Distributed Data Management

Dr.-Ing. Eike Schallehn

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

2015/2016
Organization

- **Weekly lecture**
  - Teacher: Eike Schallehn (eike@iti.cs.uni-magdeburg.de)

- **Weekly exercises with two alternative time slots**
  - Starting in the third week of the lecture period
  - Teachers: Xiao Chen, Juliana Alves Pereira

- **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
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


German Literature

Part I

Introduction
Overview

- Motivation
- Recapitulation
- Classification of Multi-Processor DBMS
Centralized Data Management

- Application
- Application
- Application

DBMS

- Data manipulation
- Data definition
- TXN management
- ...

New requirements
- Support for de-centralized organization structures
- High availability
- High performance
- Scalability
Data Management in a Network

Network

Node

Node

Node

Node
Distributed Data Management

Network

Node

Node

Node
Distributed Data Management: Example

Network

Products Magdeburg, San Francisco
Customers Magdeburg

Products München, Magdeburg, Sydney
Sydney
Customers München

Products San Francisco, Sydney
Customers San Francisco, Sydney

Products Sydney

Customers Sydney

München

Magdeburg

San Francisco

Sydney

Magdeburg

München

Sydney
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
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
Who provides Transparency?

- **Application**
  - Different parts/modules of distributed application
  - Communication / data exchange using standard protocols (RPC, CORBA, HTTP, SOAP, ...)

- **DBMS**
  - Transparent SQL-access to data on remote DB-instances
  - Requires query decomposition, transaction coordination, replication

- **Operating system**
  - Operating systems provides network transparency e.g. on file system level (NFS) or through standard protocols (TCP/IP)
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 → 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
  - Application components communicate for purpose of data exchange (distribution on application level)
- Distributed DBS
  - Distribution solely realized on the DBS-level
Differentiation /2

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

  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)
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.
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 /1

- 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
Codd’s 9 Rules for DBMS /2

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 logical unit

8 **Synchronization:** parallel accesses to the database are managed by the system

9 **Data backups:** the system provides functionality to grant long-term accessibility even in case of failures
3 Level Schema Architecture
3 Level Schema Architecture /2

- Important concept of DBMS
- Provides
  - transparency, i.e. non-visibility, of storage implementation details
  - ease of use
  - decreased application maintenance efforts
  - conceptual foundation for standards
  - portability
- Describes abstraction steps:
  - Logical data independence
  - Physical data independence
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
- ...

![Diagram of Client Server Architecture]
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
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
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 $\sigma$
- Projection $\pi$
- Cartesian product (cross product) $\times$
- Set union $\cup$
- Set difference $-$
- Rename $\beta$

Non-primitive operations
- Natural Join $\Join$
- $\theta$-Join and Equi-Join $\Join_\varphi$
- Semi-Join $\bowtie$
- Outer-Joins $= \times$
- Set intersection $\cap$
- ...
Notation for Relations and Tuples

- If $R$ denotes a relation schema (set of attributes), then 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 $\sigma$

Select tuples based on predicate or complex condition

<table>
<thead>
<tr>
<th>PROJECT</th>
<th>PNAME</th>
<th>PNUMBER</th>
<th>PLOCATION</th>
<th>DNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductX</td>
<td>1</td>
<td>Bellaire</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ProductY</td>
<td>2</td>
<td>Sugarland</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ProductZ</td>
<td>3</td>
<td>Houston</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Computerization</td>
<td>10</td>
<td>Stafford</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Reorganization</td>
<td>20</td>
<td>Houston</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Newbenefits</td>
<td>30</td>
<td>Stafford</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>PNAME</th>
<th>PNUMBER</th>
<th>PLOCATION</th>
<th>DNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computerization</td>
<td>10</td>
<td>Stafford</td>
<td>4</td>
</tr>
<tr>
<td>Newbenefits</td>
<td>30</td>
<td>Stafford</td>
<td>4</td>
</tr>
</tbody>
</table>
The Projection Operation $\pi$

Project to set of attributes - remove duplicates if necessary

<table>
<thead>
<tr>
<th>PROJECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNAME</td>
</tr>
<tr>
<td>ProductX</td>
</tr>
<tr>
<td>ProductY</td>
</tr>
<tr>
<td>ProductZ</td>
</tr>
<tr>
<td>Computerization</td>
</tr>
<tr>
<td>Reorganization</td>
</tr>
<tr>
<td>Newbenefits</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>PLOCATION</th>
<th>DNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellaire</td>
<td>5</td>
</tr>
<tr>
<td>Sugarland</td>
<td>5</td>
</tr>
<tr>
<td>Houston</td>
<td>5</td>
</tr>
<tr>
<td>Stafford</td>
<td>4</td>
</tr>
<tr>
<td>Houston</td>
<td>1</td>
</tr>
</tbody>
</table>
Cartesian or cross product \( \times \)

Create all possible combinations of tuples from the two input relations

\[
\begin{array}{cc}
\text{R} & \\
\hline
\text{A} & \text{B} \\
1 & 2 \\
3 & 4 \\
\end{array}
\quad
\begin{array}{cc}
\text{S} & \\
\hline
\text{C} & \text{D} & \text{E} \\
5 & 6 & 7 \\
8 & 9 & 10 \\
11 & 12 & 13 \\
\end{array}
\]

\[r(R) \times r(S)\]
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 ⊲⊳

- 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
  $R \cap S = \emptyset \implies r(R) \bowtie r(S) = r(R) \times r(S)$

- Can be expressed as combination of $\pi$, $\sigma$ and $\times$
  $r(R) \bowtie r(S) = \pi_{R \cup S}(\sigma_{\land a \in R \cap S} t(R.a) = t(S.a)(r(R) \times r(S)))$
The Natural Join Operation \( \bowtie /2 \)

\[
\begin{array}{c|c}
R & A & B \\
\hline
1 & 2 \\
3 & 4 \\
5 & 6 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
S & A & B & C & D \\
\hline
4 & 5 & 6 \\
6 & 7 & 8 \\
8 & 9 & 10 \\
\end{array}
\]

\[
r(R) \bowtie r(S)
\]
The Semi-Join Operation

- Results all tuples from one relation having a (natural) join partner in the other relation

\[ r(R) \Join r(S) = \pi_R(r(R) \bowtie r(S)) \]

<table>
<thead>
<tr>
<th>PERSON</th>
<th>PID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1273</td>
<td>Dylan</td>
</tr>
<tr>
<td></td>
<td>2244</td>
<td>Cohen</td>
</tr>
<tr>
<td></td>
<td>3456</td>
<td>Reed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CAR</th>
<th>PID</th>
<th>BRAND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1273</td>
<td>Cadillac</td>
</tr>
<tr>
<td></td>
<td>1273</td>
<td>VW Beetle</td>
</tr>
<tr>
<td></td>
<td>3456</td>
<td>Stutz Bearcat</td>
</tr>
</tbody>
</table>

\[ r(PERSON) \Join r(CAR) \]
Other Join Operations

- **Conditional join**: join condition $\varphi$ is explicitly specified
  \[ r(R) \bowtie_{\varphi} r(S) = \sigma_{\varphi}(r(R) \times r(S)) \]

- **$\theta$-Join**: special conditional join, where $\varphi$ is a single predicate of
  the form $a \theta b$ with $a \in R$, $b \in S$, and $\theta \in \{=, \neq, >, <, \leq, \geq, \ldots \}$

- **Equi-Join**: special $\theta$-Join where $\theta$ 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).
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

