Distributed Data Management

Dr.-Ing. Eike Schallehn

OvG Universität Magdeburg
Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

2018/2019

Organization of Lecture and Exercises

- Weekly lecture
  - Teacher: Eike Schallehn (eike@iti.cs.uni-magdeburg.de)
- Weekly exercises with two alternative time slots
  - Starting in November
  - Tutors as teachers
- Written exam at the end of the semester (registration using HISQUIS system)

Prerequisites

- Required: knowledge about database basics from database introduction course
  - Basic principles, Relational Model, SQL, database design, ER Model
- Helpful: advanced knowledge about database internals
  - Query processing, storage structures
- Helpful hands-on experience:
  - SQL queries, DDL and DML

Content Overview

1. Foundations
2. Distributed DBMS:
   architectures, distribution, query processing, transaction management, replication
3. Parallel DBMS:
   architectures, query processing
4. Federated DBS:
   architectures, conflicts, integration, query processing
5. Peer-to-peer Data Management

English Literature /1

- M. Tamer Özsu, P. Valduriez: *Principles of Distributed Database Systems*. Second Edition, Prentice Hall, Upper Saddle River, NJ, 1999.
- S. Ceri and G. Pelagatti: *Distributed Databases Principles and Systems*, McGraw Hill Book Company, 1984.
- C. T. Yu, W. Meng: *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers, San Francisco, CA, 1998.

English Literature /2

- Elmasri, R.; Navathe, S.: *Fundamentals of Database Systems*, Addison Wesley, 2003
- C. Dye: *Oracle Distributed Systems*, O'Reilly, Sebastopol, CA, 1999.
- D. Kossmann: *The State of the Art in Distributed Query Processing*, ACM Computing Surveys, Vol. 32, No. 4, 2000, S. 422-469.

German Literature

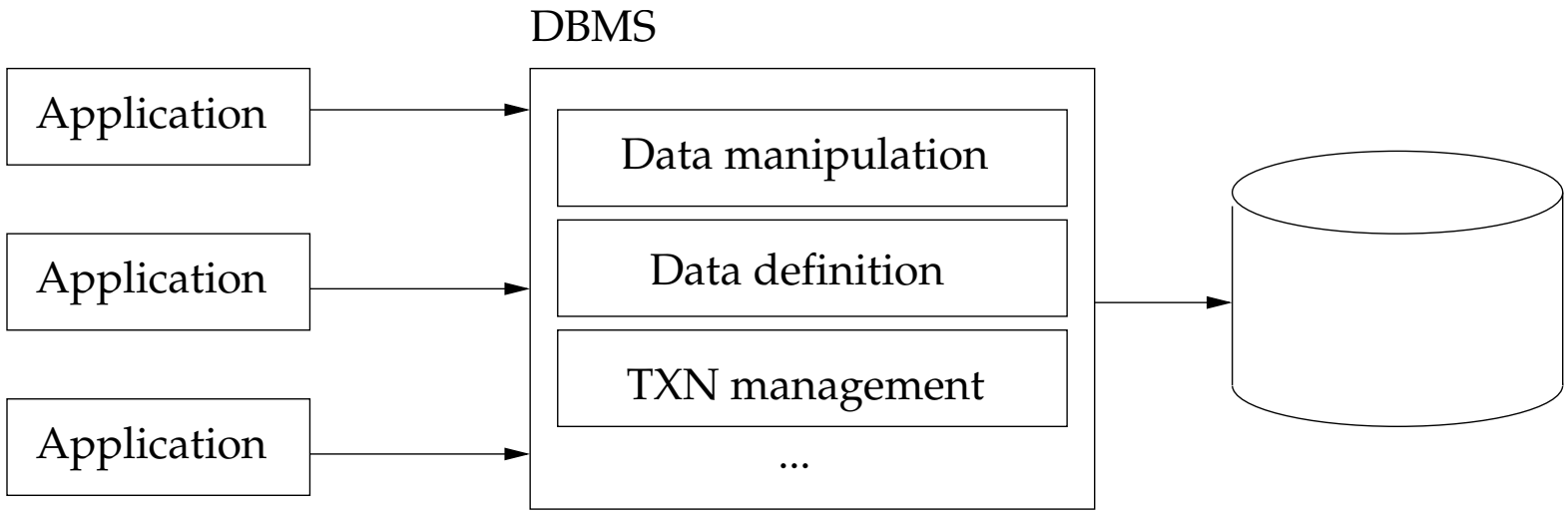
- E. Rahm, G. Saake, K.-U. Sattler: *Verteiltes und Paralleles Datenmanagement*. Springer-Verlag, Heidelberg, 2015.
- P. Dadam: *Verteilte Datenbanken und Client/Server-Systeme*, Springer-Verlag, Berlin, Heidelberg 1996.
- S. Conrad: *Föderierte Datenbanksysteme: Konzepte der Datenintegration*. Springer-Verlag, Berlin/Heidelberg, 1997.

Part I

Introduction

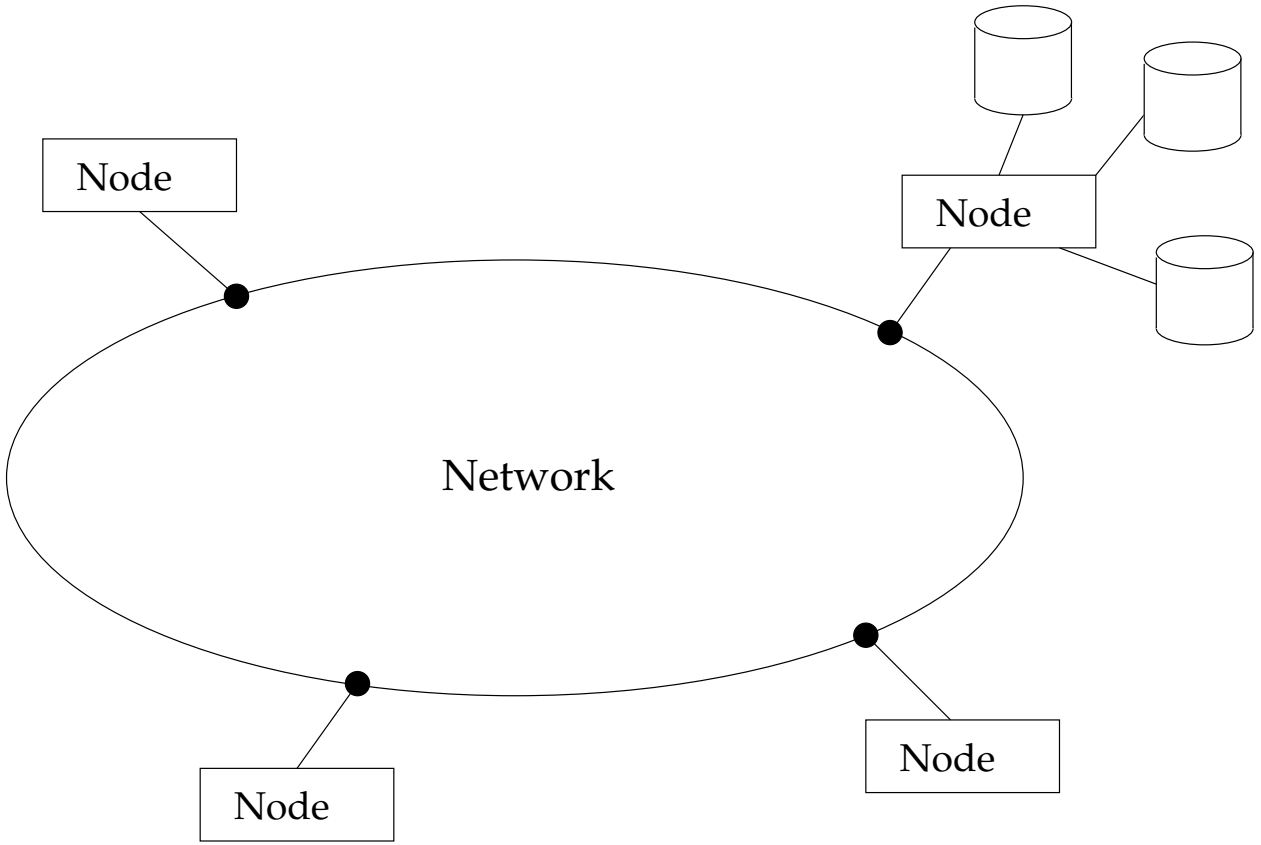
Overview

Centralized Data Management

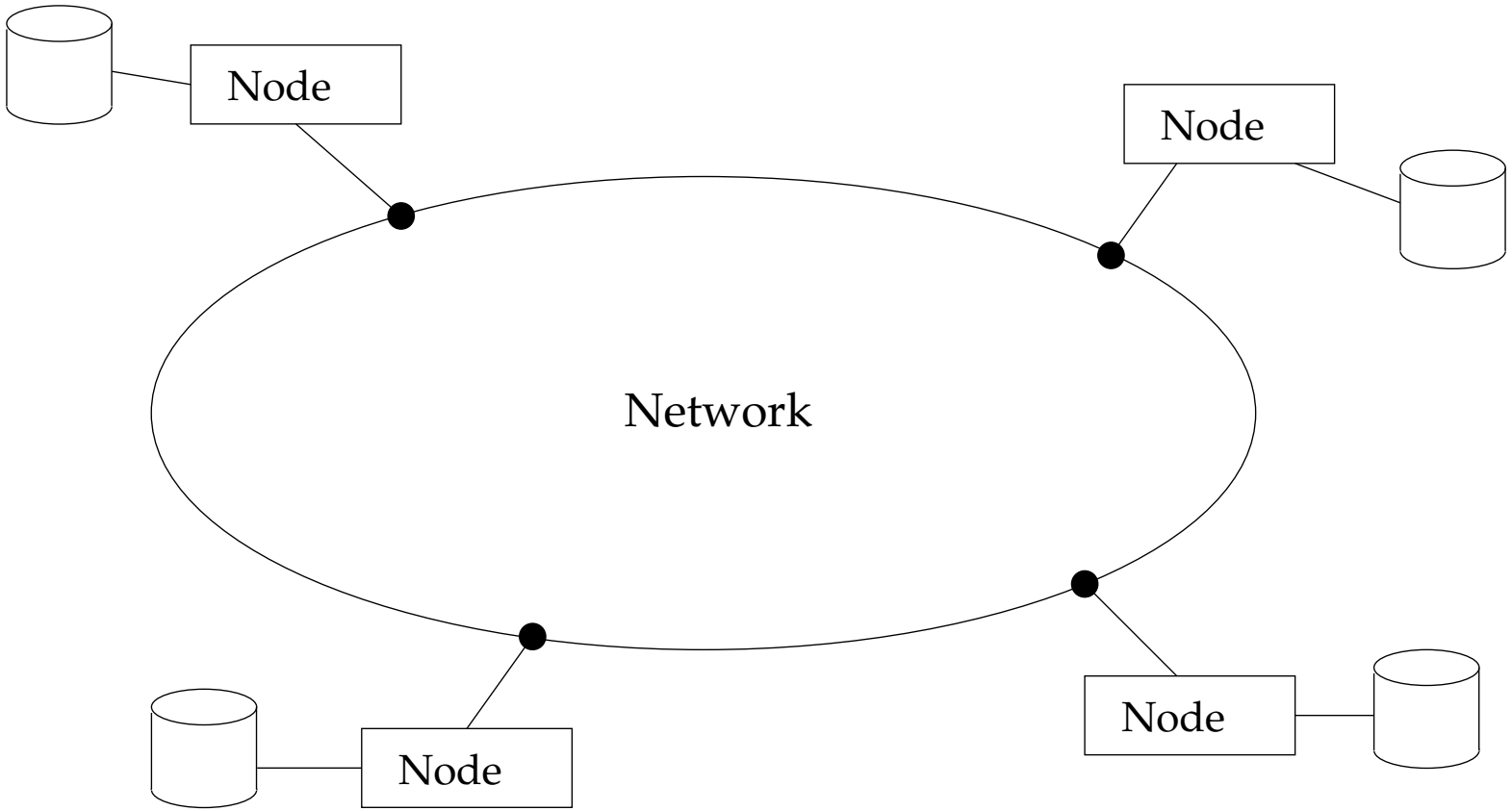


- New requirements
  - Support for de-centralized organization structures
  - High availability
  - High performance
  - Scalability

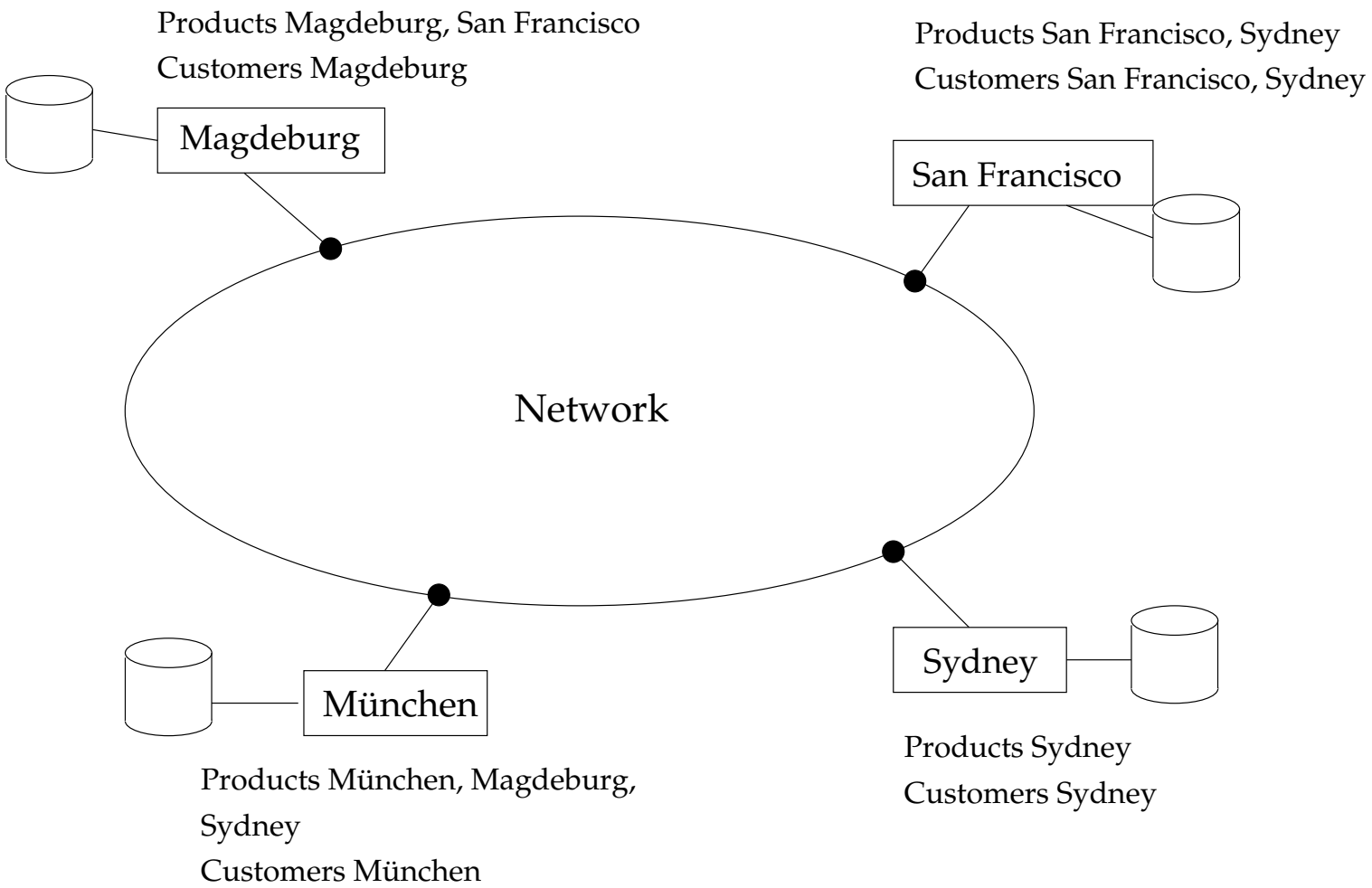
Client Server Data Management in a Network



Distributed Data Management



Distributed Data Management: Example



Advantages of Distributed DBMS

- Transparent management of distributed/replicated data
- Availability and fault tolerance
- Performance
- Scalability

Transparent Data Management

- Transparency: "‘hide"’ implementation details
- For (distributed) database systems
  - Data independence (physical, logical)
  - Network transparency
    - "‘hide"’ existence of the network
    - "‘hide"’ physical location of data
  - To applications a distributed DBS looks just like a centralized DBS

Transparent Data Management/2

- continued:
  - Replication transparency
    - ★ Replication: managing copies of remote data (performance, availability, fault-tolerance)
    - ★ Hiding the existence of copies (e.g. during updates)
  - Fragmentation transparency
    - ★ Fragmentation: decomposition of relations and distribution of resulting fragments
    - ★ Hiding decomposition of global relation

Fault-Tolerance

- Failure of one single node can be compensated
- Requires
  - Replicated copies on different nodes
  - Distributed transactions

Performance

- Data can be stored, where they are most likely used $\rightarrow$ reduction of transfer costs
- Parallel processing in distributed systems
  - Inter-transaction-parallelism: parallel processing of different transactions
  - Inter-query-parallelism: parallel processing of different queries
  - Intra-query-parallelism: parallel of one or several operations within one query

Scalability

- Requirements raised by growing databases or necessary performance improvement
  - Addition of new nodes/processors often cheaper than design of new system or complex tuning measures

Differentiation: Distributed Information System

- Distributed Information System
  - Application components communicate for purpose of data exchange (distribution on application level)
- Distributed DBS
  - Distribution solely realized on the DBS-level

Differentiation: Distributed File System

- Distributed File System provides non-local storage access by means of operating system
- DBMS on distributed file system
  - All data must be read from blocks stored on different disks
  - Processing is performed only within DBMS node (not distributed)
  - Distribution handled by operating system

Special Case: Parallel DBS

- Data management on simultaneous computer (multi processor, special hardware)
- Processing capacities are used for performance improvement
- Example
  - 100 GB relation, sequential read with 10 MB/s $\leadsto$ 17 minutes
  - parallel read on 10 nodes (without considering coordination overhead)
    $\leadsto$ 1:40 minutes

Special Case: Heterogeneous DBS

- Motivation: integration of previously existing DBS (legacy systems)
  - Integrated access: global queries, relationships between data objects in different databases, global integrity
- Problems
  - Heterogeneity on different levels: system, data model, schema, data
- Special importance on the WWW: integration of Web sources $\rightsquigarrow$ Mediator concept

Special Case: Peer-to-Peer-Systems

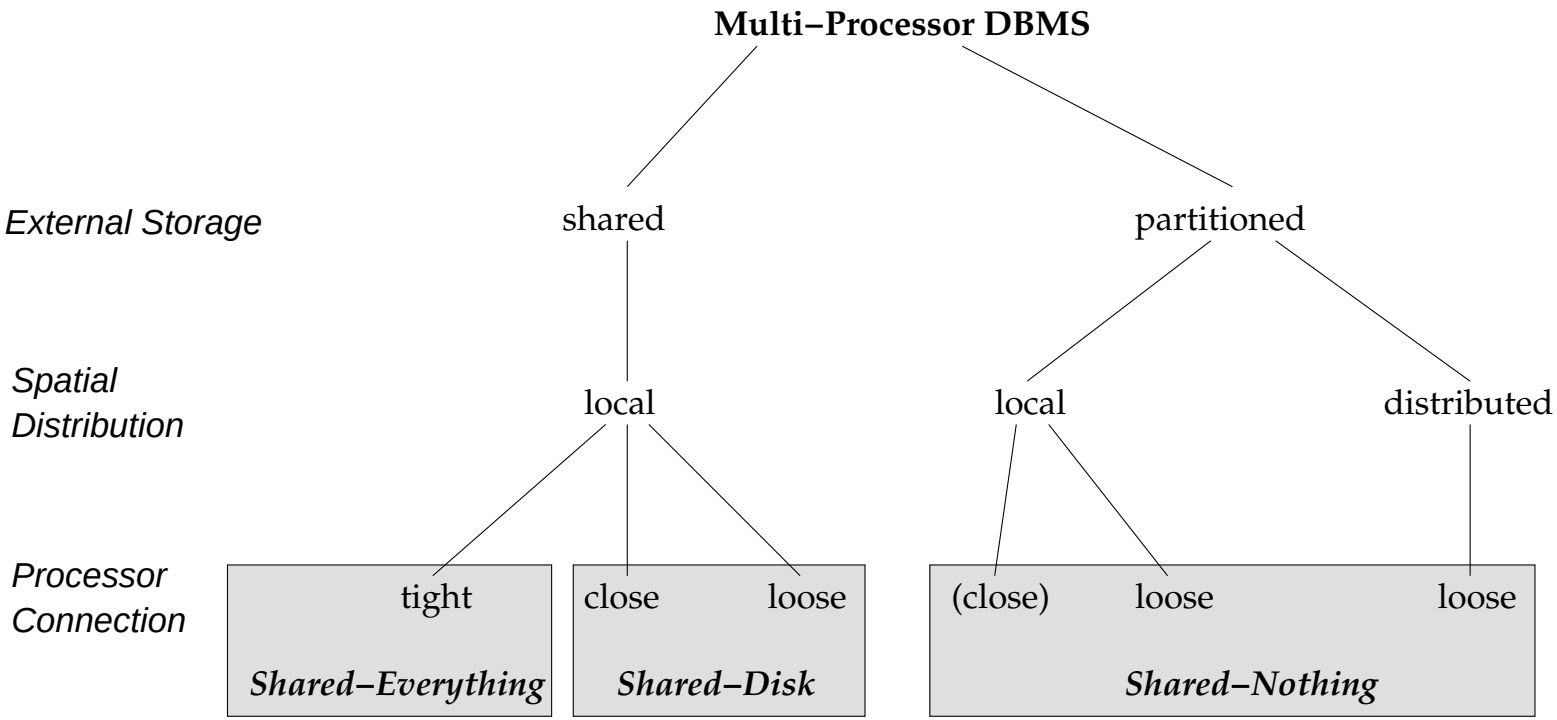
- Peer-to-Peer (P2P): networks without centralized servers
  - All / many nodes (peers) store data
  - Each node knows only some "'close"' neighbors
    - No global view
    - No centralized coordination
- Examples: Napster, Gnutella, Freenet, BitTorrent, ...
  - Distributed management of data (e.g. MP3-Files)
  - Lookup using centralized servers (Napster) or distributed (Gnutella)

Multi-Processor DBMS

- In general: DBMS which are able to use multiple processors or DBMS-instances to process database operations [Rahm 94]
- Can be classified according to different criteria
  - Processors with same or different functionality
  - Access to external storage
  - Spatial distribution
  - Processor connection
  - Homogeneous vs. heterogeneous architecture

Classification Overview

- Assumption: each processor provides the same functionality
- Classification [Rahm94]



Criterion: Access to External Storage

- Partitioned access
  - External storage is divided among processors/nodes
    - ⋆ Each processor has only access to local storage
    - ⋆ Accessing different partitions requires communication
- Shared access
  - Each processor has access to full database
  - Requires synchronisation

Criterion: Spatial Distribution

- Locally distributed: DB-Cluster
  - Fast inter-processor communication
  - Fault-tolerance
  - Dynamic load balancing possible
  - Little administration efforts
  - Application: parallel DBMS, solutions for high availabilty
- Remotely distributed: distributed DBS in WAN scenarios
  - Support for distributed organization structures
  - Fault-tolerant (even to major catastrophes)
  - Application: distributed DBS

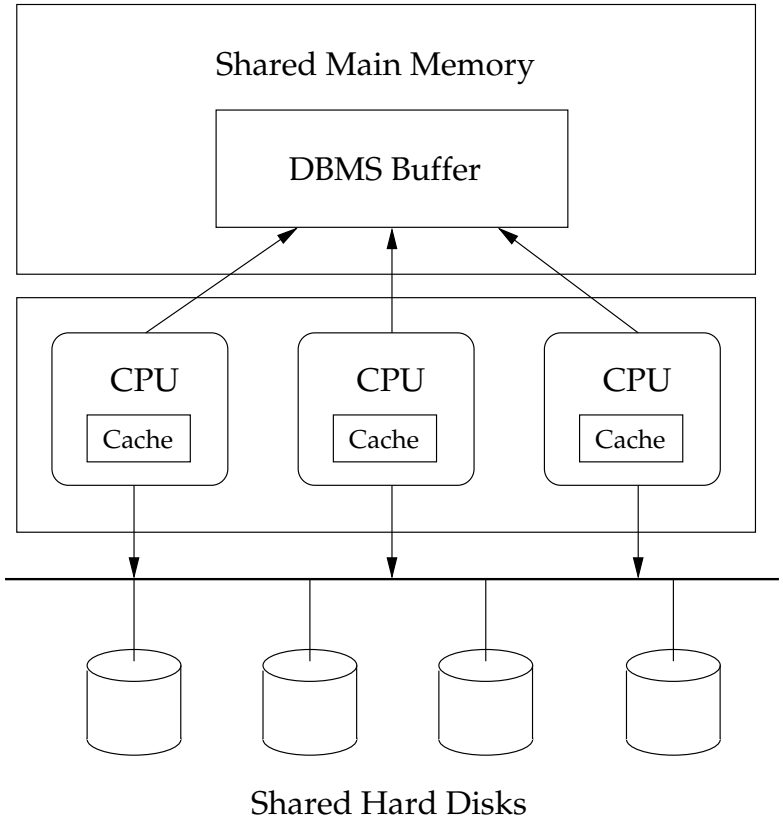
Criterion: Processor Connection

- Tight connection
  - Processors share main memory
  - Efficient co-operation
  - Load-balancing by means of operating system
  - Problems: Fault-tolerance, cache coherence, limited number of processors (≤ 20)
  - Parallel multi-processor DBMS

Criterion: Processor Connection /2

- Loose connection:
  - Independent nodes with own main memory and DBMS instances
  - Advantages: failure isolation, scalability
  - Problems: expensive network communication, costly DB operations, load balancing
- Close connection:
  - Mix of the above
  - In addition to own main memory there is connection via shared memory
  - Managed by operating system

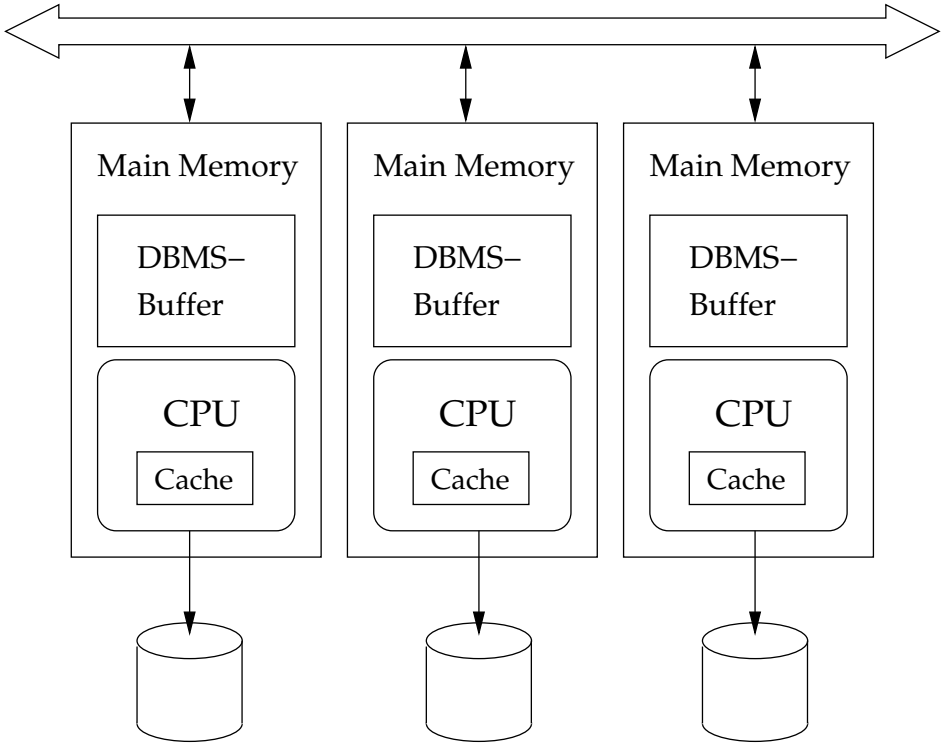
Class: Shared-Everything



Class: Shared-Everything /2

- Simple realization of DBMS
- Distribution transparency provided by operating system
- Expensive synchronization
- Extended implementation of query processing

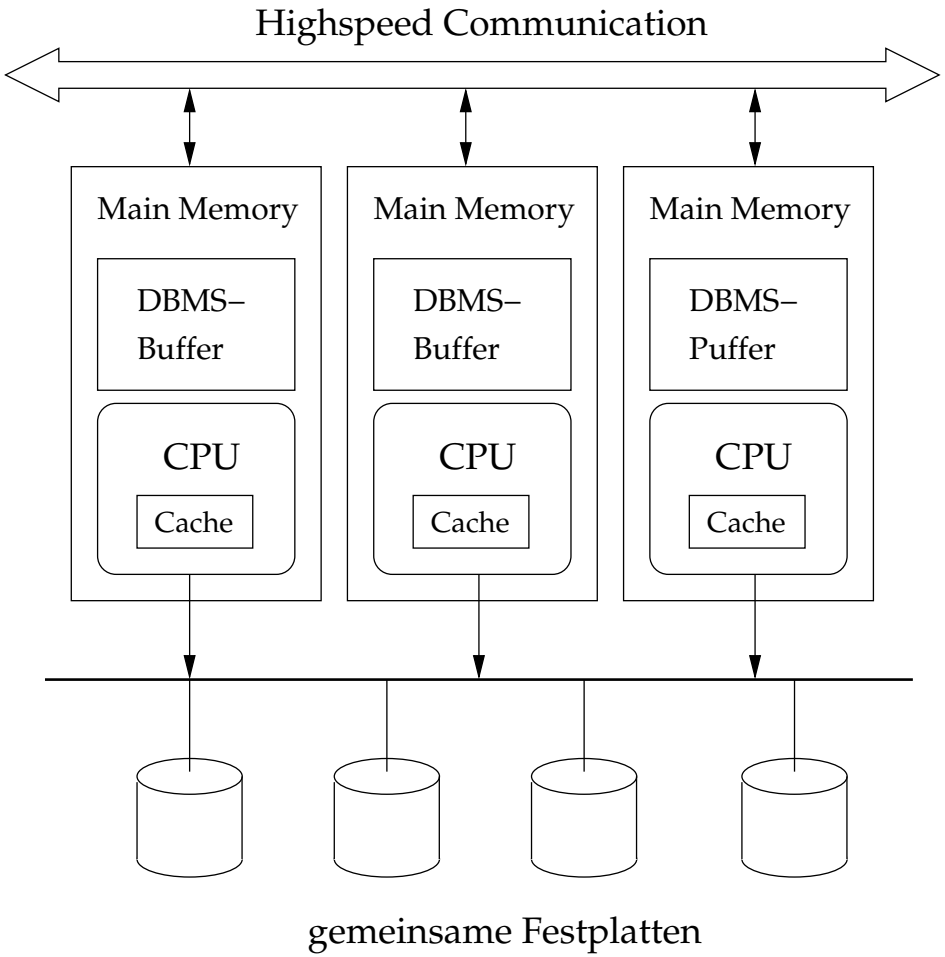
Class: Shared-Nothing



Class: Shared-Nothing /2

- Distribution of DB across various nodes
- Distributed/parallel execution plans
- TXN management across participating nodes
- Management of catalog and replicas

Class: Shared-Disk



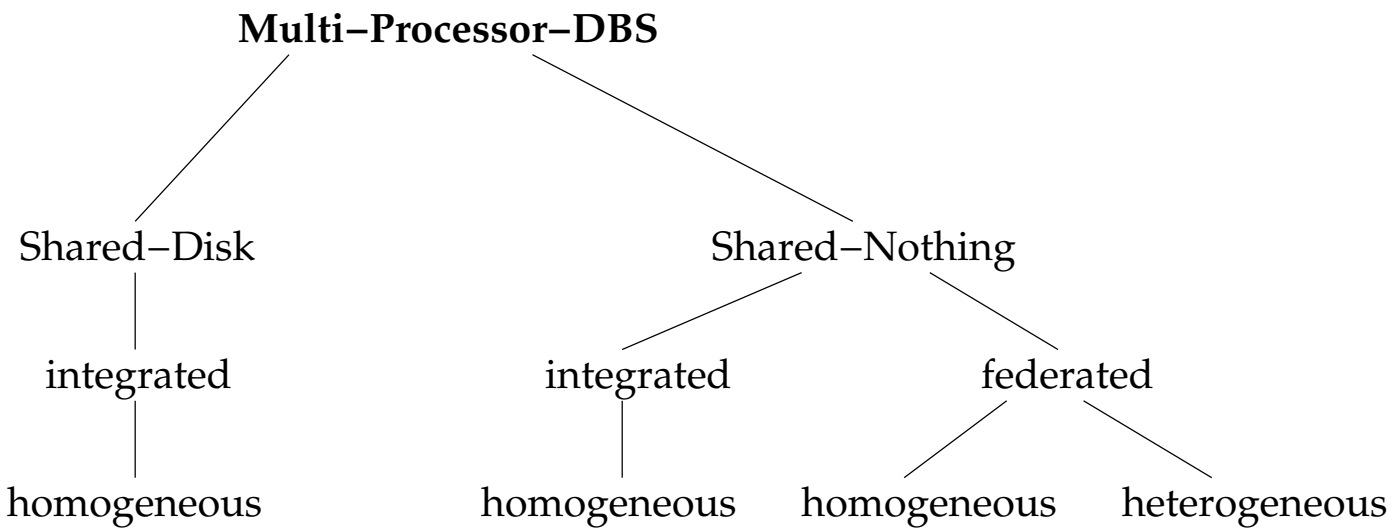
Class: Shared-Disk /2

- Avoids physical data distribution
- No distributed TXNs and query processing
- Requires buffer invalidation

Criterion: Integrated vs. Federated DBS

- Integrated:
  - Shared database for all nodes $\rightsquigarrow$ one conceptual schema
  - High distribution transparency: access to distributed DB via local DBMS
  - Requires co-operation of DBMS nodes $\rightsquigarrow$ restricted autonomy
- Federated:
  - Nodes with own DB and own conceptual schema
  - Requires schema integration $\rightsquigarrow$ global conceptual schema
  - High degree of autonomy of nodes

Criterion: Integrated vs. Federates DBS /2



Criterion: Centralized vs. De-centralized Coordination

- Centralized:
  - Each node has global view on database (directly of via master)
  - Central coordinator: initiator of query/transaction $\rightarrow$ knows all participating nodes
  - Provides typical DBS properties (ACID, result completeness, etc.)
  - Applications: distributed and parallel DBS
    - ★ Limited availability, fault-tolerance, scalabilty

Criterion: Centralized vs. De-centralized Coordination /2

- De-centralized:
  - No global view on schema $\rightarrow$ peer knows only neighbors
  - Autonomous peers; global behavior depends on local interaction
  - Can not provide typical DBMS properties
  - Application: P2P systems
    - ⋆ Advantages: availability, fault-tolerance, scalabilty

Comparison

	Parallel DBS	Distributed DBS	Federated DBS
High TXN rates	↑	→↗	→
Intra-TXN-Parallelism	↑	→↗	↘→
Scalability	↗	→↗	→
Availability	↗	↗	↘
Geogr. Distribution	↘	↗	↗
Node Autonomy	↘	→	↗
DBS-Heterogeneity	↘	↘	↗
Administration	→	↘	↘↓

Database Management Systems (DBMS)

- Nowadays commonly used
  - to store huge amounts of data persistently,
  - in collaborative scenarios,
  - to fulfill high performance requirements,
  - to fulfill high consistency requirements,
  - as a basic component of information systems,
  - to serve as a common IT infrastructure for departments of an organization or company.

