Hasso Plattner Institut
Digital Engineering • Universität Potsdam

# Big Data Systems

Winter Semester 2019 / 2020

Database Systems Recap

Prof. Tilmann Rabl
Data Engineering Systems
Hasso-Plattner-Institut

# Announcements

- Tonight 5pm:

  Maximilian Jenders (GetYourGuide)

  Recommending Tourist Activities - Data Science Challenges And The Needs for Data Pipelines

- Access to Moodle:
  - Non-HPI students, who do not have an account yet: write me an email
  - Learn how to write professional emails: https://medium.com/@lportwoodstacer/how-to-email-your-professor-without-being-annoying-af-cf64ae0e4087

- Quiz will be online soon!

# Tentative Timeline

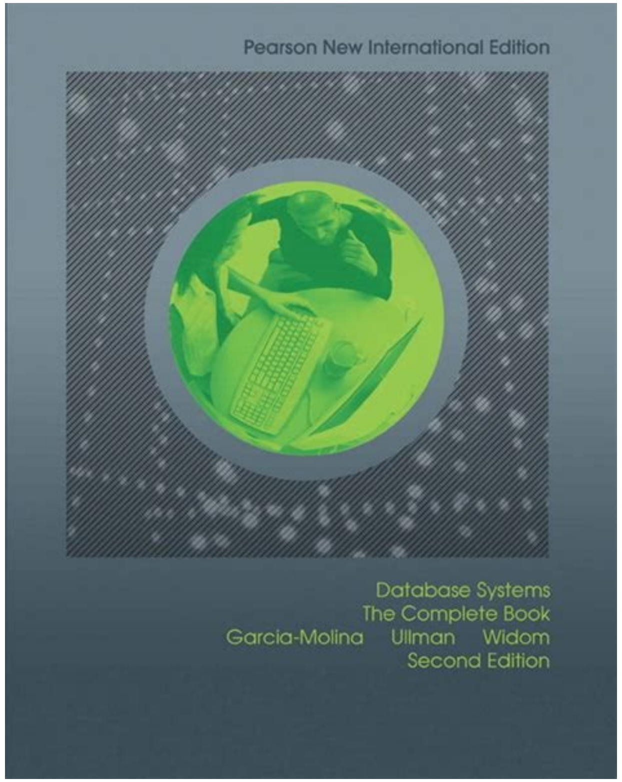
| Date | Tuesday | Thursday |
|---|---|---|
| 15./17.10. | Introduction | *No class* |
| 22./24.10. | DBS Recap | DBS Recap II |
| 29.10/31.10. | *20 Years HPI* | *Holiday* |
| 5./7.11. | Big Data Stack | *Solution Quiz I* |
| 12./14.11. | Benchmarking & Measurement | Cloud/Container |
| 19./21.11. | ***Facebook Chief Scientist*** | File Systems **(starts 20 min late)** |
| 26. /28.11. | Map/Reduce | *Solution Quiz II* |
| 3./5.12. | KV-Stores | Consistency |
| 10./12.12. | Stream Processing | Windows |
| 17./19.12. | Tables and State | *Solution Quiz III* |

# Tentative Timeline cont‘d

| Date | Tuesday | Thursday |
| --- | --- | --- |
| 7./9.1. | Stream Optimizations | *Solution Quiz IV* |
| 14./16.1. | ML Systems | ML Exec Strategies |
| 21./23.1. | ML Lifecycle | Graph Processing |
| 28./30.1. | Graph Processing II | *Solution Quiz V* |
| 4./6.2. | Q&A | Final Exam |

# This Lecture

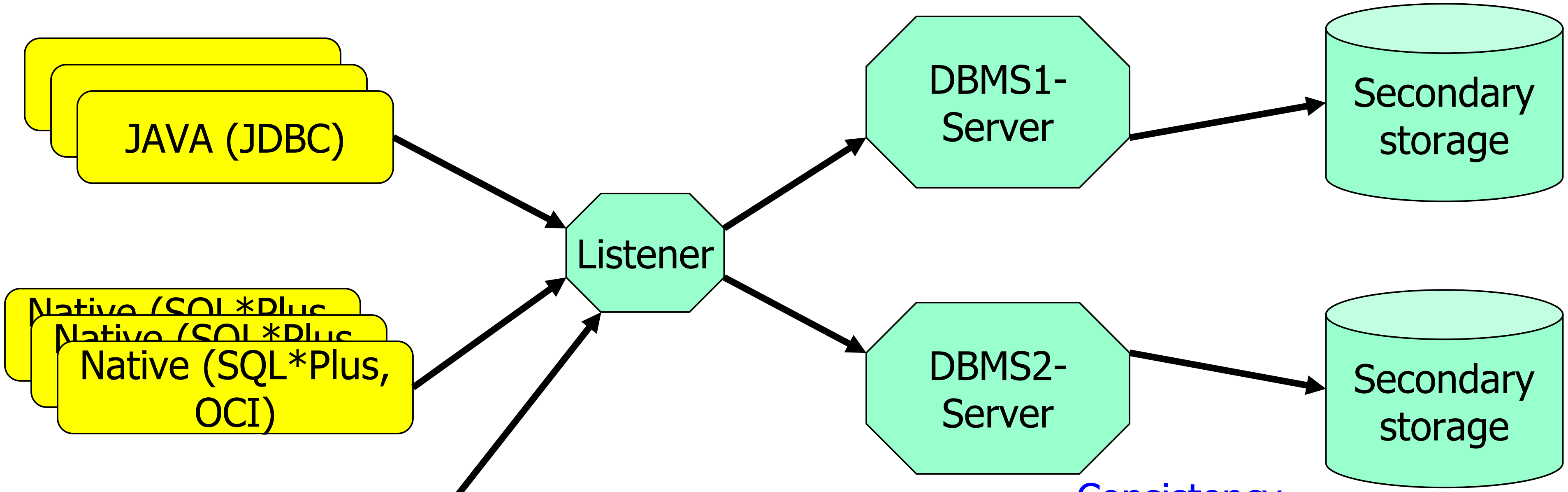
1. Review of Relational Database Management
   - Relational Model
   - Operators
   - Algebra
   - SQL
2. Review of Relational Database Systems
   - Storage and Data Representation
   - Hashing & B-Trees
   - Query Execution
   - Query Compilation / Optimization



# Relational Databases

- Relational database management system (RDBMS)
  - Server based software
  - One RDBMS – many relational databases (RDBs)
  - Responsibilities of these servers
    - Management of main storage and secondary storage
    - Transaction management
    - Query processing and optimization
    - Backup and recovery
    - Data consistency
    - User management
  - Systems
    - Oracle, DB2 (Informix), Sybase, NCR Teradata, SQL Server
    - PostgreSQL, InterBase, Berkeley DB, db4o, MySQL, Ingres, SAP DB, MonetDB, ...

# Client-Server




Consistency
Parallelization
Recovery
Load Balancing
Authentication
Authorization
....

# Three Levels of Data Representation

- Conceptual level
  - Relations, Tuple
  - Values of attributes
- Logical level
  - Files
  - Records
  - Fields
- Physical level
  - Drives
  - Blocks
  - Cylinders and Sectors

Relational Model (first part) — Conceptual level

RDBMS Internals (second part) — Logical level and Physical level

# Relational Data Model

- Representation of all data
  (e.g., Entity-Types and Relationship-Types of the ER-Model)
  through Relations
  - Relation Name
  - Attributes
  - (Data types)

Employee

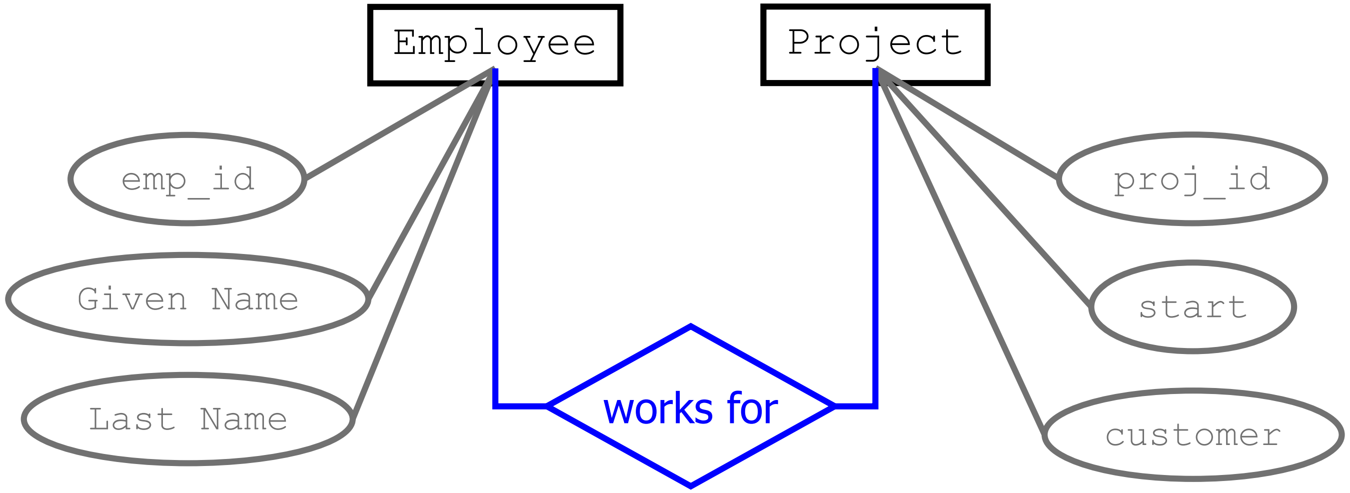
Columns/Attributes

Rows/
Tuples

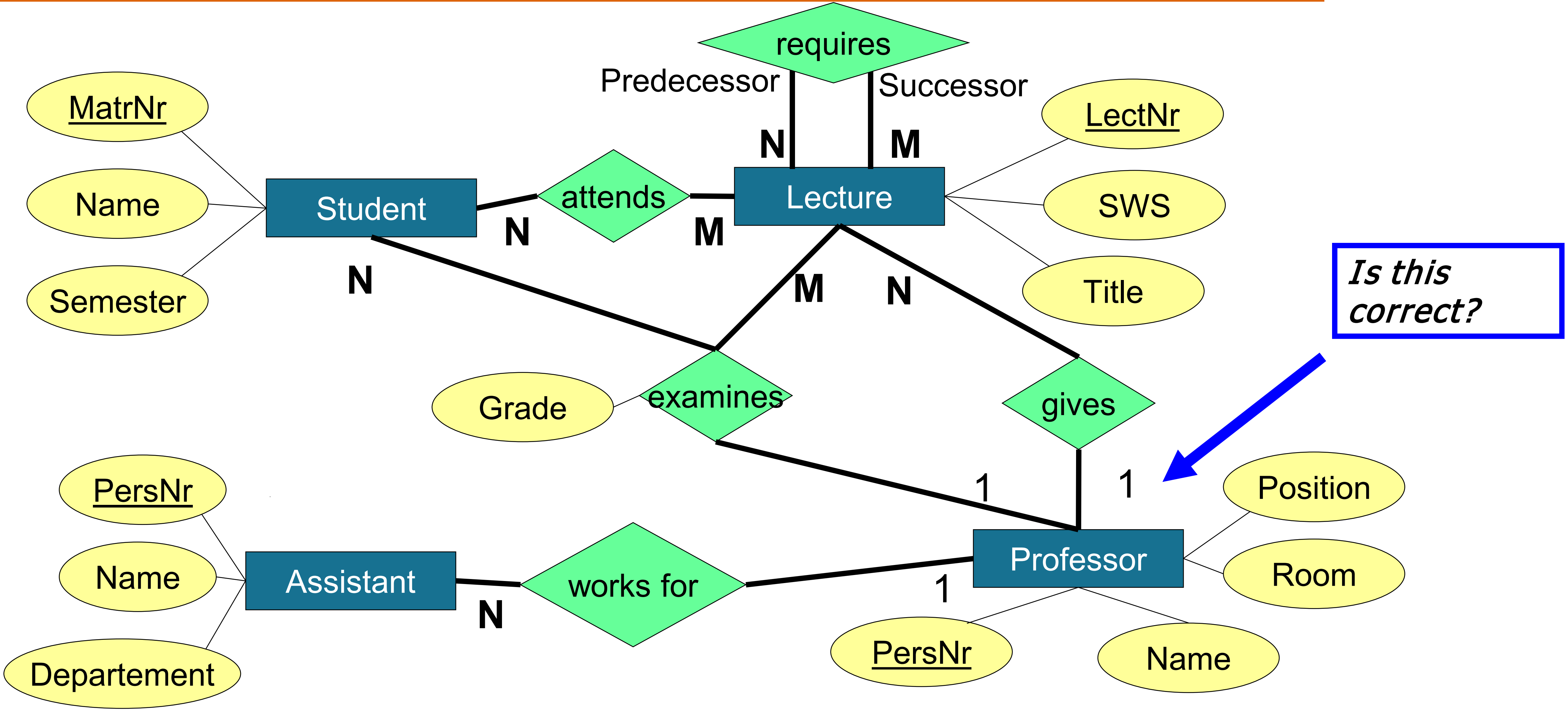
| P_ID | Given Name | Last Name | Age | Adress |
|---|---|---|---|---|
| 1 | Peter | Müller | 32 | 10101 Berlin |
| 2 | Stefanie | Meier | 34 | 11202 Berlin |
| 5 | Petra | Weger | 28 | 80223 München |
| 7 | Andreas | Zwickel | 44 | 80443 München |
| ... | ... | ... | ... | ... |

# ER-Modeling

- Relational data model has „limited semantics"
- Modeling with tables is not very expressive/intuitive
- Modeling languages: ER, EER, UML, ...
- Entity-Relationship Model



# University ER Schema



# Developing the Relational Schema

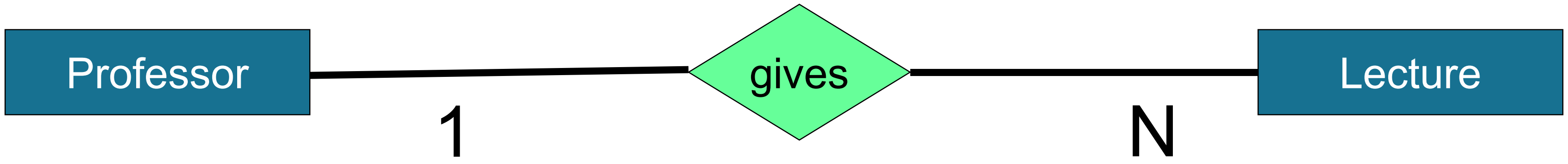
- 1:N Relationship Type

Initial Schema

Lecture : {LectNr, Title, SWS}

Professor : {PersNr, Name, Position, Room}

Gives: {LectNr, PersNr}



# Refinement of the Relational Schema

1:N-Relationship Type

- Initial schema

  Lecture : {LectNr, Title, SWS}

  Professor : {PersNr, Name, Position, Room}

  Gives: {LectNr, PersNr}

- Refinement through combination

  Lecture : {LectNr, Title, SWS, GivenBy}

  Professor : {PersNr, Name, Position, Room}

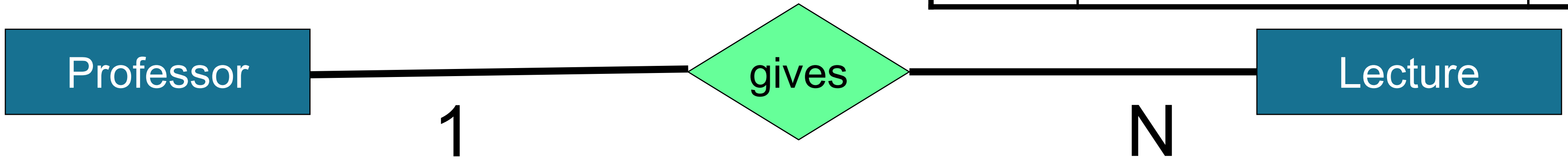
Rule

- Relations with the same key can be combined.
- But only these and no others!
- Beware of weak entity types and semantics!

# ER Model, Relational Schema, and Instance

| Professor | | | |
|---|---|---|---|
| PersNr | Name | Position | Room |
| 2125 | Sokrates | C4 | 226 |
| 2126 | Russel | C4 | 232 |
| 2127 | Kopernikus | C3 | 310 |
| 2133 | Popper | C3 | 52 |
| 2134 | Augustinus | C3 | 309 |
| 2136 | Curie | C4 | 36 |
| 2137 | Kant | C4 | 7 |

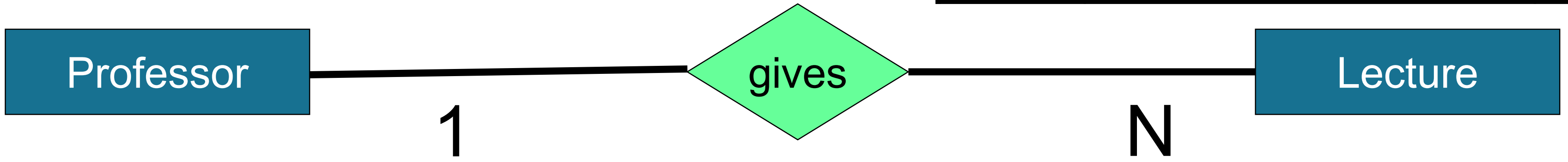
| Lecture | | | |
|---|---|---|---|
| LectNr | Title | SWS | GivenBy |
| 5001 | Grundzüge | 4 | 2137 |
| 5041 | Ethik | 4 | 2125 |
| 5043 | Erkenntnistheorie | 3 | 2126 |
| 5049 | Mäeutik | 2 | 2125 |
| 4052 | Logik | 4 | 2125 |
| 5052 | Wissenschaftstheorie | 3 | 2126 |
| 5216 | Bioethik | 2 | 2126 |
| 5259 | Der Wiener Kreis | 2 | 2133 |
| 5022 | Glaube und Wissen | 2 | 2134 |
| 4630 | Die 3 Kritiken | 4 | 2137 |



# This Does NOT Work

| Professor | | | | |
|---|---|---|---|---|
| PersNr | Name | Position | Room | Gives |
| 2125 | Sokrates | C4 | 226 | 5041 |
| 2125 | Sokrates | C4 | 226 | 5049 |
| 2125 | Sokrates | C4 | 226 | 4052 |
| ... | ... | ... | ... | ... |
| 2134 | Augustinus | C3 | 309 | 5022 |
| 2136 | Curie | C4 | 36 | ?? |
| | | | | |

| Lecture | | |
|---|---|---|
| LectNr | Title | SWS |
| 5001 | Grundzüge | 4 |
| 5041 | Ethik | 4 |
| 5043 | Erkenntnistheorie | 3 |
| 5049 | Mäeutik | 2 |
| 4052 | Logik | 4 |
| 5052 | Wissenschaftstheorie | 3 |
| 5216 | Bioethik | 2 |
| 5259 | Der Wiener Kreis | 2 |
| 5022 | Glaube und Wissen | 2 |
| 4630 | Die 3 Kritiken | 4 |