![Diagram](image)

- Some significant differences exist
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 $\sigma$

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

```sql
SELECT *
FROM EMPLOYEE
WHERE DNO=5 AND SALARY>30000
```
SQL: Projection \( \pi \)

\[ \pi_{LNAME,FNAME}(r(EMPLOYEE)) \]

**SELECT**  \( \text{LNAME, FNAME} \)  
**FROM**  \( \text{EMPLOYEE} \)

- Difference to RM: does not remove duplicates
- Requires additional  \( \text{DISTINCT} \)

**SELECT**  \( \text{DISTINCT LNAME, FNAME} \)  
**FROM**  \( \text{EMPLOYEE} \)
SQL: Cartesian Product

\[ r(\text{EMPLOYEE}) \times r(\text{PROJECT}) \]

```sql
SELECT * 
FROM EMPLOYEE, PROJECT
```
SQL: Natural Join

\[ r(\text{DEPARTMENT}) \Join r(\text{DEPARTMENT\_LOCATIONS}) \]

```
SELECT *
FROM DEPARTMENT
    NATURAL JOIN DEPARTMENT\_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
```
**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 ...
...

...
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, ...
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

- **Atomicity** means that a transaction can not be interrupted or performed only partially
  - TXN is performed in its entirety or not at all

- **Consistency** to preserve data integrity
  - A TXN starts from a consistent database state and ends with a consistent database state

- **Isolation**
  - Result of a TXN must be independent of other possibly running parallel TXNs

- **Durability 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
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

\[ R_1 \quad \text{Reflexivity} \quad \text{if } X \supseteq Y \quad \Rightarrow \quad X \rightarrow Y \]

\[ R_2 \quad \text{Accumulation} \quad \{ X \rightarrow Y \} \quad \Rightarrow \quad XZ \rightarrow YZ \]

\[ R_3 \quad \text{Transitivity} \quad \{ X \rightarrow Y, Y \rightarrow Z \} \quad \Rightarrow \quad X \rightarrow Z \]

\[ R_4 \quad \text{Decomposition} \quad \{ X \rightarrow YZ \} \quad \Rightarrow \quad X \rightarrow Y \]

\[ R_5 \quad \text{Unification} \quad \{ X \rightarrow Y, X \rightarrow Z \} \quad \Rightarrow \quad X \rightarrow YZ \]

\[ R_6 \quad \text{Pseudotransitivity} \quad \{ X \rightarrow Y, WY \rightarrow Z \} \quad \Rightarrow \quad WX \rightarrow Z \]

\( R_1 - R_3 \) 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 are 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
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
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
  - Integrated vs. federated architecture
  - Homogeneous vs. heterogeneous architecture
Criterion: Processor Functionality

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

![Diagram]

- Multi-Processor DBMS
  - Local Storage
    - Shared
    - Partitioned
  - Spatial Distribution
    - Shared
  - Processor Connection
    - Tight
    - Close
    - Loose
    - (Close)
    - Loose
  - Shared-Everything
  - Shared-Disk
    - Shared-Nothing
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 availability

- 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 ($\leq 20$)
  - 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

- Shared Main Memory
  - DBMS Buffer
  - CPU
    - Cache
  - CPU
    - Cache
  - CPU
    - Cache

- Shared Hard Disks
Simple realization of DBMS
Distribution transparency provided by operating system
Expensive synchronization
Extended implementation of query processing
- Distribution of DB across various nodes
- Distributed/parallel execution plans
- TXN management across participating nodes
- Management of catalog and replicas
Class: Shared-Disk

Highspeed Communication

Main Memory

DBMS-Buffer

CPU

Cache

Main Memory

DBMS-Buffer

CPU

Cache

Main Memory

DBMS-Buffer

CPU

Cache

gemeinsame Festplatten
Avoids physical data distribution
No distributed TXNs and query processing
Requires buffer invalidation
Criterion: Integrated vs. Federated DBS

- **Integrated:**
  - Shared database for all nodes → one conceptual schema
  - High distribution transparency: access to distributed DB via local DBMS
  - Requires co-operation of DBMS nodes → restricted autonomy

- **Federated:**
  - Nodes with own DB and own conceptual schema
  - Requires schema integration → global conceptual schema
  - High degree of autonomy of nodes
Centralized:

- Each node has global view on database (directly or via master)
- Central coordinator: initiator of query/transaction → knows all participating nodes
- Provides typical DBS properties (ACID, result completeness, etc.)
- Applications: distributed and parallel DBS
  - Limited availability, fault-tolerance, scalability
De-centralized:

- No global view on schema → 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, scalability
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Parallel DBS</th>
<th>Distributed DBS</th>
<th>Federated DBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>High TXN rates</td>
<td>↑</td>
<td>→↑</td>
<td>→</td>
</tr>
<tr>
<td>Intra-TXN-Parallelism</td>
<td>↑</td>
<td>→↑</td>
<td>↘→</td>
</tr>
<tr>
<td>Scalability</td>
<td>↑</td>
<td>→↑</td>
<td>→</td>
</tr>
<tr>
<td>Availability</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>Geogr. Distribution</td>
<td>↘</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>Node Autonomy</td>
<td>↘</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>DBS-Heterogeneity</td>
<td>↘</td>
<td>→</td>
<td>↘</td>
</tr>
<tr>
<td>Administration</td>
<td>→</td>
<td>→</td>
<td>↓</td>
</tr>
</tbody>
</table>
Part II

Distributed DBS Architecture
Overview

- Foundations of DDBS
- Catalog Management
- DDBS Design: Fragmentation
- Allocation and Replication
Architecture & Data Distribution

DBMS-Instance

Node

Network

Node

Node

Node

Node
Dimensions

Heterogeneity

Autonomy

Distribution

Centralized DBS

Client/Server-DBS

Distributed DBS
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
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
Distributed Transaction Management
- ACID properties must persist for distributed operations

Hardware independence
- Component DB processing on different hardware platforms

Operating system independence
- Component DB processing on different OS

Network independence
- DB processing using different network protocols

DBMS independence (ideal)
- Usage of different DBMS possible
Schema Architecture

- External Schema 1
- ... (omitted for brevity)
- External Schema N

- Local Conceptual Schema 1 (LCS)
- Local Conceptual Schema 2 (LCS)
- Local Conceptual Schema M (LCS)

- Local Internal Schema 1 (LIS)
- Local Internal Schema 2 (LIS)
- Local Internal Schema M (LIS)

- Global Conceptual Schema (GCS)
- Global Distribution Schema (GDS)
Global conceptual schema (GCS)
  - Logical structure of overall DB
  - Supported by all nodes
  - Ensures *transparency*

Global distribution schema (GDS)
  - Describes fragmentation, replication, allocation
System Architecture

<table>
<thead>
<tr>
<th>Global Query Processing</th>
<th>Global Catalog Management</th>
<th>Replica Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Recovery</td>
<td>Global Transaction Management</td>
<td>Global Synchronisation</td>
</tr>
</tbody>
</table>

"normal DBMS"

**global Component**

**local Component**
Catalog Management

- Catalog: collection of metadata (schema, statistics, access rights, etc.)
  - Local catalog
    - Identical to catalog of a centralized DBS
    - consists of LIS and LCS
  - Global catalog
    - 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, little space consumption
  - Disadvantages: request for each query, potential bottleneck, critical resource

- Replicated: full copy of global catalog stored on each node
  - Advantage: low communication overhead during queries, availability
  - Disadvantage: high overhead for updates

