Distributed Data Management

Christoph Lofi
José Pinto

Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
• **Additional** constraints and **cost factors** compared to “classic” query optimization
  – Network **costs**, network **model**, **shipping** policies
  – **Fragmentation & allocation** schemes
  – Different optimization goals
    • **Response time vs. resource consumption**

• **Basic techniques** try to **prune** unnecessary accesses
  – **Generic query reductions**
• This lecture only covers very basic techniques
  – In general, distributed query processing is a very complex problem
  – Many and new optimization algorithms are researched
    • Adaptive and learning optimization
    • Eddies for dynamic join processing
    • Fully dynamic optimization
    • …

• Recommended literature
4.0 Introduction

4.0 Classic Transaction Processing
4.1 Distributed Transaction Processing
  – Distributed Two-Phase Commit
4.2 Distributed Two-Phase Locking
4.3 Case Study: SAP HANA
• Most early commercial databases have been used in **banking** and **financial** sector

  – **Financial Transaction:**
    • “Agreement between a buyer and seller to exchange an asset for payment”
      – Not good: No payment, no asset, no agreement,…

  – **Database transaction**
    • A group / workflow of coherent operations accessing and updating a database to perform a complex task
Automatic teller machines (ATM)

- **User Interaction**
  - Insert your card and input PIN code
  - Select amount
  - Take card and cash

- **Basic business workflow**
  - Authenticate user
  - Ask for requested amount
  - **Query** for available balance (**read operation**): if balance is too low shred card and abort…
  - Else **deduct** amount from balance (**write operation**)
  - Return card and dispense cash
• **Travel agency**
  
  – **User interaction**
    
    • “I want to go on vacations to Hawaii in the first week of May”
  
  – **Basic business workflow**
    
    • Check for flight availability during the week (**read operation**)
    • Check for hotel accommodation availability during the week (**read operation**)
    • Align dates for flights and hotels, shift it around a little for best prices
    • Reserve suitable room from hotel (**write operation**)
    • Buy flight ticket from airline (**write operation**)

---

Distributed Data Management – Christoph Lofi – IfIS – TU Braunschweig
• While processing workflows **severe problems** can occur
  – Even if we assume that individual workflows are always sensible and correct
• **Examples**
  – What if the ATM catches fire after **withdrawing** your money, but before dispensing it..?!  
  – What if you found the perfect flight and hotel, but while your flight is booked **somebody else** takes the last hotel room?
4.0 Transactions

• The previous examples require the concept of transactions
  – A transaction is a finite set of operations (workflow, program) that has to be performed in a certain order, while ensuring certain properties

• The properties are concerned with
  – **Integrity**: transactions can always be executed safely, especially in concurrent manner, while ensuring data integrity
  – **Fail Safety**: transactions are immune to system failures
4.0 Transactions

• What are transactions in databases?
  – A database stores a data
  – There are consistency constraints defined on the data
    • Structural constraints
      – Unique primary keys, correct foreign key relationships, correct data types, etc.
    • Semantic constraints
      – All additional rules ensuring a “correct” system state from an application point of view
  – If all constraints are fulfilled, the database is in an consistent state
4.0 Transactions

• A transaction is a database program (usually multiple queries) which reads and modifies data
  – A transaction should ensure database consistency
    • i.e. the transaction transforms the database from one consistent state to another consistent state
      – May be inconsistent during the execution of the transaction
  – Furthermore, transactional consistency should be ensured
    • i.e. multiple transactions must be able to run without collisions
• **Transaction Operations**

  – Transactions are an **interface contract** of an transaction-enabled server

  • **Start**: Starts an transaction, followed by a finite sequence of operations of a workflow or program

  • **Commit**: Executes all operations since transaction begin and ends the transaction

  • **Rollback**: Cancels the current transaction and reverts to the initial consistent state
• **Flat Transactions** are transactions which have a single start and commit point

