactical/semantical analysis
- **Rewriting: heuristic** optimizations independent of the current database state (table sizes, availability of indexes, etc.). For example:
  - Apply predicates early
  - Avoid unnecessary duplicate elimination
- **Optimizer:** optimizations that rely on a **cost model** and information about the current database state
- The resulting **plan** is then evaluated by the system's **execution engine**.

Query Optimization

Torsten Grust



Query Optimization

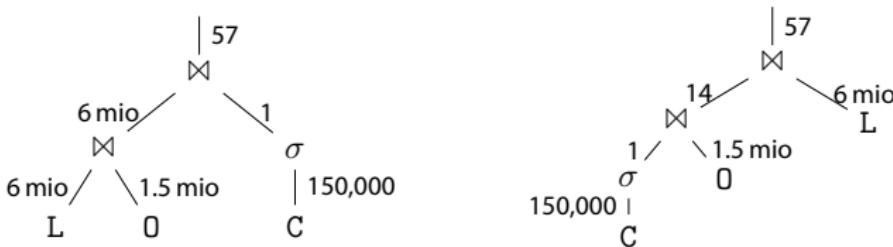
- Search Space Illustration
- Dynamic Programming
- Example: Four-Way Join
- Algorithm
- Discussion
- Left/Right-Deep vs. Bushy
- Greedy join enumeration

## Impact on Performance

Finding the right plan can dramatically impact performance.

### Sample query over TPC-H tables

```
1 SELECT L.L_PARTKEY, L.L_QUANTITY, L.L_EXTENDEDPRICE  
2   FROM LINEITEM L, ORDERS O, CUSTOMER C  
3 WHERE L.L_ORDERKEY = O.O_ORDERKEY  
4   AND O.O_CUSTKEY = C.C_CUSTKEY  
5   AND C.C_NAME = 'IBM Corp.'
```



- In terms of execution times, these differences can easily mean "seconds versus days."

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

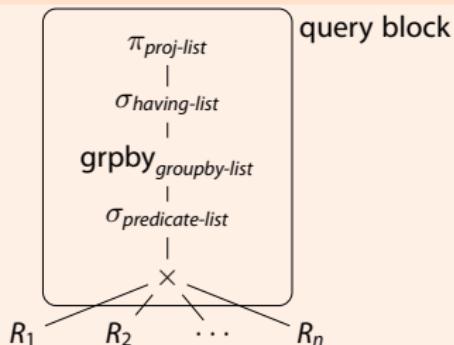
## The SQL Parser

- Besides some analyses regarding the syntactical and semantical correctness of the input query, the parser creates an **internal representation** of the input query.
- This representation still resembles the original query:
  - Each SELECT-FROM-WHERE clause is translated into a **query block**.

### Deriving a query block from a SQL SFW block

```
SELECT proj-list
      FROM  $R_1, R_2, \dots, R_n$ 
 WHERE predicate-list
 GROUP BY groupby-list
 HAVING having-list
```

→



- Each  $R_i$  can be a base relation or another query block.

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration  
Dynamic Programming  
Example: Four-Way Join  
Algorithm  
Discussion  
Left/Right-Deep vs. Bushy  
Greedy join enumeration

## Finding the “Best” Execution Plan

The parser output is fed into a **rewrite engine** which, again, yields a tree of query blocks.

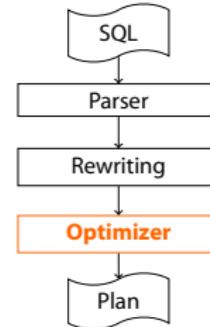
It is then the **optimizer’s** task to come up with the optimal **execution plan** for the given query.

Essentially, the optimizer

- ① **enumerates** all possible execution plans,  
(if this yields too many plans, at least enumerate the “promising”  
plan candidates)
- ② **determines** the **quality** (cost) of each plan, then
- ③ **chooses** the best one as the final execution plan.

Before we can do so, we need to answer the question

- What is a “good” execution plan at all?



Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

## Cost Metrics

Database systems judge the quality of an execution plan based on a number of **cost factors**, e.g.,

- the number of **disk I/Os** required to evaluate the plan,
- the plan's **CPU cost**,
- the overall **response time** observable by the database client as well as the total **execution time**.

A cost-based optimizer tries to **anticipate** these costs and find the cheapest plan before actually running it.

- All of the above factors depend on one critical piece of information: the **size of (intermediate) query results**.
- Database systems, therefore, spend considerable effort into accurate **result size estimates**.

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

## Result Size Estimation

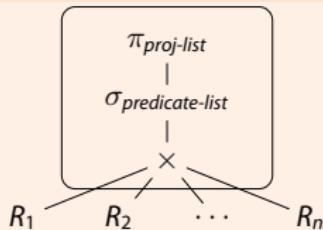
Query Optimization

Torsten Grust



Consider a query block corresponding to a simple SFW query  $Q$ .

### SFW query block



### Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

We can estimate the result size of  $Q$  based on

- the size of the input tables,  $|R_1|, \dots, |R_n|$ , and
- the **selectivity**  $sel(p)$  of the predicate  $predicate-list$ :

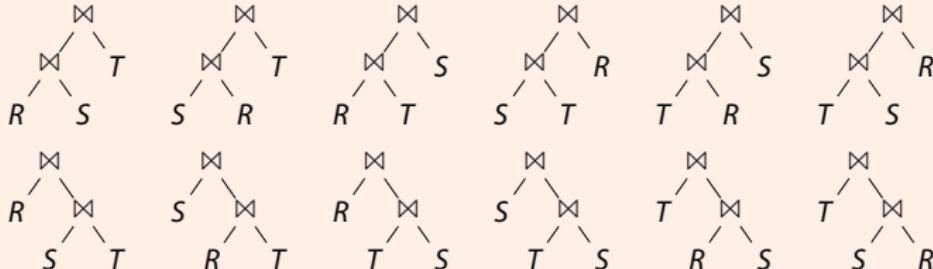
$$|Q| \approx |R_1| \cdot |R_2| \cdots |R_n| \cdot sel(predicate-list) .$$

## Join Optimization

- We've now translated the query into a graph of **query blocks**.
  - Query blocks essentially are a **multi-way** Cartesian product with a number of selection predicates on top.
- We can estimate the **cost** of a given **execution plan**.
  - Use result size estimates in combination with the cost for individual join algorithms discussed in previous chapters.

We are now ready to **enumerate** all possible execution plans, *i.e.*, all possible **2-way** join combinations for each query block.

### Ways of building a 3-way join from two 2-way joins



Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

## How Many Such Combinations Are There?

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

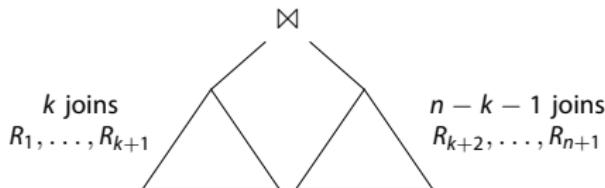
Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

- 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):

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1} .$$

## Catalan Numbers

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

This recurrence relation is satisfied by **Catalan numbers**:

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1} = \frac{(2n)!}{(n+1)!n!} ,$$

describing the number of ordered binary trees with  $n + 1$  leaves.

For **each** of these trees, we can **permute** the input relations  
(why?)  $R_1, \dots, R_{n+1}$ , leading to:

**Number of possible join trees for an  $(n + 1)$ -way relational join**

$$\frac{(2n)!}{(n+1)!n!} \cdot (n+1)! = \frac{(2n)!}{n!}$$

## Search Space

Query Optimization

Torsten Grust



The resulting search space is **enormous**:

### Possible bushy join trees joining $n$ relations

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

Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

- And we haven't yet even considered the use of  $k$  **different join algorithms** (yielding another factor of  $k^{(n-1)}$ )!

## Dynamic Programming

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

The traditional approach to master this search space is the use of **dynamic programming**.

Idea:

- Find the cheapest plan for an  $n$ -way join in  $n$  **passes**.
- In each pass  $k$ , find the best plans for all  $k$ -relation **sub-queries**.
- **Construct** the plans in pass  $k$  from best  $i$ -relation and  $(k - i)$ -relation sub-plans found in **earlier passes** ( $1 \leq i < k$ ).

Assumption:

- To find the optimal **global plan**, it is sufficient to only consider the optimal plans of its **sub-queries** ("Principle of optimality").

# Dynamic Programming

## Example (Four-way join of tables $R_1, \dots, R_4$ )

### Pass 1 (best 1-relation plans)

Find the best **access path** to each of the  $R_i$  individually  
(considers index scans, full table scans).

### Pass 2 (best 2-relation plans)

For each **pair** of tables  $R_i$  and  $R_j$ , determine the best order to join  $R_i$  and  $R_j$  (use  $R_i \bowtie R_j$  or  $R_j \bowtie R_i$ ?):

$$optPlan(\{R_i, R_j\}) \leftarrow \text{best of } R_i \bowtie R_j \text{ and } R_j \bowtie R_i .$$

→ 12 plans to consider.

### Pass 3 (best 3-relation plans)

For each **triple** of tables  $R_i$ ,  $R_j$ , and  $R_k$ , determine the best three-table join plan, using sub-plans obtained so far:

$$\begin{aligned} optPlan(\{R_i, R_j, R_k\}) \leftarrow & \text{best of } R_i \bowtie optPlan(\{R_j, R_k\}), \\ & optPlan(\{R_j, R_k\}) \bowtie R_i, \quad R_j \bowtie optPlan(\{R_i, R_k\}), \dots . \end{aligned}$$

→ 24 plans to consider.