- Mix-form: cluster-catalog with centralized catalog for certain clusters of nodes
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

- **global Name**
- Birth site
- Catalog site
- Store site
- Store site
- Store site
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
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 autonomy, user-controlled invalidation of buffered catalog data, good performance
- Disadvantage: no uniform realization of global views
In Shared-Nothing-Systems (DDBS): definition of physical distribution of data

Impact:

- Communication efforts $\leadsto$ 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 \(\leadsto\) schema integration (Federated DBS)

- **Top Down**
  - GCS of local DB designed first
  - Distribution of schema to different nodes
  - *Distribution Design*
Distribution Design Tasks

- Node 1
  - R1
  - R2.1

- Node 2
  - R3
  - R4.1

- Node 3
  - R4.1
  - R4.2

Global Relation R

Fragments

Allocations
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
**Approach:**

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

<table>
<thead>
<tr>
<th>MNo</th>
<th>MName</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>Ian Curtis</td>
<td>SW Developer</td>
</tr>
<tr>
<td>M2</td>
<td>Levon Helm</td>
<td>Analyst</td>
</tr>
<tr>
<td>M3</td>
<td>Tom Verlaine</td>
<td>SW Developer</td>
</tr>
<tr>
<td>M4</td>
<td>Moe Tucker</td>
<td>Manager</td>
</tr>
<tr>
<td>M5</td>
<td>David Berman</td>
<td>HW-Developer</td>
</tr>
</tbody>
</table>

## Project

<table>
<thead>
<tr>
<th>PNr</th>
<th>PName</th>
<th>Budget</th>
<th>Loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>DB Development</td>
<td>200.000</td>
<td>MD</td>
</tr>
<tr>
<td>P2</td>
<td>Hardware Dev.</td>
<td>150.000</td>
<td>M</td>
</tr>
<tr>
<td>P3</td>
<td>Web-Design</td>
<td>100.000</td>
<td>MD</td>
</tr>
<tr>
<td>P4</td>
<td>Customizing</td>
<td>250.000</td>
<td>B</td>
</tr>
</tbody>
</table>

## Assignment

<table>
<thead>
<tr>
<th>MNr</th>
<th>PNr</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>P1</td>
<td>5</td>
</tr>
<tr>
<td>M2</td>
<td>P4</td>
<td>4</td>
</tr>
<tr>
<td>M2</td>
<td>P1</td>
<td>6</td>
</tr>
<tr>
<td>M3</td>
<td>P4</td>
<td>3</td>
</tr>
<tr>
<td>M4</td>
<td>P1</td>
<td>4</td>
</tr>
<tr>
<td>M4</td>
<td>P3</td>
<td>5</td>
</tr>
<tr>
<td>M5</td>
<td>P2</td>
<td>7</td>
</tr>
</tbody>
</table>

## Salary

<table>
<thead>
<tr>
<th>Position</th>
<th>YSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW Developer</td>
<td>60.000</td>
</tr>
<tr>
<td>HW-Developer</td>
<td>55.000</td>
</tr>
<tr>
<td>Analyst</td>
<td>65.000</td>
</tr>
<tr>
<td>Manager</td>
<td>90.000</td>
</tr>
</tbody>
</table>
"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: $R = \bigcup_{1 \leq i \leq n} R_i$
Example: fragmentation of $\text{PROJECT}$ 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**

<table>
<thead>
<tr>
<th>PNr</th>
<th>PName</th>
<th>Budget</th>
<th>Loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>Hardware Dev.</td>
<td>150.000</td>
<td>M</td>
</tr>
<tr>
<td>P1</td>
<td>DB Development</td>
<td>200.000</td>
<td>MD</td>
</tr>
<tr>
<td>P3</td>
<td>Web-Design</td>
<td>100.000</td>
<td>MD</td>
</tr>
<tr>
<td>P4</td>
<td>Customizing</td>
<td>250.000</td>
<td>B</td>
</tr>
</tbody>
</table>
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 $S$

$$S_i = S \times R_i = S \times \sigma_{P_i}(R) = \pi_{S,*}(S \times \sigma_{P_i}(R))$$

- $P_i$ 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 $S$)
Fragmentation of relation ASSIGNMENT derived from fragmentation of PROJECT relation

\[
\text{ASSIGNMENT}_1 = \text{ASSIGNMENT} \times \text{PROJECT}_1 \\
\text{ASSIGNMENT}_2 = \text{ASSIGNMENT} \times \text{PROJECT}_2 \\
\text{ASSIGNMENT}_3 = \text{ASSIGNMENT} \times \text{PROJECT}_3
\]

<table>
<thead>
<tr>
<th>ASSIGNMENT$_1$</th>
<th>ASSIGNMENT$_2$</th>
<th>ASSIGNMENT$_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNr</td>
<td>PNr</td>
<td>Capacity</td>
</tr>
<tr>
<td>M5</td>
<td>P2</td>
<td>7</td>
</tr>
<tr>
<td>M3</td>
<td>P4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Schallehn (FIN/ITI) Distributed Data Management 2015/2016 114 / 300
Vertical Fragmentation

- Column-wise decomposition of a relation using relational projection
- Completeness: each attribute must be in at least one fragment
- Reconstructability: through natural join
  \[ \text{primary key of global relation must be in each fragment} \]
  \[ R_i := \pi_{K,A_i,...,A_j}(R) \]
  \[ R = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n \]

- Limited disjointness
Vertical Fragmentation /2

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

\[ \text{PROJECT}_1 = \pi \text{PNr, PName, Loc} (\text{PROJECT}) \]
\[ \text{PROJECT}_2 = \pi \text{PNr, Budget} (\text{PROJECT}) \]

<table>
<thead>
<tr>
<th>PROJECT_1</th>
<th>PROJECT_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNr</td>
<td>PNr</td>
</tr>
<tr>
<td>P1</td>
<td>200.000</td>
</tr>
<tr>
<td>P2</td>
<td>150.000</td>
</tr>
<tr>
<td>P3</td>
<td>100.000</td>
</tr>
<tr>
<td>P4</td>
<td>250.000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PName</th>
<th>Budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB Development</td>
<td>MD</td>
</tr>
<tr>
<td>Hardware Dev.</td>
<td>M</td>
</tr>
<tr>
<td>Web-Design</td>
<td>MD</td>
</tr>
<tr>
<td>Customizing</td>
<td>B</td>
</tr>
</tbody>
</table>
Hybrid Fragmentation

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

\[
\begin{align*}
\text{PROJECT}_1 &= \pi_{\text{Nr}, \text{Name}, \text{Loc}}(\text{PROJECT}) \\
\text{PROJECT}_2 &= \pi_{\text{Nr}, \text{Budget}}(\text{PROJECT}) \\
\text{PROJECT}_{1,1}, &\sigma_{\text{Loc} = \text{M}}(\text{PROJECT}_1) \\
\text{PROJECT}_{1,2}, &\sigma_{\text{Loc} = \text{B}}(\text{PROJECT}_1) \\
\text{PROJECT}_{1,3}, &\sigma_{\text{Loc} = \text{MD}}(\text{PROJECT}_1)
\end{align*}
\]
Fragmentation transparency

- Decomposition of a relation is for user/application not visible
- Only view on global relation
- Requires mapping of DB operations to fragments by DDBMS
- Example
  - Transparent:
    ```sql
    select * from Project where PNr=P1
    ```
  - Without transparency:
    ```sql
    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
    ```
Example (continued)