  - A flat transaction **failing** returns to its start state
• Furthermore, **nested transactions** can be defined
  – Multiple commit and start points (**subtransactions**)
    • Simple case: transaction **chains**
    • Complex case: **workflows**

  -- What happens in case of failure?
    • **Ops2** fails: revert to \$s^↓1\$ or to \$s^↓2\$? Different options!
4.0 Transactions

- The **history** of **transaction** management in DBMS can be classified into several “**ages**”

- **“Stone Age”**
  - Application had to care for fail safety themselves
  - No transactions

- **“Classic History”**
  - The age of the great DB pioneers
  - Rise of the relational model, introduction of SQL, development of transaction management
  - Focus on data integrity
  - **Flat transactions**
  - **System R** and **ACID**
  - **RDB2**!
4.0 Transactions

• “Middle Ages”
  – Rise of complex business applications
  – **Distributed databases**
  – Relaxation of ACID principles
    • Tailoring for long-running transactions
    • Less strict transaction models
    • More flexible transaction model
      – **Simple nested transactions**: chains, sagas, etc.
  – **Distributed Data Management**
4.0 Transactions

• “Renaissance”
  – Workflow Management Systems
    • Workflows: complex nested transactions
    • Departure from flat transactions
  – Focus on workflow integrity and execution reliability

• “Modern Times”
  – Web Transactions
    • Especially, web service transactions
  – Long-running, loosely-coupled workflows on potentially very unreliable functions provided by autonomous parties
• The dominant paradigm in classic transaction processing is the **ACID** paradigm
  
  – **Atomicity**
  – **Consistency**
  – **Isolation**
  – **Durability**

• **Atomicity**
  – Any transaction is either executed *completely*, or *not at all*
  – From outside view, the transaction has *no* observable *intermediate state*

• **Consistency Preservation**
  – Transactions lead from one *consistent state* of the data instance to another
    • Constraints are not violated after the transaction
• **Isolation**
  – Transactions are isolated from others, i.e. even in a concurrent scenario transactions do not interfere with each other
  – **Parallel** execution of transactions has the same effect than **serial execution**

• **Durability**
  – Once **committed**, data changes performed by a transaction **survive** subsequent system failures
• Possible problems:
  – Atomicity
    • Dirty Read
  – Consistency
    • Inconsistent read
  – Isolation
    • Lost Update
    • Phantom Read
  – Durability
    • Data loss due to system crash
• How can we deal with these problems?
  – **Transaction Protocols**!

• For understanding transactions protocols, we will need two important concepts
  – **Schedules**
    • A “plan” containing the execution **order** of the ‘**operations**’ of different transactions
    • Also, schedule also denotes when **locks** are obtained or released
  – **Locks**
    • Flags which can be attached to data items to signal that it is already in use and may / may not be used by another operation
For a set of concurrently executed transactions:

- A schedule is a sequence of operations from different transactions
  - Usually, read or write operations
- A schedule is called serial if operations of different transactions are not mixed, i.e. executed in serial order
  - Obviously, serial schedules are pose no problems wrt. to transactional consistency
  - Also, no parallelism possible
- **Big aim**: Find schedules which behave like serial schedules but do allow for parallelism
4.0 Transactions

• Example (without starts and commits)
  – \( T_1 := r(x) r(y) w(u) w(x) \)
  – \( T_2 := r(p) r(q) w(p) \)
  – \( T_3 := r(z) w(z) \)

• Schedule
  – \( S := r(x) r(y) r(p) r(z) w(u) r(q) w(x) w(z) w(p) \)

• Serial schedule
  – \( S := r(p) r(q) w(p) r(x) r(y) w(u) w(x) r(z) w(z) \)
• How can we find schedules which “behave” safely?
  – i.e. equivalent to a serial plan?