Database Management Systems

A **database management system (DBMS)** is a suite of computer programs designed to manage a database and run operations on the data requested by numerous clients.

A **database (DB)** is an organized collection of data.

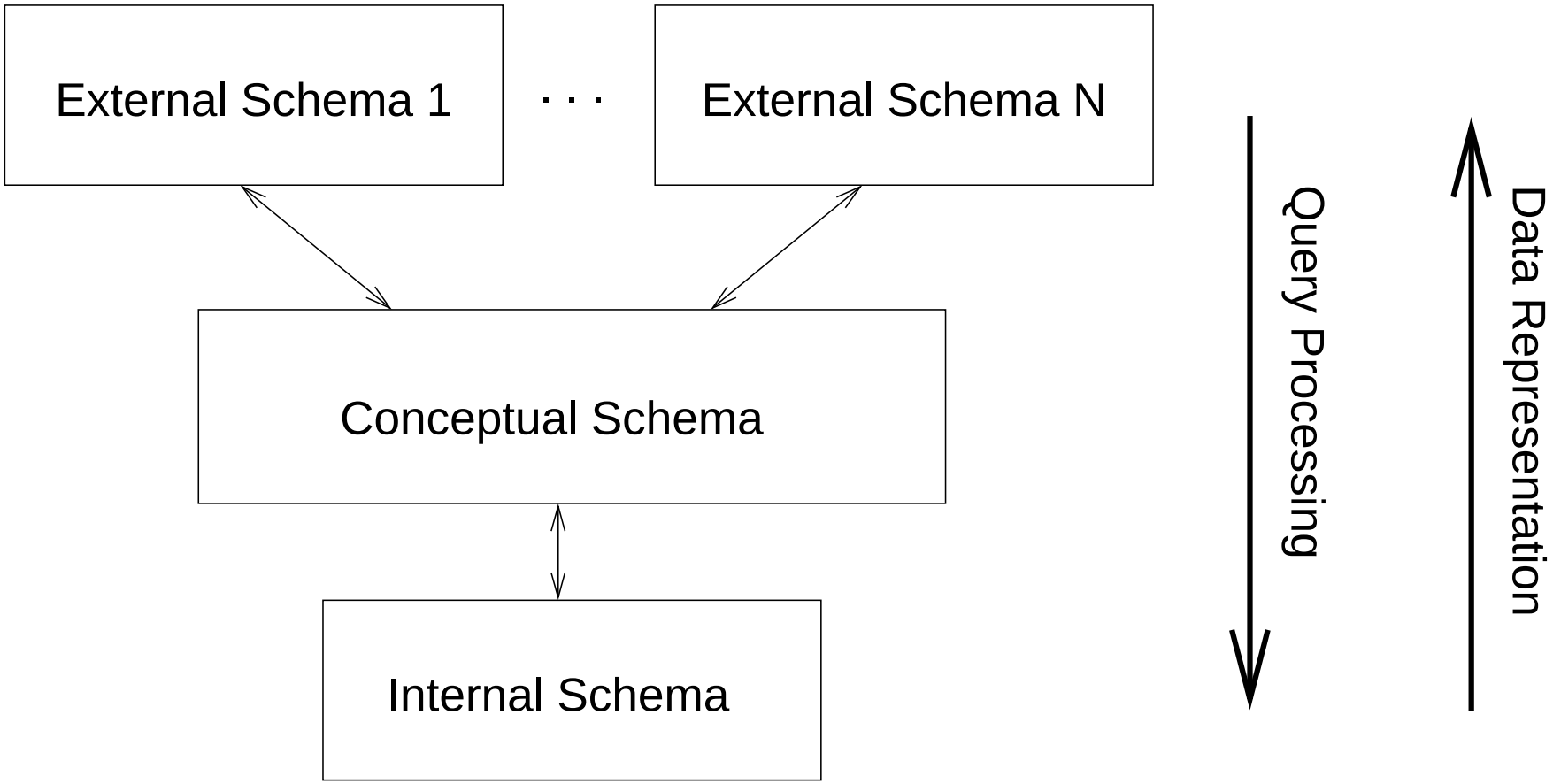
A **database system (DBS)** is the concrete instance of a database managed by a database management system.

Codd's 9 Rules for DBMS

- Differentiate DBMS from other systems managing data persistently, e.g. file systems

1. **Integration:** homogeneous, non-redundant management of data
2. **Operations:** means for accessing, creating, modifying, and deleting data
3. **Catalog:** the data description must be accessible as part of the database itself
4. **User views:** different users/applications must be able to have a different perception of the data
5. **Integrity control:** the systems must provide means to grant the consistency of data
6. **Data security:** the system must grant only authorized accesses
7. **Transactions:** multiple operations on data can be grouped into a

3 Level Schema Architecture



- Important concept of DBMS
- Provides
  - transparency, i.e. non-visibility, of storage implementation details
  - ease of use

Data Independence

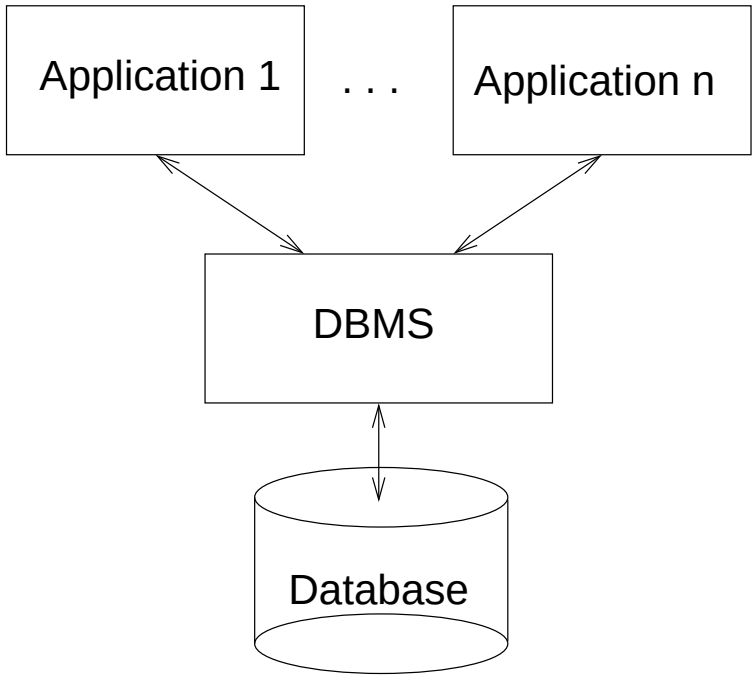
Logical data independence: Changes to the logical schema level must not require a change to an application (external schema) based on the structure.

Physical data independence: Changes to the physical schema level (how data is stored) must not require a change to the logical schema.

Architecture of a DBS

Schema architecture roughly conforms to general architecture of a database systems

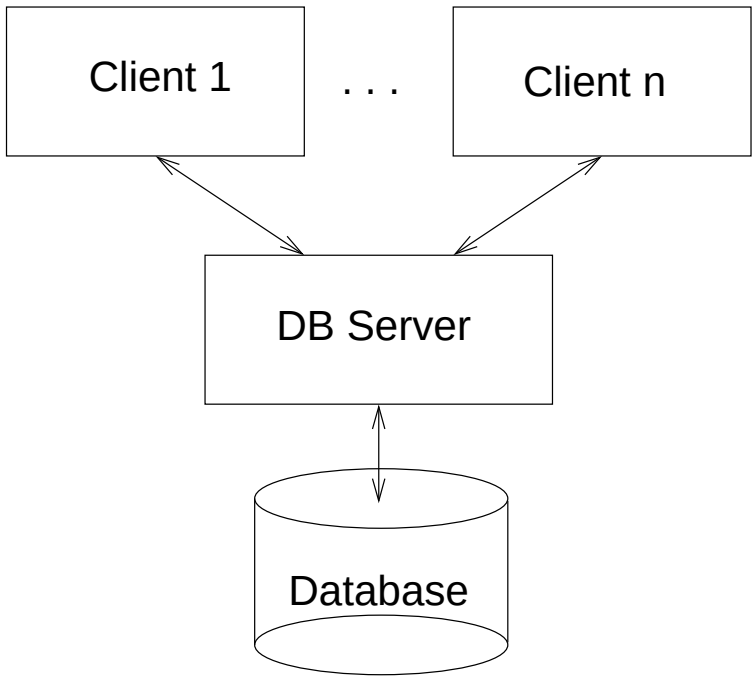
- **Applications** access database using specific **views (external schema)**
- The **DBMS** provides access for all applications using the **logical schema**
- The **database** is stored on secondary storage according to an **internal schema**



Client Server Architecture

Schema architecture does not directly relate to client server architecture (communication/network architecture)

- Clients may run several applications
- Applications may run on several clients
- DB servers may be distributed
- ...

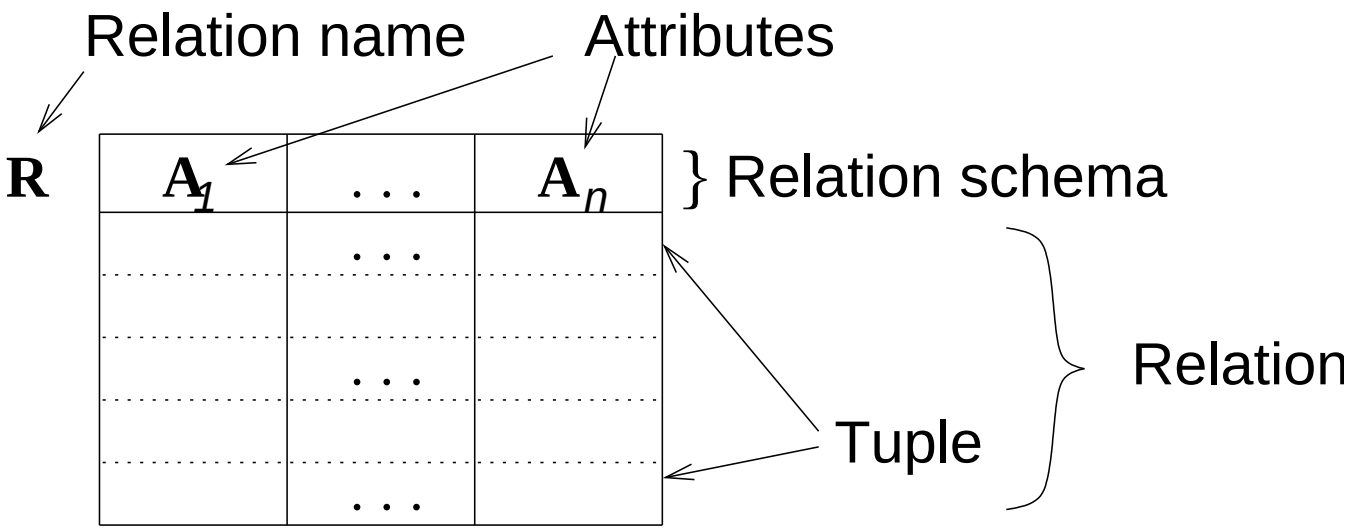


The Relational Model

- Developed by Edgar F. Codd (1923-2003) in 1970
- Derived from mathematical model of n-ary relations
- Colloquial: data is organized as tables (relations) of records (n-tuples) with columns (attributes)
- Currently most commonly used database model
- Relational Database Management Systems (RDBMS)
- First prototype: IBM System R in 1974
- Implemented as core of all major DBMS since late ’70s: IBM DB2, Oracle, MS SQL Server, Informix, Sybase, MySQL, PostgreSQL, etc.
- Database model of the DBMS language standard SQL

Basic Constructs

A **relational database** is a database that is structured according to the relational database model. It consists of a set of relations.



Integrity Constraints

- Static integrity constraints describe valid tuples of a relation
  - Primary key constraint
  - Foreign key constraint (referential integrity)
  - Value range constraints
  - ...
- In SQL additionally: uniqueness and not-NULL
- Transitional integrity constraints describe valid changes to a database

The Relational Algebra

A **relational algebra** is a set of operations that are closed over relations.

- Each operation has one or more relations as input
- The output of each operation is a relation

Relational Operations

Primitive operations:

- Selection σ
- Projection π
- Cartesian product (cross product) $\times$
- Set union $\cup$
- Set difference $-$
- Rename β

Non-primitive operations

- Natural Join $\bowtie$
- θ-Join and Equi-Join $\bowtie_{\varphi}$
- Semi-Join $\ltimes$
- Outer-Joins $= \times$
- Set intersection $\cap$
- ...

Notation for Relations and Tuples

- If R denotes a relation schema (set of attributes), than the function $r(R)$ denotes a relation conforming to that schema (set of tuples)
- R and $r(R)$ are often erroneously used synonymously to denote a relation, assuming that for a given relation schema only one relation exists
- $t(R)$ denotes a tuple conforming to a relation schema
- $t(R.a)$ denotes an attribute value of a tuple for an attribute $a \in R$

The Selection Operation σ

Select tuples based on predicate or complex condition

PROJECT			
PNAME	**PNUMBER**	**PLOCATION**	**DNUM**
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

$$\sigma_{PLOCATION='Stafford'}(r(PROJECT))$$

PNAME	**PNUMBER**	**PLOCATION**	**DNUM**
Computerization	10	Stafford	4
Newbenefits	30	Stafford	4

The Projection Operation π

Project to set of attributes - remove duplicates if necessary

PROJECT			
PNAME	**PNUMBER**	**PLOCATION**	**DNUM**
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

$$\pi_{PLOCATION,DNUM}(r(PROJECT))$$

PLOCATION	**DNUM**
Bellaire	5
Sugarland	5
Houston	5
Stafford	4
Houston	1

Cartesian or cross product $\times$

Create all possible combinations of tuples from the two input relations

R	
A	**B**
1	2
3	4

S		
C	**D**	**E**
5	6	7
8	9	10
11	12	13

$$r(R) \times r(S)$$

A	**B**	**C**	**D**	**E**
1	2	5	6	7
1	2	8	9	10
1	2	11	12	13
3	4	5	6	7
3	4	8	9	10
3	4	11	12	13

Set: Union, Intersection, Difference

- All require compatible schemas: attribute names and domains
- Union: duplicate entries are removed
- Intersection and Difference: $\emptyset$ as possible result

The Natural Join Operation $\bowtie$

- Combine tuples from two relations $r(R)$ and $r(S)$ where for
  - all attributes $a = R \cap S$ (defined in both relations)
  - is $t(R.a) = t(S.a)$.
- Basic operation for following key relationships
- If there are no common attributes result is Cartesian product
  $R \cap S = \emptyset \Longrightarrow r(R) \bowtie r(S) = r(R) \times r(S)$
- Can be expressed as combination of π, σ and $\times$
  $r(R) \bowtie r(S) = \pi_{R \cup S}(\sigma_{\bigwedge_{a \in R \cap S} t(R.a)=t(S.a)}(r(R) \times r(S)))$

The Natural Join Operation $\bowtie$ /2

R	
A	**B**
1	2
3	4
5	6

S		
B	**C**	**D**
4	5	6
6	7	8
8	9	10

$$r(R) \bowtie r(S)$$

A	**B**	**C**	**D**
3	4	5	6
5	6	7	8

The Semi-Join Operation $\ltimes$

- Results all tuples from one relation having a (natural) join partner in the other relation
  $r(R) \ltimes r(S) = \pi_R(r(R) \bowtie r(S))$

PERSON	
PID	**NAME**
1273	Dylan
2244	Cohen
3456	Reed

CAR	
PID	**BRAND**
1273	Cadillac
1273	VW Beetle
3456	Stutz Bearcat

$r(PERSON) \ltimes r(CAR)$

PID	NAME
1273	Dylan
3456	Reed

Other Join Operations

- **Conditional join:** join condition φ is explicitly specified
  $r(R) \bowtie_{\varphi} r(S) = \sigma_{\varphi}(r(R) \times r(S))$
- **θ-Join:** special conditional join, where φ is a single predicate of the form $a\theta b$ with $a \in R$, $b \in S$, and $\theta \in \{=, \neq, >, <, \leq, \geq, \dots\}$
- **Equi-Join:** special θ-Join where θ is $=$.
- **(Left or Right) Outer Join:** union of natural join result and tuples from the left or right input relation which could not be joined (requires NULL-values to grant compatible schemas).

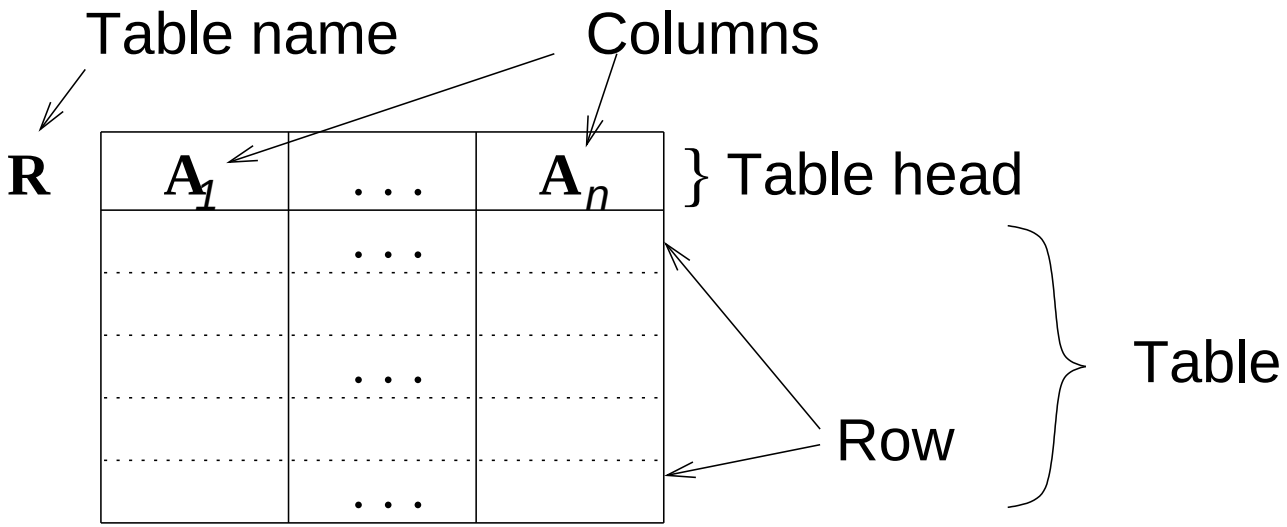
Relational Database Management Systems

A **Relational Database Management System (RDBMS)** is a database management system implementing the relational database model.

- Today, most relational DBMS implement the SQL database model
- There are some significant differences between the relational model and SQL (duplicate rows, tuple order significant, anonymous column names, etc.)
- Most distributed and parallel DBMS have a relational (SQL) data model

SQL Data Model

- Said to implement relational database model
- Defines own terms



- Some significant differences exist

Structured Query Language

- **Structured Query Language (SQL):** declarative language to describe requested query results
- Realizes relational operations (with the mentioned discrepancies)
- Basic form: `SELECT-FROM-WHERE`-block (SFW)

```
SELECT FNAME, LNAME, MGRSTARTDATE
FROM EMPLOYEE, DEPARTMENT
WHERE SSN=MGRSSN
```

SQL: Selection σ

$\sigma_{DNO=5 \wedge SALARY>30000}(r(EMPLOYEE))$

```
SELECT *
FROM EMPLOYEE
WHERE DNO=5 AND SALARY>30000
```

SQL: Projection π

$\pi_{LNAME,FNAME}(r(EMPLOYEE))$

```
SELECT LNAME,FNAME
FROM EMPLOYEE
```

- Difference to RM: does not remove duplicates
- Requires additional `DISTINCT`

```
SELECT DISTINCT LNAME,FNAME
FROM EMPLOYEE
```

SQL: Cartesian Product $\times$

$r(EMPLOYEE) \times r(PROJECT)$

```
SELECT *
FROM EMPLOYEE, PROJECT
```

SQL: Natural Join $\bowtie$

$r(DEPARTMENT) \bowtie r(DEPARTMENT\_LOCATIONS)$

```
SELECT *
FROM DEPARTMENT
     NATURAL JOIN DEPARTMEN_LOCATIONS
```

SQL: Equi-Join

$r(EMPLOYEE) \bowtie_{SSN=MGRSSN} r(DEPARTMENT)$

```
SELECT *
FROM EMPLOYEE, DEPARTMENT
WHERE SSN=MGRSSN
```

SQL: Union

$r(R) \cup r(S)$

```
SELECT * FROM R
UNION
SELECT * FROM S
```

- Other set operations: `INTERSECT`, `MINUS`
- Does remove duplicates (in compliance with RM)
- If duplicates required:

```
SELECT * FROM R
UNION ALL
SELECT * FROM S
```

SQL: Other Features

- SQL provides several features not in the relational algebra
  - Grouping And Aggregation Functions, e.g. `SUM`, `AVG`, `COUNT`, ...
  - Sorting

```
SELECT PLOCATION, AVG(HOURS)
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE SSN=ESSN AND PNO=PNUMBER
GROUP BY PLOCATION
HAVING COUNT(*) > 1
ORDER BY PLOCATION
```

SQL DDL

- **Data Definition Language** to create, modify, and delete schema objects

```
CREATE DROP ALTER TABLE mytable ( id INT, ...)
DROP TABLE ...
ALTER TABLE ...
CREATE VIEW myview AS SELECT ...
DROP VIEW ...
CREATE INDEX ...
DROP INDEX ...
...
```

Simple Integrity Constraints

```
CREATE TABLE employee(
    ssn INTEGER,
    lname VARCHAR2(20) NOT NULL,
    dno INTEGER,
    ...
    FOREIGN KEY (dno)
        REFERENCES department(dnumber),
    PRIMARY KEY (ssn)
)
```

- Additionally: triggers, explicit value domains, ...

SQL DML

- **Data Manipulation Language** to create, modify, and delete tuples

```
INSERT INTO (<COLUMNS>) mytable VALUES (...)

INSERT INTO (<COLUMNS>) mytable SELECT ...

UPDATE mytable
SET ...
WHERE ...

DELETE FROM mytable
WHERE ...
```

Other Parts of SQL

- Data Control Language (DCL):
  `GRANT, REVOKE`
- Transaction management:
  `START TRANSACTION, COMMIT, ROLLBACK`
- Stored procedures and imperative programming concepts
- Cursor definition and management

Transactions

- Sequence of database operations
  - Read and write operations
  - In SQL sequence of INSERT, UPDATE, DELETE, SELECT statements
- Build a semantic unit, e.g. transfer of an amount from one bank account to another
- Has to conform to **ACID** properties

Transactions: ACID Properties

- **A**tomicity means that a transaction can not be interrupted or performed only partially
  - TXN is performed in its entirety or not at all
- **C**onsistency to preserve data integrity
  - A TXN starts from a consistent database state and ends with a consistent database state
- **I**solation
  - Result of a TXN must be independent of other possibly running parallel TXNs
- **D**urability or persistence
  - After a TXN finished successfully (from the user's view) its results must be in the database and the effect can not be reversed

Functional Dependencies

- A functional dependency (FD) $X \rightarrow Y$ within a relation between sets $r(R)$ of attributes $X \subseteq R$ and $Y \subseteq R$ exists, if for each tuple the values of X determine the values for Y
- i.e.

$$\forall t_1, t_2 \in r(R) : t_1(X) = t_2(X) \Rightarrow t_1(Y) = t_2(Y)$$

Derivation Rules for FDs

R_1	Reflexivity	if $X \supseteq Y \implies X \to Y$
R_2	Accumulation	$\{X \to Y\} \implies XZ \to YZ$
R_3	Transitivity	$\{X \to Y, Y \to Z\} \implies X \to Z$
R_4	Decomposition	$\{X \to YZ\} \implies X \to Y$
R_5	Unification	$\{X \to Y, X \to Z\} \implies X \to YZ$
R_6	Pseudotransitivity	$\{X \to Y, WY \to Z\} \implies WX \to Z$

R_1-R_3 known as *Armstrong-Axioms* (sound, complete)

Normal Forms

- Formal criteria to force schemas to be free of redundancy
- First Normal Form (1NF) allows only *atomic attribute values*
  - i.e. all attribute values ar of basic data types like `integer` or `string` but not further structured like e.g. an `array` or a `set` of values
- Second Normal Form (2NF) avoids *partial dependencies*
  - A partial dependency exist, if a non-key attribute is functionally dependent on a real subset of the primary key of the relation

Normal Forms /2

- Third Normal Form (3NF) avoids *transitive dependencies*
  - Disallows functional dependencies between non-key attributes
- Boyce-Codd-Normal Form (BCNF) disallows *transitive dependencies* also for primary key attributes

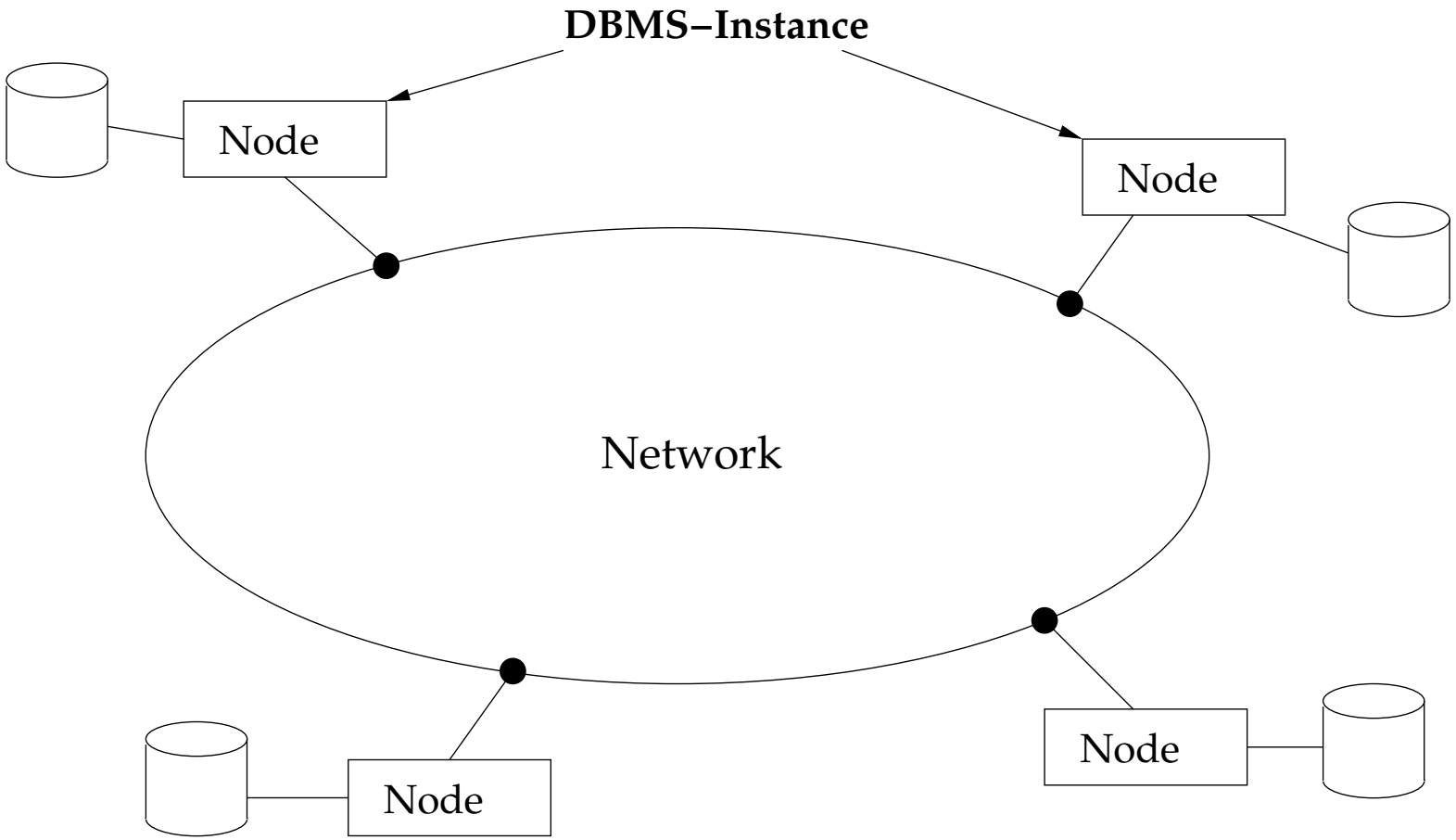
Part II

Distributed Database Systems

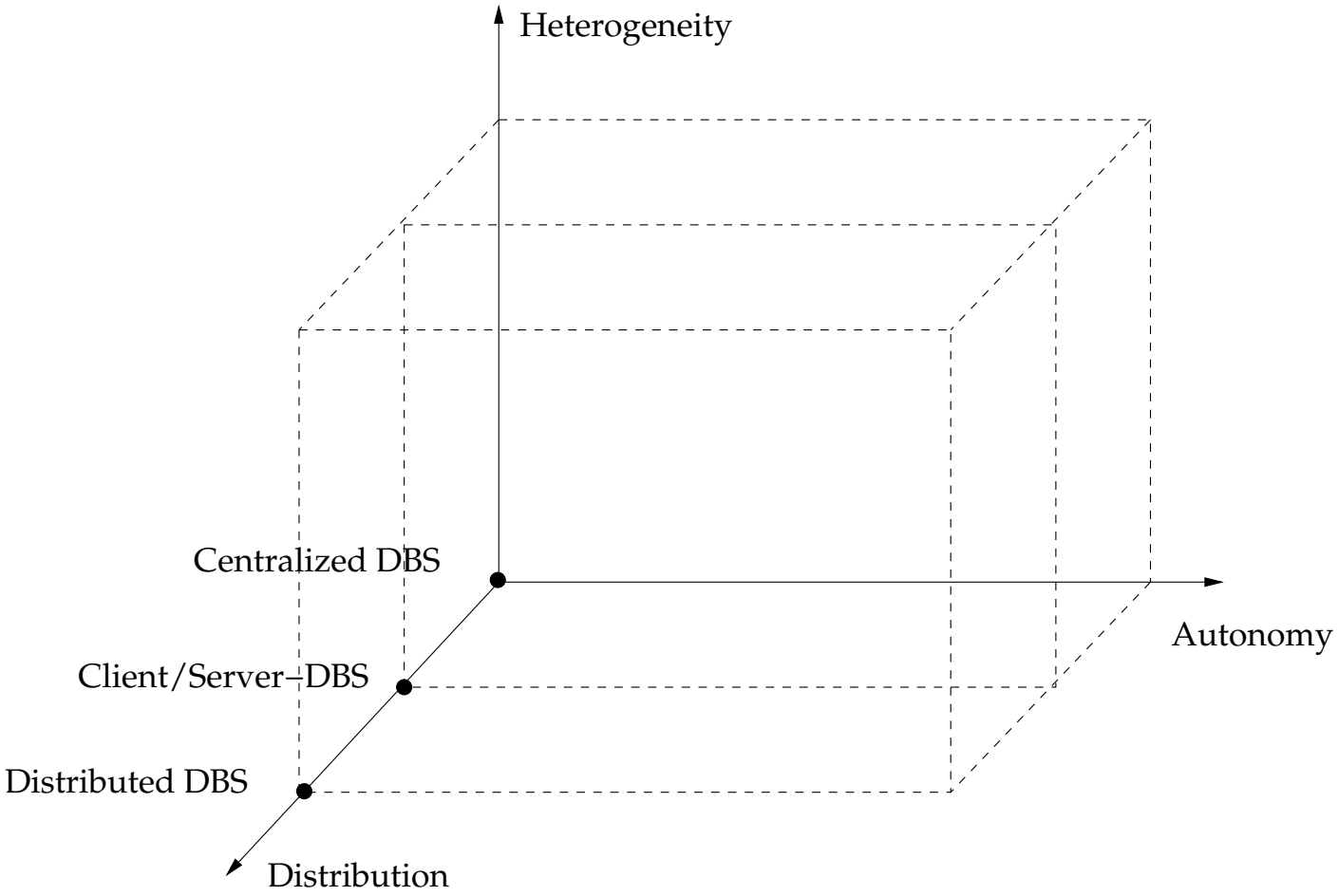
Overview

- Foundations of DDBS
- Catalog Management
- DDBS Design: Fragmentation
- Allocation and Replication
- Overview
- Data Localization
- Join Processing
- Global Optimization