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

## Dynamic Programming

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

### Example (Four-way join of tables $R_1, \dots, R_4$ (cont'd))

#### Pass 4 (best 4-relation plan)

For each set of **four** tables  $R_i, R_j, R_k$ , and  $R_l$ , determine the best four-table join plan, using sub-plans obtained so far:

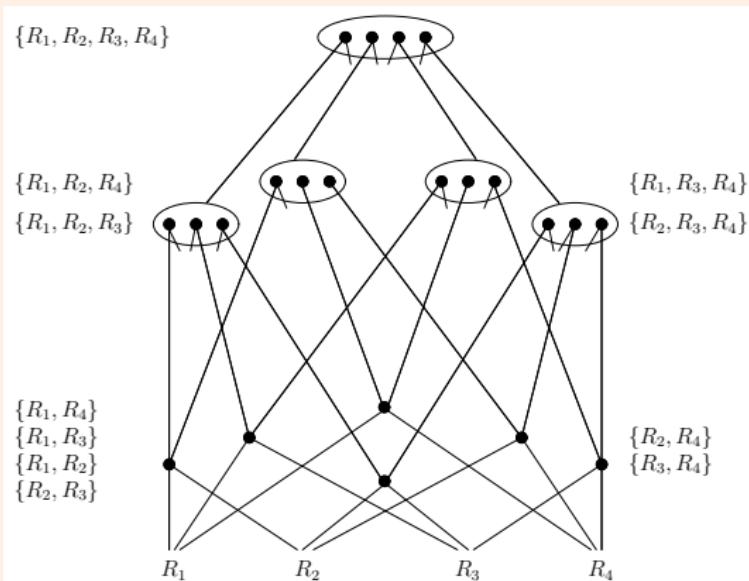
$$\begin{aligned} \text{optPlan}(\{R_i, R_j, R_k, R_l\}) &\leftarrow \text{best of } R_i \bowtie \text{optPlan}(\{R_j, R_k, R_l\}), \\ \text{optPlan}(\{R_j, R_k, R_l\}) \bowtie R_i, \quad R_j \bowtie \text{optPlan}(\{R_i, R_k, R_l\}), \dots, \\ \text{optPlan}(\{R_i, R_j\}) \bowtie \text{optPlan}(\{R_k, R_l\}), \dots . \end{aligned}$$

→ 14 plans to consider.

- Overall, we looked at only **50** (sub-)plans (instead of the possible 120 four-way join plans; ↗ slide 12).
- All decisions required the evaluation of **simple** sub-plans only (**no need to re-evaluate**  $\text{optPlan}(\cdot)$  for already known relation combinations ⇒ use lookup table).

# Sharing Under the Optimality Principle

## Sharing optimal sub-plans



Drawing by Guido Moerkotte, U Mannheim

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

# Dynamic Programming Algorithm

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

```
1 Function: find_join_tree_dp ( $q(R_1, \dots, R_n)$ )
2   for  $i = 1$  to  $n$  do
3      $optPlan(\{R_i\}) \leftarrow access\_plans(R_i)$  ;
4     prune_plans ( $optPlan(\{R_i\})$ ) ;
5   for  $i = 2$  to  $n$  do
6     foreach  $S \subseteq \{R_1, \dots, R_n\}$  such that  $|S| = i$  do
7        $optPlan(S) \leftarrow \emptyset$  ;
8       foreach  $O \subset S$  with  $O \neq \emptyset$  do
9          $optPlan(S) \leftarrow optPlan(S) \cup$ 
10        possible_joins  $\left[ optPlan(O) \bowtie optPlan(S \setminus O) \right]$  ;
11     prune_plans ( $optPlan(S)$ ) ;
12   return  $optPlan(\{R_1, \dots, R_n\})$  ;
```

- `possible_joins [R  $\bowtie$  S]` enumerates the possible joins between  $R$  and  $S$  (nested loops join, merge join, etc.).
- `prune_plans (set)` discards all but the best plan from set.

## Dynamic Programming—Discussion

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

- Enumerate all non-empty true subsets of  $S$  (using C):

```
1     O = S & -S;  
2     do {  
3         /* perform operation on O */  
4         O = S & (O - S);  
5     } while (O != S);
```

- `find_join_tree_dp()` draws its advantage from **filtering** plan candidates early in the process.
  - In our example on slide 14, pruning in Pass 2 reduced the search space by a factor of 2, and another factor of 6 in Pass 3.
- Some **heuristics** can be used to prune even more plans:
  - Try to avoid **Cartesian products**.
  - Produce **left-deep plans** only (see next slides).
- Such heuristics can be used as a handle to balance plan quality and optimizer runtime.

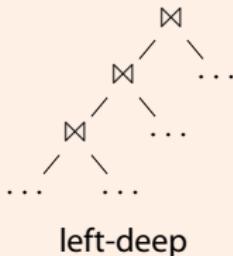
### DB2. Control optimizer investment

```
1     SET CURRENT QUERY OPTIMIZATION = n
```

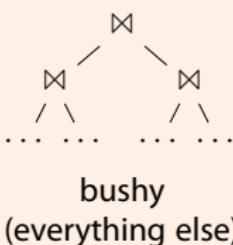
## Left/Right-Deep vs. Bushy Join Trees

The algorithm on slide 17 explores all possible shapes a join tree could take:

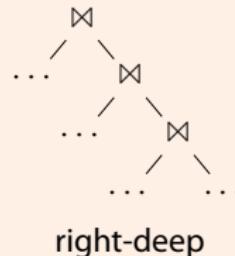
### Join tree shapes



left-deep



bushy  
(everything else)



right-deep

Actual systems often prefer **left-deep** join trees.<sup>1</sup>

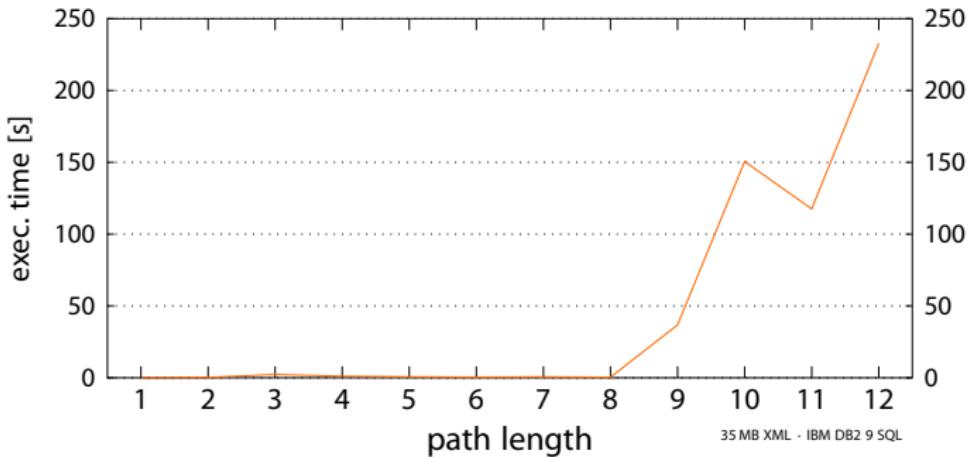
- The **inner** (rhs) relation always is a **base relation**.
- Allows the use of **index nested loops join**.
- Easier to implement in a **pipelined** fashion.



<sup>1</sup>The seminal **System R** prototype, e.g., considered only left-deep plans.

## Join Order Makes a Difference

- XPath location step evaluation over relationally encoded XML data.<sup>2</sup>
- $n$ -way self-join with a range predicate.



Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

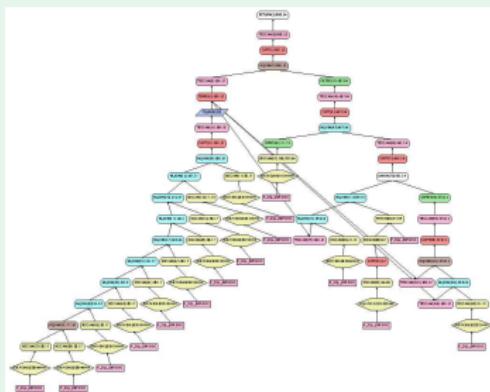
Greedy join enumeration

<sup>2</sup> ↗ Grust et al. Accelerating XPath Evaluation in Any RDBMS. *TODS* 2004.  
<http://www.pathfinder-xquery.org/>

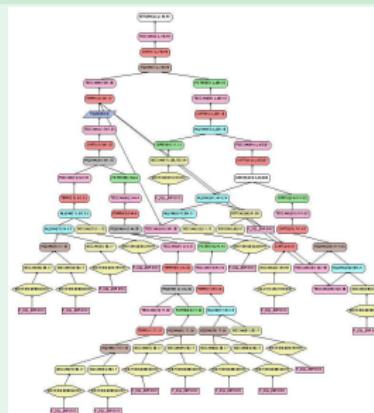
## Join Order Makes a Difference

Contrast the execution plans for a path of 8 and 9 XPath location steps:

## DB2® Join plans



## left-deep join tree



### bushy join tree

⇒ DB2's optimizer essentially gave up in the face of 9+ joins.



## Joining Many Relations

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

Dynamic programming still has **exponential** resource requirements:

(↗ K. Ono, G.M. Lohman, *Measuring the Complexity of Join Enumeration in Query Optimization*, VLDB 1990)

- time complexity:  $\mathcal{O}(3^n)$
- space complexity:  $\mathcal{O}(2^n)$

This may still be too expensive

- for joins involving many relations ( $\sim 10\text{--}20$  and more),
- for simple queries over well-indexed data (where the right plan choice should be easy to make).

The **greedy join enumeration** algorithm jumps into this gap.

## Greedy Join Enumeration

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

```
1 Function: find_join_tree_greedy ( $q(R_1, \dots, R_n)$ )
2   worklist  $\leftarrow \emptyset$  ;
3   for  $i = 1$  to  $n$  do
4     worklist  $\leftarrow$  worklist  $\cup$  best_access_plan ( $R_i$ ) ;
5   for  $i = n$  downto 2 do
6     // worklist =  $\{P_1, \dots, P_i\}$ 
7     find  $P_j, P_k \in$  worklist and  $\bowtie\dots$  such that  $\text{cost}(P_j \bowtie\dots P_k)$  is minimal ;
8     worklist  $\leftarrow$  worklist  $\setminus \{P_j, P_k\} \cup \{(P_j \bowtie\dots P_k)\}$  ;
// worklist =  $\{P_1\}$ 
8 return single plan left in worklist ;
```

- In each iteration, choose the **cheapest** join that can be made over the remaining sub-plans at that time (this is the “greedy” part).
- Observe that `find_join_tree_greedy()` operates similar to finding the optimum binary tree for **Huffman coding**.