• **Pessimistic Protocols**
  – Assume that error conditions will occur and prevent any problems beforehand
  – Spend some effort to create “safe” schedules
    • “Standard” approach for databases
    • e.g. *two phase locking*

• **Optimistic Protocols**
  – Assume everything will usually be fine and fix damage if something goes wrong
  – Just schedule something and see what happens
We will focus on locking protocols

- Pessimistic approach using locks to avoid transactional inconsistencies

- Simplified: If a transaction needs some data, it obtains a lock on it
  - Any other transaction may not use the item
  - Other transaction must wait until lock is released

- If the item is not used anymore, the lock is released
  - Other transaction may continue using the item
4.0 Transactions

- **Two types of locks**
  - **Read locks:**
    - Read locks can be *shared*
    - Multiple read locks on the same item may be issued to different transactions
      - Parallel reading!
  - **Write locks**
    - Write locks may not be shared
      - Only one simultaneous write!
    - A write lock *cannot* be *obtained* if the item is already *read-locked* by any other transaction
    - If the same transaction already holds a *read lock*, the lock can be promoted to a *write lock*
• Most commercial database systems rely on two-phase locking
  
  – Two-phase locking means that for each transaction all necessary locks are acquired before the first lock is released
• When operation **accesses** data item within transaction
  – If item **isn't locked**, then server **locks** and proceeds
  – If item is held in a conflicting lock by another transaction, transaction must **wait** till lock released
  – If item is held by non-conflicting lock, lock is **shared** and operation proceeds
  – If item is already locked by same transaction, lock is **promoted** if possible

• When transaction **commits or aborts**, locks are **released**
Two-phase locking protocols are a simple way to generate only serializable schedules

\[ S := \text{lock}(x) \ r(x) \ \text{lock}(y) \ r(y) \ \text{lock}(p) \ r(p) \ w(p) \ \text{unlock}(p) \ \text{w}(x) \ \text{unlock}(x) \ \text{unlock}(y) \]

- Transactions blue and green interleaved
- Still deadlocks, must be prevented!
  - RDB2!

Dining Philosophers Problem!

- Edward Dijkstra, 1965
- [http://ccl.northwestern.edu/netlogo/models/run.cgi?DiningPhilosophers.790.571](http://ccl.northwestern.edu/netlogo/models/run.cgi?DiningPhilosophers.790.571)
• Philosophers sitting around a round table
  – Each philosopher has a bowl of rice (or spaghetti) and one chopstick (or fork)
    • But you need two chopsticks (forks!?) to eat
  – Idea: Just grab two chopsticks and start
    • The others just wait until you are done
  – But what happen if everybody simultaneously grabs one chopstick?
    • Deadlock!
• **Conservative locking** (also called static locking or preclaiming) acquires all necessary locks before its first read or write
  
  – Restricts concurrency
  – Improves chances of successful commit
  – Only possible in restricted scenario, since read/write sets must be declared to the scheduler in advance
4.0 2-PL

• **Strict two-phase** locking holds all exclusive locks until the respective transaction terminates
  
  – Based on the notion that a running transaction may always need further locks
  
  – Output are only strict schedules that are also interesting for recovery
4.0 Transactions

• Summary “classic” transaction management
  – Flat transactions
  – Most commonly, locking protocols are used
  – Usually, full ACID properties are delivered
    • Only smaller transactions supported
    • Transactions have to be executed fast
      – Too many locks!
    • Limited degree of flexibility
4.1 Distributed Transactions

• Base idea for distributed transaction management: Just generalize known algorithms for distributed environments

• Problems:
  – Transaction may run longer and span multiple nodes
    • Network communication is slow
    • Should operations performed on one node lock resources on other nodes?
    • When somebody really needs a DDBMS, he usually has more complex queries and transactions
      – More powerful transaction models needed?
4.1 Distributed Transactions

– More potential failure sources
  • Node failures
  • Connection failures
  • Message corruption

– No global system time
  • Most time-stamp-based protocols won’t work

– Agreement problems
  • If multiple nodes participate in one transaction, how can all nodes agree on a commit?