Architecture & Data Distribution



Dimensions


Heterogeneity
Centralized DBS
Client/Server−DBS
Distributed DBS
Autonomy
Distribution

12 Rules for DDBMS by Date

1. Local Autonomy
   - Component system have maximal control over own data, local access does not require access to other components
2. No reliance on central site
   - Local components can perform independently of central component
3. Continuous operation/high availability
   - Overall system performs despite local interrupt
4. Location transparency
   - User of overall system should not be aware of physical storage location

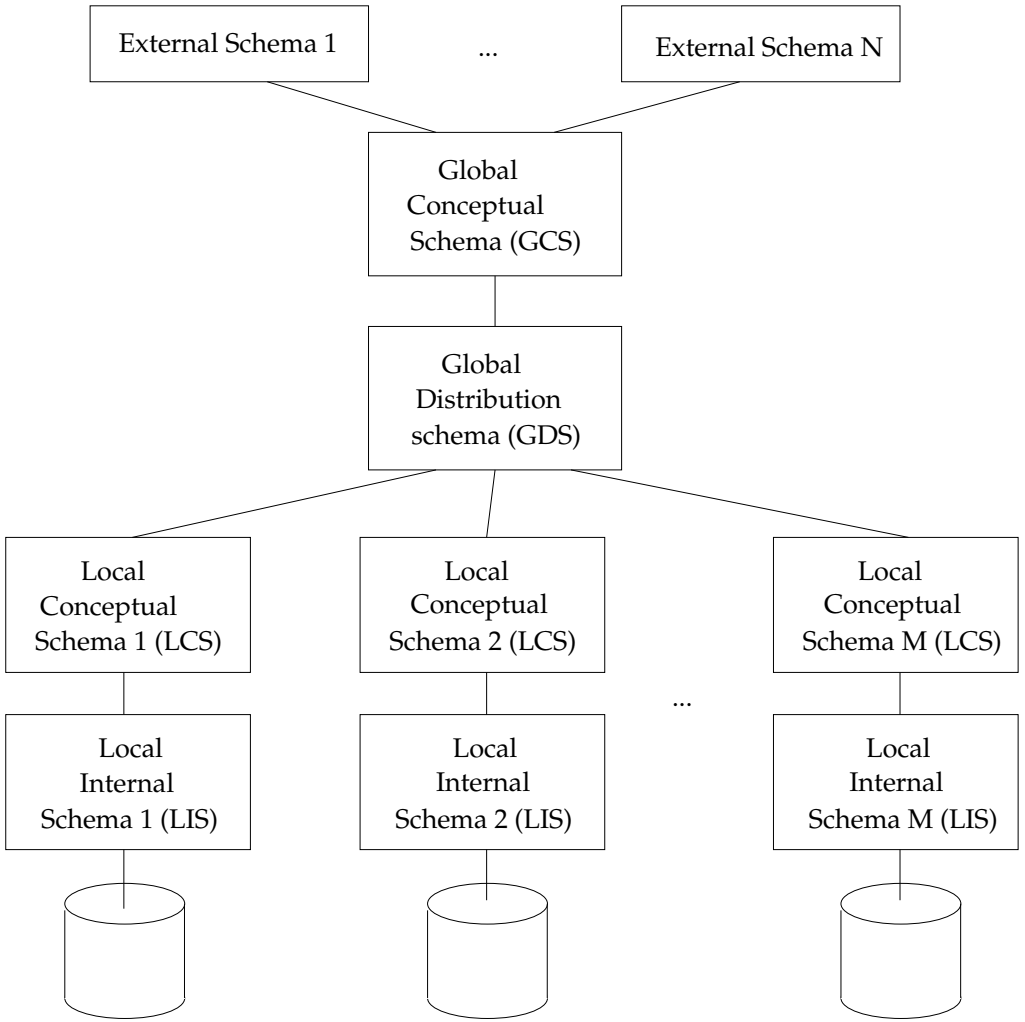
12 Rules for DDBMS by Date /2

5. Fragmentation transparency
   - If data of one relation is fragmented, user should not be aware of this
6. Replication transparency
   - User should not be aware of redundant copies of data
   - Management and redundancy is controlled by DBMS
7. Distributed query processing
   - Efficient access to data stored on different sites within one DB operation

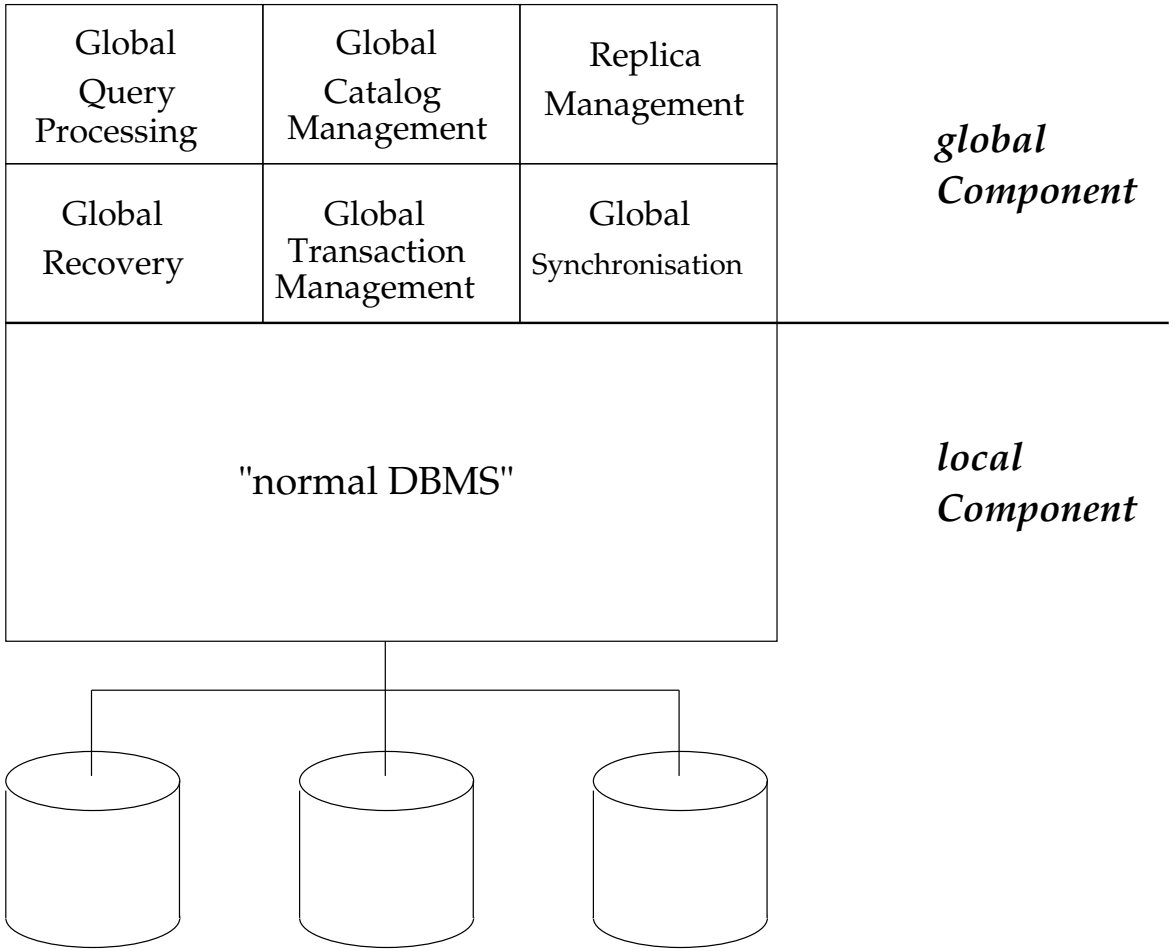
12 Rules for DDBMS by Date /3

8. Distributed Transaction Management
   - ACID properties must persist for distributed operations
9. Hardware independence
   - Component DB processing on different hardware platforms
10. Operating system independence
    - Component DB processing on different OS
11. Network independence
    - DB processing using different network protocols
12. DBMS independence (ideal)
    - Usage of different DBMS possible

Schema Architecture



System Architecture



Catalog Management

- Catalog: collection of metadata (schema, statistics, access rights, etc.)
  - Local catalog
    - ⋆ Identical to catalog of a centralized DBS
    - ⋆ consistes of LIS and LCS
  - Global ctalaog
    - ⋆ Also contains GCS and GDS
    - ⋆ System-wide management of users and access rights
- Storage
  - Local catalog: on each node
  - Global catalog: centralized, replicated, or partitioned

Global Catalog /1

- Centralized: one instance of global catalog managed by central node
  - Advantages: only one update operation required, litte space consumption
  - Disadvantages: request for each query, potential bottleneck, critical ressource
- Replicated: full copy of global catalog stored on each node
  - Advantage: low communication overhead during queries, availabilty
  - Disadvantage: high overhead for updates
- Mix- form: cluster-catalog with centralized catalog for certain clusters of nodes

Global Catalog /2

- Partitioned: (relevant) part of the catalog is stored on each node
  - No explicit GCS $\rightsquigarrow$ union of LCS
  - Partitioned GDS by extend object (relations, etc.) names (see System R*)

Coherency Control

- Idea: buffer for non-local parts of the catalog
  - Avoids frequent remote accesses for often used parts of the catalog
- Problem: invalidation of buffered copies after updates

Coherency Control /2

- Approaches
  - Explicit invalidation:
    - ★ Owner of catalog data keeps list of copy sites
    - ★ After an update these nodes are informed of invalidation
  - Implicit invalidation:
    - ★ Identification of invalid catalog data during processing time using version numbers or timestamps (see System R*)

DB Object Name Management

- Task: identification of relations, views, procedures, etc.
- Typical schema object names in RDBMS:

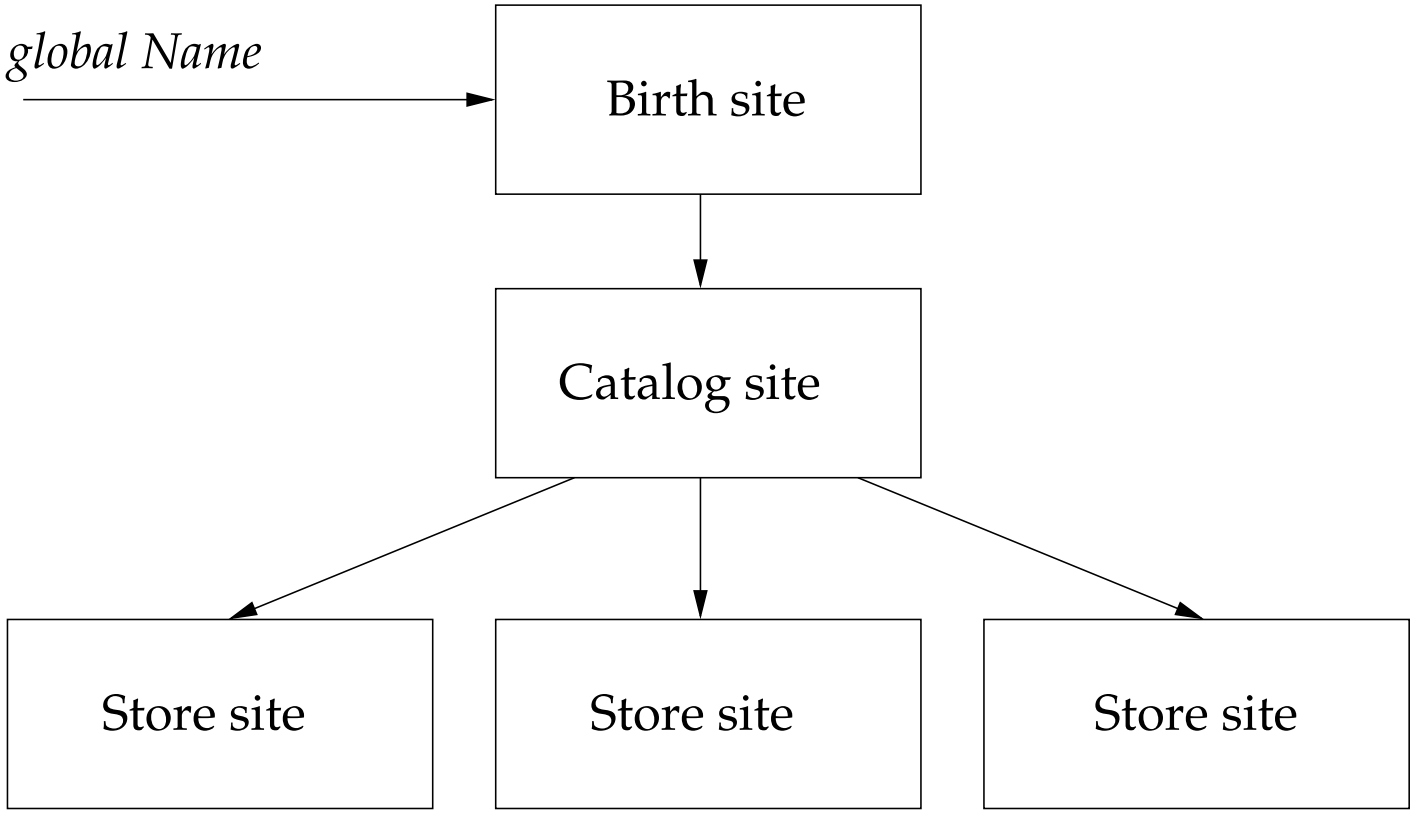
  [<*username*> .] <*objectname*>

- Requirement global uniqueness in DDBS
  - Name Server approach: management of names in centralized catalog
  - Hierarchic Naming: enrich object name with *node name*

    [[<*nodename*> .] <*username*> .] <*objectname*>

    - ⋆ Node name: birth site (or simplification via alias)

Name Management: Node Types



Catalog Management in System R*

- Birth site
  - Prefix of the relation name
  - Knows about storage sites
- Query processing
  - Executing node gets catalog entry of relevant relation
  - Catalog entry is buffered for later accesses

Catalog Management in System R* /2

- Query processing (continued)
  - Partial query plans include time stamp of catalog entry
  - Node processing partial query checks whether catalog time stamp is still current
- In case of failure: buffer invalidation, re-set query and new query translation according to current schema
- Summary:
  - Advantage: high degree of autinomy, user-controlled invalidation of buffered catalog data, good performance
  - Disadvantage: no uniform realization of global views

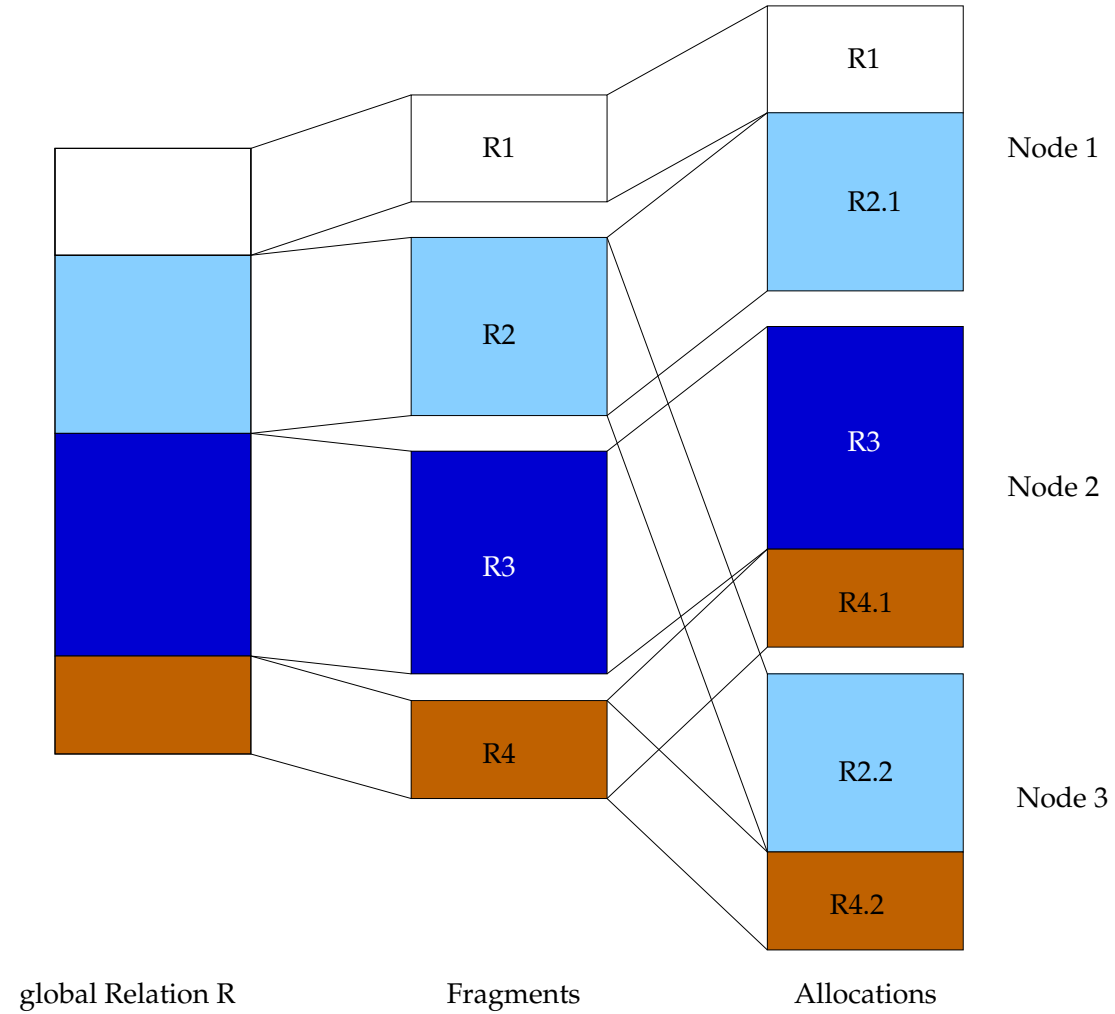
Database Distribution

- In Shared-Nothing-Systems (DDBS): definition of physical distribution of data
- Impact:
  - Communication efforts $\rightsquigarrow$ overall performance
  - Load balancing
  - Availability

Bottom Up vs. Top Down

- Bottom Up
  - Subsumption of local conceptual schemata (LCS) into global conceptual schema (GCS)
  - Integration of existing DB $\rightsquigarrow$ schema integration (Federated DBS)
- Top Down
  - GCS of local DB designed first
  - Distribution of schema to different nodes
  - *Distribution Design*

Distribution Design Tasks



Fragmentation

- Granularity of distribution: relation
  - Operations on one relation can always be performed on one node
  - Simplifies integrity control
- Granularity of distribution: fragments of relations
  - Grants locality of access
  - Load balancing
  - Reduced processing costs for operations performed only on part of the data
  - Parallel processing

Fragmentation /2

- Approach:
  - Column- or tuple-wise decomposition (vertical/horizontal)
  - Described using relational algebra expressions (queries)
  - Important rules/requirements
    - Completeness
    - Reconstructability
    - Disjointness

Example Database

MEMBER

MNo	MName	Position
M1	Ian Curtis	SW Developer
M2	Levon Helm	Analyst
M3	Tom Verlaine	SW Developer
M4	Moe Tucker	Manager
M5	David Berman	HW-Developer

PROJECT

PNr	PName	Budget	Loc
P1	DB Development	200.000	MD
P2	Hardware Dev.	150.000	M
P3	Web-Design	100.000	MD
P4	Customizing	250.000	B

ASSIGNMENT

MNr	PNr	Capacity
M1	P1	5
M2	P4	4
M2	P1	6
M3	P4	3
M4	P1	4
M4	P3	5
M5	P2	7

SALARY

Position	YSalary
SW Developer	60.000
HW-Developer	55.000
Analyst	65.000
Manager	90.000

Primary Horizontal Fragmentation

- "‘Tupel-wise"’ decomposition of a global relation R into n fragments R_i
- Defined by n selection predicates P_i on attributes from R

$$R_i := \sigma_{P_i}(R) \quad (1 \leq i \leq n)$$

- P_i: *fragmentation predicates*
- Completeness: each tuple from R must be assigned to a fragment
- Disjointness: decomposition into disjoint fragments $R_i \cap R_j = \emptyset \quad (1 \leq i, j \leq n, i \neq j)$,
- Reconstructability: $R = \bigcup_{1 \leq i \leq n} R_i$

Primary Horizontal Fragmentation /2

- Example: fragmentation of PROJECT by predicate on location attribute "'Loc"'

$$\text{PROJECT}_1 = \sigma_{\text{Loc}='\text{M}'}(\text{PROJECT})$$
$$\text{PROJECT}_2 = \sigma_{\text{Loc}='\text{B}'}(\text{PROJECT})$$
$$\text{PROJECT}_3 = \sigma_{\text{Loc}='\text{MD}'}(\text{PROJECT})$$

PROJECT_1

PNr	PName	Budget	Loc
P2	Hardware Dev.	150.000	M

PROJECT_2

PNr	PName	Budget	Loc
P4	Customizing	250.000	B

PROJECT_3

PNr	PName	Budget	Loc
P1	DB Development	200.000	MD
P3	Web-Design	100.000	MD

Derived Horizontal Fragmentation

- Fragmentation definition of relation S derived from existing horizontal fragmentation of relation R
- Using foreign key relationships
- Relation R with n fragments R_i
- Decomposition of depending relation S

$$S_i = S \ltimes R_i = S \ltimes \sigma_{P_i}(R) = \pi_{S.*}(S \bowtie \sigma_{P_i}(R))$$

- P_i defined only on R
- Reconstructability: see above
- Disjointness: implied by disjointness of R-fragments
- Completeness: granted for lossless semi-join (no null-values for foreign key in S)

Derived Horizontal Fragmentation /2

- Fragmentation of relation ASSIGNMENT derived from fragmentation of PROJECT relation

$$\text{ASSIGNMENT}_1 = \text{ASSIGNMENT} \ltimes \text{PROJECT}_1$$
$$\text{ASSIGNMENT}_2 = \text{ASSIGNMENT} \ltimes \text{PROJECT}_2$$
$$\text{ASSIGNMENT}_3 = \text{ASSIGNMENT} \ltimes \text{PROJECT}_3$$

ASSIGNMENT$_1$

MNr	PNr	Capacity
M5	P2	7

ASSIGNMENT$_2$

MNr	PNr	Capacity
M2	P4	4
M3	P4	3

ASSIGNMENT$_3$

MNr	PNr	Capacity
M1	P1	5
M2	P1	6
M4	P1	4
M4	P3	5

Vertical Fragmentation

- Comlumn-wise decomposition of a relation using relational projection
- Completeness: each attribute must be in at least one fragment
- Reconstructability: through natural join
  $\rightsquigarrow$ primary key of global relation must be in each fragment

$$R_i := \pi_{K,A_i,\ldots,A_j}(R)$$
$$R = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$$

- Limited disjointness

Vertical Fragmentation /2

- Fragmentation of PROJECT-Relation regarding Budget and project name / location

$$\text{PROJECT}_1 = \pi_{\text{PNr, PName, Loc}}(\text{PROJECT})$$
$$\text{PROJECT}_2 = \pi_{\text{PNr, Budget}}(\text{PROJECT})$$

PROJECT_1

PNr	PName	Loc
P1	DB Development	MD
P2	Hardware Dev.	M
P3	Web-Design	MD
P4	Customizing	B

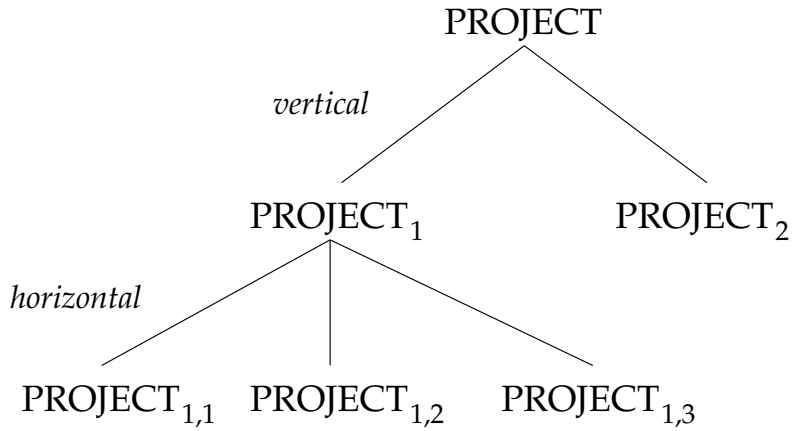
PROJECT_2

PNr	Budget
P1	200.000
P2	150.000
P3	100.000
P4	250.000

Hybrid Fragmentation

- Fragment of a relation $\rightarrow$ is relation itself
- Can be subject of further fragmentation
- Also possible: combination of horizontal and vertical fragmentation

$\text{PROJECT}_1 = \pi_{\text{PNr, PName, Loc}}(\text{PROJECT})$
$\text{PROJECT}_2 = \pi_{\text{PNr, Budget}}(\text{PROJECT})$
$\text{PROJECT}_{1,1} = \sigma_{\text{Loc='M'}}(\text{PROJECT}_1)$
$\text{PROJECT}_{1,2} = \sigma_{\text{Loc='B'}}(\text{PROJECT}_1)$
$\text{PROJECT}_{1,3} = \sigma_{\text{Loc='MD'}}(\text{PROJECT}_1)$



Fragmentation transparency

- Decomposition of a relation is for user/application not visble
- Only view on global relation
- Requires mapping of DB operations to fragments by DDBMS
- Example
  - Transparent:

```
select * from Project where PNr=P1
```

  - Without transparency:

```
select * from Project1 where PNr=P1
if not-found then
   select * from Project2 where PNr=P1
if not-found then
   select * from Project3 where PNr=P1
```

Fragmentation transparency /2

- Example (continued)
  - Transparent:

```
update Project set Ort='B' where PNr=P3
```

  - Without transparency:

```
select PNr, PName, Budget
   into :PNr, :PName, :Budget
   from Project3 where PNr=P3

insert into Project2
   values (:PNr, :PName, :Budget, 'B')

delete from Project3 where PNr=P3
```

Computation of an optimal Fragmentation

- In huge systems with many relations/nodes: intuitive decomposition often too complex/not possible
- In this case: systematic process based on access characteristics
  - Kind of access (read/write)
  - Frequency
  - Relations / attributes
  - Predicates in queries
  - Transfer volume and times

Optimal horizontal Fragmentation

- Based on [Özsu/Valduriez 99] and [Dadam 96]
- Given: relation $R(A_1, \ldots, A_n)$, operator $\theta \in \{<, \leq, >, \geq, =, \neq\}$, Domain $\mathrm{dom}(A_i)$
- Definition: **simple predicate** p_i of the form $A_j \theta \, \mathrm{const}$ with $\mathrm{const} \in \mathrm{dom}(A_j)$
  - Defines possible binary fragmentation of R
  - Example:

    $\text{PROJECT}_1 = \sigma_{\text{Budget}>150.000}(\text{PROJECT})$
    $\text{PROJECT}_2 = \sigma_{\text{Budget}\leq 150.000}(\text{PROJECT})$
- Definition: **Minterm** m is conjunction of simple predicates as $m = p_1^* \wedge p_2^* \wedge \cdots \wedge p_j^*$
  with $p_i^* = p_i$ oder $p_i^* = \neg p_i$

Optimal horizontal Fragmentation /2

- Definition: Set $M_n(P)$ of all n-ary Minterms for the set P of simple predicates:

$$M_n(P) = \{m \mid m = \bigwedge_{i=1}^{n} p_i^*, p_i \in P\}$$

  - Defines *complete* fragmentation of R without redundancies
    - ★ $R = \bigcup_{m \in M_n(P)} \sigma_m(R)$
    - ★ $\sigma_{m_i} \cap \sigma_{m_j} = \emptyset, \forall m_i, m_j \in M_n(P), m_i \neq m_j$

Optimal horizontal Fragmentation /3

- Completeness and no redundancy not sufficient:
  - $P = \{$ Budget < 100.000, Budget > 200.000, Ort $=$ 'MD', Ort $=$ 'B' $\}$
  - Minterm $p_1 \wedge p_2 \wedge p_3 \wedge p_4$ not satisfiable; but $\neg p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge \neg p_4$
- Identification of *practically relevant* Minterms $M(P)$
  1. $M(P) := M_n(P)$
  2. Remove irrelevant Minterms from $M(P)$

Elimination of irrelevant Minterms

1. Elimination of unsatisfiable Minterms
   If two terms p_i^* and p_j^* in one $m \in M(P)$ contradict, m is not satisfiable and can be removed from $M(P)$.
2. Elimination of dependent predicates
   If a p_i^* from $m \in M(P)$ implies another term p_j^* (e.g. functional dependency, overlapping domains), p_j^* can be removed from m.
3. Relevance of a fragmentation
   - Minterms m_i and m_j, m_i contains p_i, m_j contains $\neg p_i$
   - Access statistics: $\mathrm{acc}(m)$
     (e.g. derived from query log)
   - Fragment size: $\mathrm{card}(f)$
     (derived from data distribution statistics)
   - p_i is *relevant*, if $\frac{\mathrm{acc}(m_i)}{\mathrm{card}(f_i)} \neq \frac{\mathrm{acc}(m_j)}{\mathrm{card}(f_j)}$

Algorithm HORIZFRAGMENT

- Identification of a complete, non-redundant and minimal horizontal fragmentation of a relation R for a given set of predicates P
- Input:
  - P: set of predicates over R
- (Intermediate) Results:
  - $M(P)$: set of relevant Minterms
  - $F(P)$: set of Minterm-fragments from R

$$R(m) := \sigma_m(R) \text{ with } m \in M(P)$$

Algorithm HORIZFRAGMENT

forall $p \in P$ **do**
    $Q' := Q \cup \{p\}$
    compute $M(Q')$ and $F(Q')$
    compare $F(Q')$ with $F(Q)$
    if $F(Q')$ *significant improvement* over $F(Q)$ **then**
        $Q := Q'$
        forall $q \in Q \setminus \{p\}$ **do** /* *unnecessary Fragmentation?* */
            $Q' := Q \setminus \{q\}$
            compute $M(Q')$ and $F(Q')$
            compare $F(Q')$ with $F(Q)$
            if $F(Q)$ no *significant improvement* over $F(Q')$ **then**
                $Q := Q'$ /* d.h., *remove q from Q* */
            end
        end
    end

Allocation and Replication

- Allocation
  - Assignment of relations or fragments to physical storage location
  - *Non-redundant*: fragments are stored in only one place $\rightsquigarrow$ partitioned DB
  - *Redundant*: fragments can be stored more than once $\rightsquigarrow$ replicated DB
- Replication
  - Storage of redundant copies of fragments or relations
  - *Full*: Each global relation stored on every node (no distribution design, no distributed query processing, high costs for storage and updates)
  - *Partial*: Fragments are stored on selected nodes

Allocation and Replication /2

- Aspects of allocation
  - Efficiency:
    - ★ Minimization of costs for remote accesses
    - ★ Avoidance of bottlenecks
  - Data security:
    - ★ Selection of nodes depending on their "'reliability"'

Identification of an optimal Allocation

- Cost model for non-redundant allocation [Dadam 96]
- Goal:
  Minimize storage and transfer costs $\sum_{Storage} + \sum_{Transfer}$ for K fragments and L nodes
- Storage costs:

$$\sum_{Storage} = \sum_{p,i} S_p D_{pi} SC_i$$

  - S_p: Size of fragment p in data units
  - SC_i: StorageCosts per data unit on node i
  - D_{pi}: Distribution of fragment with $D_{pi} = 1$ if p stored on node i, 0 otherwise

Identification of an optimal Allocation /2

- Transfer costs:

$$\sum_{Transfer} = \sum_{i,t,p,j} F_{it} O_{tp} D_{pj} TC_{ij} + \sum_{i,t,p,j} F_{it} R_{tp} D_{pj} TC_{ji}$$

  - F_{it}: Frequency of operation of type t on node i
  - O_{tp}: Size of operation t for fragment p in data units (e.g. size of query string)
  - TC_{ij}: TransferCosts from node i to j in data units
  - R_{tp}: Size o the result of one operation of type t on fragment p

Identification of an optimal Allocation /3

- Additional constraints:

$$\sum_i D_{pi} = 1 \quad \text{for} \quad p = 1, \ldots, K$$

$$\sum_p S_p D_{pi} \leq M_i \quad \text{for} \quad p = i, \ldots, L$$

  where M_i is max. storage capacity on node i
- Integer optimization problem
- Often heuristic solution possible:
  - Identify relevant candidate distributions
  - Compute costs and compare candidates

Identification of an optimal Allocation /4

- Cost model for redundant replication
- Additional constraints slightly modified:

$$\sum_i D_{pi} \geq 1 \quad \text{for} \quad p = 1, \ldots, K$$

$$\sum_p S_p D_{pi} \leq M_i \quad \text{ffr} \quad p = i, \ldots, L$$

Identification of an optimal Allocation /5

- Transfer costs
  - Read operations on p send from node i to j with minimal TC_{ij} and $D_{pj} = 1$
  - Update operations on p send to all nodes j with $D_{pj} = 1$
  - Φ_t: of an operation $\sum$ (in case of update) or $\min$ (in case of read operation)

$$\sum\nolimits_T ransfer = \sum_{i,t,p} F_{it} \Phi_t \sum_{j:D_{pj}=1} (O_{tp} TC_{ij} + R_{tp} TC_{ji})$$

Evaluation of Approaches

- Model considering broad spectrum of applications
- Exact computation possible
- *But*:
  - High computation efforts (optimization problem)
  - Exact input values are hard to obtain

