Universität
Konstanz

Database System Architecture and Implementation

Module 9

Physical Database Design

January 29, 2019

Orientation

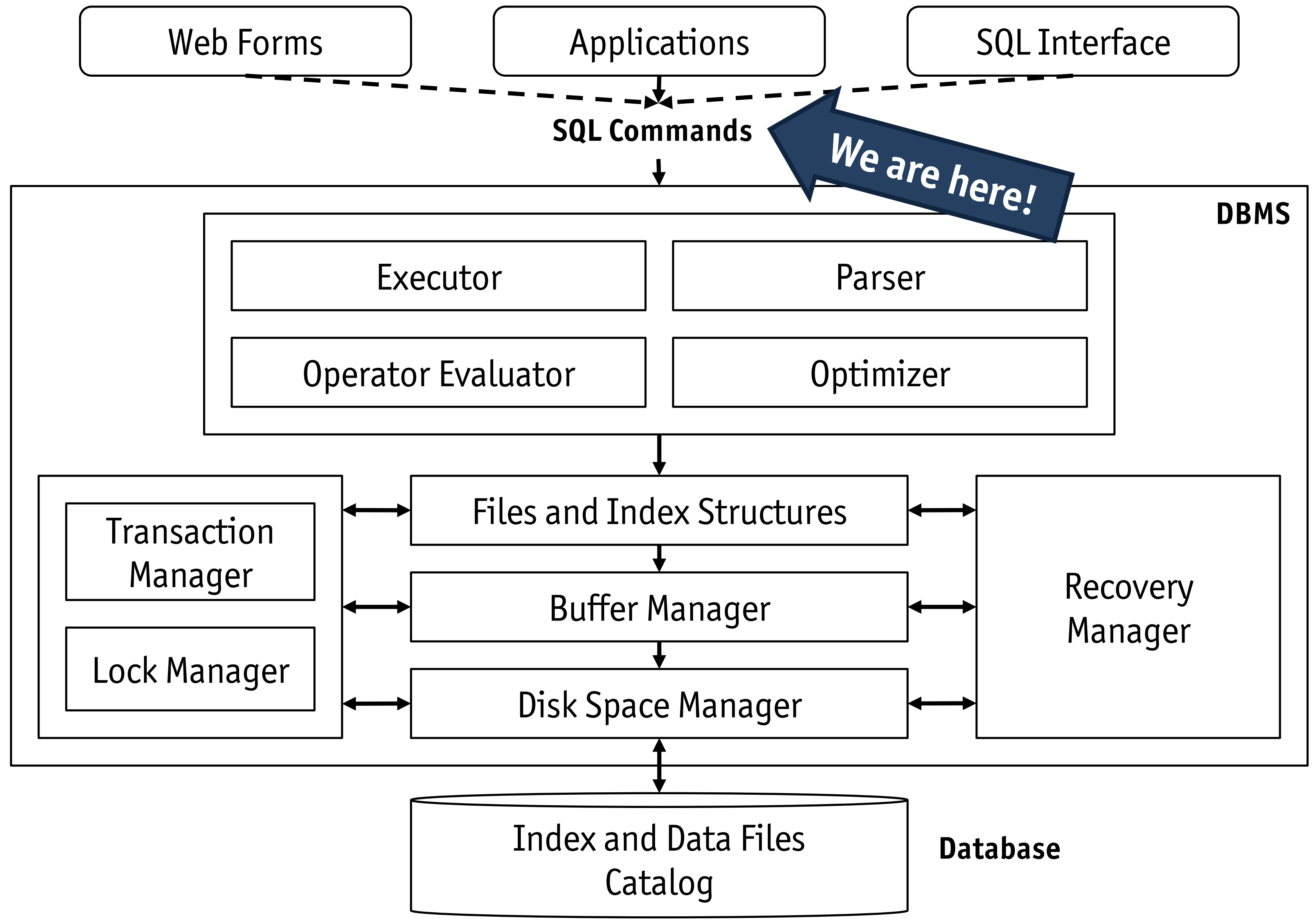


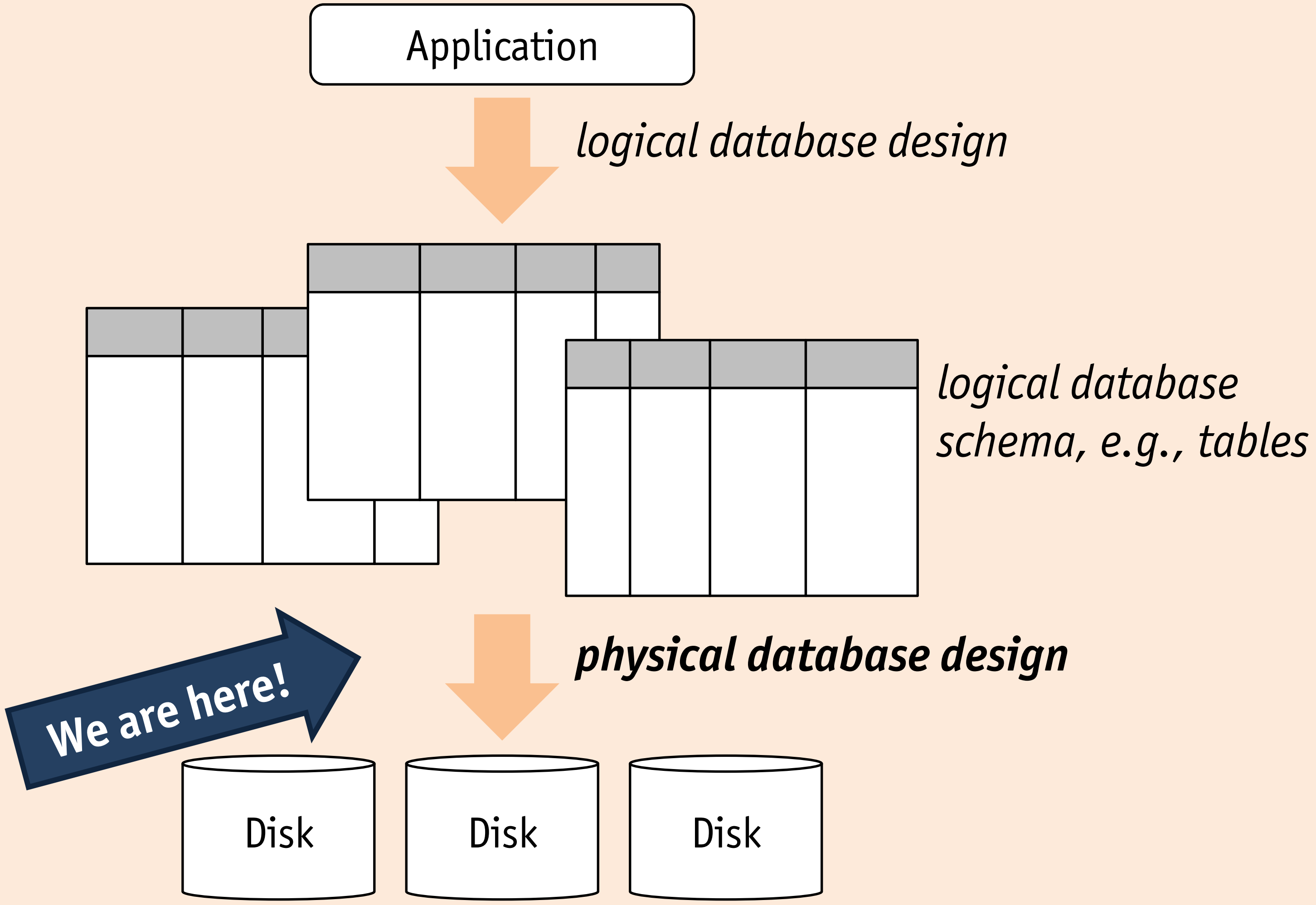
Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Module Overview

- Physical database design
  - index selection
  - clustering and indexing
- Database tuning
  - index tuning
  - conceptual schema tuning
  - query and view tuning
- Benchmarking

Outline

The physical database design problem



Physical Database Design

Problem statement

Given

- a **logical database schema**, e.g., in the form of relational table definitions
- a description of a **database work load**

Find

- an **internal data representation** that maximizes overall performance, e.g., maximal throughput, minimal response time, etc.

- Different internal representations have different, competing effects on performance of individual operations
- Optimization problem
  - data structure is chosen to **speed up** one particular type of operation
  - same data structure typically **slows down** other operations

Physical Database Design

Exercise