# Insert Anomaly

- What, if we want to insert Lawrence as a new Employee?

| EmpProj | | | | |
|---|---|---|---|---|
| EmpID | Name | Room | ProjID | ProjName |
| 1234 | Tilmann | 103 | 111 | BBDC |
| 4560 | Durgesh | 754 | 111 | BBDC |
| 3456 | Yue | 723 | 111 | BBDC |
| 5468 | Sebastian | 798 | 121 | BIFOLD |
| 8748 | Hendrik | 101 | 121 | BIFOLD |
| 8733 | Nina | 789 | 121 | BIFOLD |
| | | | | |

# Update Anomaly

- What, if we rename BBDC into BBDC2?

| EmpProj | | | | |
|---|---|---|---|---|
| EmpID | Name | Room | ProjID | ProjName |
| 1234 | Tilmann | 103 | 111 | BBDC |
| 4560 | Durgesh | 754 | 111 | BBDC |
| 3456 | Yue | 723 | 111 | BBDC |
| 5468 | Sebastian | 798 | 121 | BIFOLD |
| 8748 | Hendrik | 101 | 121 | BIFOLD |
| 8733 | Nina | 789 | 121 | BIFOLD |
| | | | | |

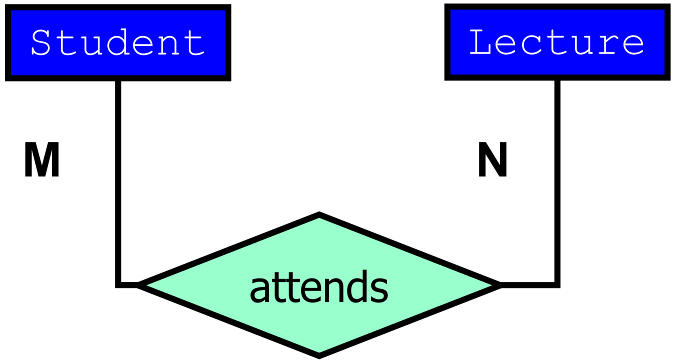
# Delete Anomaly

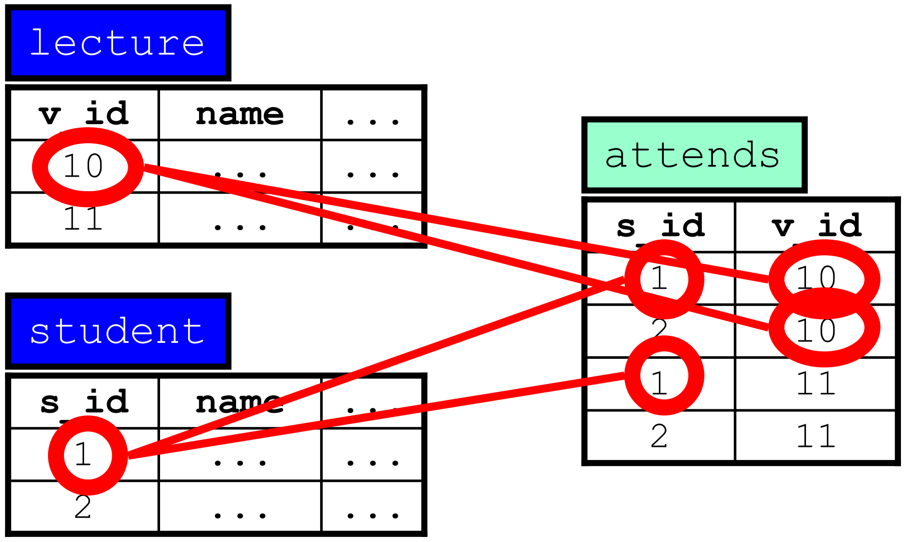
- What, if project BIFOLD is cancelled?

| EmpProj | | | | |
|---|---|---|---|---|
| EmpID | Name | Room | ProjID | ProjName |
| 1234 | Tilmann | 103 | 111 | BBDC |
| 4560 | Durgesh | 754 | 111 | BBDC |
| 3456 | Yue | 723 | 111 | BBDC |
| 5468 | Sebastian | 798 | 121 | BIFOLD |
| 8748 | Hendrik | 101 | 121 | BIFOLD |
| 8733 | Nina | 789 | 121 | BIFOLD |
| | | | | |

# Relationship-Type „attends"

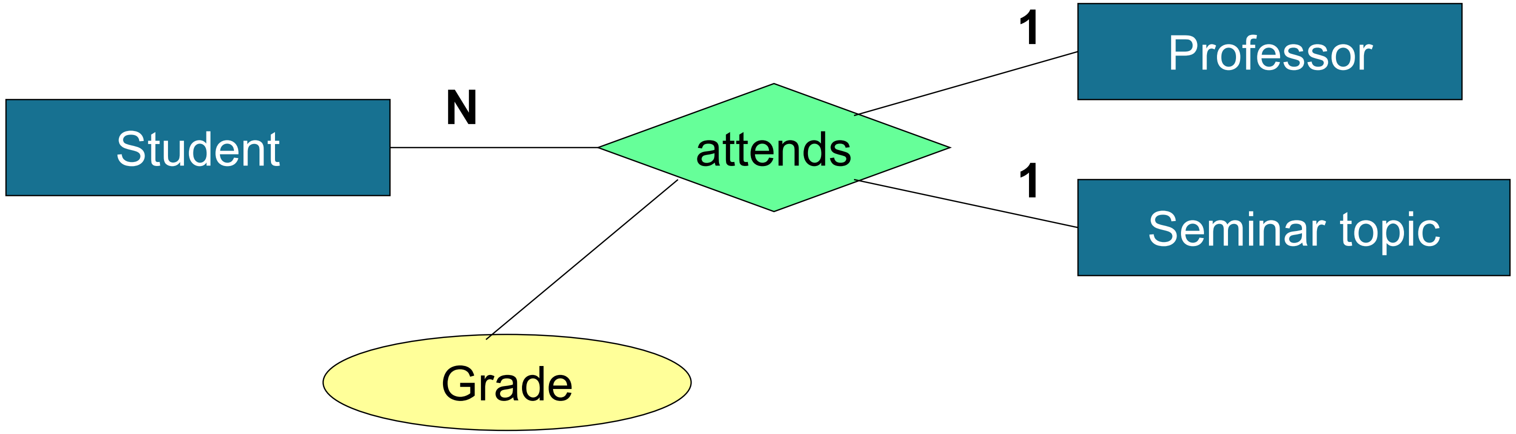
- M:N Relationship
- „Bridge table" with two foreign keys



# Relationship-Type „attends“

- attends : Professor x Student -> Seminar topic
- attends: Seminar topic x Student -> Professor



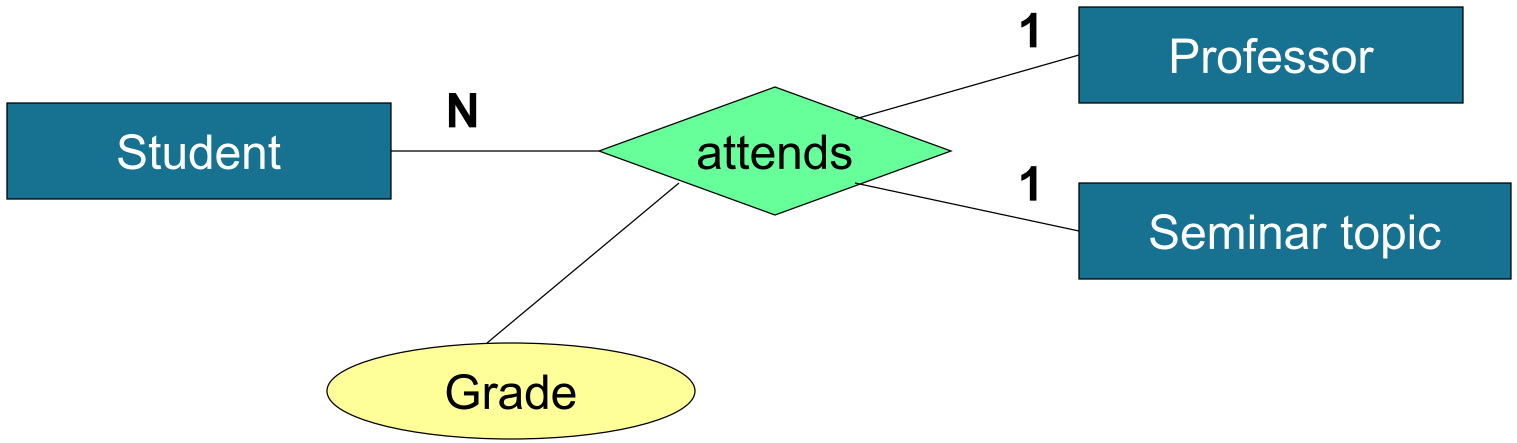
# Representing Integrity Constraints

- Students may only attend one seminar topic by one professor
- Students may work on the same topic only once – they may not have worked on the same topic with another professor

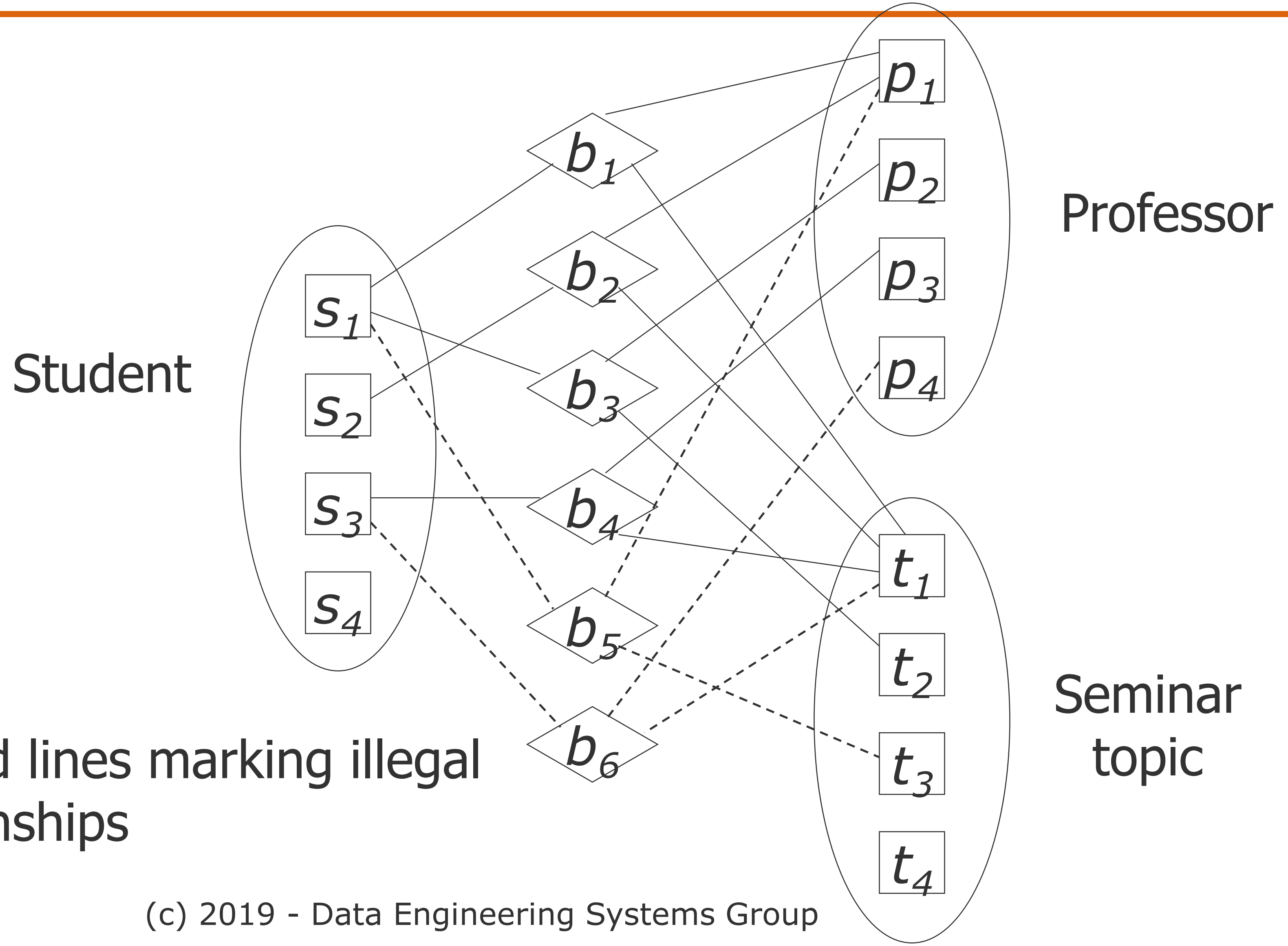
- However, the following should be possible:
  - Professors can re-use a seminar topic (i.e., can assign the same topic to multiple students)
  - The same topic may be mentored by multiple professors (for different students!)

# Relational Schema for the Constraints

- attends{ student, professor, topic, grade}
- unique(student, professor)
  - each student works only on one topic with a professor
- unique(student, topic)
  - each student works on a topic only once



# Instance of the Relation *attends*




Dashed lines marking illegal relationships

# Normal Forms

Attributes are functionally dependent

- Key → List of Values
- Examples:
  - pers_id → name, first_name, age …
  - proj_id → customer, status, …

- Candidate Key
  - Minimal set of attributes that functionally determines all the other attributes in a table

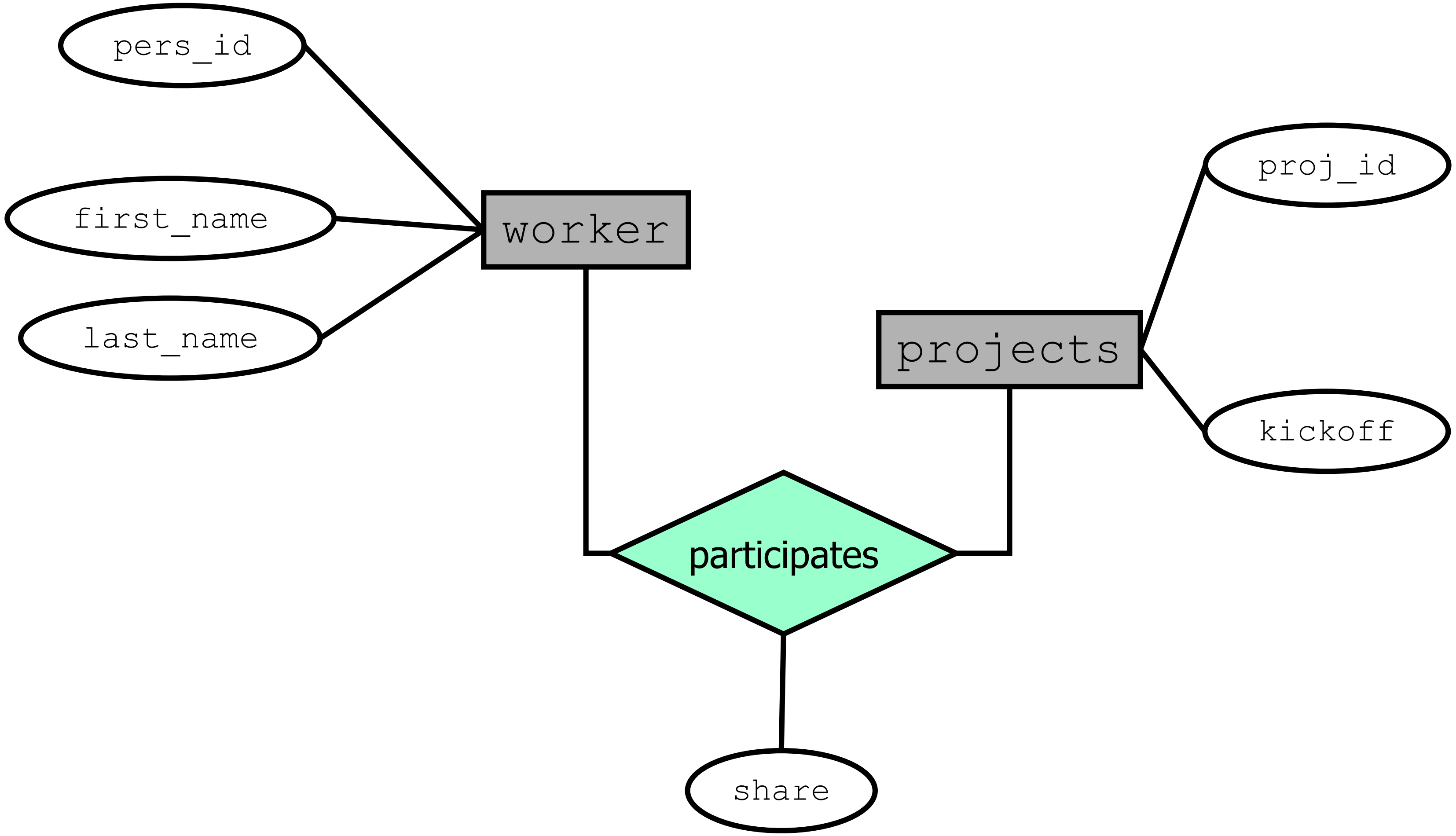
  (= minimal superkey)

- Decompose after functional dependencies
  - Done during schema design
  - Goal: No redundancy, no anomalies
  - Caveat: Normalization often hurts performance, tuning may involve de-normalization

# Normal Forms

- Relational Schema R, Candidate Keys P
- First Normal Form (1NF)
  - All attributes in R are atomic (example: address)
  - No automatic verification, depends usually on the format in which the application expects the data types
- Second Normal Form (2NF)
  - R is in 1NF
  - No attribute A, which is not part of a key, depends on a subkey
  - Violating example: teaches (prof_id, student_id, date, stud_name)
- Third Normal Form (3NF)
  - R in 2NF
  - No attribute A depends on a non-key attribute A'
  - Violating Example: residence(pers_id, zip, city)

# Relational Operations



# Basic Operations on Tables

- Selection
  - Get tuples of worker, satisfying age>40 and last_name="Anderson"

- Projection
  - Get only the worker -columns first_name,  last_name

- Cartesian Product
  - Combine all tuples from the table worker with all tuples from the table participates

- Composition/Nesting of Operators
  - Project columns last_name and proj_id in all tuples of the  Cartesian Product of worker and participates, having worker.pers_id=participates.pers_id and share > 10%

# Relational Algebra

- σ Selection
- π Projection
- x Cartesian Product
- ⋈ Join
- ρ Renaming
- – Set Difference
- ÷ Division
- ∪ Union
- ∩ Intersection
- ⋉ Left Semijoin
- ⋊ Right Semijoin
- ⟕ Left Outer Join
- ⟖ Right Outer Join
- ⟗ Full Outer Join

# Set Operations

Given two sets:

- $R = \{t_1, \ldots, t_m, t_{m+1}, \ldots, t_n\}$
- $S = \{t_{m+1}, \ldots, t_n, t_{n+1}, \ldots, t_k\}$

| $R \cup S$ | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $R - S$ | | | | $R \cap S$ | | | | $S - R$ | | | |
| $t_1$ | $t_2$ | … | $t_m$ | $t_{m+1}$ | $t_{m+2}$ | … | $t_n$ | $t_{n+1}$ | $t_{n+2}$ | … | $t_k$ |