Part II

Distributed Database Systems

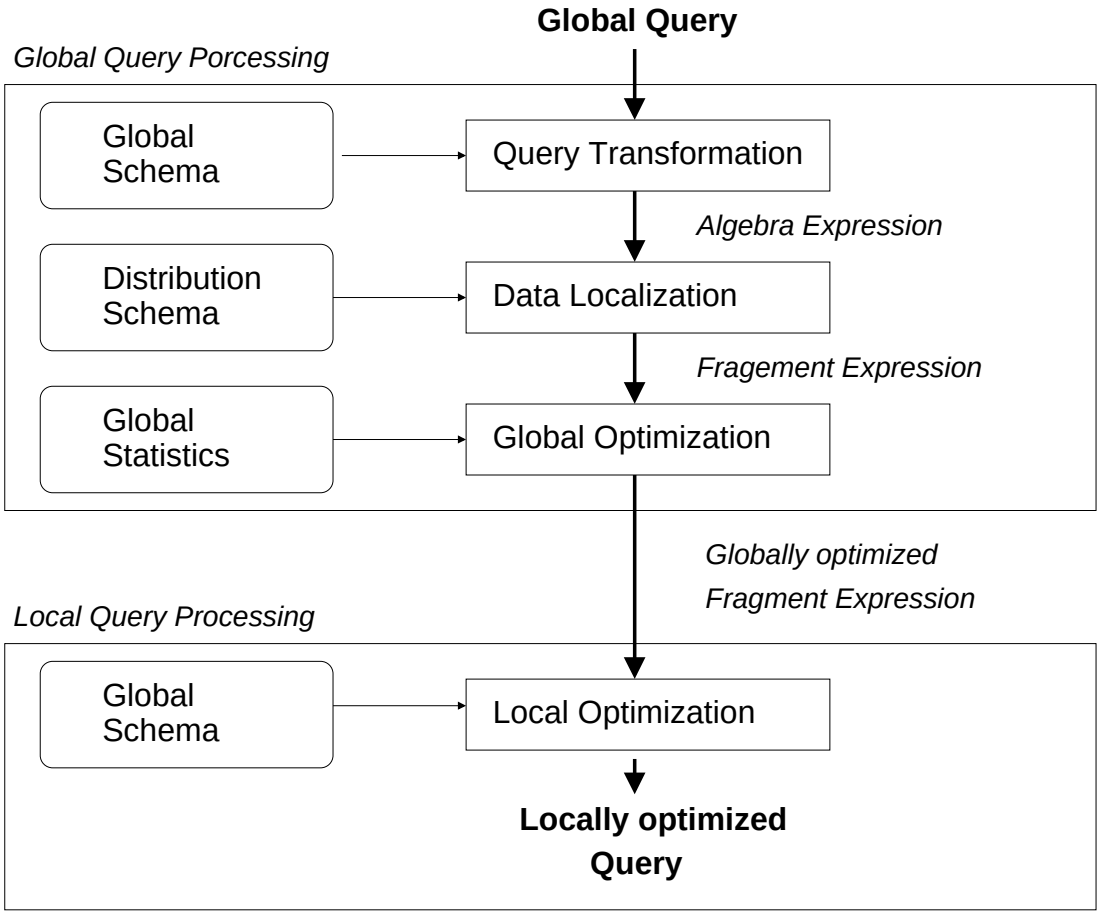
Overview

- Foundations of DDBS
- Catalog Management
- DDBS Design: Fragmentation
- Allocation and Replication
- Overview
- Data Localization
- Join Processing
- Global Optimization

Overview

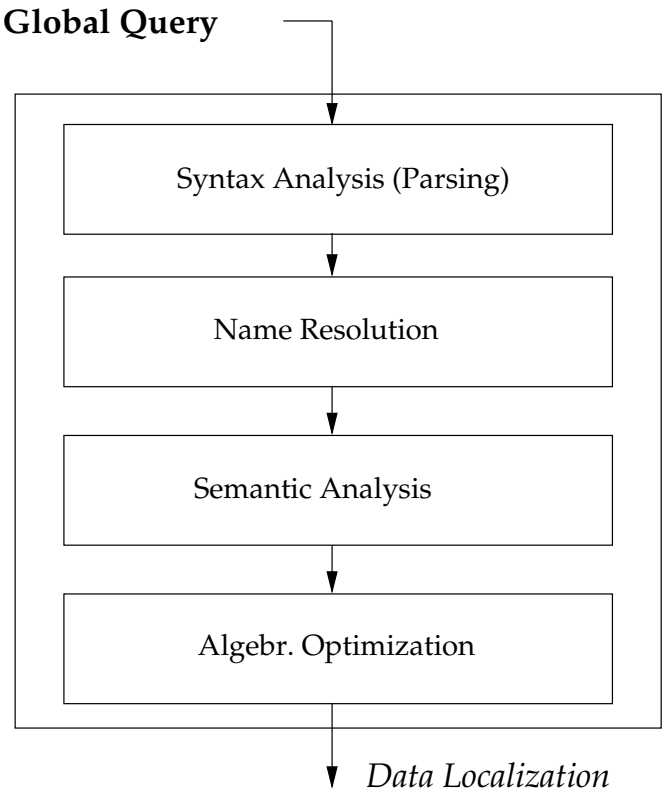
- Goal of query processing: creation of an efficient as possible query plans from a declarative query
  - Transformation to internal format (Calculus $\rightarrow$ Algebra)
  - Selection of access paths (indexes) and algorithms (e.g. Merge-Join vs. Nested-Loops-Join)
  - Cost-based selection of best possible plan
- In Distributed DBS:
  - *User view:* no difference $\rightarrow$ queries are formulated on global schema/external views
  - *Query processing:*
    - ⋆ Consideration of physical distribution of data
    - ⋆ Consideration of communication costs

Phases of Query Processing



Query Transformation

Query Transformation



Translation to Relational Algebra

```
select A1, ..., Am
from R1, R2, ..., Rn
where F
```

Initial relational algebra expression:

$$\pi_{A_1,\ldots,A_m}(\sigma_F(r(R_1) \times r(R_2) \times r(R_3) \times \cdots \times r(R_n)))$$

Improve algebra expression:

- *Detect joins* to replace Cartesian products
- *Resolution of subqueries* (**`not exists`**-queries to set difference)
- Consider SQL-operations not in relational algebra: (**`group by`**, **`order by`**, arithmetics, **`...`**)

Normalization

- Transform query to unified canonical format to s implify following optimization steps
- Special importance: selection and join conditions (from **`where`**-clause)
  - *Conjunctive normal form* vs. *disjunctive normal form*
  - Conjunktive normal form (CNF) for basic predicates p_{ij}:

$$(p_{11} \vee p_{12} \vee \cdots \vee p_{1n}) \wedge \cdots \wedge (p_{m1} \vee p_{m2} \vee \cdots \vee p_{mn})$$

  - Disjunctive normal form (DNF):

$$(p_{11} \wedge p_{12} \wedge \cdots \wedge p_{1n}) \vee \cdots \vee (p_{m1} \wedge p_{m2} \wedge \cdots \wedge p_{mn})$$

  - Transformation according to equivalence rules for logical operations

Normalization /2

- Equivalence rules
  - $p_1 \wedge p_2 \longleftrightarrow p_2 \wedge p_1$ und $p_1 \vee p_2 \longleftrightarrow p_2 \vee p_1$
  - $p_1 \wedge (p_2 \wedge p_3) \longleftrightarrow (p_1 \wedge p_2) \wedge p_3$ und $p_1 \wedge (p_2 \vee p_3) \longleftrightarrow (p_1 \vee p_2) \vee p_3$
  - $p_1 \wedge (p_2 \vee p_3) \longleftrightarrow (p_1 \wedge p2) \vee (p_1 \wedge p_3)$ und $p_1 \vee (p_2 \wedge p_3) \longleftrightarrow (p_1 \vee p2) \wedge (p_1 \vee p_3)$
  - $\neg(p_1 \wedge p_2) \longleftrightarrow \neg p_1 \vee \neg p_2$ und $\neg(p_1 \vee p_2) \longleftrightarrow \neg p_1 \wedge \neg p_2$
  - $\neg(\neg p_1) \longleftrightarrow p_1$

Normalization: Example

- Query:

```
select * from Project P, Assignment A
where P.PNr = A.PNr and
      Budget > 100.000 and
      (Loc = 'MD' or Loc = 'B')
```

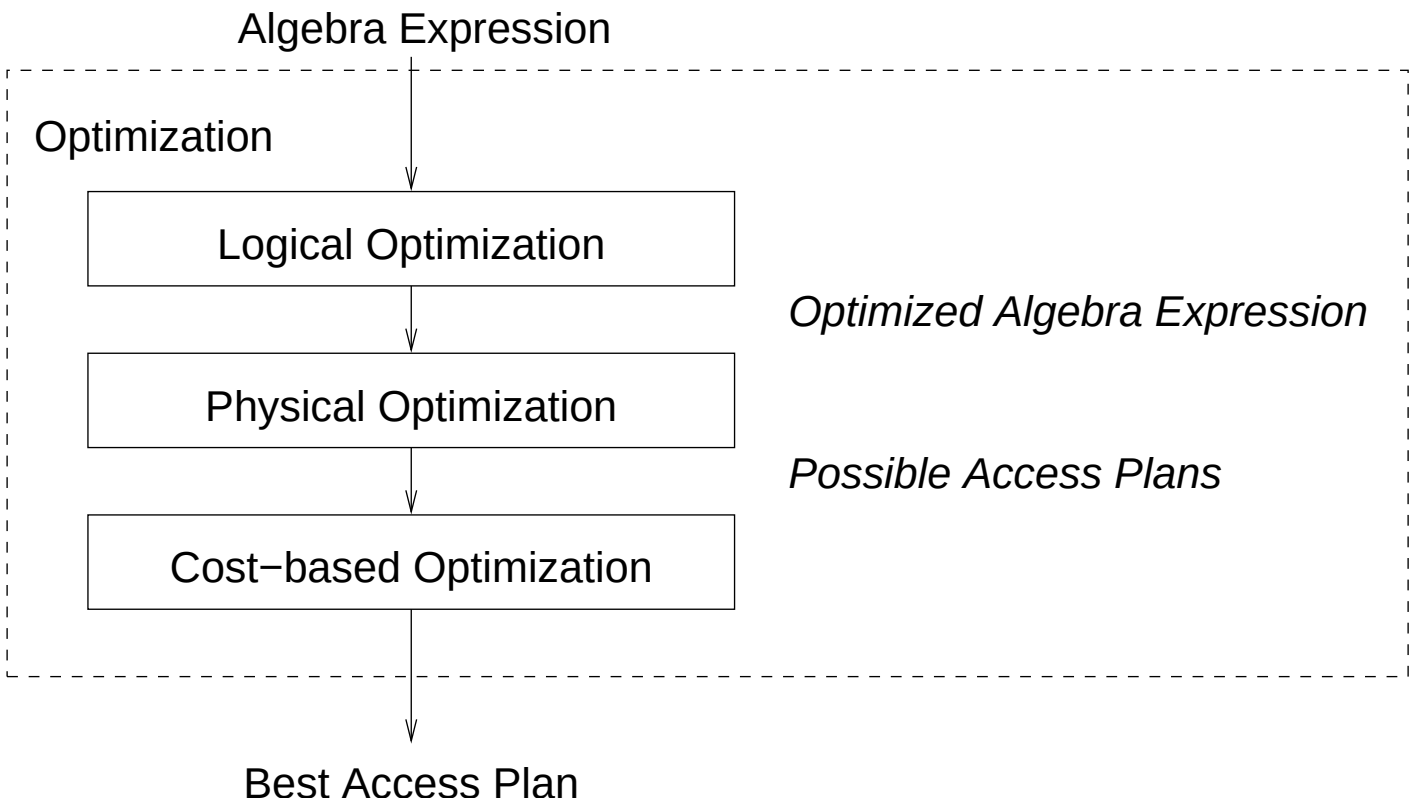
- Selection condition in CNF:

$$\text{P.PNr} = \text{A.PNr} \wedge \text{Budget} > 100.000 \wedge (\text{Loc} = \text{'MD'} \vee \text{Loc} = \text{'B'})$$

- Selection condition in DNF:

$$(\text{P.PNr} = \text{A.PNr} \wedge \text{Budget} > 100.000 \wedge \text{Loc} = \text{'MD'}) \vee (\text{P.PNr} = \text{A.PNr} \wedge \text{Budget} > 100.000 \wedge \text{Loc} = \text{'B'})$$

Phases of Optimization



Algebraic Optimization

- Term replacement based on semantic equivalences
- Directed replacement rules to *improve* processing of expression
- Heuristic approach:
  - Move operation to get smaller intermediate results
  - Indentify and remove redundancies
- Result: improved algebraic express $\Rightarrow$ operator tree $\Rightarrow$ initial query plan

Algebraic Rules /1

- Operators σ and $\bowtie$ commute, if selection attribute from one relation:

$$\sigma_F(r_1 \bowtie r_2) \longleftrightarrow \sigma_F(r_1) \bowtie r_2 \qquad \text{falls } attr(F) \subseteq R_1$$

- If selection condition can be split, such that $F = F_1 \wedge F_2$ contain predicates on attributes in only one relation, respectively:

$$\sigma_F(r_1 \bowtie r_2) \longleftrightarrow \sigma_{F_1}(r_1) \bowtie \sigma_{F_2}(r_2)$$

$$\text{if } \; attr(F_1) \subseteq R_1 \; \text{ and } \; attr(F_2) \subseteq R_2$$

- Always: decompose to F_1 with attributes from R_1, if F_2 contains attributes from R_1 and R_2:

$$\sigma_F(r_1 \bowtie r_2) \longleftrightarrow \sigma_{F_2}(\sigma_{F_1}(r_1) \bowtie r_2) \qquad \text{if } attr(F_1) \subseteq R_1$$

Algebraic Rules /2

- Combination of conditions of σ is identical to logical conjunction $\Rightarrow$ operations can change their order

$$\sigma_{F_1}(\sigma_{F_2}(r_1)) \longleftrightarrow \sigma_{F_1 \wedge F_2}(r_1) \longleftrightarrow \sigma_{F_2}(\sigma_{F_1}(r_1))$$

(uses commutativity of logic AND)

Algebraic Rules /3

- Operator $\bowtie$ is commutative:

$$r_1 \bowtie r_2 \longleftrightarrow r_2 \bowtie r_1$$

- Operator $\bowtie$ is associative:

$$(r_1 \bowtie r_2) \bowtie r_3 \longleftrightarrow r_1 \bowtie (r_2 \bowtie r_3)$$

- Domination of sequence of π operators:

$$\pi_X(\pi_Y(r_1)) \longleftrightarrow \pi_X(r_1)$$

- π and σ are commutative in some cases:

$$\sigma_F(\pi_X(r_1)) \longleftrightarrow \pi_X(\sigma_F(r_1))$$

$$\text{if } \mathit{attr}(F) \subseteq X$$

$$\pi_{X_1}(\sigma_F(\pi_{X_1X_2}(r_1))) \longleftrightarrow \pi_{X_1}(\sigma_F(r_1))$$

$$\text{if } \mathit{attr}(F) \supseteq X_2$$

Algebraic Rules /4

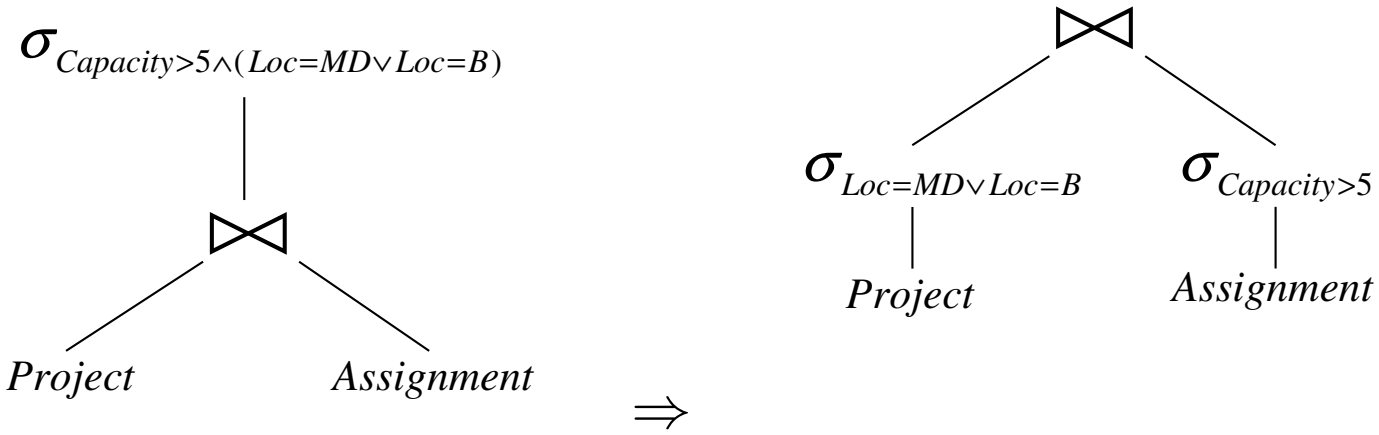
- Commutation of σ and $\cup$:

$$\sigma_F(r_1 \cup r_2) \longleftrightarrow \sigma_F(r_1) \cup \sigma_F(r_2)$$

- Commutation of σ and with other set operation $-$ and $\cap$
- Commutation of π and $\bowtie$ partially possible: join attributes must be kept and later removed (nevertheless decreases intermediate result size)
- Commutation of π und $\cup$
- Distributivity for set operations
- Idempotent expressions, e.g. $r_1 \bowtie r_1 = r_1$ and $r_1 \cup r_1 = r1$
- Operations with empty relations, e.g. $r_1 \cup \emptyset = r_1$
- Commutativity of set operations
- ...

Algebraic Optimization: Example

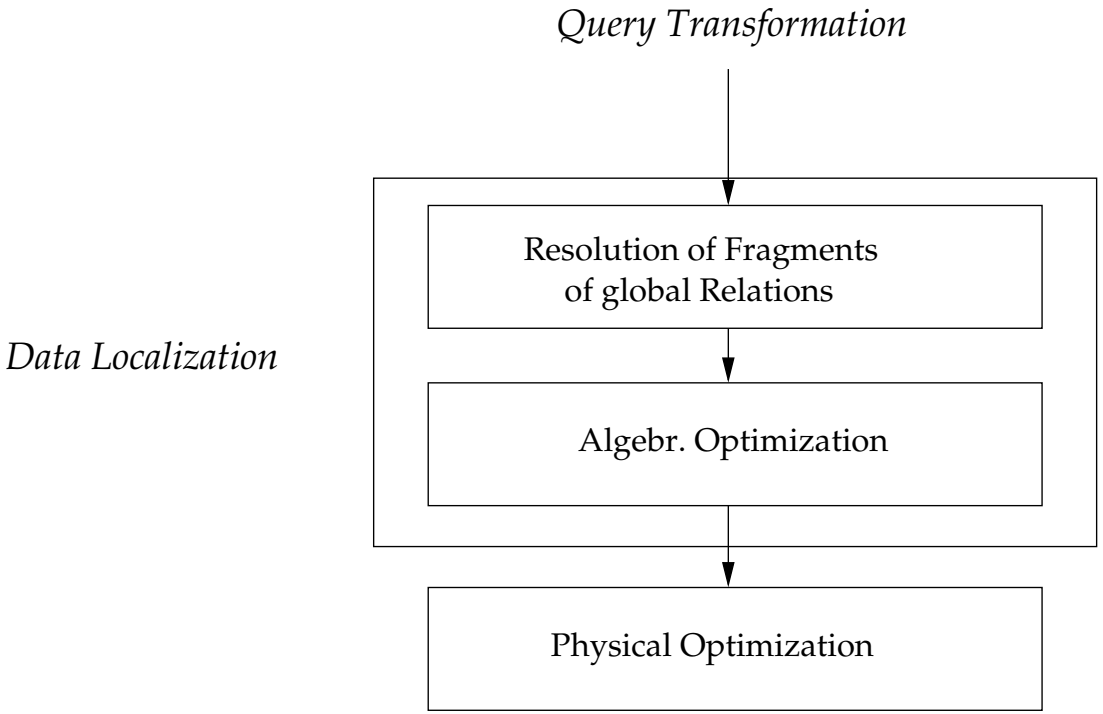
```
select * from Procekt P, Assignment A
where P.PNr = A.PNr and
      Capacity > 5 and
      (Loc = 'MD' or Loc = 'B')
```



Data Localization

- Task: create fragment queries based on data distribution
  - Replace global relation with fragments
  - Insert reconstruction expression using fragments of global relation

Data Localization Phase



Data Localization: Example I

- Schema:

  $\text{PROJ}_1 = \sigma_{\text{Budget} \leq 150.000}(\text{PROJEKT})$
  $\text{PROJ}_2 = \sigma_{150.000 < \text{Budget} \leq 200.000}(\text{PROJECT})$
  $\text{PROJ}_3 = \sigma_{\text{Budget} > 200.000}(\text{PROJECT})$

  $\text{PROJECT} = \text{PROJ}_1 \cup \text{PROJ}_2 \cup \text{PROJ}_3$

- Query:

  $\sigma_{\text{Loc}=\text{'MD'} \wedge \text{Budget} \leq 100.000}(\text{PROJECT})$

  $\Longrightarrow$

  $\sigma_{\text{Loc}=\text{'MD'} \wedge \text{Budget} \leq 100.000}(\text{PROJ}_1 \cup \text{PROJ}_2 \cup \text{PROJ}_3)$

Data Localization /2

- Requirement: further simplification of query
- Goal: eliminate queries on fragments not used in query
- Example: pushing down σ to fragments

  $\sigma_{\text{Loc}=\text{'MD'} \wedge \text{Budget} \leq 100.000}(\text{PROJ}_1 \cup \text{PROJ}_2 \cup \text{PROJ}_3)$

  because of:

  $\sigma_{\text{Budget} \leq 100.000}(\text{PROJ}_2) = \emptyset$, $\sigma_{\text{Budget} \leq 100.000}(\text{PROJ}_3) = \emptyset$

  $\Longrightarrow$

  $\sigma_{\text{Loc}=\text{'MD'}}(\sigma_{\text{Budget} \leq 100.000}(\text{PROJ}_1))$

Data Localization /3

- For horizontal fragmentation
  - Also possible simplification of join processing
  - Push down join if fragmentation on join attribute

Data Localization: Example II

- Schema:

  $M_1 = \sigma_{\text{MNr}<\text{'M3'}}(\text{MEMBER})$
  $M_2 = \sigma_{\text{'M3'}\leq\text{MNr}<\text{'M5'}}(\text{MEMBER})$
  $M_3 = \sigma_{\text{MNr}\geq\text{'M5'}}(\text{MEMBER})$

  $Z_1 = \sigma_{\text{MNr}<\text{'M3'}}(\text{ASSIGNMENT})$
  $Z_2 = \sigma_{\text{MNr}\geq\text{'M3'}}(\text{ASSIGNMENT})$

- Query: $\text{ASSIGNMENT} \bowtie \text{MEMBER}$

  $\Longrightarrow$
  $(M_1 \cup M_2 \cup M_3) \bowtie (Z_1 \cup Z_2)$
  $\Longrightarrow$
  $(M_1 \bowtie Z_1) \cup (M_2 \bowtie Z_2) \cup (M_3 \bowtie Z_2)$

Data Localization /4

- Vertical fragmentation: reduction by pushing down projections
- Example:

  $\text{PROJ}_1 = \pi_{\text{PNr,PName,Loc}}(\text{PROJECT})$
  $\text{PROJ}_2 = \pi_{\text{PNr,Budget}}(\text{PROJECT})$

  $\text{PROJECT} = \text{PROJ}_1 \bowtie \text{PROJ}_2$
- Query: $\pi_{\text{PName}}(\text{PROJECT})$

  $\Longrightarrow$
  $\pi_{\text{PName}}(\text{PROJ}_1 \bowtie \text{PROJ}_2)$
  $\Longrightarrow$
  $\pi_{\text{PName}}(\text{PROJ}_1)$

Qualified Relations

- Descriptive information to support algebraic optimization
- Annotation of fragments and intermediate results with content condition (combination of predicates that are satisfied here)
- Estimation of size of relation
- If $r' = Q(r)$, then r' *inherits* condition from r, plus additional predicates from Q
- Qualification condition q_R: $[R : q_R]$
- Extended relational algebra: $\sigma_F[R : q_R]$

Extended Relational Algebra

$$
\begin{array}{lll}
(1) & E := & \sigma_F[R : q_R] \rightarrow [E : F \wedge q_R] \\
(2) & E := & \pi_A[R : q_R] \rightarrow [E : q_R] \\
(3) & E := & [R : q_R] \times [S : q_S] \rightarrow [E : q_R \wedge q_S] \\
(4) & E := & [R : q_R] - [S : q_S] \rightarrow [E : q_R] \\
(5) & E := & [R : q_R] \cup [S : q_S] \rightarrow [E : q_R \vee q_S] \\
(6) & E := & [R : q_R] \bowtie_F [S : q_S] \rightarrow [E : q_R \wedge q_S \wedge F]
\end{array}
$$

Extended Relational Algebra /2

- Usage of rules for *description* – no processing
- Example: $\sigma_{100.000 \leq \text{Budget} \leq 200.000}(\text{PROJECT})$

$$
\begin{aligned}
E_1 &= \sigma_{100.000 \leq \text{Budget} \leq 200.000}[\text{PROJ}_1 : \text{Budget} \leq 150.000] \\
&\rightsquigarrow [E_1 : (100.000 \leq \text{Budget} \leq 200.000) \wedge (\text{Budget} \leq 150.000)] \\
&\rightsquigarrow [E_1 : 100.000 \leq \text{Budget} \leq 150.000] \\
E_2 &= \sigma_{1000 \leq \text{Budget} \leq 200.000}[\text{PROJ}_2 : 150.000 < \text{Budget} \leq 200.000] \\
&\rightsquigarrow [E_2 : (100.000 \leq \text{Budget} \leq 200.000) \wedge \\
&\quad (150.000 < \text{Budget} \leq 200.000)] \\
&\rightsquigarrow [E_2 : 150.000 < \text{Budget} \leq 200.000] \\
E_3 &= \sigma_{100.000 \leq \text{Budget} \leq 200.000}[\text{PROJ}_3 : \text{Budget} > 200.000] \\
&\rightsquigarrow [E_3 : (100.000 \leq \text{Budget} \leq 200.000) \wedge (\text{Budget} > 200.000)] \\
&\rightsquigarrow E_3 = \emptyset
\end{aligned}
$$

Join Processing

- Join operations:
  - Common task in relational DBS, very expensive ($\leq O(n^2)$)
  - In distributed DBS: join of nodes stored on different nodes
- Simple strategy: process join on one node
  - *Ship whole*: transfer the full relation beforehand
  - *Fetch as needed*: request tuples for join one at a time

"‘Fetch as needed "’ vs. "‘Ship whole"’ /1

R

A	B
3	7
1	1
4	6
7	7
4	5
6	2
5	7

S

B	C	D
9	8	8
1	5	1
9	4	2
4	3	3
4	2	6
5	7	8

R ⋈ S

A	B	C	D
1	1	5	1
4	5	7	8

Strategy	#Messages	#Values
SW at R-node	2	18
SW at S-node	2	14
SW at 3. node	4	32
FAN at S-node	$6 * 2 = 12$	$6 + 2 * 2 = 10$
FAN at R-node	$7 * 2 = 14$	$7 + 2 * 3 = 13$

"‘Fetch as needed "’ vs. "‘Ship whole"’ /2

- Comparison:
  - "‘Fetch as needed"’ with higher number of messages, useful for small left hand-side relation (e.g. restricted by previous selection)
  - "‘Ship whole"’ with higher data volume, useful for smaller right hand-side (transferred) relation
- Specific algorithms for both:
  - Nested-Loop Join
  - Sort-Merge Join
  - *Semi-Join*
  - *Bit Vector-Join*

Nested-Loop Join

Nested loop over all tuples $t_1 \in r$ and all $t_2 \in s$ for operation $r \bowtie s$

$r \bowtie_\varphi s$:

```
for each t_r ∈ r do
begin
    for each t_s ∈ s do
    begin
        if φ(t_r, t_s) then put(t_r · t_s) endif
    end
end
```

Sort Merge-Join

$X := R \cap S$; if not yet sorted, first sort r and s on join attributes X

1. $t_r(X) < t_s(X)$, read next $t_r \in r$
2. $t_r(X) > t_s(X)$, read next $t_s \in s$
3. $t_r(X) = t_s(X)$, join t_r with t_s and all subsequent tuples to t_s equal regarding X with t_s
4. Repeat for the first $t'_s \in s$ with $t'_s(X) \neq t_s(X)$ starting with original t_s and following t'_r of t_r until $t_r(X) = t'_r(X)$

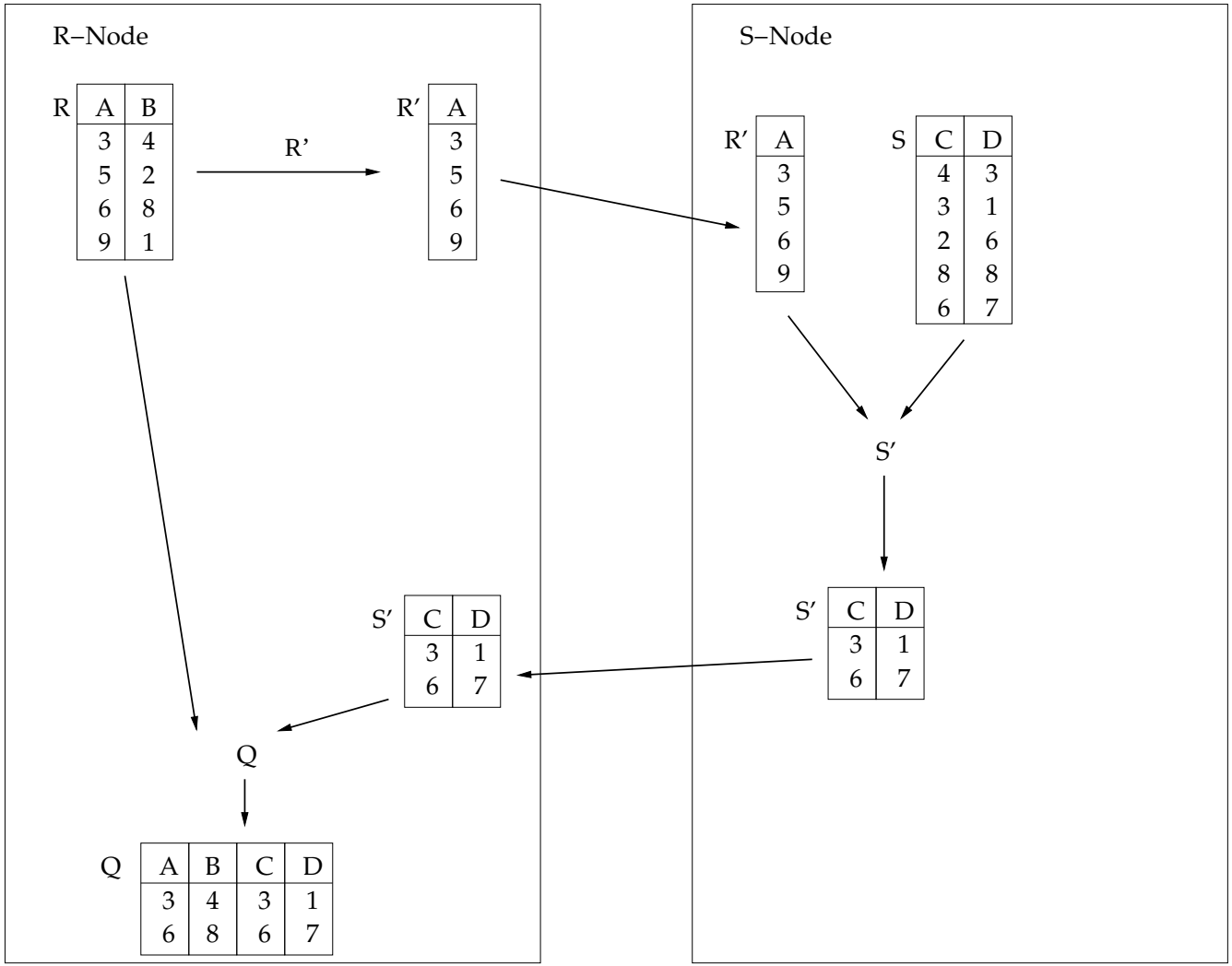
Sort Merge-Join: Costs

- Worst case: all tuples with identical X-values: $O(n_r * n_s)$
- X keys of R or S: $O(n_r \log n_r + n_s \log n_s)$
- If relations are already sorted (e.g. index on join attributes, often the case): $O(n_r + n_s)$

Semi-Join

- Idea: request join partner tuples in one step to minimize message overhead (combines advantages of SW and FAN)
- Based on: $r \bowtie s = r \bowtie (s \ltimes r) = r \bowtie (s \bowtie \pi_A(r))$ (A is set of join attributes)
- Procedure:
  1. Node N_r: computation of $\pi_A(r)$ and transfer to N_s
  2. Node N_s: computation of $s' = s \bowtie \pi_A(r) = s \ltimes r$ and transfer to N_r
  3. Node N_r: computation of $r \bowtie s' = r \bowtie s$

Semi-Join: Example



Bit Vector-Join

- Bit Vector or Hash Filter-Join
- Idea: minimize request size (semi-join) by mapping join attribute values to bit vector $B[1 \ldots n]$
- Mapping:
  - Hash function h maps values to buckets $1 \ldots n$
  - If value exists in bucket according bit is set to 1