Can you think of a **data structure** that speeds up one type of operation, but slows down another?

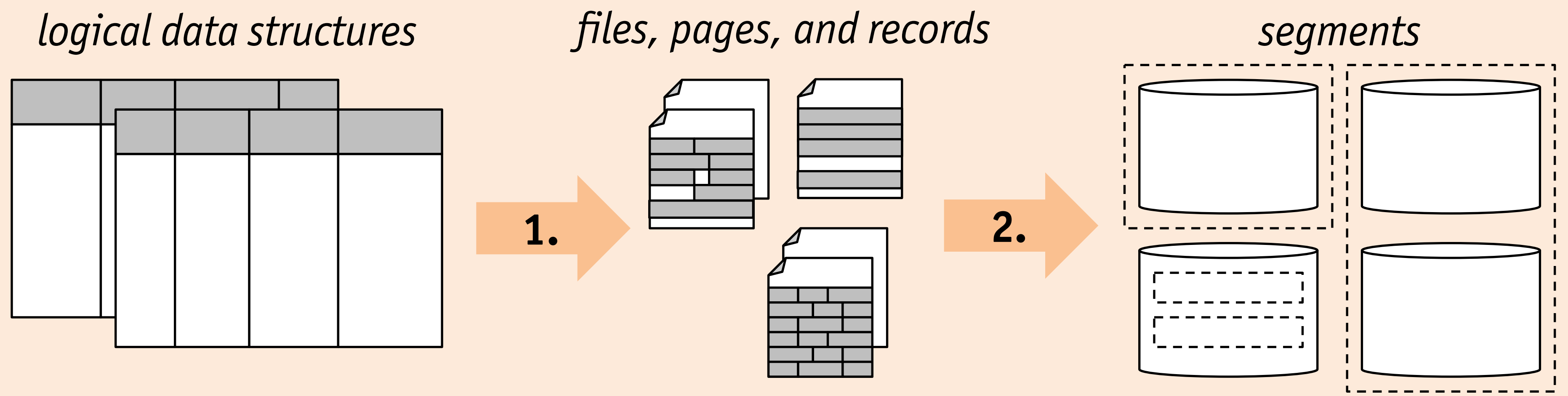
Physical Database Design

- Recall abstractions that take care of low-level data management
  - disk and file management: files of disk blocks (pages)
  - main memory management: (buffer) pages of records
- Higher-level abstraction: a continuous sequences of pages is called a **segment**
  - can think of a segment as an abstraction of a single file
  - typically, a DBMS uses more than one level of mapping segments to operating system files/disks
  - DBMS have variety of names for segments: table space, partition, storage pool, etc.
  - internally, a database comprises a collection of segments to hold data
- Physical database design denotes problem of mapping logical database schema (e.g., tables and their tuples) to segments

Physical Database Design

- Break down physical database design problem into two steps
  1. map logical data structures onto **internal data types** (and indexes)
  2. allocate instances of internal data types to (**pages of**) **segments**
- Focus of this lecture
  - techniques for first step will be discussed in detail
  - overview of second step is given on next slide