- Transparent:
  ```sql
  update Project set Ort='B' where PNr=P3
  ```

- Without transparency:
  ```sql
  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 \( \text{dom}(A_i) \)

Definition: **simple predicate** \( p_i \) of the form \( A_j \theta \text{ const} \) with \( \text{const} \in \text{dom}(A_j) \)
  - Defines possible binary fragmentation of \( R \)
  - Example:
    \[
    \text{PROJECT}_1 = \sigma_{\text{Budget}>150.000}(\text{PROJECT})
    \]
    \[
    \text{PROJECT}_2 = \sigma_{\text{Budget}\leq150.000}(\text{PROJECT})
    \]

Definition: **Minterm** \( m \) is conjunction of simple predicates as
\[
m = p_1^* \land p_2^* \land \cdots \land p_j^*
\]
with \( p_i^* = p_i \) oder \( p_i^* = \neg p_i \)
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
  - $R = \bigcup_{m \in M_n(P)} \sigma_m(R)$
  - $\sigma_{m_i} \cap \sigma_{m_j} = \emptyset$, $\forall m_i, m_j \in M_n(P), m_i \neq m_j$
Completeness and no redundancy not sufficient:

- \[ P = \{ \text{Budget} < 100.000, \text{Budget} > 200.000, \text{Ort} = 'MD', \text{Ort} = 'B' \} \]
- Minterm \( p_1 \land p_2 \land p_3 \land p_4 \) not satisfiable; but \( \neg p_1 \land \neg p_2 \land \neg p_3 \land \neg p_4 \)

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: \( \text{acc}(m) \)
     (e.g. derived from query log)
   - Fragment size: \( \text{card}(f) \)
     (derived from data distribution statistics)
   - \( p_i \) is relevant, if \( \frac{\text{acc}(m_i)}{\text{card}(f_i)} \neq \frac{\text{acc}(m_j)}{\text{card}(f_j)} \)
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 \( R \)

\[
R(m) := \sigma_m(R) \quad \text{with} \quad m \in M(P)
\]
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 $\rightarrow$ partitioned DB
  - *Redundant*: fragments can be stored more than once $\rightarrow$ 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
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 \( \sum \text{Storage} + \sum \text{Transfer} \) for \( K \) fragments and \( L \) nodes
- Storage costs:
  \[
  \sum_{\text{Storage}} = \sum_{p,i} S_p D_{pi} SC_i
  \]
  - \( S_p \): Size of fragment \( p \) in data units
  - \( SC_i \): Storage Costs 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:

\[
\sum_{\text{Transfer}} = \sum_{i,t,p,j} F_{it} O_{tp} D_{pj} T_{Cij} + \sum_{i,t,p,j} F_{it} R_{tp} D_{pj} T_{Cji}
\]

- \( F_{it} \): 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)
- \( T_{Cij} \): TransferCosts from node \( i \) to \( j \) in data units
- \( R_{tp} \): Size of the result of one operation of type \( t \) on fragment \( p \)
Identification of an optimal Allocation /3

- Additional constraints:

\[ \sum_{i} D_{pi} = 1 \text{ for } p = 1, \ldots, K \]
\[ \sum_{p} S_{p} D_{pi} \leq M_{i} \text{ for } p = i, \ldots, L \]

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
Cost model for redundant replication

Additional constraints slightly modified:

\[ \sum_{i} D_{pi} \geq 1 \text{ for } p = 1, \ldots, K \]

\[ \sum_{p} S_{p} D_{pi} \leq M_{i} \text{ for } p = i, \ldots, L \]
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 $D_{pj} = 1$
  - $\Phi_t$: of an operation $\sum$ (in case of update) or min (in case of read operation)

\[
\sum_{T \text{ transfer}} = \sum_{i, t, p} F_{it} \Phi_t \left( O_{tp} TC_{ij} + R_{tp} TC_{ji} \right)
\]

Schallehn (FIN/ITI) Distributed Data Management 2015/2016 133 / 300
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 III

Distributed DBS Query Processing
Overview

- 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

Global Query Processing
- Global Schema
- Distribution Schema
- Global Statistics

Global Query
- Query Transformation
  - Algebra Expression
  - Data Localization
    - Fragment Expression
  - Global Optimization

Locally optimized Query

Local Query Processing
- Global Schema
- Local Optimization
  - Globally optimized Fragment Expression

Locally optimized Query
Phases of Query Processing /2

- Query transformation
  - Translation of SQL to internal representation (Relational Algebra)
  - Name resolution: object names → internal names (catalog)
  - Semantic analysis: verification of global relations and attributes, view expansion, global access control
  - Normalization: transformation to canonical format
  - Algebraic optimization: improve "‘efficiency’" of algebra expression
Phases of Query Processing /3

- **Data localization:**
  - Identification of nodes with fragments of used relations (from distribution schema)

- **Global optimization:**
  - Selection of least expensive query plan
  - Consideration of costs (execution and communication, cardinalities of intermediate results)
  - Determination of execution order and place
Phases of Query Processing /4

- **Local optimization**
  - Optimization of fragment query on each node
  - Using local catalog data (statistics)
  - Usage of index structures
  - Cost-based selection of locally optimal plan

- **Code-generation**
  - Map query plan to executable code
Translation to Relational Algebra

\[
\text{select } A_1, \ldots, A_m \\
\text{from } R_1, R_2, \ldots, R_n \\
\text{where } F
\]

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 simplify 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 $p_{ij}$:
    \[(p_{11} \lor p_{12} \lor \cdots \lor p_{1n}) \land \cdots \land (p_{m1} \lor p_{m2} \lor \cdots \lor p_{mn})\]
  - Disjunctive normal form (DNF):
    \[(p_{11} \land p_{12} \land \cdots \land p_{1n}) \lor \cdots \lor (p_{m1} \land p_{m2} \land \cdots \land p_{mn})\]
  - Transformation according to equivalence rules for logical operations
Normalization /2

- **Equivalence rules**
  - $p_1 \land p_2 \iff p_2 \land p_1$ und $p_1 \lor p_2 \iff p_2 \lor p_1$
  - $p_1 \land (p_2 \land p_3) \iff (p_1 \land p_2) \land p_3$ und $p_1 \land (p_2 \lor p_3) \iff (p_1 \lor p_2) \lor p_3$
  - $p_1 \land (p_2 \lor p_3) \iff (p_1 \land p_2) \lor (p_1 \land p_3)$ und $p_1 \lor (p_2 \land p_3) \iff (p_1 \lor p_2) \land (p_1 \lor p_3)$
  - $\neg (p_1 \land p_2) \iff \neg p_1 \lor \neg p_2$ und $\neg (p_1 \lor p_2) \iff \neg p_1 \land \neg p_2$
  - $\neg (\neg p_1) \iff p_1$
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:
  $$P.PNr = A.PNr \land Budget > 100.000 \land (Loc = 'MD' \lor Loc = 'B')$$

- Selection condition in DNF:
  $$\begin{align*}
  (P.PNr = A.PNr \land Budget > 100.000 \land Loc = 'MD') \lor \\
  (P.PNr = A.PNr \land Budget > 100.000 \land Loc = 'B')
  \end{align*}$$
Phases of Optimization

Optimization

- Logical Optimization
- Physical Optimization
- Cost-based Optimization

Best Access Plan

Optimized Algebra Expression

Possible Access Plans
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
  - Identify and remove redundancies
- Result: improved algebraic expression $\Rightarrow$ operator tree $\Rightarrow$ initial query plan
Operators $\sigma$ and $\Join$ commute, if selection attribute from one relation:

$$\sigma_F(r_1 \Join r_2) \iff \sigma_F(r_1) \Join r_2 \quad \text{falls } attr(F) \subseteq R_1$$

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

$$\sigma_F(r_1 \Join r_2) \iff \sigma_{F_1}(r_1) \Join \sigma_{F_2}(r_2)$$

if $attr(F_1) \subseteq R_1$ 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 $R_2$:

$$\sigma_F(r_1 \Join r_2) \iff \sigma_{F_2}(\sigma_{F_1}(r_1) \Join r_2) \quad \text{if } attr(F_1) \subseteq R_1$$
Combination of conditions of $\sigma$ is identical to logical conjunction $\Rightarrow$ operations can change their order

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

(uses commutativity of logic AND)
Operator $\triangleright$ is commutative:

\[ r_1 \triangleright r_2 \leftrightarrow r_2 \triangleright r_1 \]

Operator $\triangleright$ is associative:

\[ (r_1 \triangleright r_2) \triangleright r_3 \leftrightarrow r_1 \triangleright (r_2 \triangleright r_3) \]

Domination of sequence of $\pi$ operators:

\[ \pi_X(\pi_Y(r_1)) \leftrightarrow \pi_X(r_1) \]

$\pi$ and $\sigma$ are commutative in some cases:

\[ \sigma_F(\pi_X(r_1)) \leftrightarrow \pi_X(\sigma_F(r_1)) \]

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

\[ \pi_{X_1}(\sigma_F(\pi_{X_1}X_2(r_1))) \leftrightarrow \pi_{X_1}(\sigma_F(r_1)) \]

if \( \text{attr}(F) \supseteq X_2 \)
Commutation of $\sigma$ and $\cup$:

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

Commutation of $\sigma$ and with other set operation $-$ and $\cap$

Commutation of $\pi$ and $\Join$ partially possible: join attributes must be kept and later removed (nevertheless decreases intermediate result size)

Commutation of $\pi$ and $\cup$

Distributivity for set operations

Idempotent expressions, e.g. $r_1 \Join r_1 = r_1$ and $r_1 \cup r_1 = r_1$

Operations with empty relations, e.g. $r_1 \cup \emptyset = r_1$

Commutativity of set operations

\ldots
Algebraic Optimization: Example

