Distributed Data Management

Christoph Lofi
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
• We will discuss three major distributed data systems
  – **Distributed Databases**
    • “Classic” approach to data distribution
    • Provides full DB feature sets
      – Transactions, query languages, schemas, etc.
    • Limited flexibility, elasticity, and scalability
  – **P2P Systems**
    • Overlay network
    • Data is stored decentrally on users’ PCs
    • Very limited feature sets in exchange for very high scalability and flexibility
– Cloud Storage & NoSQL Systems
  • Fusion of DDBMS and P2P technologies
  • Tailored for data center usage
  • Main goals: Cheap large-scale data centers
    – Focus on low-end hardware
    – High hardware elasticity
    – Near-linear performance and storage scaling
    – Flexible schemas
    – High fault tolerance
Summary

Network & hardware features

- low (elasticity, extensibility, flexibility, volatility)
- high

DB features

- many
- few

(data model complexity, query power, storage guarantees, transactional support)

P2P

Cloud

Distributed DB
2.0 Introduction

2.0 Sharing Architectures
2.1 Fragmentation
2.2 Partitioning Techniques
2.3 Allocation Techniques
A distributed database runs on several nodes

- Distributed architecture can be classified based on what is shared between the nodes

  • **Shared secondary storage?**
    - Can nodes access the same location on secondary storage?
    - e.g. shared hard drives, raid clusters, storage area networks (SAN)?

  • **Geographical distribution?**
    - Are nodes at the same location or geographically scattered?

  • **Node coupling?**
    - How strongly are nodes coupled?
      » Usually, this means: “Do they share the same main memory?”
2.0 Sharing Architectures

- Secondary Storage
  - Shared
  - Independent

- Geographic Distribution
  - Local
    - Close
    - Loose
  - Distributed
    - Loose

- Node Coupling
  - Share-Everything
  - Shared-Disk
  - Share-Nothing
2.0 Sharing Architectures

• **Sites, Nodes and Node Groups: Definitions**
  
  – A **node** is a single computational device
    • Depending on point of view a computer, a server, a blade, or a CPU
  
  – A node group is a **logical group of nodes**
    • A node group is usually shares some common agenda
      – e.g. a node group might be responsible for replicating a certain data fragment
    • A node might be member of multiple node groups
  
  – A **site** is a group of nodes which are in the same location
    • Nodes in a site are considered local and are usually connected via high-performance network
      – Usually, sites refers to individual **data centers**
    • A node is thus a member of at most one site
• **Share-Everything Architectures**
  – Early approaches to DDBMS
  – Each node can access a shared **main memory** and **secondary storage**
    • **Scalability problems** as main memory easily becomes the bottleneck, e.g. limited capacity, limited performance with parallel access (locking, latching)
    • **Reliability problems** as the main memory is a potential single point of failure
2.0 Sharing Architectures

– However, recently share-everything architectures become popular again within massively parallel computer clusters
  • Enabling technology: remote direct memory access (RDMA)
    – Individual nodes do have own main memory
    – But main memory of other machines can be directly accessed via low-latency high-performance network
    – Results in multi-machine NUMA architectures
      » e.g. InfiniBand network
  • Usually used to implement high-performance parallel DDBMS
    – Also, often specialized and customized data storage engines are used, e.g. all-to-all message data rings
2.0 Sharing Architectures

• Share-everything architectures are highly reliant on hardware, and they come in different flavors
• There are two major approaches for designing shared-everything architectures
  – **SMP**: Symmetric Multiprocessor Systems
  – **NUMA**: Non-uniform Memory Access
2.0 Sharing Architectures

• **Symmetric Multiprocessor Systems**
  
  – Centralized Main memory shared with multiple homogeneous processors under central control
    
    • Each processor may have an own cache, and is connected via a central bus to memory and IO
  
  – **Advantage:**
    
    • Any processor can work on any task, good for flexibility and load balancing
  
  – **Disadvantage:**
    
    • The bus quickly becomes a bottleneck, resulting in bad scaling
    
    • Processors are faster than memory anyway. In SMP, one processor can block data to another one, stalling the system.
2.0 Sharing Architectures

• **Non-Uniform-Memory-Access**
  – Main memory is partitioned
  – All processors can access all memory, but close memory can be accessed more efficiently
  – Disadvantage:
    • Requires careful partitioning of data, resulting in complex programming
  – Advantages:
    • Can have very good scaling if problem can be partitioned and distributed effectively
2.0 Sharing Architectures