## Join Enumeration—Discussion

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

### Greedy join enumeration:

- The greedy algorithm has  $\mathcal{O}(n^3)$  time complexity:
  - The loop has  $\mathcal{O}(n)$  iterations.
  - Each iteration looks at all remaining pairs of plans in *worklist*. An  $\mathcal{O}(n^2)$  task.

### Other join enumeration techniques:

- **Randomized algorithms:** randomly rewrite the join tree one rewrite at a time; use **hill-climbing** or **simulated annealing** strategy to find optimal plan.
- **Genetic algorithms:** explore plan space by **combining** plans (“creating offspring”) and **altering** some plans randomly (“mutations”).

## Physical Plan Properties

Query Optimization

Torsten Grust



Consider the simple equi-join query

### Join query over TPC-H tables

```
1 SELECT O.O_ORDERKEY  
2   FROM ORDERS O, LINEITEM L  
3 WHERE O.O_ORDERKEY = L.L_ORDERKEY
```

where table ORDERS is indexed with a **unclustered index**  $OK\_IDX$  on column  $O\_ORDERKEY$ .

Possible table access plans (1-relation plans) are:

- ORDERS**
  - **full table scan**: estimated I/Os:  $N_{ORDERS}$
  - **index scan**: estimated I/Os:  $N_{OK\_IDX} + N_{ORDERS}$ .
- LINEITEM**
  - **full table scan**: estimated I/Os:  $N_{LINEITEM}$ .

Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

## Physical Plan Properties

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

- Since the **full table scan** is the cheapest access method for both tables, our join algorithms will select them as the best 1-relation plans in Pass 1.<sup>3</sup>

To **join** the two scan outputs, we now have the choices

- nested loops join,**
- hash join,** or
- sort** both inputs, then use **merge join.**

Let us assume that sort-merge join is the preferable candidate, incurring a cost of  $\approx 2 \cdot (N_{\text{ORDERS}} + N_{\text{LINEITEM}})$ .

⇒ **Overall cost:**

$$N_{\text{ORDERS}} + N_{\text{LINEITEM}} + 2 \cdot (N_{\text{ORDERS}} + N_{\text{LINEITEM}}).$$

---

<sup>3</sup>Dynamic programming and the greedy algorithm happen to do the same in this example.

## Physical Plan Properties—A Better Plan

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

It is easy to see, however, that there is a better way to evaluate the query:

- ① Use an **index scan** to access ORDERS. This guarantees that the scan output is already **in O\_ORDERKEY order**.
- ② Then only **sort** LINEITEM and
- ③ join using **merge join**.

⇒ **Overall cost:**  $\underbrace{(N_{OK\_IDX} + N_{ORDERS})}_{1} + \underbrace{2 \cdot N_{LINEITEM}}_{2 / 3}$

Although more expensive as a standalone table access plan, the **use of the index (order enforcement) pays off later on** in the overall plan.

## Physical Plan Properties: Interesting Orders

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

- The advantage of the index-based access to ORDERS is that it provides beneficial **physical properties**.
  - Optimizers, therefore, keep track of such properties by **annotating** candidate plans.
  - System R introduced the concept of **interesting orders**, determined by
    - ORDER BY or GROUP BY clauses in the input query, or
    - join attributes of subsequent joins ( $\sim$  merge join).
- ⇒ In `prune_plans()`, retain
- the cheapest “unordered” plan **and**
  - the cheapest plan for each interesting order.

## Query Rewriting

- Join optimization essentially takes a set of relations and a set of join predicates to find the best join order.
- By **rewriting** query graphs beforehand, we can improve the effectiveness of this procedure.
- The **query rewriter** applies **heuristic rules**, without looking into the actual database state (no information about cardinalities, indexes, etc.).  
In particular, the optimizer
  - relocates predicates** (predicate pushdown),
  - rewrites predicates**, and
  - unnests queries**.

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

# Predicate Simplification

Rewrite

## Example (Query against TPC-H table)

```
1   SELECT *
2     FROM LINEITEM L
3    WHERE L.L_TAX * 100 < 5
```

into

## Example (Query after predicate simplification)

```
1   SELECT *
2     FROM LINEITEM L
3    WHERE L.L_TAX < 0.05
```

📎 In which sense is the rewritten predicate simpler?

Why would a RDBMS query optimizer rewrite the selection predicate as shown above?

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

# Introducing Additional Join Predicates

Query Optimization

Torsten Grust



Implicit join predicates as in

## Implicit join predicate through transitivity

```
1 SELECT *
2   FROM A, B, C
3 WHERE A.a = B.b AND B.b = C.c
```

can be turned into explicit ones:

## Explicit join predicate

```
1 SELECT *
2   FROM A, B, C
3 WHERE A.a = B.b AND B.b = C.c
4   AND A.a = C.c
```

This makes the following join tree feasible:

$$(A \bowtie C) \bowtie B .$$

(Note:  $(A \bowtie C)$  would have been a Cartesian product before.)

Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

## Nested Queries and Correlation

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

SQL provides a number of ways to write **nested queries**.

- **Uncorrelated** sub-query:

### No free variables in subquery

```
1 SELECT *
2   FROM ORDERS O
3 WHERE O.CUSTKEY IN (SELECT C.CUSTKEY
4                       FROM CUSTOMER
5                      WHERE C_NAME = 'IBM Corp.')
```

- **Correlated** sub-query:

### Row variable O occurs free in subquery

```
1 SELECT *
2   FROM ORDERS O
3 WHERE O.O_CUSTKEY IN
4       (SELECT C.C_CUSTKEY
5        FROM CUSTOMER C
6       WHERE C.C_ACCTBAL < O.O_TOTALPRICE)
```

## Query Unnesting

- Taking query nesting literally might be **expensive**.
  - An uncorrelated query, e.g., need not be re-evaluated for every tuple in the outer query.
- Oftentimes, sub-queries are only used as a syntactical way to express a **join** (or a semi-join).
- The query rewriter tries to detect such situations and **make the join explicit**.
- This way, the sub-query can become part of the regular **join order optimization**.

### 📎 Turning correlation into joins

Reformulate the correlated query of slide 32 (use SQL syntax or relational algebra) to remove the correlation (and introduce a join).

↗ Won Kim. On Optimizing an SQL-like Nested Query. ACM TODS, vol. 7, no. 3, September 1982.

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

## Summary

Query Optimization

Torsten Grust



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

### Query Parser

Translates input query into (SFW-like) **query blocks**.

### Rewriter

Logical (database state-independent) optimizations;  
predicate simplification; query unnesting.

### (Join) Optimization

Find “best” query execution plan based on a **cost model**  
(considering I/O cost, CPU cost, …); data statistics  
(histograms); dynamic programming, greedy join  
enumeration; physical plan properties (interesting orders).

Database optimizers still are true pieces of art...

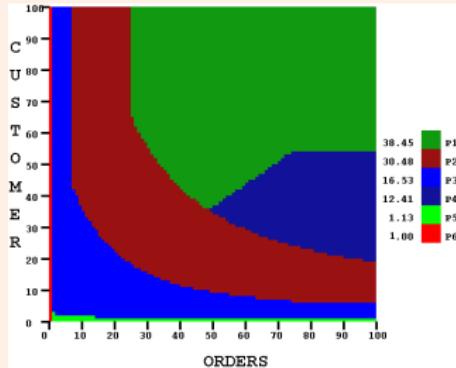
## "Picasso" Plan Diagrams

Query Optimization

Torsten Grust



Generated by "Picasso": SQL join query with filters of parameterizable selectivities (0 ... 100) against both join inputs



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

↗ Naveen Reddy and Jayant Haritsa. Analyzing Plan Diagrams of Database Query Optimizers. VLDB 2005.

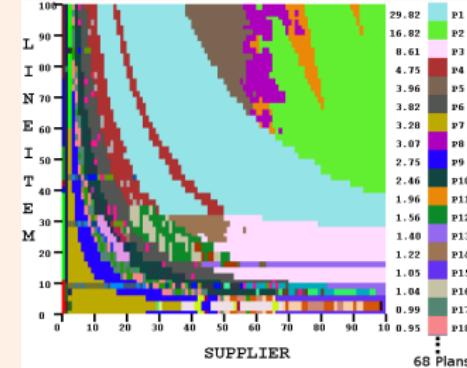
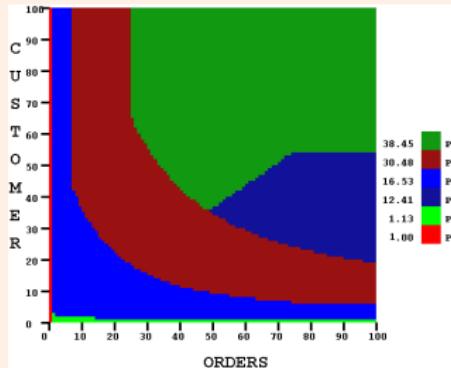
## "Picasso" Plan Diagrams

Query Optimization

Torsten Grust



Generated by "Picasso": SQL join query with filters of parameterizable selectivities (0 ... 100) against both join inputs



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

↗ Naveen Reddy and Jayant Haritsa. Analyzing Plan Diagrams of Database Query Optimizers. VLDB 2005.