```
select * from Project 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

Query Transformation

Resolution of Fragments of global Relations

Algebr. Optimization

Physical Optimization
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:

\[ \sigma_{\text{Loc}=\text{'MD'} \land \text{Budget} \leq 100.000}(\text{PROJECT}) \]

\[ \implies \]

\[ \sigma_{\text{Loc}=\text{'MD'} \land \text{Budget} \leq 100.000}(\text{PROJ}_1 \cup \text{PROJ}_2 \cup \text{PROJ}_3) \]
Requirement: further simplification of query
Goal: eliminate queries on fragments not used in query
Example: pushing down \(\sigma\) to fragments
\[
\sigma_{\text{Loc}}=\text{'MD'} \land \text{Budget} \leq 100.000 (\text{PROJ}_1 \cup \text{PROJ}_2 \cup \text{PROJ}_3)
\]
because of:
\[
\sigma_{\text{Budget} \leq 100.000} (\text{PROJ}_2) = \emptyset, \sigma_{\text{Budget} \leq 100.000} (\text{PROJ}_3) = \emptyset
\]
\[\Rightarrow\]
\[
\sigma_{\text{Loc}}=\text{'MD'}(\sigma_{\text{Budget} \leq 100.000}(\text{PROJ}_1))
\]
For horizontal fragmentation

- Also possible simplification of join processing
- Push down join if fragmentation on join attribute
Data Localization: Example II

- **Schema:**
  
  \[
  M_1 = \sigma_{MNr<'M3'}(MEMBER)
  \]
  
  \[
  M_2 = \sigma_{M3'<MNr<'M5'}(MEMBER)
  \]
  
  \[
  M_3 = \sigma_{MNr>'M5'}(MEMBER)
  \]

  
  \[
  Z_1 = \sigma_{MNr<'M3'}(ASSIGNMENT)
  \]
  
  \[
  Z_2 = \sigma_{MNr>'M3'}(ASSIGNMENT)
  \]

- **Query:** ASSIGNMENT \texttimes MEMBER

  
  \[
  \Rightarrow (M_1 \cup M_2 \cup M_3) \texttimes (Z_1 \cup Z_2)
  \]

  
  \[
  \Rightarrow (M_1 \texttimes Z_1) \cup (M_2 \texttimes Z_2) \cup (M_3 \texttimes Z_2)
  \]
Vertical fragmentation: reduction by pushing down projections

Example:

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

\[
\text{PROJECT} = \text{PROJ}_1 \Join \text{PROJ}_2
\]

Query: \(\pi_{\text{PName}}(\text{PROJECT})\)

\[\Rightarrow\]

\[
\pi_{\text{PName}}(\text{PROJ}_1 \Join \text{PROJ}_2)
\]

\[\Rightarrow\]

\[
\pi_{\text{PName}}(\text{PROJ}_1)
\]
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

(1) \( E := \sigma_F[R : q_R] \quad \rightarrow \quad [E : F \land q_R] \)
(2) \( E := \pi_A[R : q_R] \quad \rightarrow \quad [E : q_R] \)
(3) \( E := [R : q_R] \times [S : q_S] \quad \rightarrow \quad [E : q_R \land q_S] \)
(4) \( E := [R : q_R] - [S : q_S] \quad \rightarrow \quad [E : q_R] \)
(5) \( E := [R : q_R] \cup [S : q_S] \quad \rightarrow \quad [E : q_R \lor q_S] \)
(6) \( E := [R : q_R] \bowtie_F [S : q_S] \quad \rightarrow \quad [E : q_R \land q_S \land F] \)
Usage of rules for description – no processing

Example: $\sigma_{100.000 \leq \text{Budget} \leq 200.000} (\text{PROJECT})$

\begin{align*}
E_1 &= \sigma_{100.000 \leq \text{Budget} \leq 200.000} [\text{PROJ}_1 : \text{Budget} \leq 150.000] \\
\implies & [E_1 : (100.000 \leq \text{Budget} \leq 200.000) \land (\text{Budget} \leq 150.000)] \\
\implies & [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] \\
\implies & [E_2 : (100.000 \leq \text{Budget} \leq 200.000) \land \\
& (150.000 < \text{Budget} \leq 200.000)] \\
\implies & [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] \\
\implies & [E_3 : (100.000 \leq \text{Budget} \leq 200.000) \land (\text{Budget} > 200.000)] \\
\implies & E_3 = \emptyset
\end{align*}
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”

<table>
<thead>
<tr>
<th>Strategy</th>
<th>#Messages</th>
<th>#Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW at R-node</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>SW at S-node</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>SW at 3. node</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>FAN at S-node</td>
<td>$6 \times 2 = 12$</td>
<td>$6 + 2 \times 2 = 10$</td>
</tr>
<tr>
<td>FAN at R-node</td>
<td>$7 \times 2 = 14$</td>
<td>$7 + 2 \times 3 = 13$</td>
</tr>
</tbody>
</table>
Join Processing /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 \( r \bowtie s \):

\[
\text{for each } t_r \in r \text{ do begin}
\quad \text{for each } t_s \in s \text{ do begin}
\quad \quad \text{if } \varphi(t_r, t_s) \text{ then put}(t_r \cdot t_s) \text{ endif}
\quad \text{end}
\text{end}
\]