# Natural Join

- Natural-Join / Equi-Join

**L**

| A | B | C |
|---|---|---|
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | $c_2$ |
| $a_3$ | $b_3$ | $c_3$ |
| $a_4$ | $b_4$ | $c_4$ |
| $a_5$ | $b_5$ | $c_5$ |

$\bowtie$

**R**

| C | D | E |
|---|---|---|
| $c_1$ | $d_A$ | $e_A$ |
| $c_1$ | $d_B$ | $e_B$ |
| $c_4$ | $d_C$ | $e_C$ |
| $c_6$ | $d_D$ | $e_D$ |

=

**Result**

| A | B | C | D | E |
|---|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_A$ | $e_A$ |
| $a_1$ | $b_1$ | $c_1$ | $d_B$ | $e_B$ |
| $a_4$ | $b_4$ | $c_4$ | $d_C$ | $e_C$ |

- $L \bowtie R = \Pi_{L.A,\ L.B,\ L.C,\ R.D,\ R.E}\,(\sigma_{L.C=R.C}\,(L \times R))$

# Other Join Types

- Left Outer Join

| L | | |
|---|---|---|
| A | B | C |
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | $c_2$ |

$⟕$

| R | | |
|---|---|---|
| C | D | E |
| $c_1$ | $d_1$ | $e_1$ |
| $c_3$ | $d_2$ | $e_2$ |

=

| Result | | | | |
|---|---|---|---|---|
| A | B | C | D | E |
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ |
| $a_2$ | $b_2$ | $c_2$ | - | - |

- Left Semi Join

| L | | |
|---|---|---|
| A | B | C |
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | $c_2$ |

$\ltimes$

| R | | |
|---|---|---|
| C | D | E |
| $c_1$ | $d_1$ | $e_1$ |
| $c_3$ | $d_2$ | $e_2$ |

=

| Result | | |
|---|---|---|
| A | B | C |
| $a_1$ | $b_1$ | $c_1$ |

# Structured Query Language (SQL)

- ANSI-SQL, SQL-92, SQL-99, SQL-3
- Declarative: What to execute, not how to execute!
- Four basic commands (CRUD): Insert, Update, Delete, Select
- DDL defines schema, DML works on the data
- Other languages:
  - Tuple-Calculus, Relational Algebra, Query By Example
- Most common: Select Query:

```
SELECT      <columns, renaming, (aggr.) functions)
FROM        <tables, table expressions, joins>
WHERE       <predicates>
GROUP BY    <grouping columns>
HAVING      <predicates on groups/aggregates>
ORDER BY    <order columns, ascending/descending)
```

# INSERT

- Inserting of values into a table

```
INSERT INTO worker VALUES
       (1, "John", "Smith", 38, "95112 San José");

INSERT INTO projects (proj_id, name, customer) VALUES
       (1, "BMW World", "BMW");


INSERT INTO worker SELECT * FROM worker_backup;


INSERT INTO … WHEN … INTO … WHEN …;
```

# UPDATE

- Edit values in a table
- Set semantics: Edit multiple values

```
UPDATE projects
SET    status = “cancelled“
WHERE customer=“Lehman Brothers“
```

- Typical pattern

```
UPDATE table
SET ... = (SELECT ... FROM ... WHERE)
WHERE id in (SELECT ... FROM ... WHERE)
```

- Extensions

```
UPSERT, MERGE
```

# DELETE

- Remove tuples from a table

```
DELETE FROM projects
WHERE status = "finished"
```

- Typical use case

```
DELETE FROM projects
WHERE proj_id in (SELECT ... FROM ... WHERE)
```

- Deletion alternatives for performance reasons

```
DELETE, DROP TABLE, TRUNCATE
```

# SELECT

- Query values across tables

```
SELECT      w.last_name, pt.share
FROM        worker w, participates pt
WHERE       w.pers_id = pt.pers_id AND
            pt.share > 0.1

SELECT      w.last_name, p.name, pt.share
FROM        worker w, projects p, participates pt
WHERE       w.pers_id = pt.pers_id AND
            pt.proj_id = p.proj_id

SELECT      w.last_name
FROM        worker w, participates pt
WHERE       w.pers_id = pt.pers_id
```

- Result again a table
- Physical execution up to RDBMS

# Additional Concepts

- Subqueries
  - Correlated / Uncorrelated
  - Does Uncorrelated form exist?

```
SELECT  first_name, last_name
FROM    worker w
WHERE   EXISTS (
        SELECT pt.pers_id
        FROM   participates pt
     WHERE   pt.pers_id = w.pers_id )
```

- Self-Join

```
SELECT p1.name, p2.name
FROM   projects p1, projects p2
WHERE  p1.predecessor=p2.proj_id AND
         p2.status="closed"
```

- Nested Table Expressions
  - SQL in FROM clause

```
SELECT X.last_name, X.status
FROM   (
  SELECT w.last_name, p.status
  FROM worker w, projects p, participates pt
  WHERE w.pers_id = pt.pers_id AND
          pt.proj_id = p.proj_id
  ) X
 WHERE  X.status="acquisition"
```

# Correlation in Subqueries

- Often possible to transform correlated subqueries to uncorrelated ones

```
SELECT s.*
FROM students s
WHERE EXISTS
        (SELECT p.*
         FROM professors p
         WHERE p.dob > s.dob);
```

```
SELECT s.*
FROM students s
WHERE s.dob <
        (SELECT max(p.dob)
         FROM professors p);
```

# Aggregating and Sorting

- Aggregation und GROUP BY

```
SELECT proj_id, COUNT(*), SUM(age)/COUNT(*)
FROM   worker w, participates pt, projects p
WHERE  w.pers_id = pt.pers_id AND
       pt.proj_id = p.proj_id
GROUP BY p.proj_id
```

- ORDER BY

```
SELECT p.name, w.last_name
FROM   worker w, participates pt, projects p
WHERE  w.pers_id = pt.pers_id AND
       pt.proj_id = p.proj_id
ORDER BY p.name, w.last_name
```

# Views

- Abstraction layer through “named queries”

```
CREATE VIEW proj_pers AS
   SELECT p.proj_id, p.name,
          w.pers_id, w.last_name, w.age,
   FROM   worker w, participates pt, projects p
   WHERE  w.pers_id = pt.pers_id AND
          pt.proj_id = p.proj_id;
```

- Save common parts in queries

```
SELECT proj_id, COUNT(*), SUM(age)/COUNT(*)
FROM   proj_pers
GROUP BY proj_id;
```

- Can be used for tuple-wise access control
- During query execution, views are syntactically expanded
- Additional concepts
  - Materialized Views (MQTs), Indexes on Views
  - Statistical Views

# DDL vs. DML

- DML: Data Manipulation Language
- DDL: Data Definition Language
- Definition of
  - Tables, Indexes, Views, ...
  - Administration: Tablespaces, Segments, Roles
  - Access Control: User, Groups, Privileges, ...

```
CREATE TABLE worker (
  pers_id        NUMBER,
  first_name     VARCHAR2(100),
  last_name      VARCHAR2(100),
  age            NUMBER(2) CHECK (age > 0 AND age < 150),
  address        VARCHAR2(1000)
);
```

# Data Integrity

- Semantically consistent state of the data
  - Constraints have to be defined in context of the application

- RDBMS monitors those constraints
  - Referential Integrity (key/foreign-key)
  - CHECK Constraints in DDL
  - Trigger

- When to perform the checks
  - Operation wise
  - Transaction wise

# ACID Principle

**A**tomicity

- A transaction is either executed completely (commit) or not at all (abort)

**C**onsistency

- A transation transforms a consistent database state into a (possibly different) consistent database state

**I**solation

- A transaction is executed in isolation, i.e., does not see any effect of other concurrently running („uncommitted") transactions.

**D**urability

- A successfully completed („committed") transaction has a permanent effect on the database

Note: ACID is important for OLTP (online transaction processing) applications. Other applications (e.g., OLAP (online analytical processing) with fewer write operations often trade off the ACID principle with performance. Esp. true for Big Data.

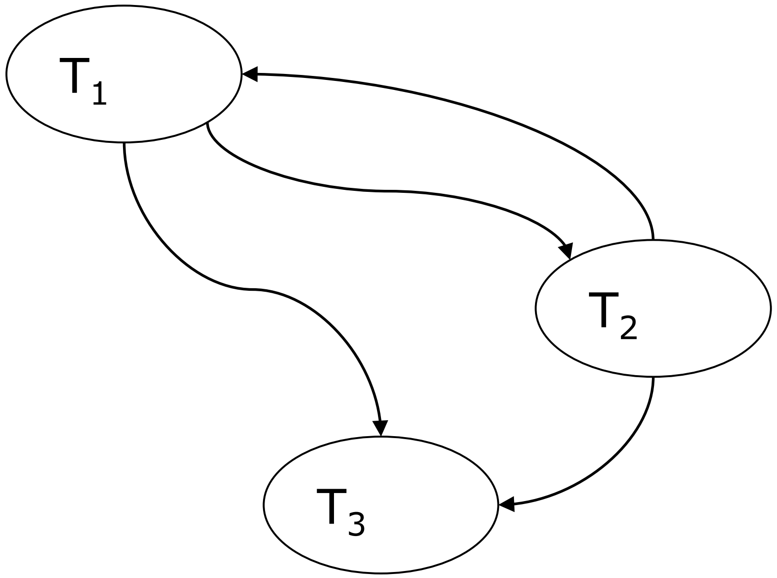
# Problems with Concurrent Access

- Nonrepeatable Read
  - A transaction T1 modifies a data item. Another transaction T2 reads the same item before T1 commits or rolls back. If T1 rolls back, T2 has read a value that never existed.
- Dirty Read
  - T1 reads a data item. T2 modifies or deletes the data item and commits. T1 attempts to reread the data item. It discovers another value or that the item has been deleted.
- Phantom-Problem
  - T1 searches using a < X < b. T2 creates some items that fall in the range (or updates items in the range so they do not qualify anymore). T1 repeats its search, and discovers a different set of items.
- Lost Update
  - Two transactions, T1 and T2, read a data item concurrently. T1 updates the item first and then T2, without considering T1‘s update. T1‘s update is lost.

# Transactions and Serializability

- Which operations are in conflict? How do determine if serializable?

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| r(y) | | |
| | | r(u) |
| | r(y) | |
| w(y) | | |
| w(x) | | |
| | w(x) | |
| | w(z) | |
| | | w(x) |




$$S = r_1(y)r_3(u)r_2(y)w_1(y)w_1(x)w_2(x)w_2(z)w_3(x)$$

# Locking Overview

- Serializability enforced by locking data items
- Lock manager: global in-memory data structure that keeps tracks of locks

- Two types of locks
  - Shared (S) lock: Used to protect read access
  - Exclusive (X) lock: Used to protect write access

- Schedule with locks:

*X1(A)*;R1(A);W1(A);*U1(A)*;*S2(A)*;R2(A);*S2(B)*;R2(B);
*U2(A)*;*U2(B)*;*S1(B)*;R1(B);*U1(B)*;*X2(B)*;W2(B);*U2(B)*

Lock requested

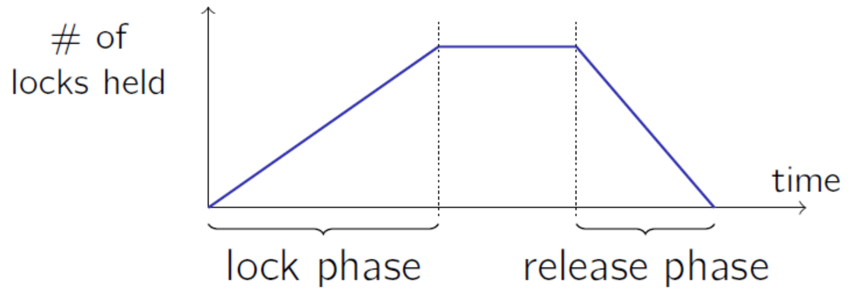
| Lock held | S | X |
|---|---|---|
| S | yes | no |
| X | no | no |

# Two Phase Locking (2PL)

- Transaction is well formed if
  - It holds an S or X lock on a data item while reading it
  - It holds an X lock on a data item while writing it

- Two phase locking (2PL)
  - Every transaction is well formed
  - Once a transaction has released a lock, it is not allowed to obtain any additional locks

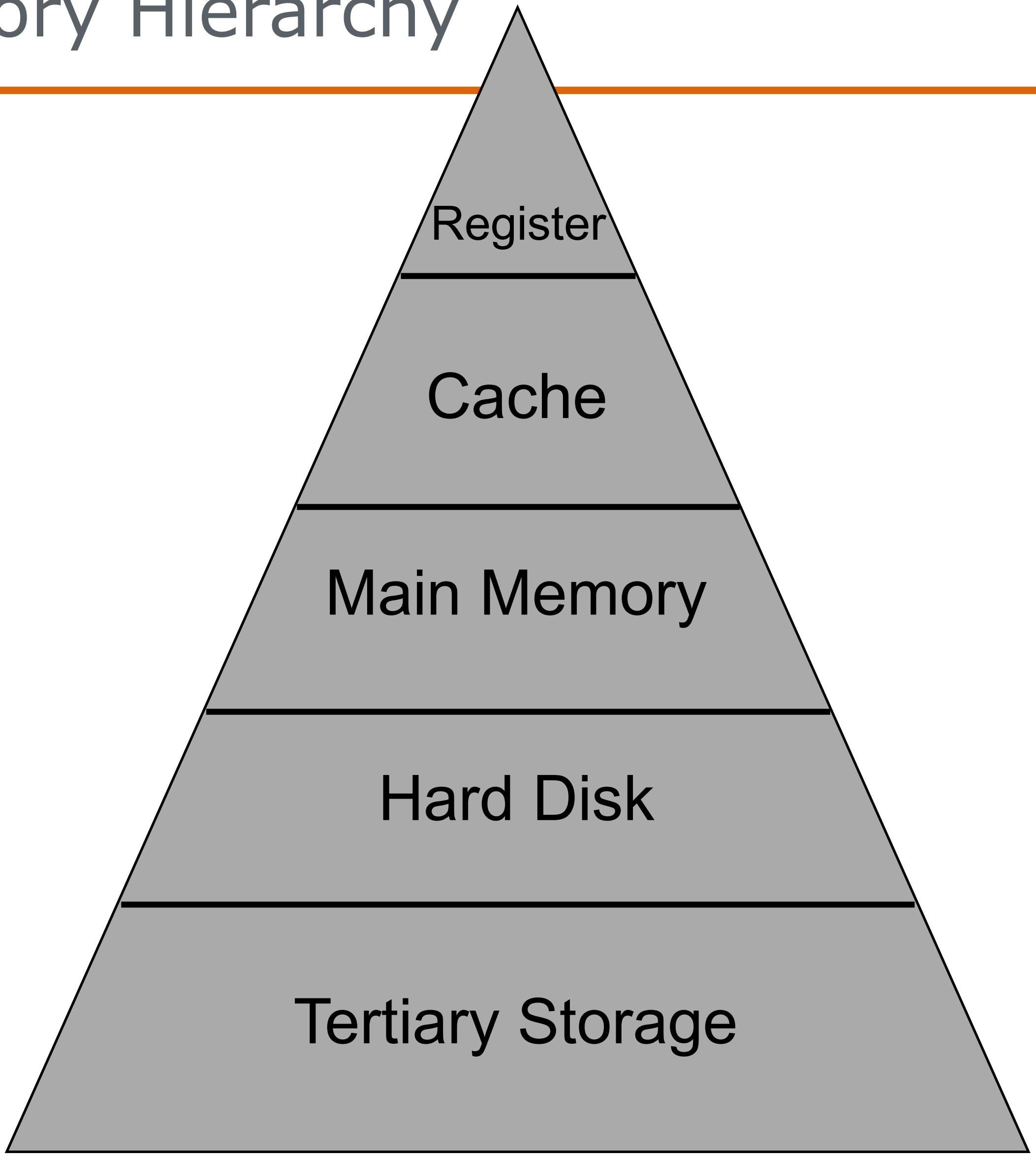
- Transactions have two phases
  - Growing phase: Acquiring locks
  - Shrinking phase: Releasing locks
  - Transition from growing to shrinking as soon as the first lock is released