The physical database design problem



Internal Data Types to Segments

- When allocating instances of internal data types to segments, **locality of access operations** needs to be considered
  - records within a page can be accessed within same disk I/O operation
  - neighboring pages can be accessed faster than distant pages
  - different segments are mapped to disks individually and independently
- Main decision is whether to map different internal data type into the **same segment** or **not**
  - in this lecture, we assume **1:1-mapping** in most cases, i.e., separate segments per internal data type
  - in this case, segment free space manager is only DBMS component that makes allocation decisions
  - therefore, any other influence has to be exercised through the mapping of logical to internal data types (Step 1 on previous slide)

Logical Structures to Internal Data Types

Standard solution

1 logical table/relation	⟷	1 internal record type (file)
1 tuple	⟷	1 record
1 attribute	⟷	1 field

+

indexes to speed up search

- Depending on database workload other options may be useful
  - decomposing tuples of a relation (horizontally/vertically)
  - grouping related tuples from different relations
  - replicating (parts of) tuples
  - materializing results of important/frequent queries
  - combining any of the above options

Database Workload

- A database workload description includes following information
  - list of **queries** (with their frequencies, as a ratio of all queries/updates)
  - list of **updates** and their frequencies
  - **performance goals** for each type of query and update
- For each query, the description identifies
  - which relations are accessed
  - which attributes are retained (in **`SELECT`** clause)
  - which attributes have selection or join conditions on them (in **`WHERE`** clause) and how selective these conditions are likely to be
- For each update, the description identifies
  - which attributes have selection or join conditions on them (in **`WHERE`** clause) and how selective these conditions are likely to be
  - type of update (**`INSERT`**, **`DELETE`**, or **`UPDATE`**) and updated relation
  - fields modified by the update (for **`UPDATE`** commands)

Physical Design and Tuning Decision

- Choice of indexes to create
  - which **relations to index** and which (combination of) fields to chose as index search keys
  - for each index, whether it should be **clustered** or **unclustered**
- Tuning conceptual schema
  - **alternative normalized schemas:** choose one of multiple possible ways to decompose a schema into a desired normal form (BCNF, 3NF)
  - **denormalization:** partially “undo” normalization to improve performance of queries that involve attributes from several decomposed relations
  - **vertical or horizontal partitioning:** split relations to improve performance of queries that only involve a subset of attributes or tuples
  - **views:** add views to mask changes in conceptual schema from user
- Query and transaction tuning by **rewriting** frequently executed queries to run faster

Guidelines for Index Selection

Guideline #1: Whether to Index

- do not build an index unless some query (including query components of updates) **benefits** from it
- if possible, choose indexes that speed up **more than one** query

Guideline #2: Choice of Search Key

Attributes mentioned in **`WHERE`** clause are candidates for indexing

- **exact match selection condition:** consider a (hash) index on selected attributes
- **range selection condition:** consider a B+ tree (or ISAM) index on selected attributes

Exercise: B+ tree vs. ISAM index

When is an ISAM index worth considering?

Guidelines for Index Selection

Guideline #3: Multi-Attribute Search Keys

Indexes with multi-attribute search keys should be considered

- if a **`WHERE`** clause includes conditions on more than one attribute of a relation
- if they enable **index-only evaluation strategies** for important queries

Note that order of attributes in search key is particularly important of range queries are expected.

Guideline #4: Whether to Cluster

Choice of clustered index is important as only one index per relation can be clustered and clustering affects performance greatly

- **rule of thumb:** range queries are likely to benefit most from clustered index
- if **several range queries** involve different sets of attributes, choice of clustered index should be guided by selectivity of conditions and relative frequency of queries
- for indexes that are used to enable an **index-only evaluation strategy**, clustering is not required

Guidelines for Index Selection

Guideline #5: Hash vs. Tree Index

Tree indexes support more types of selection conditions and are usually preferable, but hash indexes are better in the following situations

- index is intended to support **nested loops join** (indexed relation is inner relation, search key includes join columns): slight improvement of hash index over tree index is magnified by iterations of outer loop
- there is a **very important equality query**, but no range queries, involving search key attributes

Guideline #6: Balancing Cost of Index Maintenance

After drawing up a "wishlist" of indexes to create, consider impact of each index on updates in workload

- if maintaining a potential index slows down **frequent update operations**, consider dropping it
- however, adding an index may well **speed up** a given update operation (query component of updates)

Guidelines for Index Selection

Exercise: Which indexes should be created?

```
SELECT E.ename, D.mgr
  FROM Employees E, Departments D
 WHERE D.dname = “Toy” AND E.dno = D.dno
```

Index selection guidelines suggest that **`D.dname`**, **`D.dno`**, and **`E.dno`** are candidate attributes to be indexed. What indexes and what types of indexes would you create, under which assumptions?

Index attribute **`D.dname`**?

Index attribute **`D.dno`**?

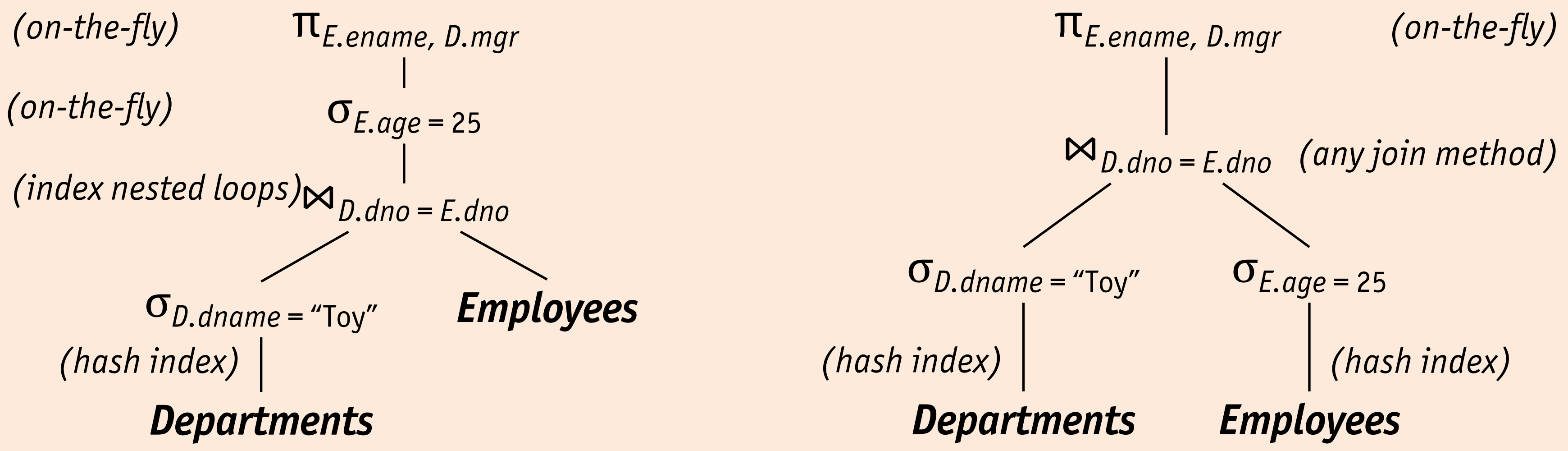
Index attribute **`E.dno`**?