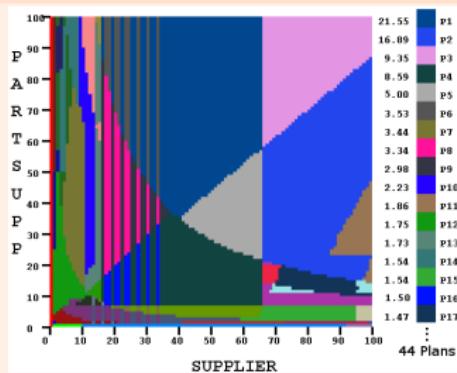
## "Picasso" Plan Diagrams

Query Optimization

Torsten Grust



Generated by "Picasso": each distinct color represent a distinct plan considered by the DBMS



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

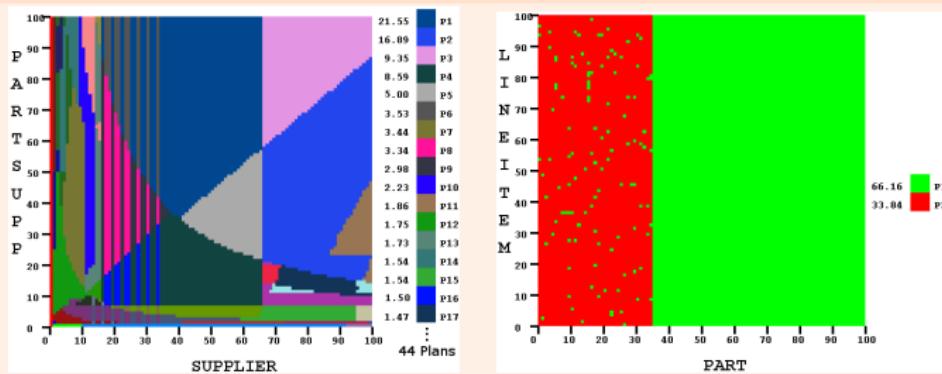
## "Picasso" Plan Diagrams

Query Optimization

Torsten Grust



Generated by "Picasso": each distinct color represent a distinct plan considered by the DBMS



Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

Discussion

Left/Right-Deep vs. Bushy

Greedy join enumeration

Download Picasso at

<http://dsl.serc.iisc.ernet.in/projects/PICASSO/index.html>.



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

# Chapter 11

## Transaction Management

### Concurrent and Consistent Data Access

*Architecture and Implementation of Database Systems*  
Summer 2014



# The “Hello World” of Transaction Management

- My bank issued me a debit card to access my account.
- Every once in a while, I'd use it at an ATM to draw some money from my account, causing the ATM to perform a **transaction** in the bank's database.

## Example (ATM transaction)

```
1 bal ← read_bal (acct_no) ;  
2 bal ← bal - 100 € ;  
3 write_bal (acct_no, bal) ;
```



- My account is **properly updated** to reflect the new balance.

Transaction  
Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic

Concurrency Protocol

Multi-Version

Concurrency Control

## Concurrent Access

The problem is: My wife has a card for the very same account, too.

- ⇒ We might end up using our cards at different ATMs at the **same time**, i.e., **concurrently**.

### Example (Concurrent ATM transactions)

me

$bal \leftarrow \text{read}(acct);$

$bal \leftarrow bal - 100;$

$\text{write}(acct, bal);$

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## Concurrent Access

The problem is: My wife has a card for the very same account, too.

- ⇒ We might end up using our cards at different ATMs at the **same time**, i.e., **concurrently**.

### Example (Concurrent ATM transactions)

me

```
bal ← read (acct) ;  
  
bal ← bal - 100 ;  
  
write (acct, bal) ;
```

my wife

```
bal ← read (acct) ;  
  
bal ← bal - 200 ;  
  
write (acct, bal) ;
```

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic

Concurrency Protocol

Multi-Version

Concurrency Control

## Concurrent Access

The problem is: My wife has a card for the very same account, too.

- ⇒ We might end up using our cards at different ATMs at the **same time**, i.e., **concurrently**.

### Example (Concurrent ATM transactions)

me	my wife	DB state
$bal \leftarrow \text{read}(acct);$		1200
	$bal \leftarrow \text{read}(acct);$	1200
$bal \leftarrow bal - 100;$		1200
	$bal \leftarrow bal - 200;$	1200
$\text{write}(acct, bal);$		1100
	$\text{write}(acct, bal);$	1000

- The first **update was lost** during this execution. Lucky me!

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic

Concurrency Protocol

Multi-Version

Concurrency Control

## If the Plug is Pulled ...

- This time, I want to **transfer** money over to another account.

### Example (Money transfer transaction)

```
// Subtract money from source (checking) account  
1 chk_bal ← read_bal(chk_acct_no);  
2 chk_bal ← chk_bal - 500 €;  
3 write_bal(chk_acct_no, chk_bal);  
  
// Credit money to the target (savings) account  
4 sav_bal ← read_bal(sav_acct_no);  
5 sav_bal ← sav_bal + 500 €;  
6 ↓  
7 write_bal(sav_acct_no, sav_bal);
```

- Before the transaction gets to step 7, its execution is **interrupted/cancelled** (power outage, disk failure, software bug, ...). My money is **lost** 😞.

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## ACID Properties

To prevent these (and many other) effects from happening, a DBMS guarantees the following **transaction properties**:

- A Atomicity** Either **all** or **none** of the updates in a database transaction are applied.
- C Consistency** Every transaction brings the database from one **consistent** state to another. (While the transaction executes, the database state may be temporarily inconsistent.)
- I Isolation** A transaction must not see any effect from other transactions that run in parallel.
- D Durability** The effects of a **successful** transaction remain persistent and may not be undone for system reasons.



### ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

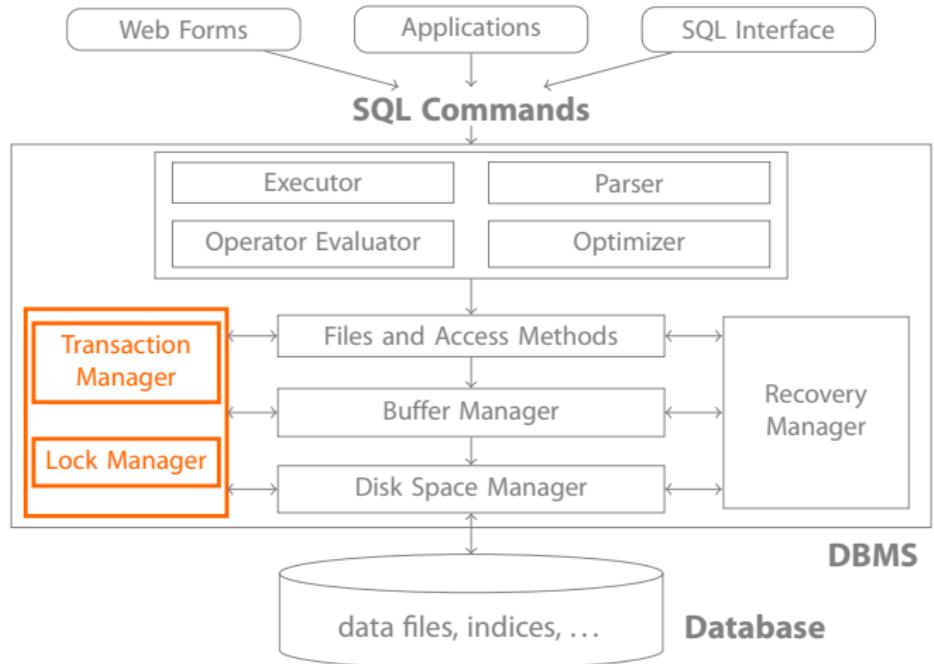
Optimistic

Concurrency Protocol

Multi-Version

Concurrency Control

# Concurrency Control



Transaction Management  
Torsten Grust



- ACID Properties
- Anomalies
- The Scheduler
- Serializability
- Query Scheduling
- Locking
- Two-Phase Locking
- Optimistic Concurrency Protocol
- Multi-Version Concurrency Control

## Anomalies: Lost Update

- We already saw an example of the **lost update** anomaly on slide 3:

The effects of one transaction are lost due to an uncontrolled overwrite performed by the second transaction.

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic

Concurrency Protocol

Multi-Version

Concurrency Control

## Anomalies: Inconsistent Read

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

Reconsider the money transfer example (slide 4), expressed in SQL syntax:

### Example

#### Transaction 1

```
1 UPDATE Accounts
2   SET balance = balance - 500
3 WHERE customer = 1904
4   AND account_type = 'C';
5
6 UPDATE Accounts
7   SET balance = balance + 500
8 WHERE customer = 1904
9   AND account_type = 'S';
```

#### Transaction 2

```
1 SELECT SUM(balance)
2   FROM Accounts
3 WHERE customer = 1904;
```

- Transaction 2 sees a temporary, **inconsistent** database state.

## Anomalies: Dirty Read

At a different day, my wife and me again end up in front of an ATM at roughly the same time. This time, my transaction is cancelled (**aborted**):

### Example

me	my wife	DB state
$bal \leftarrow \text{read}(acct);$		1200
$bal \leftarrow bal - 100;$		1200
$\text{write}(acct, bal);$		1100
	$bal \leftarrow \text{read}(acct);$	1100
	$bal \leftarrow bal - 200;$	1100



## Anomalies: Dirty Read

At a different day, my wife and me again end up in front of an ATM at roughly the same time. This time, my transaction is cancelled (**aborted**):

### Example

me	my wife	DB state
$bal \leftarrow \text{read}(acct);$		1200
$bal \leftarrow bal - 100;$		1200
$\text{write}(acct, bal);$		1100
	$bal \leftarrow \text{read}(acct);$	1100
	$bal \leftarrow bal - 200;$	1100
abort;		1200



## Anomalies: Dirty Read

At a different day, my wife and me again end up in front of an ATM at roughly the same time. This time, my transaction is cancelled (**aborted**):

### Example

me	my wife	DB state
$bal \leftarrow \text{read}(acct);$		1200
$bal \leftarrow bal - 100;$		1200
$\text{write}(acct, bal);$		1100
	$bal \leftarrow \text{read}(acct);$	1100
	$bal \leftarrow bal - 200;$	1100
abort;		1200
	$\text{write}(acct, bal);$	900

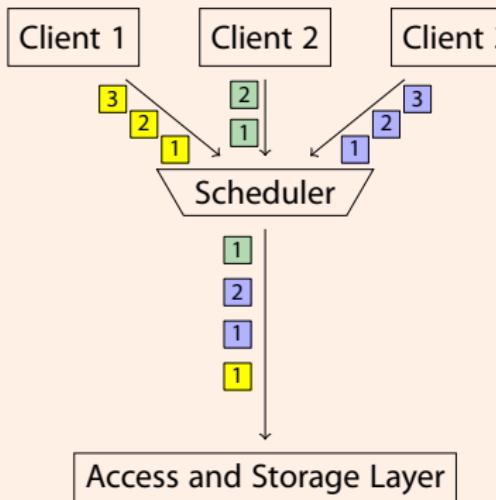
- My wife's transaction has already read the modified account balance before my transaction was **rolled back** (i.e., its effects are undone).