– Replication may have been used
  • Is it safe to assume that all replicated fragments contain the same data?
4.1 Distributed Transactions

• **Problem: replication consistency**
  – What happens, if a fragment is *replicated* multiple times?
  – **Mutually consistent data states**
    • All copies of a given data item have identical values
    • Also called *one-copy equivalence*
  – In some cases it may be beneficial to sacrifice one-copy equivalence and allow the replicas to diverge
    • Eventually, all replicas are synchronized
    • So called *eventually consistent* approaches
4.1 Distributed Transactions

- In any case: transaction operations have to be distributed over different nodes
  - Data and resources are distributed!

- **Example**: simple flat transaction
  - Op1 and Op2 are executed at node 1
  - Op3 is executed at node 2
  - Op4 is executed at node 3
4.1 Distributed Transactions

• Basic idea
  – Use a **central** transaction controller handling everything
    • Granting and releasing **locks**
    • Generation of schedules
    • **Aborting** and **committing** transactions
• Obviously, the central controller needs **full access** to all relevant node system resources
  – This is usually only the case in **homogenous distributed** databases
    • What happens in **inhomogeneous**, e.g. federated databases?

• **Base idea**: Delegate responsibility
  – Local DDBMS should be responsible for execution of transactions
  – **Mediator layer** of the DDBMS supervises local execution (middleware transaction manager)
4.1 Distributed Transactions

• **Use nested transactions** for distributing transactions to nodes!
  – Split the transaction hierarchically into multiple **smaller transaction** spanning just one node each
    • **Transaction trees**!
  – Each node handles its own transaction **locally**
    • **Additional operations**; **Vote-Commit** and **Ready-to-Commit / Ready-to-Abort**
  – Transaction manager just **moderates** and alone decides on **final commit** or **rollback**
4.1 Distributed Transactions

- **Original transaction**

- **Final state** broken down on sub-transactions
  - T1: Start – Op1 – Op2 – Commit
  - T2: Start – Op3 – Commit
  - T3: Start – Op4 – Commit

- **Transaction tree**
  - Sub-Transactions could even be further split into sub-sub transactions
4.1 Distributed Transactions

• Executing the transaction tree
  – Execution is initiated from the root transaction
  – Child transactions are recursively started
    • If child transactions are independent, they can be executed in parallel
      – Good for performance!
    • Dependent transactions must be executed sequentially
    • As soon as one child transaction fails, all others child transactions also have to be aborted or rolled-back
      – Failures propagate to the root
      – A single failure forces the whole tree to abort!
4.1 Distributed Transactions

• Assume there are no communication or node failures
  
  – Declare a **single** node as ‘commit coordinator’ (CC)
    • Only the CC will decide about **global** commit/abort
    • The CC initiates a **voting phase** among all nodes
  
  – Every participating node decides **locally** about safe commit or necessary abortion of its local transaction
    • If asked, it will send **either** Ready-to-Commit, **or** Ready-to-Abort
    • Once a decision has been sent **it may not be reversed**
    • In the state **Ready-to-Commit** recovery and commit both have to be possible (Redo/Undo log files!)
4.1 Distributed Transactions

- Commit coordinator

Decision: **Commit**

- Decision: **Abort**

Aborted

send to all **Abort**

receive at least one **Vote-Abort**

Decision: **Commit**

send to all **Commit**

receive all **Vote-Commit**

send out **Begin-Vote**
4.1 Distributed Transactions

- Participating transactions

[Diagram showing the states and transitions for distributed transactions:]
- Received `Begin-Vote`
- Send `Vote-Commit`
- Send `Vote-Abort`
- Receive `Abort`
- Receive `Commit`

States:
- `Aborted`
- `Committed`
• Now also consider network and node failures
  – What if a node does not respond to the **Begin-Vote**?
  – What if a node does not receive further **information**
    from the coordinator?