\( r \bowtie \varphi s: \)
Sort Merge-Join

\[ X := R \cap S; \text{ if not yet sorted, first sort } r \text{ and } s \text{ on join attributes } X \]

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: \( O(n_r \times n_s) \)
- \( X \) 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: \( r \bowtie s = r \bowtie (s \bowtie r) = r \bowtie (s \bowtie \pi_A(r)) \) (\( A \) 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 \( s' = s \bowtie \pi_A(r) = s \bowtie r \) 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 $s' = \{ t \in s \mid B[h(t.A)]$ is set $\}$ and transfer to $N_r$
3. Node $N_r$: compute $r \bowtie s' = r \bowtie s$
Comparison:

- Decreased size of request message compared to semi-join
- Hash-mapping not injective ⟷ only potential join partners in bit vector
  ⟷ sufficiently great $n$ and suitable hash function $h$ required
Bit Vector-Join: Example

R–Node

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

\[ h(t.A(R)) \]  
\[ (1101100) \]

\[ h(v) = v \mod 7 \]

S–Node

<table>
<thead>
<tr>
<th>S</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>h(C)</th>
<th>Hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>+</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>+</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>+</td>
</tr>
</tbody>
</table>

Q

<table>
<thead>
<tr>
<th>Q</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

S’  

<table>
<thead>
<tr>
<th>S’</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Schallehn (FIN/ITI)  
Distributed Data Management  
2015/2016  175 / 300
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

- Query
- Generate the Search Space
- Equivalent Plans
- Search Strategy
- Best Plan
- Transformation Rules
- Cost Model
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** $\mapsto$ join order as nested structure/loops, all inner nodes (operations) have at least one input relation
- **Bushy trees** $\mapsto$ 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
    
    $$T_{total} = T_{CPU} \times \#insts + T_{I/O} \times \#I/Os + T_{MSG} \times \#msgs + T_{TR} \times \#bytes$$
  
  - Communication time:
    
    $$CT(\#bytes) = T_{MSG} + T_{TR} \times \#bytes$$

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

- Timespan from initiation of query until availability of full results

\[ T_{total} = T_{CPU} \times seq\_#insts + T_{I/O} \times seq\_#I/Os + T_{MSG} \times seq\_#msgs + T_{TR} \times seq\_#bytes \]

- With seq\_#x is maximum number x that must be performed sequentially
Total Time vs. Response Time

\[ T_{\text{total}} = 2T_{\text{MSG}} + T_{\text{TR}}(x + y) \]
\[ T_{\text{response}} = \max\{T_{\text{MSG}} + T_{\text{TR}} \times x, T_{\text{MSG}} + T_{\text{TR}} \times y\} \]
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: $\text{length}(A_i)$ (in Byte)
- Number of distinct values of $A_i$ for each fragment $R_j$: $\text{val}(A_i, R_j)$
- Min and max attribute values: $\text{min}(A_i)$ and $\text{max}(A_i)$
- Cardinality of value domain of $A_i$: $\text{card}(\text{dom}(A_i))$
- Number of tuples in each fragment: $\text{card}(R_j)$
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 $R$
    \[ SF = 0.1 \]

- Size of an intermediate relation:

  \[
  \text{size}(R) = \text{card}(R) \times \text{length}(R)
  \]
Cardinality of Selections

- **Cardinality**
  \[ \text{card}(\sigma_F(R)) = SF_S(F) \times \text{card}(R) \]

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

  \[
  SF_S(A = v) = \frac{1}{\text{val}(A, R)} \\
  SF_S(A > v) = \frac{\text{max}(A) - v}{\text{max}(A) - \text{min}(A)} \\
  SF_S(A < v) = \frac{v - \text{max}(A)}{\text{max}(A) - \text{min}(A)}
  \]
Cardinality of Selections /2

\[
SF_S(p(A_i) \land p(A_j)) = SF_S(p(A_i)) \ast SF_S(p(A_j))
\]

\[
SF_S(p(A_i) \lor p(A_j)) = SF_S(p(A_i)) + SF_S(p(A_j)) - (SF_S(p(A_i)) \ast SF_S(p(A_j)))
\]

\[
SF_S(A \in \{v_1, \ldots, v_n\}) = SF_S(A = v) \ast \text{card}(\{v_1, \ldots, v_n\})
\]
Cardinality of Projections

- Without duplicate elimination
  \[\text{card}(\pi_A(R)) = \text{card}(R)\]

- With duplicate elimination (for non-key attributes \(A\))
  \[\text{card}(\pi_A(R)) = \text{val}(A, R)\]

- With duplicate elimination (a key is subset of attributes in \(A\))
  \[\text{card}(\pi_A(R)) = \text{card}(R)\]
Cardinality of Joins

- Cartesian products

\[ \text{card}(R \times S) = \text{card}(R) \ast \text{card}(S) \]

- 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

\[ \text{card}(R \bowtie S) = SF_J \ast \text{card}(R) \ast \text{card}(S) \]
Cardinality of Semi-joins

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

\[
SF_{SJ}(R \bowtie_A S) = \frac{\text{val}(A, S)}{\text{card}(\text{dom}(A))}
\]

- Cardinality:

\[
\text{card}(R \bowtie_A S) = SF_{SJ}(S.A) \times \text{card}(R)
\]
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:**
  - \( \text{card}(\text{PROJECT}_1) = 3.500 \)
  - \( \text{card}(\text{PROJECT}_2) = 4.000 \)
  - \( \text{card}(\text{PROJECT}_3) = 2.500 \)
  - \( \text{length}(\text{PROJECT}) = 30 \)
  - \( \text{min}(\text{Budget}) = 50.000 \)
  - \( \text{max}(\text{Budget}) = 300.000 \)
  - \( T_{\text{MSG}} = 0.3s \)
  - \( T_{\text{TR}} = 1/1000s \)
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 $R \bowtie S$
  
  - if $\text{size}(R) < \text{size}(S)$
  
  - if $\text{size}(R) > \text{size}(S)$

- Join order for 3 relations $R \bowtie_A S \bowtie_B T$
  
  - $K_2$

  - $K_1$

  - $K_3$
(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 $R \Join_A S$
  1. $(R \Join_A S) \Join_A S$
  2. $R \Join_A (S \Join_A R)$
  3. $(R \Join_A S) \Join_A (S \Join_A R)$
- Comparison $R \Join_A S$ vs. $(R \Join_A S) \Join_A S$ for size($R$) < size($S$)
- Costs for $R \Join_A S$: transfer of $R$ to $N_2 \leadsto T_{TR} \ast \text{size}(R)$
Utilization of Semi-Joins /2

- Processing of semi-join variant
  1. \( \pi_A(S) \rightarrow N_2 \)
  2. At node \( N_2 \): computation of \( R' := R \times_A S \)
  3. \( R' \rightarrow N_1 \)
  4. At node \( N_1 \): computation of \( R' \times_A S \)

- Costs: costs for step 1 + costs for step 2

\[
T_{TR} \times \text{size}(\pi_A(S)) + T_{TR} \times \text{size}(R \times_A S)
\]

- Accordingly: semi-join is better strategy if

\[
\text{size}(\pi_A(S)) + \text{size}(R \times_A S) < \text{size}(R)
\]
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 IV

Distributed DBS - Transaction Processing
Overview

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

- **Atomicity** means that a transaction can not be interrupted or performed only partially
  - TXN is performed in its entirety or not at all