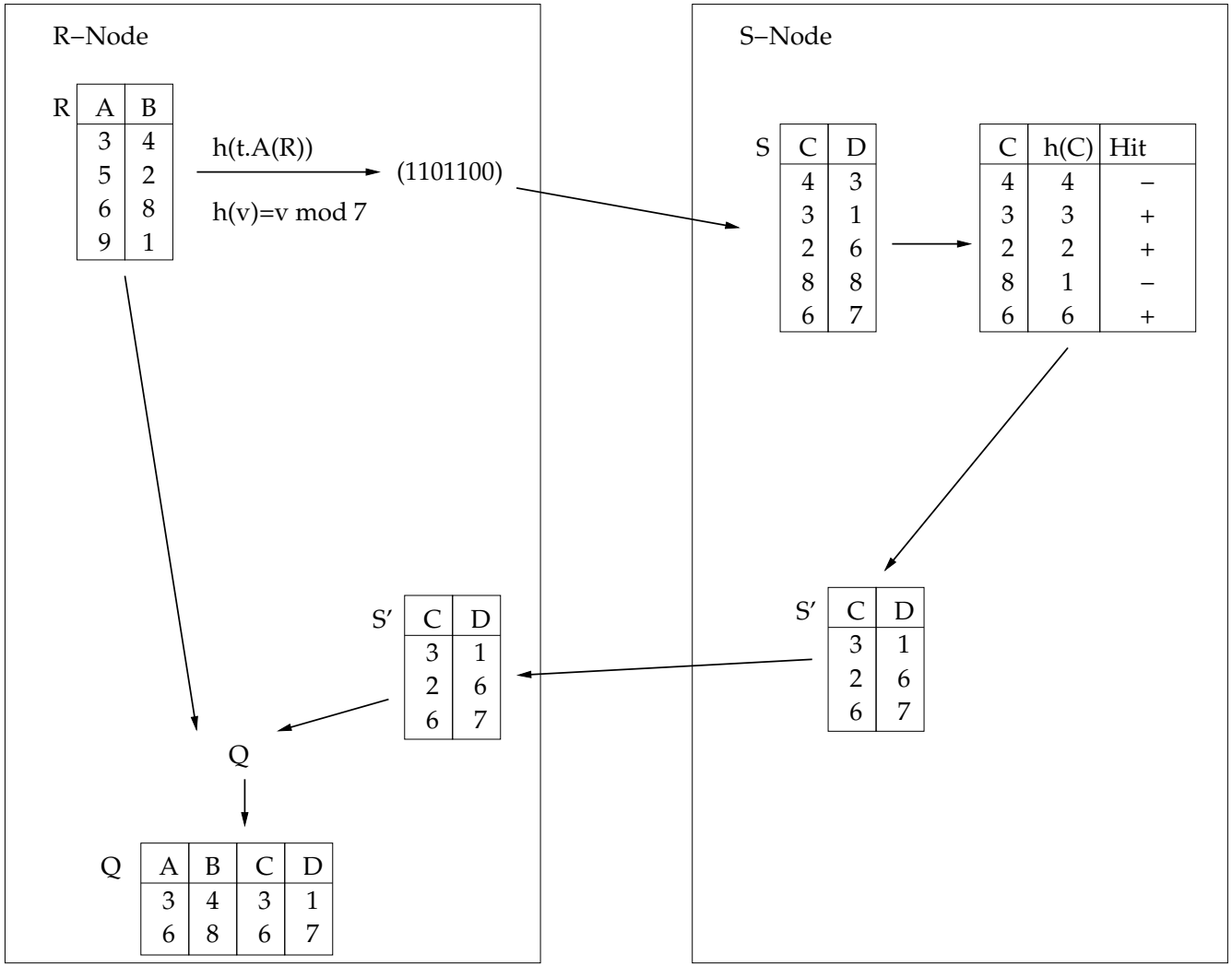
Bit Vector-Join /2

- Procedure:
  1. Node N_r: for each value v in $\pi_A(r)$ set according bit in $B[h(v)]$ and transfer bit vector B to N_s
  2. Node N_s: compute $s' = \{t \in s \mid B[h(t.A)] \text{ is set } \}$ and transfer to N_r
  3. Node N_r: compute $r \bowtie s' = r \bowtie s$

Bit Vector-Join /3

- Comparison:
  - Decreased size of request message compared to semi-join
  - Hash-mapping not injective $\rightarrow$ only potential join partners in bit vector
    $\rightsquigarrow$ sufficiently great n and suitable hash function h required

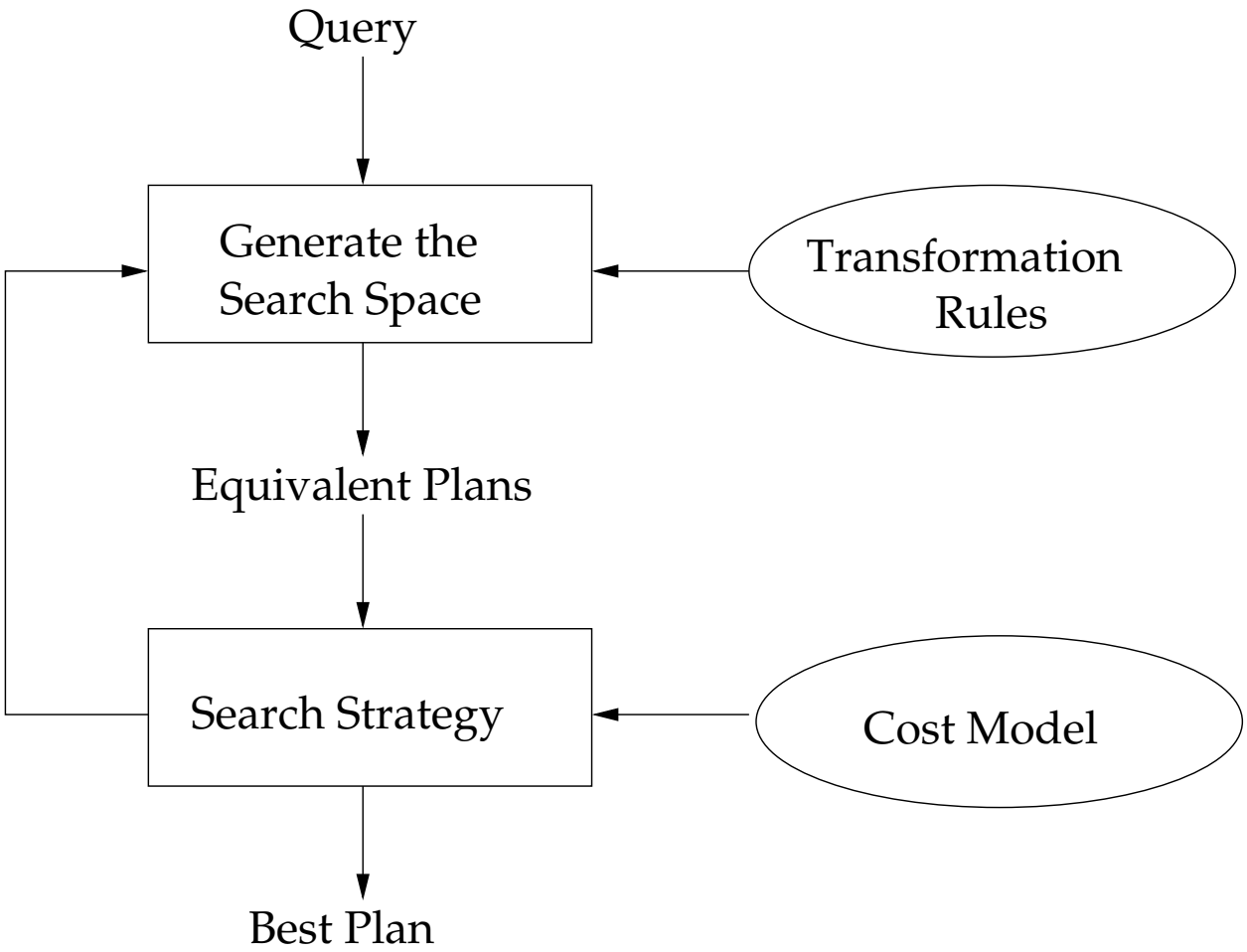
Bit Vector-Join: Example



Global Optimization

- Task: selection of most cost-efficient plan from set of possible query plans
- Prerequisite: knowledge about
  - Fragmentation
  - Fragment and relation sizes
  - Value ranges and distributions
  - Cost of operations/algorithms
- In Distributed DBS often details for nodes not known:
  - Existing indexes, storage organization, ...
  - Decision about usage is task of local optimization

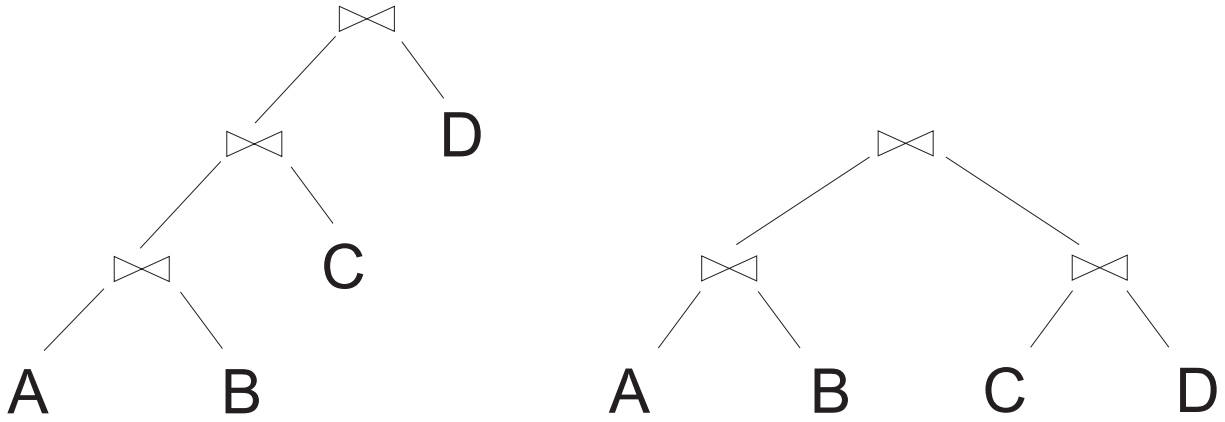
Cost-based optimization: Overview



Optimization: Search Space

- Search space: set of all equivalent query plans
- Generated by transformation rules:
  - Algebraic rules with no preferred direction, e.g. join commutativity and associativity (join trees)
  - Assignment of operation implementation/algorithm, e.g. distributed join processing
  - Assignment of operations to nodes
- Constraining the search space
  - Heuristics (like algebraic optimization)
  - Usage of "'preferred"' query plans (e.g. pre-defined join trees)

Optimization: Join Trees



- *Left deep trees* or *right deep trees* $\rightsquigarrow$ join order as nested structure/loops, all inner nodes (operations) have at least one input relation
- *Bushy trees* $\rightsquigarrow$ better potential for parallel processing, but higher optimization efforts required (greater number of possible alternatives)

Optimization: Search Strategy

- Traversing the search space and selection of best plan based on cost model:
  - *Which plans* are considered: full or partial traversal
  - In *which order* are the alternatives evaluated
- Variants:
  - **Deterministic**: systematic generation of plans as bottom up construction, simple plans for access to base relations are combined to full plans, grants best plan, computationally complex (e.g. dynamic programming)
  - **Random-based**: create initial query plan (e.g. with greedy strategy or heuristics) and improve these by randomly creating "‘neighbors"’, e.g. exchanging operation algorithm or processing location or join order, less expensive (e.g. genetic algorithms) but does not grant best plan

Cost Model

- Allows comparison/evaluation of query plans
- Components
  - Cost function
    - ⋆ Estimation of costs for operation processing
  - Database statistics
    - ⋆ Data about relation sizes, value ranges and distribution
  - Formulas
    - ⋆ Estimation of sizes of intermediate results (input for operations)

Cost Functions

- **Total time**
  - Sum of all time components for all nodes / transfers

$$T_{\text{total}} = T_{\text{CPU}} * \#insts + T_{\text{I/O}} * \#I/Os + T_{\text{MSG}} * \#msgs + T_{\text{TR}} * \#bytes$$

  - Communication time:

$$CT(\#bytes) = T_{\text{MSG}} + T_{\text{TR}} * \#bytes$$

  - Coefficients characteristic for Distributed DBS:
  - WAN: communication time (T_{MSG}, T_{TR}) dominates
  - LAN: also local costs (T_{CPU}, $T_{\text{I/O}}$) relevant

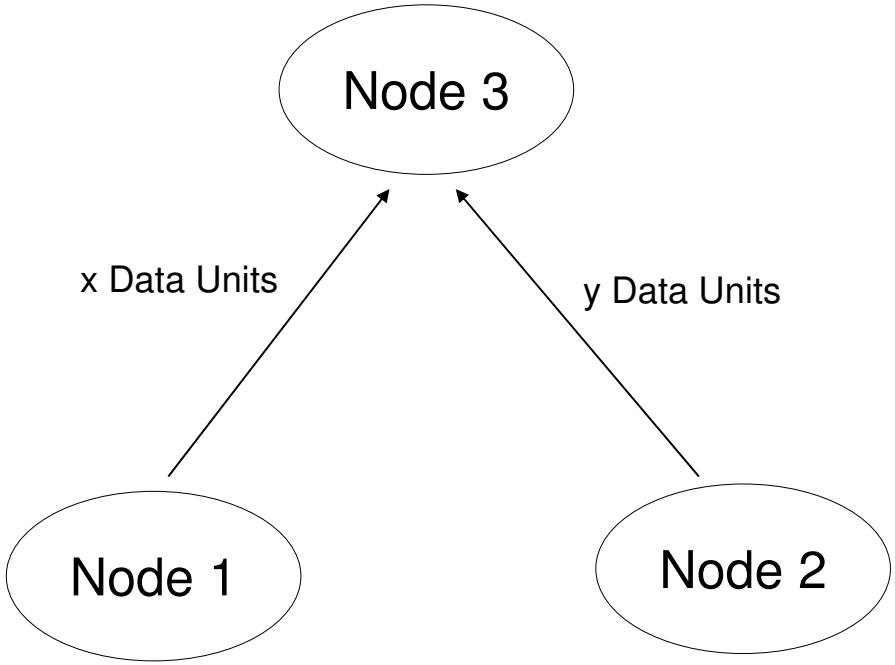
Cost Functions /2

- **Response time**
  - Timespan from initiation of query until availability of full results

$$T_{\text{total}} = T_{\text{CPU}} * \textit{seq\_\#insts} + T_{\text{I/O}} * \textit{seq\_\#I/Os} + T_{\text{MSG}} * \textit{seq\_\#msgs} + T_{\text{TR}} * \textit{seq\_\#bytes}$$

  - With *seq_#x* is maximum number *x* that must be performed sequentially

Total Time vs. Response Time



$$
\begin{aligned}
T_{\text{total}} &= 2T_{\text{MSG}} + T_{\text{TR}}(x + y) \\
T_{\text{response}} &= \max\{T_{\text{MSG}} + T_{\text{TR}} * x, T_{\text{MSG}} + T_{\text{TR}} * y\}
\end{aligned}
$$

Database statistics

- Main factor for costs: size of intermediate results
- Estimation of sizes based on *statistics*
- For relation R with attributes $A_1, \ldots, A_n$ and fragments $R_1, \ldots, R_f$
  - Attribute size: $\mathsf{length}(A_i)$ (in Byte)
  - Number of distinct values of A_i for each fragment R_j: $\mathsf{val}(A_i, R_j)$
  - Min and max attribute values: $\mathsf{min}(A_i)$ and $\mathsf{max}(A_i)$
  - Cardinality of value domain of A_i: $\mathsf{card}(\mathsf{dom}(A_i))$
  - Number of tuples in each fragment: $\mathsf{card}(R_j)$

Cardinality of Intermediate Results

- Estimation often based on following simplifications
  - Independence of different attributes
  - Equal distribution of attribute values
- Selectivity factor SF:
  - Ratio of result tuples vs. input relation tuples
  - Example: $\sigma_F(R)$ returns 10% of tuples from R
    $\rightsquigarrow SF = 0.1$
- Size of an intermediate relation:

$$\text{size}(R) = \text{card}(R) * \text{length}(R)$$

Cardinality of Selections

- Cardinality

$$\text{card}(\sigma_F(R)) = SF_S(F) * \text{card}(R)$$

- SF depends on selection condition with predicates $p(A_i)$ and constants v

$$\begin{aligned} SF_S(A = v) &= \frac{1}{\text{val}(A, R)} \\ SF_S(A > v) &= \frac{\max(A) - v}{\max(A) - \min(A)} \\ SF_S(A < v) &= \frac{v - \min(A)}{\max(A) - \min(A)} \end{aligned}$$

Cardinality of Selections /2

$$
\begin{aligned}
SF_S(p(A_i) \wedge p(A_j)) &= SF_S(p(A_i)) * SF_S(p(A_j)) \\
SF_S(p(A_i) \vee p(A_j)) &= SF_S(p(A_i)) + SF_S(p(A_j)) - \\
&\quad (SF_S(p(A_i)) * SF_S(p(A_j))) \\
SF_S(A \in \{v_1, \ldots, v_n\}) &= SF_S(A = v) * \mathrm{card}(\{v_1, \ldots, v_n\})
\end{aligned}
$$

Cardinality of Projections

- Without duplicate elimination

$$\mathsf{card}(\pi_A(R)) = \mathsf{card}(R)$$

- With duplicate elimination (for non-key attributes A)

$$\mathsf{card}(\pi_A(R)) = \mathsf{val}(A, R)$$

- With duplicate elimination (a key is subset of attributes in A)

$$\mathsf{card}(\pi_A(R)) = \mathsf{card}(R)$$

Cardinality of Joins

- Cartesian products

$$\text{card}(R \times S) = \text{card}(R) * \text{card}(S)$$

- Join
  - Upper bound: cardinality of Cartesian product
  - Better estimation for foreign key relationships $S.B \rightarrow R.A$:

$$\text{card}(R \bowtie_{A=B} S) = \text{card}(S)$$

  - Selectivity factor SF_J from database statistics

$$\text{card}(R \bowtie S) = SF_J * \text{card}(R) * \text{card}(S)$$

Cardinality of Semi-joins

- Operation $R \ltimes_A S$
- Selectivity factor for attribute A from relation S: $SF_{SJ}(S.A)$

$$SF_{SJ}(R \ltimes_A S) = \frac{\text{val}(A, S)}{\text{card}(\text{dom}(A))}$$

- Cardinality:

$$\text{card}(R \ltimes_A S) = SF_{SJ}(S.A) * \text{card}(R)$$

Cardinality of Set Operations

- Union $R \cup S$
  - Lower bound: $\max\{\text{card}(R), \text{card}(S)\}$
  - Upper bound: $\text{card}(R) + \text{card}(S)$
- Set difference $R - S$
  - Lower bound: 0
  - Upper bound: $\text{card}(R)$

Example

- Fragmentation:
  $\text{PROJECT} = \text{PROJECT}_1 \cup \text{PROJECT}_2 \cup \text{PROJECT}_3$
- Query:
  $$\sigma_{\text{Budget}>150.000}(\text{PROJECT})$$
- Statistics:
  - $\text{card}(\text{PROJECT}_1) = 3.500$, $\text{card}(\text{PROJECT}_2) = 4.000$, $\text{card}(\text{PROJECT}_3) = 2.500$
  - $\text{length}(\text{PROJECT}) = 30$
  - $\min(\text{Budget}) = 50.000$, $\max(\text{Budget}) = 300.000$
  - $T_{\text{MSG}} = 0.3s$
  - $T_{\text{TR}} = 1/1000s$

Example: Query Plans

- Variant 1:

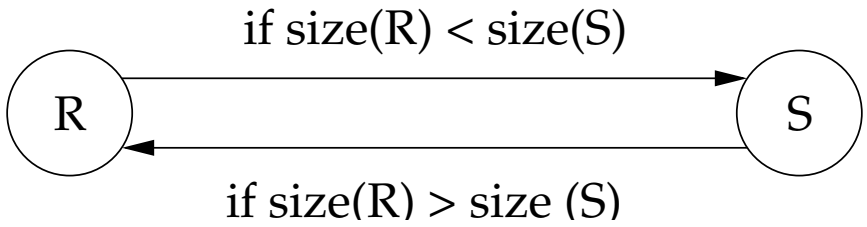
$$\sigma_{\text{Budget}>150.000}(\text{PROJECT}_1 \cup \text{PROJECT}_2 \cup \text{PROJECT}_3)$$

- Variant 2:

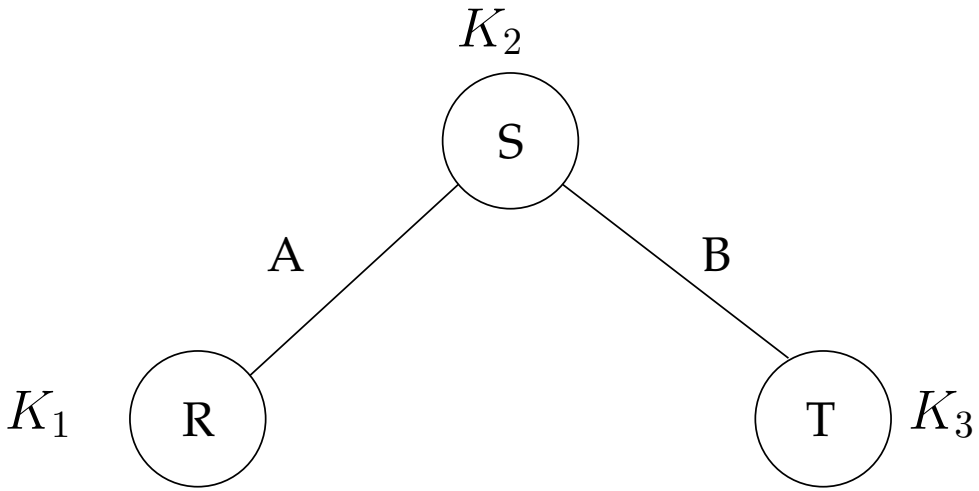
$$\sigma_{\text{Budget}>150.000}(\text{PROJECT}_1) \cup$$
$$\sigma_{\text{Budget}>150.000}(\text{PROJECT}_2) \cup$$
$$\sigma_{\text{Budget}>150.000}(\text{PROJECT}_3)$$

Join Order in DDBS

- Huge influence on overall performance
- General rule: avoid Cartesian products where possible
- Join order for 2 relations $R \bowtie S$



- Join order for 3 relations $R \bowtie_A S \bowtie_B T$



Join Order in DDBS /2

- (cont.) Possible strategies:
  1. $R \rightarrow N_2$; N_2 computes $R' := R \bowtie S$; $R' \rightarrow N_3$; N_3 computes $R' \bowtie T$
  2. $S \rightarrow N_1$; N_1 computes $R' := R \bowtie S$; $R' \rightarrow N_3$; N_3 computes $R' \bowtie T$
  3. $S \rightarrow N_3$; N_3 computes $S' := S \bowtie T$; $S' \rightarrow N_1$; N_1 computes $S' \bowtie R$
  4. $T \rightarrow N_2$; N_2 computes $T' := T \bowtie S$; $T' \rightarrow N_1$; N_1 computes $T' \bowtie R$
  5. $R \rightarrow N_2$; $T \rightarrow N_2$; N_2 computes $R \bowtie S \bowtie T$
- Decision based on size of relations and intermediate results
- Possible utilization of parallelism in variant 5

Utilization of Semi-Joins

- Consideration of semi-join-based strategies
- Relations R at node N_1 and S at node N_2
- Possible strategies $R \bowtie_A S$
  1. $(R \ltimes_A S) \bowtie_A S$
  2. $R \bowtie_A (S \ltimes_A R)$
  3. $(R \ltimes_A S) \bowtie_A (S \ltimes_A R)$
- Comparison $R \bowtie_A S$ vs. $(R \ltimes_A S) \bowtie_A S)$ for $\text{size}(R) < \text{size}(S)$
- Costs for $R \bowtie_A S$: transfer of R to $N_2 \rightsquigarrow T_{TR} * \text{size}(R)$

Utilization of Semi-Joins /2

- Processing of semi-join variant
  1. $\pi_A(S) \rightarrow N_2$
  2. At node N_2: computation of $R' := R \ltimes_A S$
  3. $R' \rightarrow N_1$
  4. At node N_1: computation of $R' \bowtie_A S$
- Costs: costs for step 1 + costs for step 2

$$T_{TR} * \text{size}(\pi_A(S)) + T_{TR} * \text{size}(R \ltimes_A S)$$

- Accordingly: semi-join is better strategy if

$$\text{size}(\pi_A(S)) + \text{size}(R \ltimes_A S) < \text{size}(R)$$

Summary: Global Optimization in DDBS

- Extension of centralized optimization regarding distribution aspects
  - Location of processing
  - Semi Join vs. Join
  - Fragmentation
  - Total time vs. response time
  - Consideration of additional cost factors like transfer time and number of message messages
- Current system implementations very different regarding which aspects are considered or not

Part III

Distributed DBS - Transaction Processing

Overview

- Foundations
- Distributed TXN Processing
- Transaction Deadlocks
- Transactional Replication

Foundations of TXN Management

A *Transaction* is a sequence of operations which represent a semantic unit and transfer a database from one consistent state to another consistent state adhering to the *ACID-principle*.

Aspects:

- Semantic integrity: consistent state must be reached after transaction, no matter if it succeeded or failed
- Process integrity: avoid failures due to concurrent parallel access by multiple users/transactions

Transactions: ACID Properties

- **A**tomicity means that a transaction can not be interrupted or performed only partially
  - TXN is performed in its entirety or not at all
- **C**onsistency to preserve data integrity
  - A TXN starts from a consistent database state and ends with a consistent database state
- **I**solation
  - Result of a TXN must be independent of other possibly running parallel TXNs
- **D**urability or persistence
  - After a TXN finished successfully (from the user's view) its results must be in the database and the effect can not be reversed

Commands of a TXN Language

- Begin of Transaction **`BOT`** (in SQL implicated by first statement)
- **`commit`**: TXN ends successfully
- **`abort`**: TXN must be aborted during processing

Problems with Processing Integrity

- Parallel accesses in multi-user DBMS can lead to the following problems
  - Non-repeatable reads
  - Dirty reads
  - The phantom problem
  - Lost updates

Non-repeatable Read

Example:

- Assertion: $X = A + B + C$ at the end of txn T_1
- X and Y are local variables
- T_i is txn i
- Integrity constraint on persistent data $A + B + C = 0$

Non-repeatable Read /2

T_1	T_2
$X := A;$	
	$Y := A/2;$
	$A := Y;$
	$C := C + Y;$
	commit;
$X := X + B;$	
$X := X + C;$	
commit;	

Dirty Read

T_1	T_2
read(X);	
$X := X + 100$;	
write(X);	
	read(X);
	$Y := Y + X$;
	write(Y);
	commit;
abort;	

The Phantom Problem

T_1	T_2
select count (*) **into** X **from** Employee;	
	insert **into** Employee **values** (*Meier*, 50000, $\cdots$); **commit**;
update Employee **set** Salary = Salary + 10000/X; **commit**;	

Lost Update

T_1	T_2	X
read(X);		10
	read(X);	10
$X := X + 1$;		10
	$X := X + 1$;	10
write(X);		11
	write(X);	11

Simplified TXN Model

Representation of (abstract) data object (values, tuples, pages) access

- **`read`**(A,x): assign value of DB object A to variable x
- **`write`**(x, A): assign value of x to DB object A

Example of a txn T:

`read`$(A, x);\ x := x - 200;$ ***`write`***$(x, A);$ ***`read`***$(B, y);$
$y := y + 100;$ ***`write`***$(y, B);$***`commit`***

Schedules: possible processing of two txns T_1, T_2:

- Serial schedule: T_1 before T_2 or T_2 before T_1
- Intertwined schedule: mixed execution of operations from both txns

Serializability

An intertwined schedule of a number of transactions is called **serializable**, if the effect of the intertwined schedule is identical to the effect of any of the possible serial schedules. The intertwined schedule is then called correct and **equivalent** to the serial schedule.

- Practical approaches for deciding about serializabilty most often only considering read/write operations and their conflicts $\rightarrow$ **Conflict Serializability**
- Considering other operations requires analysis of operations semantics
- Rules out subset of serializable schedules which are hard to detect

Conflicting Operations

T_1	T_2
read A	
	read A

independent of order

T_1	T_2
read A	
	write A

dependent on order

T_1	T_2
	write A
read A	

dependent on order

T_1	T_2
	write A
write A	

dependent on order

Conflict Serializability

A schedule s is called conflict-serializable, if the order of all pairs of conflicting operations is equivalent to the order of any serial schedule s' for the same transactions.

Tested using conflict graph $G(s) = (V, E)$ of schedule s:

1. Vertex set V contains all txns of s
2. Edge set E contains an edge for each pair of conflicting operations

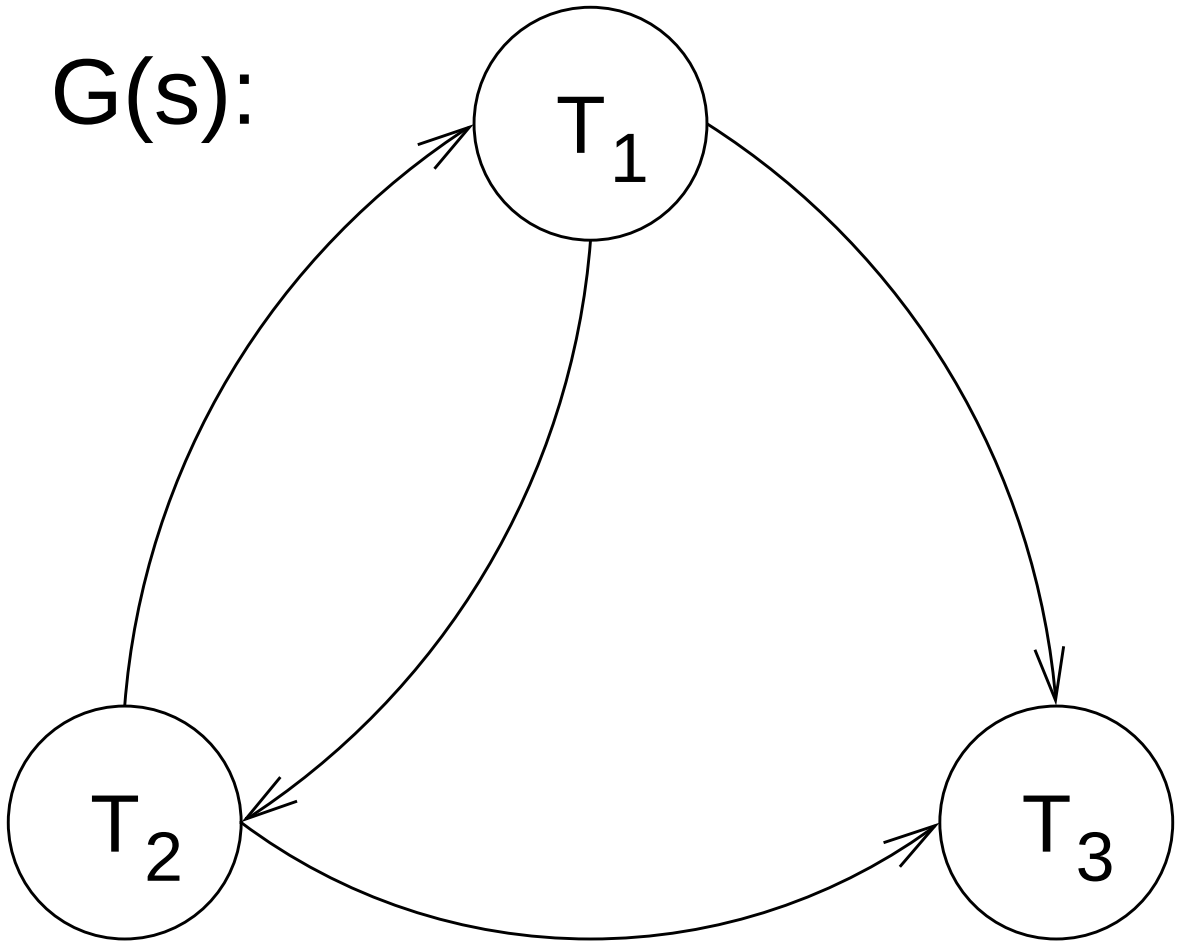
In serial schedules s' there can no cycles, i.e. if a cycle exists the schedule s can not be equivalent to a serial schedule and must be rejected.