Guidelines for Index Selection

Example: alternative evaluation plans

Let us consider alternative evaluation plans for this slight variant of the previous query.

```
SELECT E.ename, D.mgr
  FROM Employees E, Departments D
 WHERE D.dname = "Toy" AND E.dno = D.dno AND E.age = 25
```



↳ If there is already an index on **`E.dno`**, adding another index on **`E.age`** is **not justified** by this additional evaluation plan.

Guidelines for Index Selection

Exercise: Which indexes should be created?

```
SELECT E.ename, D.mgr
  FROM Employees E, Departments D
 WHERE E.sal BETWEEN 10000 AND 20000
       AND E.hobby = "Stamps" AND E.dno = D.dno
```

Index selection guidelines suggest that `D.dno`, `E.hobby`, `E.sal`, and `E.dno` are candidate attributes to be indexed. What indexes and what types of indexes would you create, under which assumptions?

Index attribute `E.hobby` or `E.sal`?

Index attribute `E.dno`?

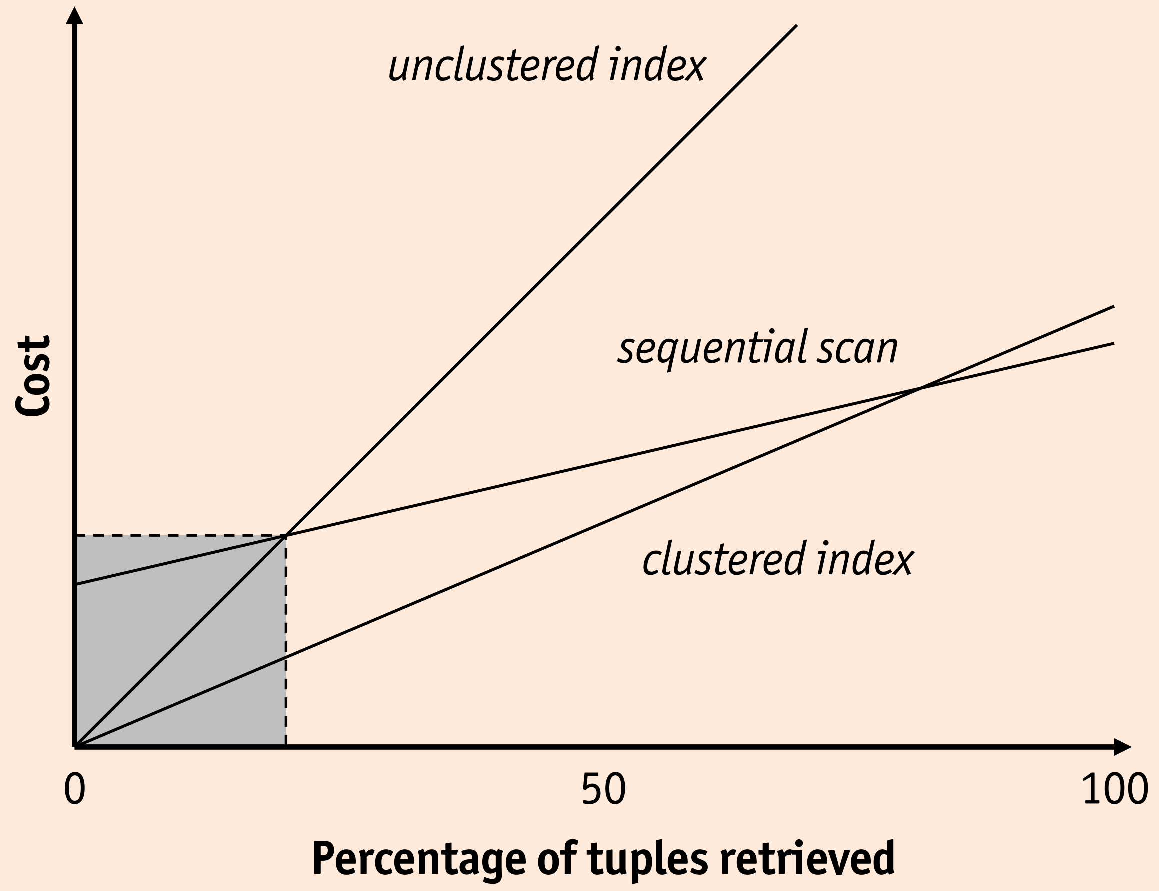
Index attribute `D.dno`?