• Two new phases and new messages
  – **Voting Phase** followed by **Decision Phase**
  – **Time-Out** and **Help-Me**
4.1 Distributed Transactions

- These considerations result in the **Two-Phase-Commit Protocol**
- The **coordinator** starts the voting phase and collects votes...
  - If at least one vote did not arrive after a predefined time interval the coordinator declares a **time-out** and decides for **global abort**
4.1 Distributed Transactions

• If any participant in Ready-to-Commit state does not hear from the coordinator...
  – It declares a **time-out** and sends out **Help-Me** messages to other participating nodes
    • If some other node has **committed**, it must have come from the coordinator thus it is safe to commit
    • If some other node has **aborted**, it is safe to abort
    • If some other node has **not yet voted**, it may (after knowing there is at least one time-out) immediately **Vote-Abort** and thus kill the global transaction
    • If all other nodes are also ready to commit, Help-Me does not help
4.1 Distributed Transactions

• Commit coordinator

- **Commit coordinator**
  - Send out **Begin-Vote**
  - Wait
  - Receive all **Vote-Commit**
  - Decision: **Commit**
  - Send to all **Commit**
  - Receive at least one **Vote-Abort**
  - Or time-out
  - Decision: **Abort**
  - Send to all **Abort**
  - **Aborted**
  - **Committed**
4.1 Distributed Transactions

• Participating nodes

- Start decision phase
  - send Vote-Commit
  - time-out
  - send Vote-Abort

- Ready-to-Commit
  - receive Commit
  - receive Abort
  - send Vote-Commit

- Aborted
  - time-out
  - receive Abort

- Blocked
  - receive Help-Me
  - receive Commit

- Recover
  - Send Help-Me
  - time-out
• In the previous slides, we assumed that sub-transactions are handled locally by the nodes
  – Works fine as long as the sub-transactions are independent
    • If not, no parallelism easily possible
    • Same problem as with transaction schedules in central DBMS! Same solutions possible?
  – Idea: Generalize two phase locking (> D2PL) for a distributed setting!
• **Two Phase Locking (2PL) in a distributed environment**
  
  – **Remember 2PL:**
    
    • First obtain all required locks, then release all locks
  
  – Several types of parties are involved in a distributed 2PL locking scheme
    
    • **Central Lock Manager (LM)**
      
      – Manages which data is locked by which transaction
    
    • **Coordination Manager (CM)**
      
      – Manages the transaction, e.g. obtains locks from LMs and distributes operation to DPs
    
    • **Data Processors (DP)**
      
      – Execute a single operation assigned by CMs
• **Types of lock managers used**

  – **Centralized 2PL**
    - Use a single **central lock manager** for managing all necessary locks

  – **Primary Copy 2PL**
    - **Multiple lock managers**, each responsible for a certain data partition

  – **Distributed 2PL**
    - **Every node** may potentially be a **lock manager**
• Careful with replication
  – If data is replicated, this must be known by the lock managers and transaction managers!
    • Replication Protocol needed!
  – Simple Version:
    • If a lock on a replicated data item is needed, all copies need to be locked
    • If an update is performed on a replicated item, the TM needs to issue updates to all copies!
4.2 D-2PL

- Centralized 2-PL

![Diagram of D-2PL]

- DP
- Coordinating TM
- Central LM

- Request Lock
- Grant Lock
- Release Lock
- Operation
- Done
4.2 D-2PL

• In centralized 2-PL, the **lock manager** is the bottleneck
  – **Scalability issues** with just one lock manager
  – **Central point of failure**
    • No lock manager ⇒ No transactions

• **Primary Copy 2-PL** helps by introducing multiple **lock managers**
  – Each lock manager is responsible for defined partitions of the data

• Finally, fully **distributed 2PL** expects a lock manager at each site
  – Especially suited for dealing with **heavy replication**
  – Each lock manager “knows” its own data and reaches agreements with other lock managers
    • Lock managers coordinate replication
• **Enterprise applications** usually involve multiple data sources
  