Conflict Serializability: Example /1

T_1	T_2	T_3
r(y)		
		r(u)
	r(y)	
w(y)		
w(x)		
	w(x)	
	w(z)	
		w(x)

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

Conflict Serializability: Example /2



Transaction Synchronization

1. Most common practical solution: Locking Protocols
   - TXNs get temporarily exclusive access to DB object (tuple, page, etc.)
   - DBMS manages temporary locks
   - Locking protocol grants cnflict serializabilty without further tests
2. In distributed DBMS also: timestamp-based protocols

Locking Protocols

Read and write locks using the following notation:

- $rl(x)$: read lock on object x
- $wl(x)$: write lock on object x

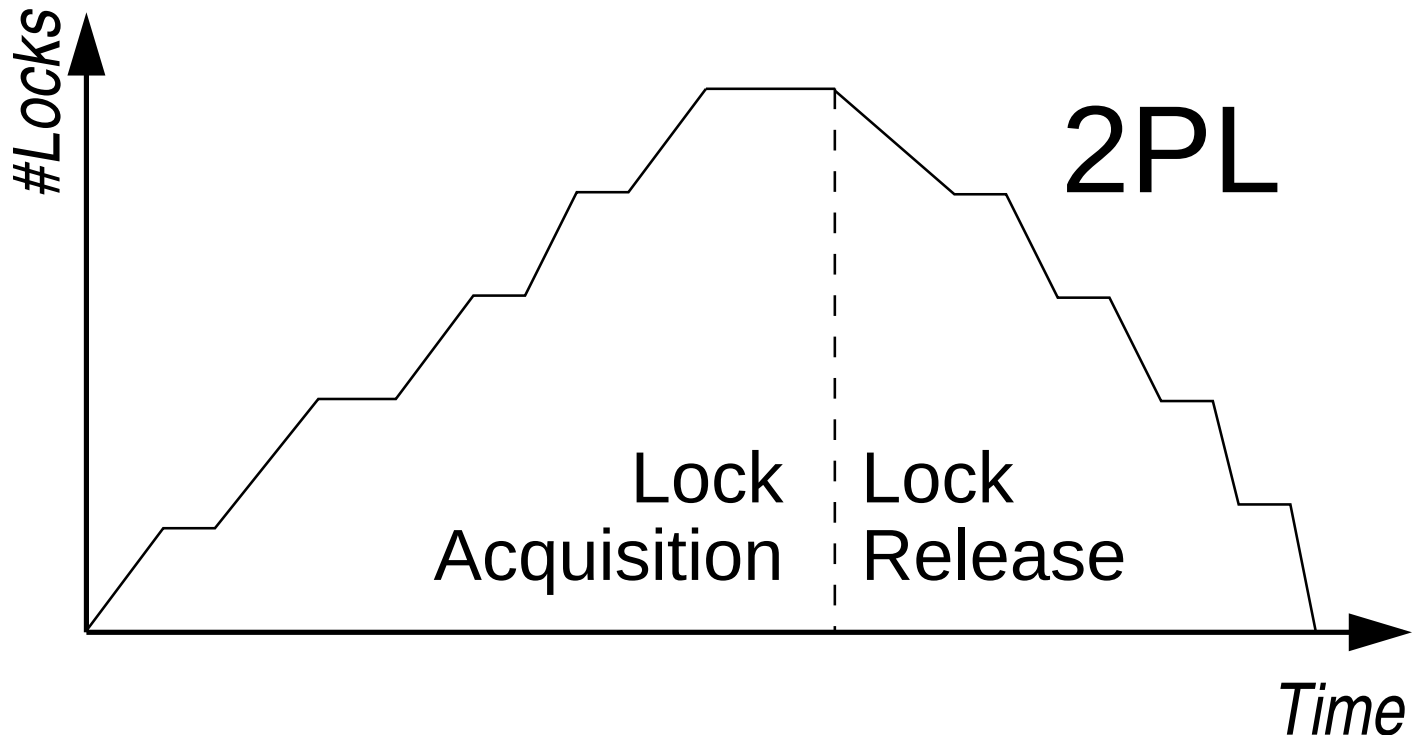
Unlock $ru(x)$ and $wu(x)$, often combined $u(x)$ *unlock* object x

Locks: Compatibility Matrix

- For basic locks

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	$\surd$	—
$wl_j(x)$	—	—

2-Phase-Locking Protocol

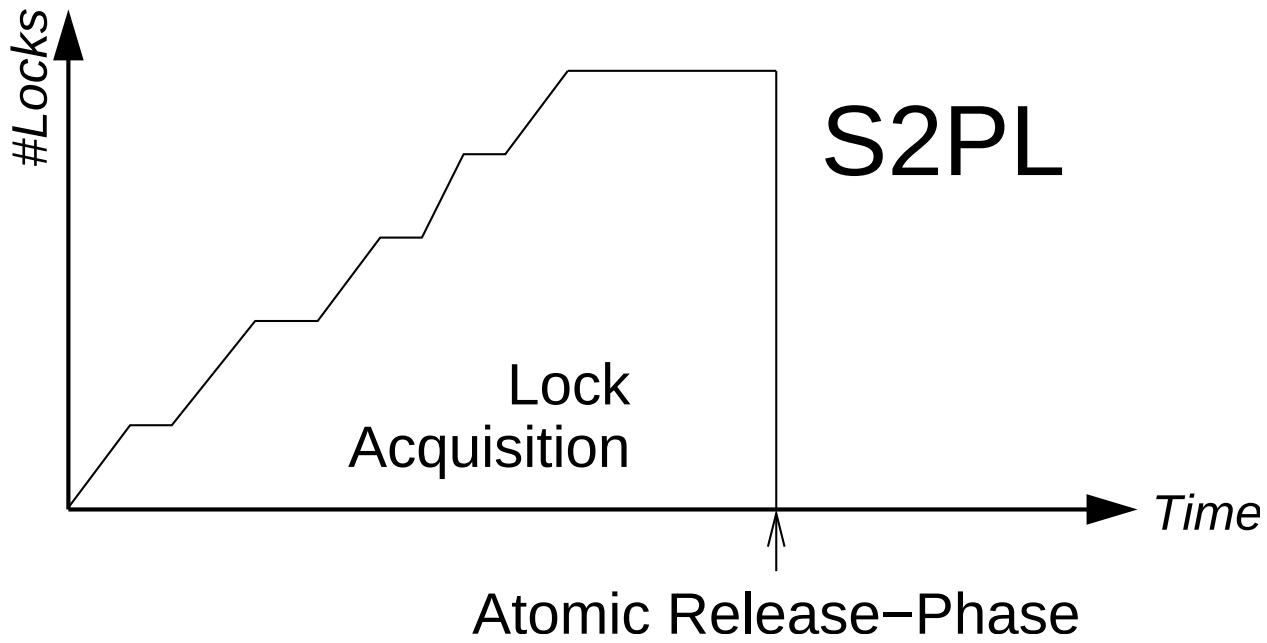


2-Phase-Locking: Example

T_1	T_2
$u(x)$	
	$wl(x)$
	$wl(y)$
	$\vdots$
	$u(x)$
	$u(y)$
$wl(y)$	
$\vdots$	

Strict 2-Phase-Locking

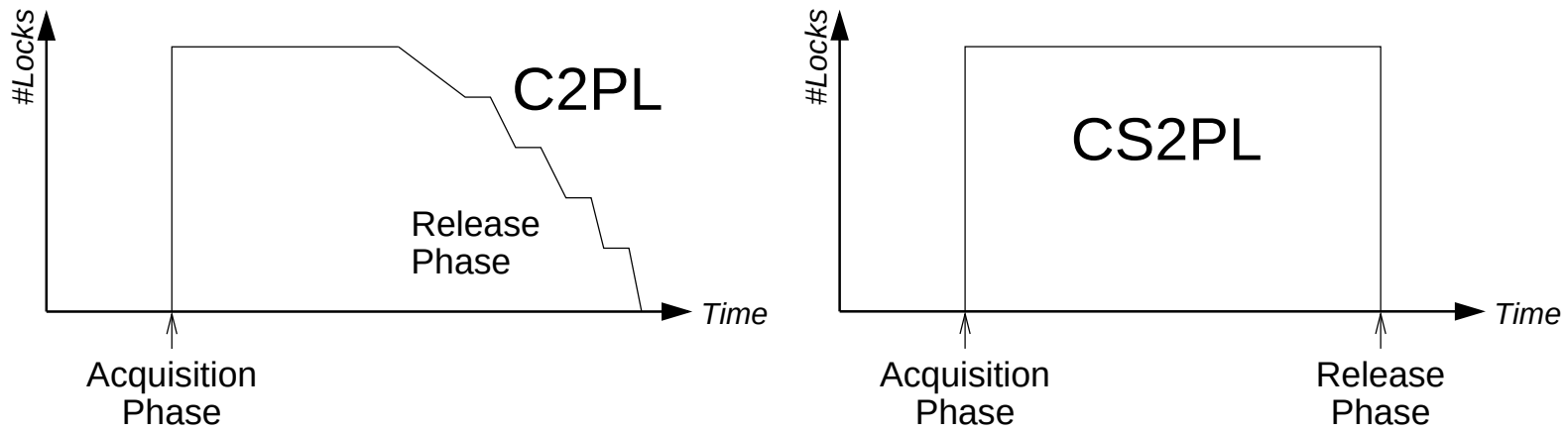
Current practice in most DBMS:



Avoids cascading aborts!

Conservative 2-Phase-Locking

To avoid Deadlocks:

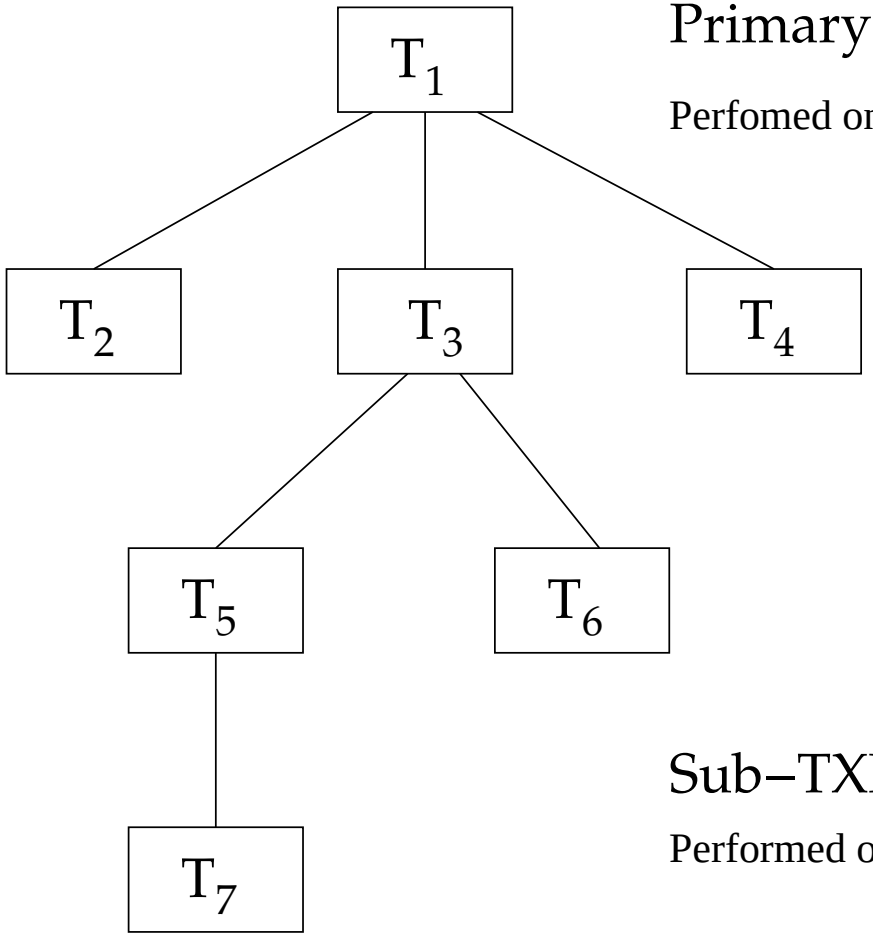


Most often not practical!

Distributed TXN Processing

- In DDBS one TXN can run on multiple nodes
- Distributed synchronization required for parallel TXNs required
- Commit as atomic event $\rightarrow$ same result on all nodes
- Deadlocks (blocking/blocked TXNs) harder to detect

Structure of Distributed TXNs



Primary TXN

Perfomed on coordinator node

Sub–TXNs

Performed on other nodes

Distributed Synchronization with Locking

- One central node for lock management
  - Dedicated node becomes bottleneck
  - Low node autonomy
  - High number of messages for lock acquisition/release
- Distributed lock management on all nodes
  - Possible, if data (relations, fragments) is stored non-redundantly
  - Special strategies for replicated data
  - Disadvantage: deadlocks are hard to detect
- Latter, i.e. distributed S2PL, is state-of-the-art
- Alternative: Timestamp-based Synchronisation

Timestamp-based Synchronization

- Unique timestamps are sequentially assigned to TXNs
- For each data object (tuple, page, etc.) x two values are stored:
  - **`max-r-scheduled`**$[x]$:
    timestamp of last TXN performing a read operation on x
  - **`max-w-scheduled`**$[x]$:
    timestamp of last TXN performing a write operation on x

Timestamp-Ordering

An operation $p_i[x]$ can be performed before a conflicting operation $q_k[x]$ iff $ts(T_i) < ts(T_k)$. Otherwise, $q_k[x]$ must be rejected.

T_1	T_2	T_3	A		B		C	
			mrs	**mws**	**mrs**	**mws**	**mrs**	**mws**
$ts = 200$	$ts = 150$	$ts = 175$	0	0	0	0	0	0
read B			0	0	200	0	0	0
	read A		150	0	200	0	0	0
		read C	150	0	200	0	175	0
write B			150	0	200	200	175	0
write A			150	200	200	200	175	0
	write C ⇓		150	200	200	200	175	⇓
	abort		150	200	200	200	175	0

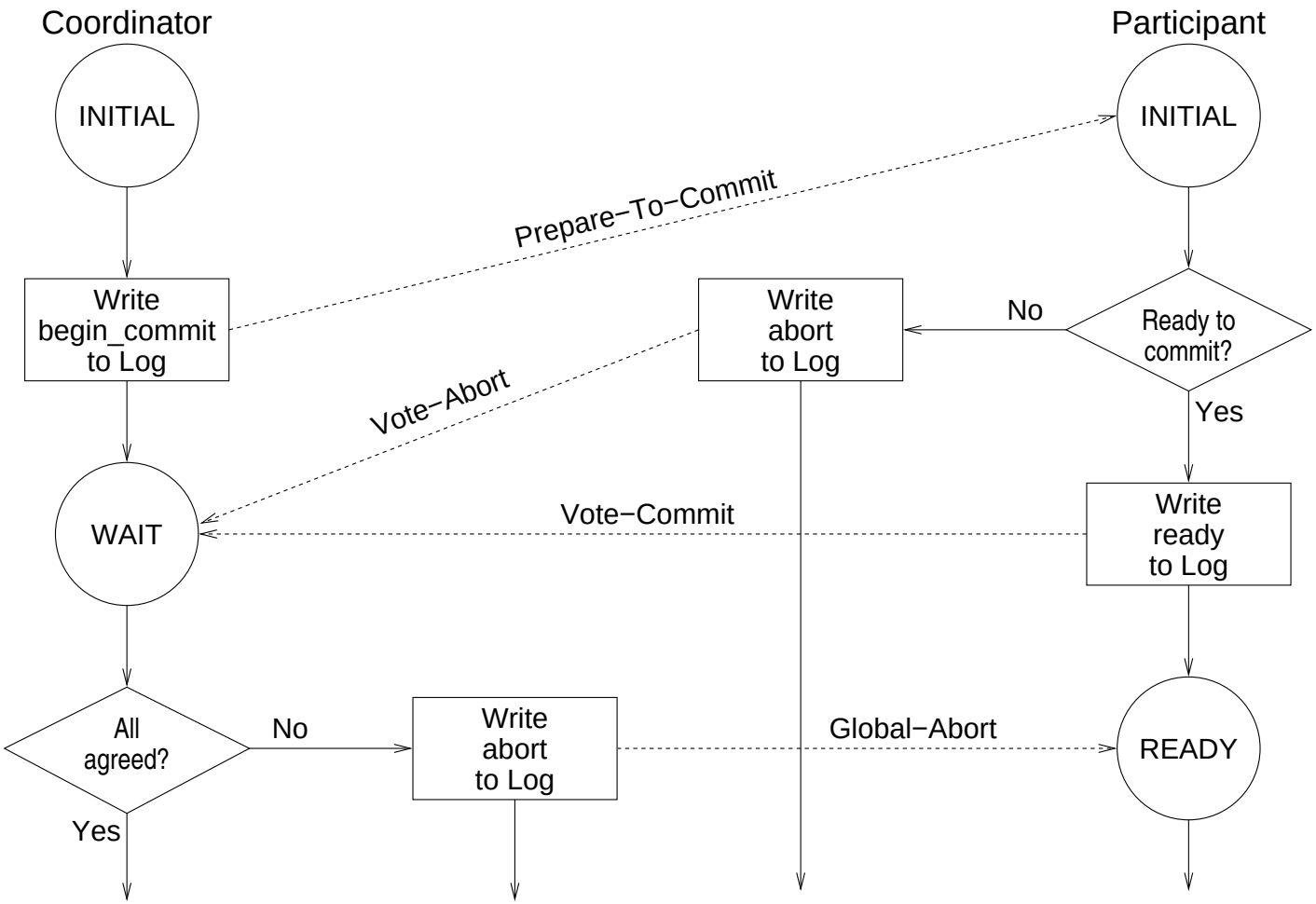
Distributed Commit

- Synchronization provides **C**onsistency and **I**solation
- Commit Protocol provides **A**tomicity and **D**urability
- Requirements in DDBS:
  - All participating nodes of one TXN with same result (**`Commit`**, **`Abort`**)
  - **`Commit`** only if all nodes vote "‘yes"’
  - **`Abort`** if at least one node votes "‘no"’
- X/open XA standard for 2-Phase-Commit Protocol (used in DBMS, request brokers, application servers, etc.)

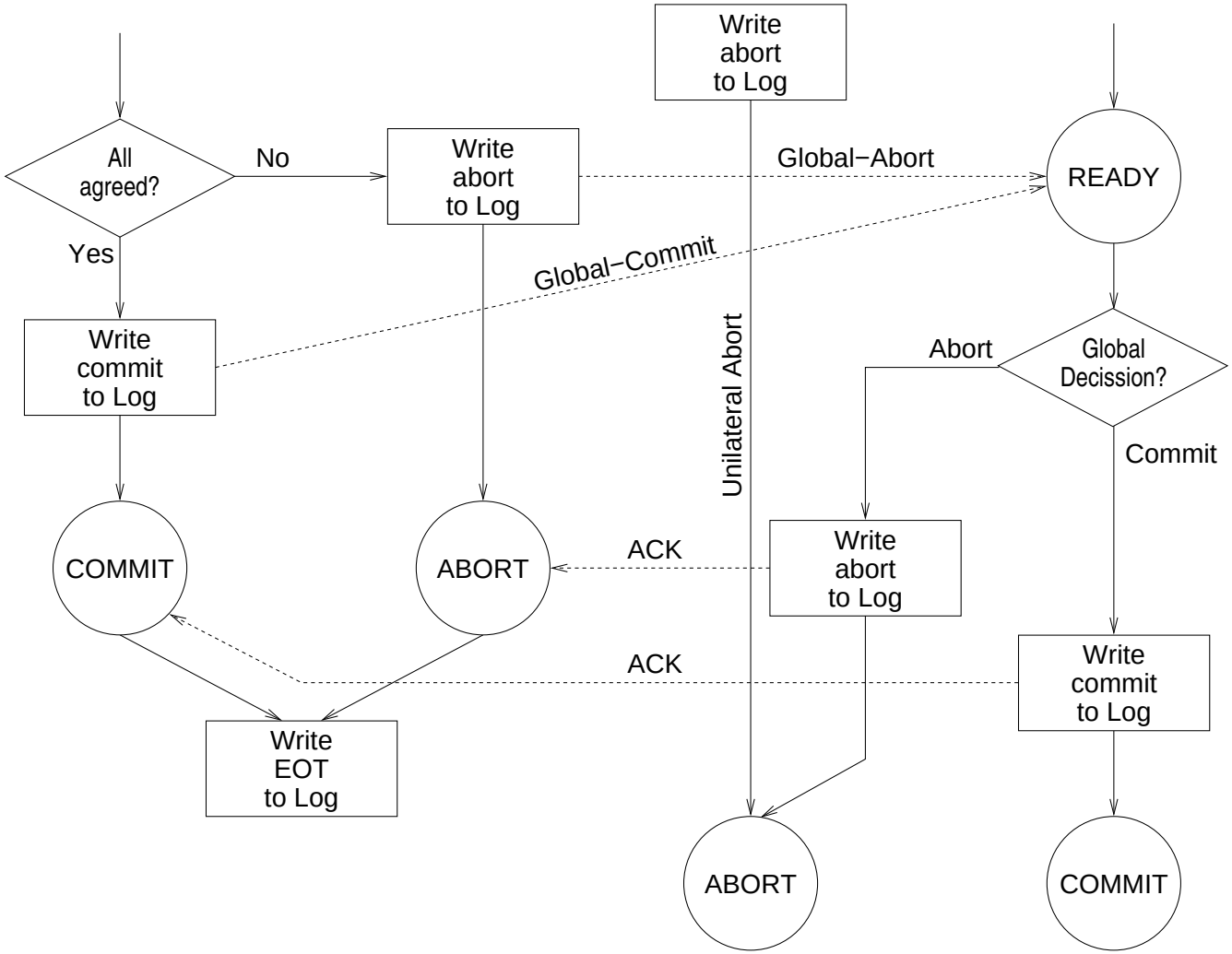
2-Phase-Commit Protocol (2PC)

- Roles: 1 coordinator, several other participants
- Procedure:
  1. *Commit Request Phase*
     1. Coordinator queries all participants, if **`Commit`** can be executed
     2. Participants send their local reply message, if they agree with the commit
  2. *Commit Phase*
     1. Coordinator decides globally: (all messages **`Commit`** $\rightarrow$ **`Global-Commit`**; at least one **`Abort`** $\rightarrow$ **`Global-Abort`**
     2. Participants which voted "'yes"' have to wait for final result

2PC: Procedure /1



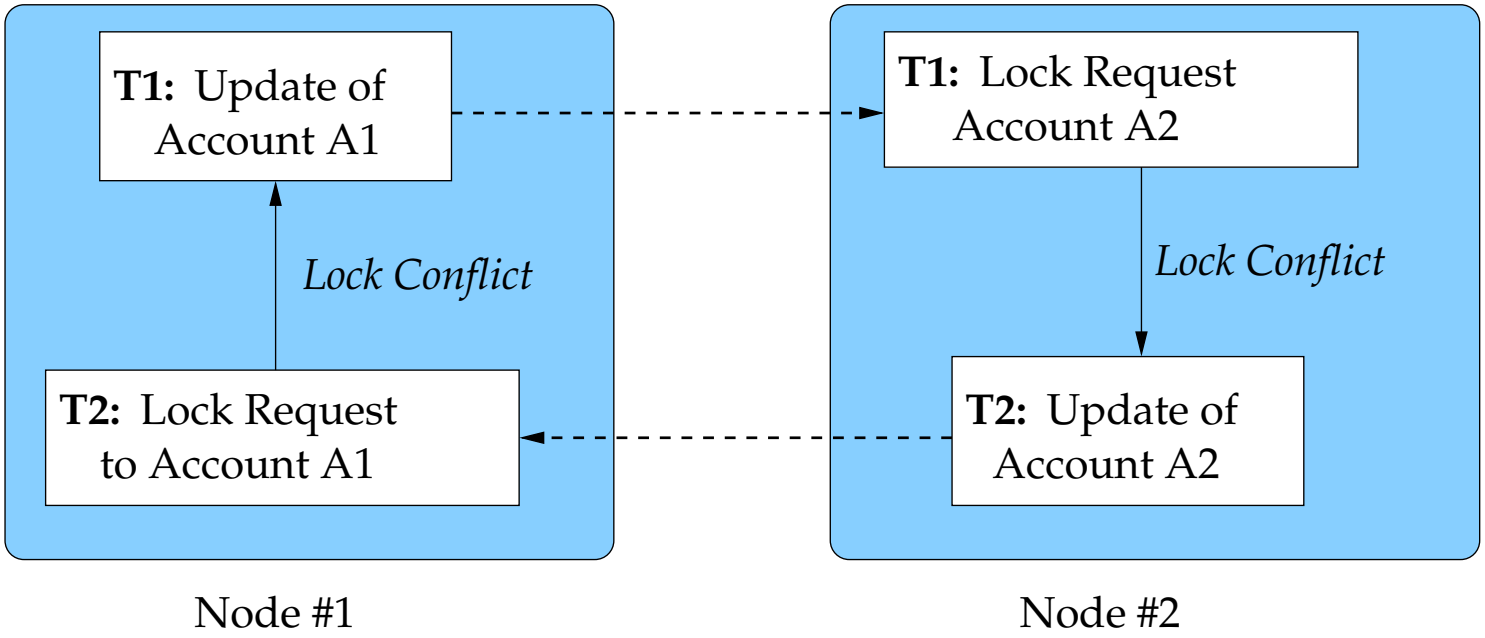
2PC: Procedure /2



3-Phase-Commit Protocol (3PC)

- Possible problem with 2PC: if coordinator fails while other participants are in **`READY`** state, these may **block indefinitely**
- Solution:
  - Intermediate **`PRE-COMMIT`** phase added, so that a certain number K (system parameter) of other participants know about possible positive result
  - Timeout values to avoid blocking: if communication timed out an no other node is in **`PRE-COMMIT`** state, TXN must abort
- Disadvantagees
  - 3PC has increased message number
  - Still problematic in case of network partitioning

Transaction Deadlocks



Dealing with TXN Deadlocks

- **Deadlock Prevention:**
  - Implement TXN management in a way that makes deadlocks impossible, e.g. lock pre-claiming in Conservative 2PL
  - Most often not practical or efficient
- **Deadlock Avoidance:**
  - TXN management reacts to possible deadlock situations, e.g. timeout for lock requests
  - Easy to implement, but overly restrictive and not always efficient
- **Deadlock Detection and Resolution:**
  - TXM management detects deadlocks, e.g. using TXN Wait graphs
  - Efficient, but harder to implement - especially for DDBMS

Deadlock Avoidance: Timeouts

- Reset TXN if waiting time for lock acquisition exceeds pre-defined threshold
- Problem: setting the timeout threshold
  - Too short: unnecessary aborts
  - Too long: system throughput declines
- Timeout threshold specific for certain applications (typical TXN running times must be considered)
- Implemented in most commercial systems

Deadlock Avoidance: Timestamp-based

- Alternative: consider unique TXN timestamps assigned to each TXN with **BOT**
- Avoid deadlocks in case of lock conflict considering timestamp:
- In case of conflict only the
  - younger TXN (Wound/Wait)
  - older TXN (Wait/Die)

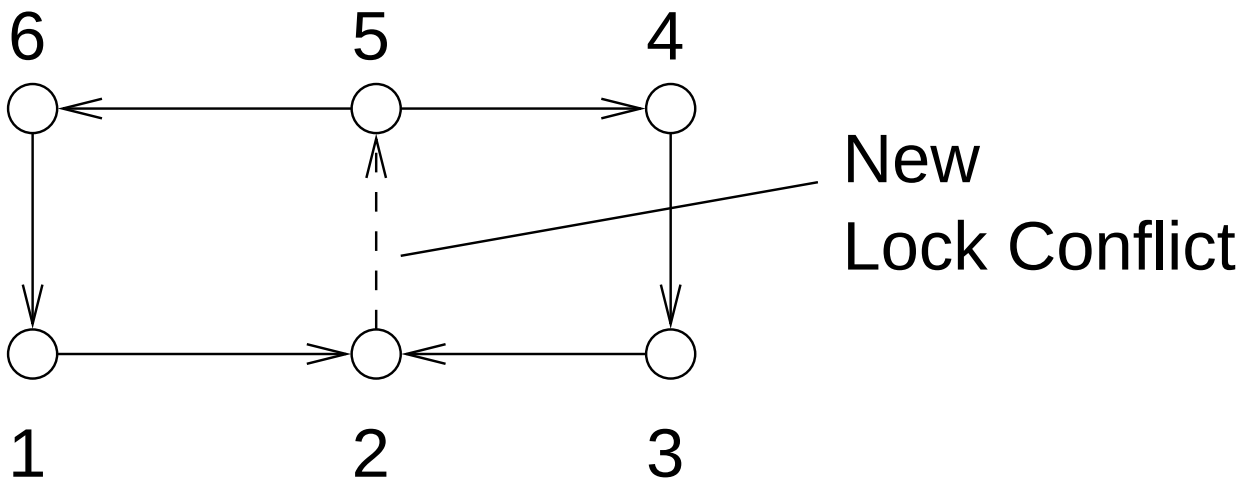
  will continue
- Can be combined with timeout (before timestamp check)

Deadlock Detection

- Common approach
  - Protocol each lock conflict in wait graph (nodes are TXNs, directed edge $T_1 \rightarrow T_2$ means that T_1 waits for a lock held by T_2
  - Deadlocks exist iff there is a cycle in the wait graph
  - Lock request leads to lock conflict $\rightarrow$ insert new edge in wait graph $\rightarrow$ check for cycles containing new edge

Deadlock Resolution

- Wait graph:



- Resolution choosing one TXN to abort upon following criteria
  - Number of resolved cycles
  - Age of TXN
  - Effort to rollback TXN
  - Priority of TXN, ...

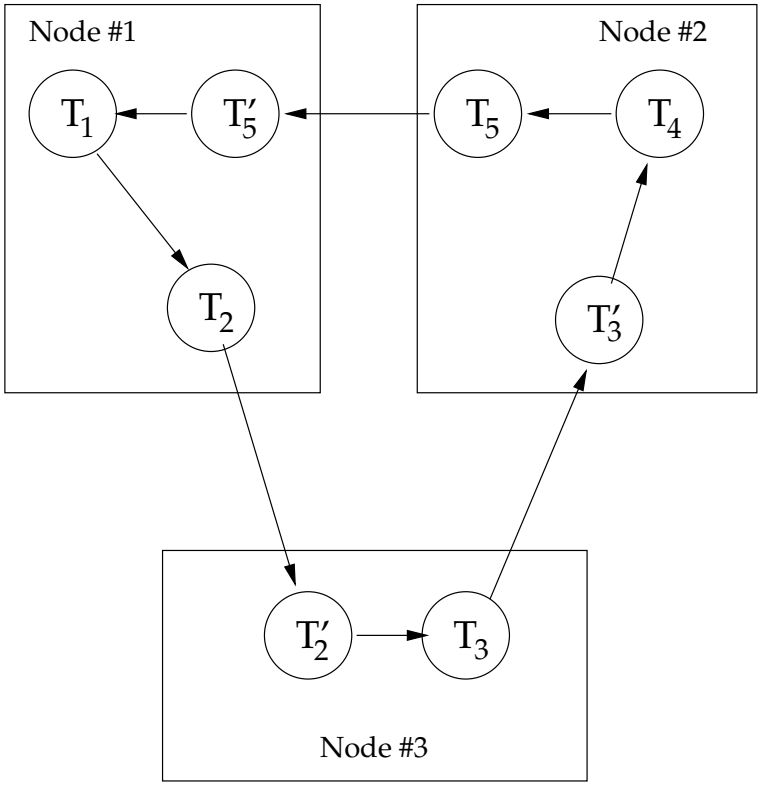
Centralized Deadlock Detection

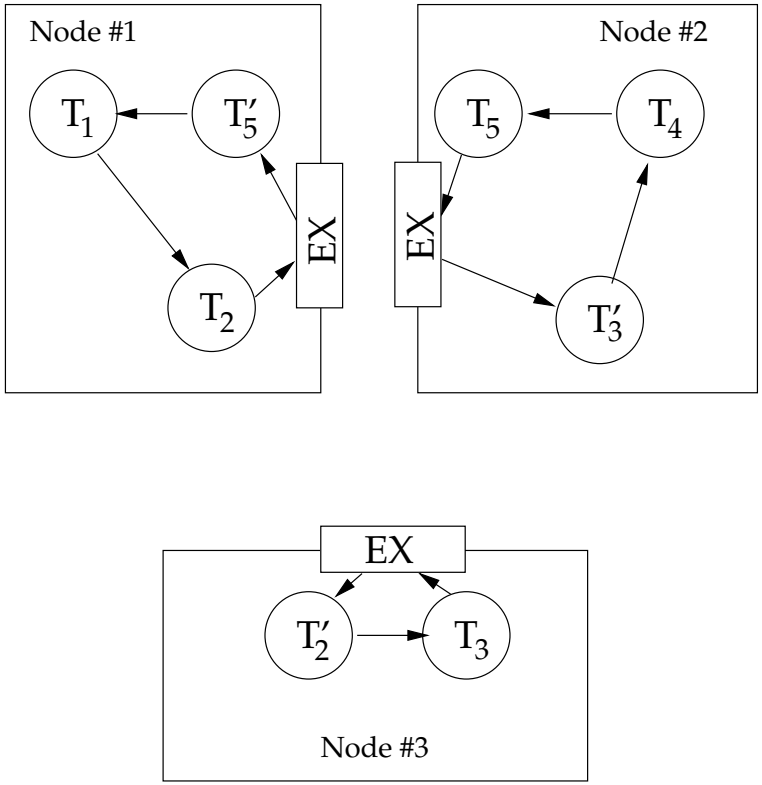
- Wait graph stored on one dedicated node
- Decrease number of messages by sending frequent bundles of collected lock waits
- Problems:
  - Late detection of deadlocks
  - Phantom-Deadlocks
  - Limited availability and node autonomy

Distributed Deadlock Detection

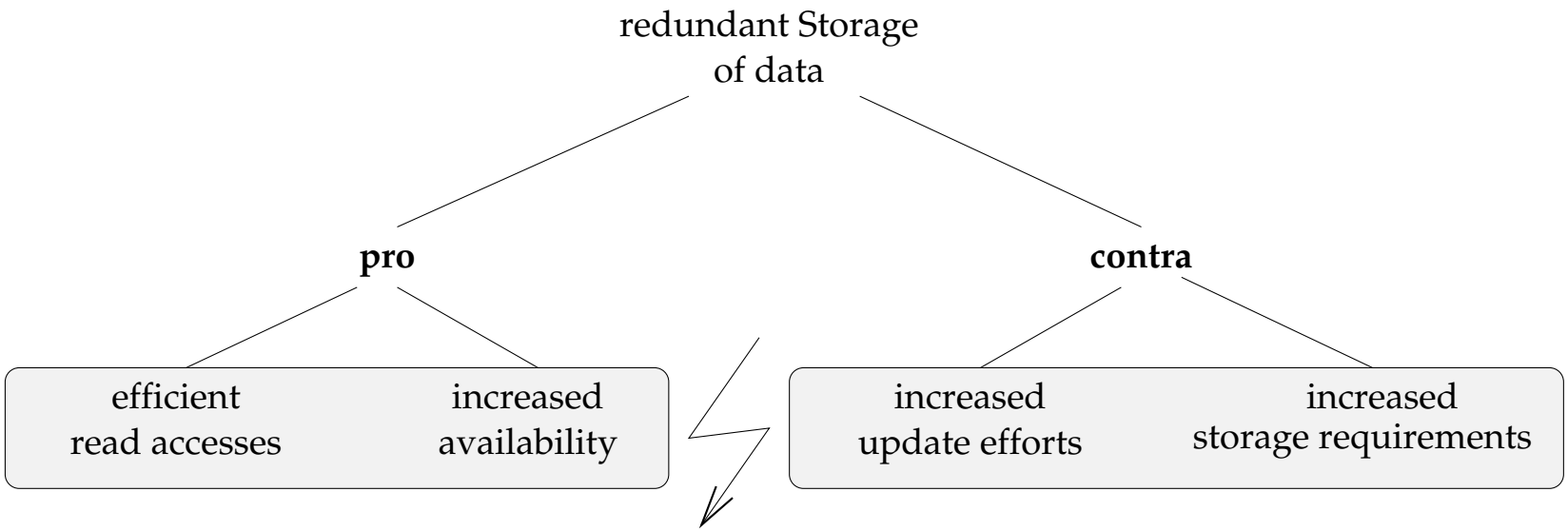
- Avoids dependency on global node
- Hard to realize: cycles can exist across different nodes $\rightarrow$ wait graph information needs to be exchanged between nodes
- Obermarck-Verfahren (System R*)
  - Each node with its own *deadlock detector*
  - Detector manages local wait graph + dependencies on locks of external transactions: one special node **`EX`** represents external transactions
  - Local deadlocks (not involving **`EX`** node) can be detected locally
  - Cycle including **`EX`** node may hint at distributed deadlock: cycle sub-graph is sent to node which created local cycle and possibly further to other involved nodes, until **`EX`** is resolved and decision can be made