- **Consistency** to preserve data integrity
  - A TXN starts from a consistent database state and ends with a consistent database state

- **Isolation**
  - Result of a TXN must be independent of other possibly running parallel TXNs

- **Durability 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
- $T_i$ istxn $i$
- Integrity constraint on persistent data $A + B + C = 0$
Non-repeatable Read /2

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X := A$;</td>
<td>$Y := A/2$;</td>
</tr>
<tr>
<td></td>
<td>$A := Y$;</td>
</tr>
<tr>
<td></td>
<td>$C := C + Y$;</td>
</tr>
<tr>
<td>$X := X + B$;</td>
<td>commit;</td>
</tr>
<tr>
<td>$X := X + C$;</td>
<td>commit;</td>
</tr>
</tbody>
</table>
## Dirty Read

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
</table>
| `read(X);`  
`X := X + 100;`  
`write(X);` | `read(X);`  
`Y := Y + X;`  
`write(Y);`  
`commit;` |
| `abort;` | |
The Phantom Problem

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>select count (*) into $X$ from Employee;</td>
<td>insert into Employee values (<em>Meier</em>, 50000, · · · );</td>
</tr>
<tr>
<td>update Employee set Salary = Salary + 10000/$X$;</td>
<td>commit;</td>
</tr>
<tr>
<td>commit;</td>
<td>commit;</td>
</tr>
</tbody>
</table>
Lost Update

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($X$);</td>
<td>read($X$);</td>
<td>10</td>
</tr>
<tr>
<td>$X := X + 1$;</td>
<td>$X := X + 1$;</td>
<td>10</td>
</tr>
<tr>
<td>write($X$);</td>
<td>write($X$);</td>
<td>11</td>
</tr>
</tbody>
</table>
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 serializability most often only considering read/write operations and their conflicts → **Conflict Serializability**
- Considering other operations requires analysis of operations semantics
- Rules out subset of serializable schedules which are hard to detect
## Conflicting Operations

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read $A$</td>
<td>read $A$</td>
<td>read $A$</td>
<td>write $A$</td>
</tr>
<tr>
<td><strong>independent of order</strong></td>
<td><strong>dependent on order</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write $A$</td>
<td>read $A$</td>
<td>write $A$</td>
<td>write $A$</td>
</tr>
<tr>
<td><strong>dependent on order</strong></td>
<td><strong>dependent on order</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r(y)$</td>
<td></td>
<td>$r(y)$</td>
<td>$r(u)$</td>
</tr>
<tr>
<td>$w(y)$</td>
<td>$r(y)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$w(x)$</td>
<td>$w(x)$</td>
<td>$w(x)$</td>
<td></td>
</tr>
<tr>
<td>$w(z)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$s = r_1(y)r_3(u)r_2(y)w_1(y)w_1(x)w_2(x)w_2(z)w_3(x)$
Conflict Serializability: Example /2

G(s):

\[ T_1 \]

\[ T_2 \] \[ T_3 \]
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 conflict serializability 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

<table>
<thead>
<tr>
<th></th>
<th>( r_l_i(x) )</th>
<th>( w_l_i(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_l_j(x) )</td>
<td>( \sqrt{\ } )</td>
<td>—</td>
</tr>
<tr>
<td>( w_l_j(x) )</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
2-Phase-Locking Protocol

![Diagram of 2PL Protocol]

- **Time**
- **#Locks**
- **Lock Acquisition**
- **Lock Release**

Schallehn (FIN/ITI)
2-Phase-Locking: Example

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u(x)$</td>
<td>$wl(x)$</td>
</tr>
<tr>
<td></td>
<td>$wl(y)$</td>
</tr>
<tr>
<td>$wl(y)$</td>
<td>$u(x)$</td>
</tr>
<tr>
<td></td>
<td>$u(y)$</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Strict 2-Phase-Locking

Current practice in most DBMS:

Avoids cascading aborts!

Schallehn (FIN/ITI)  Distributed Data Management  2015/2016  222 / 300
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 → same result on all nodes
- Deadlocks (blocking/blocked TXNs) harder to detect
Structure of Distributed TXNs

Primary TXN
Performed on coordinator node

Sub-TXNs
Performed on other nodes
Distributed Synchronization

- 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:
  - \( \text{max-r-scheduled}[x] \):
    timestamp of last TXN performing a read operation on \( x \)
  - \( \text{max-w-scheduled}[x] \):
    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.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ts = 200$</td>
<td>$ts = 150$</td>
<td>$ts = 175$</td>
<td>mrs  mws</td>
<td>mrs  mws</td>
<td>mrs  mws</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
</tbody>
</table>

- read $B$
- write $B$
- read $A$
- write $A$
- write $C$ ↓
- abort

- read $C$
- write $B$
- read $A$
- write $A$
- write $C$ ↓
- abort

|       |       |       | 0 0     | 200 0   | 0 0     |
| 150 0 | 200 0 | 0 0   |
| 150 0 | 200 0 | 175 0 |
| 150 200 | 200 200 | 175 0 |
| 150 200 | 200 200 | 175 ↓ |

0 0 0
Distributed Commit

- Synchronization provides **Consistency** and **Isolation**
- Commit Protocol provides **Atomicity** and **Durability**

**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

Coordinator

INITIAL

Write begin_commit to Log

WAIT

All agreed?

Yes

No

Participant

INITIAL

Write abort to Log

Vote-Abort

Write to Log

Vote-Commit

Write abort to Log

Global-Abort

READY

Prepare-To-Commit

Write to Log

abort

No

Yes

Ready to commit?

Yes

No

Write ready to Log
2PC: Procedure /2

- Write to Log
- Global Decision?
  - Write abort to Log
  - Global-Abort
  - READY
  - Write to Log
  - Global-Commit
  - COMMIT
  - Write commit to Log
  - EOT to Log

- No
  - Write abort to Log
  - Global-Abort
  - READY
  - Write to Log
  - Global-Commit
  - COMMIT
  - Write commit to Log
  - EOT to Log

- Yes
  - Write commit to Log
  - COMMIT
  - Write EOT to Log

- Unilateral Abort
  - Abort
  - Global Decision?
    - Global-Commit
    - COMMIT
    - Write commit to Log
    - COMMIT
    - Write to Log
    - COMMIT

- ACK

- All agreed?
  - Yes
    - Write commit to Log
    - COMMIT
  - No
    - Write abort to Log
    - Global-Abort
    - READY
    - Write to Log
    - Global-Commit
    - COMMIT
    - Write commit to Log
    - COMMIT
    - Write to Log
    - COMMIT

- ACK
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 and no other node is in **PRE-COMMIT** state, TXN must abort

**Disadvantages**
- 3PC has increased message number
- Still problematic in case of network partitioning
Transaction Deadlocks

Node #1

T1: Update of Account A1

T2: Lock Request to Account A1

Lock Conflict

Node #2

T1: Lock Request Account A2

T2: Update of Account A2

Lock Conflict
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:

  ![Wait graph diagram]

- 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 → 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

Node #1

\[ T_1 \rightarrow T'_5 \rightarrow T_5 \rightarrow T_4 \]

\[ T_2 \rightarrow T'_3 \rightarrow T_3 \]

Node #2

\[ T_1 \rightarrow T'_5 \rightarrow T_5 \rightarrow T_4 \]

\[ T_2 \rightarrow T'_3 \rightarrow T_3 \]

EX

Node #3

\[ T_2' \rightarrow T_3 \]

EX

Node #1

\[ T_1 \rightarrow T'_5 \rightarrow T_5 \rightarrow T_4 \]

\[ T_2 \rightarrow T'_3 \rightarrow T_3 \]

EX

Node #2

\[ T_1 \rightarrow T'_5 \rightarrow T_5 \rightarrow T_4 \]

\[ T_2 \rightarrow T'_3 \rightarrow T_3 \]

EX
Motivation for Replication

- **pro**
  - efficient read accesses
  - increased availability

- **contra**
  - increased update efforts
  - increased storage requirements

- **Replication transparency** \(\leadsto\) DDBMS manages updates on redundant data
Transaction Replication

- Data integrity must be granted across replicas $\xrightarrow{\sim}$ 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
"Read One Write All"

- one logical read operation → 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 → 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 availability of all nodes storing replicated data
  - Longer run-time of update TXNs $\Rightarrow$ decreased throughput and availability
  - 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 **asynchronously** 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
Secondary copies get changes from update queue (FIFO) consistency with primary copy.