# Concurrent Execution

- The **scheduler** decides the execution order of concurrent database accesses.

## The transaction scheduler



Transaction Management  
Torsten Grust



ACID Properties

Anomalies

**The Scheduler**

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## Database Objects and Accesses

- We now assume a slightly simplified model of database access:
  - ① A database consists of a number of named **objects**. In a given database state, each object has a **value**.
  - ② Transactions access an object  $o$  using the two operations `read  $o$`  and `write  $o$` .
- In a **relational** DBMS we have that

object  $\equiv$  attribute .

This defines the **granularity** of our discussion. Other possible granularities:

object  $\equiv$  row, object  $\equiv$  table .

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

# Transactions

## Database transaction

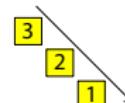
A **database transaction**  $T$  is a (strictly ordered) sequence of steps. Each **step** is a pair of an **access operation** applied to an **object**.

- Transaction  $T = \langle s_1, \dots, s_n \rangle$
- Step  $s_i = (a_i, e_i)$
- Access operation  $a_i \in \{r(\text{ead}), w(\text{rite})\}$

The **length** of a transaction  $T$  is its number of steps  $|T| = n$ .

We could write the money transfer transaction as

$$T = \langle (\text{read}, \textit{Checking}), (\text{write}, \textit{Checking}), \\ (\text{read}, \textit{Saving}), (\text{write}, \textit{Saving}) \rangle$$



or, more concisely,

$$T = \langle r(C), w(C), r(S), w(S) \rangle .$$

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

# Schedules

## Schedule

A **schedule**  $S$  for a given set of transactions  $\mathbf{T} = \{T_1, \dots, T_n\}$  is an arbitrary sequence of execution steps

$$S(k) = (T_j, a_i, e_i) \quad k = 1 \dots m ,$$



such that

- ①  $S$  contains all steps of all transactions and nothing else and
- ② the order among steps in each transaction  $T_j$  is preserved:

$$(a_p, e_p) < (a_q, e_q) \text{ in } T_j \Rightarrow (T_j, a_p, e_p) < (T_j, a_q, e_q) \text{ in } S$$

(read “ $<$ ” as: *occurs before*).

We sometimes write

$$S = \langle r_1(B), r_2(B), w_1(B), w_2(B) \rangle$$

to abbreviate

$$\begin{aligned} S(1) &= (T_1, \text{read}, B) & S(3) &= (T_1, \text{write}, B) \\ S(2) &= (T_2, \text{read}, B) & S(4) &= (T_2, \text{write}, B) \end{aligned}$$



# Serial Execution

## Serial execution

One particular schedule is **serial execution**.

- A schedule  $S$  is **serial** iff, for each contained transaction  $T_j$ , all its steps are adjacent (no interleaving of transactions and thus **no concurrency**).

Briefly:

$$S = T_{\pi 1}, T_{\pi 2}, \dots, T_{\pi n} \quad (\text{for some permutation } \pi \text{ of } 1, \dots, n)$$

Consider again the ATM example from slide 3.

- $S = \langle r_1(B), r_2(B), w_1(B), w_2(B) \rangle$
- This is a schedule, but it is **not** serial.



If my wife had gone to the bank one hour later (initiating transaction  $T_2$ ), the schedule probably would have been serial.

- $S = \langle r_1(B), w_1(B), r_2(B), w_2(B) \rangle$



Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## Correctness of Serial Execution

- Anomalies such as the “lost update” problem on slide 3 can **only** occur in multi-user mode.
  - If all transactions were fully executed one after another (no concurrency), no anomalies would occur.
- ⇒ **Any serial execution is correct.**
- Disallowing concurrent access, however, is **not practical**.

### Correctness criterion

Allow concurrent executions if their **overall effect is equivalent to an (arbitrary) serial execution**.

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

**Serializability**

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## Conflicts

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

What does it mean for a schedule  $S$  to be **equivalent** to another schedule  $S'$ ?

- Sometimes, we may be able to **reorder** steps in a schedule.
  - We must not change the order among steps of any transaction  $T_j$  ( $\nearrow$  slide 13).
  - Rearranging operations must not lead to a different **result**.
- Two operations  $(T_i, a, e)$  and  $(T_j, a', e')$  are said to be **in conflict**  $(T_i, a, e) \leftrightarrow (T_j, a', e')$  if their order of execution matters.
  - When reordering a schedule, we must not change the relative order of such operations.
- Any schedule  $S'$  that can be obtained this way from  $S$  is said to be **conflict equivalent** to  $S$ .

## Conflicts

Based on our read/write model, we can come up with a more machine-friendly definition of a conflict.

### Conflicting operations

Two operations  $(T_i, a, e)$  and  $(T_j, a', e')$  are **in conflict** ( $\leftrightarrow$ ) in  $S$  if

- ① they belong to two **different transactions** ( $T_i \neq T_j$ ), and
- ② they access the **same database object**, i.e.,  $e = e'$ , and
- ③ at least one of them is a write operation.

- This inspires the following **conflict matrix**:

	read	write
read		x
write	x	x

- **Conflict relation**  $\prec_S$ :

$$(T_i, a, e) \prec_S (T_j, a', e') \\ :=$$

$(T_i, a, e) \leftrightarrow (T_j, a', e') \wedge (T_i, a, e)$  occurs before  $(T_j, a', e')$  in  $S$

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic

Concurrency Protocol

Multi-Version

Concurrency Control

# Conflict Serializability

Transaction  
Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

**Serializability**

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## Conflict serializability

A schedule  $S$  is **conflict serializable** iff it is **conflict equivalent to some serial schedule  $S'$ .**

- The execution of a conflict-serializable  $S$  schedule is correct.
- Note:  $S$  does **not** have to be a serial schedule.



## Serializability: Example

Example (Three schedules  $S_i$  for two transactions  $T_{1,2}$ , with  $S_2$  serial)

Schedule $S_1$	
$T_1$	$T_2$
readA	
writeA	readA writeA

Schedule $S_2$	
$T_1$	$T_2$
readA	
writeA	readB writeB
readB	readA writeA
writeB	readB writeB

Schedule $S_2$	
$T_1$	$T_2$
readA	
writeA	readB writeB
readB	readA writeA
writeB	readB writeB

Schedule $S_3$	
$T_1$	$T_2$
readA	
writeA	readA writeA
readB	readB writeB
writeB	readB writeB

- Conflict relations:

$$\left. \begin{array}{l} (T_1, r, A) \prec_{S_1} (T_2, w, A), (T_1, r, B) \prec_{S_1} (T_2, w, B), \\ (T_1, w, A) \prec_{S_1} (T_2, r, A), (T_1, w, B) \prec_{S_1} (T_2, r, B), \\ (T_1, w, A) \prec_{S_1} (T_2, w, A), (T_1, w, B) \prec_{S_1} (T_2, w, B) \end{array} \right\} \Rightarrow S_1 \text{ serializable}$$

(Note:  $\prec_{S_2} = \prec_{S_1}$ )

$$\left. \begin{array}{l} (T_1, r, A) \prec_{S_3} (T_2, w, A), (T_2, r, B) \prec_{S_3} (T_1, w, B), \\ (T_1, w, A) \prec_{S_3} (T_2, r, A), (T_2, w, B) \prec_{S_3} (T_1, r, B), \\ (T_1, w, A) \prec_{S_3} (T_2, w, A), (T_2, w, B) \prec_{S_3} (T_1, w, B) \end{array} \right\}$$

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## The Conflict Graph

- The serializability idea comes with an effective test for the correctness of a schedule  $S$  based on its **conflict graph**  $G(S)$  (also: **serialization graph**):
  - The **nodes** of  $G(S)$  are all transactions  $T_i$  in  $S$ .
  - There is an **edge**  $T_i \rightarrow T_j$  iff  $S$  contains operations  $(T_i, a, e)$  and  $(T_j, a', e')$  such that  $(T_i, a, e) \prec_S (T_j, a', e')$  (read: *in a conflict equivalent serial schedule,  $T_i$  must occur before  $T_j$* ).
- $S$  is conflict serializable iff  $G(S)$  is **acyclic**.  
An equivalent serial schedule for  $S$  may be immediately obtained by sorting  $G(S)$  **topologically**.

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

**Serializability**

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

# Serialization Graph

## Example (ATM transactions (↗ slide 3))

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$
- Conflict relation:
  - $(T_1, r, A) \prec_S (T_2, w, A)$
  - $(T_2, r, A) \prec_S (T_1, w, A)$
  - $(T_1, w, A) \prec_S (T_2, w, A)$



Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## Serialization Graph

### Example (ATM transactions (↗ slide 3))

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$

- Conflict relation:

$$\begin{aligned}(T_1, r, A) &\prec_S (T_2, w, A) \\ (T_2, r, A) &\prec_S (T_1, w, A) \\ (T_1, w, A) &\prec_S (T_2, w, A)\end{aligned}$$



### Example (Two money transfers (↗ slide 4))

- $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$

- Conflict relation:

$$\begin{aligned}(T_1, r, C) &\prec_S (T_2, w, C) \\ (T_1, w, C) &\prec_S (T_2, r, C) \\ (T_1, w, C) &\prec_S (T_2, w, C)\end{aligned}$$

⋮



Transaction Management  
Torsten Grust



ACID Properties

Anomalies

The Scheduler

**Serializability**

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

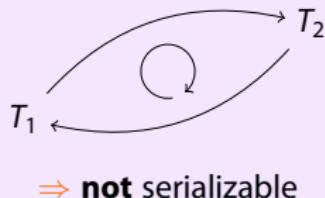
# Serialization Graph

## Example (ATM transactions (↗ slide 3))

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$

- Conflict relation:

$$\begin{aligned}(T_1, r, A) &\prec_S (T_2, w, A) \\ (T_2, r, A) &\prec_S (T_1, w, A) \\ (T_1, w, A) &\prec_S (T_2, w, A)\end{aligned}$$



## Example (Two money transfers (↗ slide 4))

- $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$

- Conflict relation:

$$\begin{aligned}(T_1, r, C) &\prec_S (T_2, w, C) \\ (T_1, w, C) &\prec_S (T_2, r, C) \\ (T_1, w, C) &\prec_S (T_2, w, C)\end{aligned}$$

⋮



Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

**Serializability**

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

# Query Scheduling

Transaction Management  
Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

Can we build a scheduler that **always** emits a serializable schedule?