Distributed Wait Graphs





Motivation for Replication



- Replication transparency $\rightsquigarrow$ DDBMS manages updates on redundant data

Transactional Replication

- Data integrity must be granted across replicas $\rightsquigarrow$ 1-Copy-Equivalence and 1-Copy-Serializability
- Problems to be solved:
  - How to efficiently update the copies?
  - How to handle failures (single nodes, network fragmentation)?
  - How to synchronize concurrent updates?
- Approaches
  - ROWA
  - Primary Copy
  - Consensus approaches
  - Others: Snapshot-, Epidemic, and Lazy Replication

ROWA Synchronization

- "'Read One Write All"'
  - **one** logial read operation $\rightarrow$ one physical read operation on **any** copy ("'read one"'), where read access can be performed most efficiently (local or closest copy)
  - **one** logical write operation $\rightarrow$ physical write operations on **all** copies ("'write all"')
  - Equivalent to "'normal"' transaction synchronization: all updates within **one** transaction context

ROWA: Evaluation

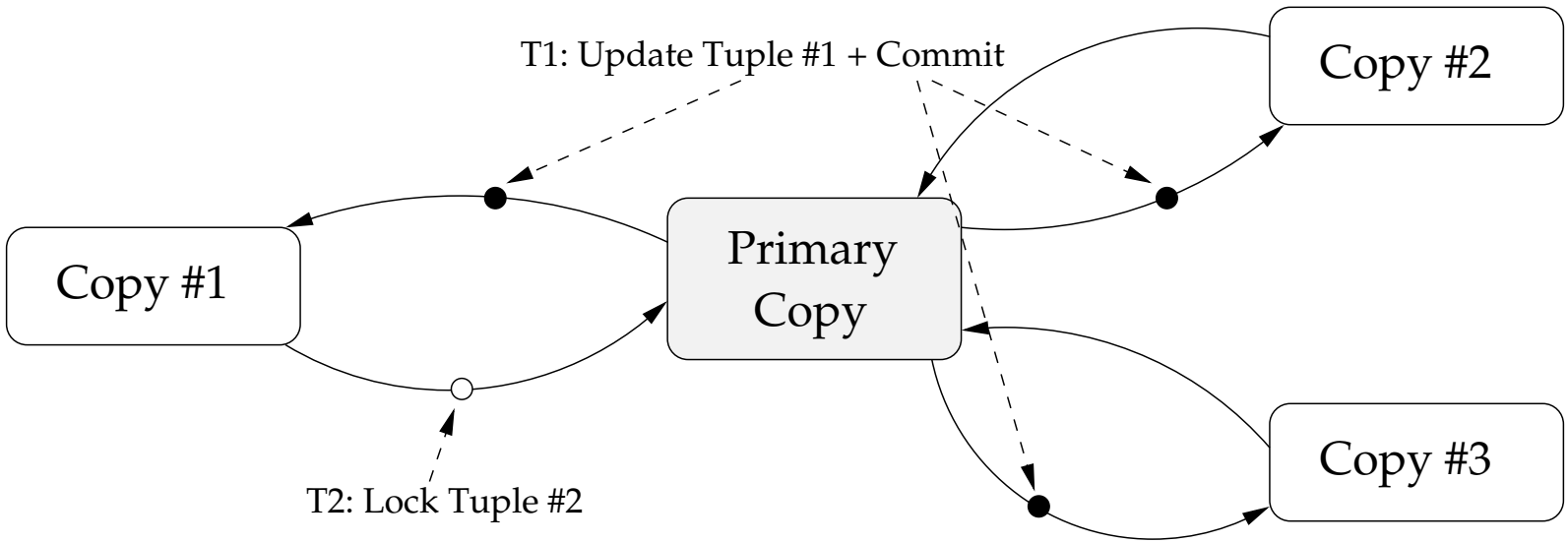
- Advantages
  - Approach is part of normal TXN processing
  - Easy to implement
  - Grants full consistency (1-Copy-Serializability)
  - Efficient local read operations
- Disadvantages
  - Updates dependent on availabilty of **all** nodes storing replicated data
  - Longer run-time of update TXNs $\rightsquigarrow$ decreased throughput and availabilty
  - Deadlocks more likely

Primary-Copy Replication

- Also: Master-Slave Replication
- First implemented in Distributed INGRES (Stonebraker 1979)
- Basic principle
  - Designation of a primary (master) copy ("‘original version"’); all secondary (slave) copies are derived from that original
  - All updates must first be performed on primary copy (lock relevant data objects, perform update, release locks)
  - In case of successful update, primary copy forwards updates **asychronously** to all secondary copies in separate TXNs
  - Read operations are performed locally

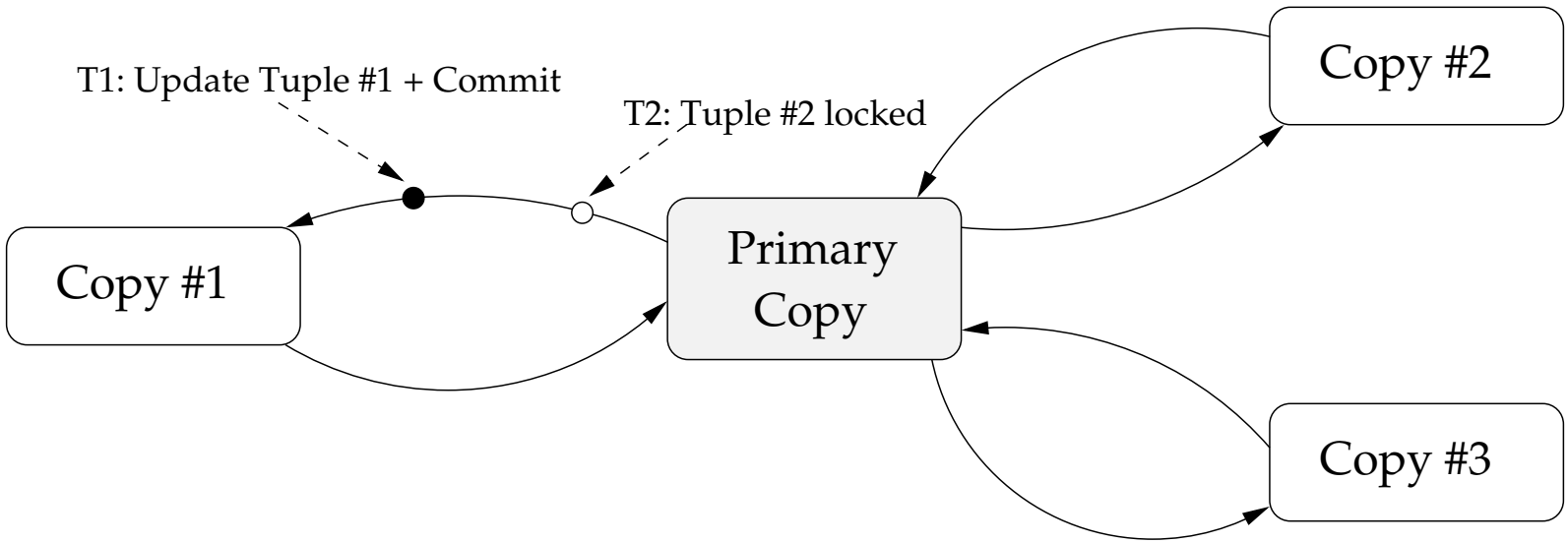
Primary-Copy: Example

- Secondary copies get changes from update queue (FIFO) $\rightsquigarrow$ consistency with primary copy
- In case of failure of secondary node, queue is processed when node is restarted



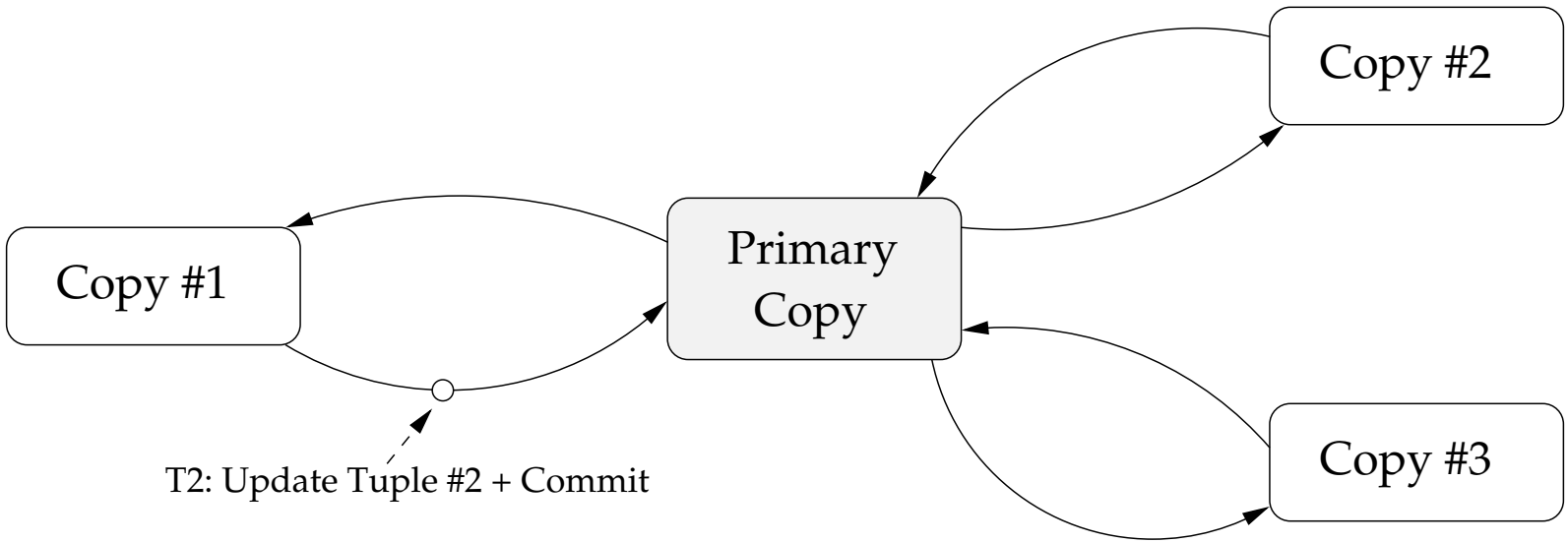
Primary-Copy: Example

- Secondary copies get changes from update queue (FIFO) $\rightsquigarrow$ consistency with primary copy
- In case of failure of secondary node, queue is processed when node is restarted



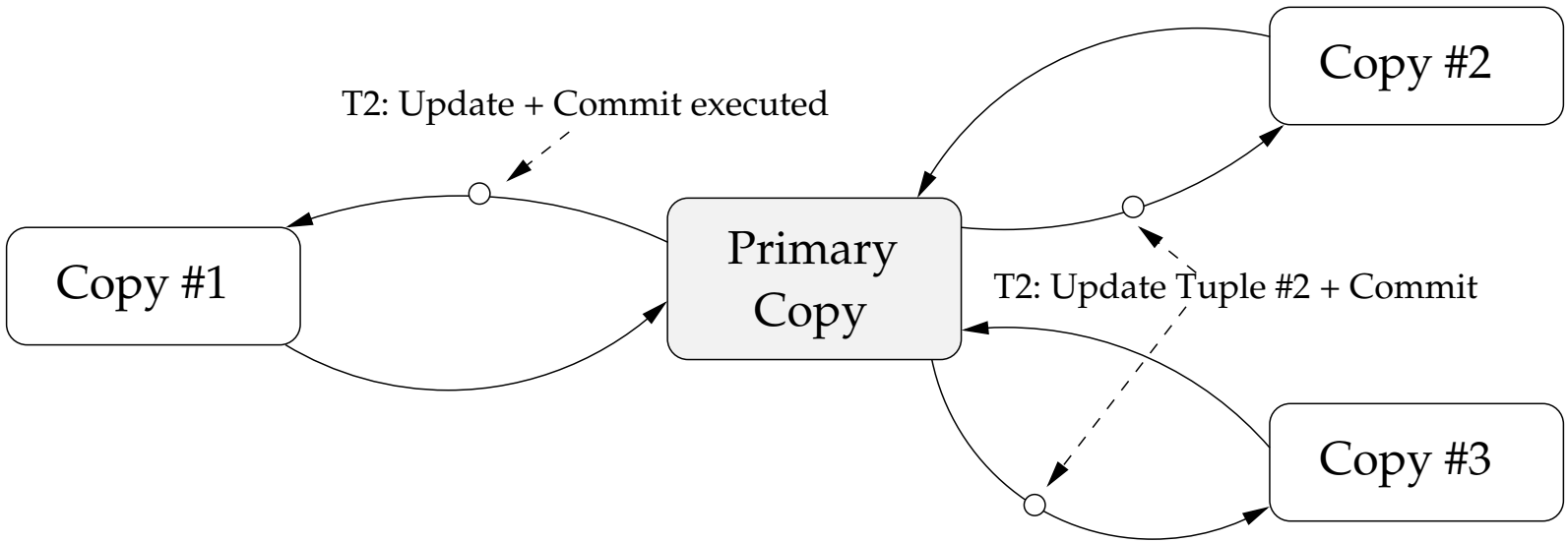
Primary-Copy: Example

- Secondary copies get changes from update queue (FIFO) $\rightsquigarrow$ consistency with primary copy
- In case of failure of secondary node, queue is processed when node is restarted



Primary-Copy: Example

- Secondary copies get changes from update queue (FIFO) $\rightsquigarrow$ consistency with primary copy
- In case of failure of secondary node, queue is processed when node is restarted



Primary-Copy: Evaluation

- Advantages compared to ROWA
  - No dependence on availability of all secondary nodes
  - Better throughput, less conflicts
  - Failure of primary copy $\leadsto$ one of the secondary copies can become primary copy
- Disadvantages
  - Read consistency not granted
    - ★ Often acceptable, because state of secondary copies is consistent regarding previous point in time ("‘snapshot semantics"’)
    - ★ Read-consistency can be implemented by requesting read locks from primary copy $\leadsto$ decreases advantage of local reads
  - Network fragmentation: cut-off subnet without primary copy can not continue, though it may have greater number of nodes

Consensus Approaches

- To avoid problems in case of network fragmentation and dependence on one centralized node
- Basic idea: update can be processed, if a node gets necessary if majority of nodes with copies agree
- Based on voting - overall number of possible votes: quorum Q
  - Required number of votes for read operations: read quorum Q_R
  - Required number of votes for update operations: update quorum Q_U
- Quorum-Overlap-Rules: grants 1-Copy-Serializablity
  1. $Q_R + Q_U > Q$
  2. $Q_U + Q_U > Q$

Consensus Approaches: Alternatives

- **Weighted vs. equal votes**
  - Equal votes: each node has a vote with $weight = 1$
  - Weighted votes: nodes get assigned different weight, allows decreased message number by first asking nodes great weight
- **Static vs. dynamic voting**
  - Static: Q_R and Q_U do not change
  - Dynamic: Q_R and Q_U are adjusted to new Q in case of node failures or network fragmentation
- **Tree Quorum**
  - Nodes are hierarchically structured and separate quorums are defined for each level

Majority Consensus

- "'Origin"' of all consensus approaches: equal votes and static
- Works in case of node failures and network fragmentation
- Supports synchronization based on timestamps (without locks)
- Basic principles:
  - Communication along logical ring of nodes $\rightsquigarrow$ request and decisions are passed on from node to node
  - Quorum-Rules grant consistency in case of concurrent conflicting operations
  - With equal votes: $Q_U > Q/2$, i.e. majority of nodes in ring has to agree to operation

Majority Consensus: Procedure

1. **Origin node** of operation/TXN performs update locally and passes on changed objects with update timestamp to next node in ring
2. **Other nodes** vote
   - **reject**, if there is a timestamp conflict $\rightsquigarrow$ abort message is sent back to previous nodes, TXN is restarted at origin node
   - **okay**, if there is no conflict, request is marked *pending* until final decision
   - **pass**, if there is no timestamp conflict but a conflict with a concurrent update which is also *pending*
3. **Last node** which votes okay and by that satisfies quorum rule passes on commit to all nodes (backward and forward), update is propagated to all further nodes, pending updates are finalized on all previous nodes

Data Patches

- Problem: resolution of inconsistencies after node failures/ network fragmentation
- Idea: application-specific rules designed during database design $\rightarrow$ individual rules for each relation
- Rules:
  - Tuple insert rules: to add new tuples (keep, remove, notify, program)
  - Tuple integration rules: merge values of independently changed tuples with the same key (latest, primary, arithmetic,notify, program)

Snapshot Replication

- Snapshots: define (remote) materialized view on master-table
- Similar to primary copy, but also allows operations (filters, projection, aggregation, etc.) as part of view definition
- Synchronization of view by explicit **refresh** (manual or frequent)
- Handling of updates:
  - **Read-only snapshots**: updates only on master table
  - **Updateable views**: requires conflict resolution with master table (e.g. by rules, triggers)

Epidemic Replication

- Updates possible on every node
- Asynchronous forwarding of updates to "‘neighbours"’ using version information (timestamps)
- E.g. possible with Lotus Notes

Lazy Replication

- Updates on all nodes possibles
- Each node can start separate TXNs to perform updates on other nodes asynchronously
- Requires (application-specific) conflict resolution strategies
- Works in mobile scenarios, where node can connect/disconnect to/from the network

Part IV

Parallel DBS

Overview

- Foundations
- Parallel Query Processing

Foundations

- Parallel Database Systems (PDBS):
  - DB-Processing on Parallel Hardware Architectures
  - Goal: performance improvement by using parallel processors
- General overview of approaches:
  - Inter-TXN parallelization : simultaneous execution of independent parallel txns $\rightsquigarrow$ improved throughput
  - Intra-TXN parallelization : simultaneous execution processing of operations within one txn $\rightsquigarrow$ improved response time

Speedup

- **Speedup**: Measure for performance improvement of a IT system through optimization
- For parallelization: Response Time (RT) speedup by using n processors

$$\text{RT Speedup}(n) = \frac{\text{RT for sequential processing with 1 processor}}{\text{RT for parallel processing on } n \text{ Processors}}$$

- According to Amdahl's Law optimization improvement limited by fraction $0 \leq F_{\text{opt}} \leq 1$ of operations which can be optimized by parallelized execution

$$\text{RT Speedup} = \frac{1}{(1 - F_{\text{opt}}) + \frac{F_{\text{opt}}}{Speedup_{\text{opt}}}}$$

Speedup

- Speedup furthermore limited by
  - Overhead for coordination and communication for parallel execution
  - Interferences between parallel executions through shared ressource and locks
  - response time depending on slowest execution thread (non-equal distribution is called Skew: Processing Skew and Data Skew)
- Speedup limitation: there is a limit number n_{max} of processors form where on more processors do not improve performance or performance even declines

Scaleup

- Speedup: increase processor number to improve response time for same problem size
- **Scaleup**: linear growth of number of processors with growing problem size (e.g. more users, more data, etc.)
- Response Time Scaleup:
  - Ratio of the Reponse Time for n processors and n times DB size compared to original problem with 1 processor
  - Goal: Response Time Scaleup = 1
- Throughput Scaleup:
  - Ratio of TXN using n Prozessoren compared to solution with 1 processor
  - Goal: Throughput Scaleup = n

Architectures of PDBMS

- Shared Everything $\rightarrow$
  - Typical architecture: DBMS support for multi-processor computers
  - Ideal for Response Time Speedup
- Shared Disk $\rightarrow$
  - DBMS support for tightly connected (e.g. fast network) nodes with shared disk access
  - Good for Response Time and Scalabilty
  - Requires: synchronization of disk accesses
- Shared Nothing $\rightarrow$
  - Connected nodes with their own disks
  - Ideal for Scalability
  - Requires thoughtful data fragmentation

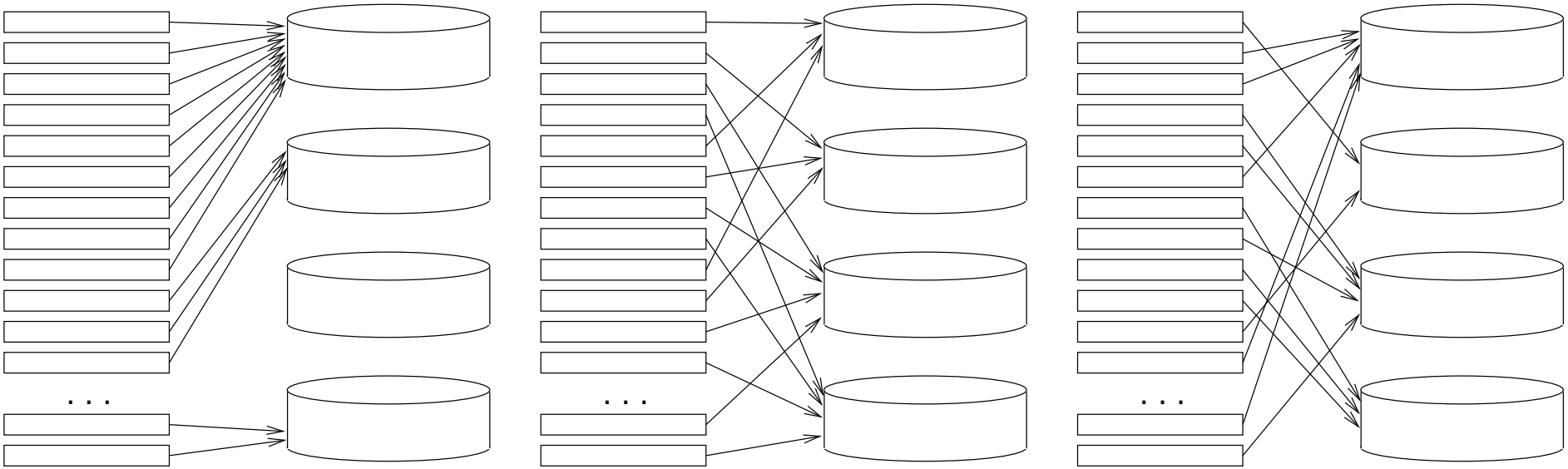
Fragmentation in PDBMS

- Goal: use of horizontal fragmentation to avoid data skew
  - Perform part of operation on equal size fragments
  - Less data skew
    $\rightarrow$ less processing skew
    $\rightarrow$ optimal parallel execution
    $\rightarrow$ optimal speedup
- Fixed or dynamic assignment of processors to fragments possible

Fragmentation in PDBMS /2

- Approaches:
  - **Range-based Fragmentation:**
    - ⋆ Assignment of tuples to disk based on pre-defined or dynamic range specifications for relation attributes
    - ⋆ Complex task to define ranges that minimize data and processing skew
  - **Hash-based Fragmentation:**
    - ⋆ Assignment of tuples to disk based on hash function on relation attributes
    - ⋆ More overhead for range queries
  - **Round Robin Fragmentation:**
    - ⋆ Assignment of tuples to disk upon creation: record i is assigned to disk $(i \mod M) + 1$ for M disks
    - ⋆ Avoids skew, but no support for exact match or range queries

Fragmentation in PDBMS /2



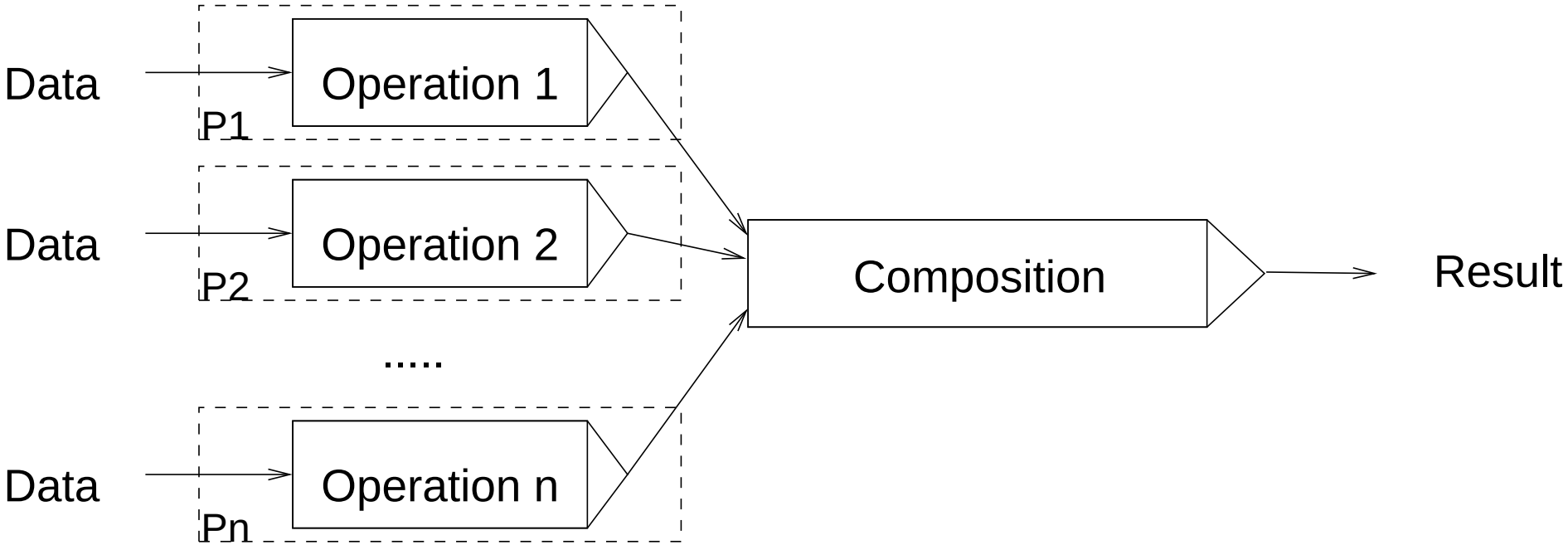
Parallel Query Processing

- Intra-Query Parallelization
- Intra-Operation Parallelization
  - Unary operations (Selection, Projection, Aggregation)
  - Sorting
  - Joins
- Processor Allocation

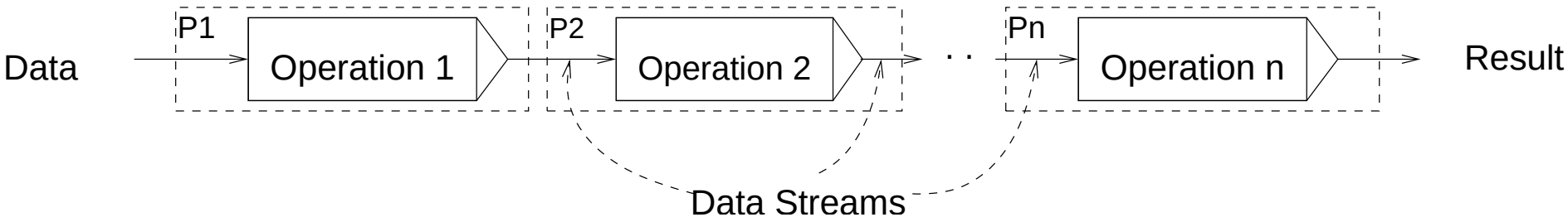
Intra-Query Parallelization

- Independent Parallelization
  - Parallel execution of independent parts of a query, e.g. multi-way joins, seperate execution threads below a union, etc.
- Pipelining Parallelization
  - Pipelining: processed data is considered as a data stream through sequence of operations (path from base relation to top of query plan tree)
  - Operations are executed by different processors with incoming data from processor handling lower operations in the query plan
- Intra-Operator Parallelization
  - Parallel processing of parts of one operation, e.g. selections from fragments, hash-based problem decomposition for join operations, etc.

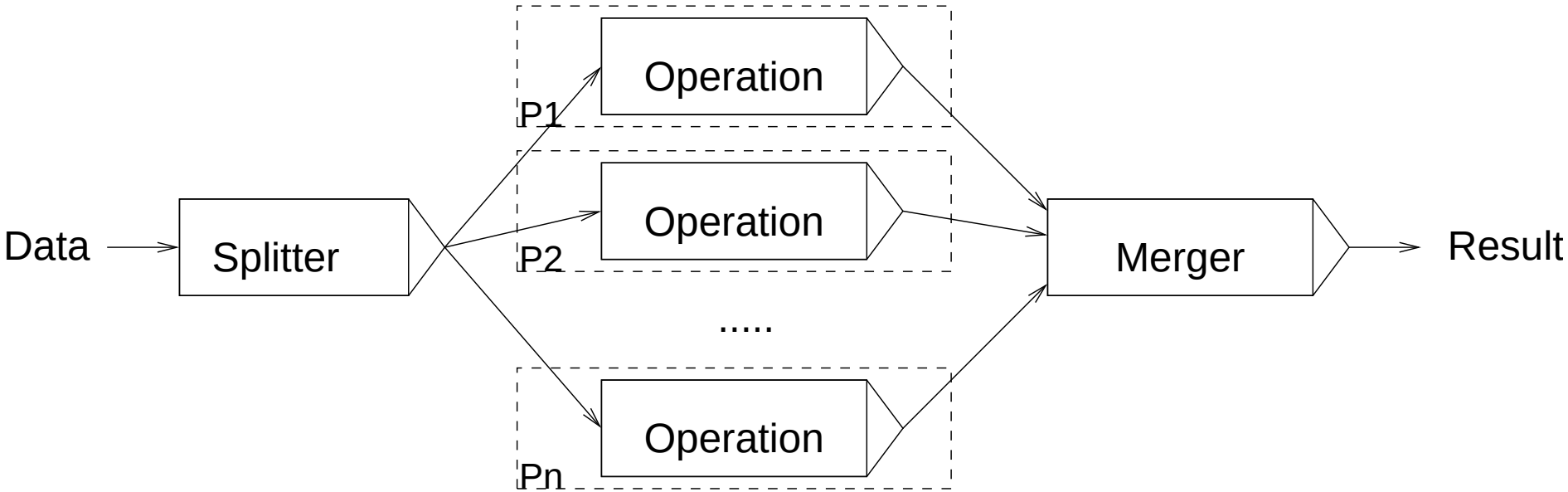
Independent Parallelization



Pipelining Parallelization



Intra-Operator Parallelization



Parallelization of unary Operations

- Selection:
  $r = \bigcup_i r_i \quad \Rightarrow \quad \sigma_P(r) = \bigcup_i \sigma_P(r_i)$
- Projection without duplicate elimination: dito
- Duplicate elimination: using sorting ($\rightarrow$)
- Aggregate functions:
  - $\mathbf{min}(r.\text{Attr}) = \mathbf{min}(\mathbf{min}(r_1.\text{Attr}), \ldots, \mathbf{min}(r_n.\text{Attr}))$
  - $\mathbf{max}(r.\text{Attr}) = \mathbf{max}(\mathbf{max}(r_1.\text{Attr}), \ldots, \mathbf{max}(r_n.\text{Attr}))$

Parallelization of Unary Operations /2

- Aggregate functions (if no duplicate elimination necessary):
  - $\texttt{count}(r.\text{Attr}) = \sum_i \texttt{count}(r_i.\text{Attr})$
  - $\texttt{sum}(r.\text{Attr}) = \sum_i \texttt{sum}(r_i.\text{Attr})$
  - $\texttt{avg}(r.\text{Attr}) = \texttt{sum}(r.\text{Attr}) \,/\, \texttt{count}(r.\text{Attr})$