# RDBMS Internals

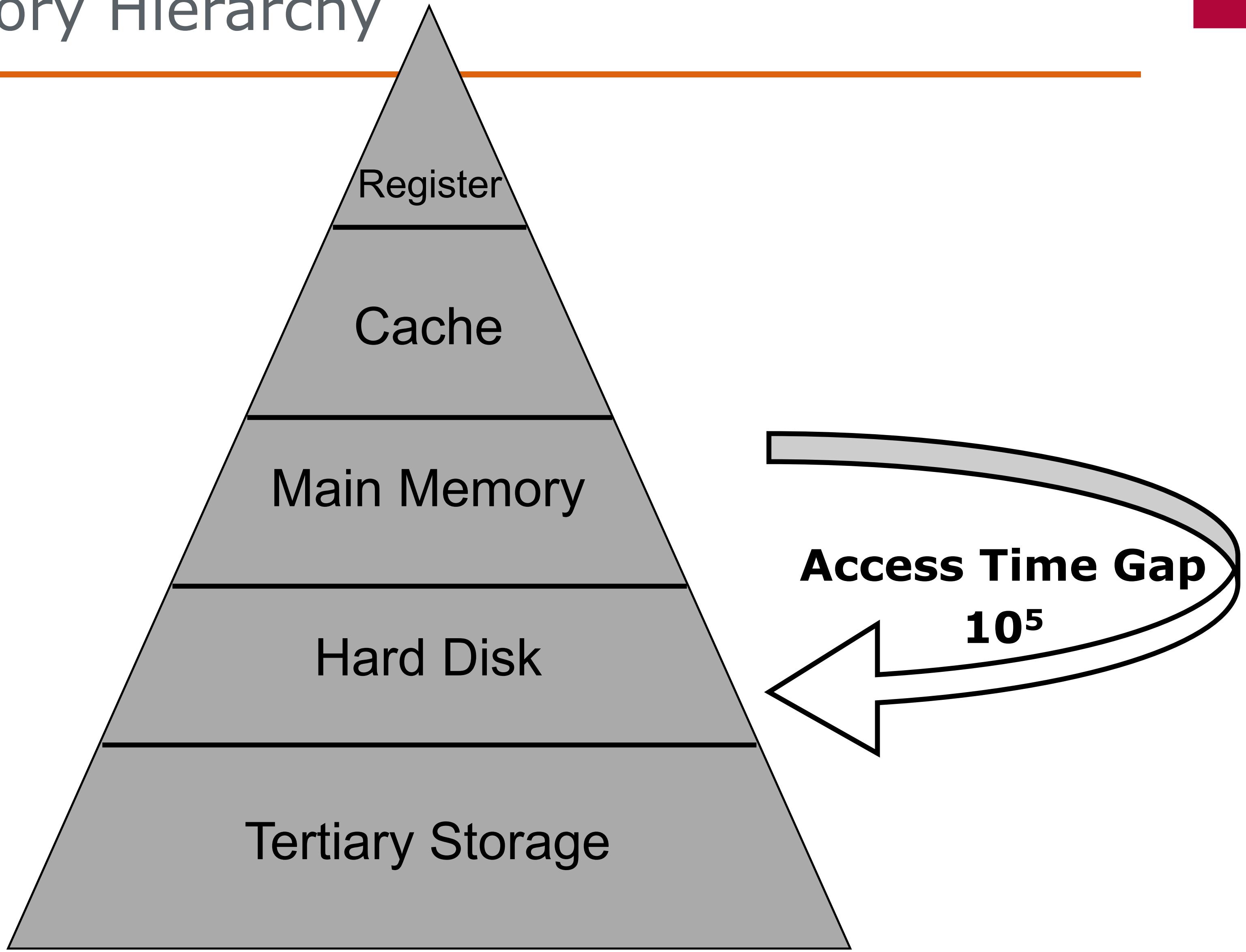
# Overview: Memory Hierarchy

- Very Expensive
- Very Expensive
- ~ 10 € / GB
- ~ 0.05 € / GB
- < 0.02 €/GB

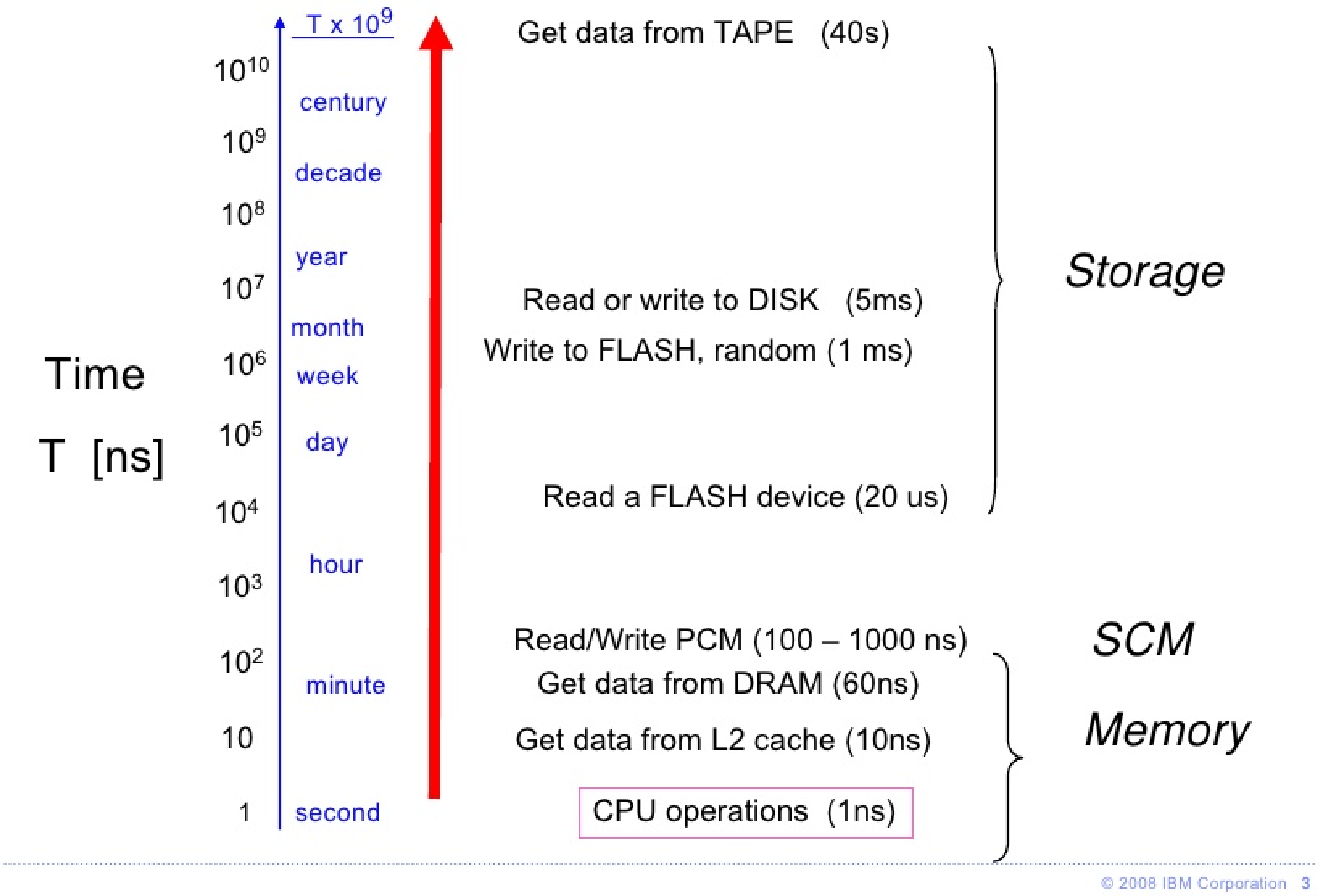


# Overview: Memory Hierarchy

- Head (1min)
- Room (10 min)
- Berlin (1.5h)
- Pluto (2 years)
- Andromeda (2000 years)

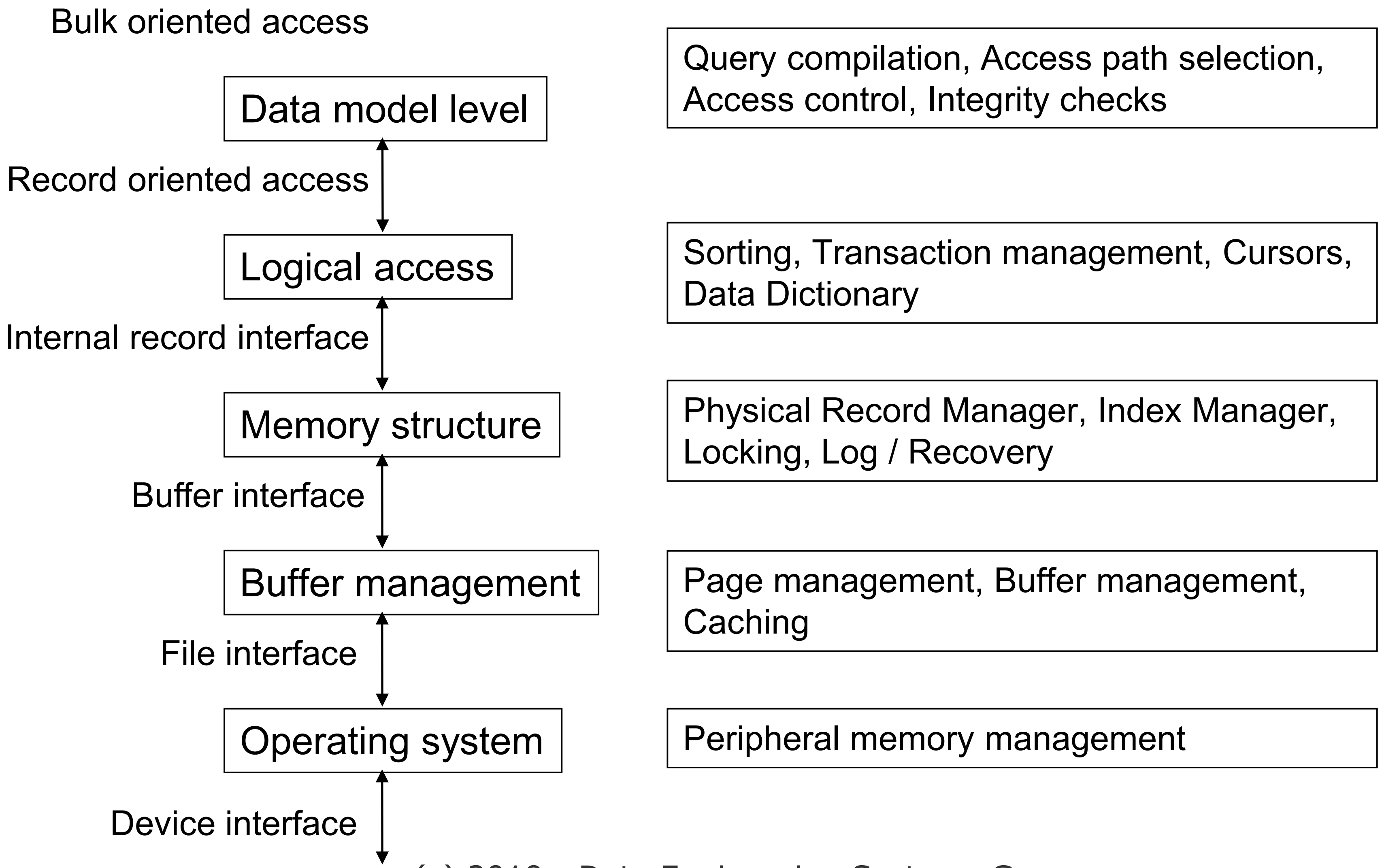


# More Modern View (by IBM)




https://www.slideshare.net/Flashdomain/flash-and-storage-class-memories-technology-overview

# 5 Layer Architecture

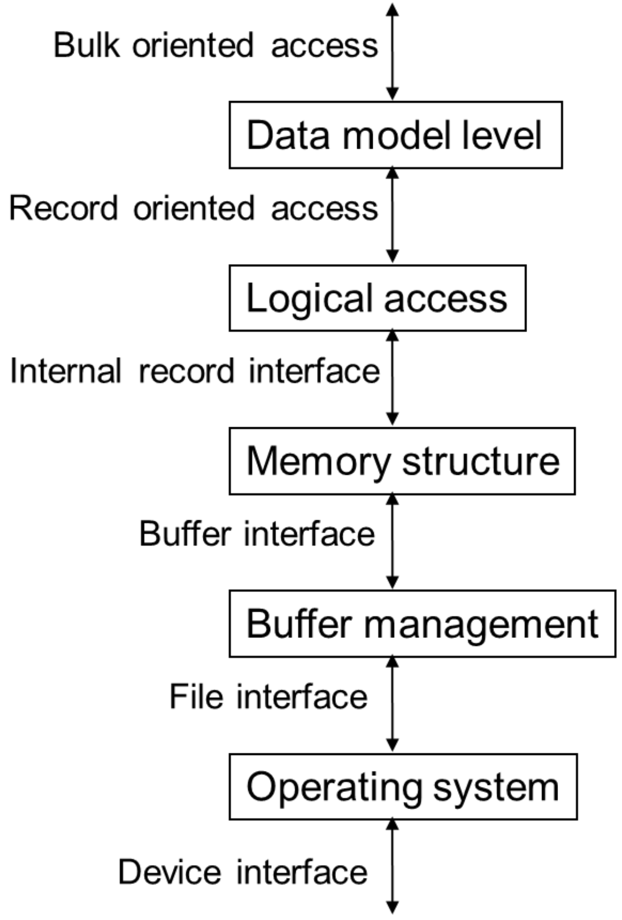


# Objects and Operations

| Interface | Level | Objects and operations |
| --- | --- | --- |
| Bulk oriented access | Data model level | SQL: select ... from ... Where<br>Grant access to ...<br>Create index on ... |
| Record oriented access | Logical access | OPEN – FETCH – CLOSE (Tab o. Index)<br>STORE record |
| Internal record interface | Memory structure | STORE record on pages, UPDATE all access paths, B*-Tree |
| Buffer interface | Buffer management | READ page<br>WRITE page |
| File interface | Operating system | Disc driver<br>MOVE head ... |
| Device interface | | |

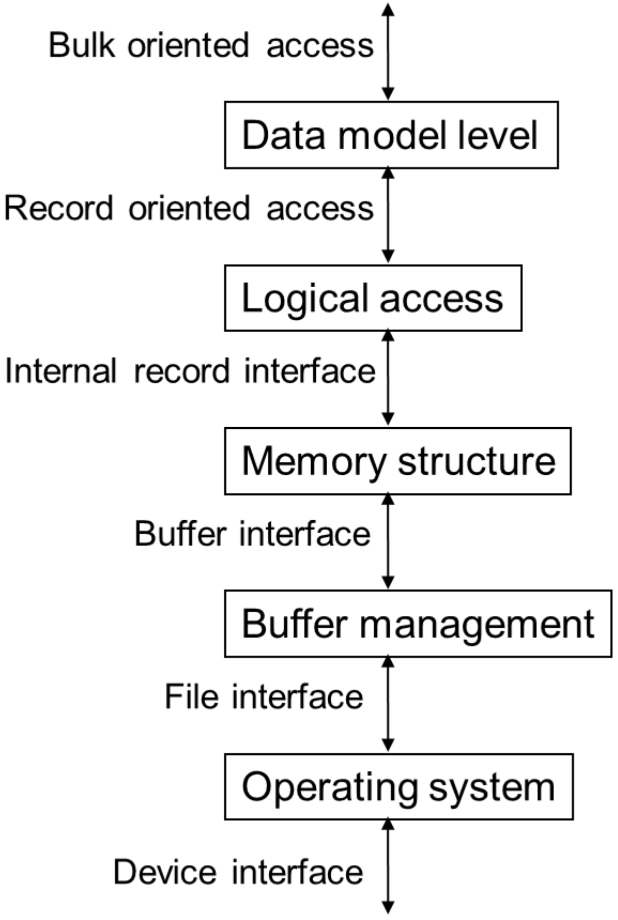
# Interfaces

- Set-oriented interface
  - Access to sets of tuple by a declarative language
  - SELECT … FROM … WHERE …
  - Monitoring of data integrity and authorization
- Record-oriented interface
  - Access to typed tuple
  - Access through logical access paths (Indexes, Scans)
  - Open/Next/Close Interface
  - Partition management
- Generic record interface
  - Access to uniform and un-typed tuple
  - Locking
  - Mapping tuples (logical objects) to pages



# Interfaces

- Buffer interface
  - Uniform access to all blocks within the virtual address space
  - Mapping of virtual block addresses to physical block addresses
  - Synchronization of blocks (cache management, concurrent access) ("locking", but different to "transaction locks", often called "latching" or "pinning")
- File interface
  - Access to physical blocks
  - Managing the mapping between block and segment, tablespaces, files
  - Software-RAID
- Device interface
  - Access to hard drive data
  - Addressing discs – Disc, Track, Sector
  - Controller cache, Prefetching
  - Hardware RAID



# 5 Layer Architecture

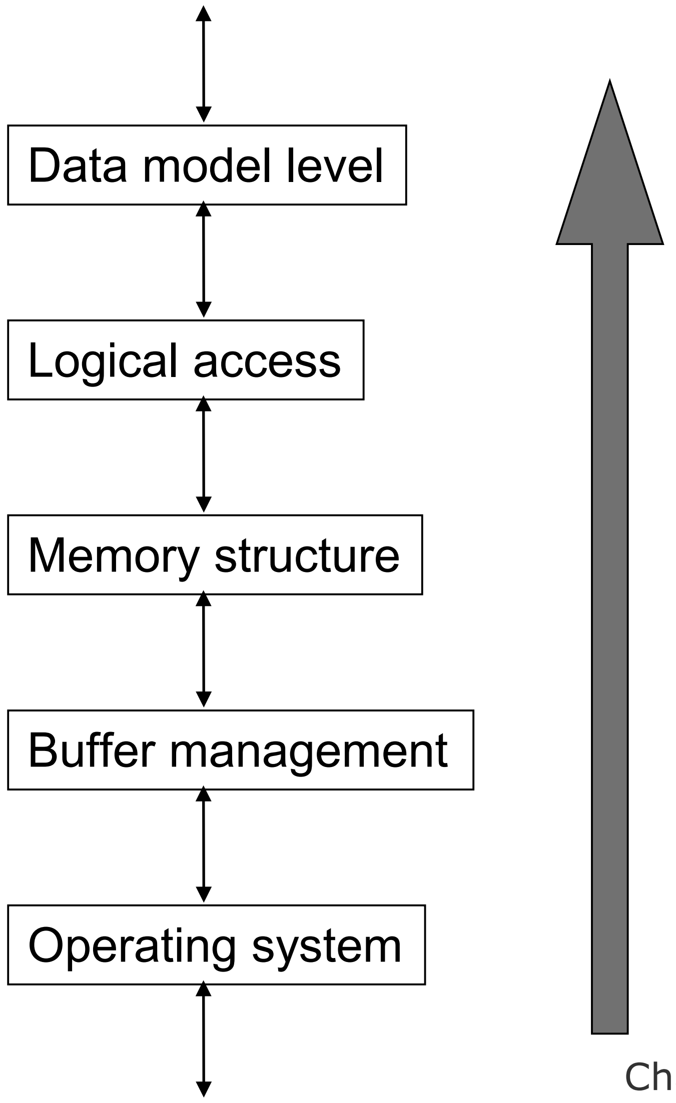
- Idealized representation
  - No need to strictly stick to that model
  - Some techniques cut through layers, e.g., synchronization, recovery

- Combination of layers is possible
  - E.g. „Record oriented and internal record interface“

- Often a direct access to another layer
  - Prefetching: Caching needs information about the actual workload; not only about the actual tuple
    - From layer logical record layer to buffer/OS layer
    - Perhaps from data model layer to buffer/OS layer
  - The optimizer needs information about physical allocation of blocks From OS layer to logical record/data model layer
  - Thus: In many DBMS implementations, the principle of „Information Hiding“ is not 100% adhered to for performance reasons

# Bottom Up

- Many topics cannot simply be associated with a single layer
  - Locking
  - Recovery
  - Request optimization
  - …