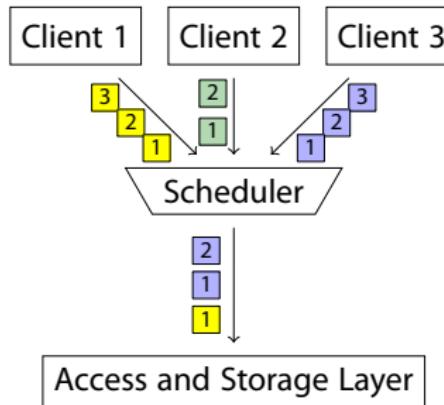
Idea:

- Require each transaction to obtain a **lock** before it accesses a data object  $o$ :

## Locking and unlocking of $o$

```
1 lock o;  
2 ...access o ...;  
3 unlock o;
```

- This prevents **concurrent** access to  $o$ .



## Locking

- If a lock cannot be granted (e.g., because another transaction  $T'$  already holds a **conflicting** lock) the requesting transaction  $T$  gets **blocked**.
- The scheduler **suspends** execution of the blocked transaction  $T$ .
- Once  $T'$  **releases** its lock, it may be granted to  $T$ , whose execution is then **resumed**.
- ⇒ Since other transactions can continue execution while  $T$  is blocked, locks can be used to **control the relative order of operations**.

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic

Concurrency Protocol

Multi-Version

Concurrency Control

# Locking and Scheduling

## Example (Locking and scheduling)

- Consider two transactions  $T_{1,2}$ :
- Two valid schedules (respecting lock and unlock calls) are:

$T_1$	$T_2$
lock A	lock A
write A	write A
lock B	lock B
unlock A	write B
write B	write A
unlock B	unlock A
	write B
	unlock B

Schedule $S_1$	
$T_1$	$T_2$
lock A	
write A	
lock B	
unlock A	
	lock A
	write A
write B	
unlock B	

Schedule $S_2$	
$T_1$	$T_2$
	lock A
	write A
	lock B
	write B
	write A
	unlock A
lock A	
write A	
	lock B
	write B
	write A
	unlock A
	write B
	unlock B
lock B	
write B	
unlock B	
	unlock A

- Note: Both schedules  $S_{1,2}$  are serializable. **Are we done yet?**

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

# Locking and Serializability

Example (Proper locking does not guarantee serializability yet)

Even if we adhere to a properly nested lock/unlock discipline, the scheduler might still yield **non-serializable schedules**:

Schedule  $S_1$

$T_1$	$T_2$
lock A	
lock C	
write A	
write C	
unlock A	
	lock A
	write A
	lock B
	unlock A
	write B
	unlock B
unlock C	
lock B	
write B	
unlock B	
	lock C
	write C
	unlock C

⌚ What is the conflict graph of this schedule?

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## ATM Transaction with Locking

### Example (Two concurrent ATM transactions with locking)

Transaction 1	Transaction 2	DB state
		1200
read ( <i>acct</i> ) ;		
	read ( <i>acct</i> ) ;	
write ( <i>acct</i> ) ;		1100
	write ( <i>acct</i> ) ;	1000

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## ATM Transaction with Locking

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

### Example (Two concurrent ATM transactions with locking)

Transaction 1	Transaction 2	DB state
lock (acct) ; read (acct) ; unlock (acct) ;		1200
	lock (acct) ; read (acct) ; unlock (acct) ;	
lock (acct) ; write (acct) ; unlock (acct) ;		1100
	lock (acct) ; write (acct) ; unlock (acct) ;	1000

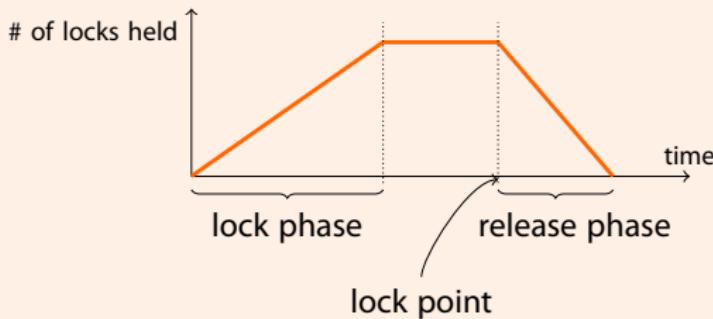
- ⇒ Again: on its own, proper locking does **not** guarantee serializability yet.

## Two-Phase Locking (2PL)

The **two-phase locking protocol** poses an additional restriction on how transactions have to be written:

### Definition (Two-Phase Locking)

- Once a transaction has **released** any lock (*i.e.*, performed the first unlock), it must **not** acquire any new lock:



- Two-phase locking is **the** concurrency control protocol used in database systems today.



## Again: ATM Transaction

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

Transaction 1	Transaction 2	DB state
lock (acct) ; read (acct) ; unlock (acct) ;		1200
	lock (acct) ; read (acct) ; unlock (acct) ;	
lock (acct) ; write (acct) ; unlock (acct) ;		1100
	lock (acct) ; write (acct) ; unlock (acct) ;	1000

## Again: ATM Transaction

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

Example (Two concurrent ATM transactions with locking,  $\neg$  2PL)

Transaction 1	Transaction 2	DB state
lock (acct) ; read (acct) ; unlock (acct) ;		1200
	lock (acct) ; read (acct) ; unlock (acct) ;	
lock (acct) ;  write (acct) ; unlock (acct) ;	lock (acct) ;  write (acct) ; unlock (acct) ;	1100 1000

 These locks violate the 2PL principle.

## A 2PL-Compliant ATM Transaction

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

- To comply with the two-phase locking protocol, the ATM transaction must not acquire any new locks after a first lock has been released:

### A 2PL-compliant ATM withdrawal transaction

```
1 lock (acct) ;                                } lock phase
2 bal ← read_bal (acct) ;
3 bal ← bal - 100 € ;
4 write_bal (acct, bal) ;
5 unlock (acct) ;                                } unlock phase
```

## Resulting Schedule

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

### Example

#### Transaction 1

```
lock (acct) ;  
read (acct) ;  
  
write (acct) ;  
unlock (acct) ;
```

#### Transaction 2

```
lock (acct) ;
```

Transaction  
blocked

```
lock (acct) ;
```

```
read (acct) ;
```

```
write (acct) ;
```

```
unlock (acct) ;
```

#### DB state

1200

1100

900

## Resulting Schedule

Transaction Management  
Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

### Example

Transaction 1	Transaction 2	DB state
lock (acct) ; read (acct) ;		1200
write (acct) ; unlock (acct) ;	lock (acct) ; <b>Transaction blocked</b>	1100
	lock (acct) ; read (acct) ; write (acct) ; unlock (acct) ;	900

- **Theorem:** The use of 2PL-compliant locking **always** leads to a correct and **serializable** schedule or to a **deadlock**.

## Lock Modes

- We saw earlier that two **read** operations do not conflict with each other.
- Systems typically use different types of locks (**lock modes**) to allow read operations to run concurrently.
  - **read locks** or **shared locks**: mode S
  - **write locks** or **exclusive locks**: mode X
- Locks are only in conflict if at least one of them is an X lock:

### Shared vs. exclusive lock compatibility

	shared (S)	exclusive (X)
shared (S)		×
exclusive (X)	×	×

- It is a safe operation in two-phase locking to (try to) **convert a shared lock into an exclusive lock during the lock phase** (lock upgrade)  $\Rightarrow$  improved concurrency.

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## Deadlocks

- Like many lock-based protocols, two-phase locking has the risk of **deadlock** situations:

### Example (Proper schedule with locking)

#### Transaction 1

```
lock (A) ;
```

```
:
```

```
do something
```

```
:
```

```
lock (B) ;
```

```
[wait for  $T_2$  to release lock]
```

#### Transaction 2

```
lock (B) ;
```

```
:
```

```
do something
```

```
:
```

```
lock (A) ;
```

```
[wait for  $T_1$  to release lock]
```

Transaction  
Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

- Both transactions would wait for each other **indefinitely**.

# Deadlock Handling

- **Deadlock detection:**

- ➊ The system maintains a **waits-for graph**, where an edge  $T_1 \rightarrow T_2$  indicates that  $T_1$  is blocked by a lock held by  $T_2$ .
- ➋ Periodically, the system tests for **cycles** in the graph.
- ➌ If a cycle is detected, the deadlock is **resolved** by **aborting** one or more transactions.
- ➍ Selecting the **victim** is a challenge:
  - Aborting a **young** transaction may lead to **starvation**: the same transaction may be cancelled again and again.
  - Aborting an **old** transaction may cause a lot of computational investment to be thrown away (but the **undo** costs may be high).

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

# Deadlock Handling

- **Deadlock detection:**

- ① The system maintains a **waits-for graph**, where an edge  $T_1 \rightarrow T_2$  indicates that  $T_1$  is blocked by a lock held by  $T_2$ .
- ② Periodically, the system tests for **cycles** in the graph.
- ③ If a cycle is detected, the deadlock is **resolved** by **aborting** one or more transactions.
- ④ Selecting the **victim** is a challenge:
  - Aborting a **young** transaction may lead to **starvation**: the same transaction may be cancelled again and again.
  - Aborting an **old** transaction may cause a lot of computational investment to be thrown away (but the **undo** costs may be high).