**SMP**

**NUMA**

- CPU
- Cache
- Main Memory
- I/O

- CPU
- Cache
- Main Memory
- I/O
2.0 Sharing Architectures

SMP

NUMA

CPU

Cache

Cache

Cache

CPU

Cache

Cache

Cache

Main Memory

I/O

Main Memory

I/O

CPU

Cache

Cache

Cache

CPU

Cache

Cache

Cache

CPU

Cache

Cache

Cache

CPU

Cache

Cache

Cache

CPU
• Example:
  – NUMA-on-a-Chip: AMD Bulldozer CPU
2.0 Sharing Architectures

• Shared-Disk Architecture
  – Nowadays, most common architecture for enterprise level DDBMS
    • Good performance for complex data and transactions
  – Usually, data is stored in a shared high-performance SAN
• Share-Nothing Architecture
  – Nodes don’t share any storage and communicate via network
    • If data is needed which is not present at current node, it needs to be shipped (high communication costs)
    • If data can be distributed in such a way that no shipping is necessary, theoretically linear scalability is possible
      – Rare requirement for “traditional” enterprise applications…
  – Nowadays, most notable application for shared-nothing architectures are web-age applications
    • e.g. Google, Yahoo, etc…
• Share-Nothing Architecture

• Which of the architecture is “best” depends highly on the application requirements
## 2.0 Sharing Architectures

### Shared Disk vs. Shared Nothing

- **Shared Nothing may perform extremely well if inter-node processing / data shipping / transactions can be avoided**

<table>
<thead>
<tr>
<th></th>
<th>Shared Disk</th>
<th>Shared Nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Setup &amp; Maintenance</td>
<td>Easy</td>
<td>Continuous Distribution / Redistribution</td>
</tr>
<tr>
<td>Data Shipping</td>
<td>Cheap</td>
<td>Expensive</td>
</tr>
<tr>
<td>Transaction Costs</td>
<td>Cheap</td>
<td>Expensive</td>
</tr>
<tr>
<td>Node Latency</td>
<td>Moderate</td>
<td>Low</td>
</tr>
<tr>
<td>Parallel Processing</td>
<td>Intra-Node: Good</td>
<td>Intra-Node: Very Good</td>
</tr>
<tr>
<td></td>
<td>Inter-Node: Okay</td>
<td>Inter-Node: Bad</td>
</tr>
<tr>
<td>Hardware Costs</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Maximum Size</td>
<td>Large</td>
<td>Extremely Large</td>
</tr>
</tbody>
</table>
2.1 Fragmentation

• In shared-nothing architectures, data has to be **distributed** across the nodes. But how?

  – **Fragmentation**
    • Relations are decomposed into smaller, disjunctive **fragments**. These fragments are **distributed** across the nodes.

  – **Replication**
    • Relation fragments are **replicated** and **copied** across the nodes.

• Of course, **hybrid approaches** are possible
2.1 Fragmentation

• First, we consider **fragmentation**. Several major issues arise:

  – **Fragmentation Granularity**
    • How big should a fragment be?
    • Which parts of the relation should be assigned to which fragment?

  – **Allocation**
    • Which fragments should be assigned to which node?
    • Which fragments should be replicated, which should only be stored once?

• If each fragment is only stored **once** (i.e. no replication used), this is called **partitioning**
2.1 Fragmentation

- Decompose R to **fragments** and **allocate** to nodes
2.1 Fragmentation

- Fragmentation decomposes a relation $R$ into multiple fragments $F_R := \{R_1, R_2, R_3, ..., R_n\}$
- Proper fragmentation has to obey some correctness rules
  - Rules vary wrt. the chosen fragmentation strategy
  - **Completeness Rule**
    - Fragments contain all data
  - **Disjointness Rule**
    - Fragments do not overlap
  - **Reconstruction Rule**
    - In general, fragmentation must *preserve* the data and properties of the original relation
      - e.g. tuples and attributes, constraints and keys, etc
## 2.1 Fragmentation