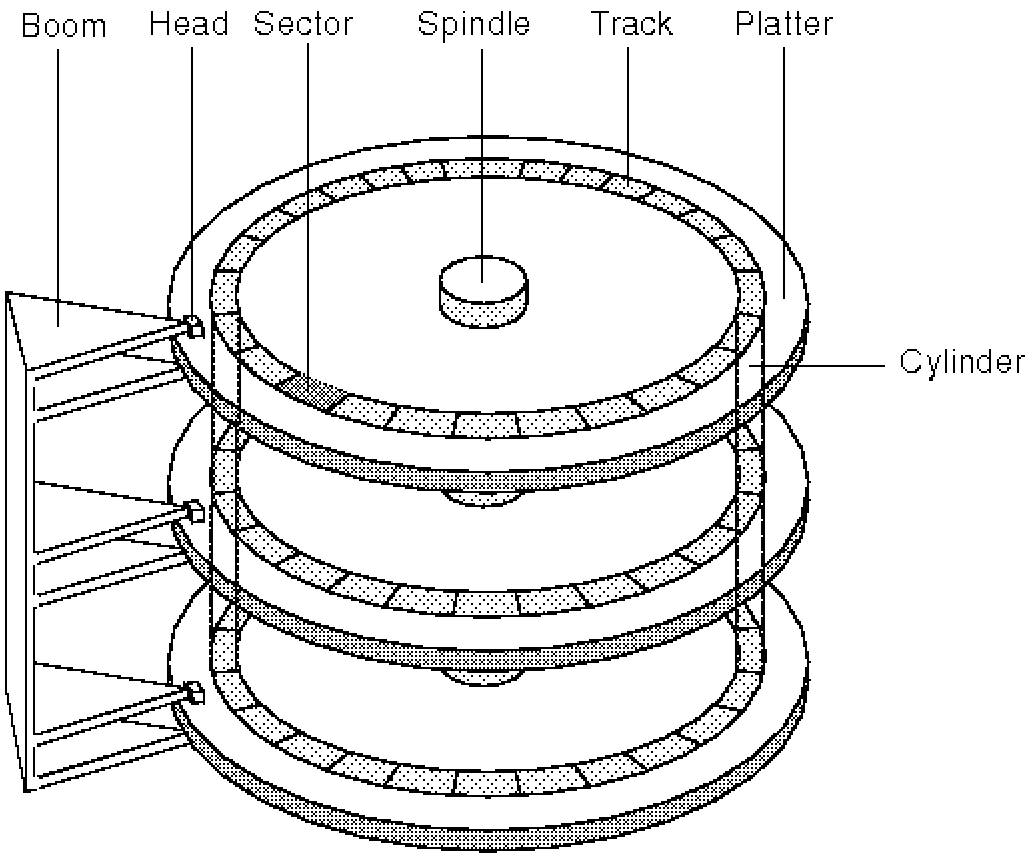
# Magnetic Hard Disk

- Access time for a disk page:
  - Positioning time (track) (~4-8 ms)
  - Rotational delay (sector)(~8 ms for 7200 rpm)
  - Transfer time (sector) (> 1 GB/s)

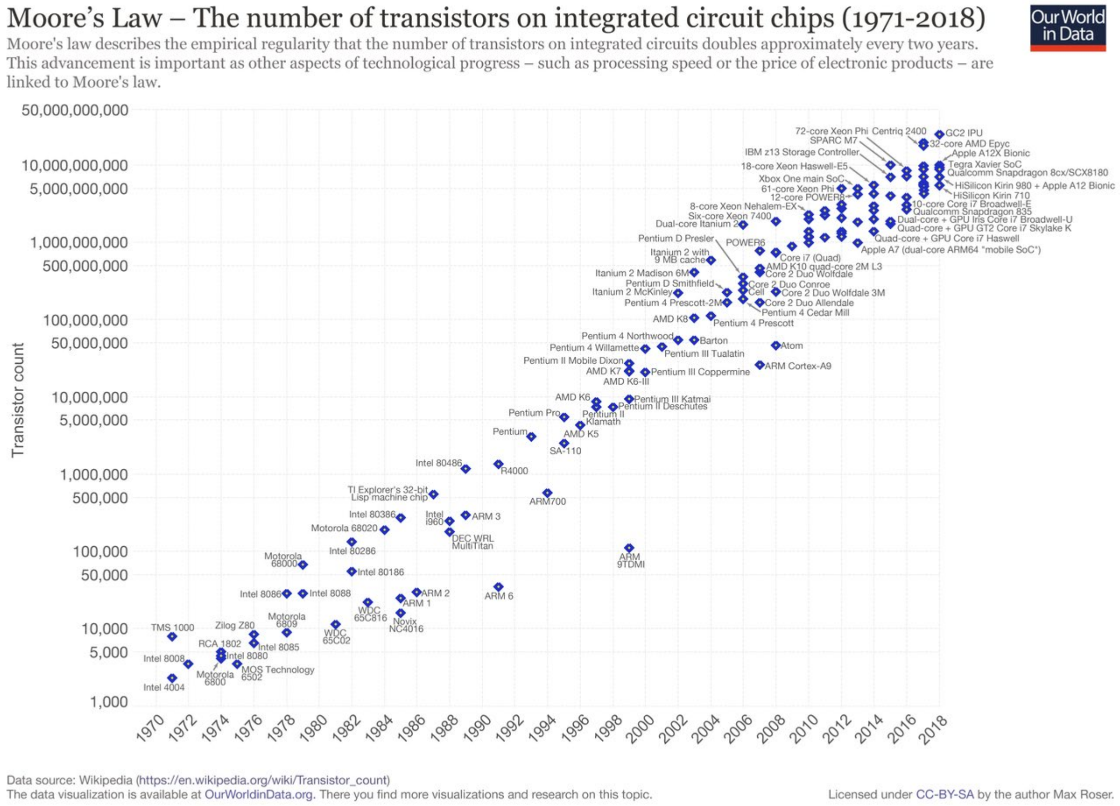
- Distinction
  - random I/O
  - sequential I/O

- disk page# =

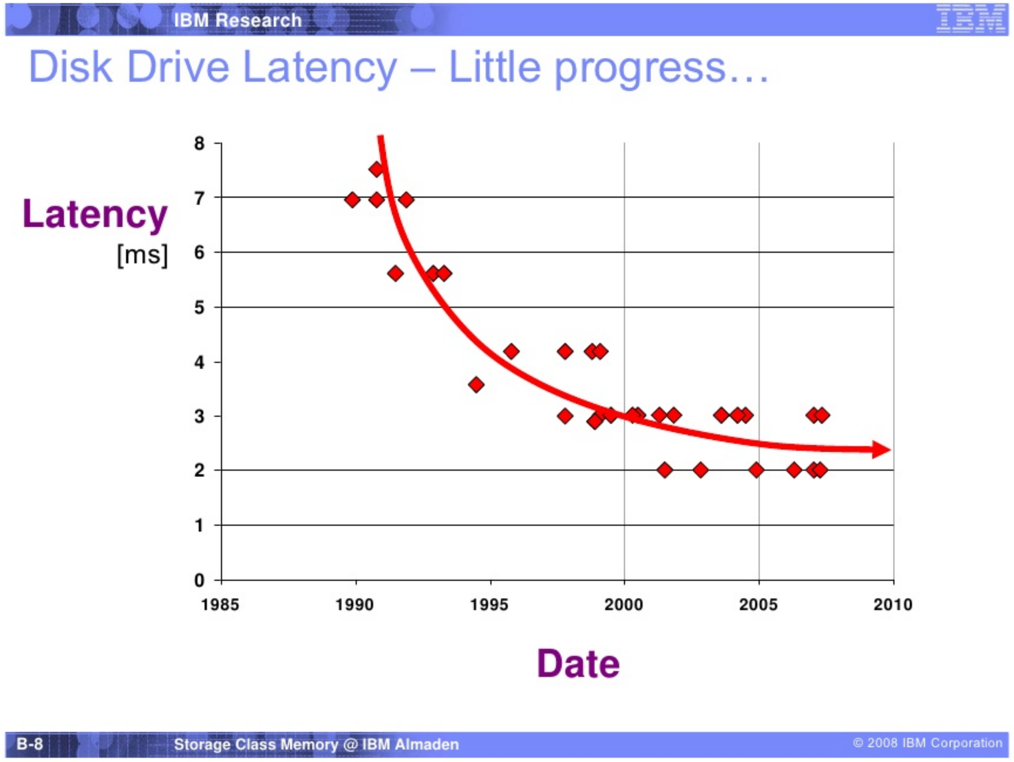
  f(cylinder#, platter#, track# , sector#)
  - usual size: (2, 4 ,8, 16, 32, 64, 128 kB)



# Disk vs. CPU



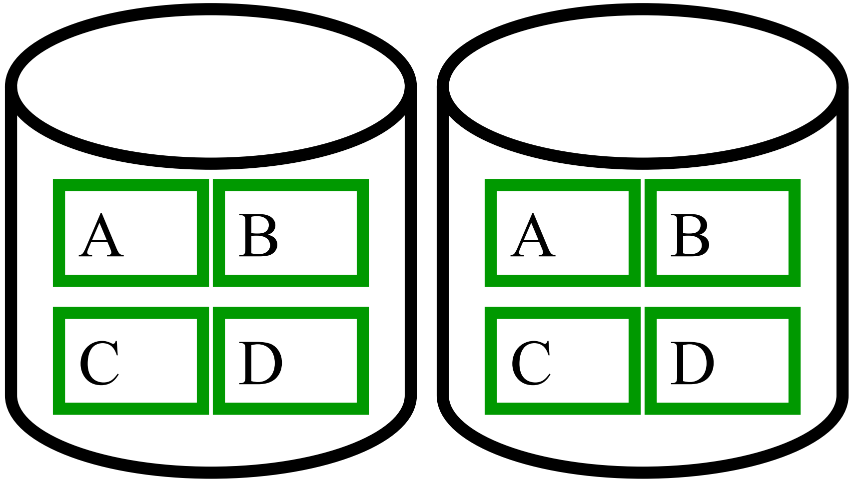

Transistors per CPU




https://www.slideshare.net/Flashdomain/flash-and-storage-class-memories-technology-overview

Disk Speed

# RAID 1

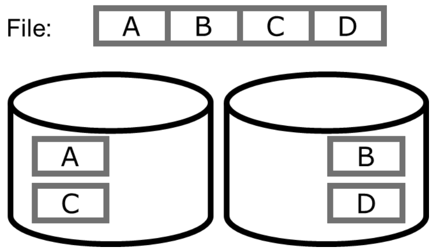



- Data security: redundancy of all data (mirror)
  - But no help when bit errors occur – who‛s right
- Double amount of capacity will be needed
- Load sharing when reading: e.g. block A can be read from the left or the right hard drive
- But upon write accesses, both copies must be written
  - It may be parallelized
  - The needed time is the same as writing on a single hard disk

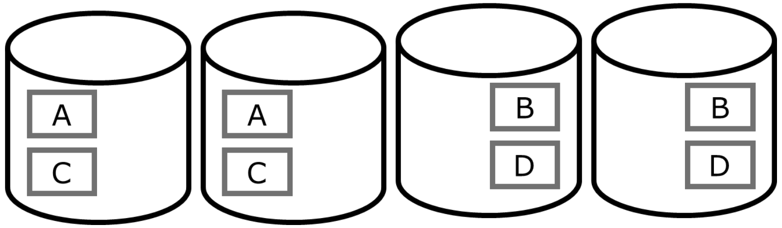
# Other RAID Levels



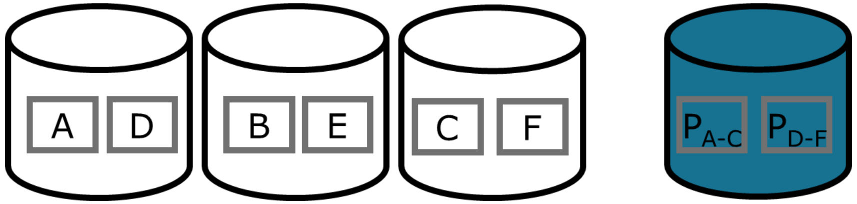
- RAID 0: Block-level striping



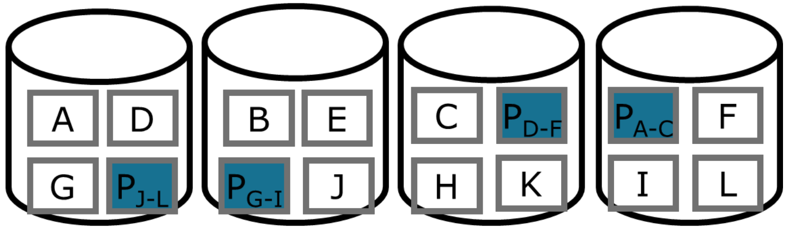

- RAID 0+1: Mirrored striping




- RAID 4: Block-level striping with parity disk




- RAID 5: Block-level striping with distributed parity



# Introduction to Access Methods

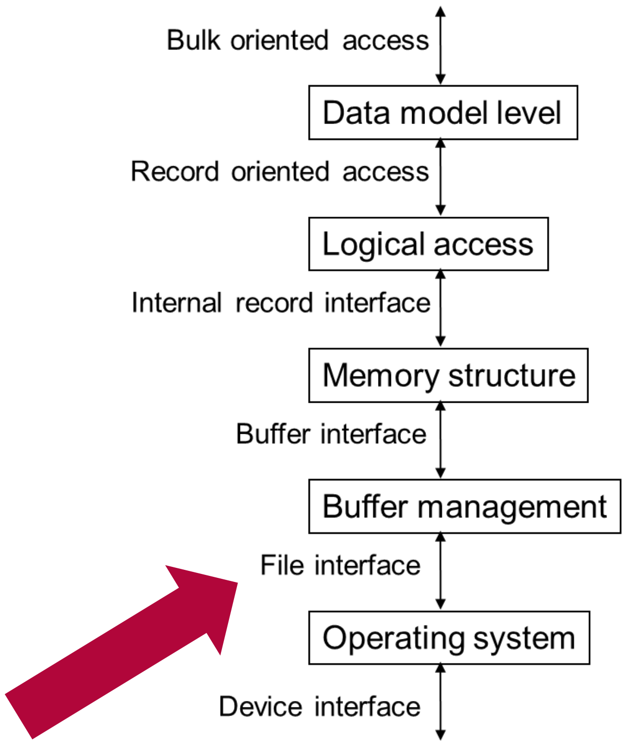
Sequential File

- Access to records by record/tuple identifier (“RID” or ”TID”)

| | | |
|---|---|---|
| 1522 | Bond | ... |
| 123 | Mason | ... |
| ... | ... | ... |
| 1754 | Miller | ... |

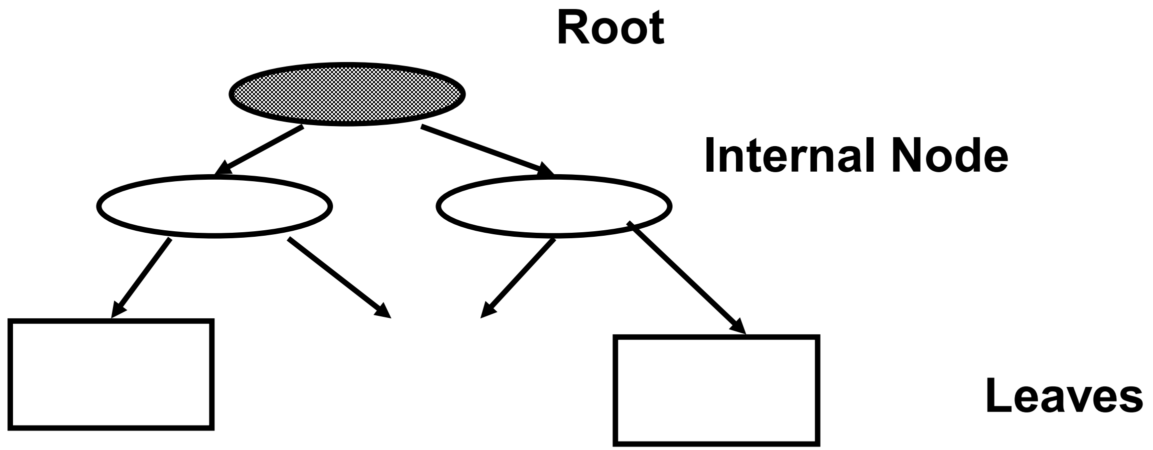
Operations:

- INSERT(Record): Move to end of file and add, O(1)
- SEEK(TID): Sequential scan, O(n)
  - FIRST (File): O(1)
  - NEXT(File): O(1)
  - EOF (File): O(1)
- DELETE(TID): Seek TID; flag as deleted
- REPLACE(TID, Record):Seek TID; write record



# Introduction to Access Methods 2

- Index File
- Access by search key (note: not necessarily data model key)



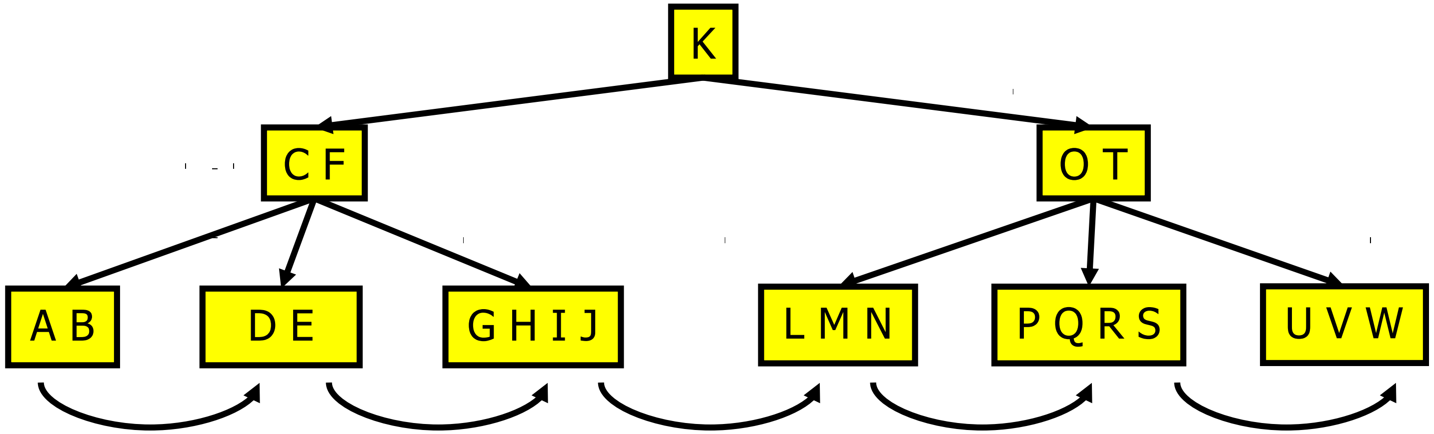

- Operations:
  - SEEK( key):Use order in TIDs; O(log(n))
    - Only if tree is perfectly balanced
  - INSERT( key): Seek key and insert; might require restructuring
  - DELETE( key):Seek key and remove; might require restructuring
  - REPLACE( key):Seek key and write
    - Variable size keys?