- **Deadlock prevention:**

Define an **ordering**  $\ll$  **on all database objects**. If  $A \ll B$ , then order the lock operations in all transactions in the same way ( $\text{lock}(A)$  before  $\text{lock}(B)$ ).



## Deadlock Handling

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

Other common technique:

- **Deadlock detection via timeout:**

Let a transaction  $T$  block on a lock request only until a **timeout** occurs (counter expires). On expiration, *assume* that a deadlock has occurred and **abort**  $T$ .

### DB2. Timeout-based deadlock detection

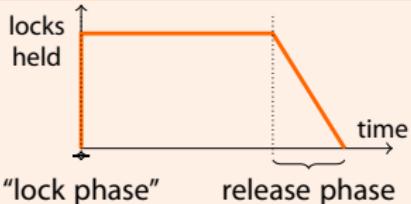
```
db2 => GET DATABASE CONFIGURATION;  
:  
Interval for checking deadlock (ms)      (DLCHKTIME) = 10000  
Lock timeout (sec)                      (LOCKTIMEOUT) = 30  
:  
:
```

- Also: lock-less **optimistic concurrency control** (↗ slide 42).

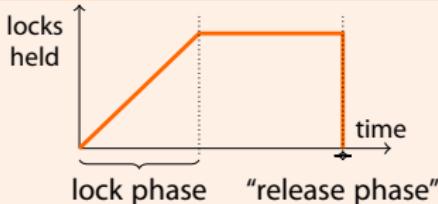
## Variants of Two-Phase Locking

- The two-phase locking discipline does not prescribe exactly when locks have to be acquired and released.
- Two possible variants:

### Preclaiming and strict 2PL



Preclaiming 2PL



Strict 2PL

### What could motivate either variant?

- 1 Preclaiming 2PL:
- 2 Strict 2PL:

Transaction Management  
Torsten Grust



ACID Properties  
Anomalies  
The Scheduler  
Serializability  
Query Scheduling  
Locking

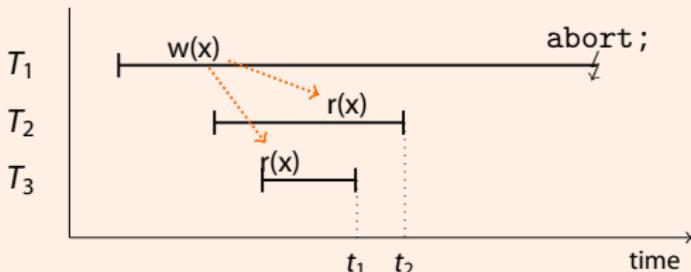
### Two-Phase Locking

Optimistic Concurrency Protocol  
Multi-Version Concurrency Control

## Cascading Rollbacks

Consider three transactions:

Transactions  $T_{1,2,3}$ ,  $T_1$  fails later on

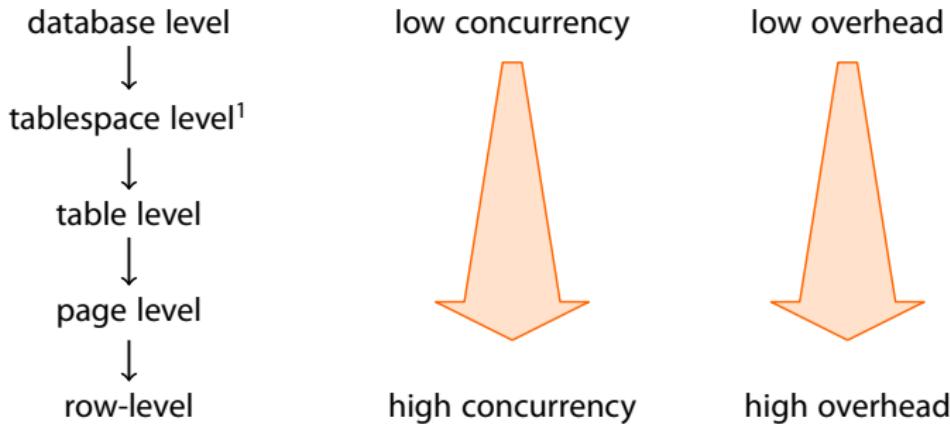


- When transaction  $T_1$  aborts, transactions  $T_2$  and  $T_3$  have already read data written by  $T_1$  ( $\nearrow$  dirty read, slide 9)
  - $T_2$  and  $T_3$  need to be **rolled back**, too (cascading roll back).
  - $T_2$  and  $T_3$  **cannot** commit until the fate of  $T_1$  is known.
- ⇒ Strict 2PL can avoid cascading roll backs altogether. (How?)



## Granularity of Locking

The **granularity** of locking is a trade-off:



⇒ Idea: **multi-granularity** locking.

<sup>1</sup>An DB2 tablespace represents a collection of tables that share a physical storage location.



## Multi-Granularity Locking

- Decide the granularity of locks held **for each transaction** (depending on the characteristics of the transaction):
  - For example, acquire a **row lock** for

### Row-selecting query (C\_CUSTKEY is key)

```
1   SELECT *
2   FROM   CUSTOMERS
3   WHERE  C_CUSTKEY = 42
```

$Q_1$

and a **table lock** for

### Table scan query

```
1   SELECT *
2   FROM   CUSTOMERS
```

$Q_2$

- How do such transactions know about each others' locks?
  - Note that locking is **performance-critical**.  $Q_2$  does not want to do an extensive search for row-level conflicts.



## Intention Locks

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

Databases use an additional type of locks: **intention locks**.

- Lock mode **intention share**: IS
- Lock mode **intention exclusive**: IX

### Extended conflict matrix

	S	X	IS	IX
S		×		×
X	×	×	×	×
IS		×		
IX	×	×		

- A lock  $I\square$  on a coarser level of granularity means that there is some  $\square$  lock on a lower level.

## Intention Locks

### Multi-granularity locking protocol

- ① Before a granule  $g$  can be locked in  $\square \in \{S, X\}$  mode, the transaction has to obtain an  $I\square$  lock on **all** coarser granularities that contain  $g$ .
- ② If all intention locks could be granted, the transaction can lock granule  $g$  in the announced  $\square$  mode.

### Example (Multi-granularity locking)

Query  $Q_1$  (↗ slide 38) would, e.g.,

- obtain an **IS** lock on **table CUSTOMERS** (also on the containing tablespace and database) and
- obtain an **S** lock on the **row(s)** with  $C\_CUSTKEY = 42$ .

Query  $Q_2$  would place an

- **S** lock on **table CUSTOMERS** (and an **IS** lock on tablespace and database).

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## Detecting Conflicts

- Now suppose an updating query comes in:

### Update request

```
1 UPDATE CUSTOMERS          Q3
2 SET NAME = 'SevenTeen'
3 WHERE C_CUSTKEY = 17
```

- $Q_3$  will want to place
  - an IX lock on **table** CUSTOMER (and all coarser granules) and
  - an X lock on the **row** holding customer 17.

As such it is

- compatible** with  $Q_1$   
(there is no conflict between IX and IS on the table level),
- but **incompatible** with  $Q_2$   
(the table-level S lock held by  $Q_2$  is in **conflict** with  $Q_3$ 's IX lock request).

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## Optimistic Concurrency Control

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

- Up to here, the approach to concurrency control has been **pessimistic**:
  - Assume that transactions **will conflict** and thus protect database objects by locks and lock protocols.
- The converse is a **optimistic concurrency control** approach:
  - Hope for the best and let transactions freely proceed with their read/write operations.
  - Only just before updates are to be committed to the database, perform a check to see whether conflicts indeed did not happen.
- Rationale: Non-serializable conflicts are not that frequent. **Save the locking overhead** in the majority of cases and only invest effort if really required.

# Optimistic Concurrency Control

Under **optimistic concurrency control**, transactions proceed in **three phases**:

## Optimistic concurrency control

- ① **Read Phase.** Execute transaction, but do **not** write data back to disk immediately. Instead, collect updates in the transaction's **private workspace**.
- ② **Validation Phase.** When the transaction wants to **commit**, test whether its execution was correct (only acceptable conflicts happened). If it is not, **abort** the transaction.
- ③ **Write Phase.** Transfer data from private workspace into database.

- Note: Phases ② and ③ need to be performed in a non-interruptible *critical section* (thus also called the **val-write phase**).

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## Validating Transactions

Validation is typically implemented by looking at transaction  $T_j$ 's

- **Read Set**  $RS(T_j)$  (attributes read by  $T_j$ ) and
- **Write Set**  $WS(T_j)$  (attributes written by  $T_j$ ).

### Backward-oriented optimistic concurrency control (BOCC)

Compare  $T_j$  against all **committed** transactions  $T_i$ .

Check **succeeds** if

$T_i$  committed before  $T_j$  started    or     $RS(T_j) \cap WS(T_i) = \emptyset$  .

### Forward-oriented optimistic concurrency control (FOCC)

Compare  $T_j$  against all **running** transactions  $T_i$ .

Check **succeeds** if

$WS(T_j) \cap RS(T_i) = \emptyset$  .

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## Optimistic Concurrency Control in IBM DB2

- DB2 V9.5 provides SQL-level constructs that enable database applications to implement optimistic concurrency control:
  - RID( $r$ ): return row identifier for row  $r$ ,
  - ROW CHANGE TOKEN FOR  $r$ : unique number reflecting the time row  $r$  has last been updated.