Clustering and Indexing

- Apart from which indexes to create, whether these indexes are **clustered** or **unclustered** is an important decision
  - clustered indexes: use for selection predicates with **low selectivity**, e.g., range selections or selections attributes that are not candidate keys
  - unclustered indexes: use for selection predicates with **high selectivity**, e.g., equality selections on attributes that are candidate keys
- Impact on clustering depends on number of retrieved tuples
  - to retrieve **one tuple**, unclustered index is just as good as clustered index
  - to retrieve **a few tuples**, unclustered index is better than a sequential scan of entire relation
  - to retrieve **all tuples**, sequential scan of entire relation is better than clustered index
- Use of blocked I/O improves relative advantage of sequential scan over unclustered index

Clustering and Indexing

Impact of clustering



Note that **grey area** denotes range in which unclustered index is better than sequential scan of entire relation.

Clustering and Indexing

Exercise: Clustered or unclustered index?

```
SELECT E.ename, D.mgr
  FROM Employees E, Departments D
 WHERE D.dname = "Toy" AND E.dno = D.dno
```

For this example query, we have already decided to create an index on **`D.dname`** and **`E.dno`**. Should these indexes be clustered or unclustered?

Clustered or unclustered index on **`D.dname`**?

Clustered or unclustered index on **`E.dno`**?

Clustering and Indexing

Example: Clustered or unclustered index?

```
SELECT E.ename, D.mgr
  FROM Employees E, Departments D
 WHERE E.hobby = "Stamps" AND E.dno = D.dno
```

In contrast to the previous example (`D.dname = "Toy"`), the selection condition on `E.hobby` is not likely to be very selective as many employees might collect stamps.

- a plan using a **block nested loops** or a **sort-merge join**, rather than a plan using an index nested loops join is likely to be chosen by query optimizer
- a sort-merge join would benefit from a **clustered tree index** on `D.dno`
- if there is no index on `E.dno`, query optimizer might still be able to use another index on `Employees`, say a clustered index on `E.hobby`, to retrieve tuples

Checking Query Execution Plans

The Real World

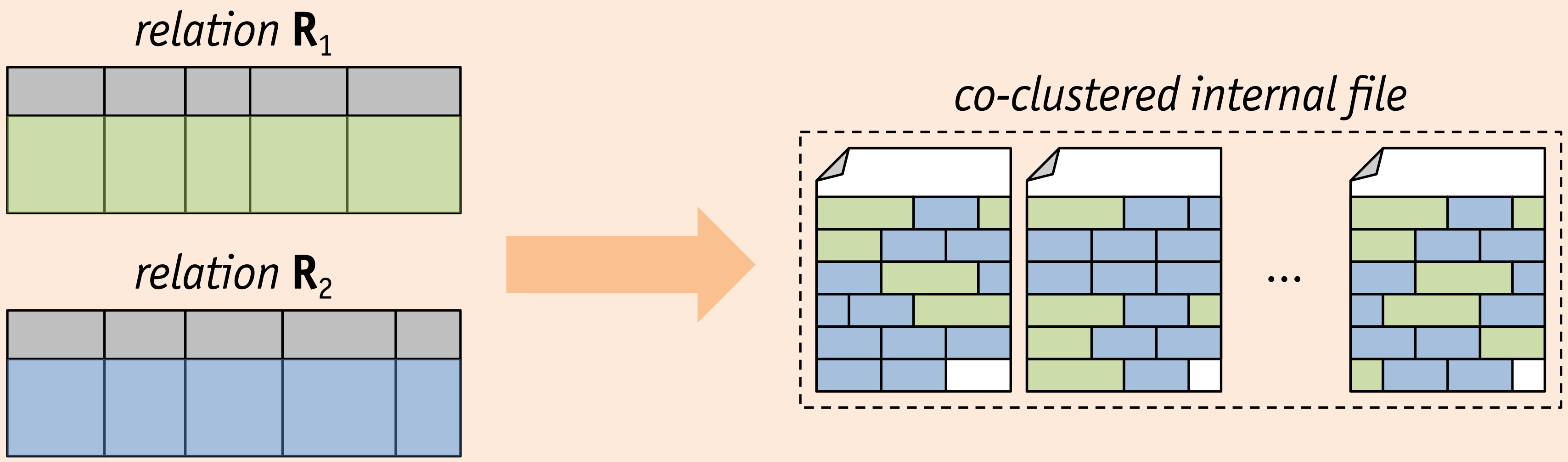
All major RDBMS provide functionality to display the execution plan chosen by the query optimizer for a particular query.

- **IBM DB2** and **PostgreSQL**
  - offer an **`EXPLAIN`** command that can be used for any explainable statement
- **Oracle 10*g***
  - provides an **`EXPLAIN PLAN`** statement, which creates a relation **`PLAN_TABLE`** with a row for each step of the query execution plan
- **Microsoft SQL Server**
  - SQL Server Management studio can display graphical execution plans (CTRL+M)
  - command **`SET SHOWPLAN_TEXT ON`** will display textual execution plans
  - command **`SET SHOWPLAN_XML ON`** will display execution plans in XML

Co-Clustering Two Relations

- Some DBMS can store records from **more than one** relation to be stored in a **single** file
  - physical interleaving can be configured by database administrator
  - this data layout is sometimes referred to as **co-clustering** two relations

Co-clustering two relations



relation $\mathbf{R}_1$
relation $\mathbf{R}_2$
co-clustered internal file
...

Co-Clustering Two Relations

Example: Co-clustering two relations

Parts (*<u>pid</u>*<u>: `integer`</u>, *pname*: `string`, *cost*: `integer` , *supplierid*: `integer`)
Assembly (*partid*: `integer`, <u>*componentid*: `integer`</u> , *quantity*: `integer`)

Relation **Assembly** represents a 1:N relationship between parts and their sub-parts: a part can have many sub-parts, but each part is the sub-part of at most one part.

```
SELECT P.pid, A.componentid
  FROM Parts P, Assembly A
 WHERE P.pid = A.partid AND P.cost = 10
```

Performance of this query is improved by an index on **P**.*cost* and one on **A**.*partid*. Parts with *cost* = 10 can be retrieved using first index. However, if many parts have this cost, the second index has to be accessed for each of these records.