# Indexing: B and B* Trees

Tree with degree m

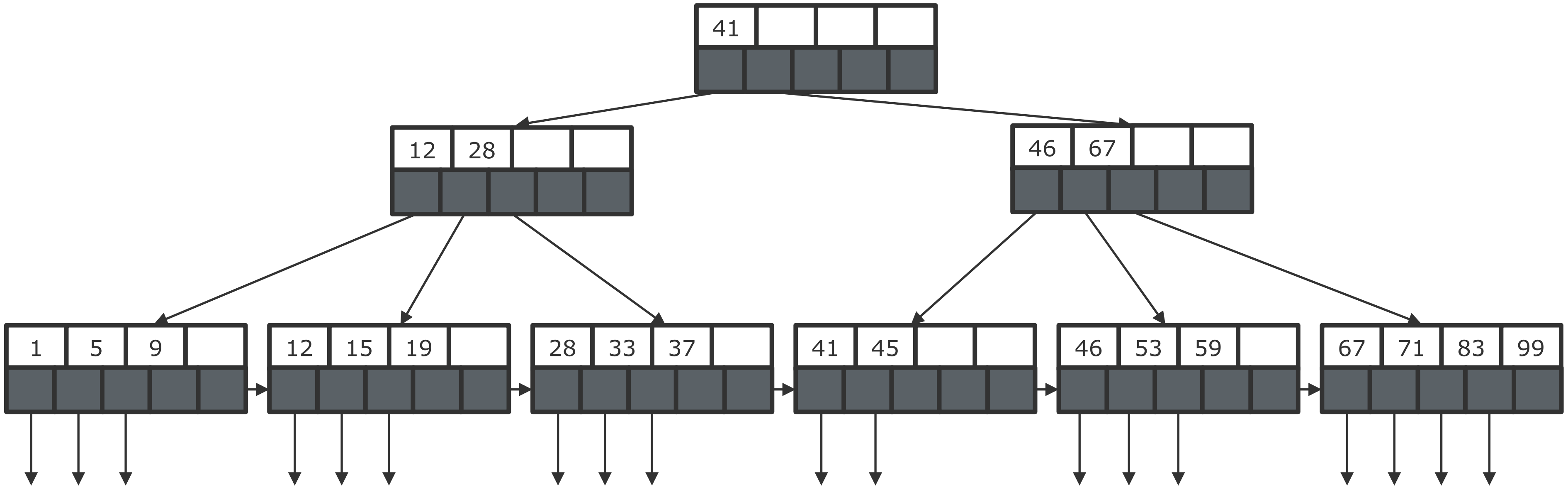
- Nodes have at most 2m keys
- Nodes have at least m keys, the root at least 1 key
- Node with x keys has x+1 children
- Balance: All leaves have the same depth

B*-Tree: data only in leaves, intermediate nodes only store separator of search key



# Example

- $n$ = 2
  - All nodes: At most 4 keys and 5 pointers
  - Root: At least 1 key and 2 pointers
  - Inner Nodes: At least 2 key and 3 pointers
  - leaves: At least 2 keys and 3 pointers

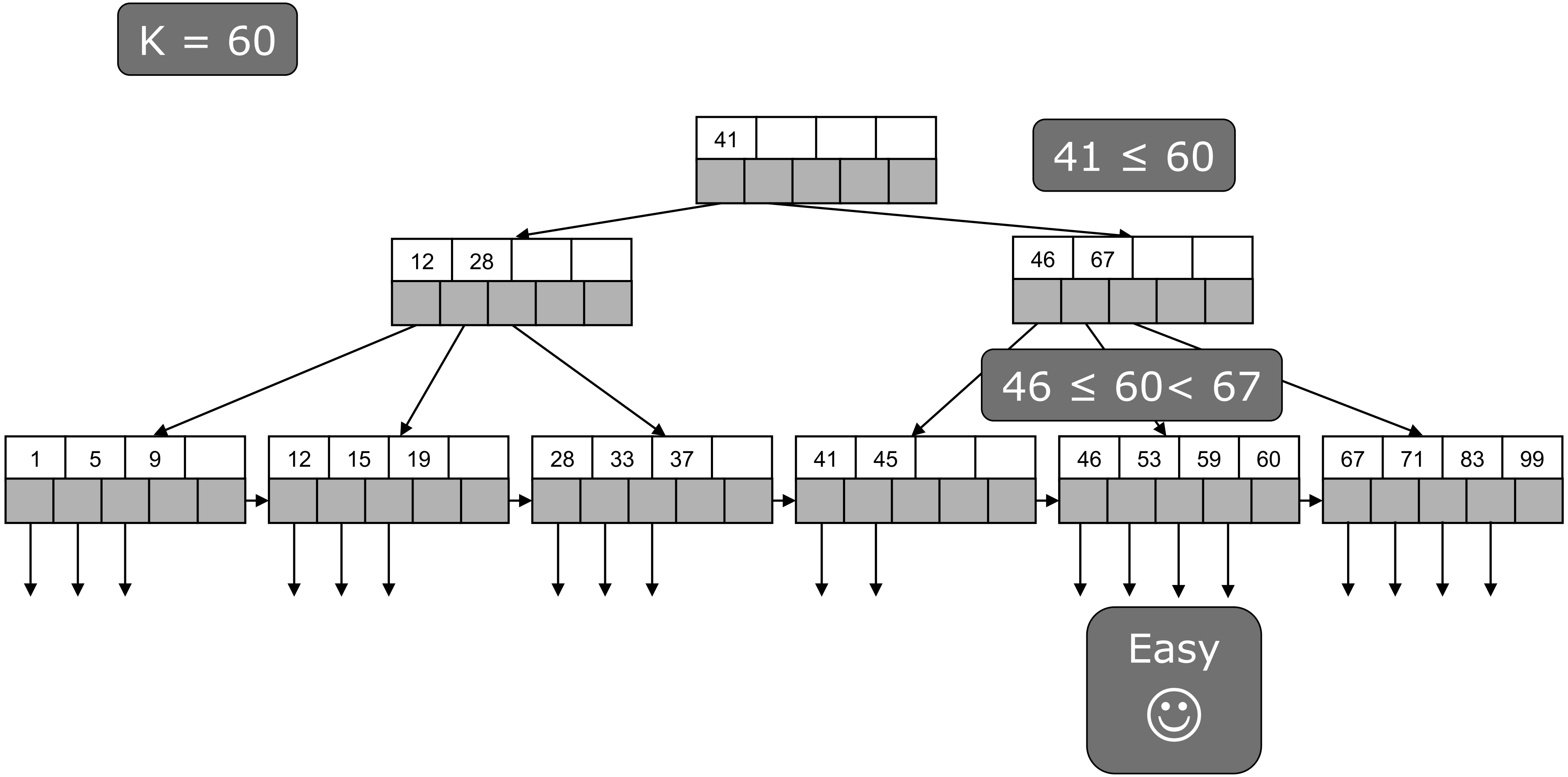


# Inserting into a B-Tree

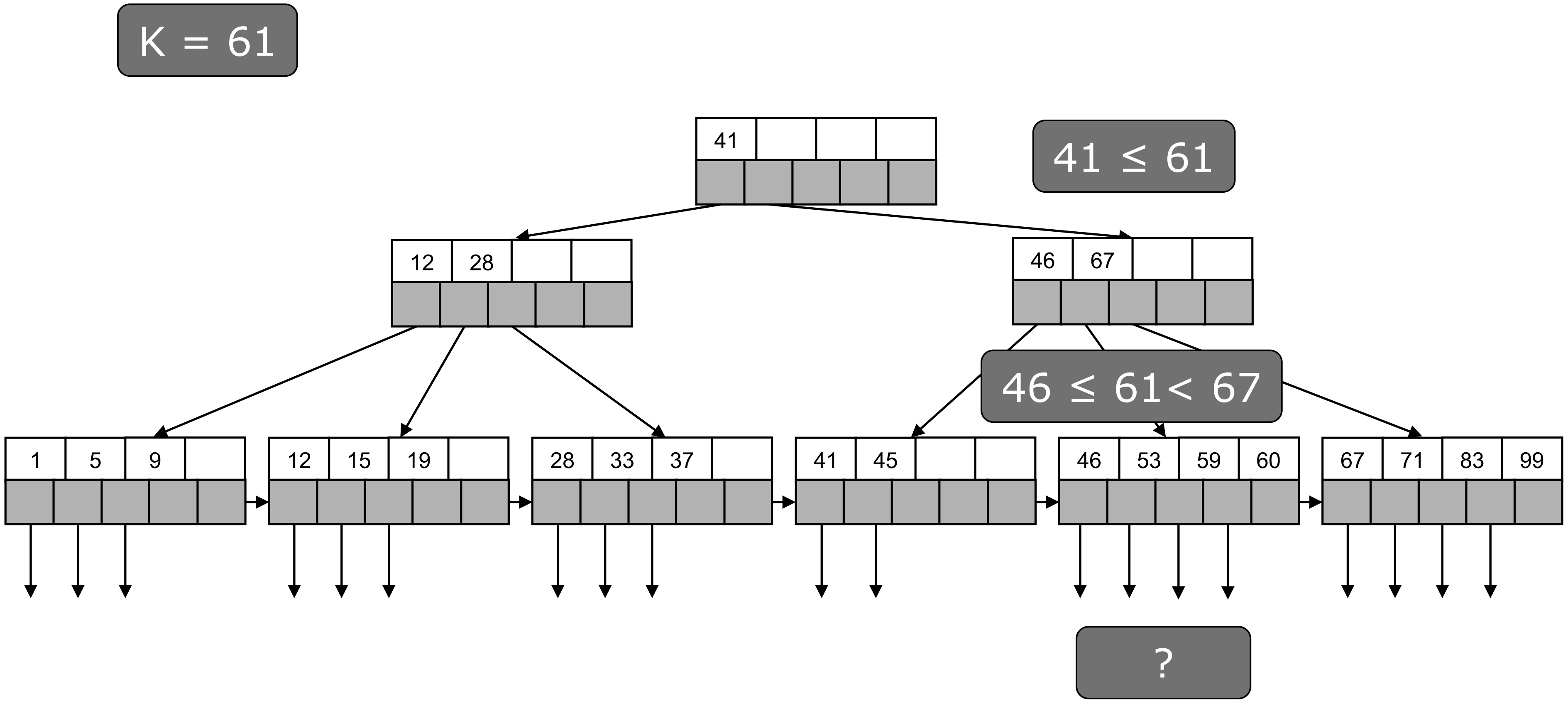
Recursive Algorithm:

- Search corresponding leaf.
  - If room, insert key and pointer.
- If no room: Overflow
  - Split leaf in two parts and distribute keys equally
- Split requires inserting a new key/pointer pair in parent node
  - Recursively ascend the tree
- Exception: If no space in root
  - Split root
  - Create new root (with only one key)