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

### DB2. Optimistic concurrency control

```
1 db2 => SELECT * FROM EMPLOYEES
2
3 ID      NAME DEPT SALARY
4 -----
5      1 Alex  DE      300
6      2 Bert  DE      100
7      3 Cora  DE      200
8      4 Drew   US     200
9      5 Erik   US     400
10
11 db2 => SELECT E.NAME, E.SALARY,
12           RID(E) AS RID, ROW CHANGE TOKEN FOR E AS TOKEN
13           FROM EMPLOYEES E
14           WHERE E.NAME = 'Erik'
15
16 NAME SALARY      RID          TOKEN
17 -----
18 Erik    400        16777224    74904229642240
```

# Optimistic Concurrency Control in IBM DB2

## DB2. Optimistic concurrency control

- The 'Erik' row belongs to our read set. Save its RID and TOKEN values to perform validation later.
- (...Time passes ...) Now try to save our changes to the row. Perform BOCC-style validation:

```
1 db2 => UPDATE EMPLOYEES E
2      SET E.SALARY = 450
3      WHERE RID(E) = 16777224          -- identify row
4      AND ROW CHANGE TOKEN FOR E = 74904229642240 -- row changed?
5
6 SQL0100W No row was found for FETCH, UPDATE or DELETE; or the
7 result of a query is an empty table. SQLSTATE=02000
8
9 db2 => SELECT E.ID, E.NAME
10        RID(E) AS RID, ROW CHANGE TOKEN FOR E AS TOKEN
11        FROM EMPLOYEES E
12
13    ID      NAME    RID              TOKEN
14    -----  -----
15      1 Alex    16777220    74904229642240
16      2 Bert    16777221    74904229642240
17      3 Cora    16777222    74904229642240
18      4 Drew    16777223    74904229642240
19      5 Erik    16777224    141378732653941710
```

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## Multi-Version Concurrency Control

Looking back at the concurrency control strategies discussed up to this point, we have seen

- ① **Wait Mechanisms**, i.e., **locks** and the associated two-phase locking protocol,
- ② **Rollback Mechanisms**, i.e., a conditional write phase that makes it trivial to **take back any changes** made by a transaction.

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## Multi-Version Concurrency Control

Looking back at the concurrency control strategies discussed up to this point, we have seen

- ① **Wait Mechanisms**, i.e., **locks** and the associated two-phase locking protocol,
- ② **Rollback Mechanisms**, i.e., a conditional write phase that makes it trivial to **take back any changes** made by a transaction.

We now add

- ③ **Timestamp Mechanisms** that use a **DBMS-wide clock** to order transactions and decide visibility of rows.

The resulting **Multi-Version Concurrency Control (MVCC)** protocol is lock-less but comes with a space overhead (that requires **garbage collection** of rows).

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## MVCC: Timestamps

- In MVCC, each transaction  $T_i$  is assigned a **timestamp**  $t_i$  that represents the point in time when  $T_i$  started.
- Can implement timestamps based on
  - **actual system clock** (resolution, uniqueness, portability across OSs?),  
or
  - **a DBMS-internal counter** used to assign transaction IDs (xid).
- Timestamp requirements:
  - ① unique:  $t_i \neq t_j$  if  $i \neq j$ ,
  - ② ordered:  $t_i < t_j$  if  $T_i$  has started before  $T_j$ .

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## MVCC: Timestamps

- In MVCC, each transaction  $T_i$  is assigned a **timestamp**  $t_i$  that represents the point in time when  $T_i$  started.
- Can implement timestamps based on
  - **actual system clock** (resolution, uniqueness, portability across OSs?),  
or
  - **a DBMS-internal counter** used to assign transaction IDs (xid).
- Timestamp requirements:
  - ① unique:  $t_i \neq t_j$  if  $i \neq j$ ,
  - ② ordered:  $t_i < t_j$  if  $T_i$  has started before  $T_j$ .

### MVCC: Semantics

Under MVCC, a transaction  $T_i$  operates on **the consistent state of the database that was current at time  $t_i$** .

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## MVCC: Versions and Snapshots

In a concurrent DBMS, operations can **conflict** if they write the **same database object** (recall relation  $\leftrightarrow$ ). Thus:

### MVCC: Versions

Under MVCC, **multiple versions of the same database object** may exist at one time. Different transactions may read/write different (not necessarily the most recent) object versions.

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## MVCC: Versions and Snapshots

In a concurrent DBMS, operations can **conflict** if they write the **same database object** (recall relation  $\leftrightarrow$ ). Thus:

### MVCC: Versions

Under MVCC, **multiple versions of the same database object** may exist at one time. Different transactions may read/write different (not necessarily the most recent) object versions.

MVCC uses **snapshots** to identify exactly which version of each database object are visible to a transaction:

### MVCC: Snapshot

To take a **snapshot**, gather the following information:

- the highest `xid` of all committed transactions,
- a list of `xids` of all transactions currently executing.

Typically, a snapshot is taken at the time the transaction starts.

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## MVCC: Row Timestamps



### Database Object ≡ Row

PostgreSQL implements MVCC at a granularity of rows: **multiple versions of the same row** may exist. Adopt this model in what follows.

- To help decide whether a particular row version is included in (or excluded from) a snapshot, attach to each row version two virtual/hidden attributes:
  - ① xmin: the xid of the transaction that **created** this row,
  - ② xmax: the xid of the transaction that **deleted** this row.
- Row **updates** are modelled as the two-step operation row deletion, then creation.

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## MVCC: Row Timestamps



Database Object ≡ Row

PostgreSQL implements MVCC at a granularity of rows: **multiple versions of the same row** may exist. Adopt this model in what follows.

- To help decide whether a particular row version is included in (or excluded from) a snapshot, attach to each row version two virtual/hidden attributes:
  - ① xmin: the xid of the transaction that **created** this row,
  - ② xmax: the xid of the transaction that **deleted** this row.
- Row **updates** are modelled as the two-step operation row deletion, then creation.

### MVCC: No Physical Deletion!

Note: ② implies that rows are **not actually physically deleted**. Instead, their xmax attribute is modified to record the deleting/updating transaction (⇒ eventual row garbage).



Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## MVCC: Row Visibility (1)

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

- Current snapshot:<sup>2</sup>  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$

---

<sup>2</sup>For simplicity: assume that all other xids have committed (and not rolled back their work).

## MVCC: Row Visibility (1)

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

- Current snapshot:<sup>2</sup>  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

1	xmin	xmax	... data ...
	30	—	...

<sup>2</sup>For simplicity: assume that all other xids have committed (and not rolled back their work).

## MVCC: Row Visibility (1)

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

### Example (Decide Row Visibility)

- Current snapshot:<sup>2</sup>  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

1	xmin	xmax	... data ...	✓
	30	—	...	

<sup>2</sup>For simplicity: assume that all other xids have committed (and not rolled back their work).

## MVCC: Row Visibility (1)

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

### Example (Decide Row Visibility)

- Current snapshot:<sup>2</sup>  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

	xmin	xmax	... data ...	
1	30	—	...	✓
2	50	—	...	

<sup>2</sup>For simplicity: assume that all other xids have committed (and not rolled back their work).

## MVCC: Row Visibility (1)

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

### Example (Decide Row Visibility)

- Current snapshot:<sup>2</sup>  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

	xmin	xmax	... data ...	
1	30	—	...	✓
2	50	—	...	✗

<sup>2</sup>For simplicity: assume that all other xids have committed (and not rolled back their work).

## MVCC: Row Visibility (1)

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

### Example (Decide Row Visibility)

- Current snapshot:<sup>2</sup>  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

	xmin	xmax	... data ...	
1	30	—	...	✓
2	50	—	...	✗
3	110	—	...	

<sup>2</sup>For simplicity: assume that all other xids have committed (and not rolled back their work).

## MVCC: Row Visibility (1)

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

### Example (Decide Row Visibility)

- Current snapshot:<sup>2</sup>  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

	xmin	xmax	... data ...	
1	30	—	...	✓
2	50	—	...	✗
3	110	—	...	✗

<sup>2</sup>For simplicity: assume that all other xids have committed (and not rolled back their work).

## MVCC: Row Visibility (2)

### Example (Decide Row Visibility)

- Current snapshot:  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

1	xmin	xmax	$\dots$ data $\dots$
	30	80	$\dots$

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## MVCC: Row Visibility (2)

### Example (Decide Row Visibility)

- Current snapshot:  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

1	xmin	xmax	$\dots$ data $\dots$	$\nexists$
	30	80	$\dots$	

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

## MVCC: Row Visibility (2)

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

- Current snapshot:  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

1	xmin	xmax	$\dots$ data $\dots$	
	30	80	$\dots$	↙

2	xmin	xmax	$\dots$ data $\dots$
	30	75	$\dots$

## MVCC: Row Visibility (2)

Transaction Management

Torsten Grust



### Example (Decide Row Visibility)

- Current snapshot:  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

1	xmin	xmax	$\dots$ data $\dots$	✗
	30	80	$\dots$	

2	xmin	xmax	$\dots$ data $\dots$	✓
	30	75	$\dots$	

ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## MVCC: Row Visibility (2)

Transaction Management

Torsten Grust



### Example (Decide Row Visibility)

- Current snapshot:  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

1	xmin	xmax	$\dots$ data $\dots$	✗
	30	80	$\dots$	

2	xmin	xmax	$\dots$ data $\dots$	✓
	30	75	$\dots$	

3	xmin	xmax	$\dots$ data $\dots$
	30	110	$\dots$

ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

## MVCC: Row Visibility (2)

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic  
Concurrency Protocol

Multi-Version  
Concurrency Control

- Current snapshot:  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

	xmin	xmax	... data ...	
1	30	80	...	✗
2	30	75	...	✓
3	30	110	...	✓

## MVCC: Row Visibility (2)

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control

- Current snapshot:  $\langle \underbrace{100}_{\text{highest committed xid}}, \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

1	xmin	xmax	... data ...	✗
	30	80	...	

2	xmin	xmax	... data ...	✓
	30	75	...	

3	xmin	xmax	... data ...	✓
	30	110	...	

- Given the current state of the system, may row 1 be considered garbage that can be collected?

## MVCC: Garbage Collection

- The creation of new row versions during UPDATE (rather than replacing the existing row) requires the **reclamation of storage space** used by old row versions.
- Delay such **row garbage collection** until the old versions are guaranteed to be invisible to all current and future transactions.

### Delayed Row Garbage Collection

Exactly when is it safe to declare a row as garbage and mark it for collection?



### Row Cleanup

- **On-demand cleanup of a single page** when page is accessed during SELECT, UPDATE, DELETE.
- **Bulk cleanup** by scheduled auto-vacuum process or via an explicit VACUUM command.

Transaction Management

Torsten Grust



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Optimistic Concurrency Protocol

Multi-Version Concurrency Control