In case of failure of secondary node, queue is processed when node is restarted.

- T1: Update Tuple #1 + Commit
- T2: Tuple #2 locked

Diagram:
- Copy #1
- Primary Copy
- Copy #2
- Copy #3

Schallehn (FIN/ITI)
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
Secondary copies get changes from update queue (FIFO) to maintain 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-Serializability
  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 \( \rightarrow \) 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 \(\leadsto\) 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 → 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 possible
- 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 V
Parallel DBS
Overview

- Foundations
- Parallel Query Processing
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 $\rightarrow$ improved throughput
- Intra-TXN parallelization: simultaneous execution processing of operations within one txn $\rightarrow$ 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 \( 0 \leq F_{opt} \leq 1 \) of operations which can be optimized by parallelized execution

\[
\text{RT Speedup} = \frac{1}{(1 - F_{opt}) + \frac{F_{opt}}{\text{Speedup}_{opt}}}
\]
Speedup

- Speedup furthermore limited by
  - Overhead for coordination and communication for parallel execution
  - Interferences between parallel executions through shared resource 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 Response 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**
  - Typical architecture: DBMS support for multi-processor computers
  - Ideal for Response Time Speedup

- **Shared Disk**
  - DBMS support for tightly connected (e.g. fast network) nodes with shared disk access
  - Good for Response Time and Scalability
  - Requires: synchronization of disk accesses

- **Shared Nothing**
  - Connected nodes with their own disks
  - Ideal for Scalability
  - Requires thoughtful data fragmentation
Goal: use of horizontal fragmentation to avoid data skew
  ▶ Perform part of operation on equal size fragments
  ▶ Less data skew
    → less processing skew
    → optimal parallel execution
    → optimal speedup

Fixed or dynamic assignment of processors to fragments possible
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

range round robin hash

range

range

round robin

hash
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, separate 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

Data → Operation 1

Data → Operation 2

.....

Data → Operation n

Composition → Result
Pipelining Parallelization

Data Streams

Operation 1

Operation 2

Operation n

Result
Intra-Operator Parallelization

Data → Splitter → Operation (P1) → Merger → Result

Operation (P2) → .... → Operation (Pn)

Schallehn (FIN/ITI) Distributed Data Management 2015/2016 275 / 300
Parallelization of unary Operations

- Selection:
  \[ r = \bigcup_i r_i \Rightarrow \sigma_P(r) = \bigcup_i \sigma_P(r_i) \]

- Projection without duplicate elimination: dito

- Duplicate elimination: using sorting (\(\rightarrow\))

- Aggregate functions:
  - \(\min(r.\text{Attr}) = \min(\min(r_1.\text{Attr}), \ldots, \min(r_n.\text{Attr}))\)
  - \(\max(r.\text{Attr}) = \max(\max(r_1.\text{Attr}), \ldots, \max(r_n.\text{Attr}))\)
Aggregate functions (if no duplicate elimination necessary):

- \( \text{count}(r.\text{Attr}) = \sum_i \text{count}(r_i.\text{Attr}) \)
- \( \text{sum}(r.\text{Attr}) = \sum_i \text{sum}(r_i.\text{Attr}) \)
- \( \text{avg}(r.\text{Attr}) = \frac{\text{sum}(r.\text{Attr})}{\text{count}(r.\text{Attr})} \)
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: fragments 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

\[
\begin{align*}
P_1 &: 3,6 \\
P_2 &: 1,4 \\
P_3 &: 2,3 \\
P_4 &: 3,6 \\
P_5 &: 1,5 \\
P_6 &: 1,7 \\
P_7 &: 2,8 \\
P_8 &: 4,9
\end{align*}
\]

\[
\begin{align*}
P_1 &: 1,3,4,6 \\
P_2 &: 1,2,3,3,3,4,6,6 \\
P_3 &: 2,3,3,6 \\
P_4 &: 1,1,2,2,3,3,3,4,4,5,6,6,7,8,9 \\
P_5 &: 1,1,5,7 \\
P_6 &: 1,1,2,4,5,7,8,9 \\
P_7 &: 2,4,8,9 \\
P_8 &: 1,3,4,6,1,2,3,3,6,3,1,1,5,7,5,2,4,8,9,7,1,2,3,3,3,4,6,6,2,1,1,2,4,5,7,8,9,6,1,1,1,2,2,3,3,3,4,4,5,6,6,7,8,9,4
\end{align*}
\]
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
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
Parallel Join-Processing

- Join-Processing on Multiple Processors supported by 2 main approaches
  - Dynamic Replication
    - Replication (transfer) of the smaller relation \( r \) to each join node
    - Local execution of partial joins on each processor
    - Full result by union of partial results
  - Dynamic Partitioning
    - Partition tuples 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 across 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 \( r_i \):
   - \( s := \bigcup s_j \)
   - Compute \( t_i := r_i \Join s \)
   - send \( t_i \) to coordinator

5. Coordinator: receive and merge all \( t_i \) (union)
Dynamic Replication /2

Join-Knoten (R-Knoten)
S-Datenknoten

$\begin{align*}
& r_1 \\
& r_2 \\
& r_n \\
& S_1 \\
& S_m
\end{align*}$
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 responsible join node
   - parallel on each $S$-node: read and transfer each tuple of $s_i$ to responsible join node

3. Responsible node is computed by hash or range function $\rightarrow$ avoid or handle skew
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

Coordinator: receive and merge $t_k$
Dynamische Partitionierung

![Diagram](image)

- Join-Knoten
- R-Datenknoten
- S-Datenknoten
Multi-Way-Joins

- Variants considered: left and right-deep trees
Assumption: single join is executed as a Hash Join consisting of 2 phases

1. **Build Phase**: build hash table for first join relation
2. **Probe-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
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 decision about segmentation
Processor Allocation

- Assignment of operations (for now, join operations) to processors

**2-Phase Optimization**

1. Find an 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
Thresholds for number of processors: minimal response time point $T_t$, Execution efficiency point $T_e$

Execution efficiency for $n$ processors

\[
\text{Execution efficiency} = \frac{\text{Response time for 1 processor}}{n \cdot \text{Response time for } n \text{ processors}}
\]
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. \text{if } T(J) \leq N \text{ then }
   \{ \text{allocate } T(J) \text{ processors; } N := N - T(J) \}
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
   → if \( J \) has only one child \( J_1 \), allocate \( \min\{N_J, T_t(J_1)\} \) processors for \( J_1 \)
   → if \( J \) has two children \( J_1 \) 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 ($J_2: 9), J_3: 8, J_4: 12$