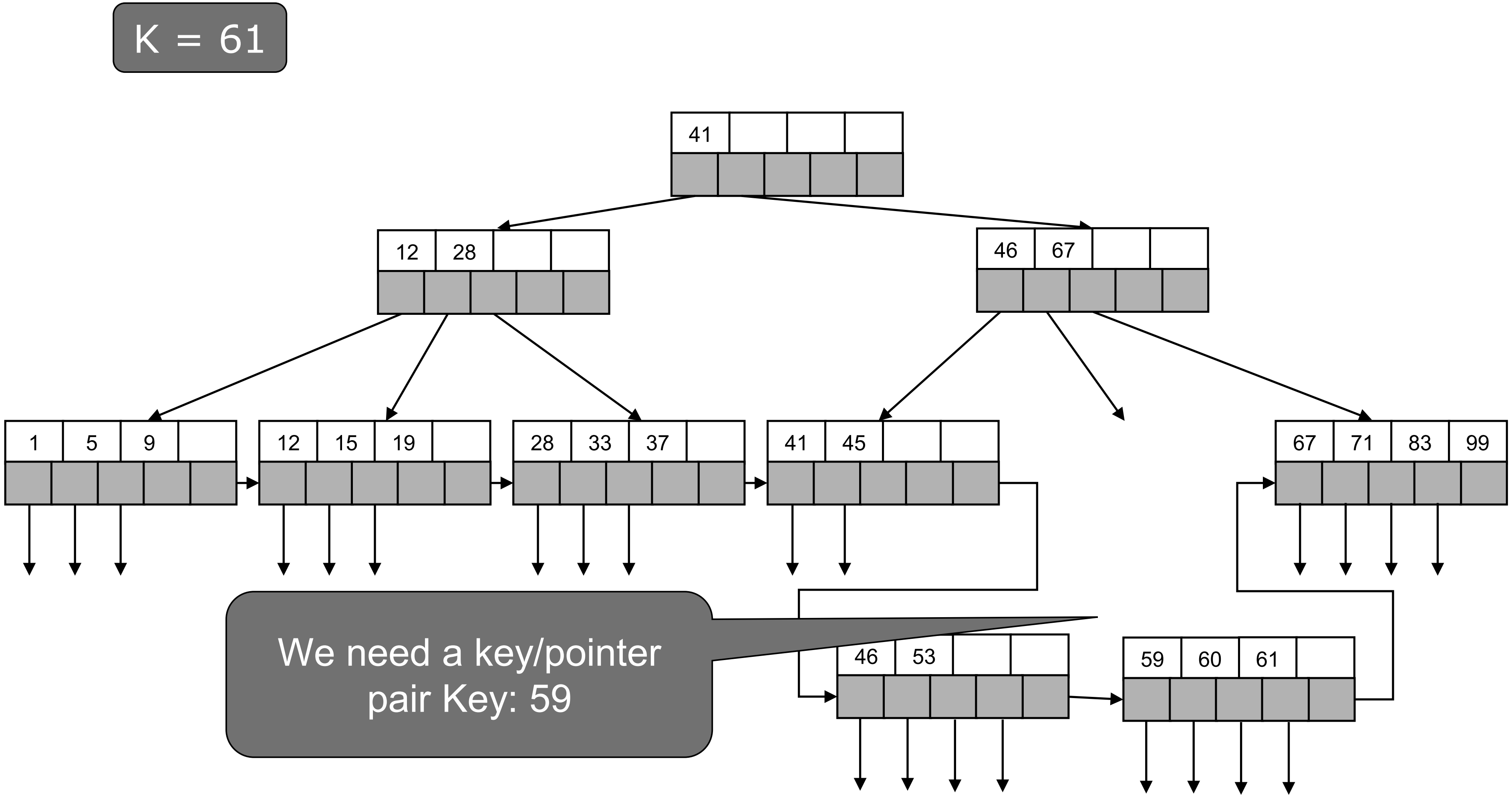
# Example of B-Tree Insertion



# Example of B-Tree Insertion

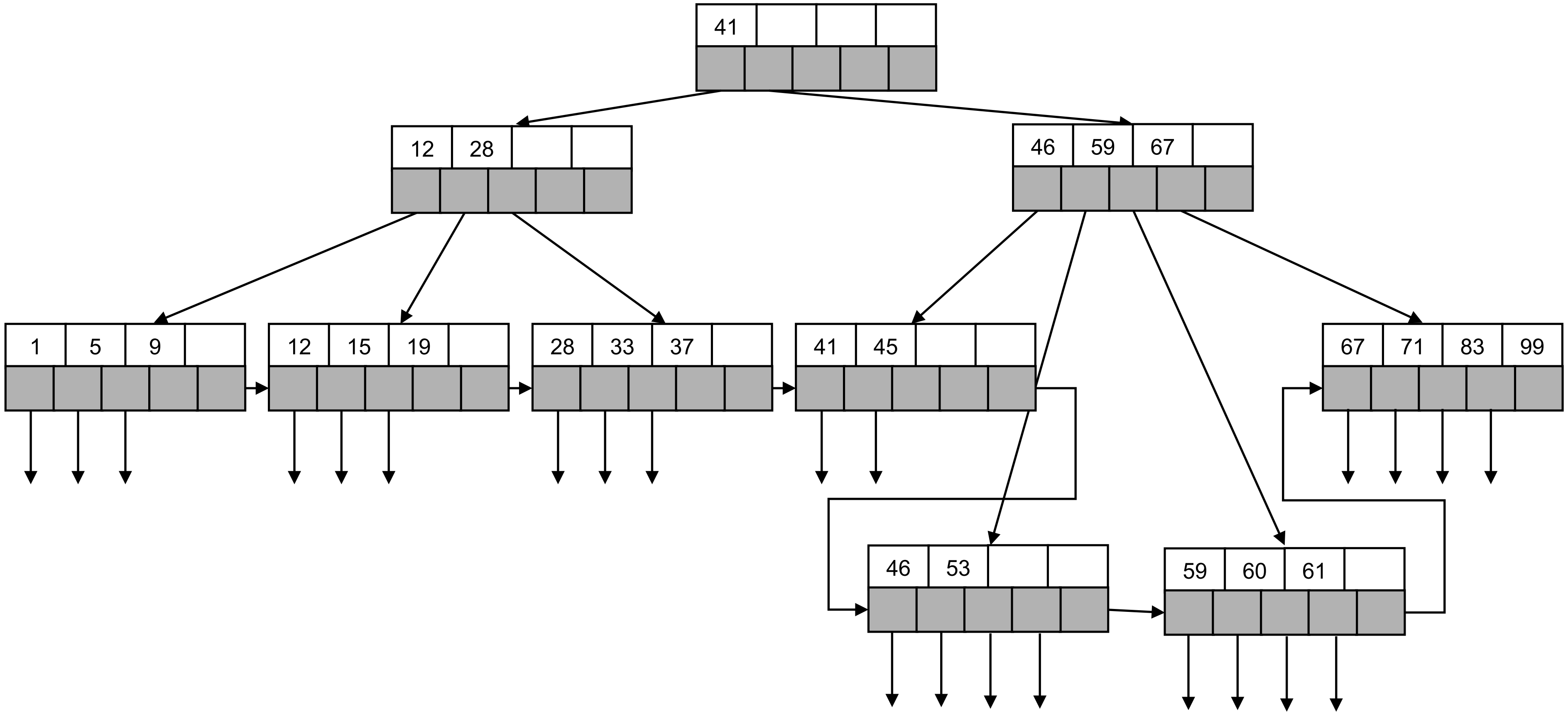


# Example of B-Tree Insertion



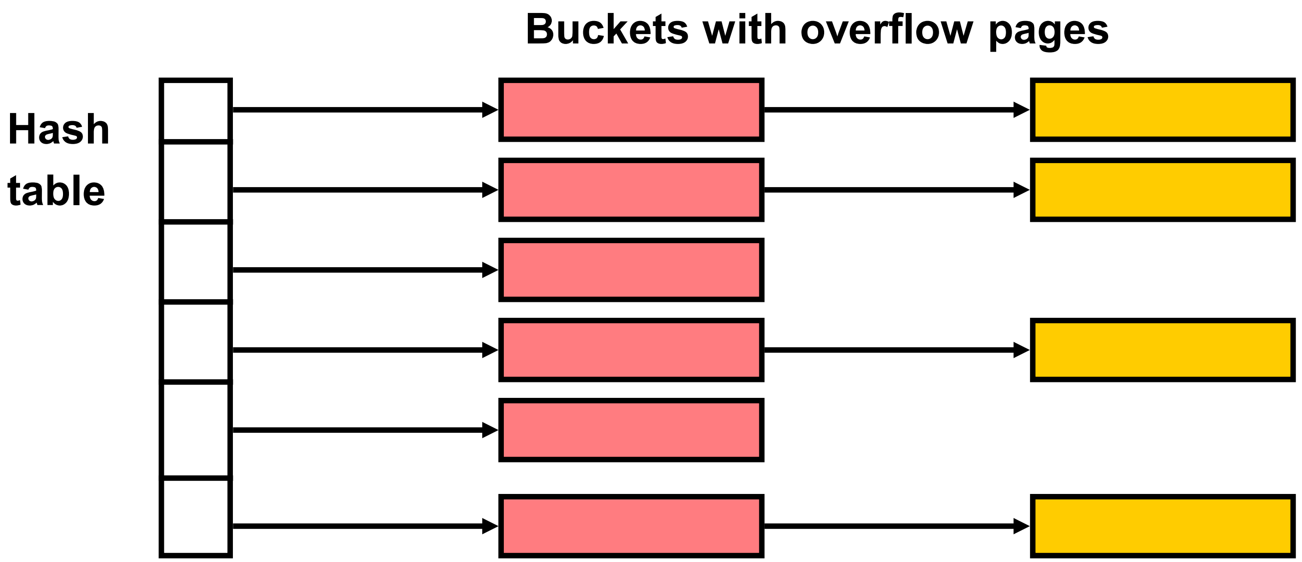
# Example of B-Tree Insertion

K = 61



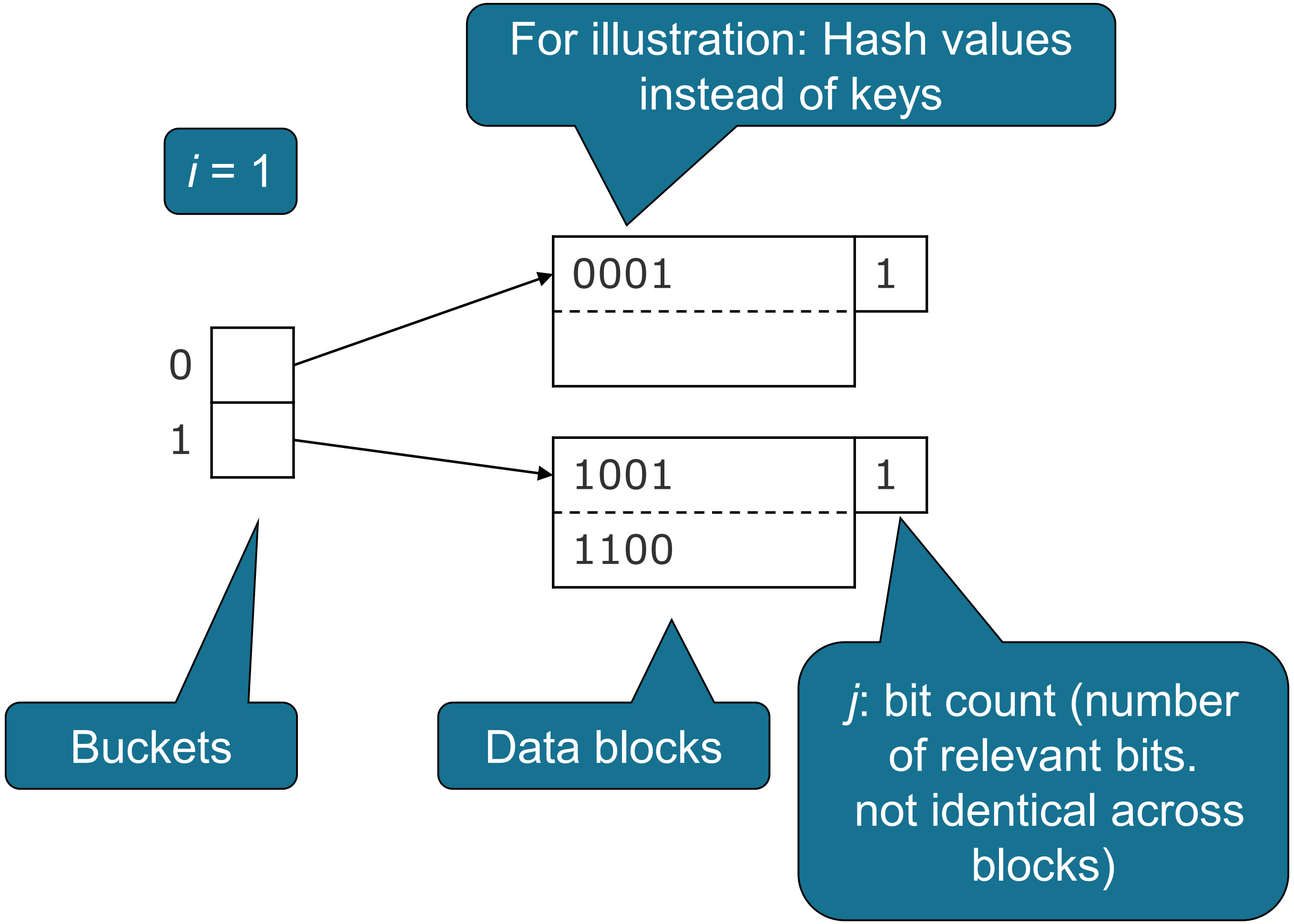
# Hashing

- Hash file consists of
  - Set of buckets (one or more pages)
    - $B_0$, $B_1$, ..., $B_{m-1}$, m>1;
  - A hash function h(K) = {0 ,...m-1 } on a set K of keys;
  - A hash table (bucket directory) as array of size m with pointers to buckets
- Hash files are structured according to one attribute value only



# Hashing 2

- Extensible Hashing



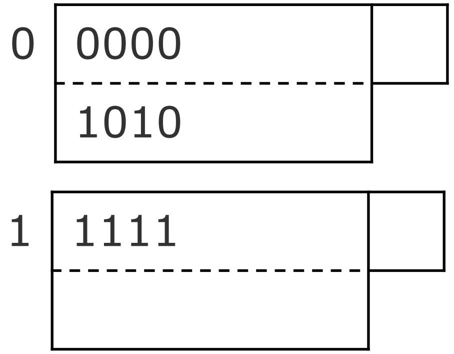

- Linear Hashing

*i* = 1 (number of relevant bits)

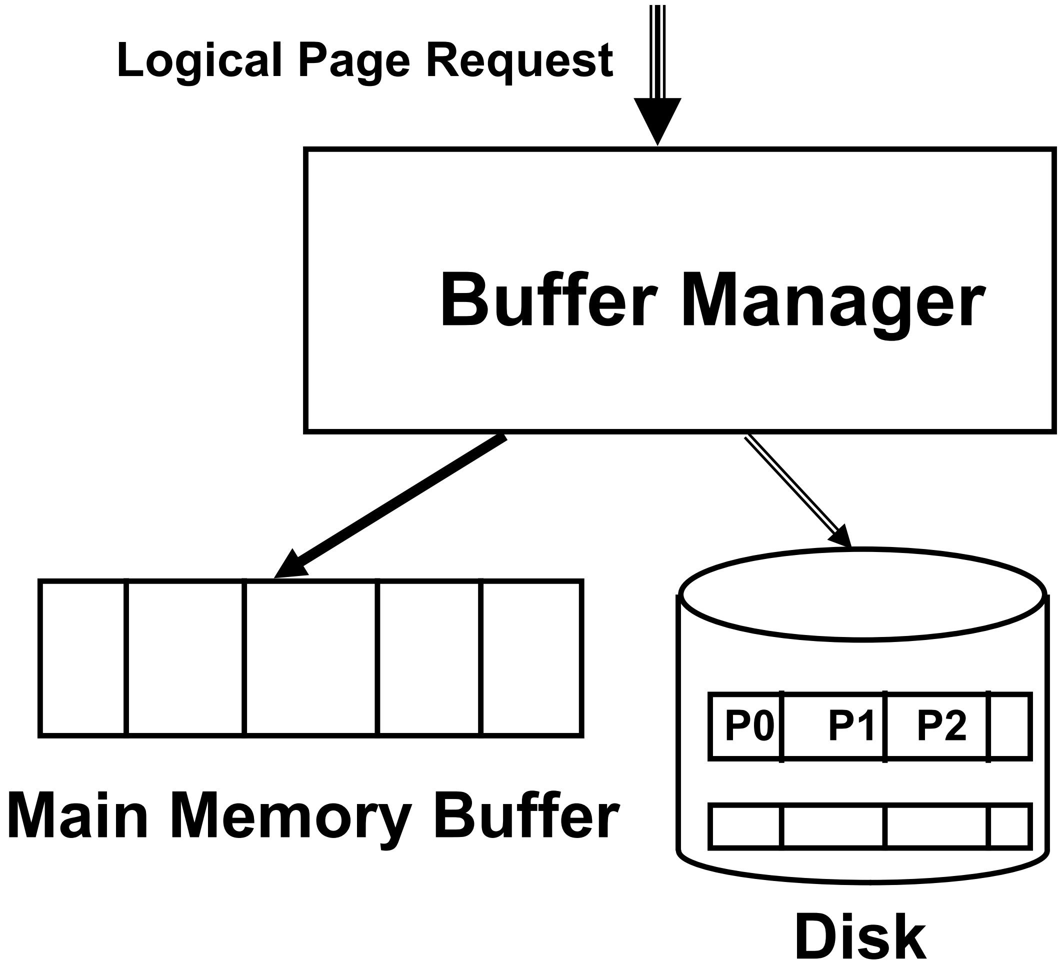
*n* = 2 (number of buckets)

*r* = 3 (number of records)

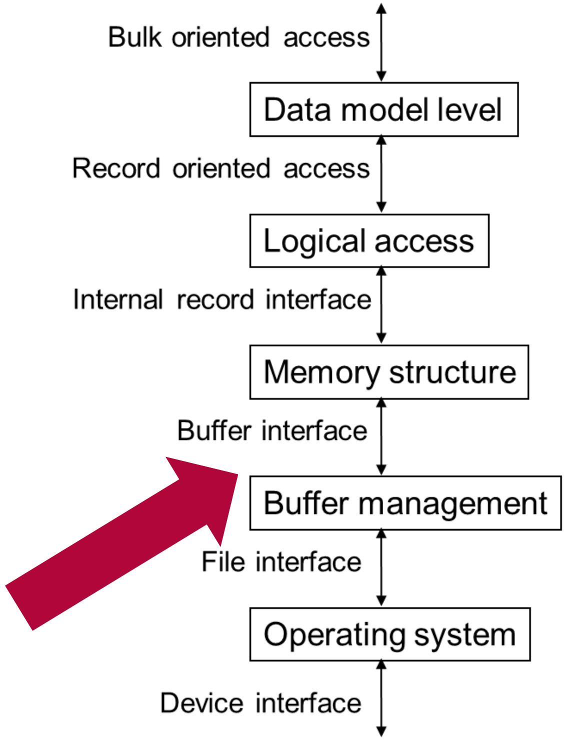
Choose *n* such that $r \leq 1{,}7 \cdot n$ (for block size 2)



# Caching = Buffer Management




- First-In-First-Out (FIFO)
  - Replace oldest block
- Least Recently Used (LRU)
  - Replace block with oldest access timestamp
- CLOCK
  - Fast approximation for LRU
- Least Frequently Used (LFU)
  - Replace block with smallest access count
- Least Reference Density (LRD)
  - Replace block with smallest reference count

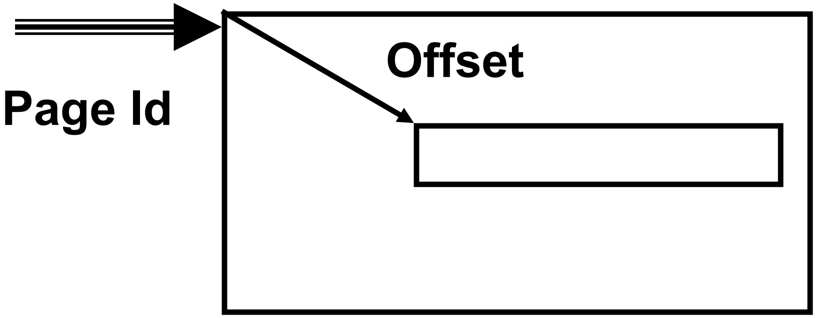


# Record Addressing

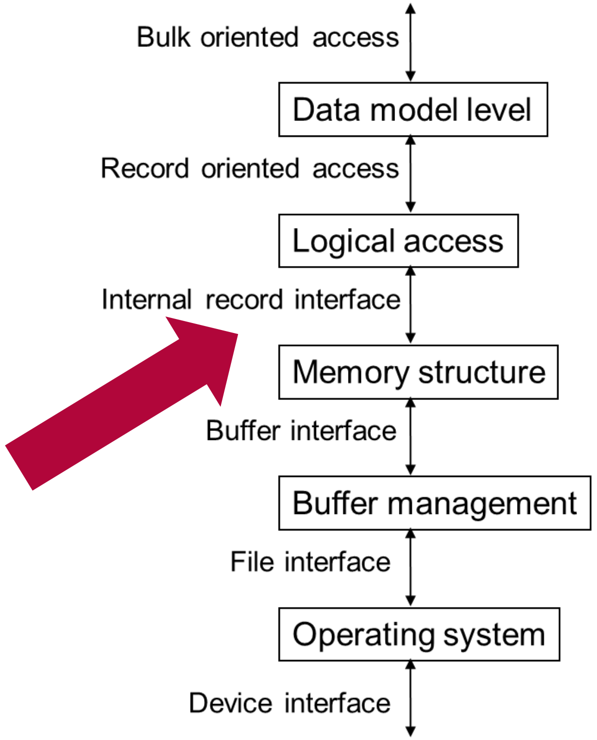


Mapping alternatives

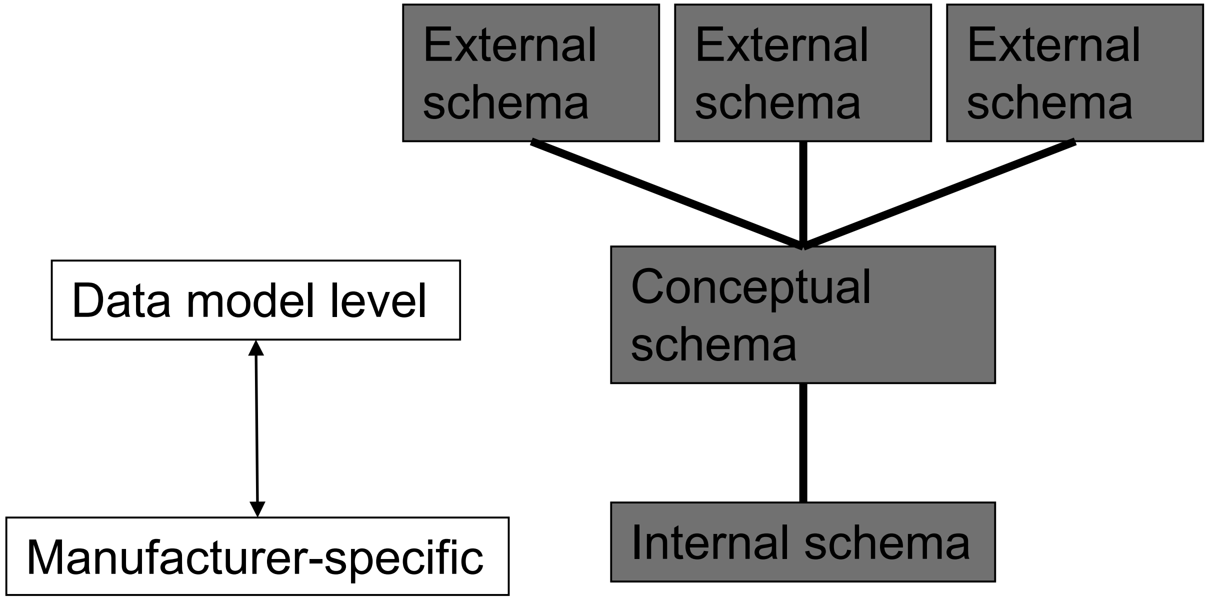
- absolute addressing: rid = <PageId, Offset>




- absolute addressing + search: rid = <PageId>

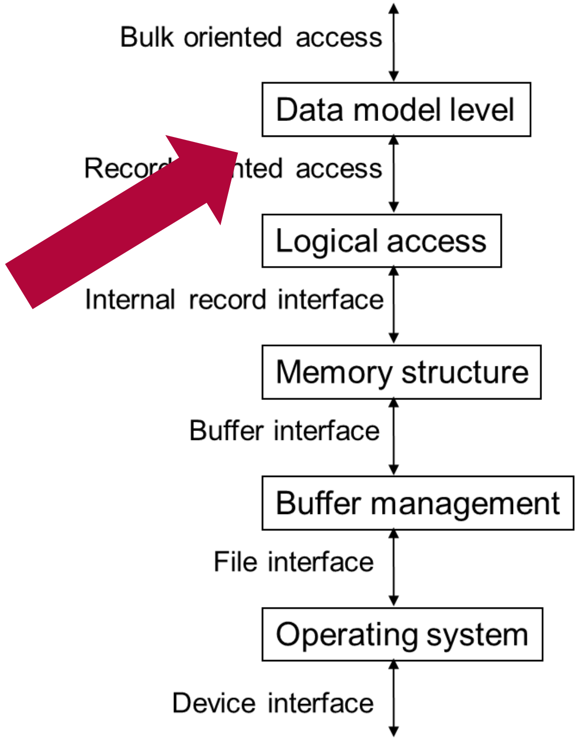
# Three Layer Model



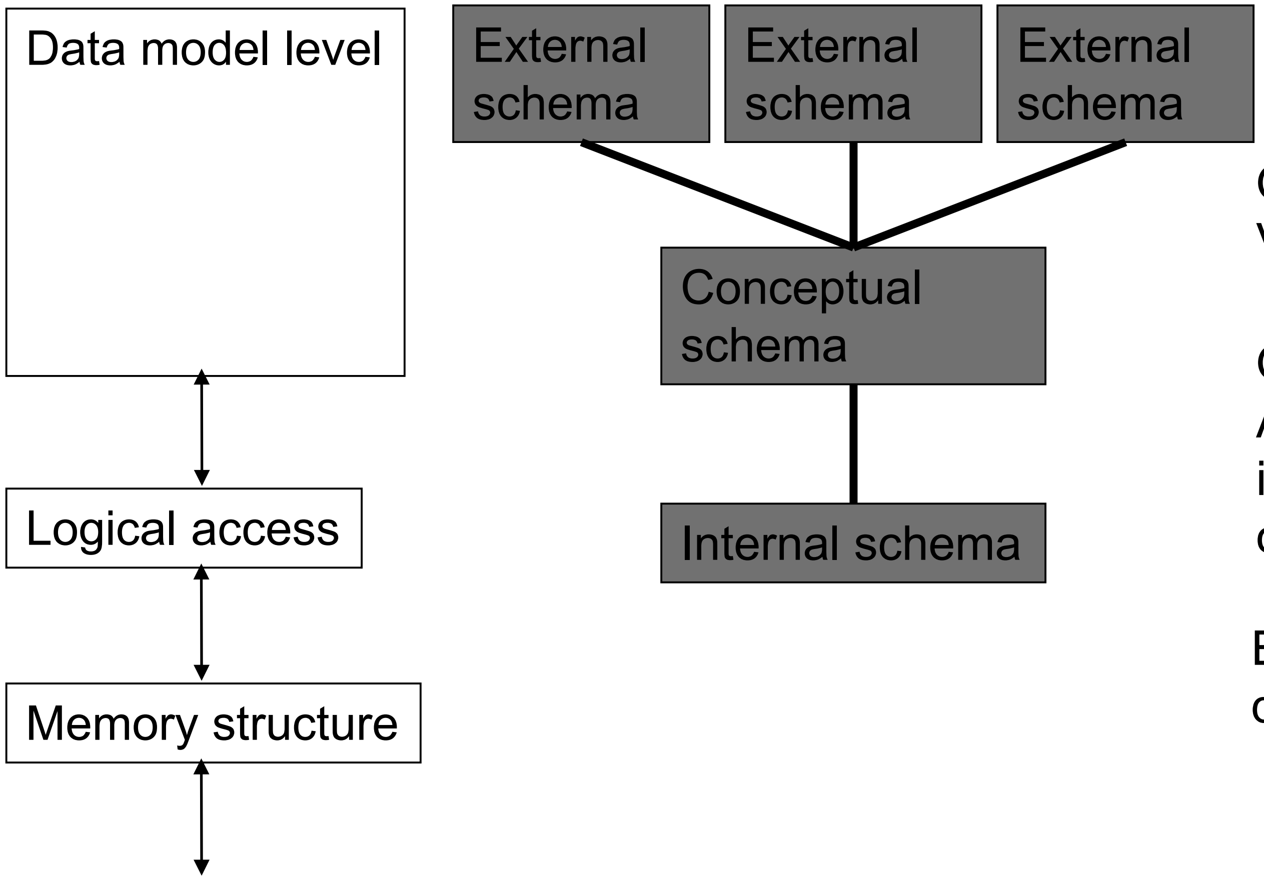

Views

Logical model (tables, attributes, requests)