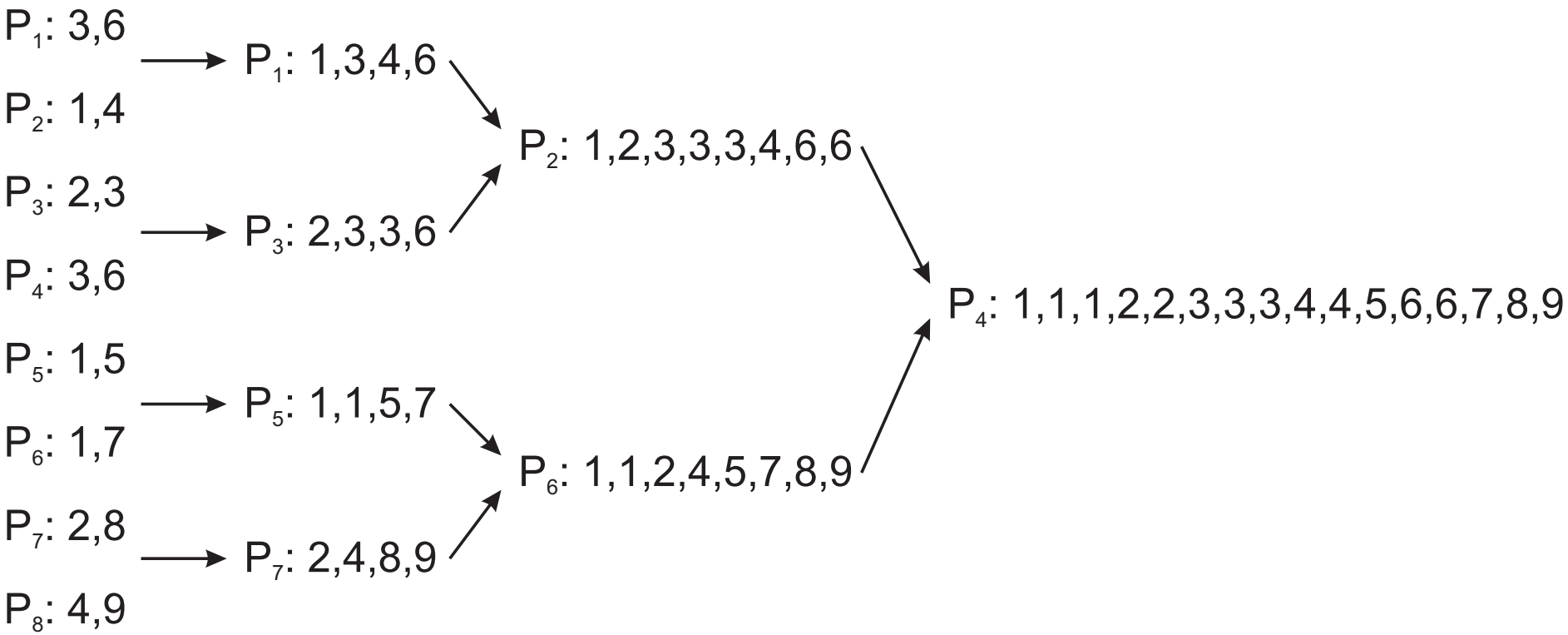
Parallel Sorting

- Classification regarding number of in- and output streams:
  - 1:1, 1:many, many:1, many:many
- Requirements for (?:many):
  - Partial result on each node is sorted
  - Complete final result can be achieved by simple concatenation of partial results (no further merging necessary)

Parallel Binary Merge Sort

- Many:1 approach
- Processing:
  - Phase 1: fragements are sorted locally on each node (Quicksort, External Merge Sort)
  - Phase 2: merging two partial results on one node at a time until all intermediate results are merged in one final result
- Merging can also be performed in parallel and even be pipelined

Parallel Binary Merge Sort /2



Block Bitonic Sort

- Many:many-approach applying fixed number of processors (e.g. those managing a fragmented relation)
- 2 Phases

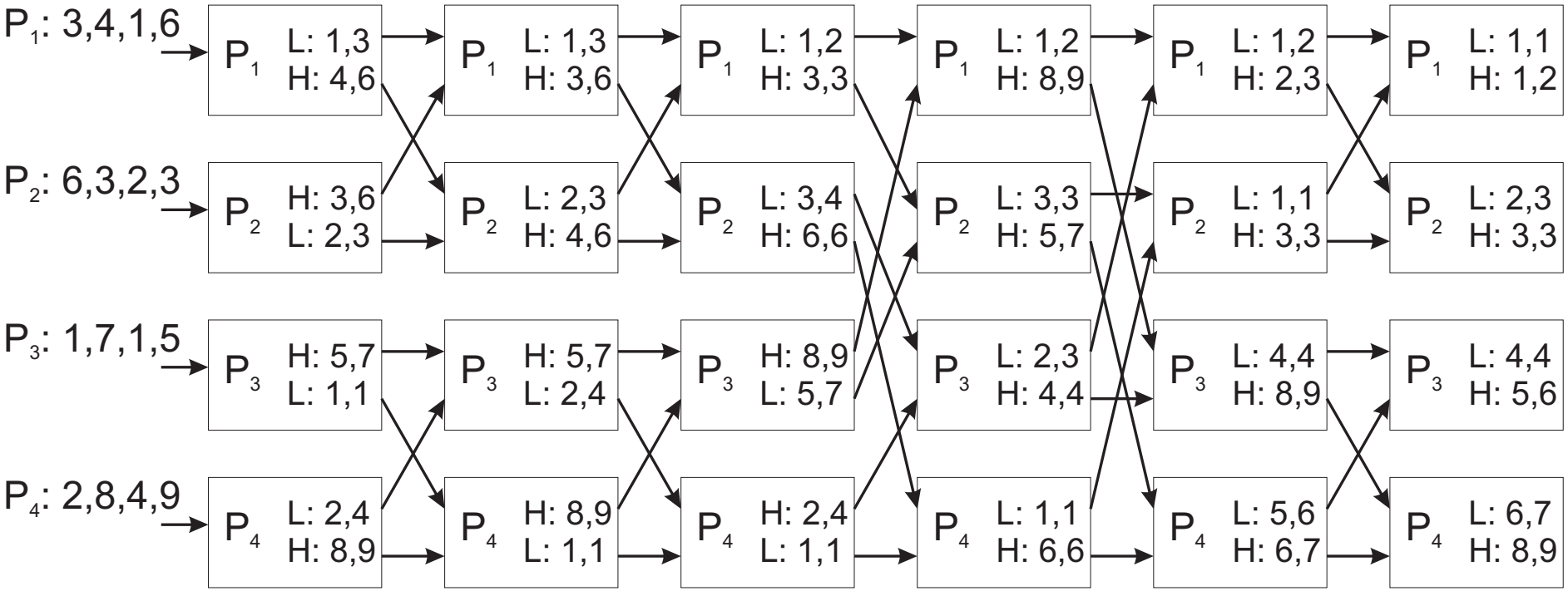
1. Sort-Split-Ship
   (a) Sort fragments on each node locally
   (b) Split sorted fragments in two parts of equal size
      - ⋆ every value in one (lower) sub-fragment $\leq$ every value in the other (higher) sub-fragment
   (c) Ship sub-fragments to other nodes according to predefined scheme

Block Bitonic Sort /2

2. 2-Way-Merge Split
   (a) On arrival of two fragments: 2-Way-Merge to get one sorted fragment
   (b) Split and ship fragments according to 1(b) until
      (1) every node P_i has a sorted fragment and
      (2) every value in fragment at $P_i \leq$ every value in fragment at P_j for $i \leq j$

- Key point is shipping scheme, which is fixed for a certain number of nodes n (see following example)
- Number of necessary steps: $\frac{1}{2} \log 2n(\log 2n + 1)$ for n nodes

Block Bitonic Sort /3



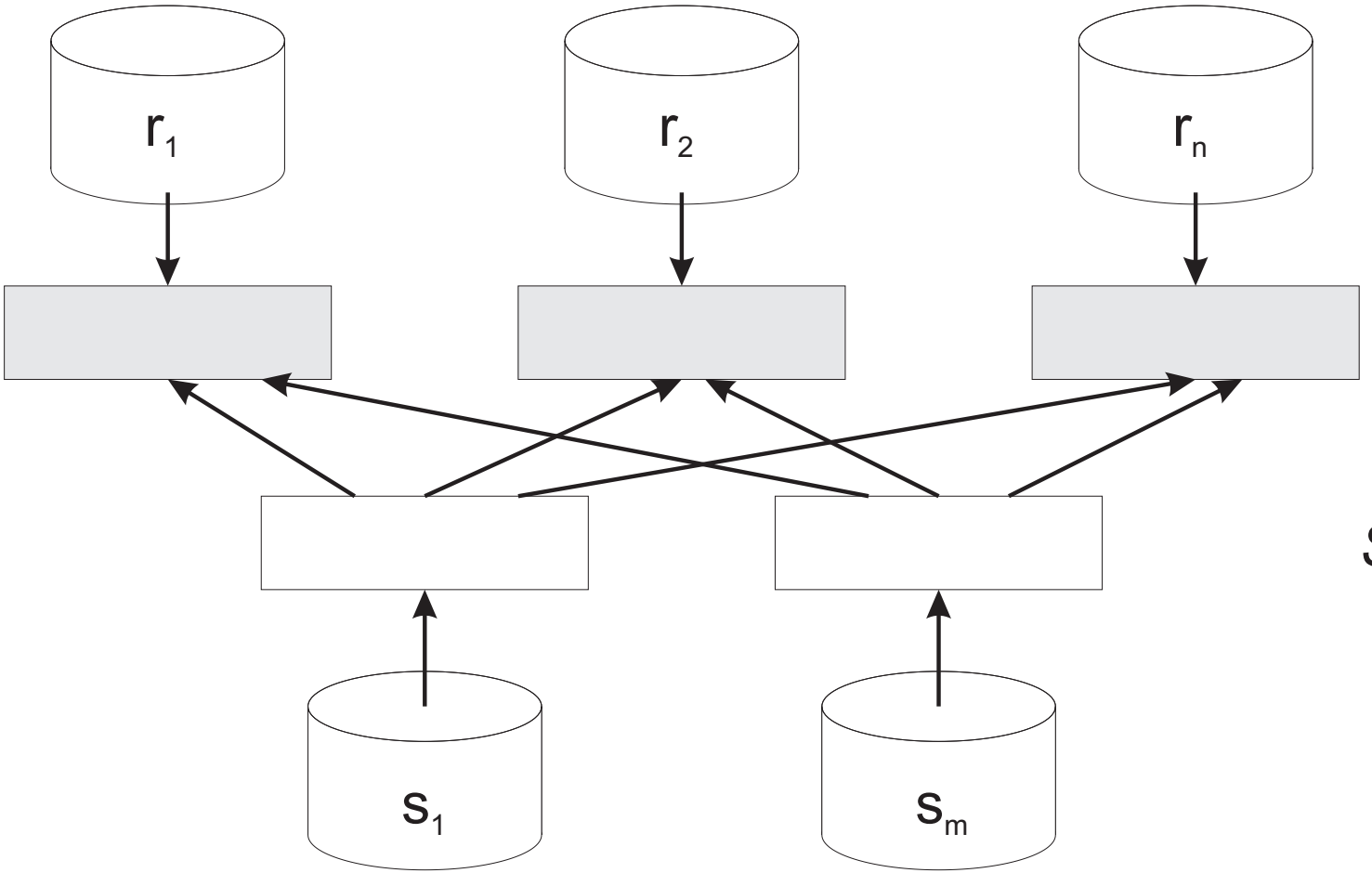
Parallel Join-Processing

- Join-Processing on Multiple Processors supported by 2 main approaches
- Dynamic Replication
  - Replication (transfer) of the smaller relation r toeach join node
  - Local execution of partial joins on each processor
  - Full result by union of partial results
- Dynamic Partitioning
  - Partition tupels to (ideal) same size partitions on each node
  - Distribution: hash function h or range partitioning

Dynamic Replication

1. Assumption: relations S and R are fragmented and stored accross several nodes
2. Coordinator: initiate join on all nodes R_i (join nodes) and S_j (data nodes) ($i = 1 \ldots n, j = 1 \ldots m$)
3. Scan Phase: parallel on each S-node:
   read and transfer s_j to each R_i
4. Join-Phase: parallel on each R-node with partition r_i:
   - $s := \bigcup s_j$
   - Compure $t_i := r_i \bowtie s$
   - send t_i to coordinator
5. Coordinator: receive and merge all t_i (union)

Dynamic Replication /2



Join-Knoten (R-Knoten)

S-Datenknoten

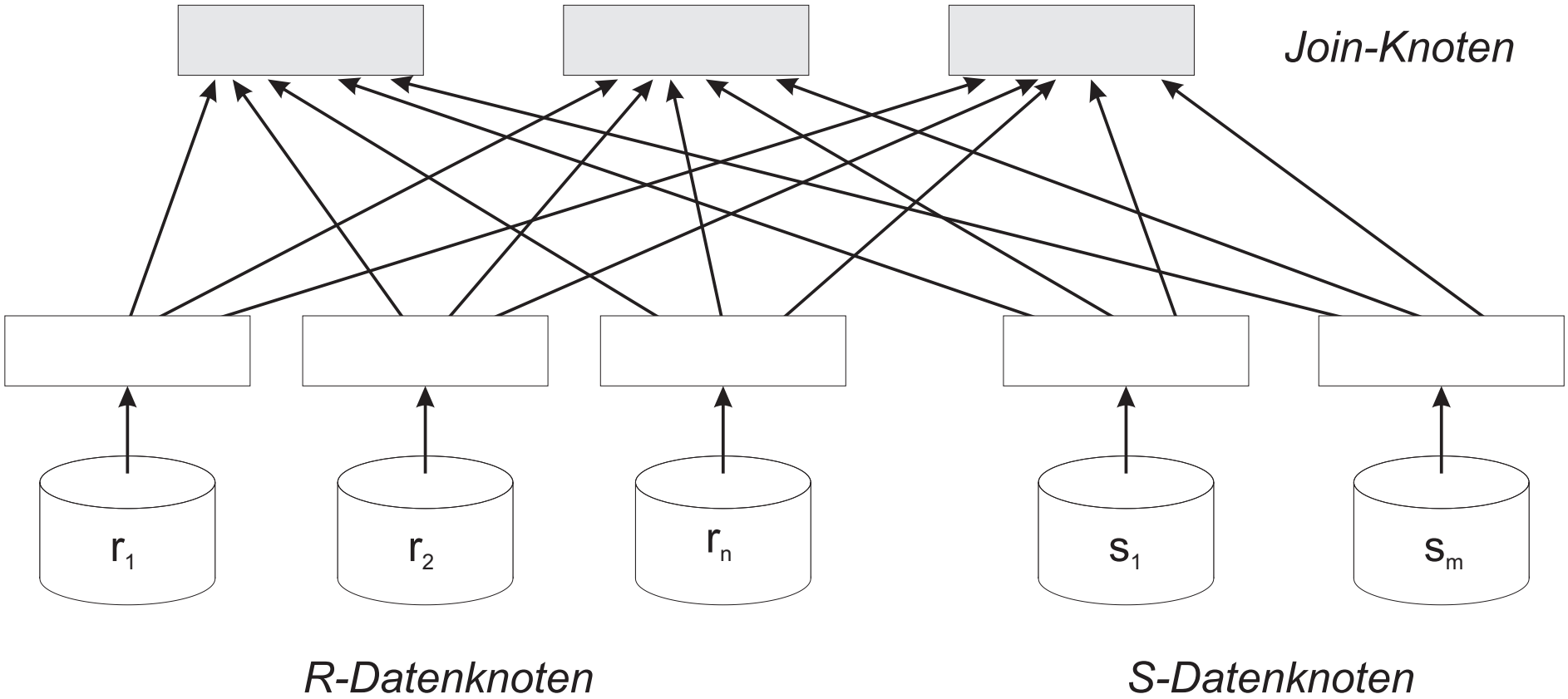
Dynamic Partitioning

1. Coordinator: initiate joins on all R, S and (possibly separate) join nodes
2. Scan Phase
   - parallel on each R-node:
     read and transfer each tuple of r_i to **resonsible** join node
   - parallel on each$S$-node:
     read and transfer each tuple of s_i to **resonsible** join node
3. Responsibel node is computed by hash or range function $\rightarrow$ avoid or handle skew

Dynamic Partitioning /2

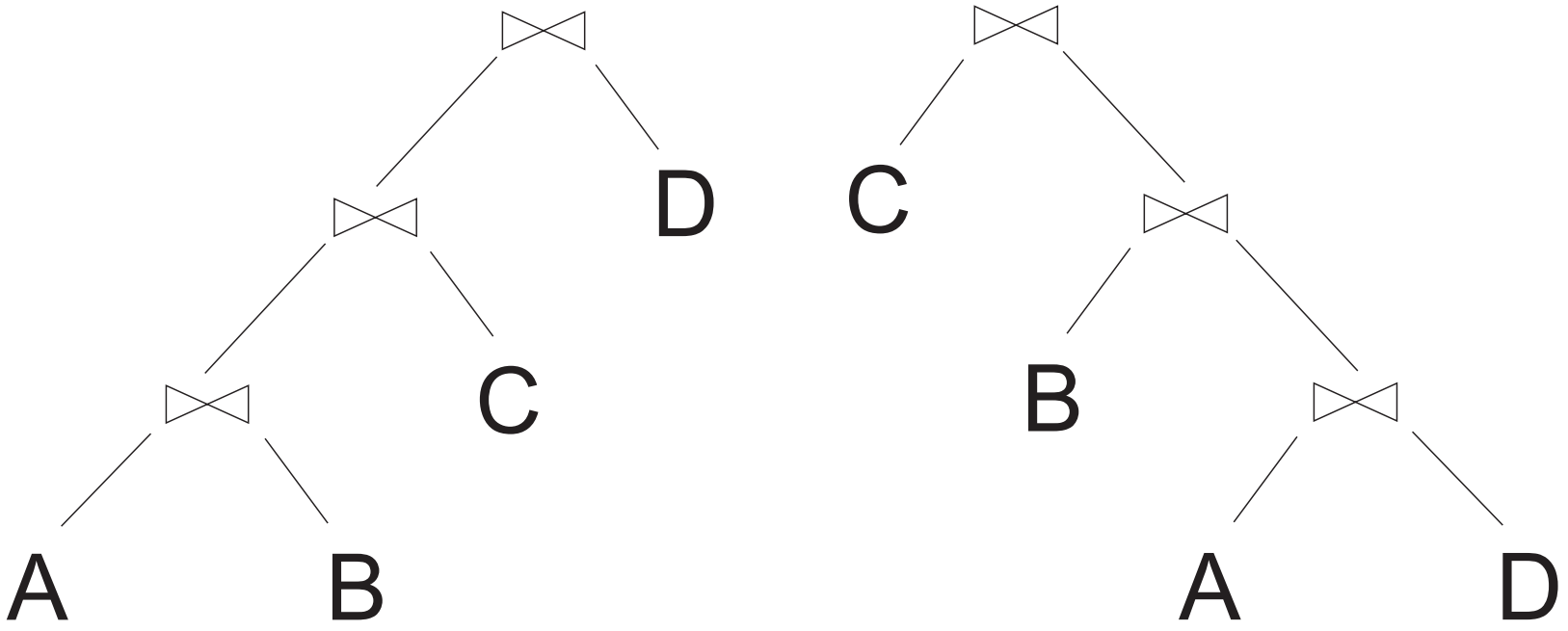
3. Join Phase: parallel on each join node $k (k = 1 \ldots p)$
   - $r'_k := \bigcup r_{ik}$ (set of r-tuples received at node k)
   - $s'_k := \bigcup s_{ik}$ (set of r-tuples received at node k)
   - compute $t_k := r'_k \bowtie s'_k$
   - transfer t_k to coordinator
4. Coordinator: receive and merge t_k

Dynamische Partitionierung /3



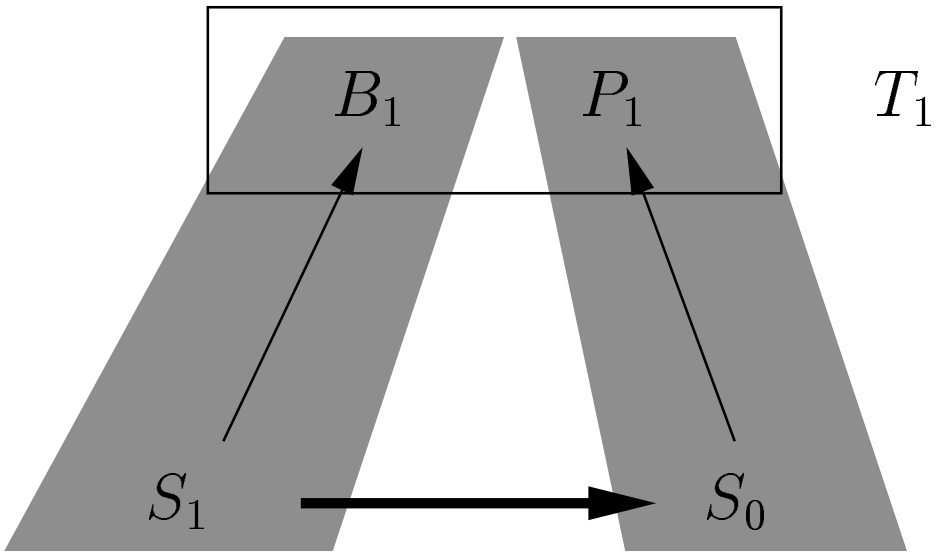
Multi-Way-Joins

- Variants considered: left and right-deep trees



Multi-Way-Joins /2

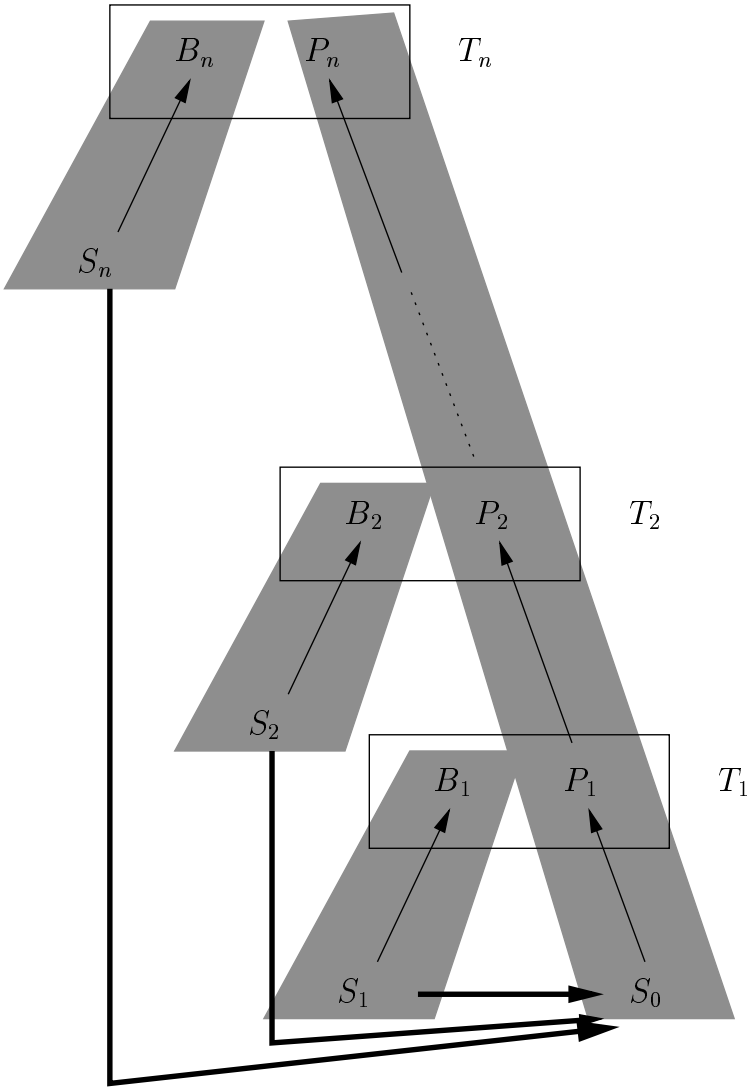
- Assumption: single join is executed as a Hash Join consisting of 2 pahse
  (1) **B**uild Phase: build hash table for first join relation
  (2) **P**robe-Phase: check whether there are join partners from second relation
- Dependency: probe phase can only start after build phase has finished



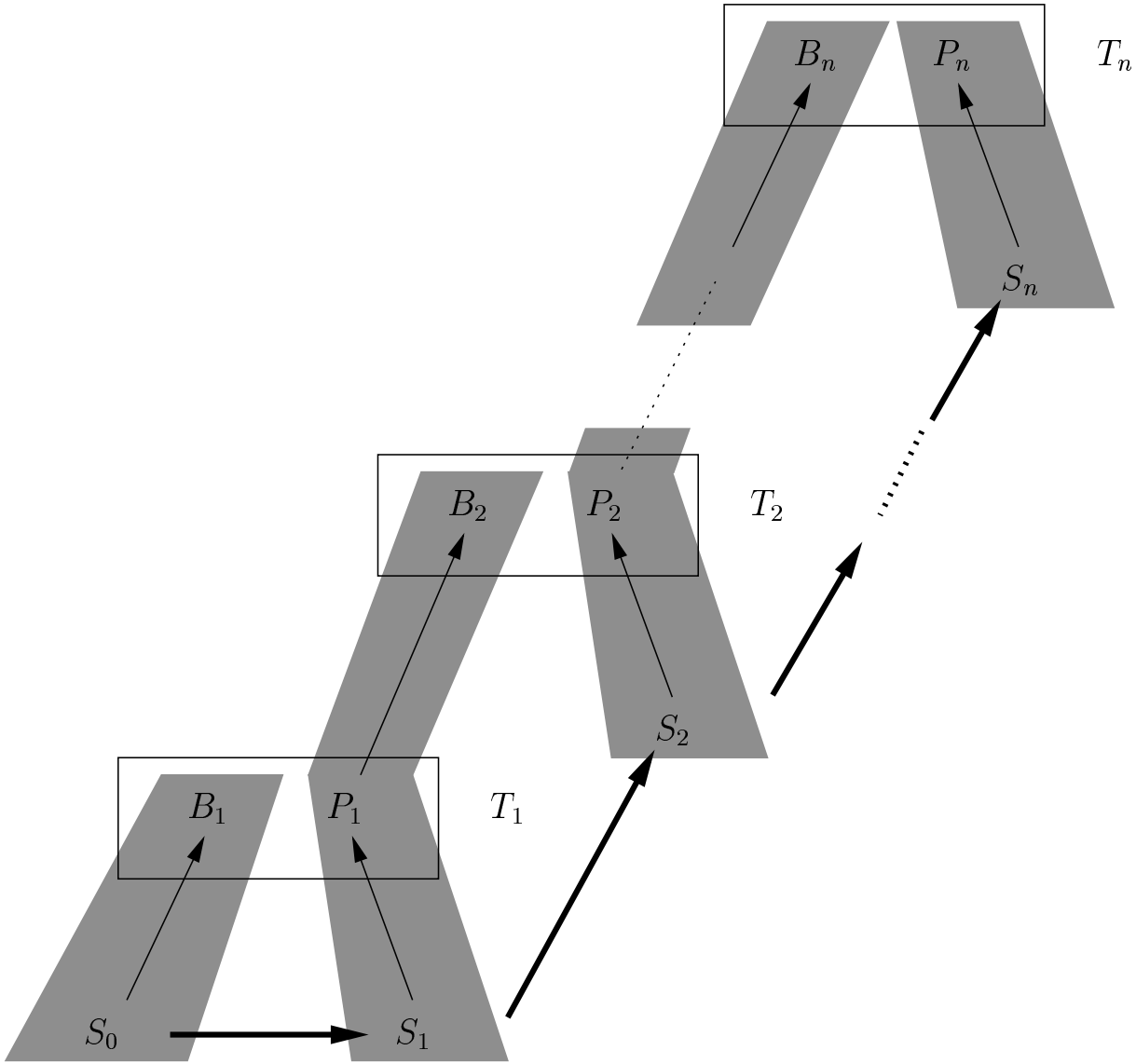
Multi-Way-Joins /3

- Right Deep Trees
  - Perform build phases in parallel for all joins by independent execution
  - Perform probe phases in parallel by pipelined execution
- Left Deep Trees
  - Perform build phase of each join in parallel with probe phase of previous join by pipelined execution

Join: Right Deep Tree



Join: Left Deep Tree



Multi-Way-Joins /4

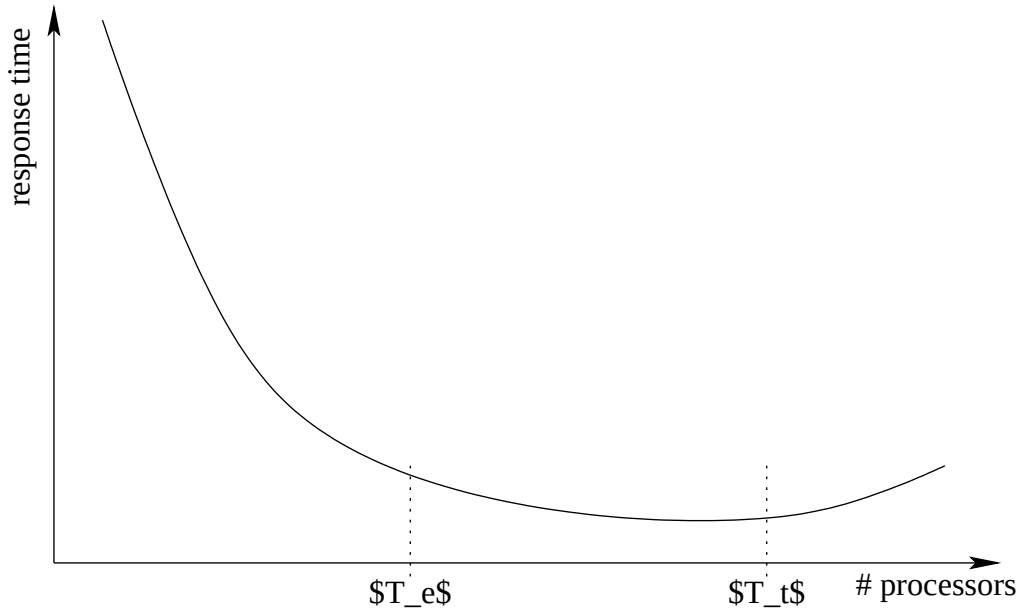
- Problem: multi-way-join processing with high requirements regarding memory, e.g. for hash tables
- Solution:
  - Scheduling of joins, e.g. breaking up deep trees to process only part of the join that fits in memory
  - Static or dynamic decission about segmentation

Processor Allocation

- Assignment of operations (for now, join operations) to processors
- *2-Phase Optimization*
  1. Find a optimal plan based on costs
  2. Assign execution order and processor allocation
- *1-Phase Optimization*
  - perform both tasks as one step (part of query optimization)
- Relevant aspects
  - Dependencies between operations: e.g. consider join order to avoid waiting
  - Number of processors per join: increasing number does not yield linear improvement of processing time because of initialisation and merging, which can not be parallelized

Processor Allocation /2

- Thresholds for number of processors: minimal response time point T_t , Execution efficiency point T_e



- Execution efficiency for n processors

$$\text{Execution efficiency} = \frac{\text{Response time for 1 processor}}{n \cdot \text{Response time for } n \text{ processors}}$$

Processor Allocation: Strategies

- Strategy 1: sequential execution of the joins with T_t processors
- Strategy 2: Time Efficiency Point
  - For each join, which can be started, allocate T processors

$$T = c \cdot T_e + (1 - c) \cdot T_t \quad 0 \leq c \leq 1$$

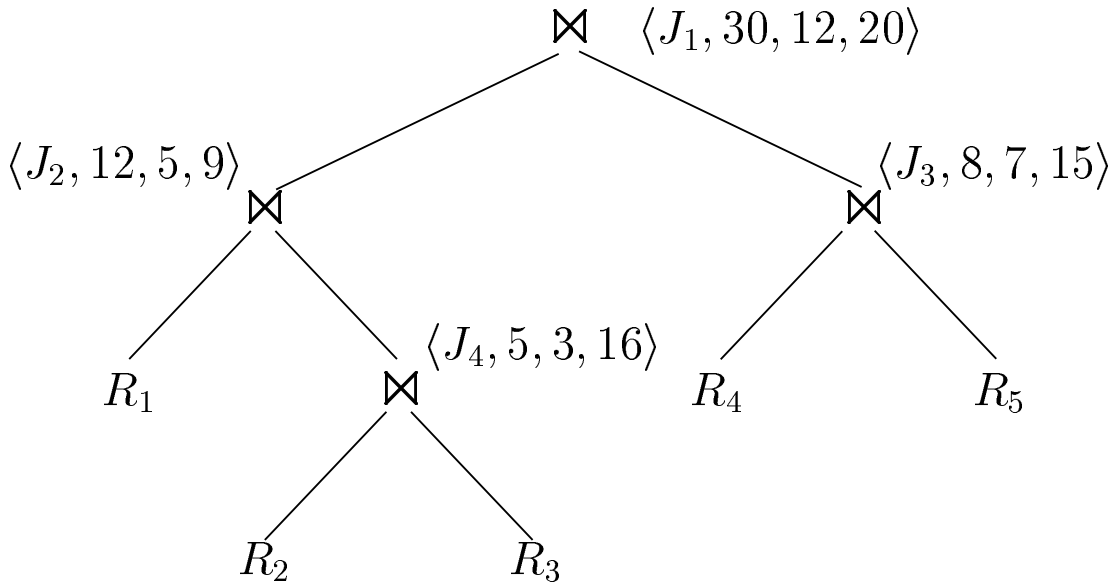
  - Execution for N available processors and each join J
    1. $T(J) = c \cdot T_e(J) + (1 - c) \cdot T_t(J)$
    2. **if** $T(J) \leq N$ **then**
       $\{$ allocate $T(J)$ processors; $N := N - T(J)$ $\}$

Processor Allocation: Strategies /2

- Strategy 3: Synchronous Top Down Allocation
  - Presumption: costs for each join are known (costs for execution of query plan subtree)
  - Processing: Top-down traversal of the join tree and allocation of processors
    1. Allocate $\min\{N, T_t(\text{root})\}$ processors for the join tree root: minimal response time for last join;
    2. For join J is N_J the number of available processors for that join
       $\rightarrow$ if J has only one child J_1, allocate $\min\{N_J, T_t(J_1)\}$ processors for J_1
       $\rightarrow$ ff J has to childrene J_1 and J_2, split N_J into two processor sets for J_1 and J_2 with size $\sim$ costs
    3. Replace J by J_1 and J_2 and continue allocation recursively

Processor-Allocation: Example

- 20 processors, $\langle J_i, \text{costs}, T_e, T_t \rangle$



- Sequential Execution: J_4: 16 $\rightarrow$ J_3: 15 $\rightarrow$ J_2: 9 $\rightarrow$ J_1: 20
- Time-Efficiency Point with $c = 0.5$: (J_4: 9, J_3: 11) $\rightarrow$ J_2: 7 $\rightarrow$ J_1:16
- Synchronous Top Down Allocation: J_1: 20, J_2-Subtree: 12 (J_2: 9), J_3: 8, J_4: 12