This second index access can be avoided by co-clustering the two relations: each **Parts** record **P** is followed on disk by all **Assembly** records **A**, such that **P**.*pid* = **A**.*partid*.

This optimization is especially important if the parts hierarchy is traversed **recursively**!

Co-Clustering Two Relations

- **Pros**
  - can speed up joins, in particular key-foreign key joins corresponding to 1:N relationships
- **Cons**
  - sequential scan of either relation becomes slower, since in both cases (possibly multiple) retrieved records have to be skipped, wasting I/O
  - all inserts, deletes, and updates that alter record lengths become slower, due to the overhead involved in maintaining the clustering

Index-Only Plans

- Finding indexes that enable **index-only plans** is an important goal of index selection
  - avoid retrieving tuples from one of the referenced relations
  - scan an associated index instead, which is likely to be much smaller
- An index that is used (only) for index-only plans does **not** have to be clustered
  - no tuples from indexed relation need to be retrieved
  - index contains all data required to evaluate query
- Recall that index search key can consist of many attribute values

Index-Only Plans

Example: Index-only plan

```
SELECT D.mgr, E.eid
  FROM Departments D, Employees E
 WHERE D.dno = E.dno
```

Considerations

- performance of this query is improved by an index on **`E.dno`**, as it enables an index nested loops join with **`Departments`** as outer relation
- since **`E.dno`** is not a candidate key, this index needs to be clustered in order for this plan to be efficient
- however, if there is already a clustered index on **`Employees`**, another one cannot be created
- by creating an unclustered index with search key ⟨**`dno`**, **`eid`**⟩, an index-only plan can be enabled
- this plan uses an index nested loops join with **`Departments`** as outer relation and an index-only scan of the inner relation

Database Tuning

- Actual use of database provides detailed information that can be used to **refine** its initial design
  - assumptions about workload can be replaced by observed usage patterns
  - guesses about size of data can be replaced with actual statistics
- Continued database tuning is important to get best performance
  - index tuning
  - conceptual schema tuning
  - query and view tuning

Tuning Indexes

- Reasons to refine choice of indexes
  - queries and updates expected to be important are not very frequent
  - new queries and updates may be identified to be important
  - optimizer does not find the plans that it was expected to
- Index tuning actions
  - drop indexes created for queries and updates that are infrequent
  - add new indexes to support frequent queries and updates
  - alter existing indexes (search key, clustered vs. unclustered)
  - periodically rebuild static indexes (e.g., ISAM) or dynamic B+ trees that do not merge pages on delete to improve space occupancy

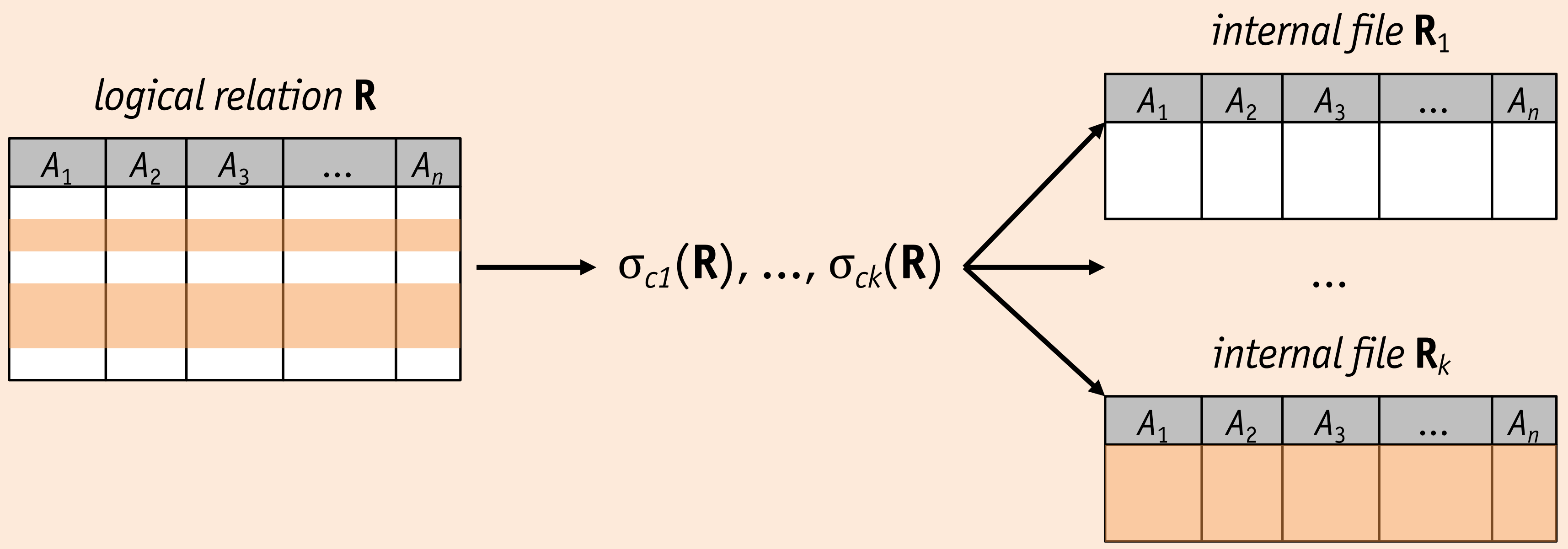
Tuning Conceptual Schema

- Redesign conceptual schema (and re-examine physical database design) in order to meet performance objectives
  - once database has been designed and populated with tuples, changing the conceptual schema requires significant effort
  - changes to schema of an operational database sometimes referred to as **schema evolution**
- Choice of conceptual schema should be guided by consideration of queries and updates in workload
  - **decomposition** of logical relations
  - **denormalization** of logical relations

Horizontal Decomposition

- Map tuples of logical relation into several (disjoint or overlapping) subsets that are stored in separate internal files
  - all resulting internal files share the **same structure**
  - internal files can be viewed as materialized results of a set of **selection queries** on the logical relation