  – Transaction may also span **multiple** heterogeneous data sources
    
    • e.g. book a flight within one system and an hotel in another
  
  – Need for **federated transaction management**
    
    • Additional coordination layer necessary, i.e. **transaction manager**
      
      – Usually provided by an **application server**
    
    • All participating databases need a common interface for coordinating transactions
      
      – e.g. XOpen **XA**
4.2 Applications

Application Layer

Application Management Layer

DBMS Layer

ATM

travel agency

app₁

...

appₙ

Transaction Manager

Application Server

DBMS

DB pages

DBMS

DB pages

Encapsulated data

exposed data

Encapsulated data

exposed data

Applications Clients

bookkeeper

ApplicaCons

ApplicaCons Clients

DB pages

exposed data

Encapsulated data

DBMS view

Transaction Manager

Transaction Manager

ApplicaCons

Applications
4.2 Applications

- **Example:** **JTA**
- **Java Transaction API**
- **Uses Application Server**
  - *e.g.* **J2EE Server**
  - Provides centralized **Transaction Manager**
    - Provided by AppServer
  - **User Transaction** interface for applications
  - **XOpen XA Adapter** connecting to databases
4.2 Applications

• **J2EE Application Servers** with JTA Transaction Manager Implementations
  – JBoss
  – Apache Geronimo
  – Sun Glassfish
  – Bea WebLogic Server
  – IBM WASCE
  – Oracle Application Server
  – SAP NetWeaver
  – …
Open Group XOpen XA

- Vendor-spanning standard protocol for Distributed Transaction Processing
- Each DBMS / data source participating within a transaction needs to support XA
- Uses Distributed 2-Phase Locking
- Each DBMS is responsible for maintaining integrity of its own data
  - Centralized transaction manager necessary to coordinate individual commits

4.2 Applications
4.2 Applications

- **Example: JTA-Transactions**

```java
UserTransaction ut = envCtx.lookup("jta/UserTransaction");
DataSource ds = envCtx.lookup("jdbc/Datasource");
// note: explicit transaction handling necessary!
ut.begin();
boolean success = false;
try {
    Connection conn = ds.getConnection();
    // do stuff here
}
finally {
    if (success)
        ut.commit();
    else
        ut.rollback();
}
```
4.3 SAP HANA

A new In-Memory Data Platform:
1. OLTP & OLAP
2. Structured & Unstructured Data
3. Legacy & New Applications

References:
- SAP HANA Tutorial http://saphanatutorial.com/
4.3 SAP HANA

Features:

- SQL
- ACID: isolation (MVCC), logging and recovery
- Stored procedures
- Column store, row store, graph store
- In-memory and disk based
- Efficient compression techniques
- Massive parallelization over CPU cores and nodes
- Data structures optimized for main memory
4.3 SAP HANA

- Motivation.
- HANA database layered architecture.
- Lifecycle Management of Database Records
- Transaction processing
4.3 SAP HANA: motivation

- 1980: Memory $10,000/MB
- 2000: Memory $1/MB
- 2013: Memory $0.004/MB

Time vs. Memory Cost/Speed
4.3 SAP HANA: motivation

**2002**
- 1 core
- 32 bits
- 4MB

**2007**
- 2 cores
- 2 CPUs per server
- External Controllers

**2010**
- 8 cores -16 threads / CPU
- 4 CPUs per server
- On-chip memory control
- Quick interconnect
- VM and vector support
- 64 bits; 256 GB - 1 TB

**2013**
- More cores, bigger caches
- 16 ... 64 CPUs per server
- Greater on-chip integration (PCIe, network, ...)
- Data-direct I/O
- Tens of TBs
4.3 SAP HANA: motivation

Software advances: build for in-memory computing

In-Memory Computing: not just tables but all data-structures
Parallelism: take advantage of the number of cores
On-chip cache awareness
4.3 SAP HANA