Physical model (indexing, storage)



# Query Processing




Query transformation;
View expansion

Query translation:
Access paths, choosing index, order of joins, optimization

Execution of the chosen query plan

# Query Processing

Declarative query

```
SELECT Name, Address, Checking, Balance
  FROM customer C, account A
 WHERE Name = "Bond" and C.Account# = A.Account#
```

Generate a Query Execution Plan

```
FOR EACH c in CUSTOMER DO
  IF c.Name = "Bond" THEN
    FOR EACH  a IN ACCOUNT DO
      IF a.Account# = c.Account# THEN
        Output ("Bond", c.Address, a.Checking, a.Balance)
```

Query Execution Plan (QEP)

- Procedural Specification
- Semantically equivalent to query

# Query Translation and Optimization

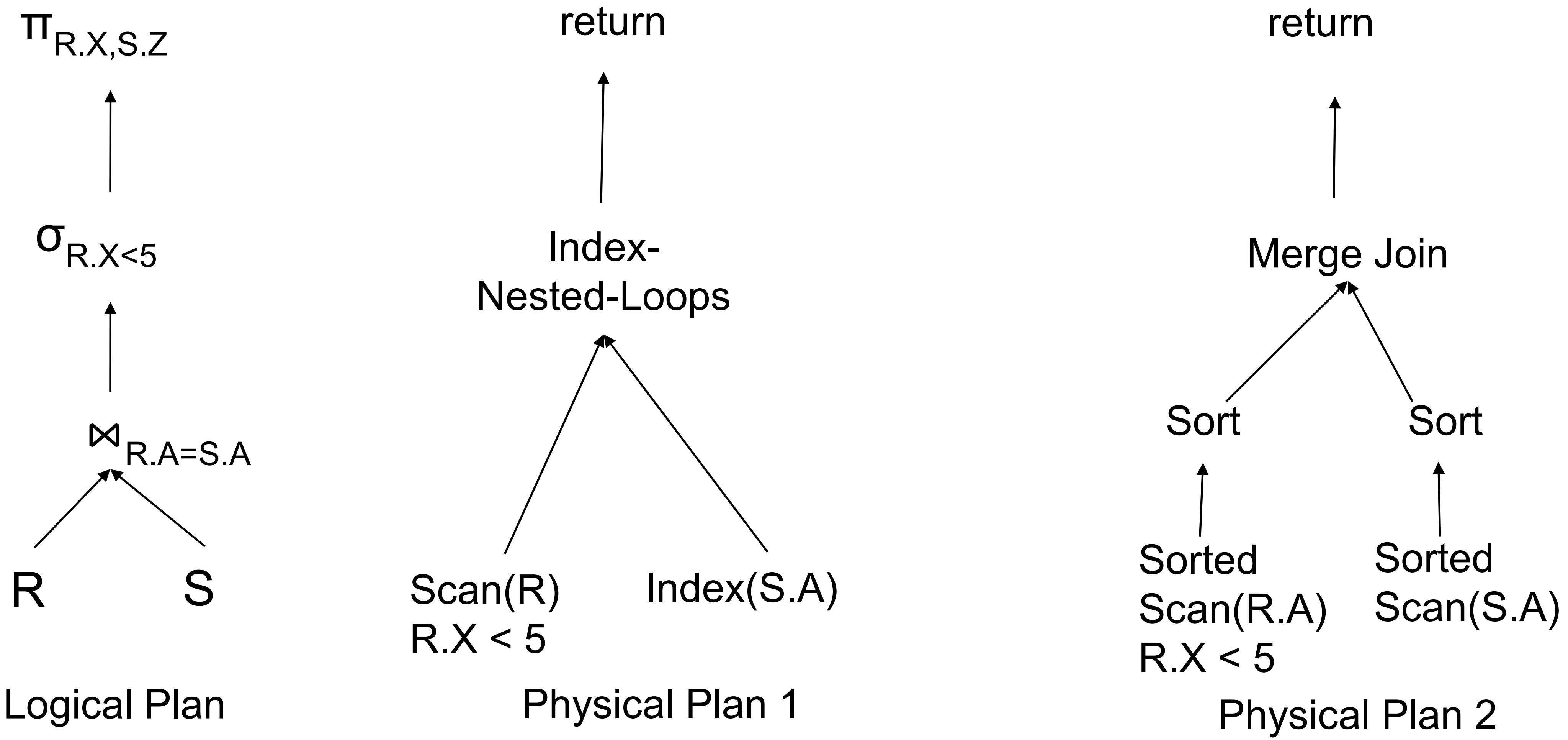
- Parse the query (check syntax)
  - Check if the semantics of schema elements match

- Generic rewriting
  - View Expansion, Common Subexpressions, …

- Pick optimal execution plan
  - Rule-Based Optimizer: Iteratively apply rules
  - Cost-Based Optimizer:
    - Generate candidate plans (exponentially in number)
    - Compare plans by applying cost functions
    - Requires statistics on the data

- Execute the query plan
  - Possibly involves dynamic runtime refinement

# Logical and Physical Plans



# Logical – Physical Mapping

- Relation -> Scan
- σ -> Filter, or index-access
- π (with duplicates) -> Trivial
- x -> Nested-loops-join
- ⋈ -> Hash-, sort-merge-, index-nested-loops-join
- γ -> Hash-aggregation, sorted-aggregation
- π (eliminating duplicates) -> Special case of aggregation
- ∩ -> Special case of a join
- - (difference) -> Inverse case of a join (anti join)
- U -> Union

# Degrees of Freedom

Choices to be made

- Algebraic transformations
- Order of joins
- Join method/algorithm
  - Nested Loop, Sort-Merge, Hash, ...
- Access path: Index (which?) vs. Full-Table-Scan
- Order of operators
  - push down predicates/aggregation
- Correlate / un-correlate subqueries

# Cost Based Optimization

- Enumerate Plans and estimate their execution costs
- Use statistics to estimate costs
  - Table Cardinalities: Size of base table;
  - Column Cardinalities & Frequent Values: Selectivity of equality predicates
  - High/low keys & Histograms: Selectivity of range predicates
  - Indexes depth/density/cluster-ratio: Cost of index seeks
- Statistics are always flawed
- Using sampling is expensive

# Rule Based Optimization

- Employ Heuristics
  - Minimize intermediate results
  - Minimize materialization
  - Minimize access to secondary storage

- Example
  - Push selections as far as possible
  - Push projections as far as possible
  - Does not use information about current state of relations and indexes
  - Does not help much for join order

# Join Methods

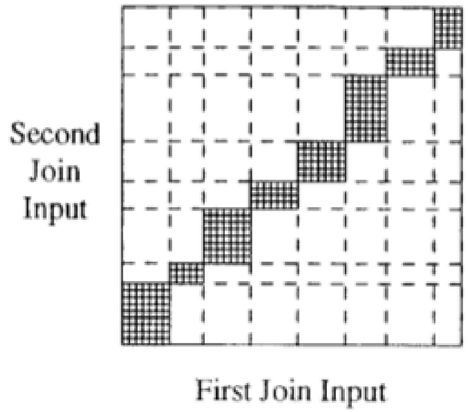
- Nested loop join has complexity O(m*n)
  - m,n: sizes of joined relations

- Other methods
  - Sort-merge join
    - First sort relations in O(n*log(n)+m*log(m))
    - Merge results in O(m+n)
  - Might be better, but ...
    - external sorting is expensive
    - does not pay off if relations already in cache
  - Hash join, …

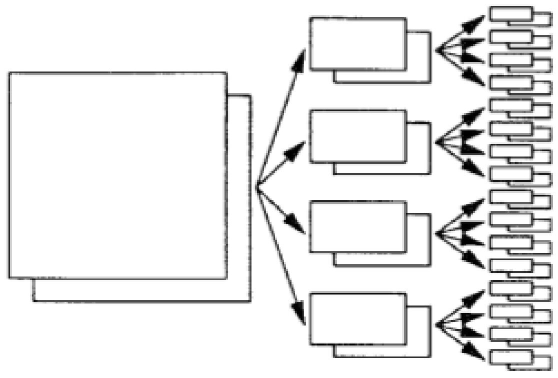
- Note: Usual complexities measure number of comparisons
  - This is "main-memory" viewpoint
  - Should not be used for I/O tasks
  - For data intensive operations, we need to look at number of I/Os (or communications) as bottleneck

# Grace Hash Join

```
partition R into n buckets so that each bucket fits in memory;
partition S into n buckets;
for each bucket j do
   for each record r in Rj do
      insert into a hash table;
   for each record s in Sj do
      probe the hash table.
```




- Works good when memory is small
- Otherwise: Hybrid-Hash-Join



# Data Dictionary

- Statistics are useful but
  - Need to be stored and accessed
  - Need to be kept current
  - Difficult problem!

- Query transformation and optimization needs data dictionary
  - Semantic parsing of query: Which relations exist?
  - Which indexes exists?
  - Cardinality estimates of relations?
  - Size of buffer for in-memory sorting?
  - ...

| **Table_name** | **Att_name** | **Att_type** | **size** | **Avg_size** |
|---|---|---|---|---|
| Customer | Name | Varchar2 | 100 | 24 |
| Customer | account# | Int | 8 | 8 |
| Customer | ... | | | |

# Access Control

- Read and write access on objects
- Read and write access on system operations
  - Create user, kill session, export database, …
- GRANT, REVOKE Operations

- Example:
  - `GRANT ALL PRIVILIGES ON ACCOUNT TO Lawrence WITH GRANT OPTION`
  - "User Lawrence has Read/Write access to the ACCOUNT relation
  - It is possible for Lawrence to grant this rights to others"

- No complete protection
  - Granularity of access rights usually relation/attribute – not tuple
  - Access to data without DBMS
  - Ask several questions to derive requested data
  - In addition: file protection, encryption of data

# Transactions

- Transaction: “Logical unit of work”

```
Begin_Transaction
  UPDATE ACCOUNT
  SET Savings = Savings + 1M
  SET Checking = Checking - 1M
  WHERE Account# = 007;
  INSERT JOURNAL <007, NNN, “Transfer”, ...>
End_Transaction
```

# Synchronization and Locking

- When are two schedules „conflict-free“?
  - when they are serializable
  - when they are equivalent to a serial schedule
  - Prove serializability of schedules

- Checking after execution is wasteful
  - Synchronization protocols
  - Guarantee only serializable schedules
  - Require certain well-behavior of transactions
  - Methods
    - Two phase locking
    - Multi-version synchronization
    - Timestamp synchronization

# Transaction Manager

Synchronization is the “I” in ACID

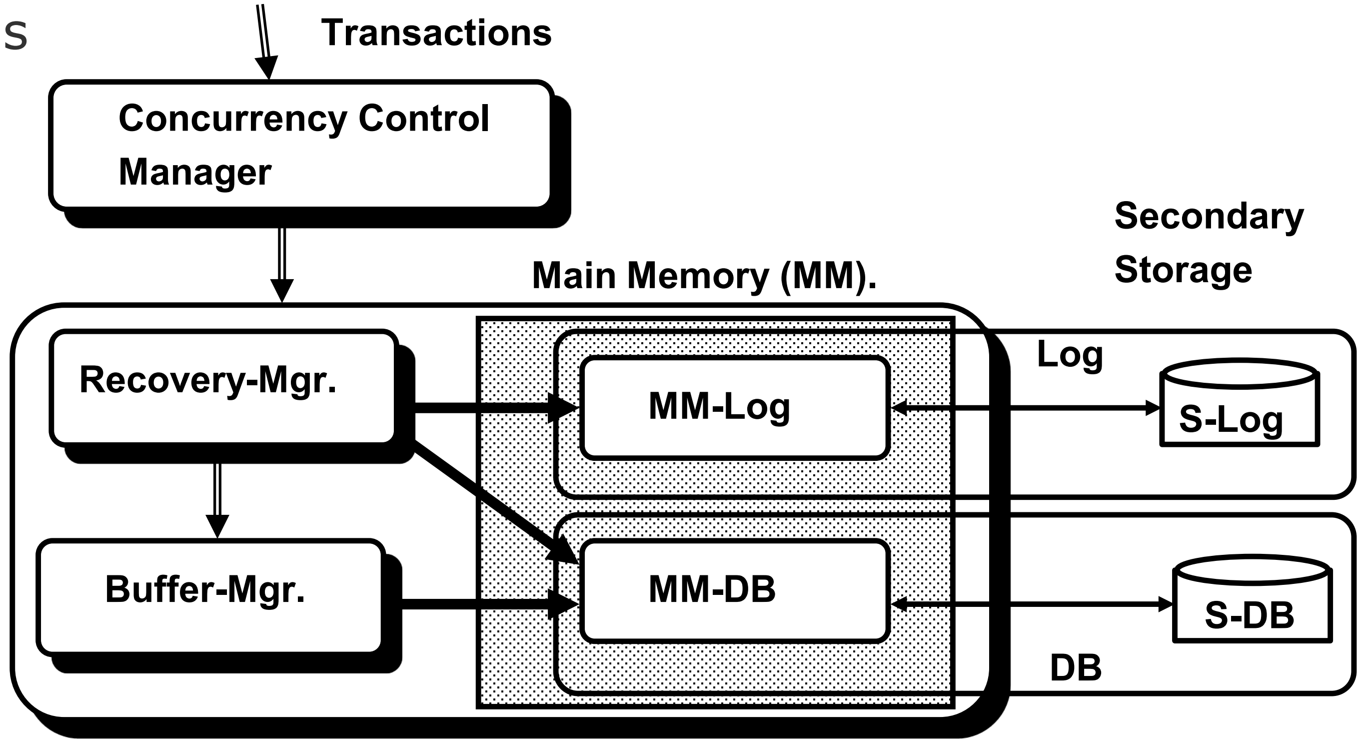
Transaction manager is responsible for

- Concurrency control
  - Concurrent access to data objects
  - Synchronization & locking
  - Deadlock detection and deadlock resolution

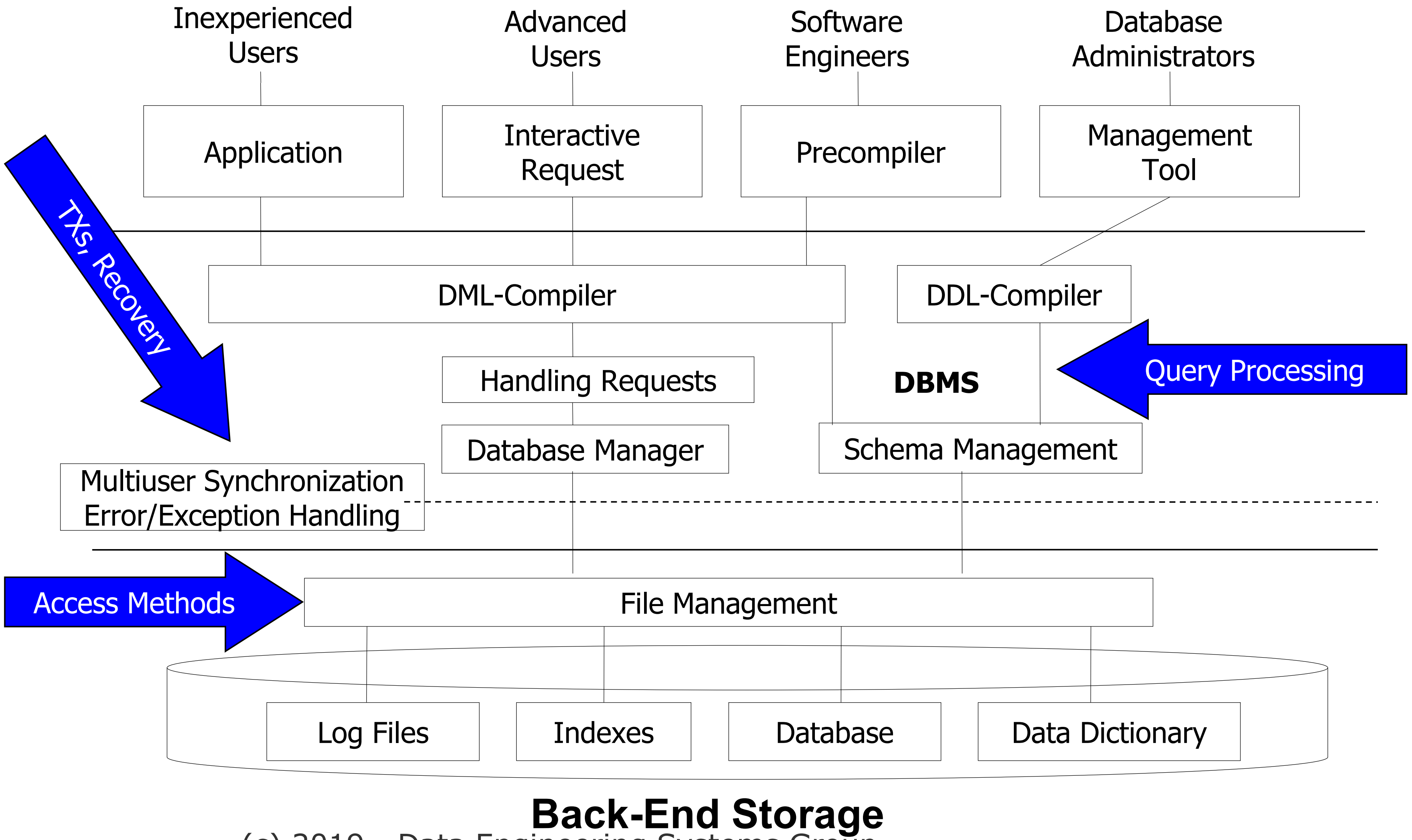
- Logging & recovery
  - Compensate for system und transaction errors
  - Based on log files (redundant storage of information)
  - Error recovery protocols – undo; redo

# Recovery – Broad Principle

- Store data redundantly
  - Save old values
- Uses different file format, adapted to different access characteristics
  - Sequential write, rare reads



# DBMS Overview



# Thank you for your attention!

- Next week: no lectures!
- Next lecture:
  - Big Data Stack and Overview
- Questions?