Horizontal decomposition



logical relation **R**
A_1 A_2 A_3 … A_n
$\sigma_{c1}(\mathbf{R}), \ldots, \sigma_{ck}(\mathbf{R})$
internal file $\mathbf{R}_1$
A_1 A_2 A_3 … A_n
…
internal file $\mathbf{R}_k$
A_1 A_2 A_3 … A_n

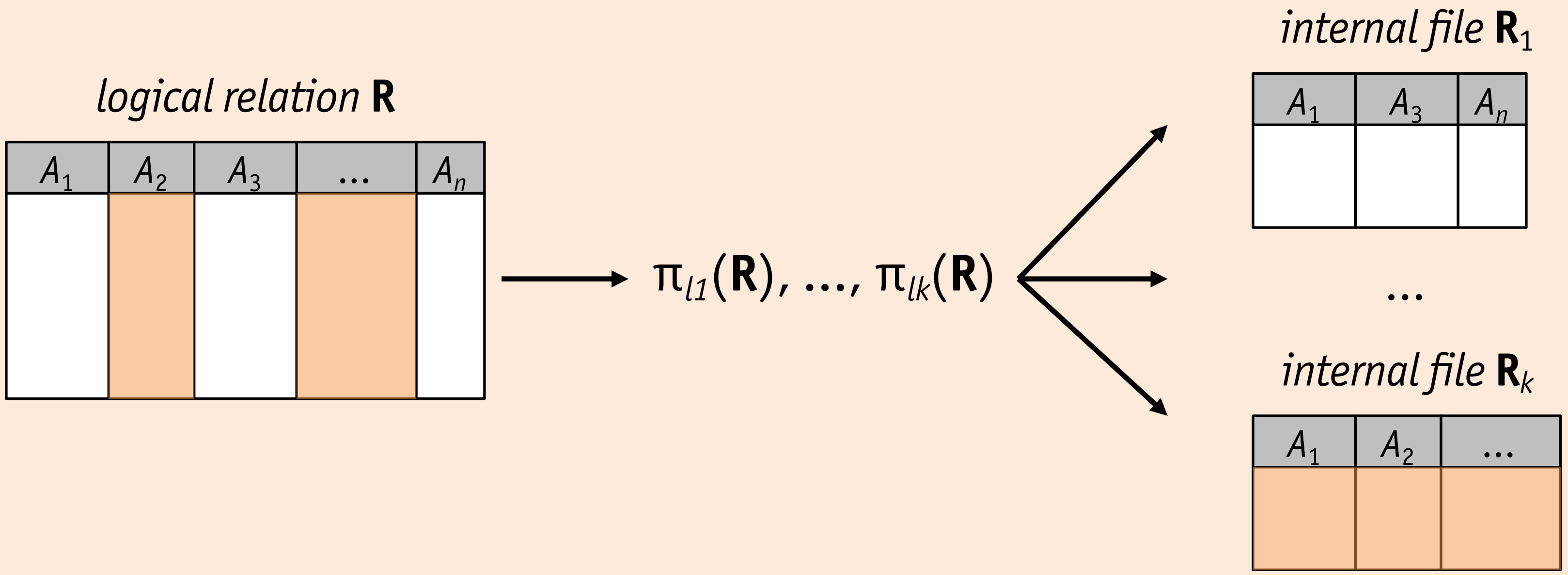
Horizontal Decomposition

- Each internal file may now be treated individually (in a separate segment) in further mapping to the disks
- **Pros**
  - access (read and write) to parts of relation are faster, if the only refer to a subset of the internal files, e.g., suitable selection queries
  - files/segments may be distributed across several servers
  - indexes can be defined independently on some of the internal files
- **Cons**
  - access to whole relation requires more effort (union operation)
  - change of attribute value may result in deletion from one and insertion into another internal file
  - index over all tuples may no longer be possible
- **Caution:** make sure that $c_1 \vee \ldots \vee c_k \equiv$ true!

Vertical Decomposition

- Map different attributes of logical tuples into fields of different internal sub-records stored in separate files
  - all resulting internal files share **different structures**
  - internal files can be viewed as materialized results of a set of **projection queries** on the logical relation

Vertical decomposition



logical relation **R**
A_1
A_2
A_3
…
A_n
$\pi_{l1}(\mathbf{R}), \ldots, \pi_{lk}(\mathbf{R})$
internal file $\mathbf{R}_1$
A_1
A_3
A_n
…
internal file $\mathbf{R}_k$
A_1
A_2
…