<table>
<thead>
<tr>
<th>Order</th>
<th>Country</th>
<th>Product</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>456</td>
<td>France</td>
<td>corn</td>
<td>1000</td>
</tr>
<tr>
<td>457</td>
<td>Italy</td>
<td>wheat</td>
<td>900</td>
</tr>
<tr>
<td>458</td>
<td>Italy</td>
<td>corn</td>
<td>600</td>
</tr>
<tr>
<td>459</td>
<td>Spain</td>
<td>rice</td>
<td>800</td>
</tr>
</tbody>
</table>

Typical Database

```
SELECT Country, SUM(sales) FROM SalesOrders
WHERE Product = 'corn'
GROUP BY Country
```
4.3 SAP HANA

Column „Name“ (uncompressed)

Miller
Jones
Millman
Zsuwalski
Baker
Miller
John
Miller
Johnson
Jones

Column „Name“ (dictionary compressed)

Value-ID sequence
One element for each row in column

Value IDs

4
1
5
N
0
4
2
4
3
1

Dictionary

0 Baker
1 Jones
2 John
3 Johnson
4 Miller
5 Millman

Value ID implicitly given by sequence in which values are stored

point into dictionary

sorted

Value

 Deztour
4.3 SAP HANA Database

Business Applications

Connection and Session Management

- SQL
- SQL Script
- MDX
- WIPE (Graphs)
- FOX (Planning)

Calculation Engine

Optimizer and Plan Generator

Execution Engine

In-Memory Processing Engines

- Relational Engine (Row and Column Store)
- Graph Engine
- Text Engine

Persistency Layer

Logging and Recovery

Page Management
Persistency mechanisms of the unified table

4.3 SAP HANA

common unified table access methods

bulk inserts
update/insert/delete

incremental merge

L1-delta

L2-delta

partial/full merge

main store

REDO log

savepoint data area
• Advantages of column-based tables
  – Faster data access
  – Better compression
  – Better parallel processing
• **Transaction processing:**
  
  – Multi-Version-Concurrency-Control (MVCC)
  
  – Distributed snapshot isolation
- Consistent view manager decides on the visibility of records per table.
  - Transaction token
• Distributed Snapshot Isolation Optimization

  – Optimization for worker-node local read transactions or statements.
  – Optimization for worker-side local write transactions.
  – Optimization for multi-node write transactions.
• Optimizing Two-Phase Commit Protocol
  – Early commit acknowledgment after the first commit phase.
  – Skipping writes of prepare logs.
  – Group two-phase commit protocol.
• Adapt methods already known from \textbf{centralized transactions management}
  – But: distributed databases have more potential failure sources
    • Network failures, replication, allocation, node failure, untrustworthiness,…
  – One approach: provide full ACID properties
    • Federated approach: use distributed commits (D2PC)
      – i.e. partition global transaction into \textbf{sub-transactions}
      – Each sub-transaction is executed \textbf{locally}
      – At the end, the \textbf{coordinator} votes if final commit should be performed
        » Sub-transaction either OK or failed
        » One sub-failure $\Rightarrow$ global failure
Homogenous approach: **Distributed 2-Phase-Locking (D2PL)**
- Adapt 2PL for distributed usage
- Distributed lock management necessary
- Control over internal transaction management of nodes necessary

Problem: how to deal with **untrustworthy** nodes?
- Important in P2P or in loosely-coupled autonomous settings
  - e.g. web services
- Nodes may be **malicious** or just **malfuctioning**
- Byzantine Agreements!
  - Nodes **echo** received messages among each other to filter false information and untrustworthy nodes
• Recommended Reading:
• **Peer-Two-Peer Systems**
  
  – Classification of Peer-To-Peer Systems
  – Decentralized and Centralized P2P
  – Structured and Unstructured P2P
  – Early protocols