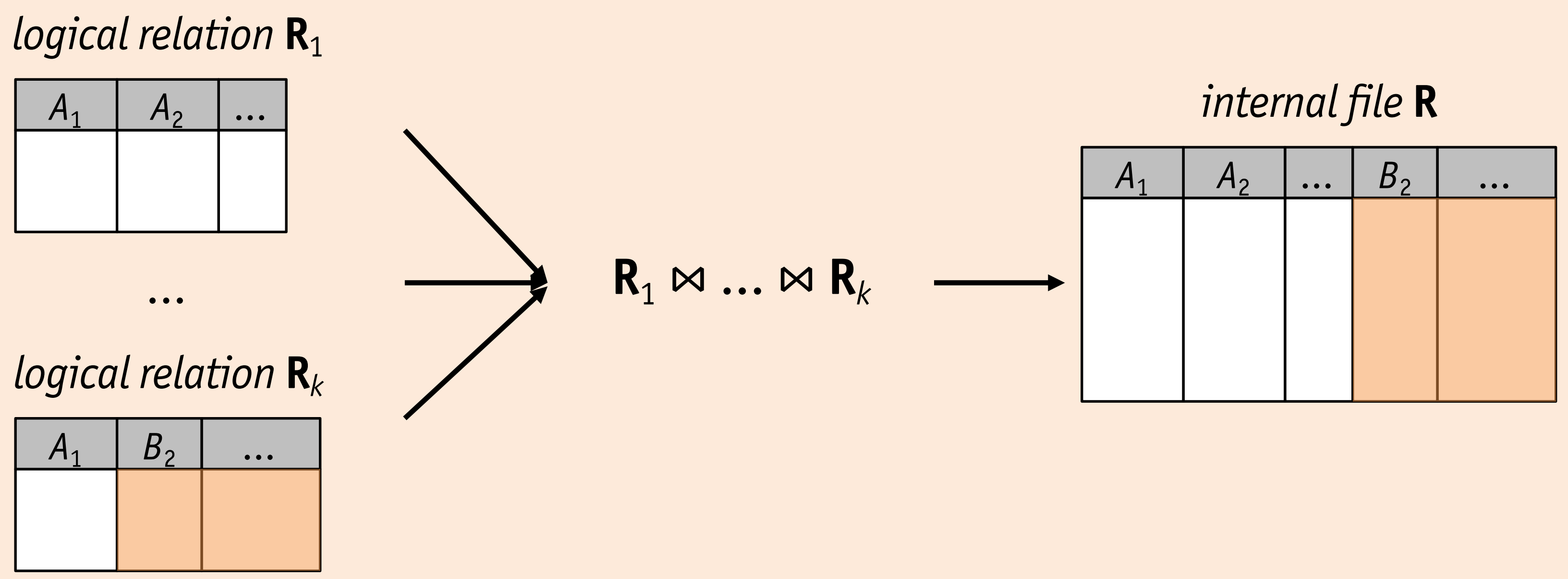
Vertical Decomposition

- Again, each internal file may now be treated individually in the further mapping to the disks
- **Pros**
  - access (read and write) to parts of tuples are faster, if the only refer to a subset of the internal files, e.g., suitable projection queries
  - files/segments may be distributed across several servers
  - indexes can be defined independently on some of the internal files
- **Cons**
  - access to whole tuples requires significant effort (join operation)
  - insert/delete needs to manipulate multiple internal files
  - indexes over multiple attributes are no longer possible, if they are stored in separate files
- **Caution:** make sure that $\pi_{l1}(\mathbf{R}) \bowtie \ldots \bowtie \pi_{lk}(\mathbf{R}) \equiv \mathbf{R}$!

Denormalization

- Map tuples related via a key-foreign key relationship from different relations into single internal records
  - internal files can be viewed as materialized results of a **join query** between two (or more) logical relations

Denormalization



logical relation $\mathbf{R}_1$
A_1 A_2 …
…
logical relation $\mathbf{R}_k$
A_1 B_2 …
$\mathbf{R}_1 \bowtie \ldots \bowtie \mathbf{R}_k$
internal file $\mathbf{R}$
A_1 A_2 … B_2 …

Denormalization

- Internal file is treated as a whole in subsequent mapping to the disks and all fields of its larger records will stay closely together
- **Pros**
  - access to related tuples is much faster, since no join is required
  - index on join attributes can be exploited for all participating relations
- **Cons**
  - access to tuples of single relations requires extra effort (projection, duplicate elimination, more page I/O operations due to larger records)
  - insert/delete gets more tricky
  - change in join attribute value requires significant overhead
  - change in other attribute value requires more effort for replicated tuples
  - significant space overhead, depending on join selectivity
- **Caution**: make sure that $\forall_i : \pi_{R_i}(\mathbf{R}) = \mathbf{R}_i$, possibly using outer join

Tuning Queries and Views

- Examine queries and views that run much slower than expected to identify problems
  - verify that DBMS uses expected query evaluation plan
  - tune indexes to improve performance of expected plan
  - rewrite query or view to help optimizer find expected plan
- Possible reasons why optimizer is not finding best plan
  - selection conditions involving **`NULL`** values
  - selection conditions involving arithmetic or string expressions
  - selection conditions using **`OR`** connective
  - inability to recognize a sophisticated plan (e.g., index-only scan) for aggregation queries involving **`GROUP BY`** clause
  - nested queries
  - temporary relations

Tuning Queries and Views

Exercise: How can these queries be rewritten to help the optimizer?

1. There is an index on **`E.age`**, but the optimizer does not use it to evaluate the condition **`D.age = 2 * E.age`**.

2. There are indexes on **`E.hobby`** and **`E.age`**, but the optimizer does not use them to evaluate the following query.

```
SELECT E.dno
  FROM Employees E
 WHERE E.hobby = “Stamps” OR E.age = 25
```

DBMS Benchmarks

- Several benchmarks exist to compare performance of different DBMS products with respect to a certain class of applications
  - **Online Transaction Processing (OLTP)**, e.g., TPC-E and TPC-C
  - **Online Analytical Processing (OLAP)**, e.g., TPC-H and TPC-DS
  - **Object Databases**, e.g., 007 and PolePosition
  - **XML Databases**, e.g., XMark and XBench
  - **RDF Databases**, e.g., LUBM
- Criteria for database performance benchmarks
  - **portable:** can be run on all DBMS products
  - **easy to understand**
  - **scale** naturally to larger problem instances
  - measure **peak performance** (transactions per seconds, i.e., tps)
  - measure **price/performance ratio**, i.e., $/tps

DBMS Benchmarks

- Benchmarks should be used with good understanding of what is measured and in what application environment DBMS is used
- How meaningful is a given benchmark?
  - performance of a complex system **cannot** be distilled into one number
  - good benchmarks have **suite of tasks** to test several relevant features
- How well does a benchmark reflect application workload?
  - **compare** expected application workload with workload of benchmark
  - select benchmark tasks that are **relevant** to intended application
  - consider **how** performance is measured (e.g., single-user vs. multi-user)
- Create or adapt benchmark
  - vendors often **tune** their DBMS to tasks of important benchmarks
  - **modify** standard benchmark tasks slightly
  - **replace** standard tasks with similar tasks from application workload

Database System Architecture and Implementation

THAT’S ALL FOLKS!