### Entities

<table>
<thead>
<tr>
<th>E_ID</th>
<th>EName</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>Poseidon</td>
<td>God</td>
</tr>
<tr>
<td>E2</td>
<td>Hyperion</td>
<td>Titan</td>
</tr>
<tr>
<td>E3</td>
<td>Hercules</td>
<td>Hero</td>
</tr>
<tr>
<td>E4</td>
<td>Hydra</td>
<td>Monster</td>
</tr>
<tr>
<td>E5</td>
<td>Mnemosyne</td>
<td>Titan</td>
</tr>
<tr>
<td>E6</td>
<td>Athena</td>
<td>God</td>
</tr>
<tr>
<td>E7</td>
<td>Theseus</td>
<td>Hero</td>
</tr>
<tr>
<td>E8</td>
<td>Cronus</td>
<td>Titan</td>
</tr>
</tbody>
</table>

### Entity Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>God</td>
<td>Immortal</td>
</tr>
<tr>
<td>Titan</td>
<td>Tough</td>
</tr>
<tr>
<td>Hero</td>
<td>Mortal</td>
</tr>
<tr>
<td>Monster</td>
<td>Ugly</td>
</tr>
</tbody>
</table>

### Artifacts

<table>
<thead>
<tr>
<th>E_ID</th>
<th>C_ID</th>
<th>Artifact</th>
<th>Grandiosity</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>C1</td>
<td>Temple</td>
<td>2</td>
</tr>
<tr>
<td>E2</td>
<td>C1</td>
<td>Statue</td>
<td>1</td>
</tr>
<tr>
<td>E2</td>
<td>C2</td>
<td>Statue</td>
<td>3</td>
</tr>
<tr>
<td>E3</td>
<td>C3</td>
<td>Mosaic</td>
<td>2</td>
</tr>
<tr>
<td>E3</td>
<td>C4</td>
<td>Statue</td>
<td>1</td>
</tr>
<tr>
<td>E4</td>
<td>C2</td>
<td>Mosaic</td>
<td>2</td>
</tr>
<tr>
<td>E5</td>
<td>C2</td>
<td>Monument</td>
<td>1</td>
</tr>
<tr>
<td>E6</td>
<td>C4</td>
<td>Temple</td>
<td>4</td>
</tr>
<tr>
<td>E7</td>
<td>C3</td>
<td>Monument</td>
<td>2</td>
</tr>
<tr>
<td>E8</td>
<td>C3</td>
<td>Statue</td>
<td>1</td>
</tr>
</tbody>
</table>

### Cities

<table>
<thead>
<tr>
<th>C_ID</th>
<th>CName</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Athens</td>
<td>Democracy</td>
<td>Attica</td>
</tr>
<tr>
<td>C2</td>
<td>Sparta</td>
<td>Monarchy</td>
<td>Laconia</td>
</tr>
<tr>
<td>C3</td>
<td>Thebes</td>
<td>Oligarchy</td>
<td>Boeotia</td>
</tr>
<tr>
<td>C4</td>
<td>Corinth</td>
<td>Democracy</td>
<td>Arcadia</td>
</tr>
</tbody>
</table>
## 2.1 Fragmentation

- **Horizontal Partitioning**
  - Relation is split horizontally, tuples are distributed.

<table>
<thead>
<tr>
<th>C_ID</th>
<th>CName</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Athens</td>
<td>Democracy</td>
<td>Attica</td>
</tr>
<tr>
<td>C2</td>
<td>Sparta</td>
<td>Monarchy</td>
<td>Laconia</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C_ID</th>
<th>CName</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>Thebes</td>
<td>Oligarchy</td>
<td>Boeotia</td>
</tr>
<tr>
<td>C4</td>
<td>Corinth</td>
<td>Democracy</td>
<td>Arcadia</td>
</tr>
</tbody>
</table>
2.1 Fragmentation

• Typically, horizontal distribution follows one or several of the following techniques

  – **Range Partitioning:**
    • Each fragment is responsible for another value range, e.g.:
      – fragment₁: t.city_population ≥ 100,000
      – fragment₂: t.city_population < 100,000

  – **Value Lists**
    • Provide the attribute values for each partition
      – fragment₁: t.city_location ∈ {Attica, Laconia}
      – fragment₂: t.city_location ∈ {Boetia, Arcadia}

  – **Hash Values**
    • Provide a hash function on tuples. Each fragment is responsible for a given hash range.
      – fragment₁: 0 ≤ h(t) ≤ 50
      – fragment₂: 50 < h(t) ≤ 100
2.1 Fragmentation

• **Horizontal Completeness Rule**
  – Every tuple of $R$ shows up in one of the fragments
  – $R = R_1 \cup R_2 \cup \cdots \cup R_n$

• **Horizontal Disjointness Rule**
  – Each tuple may only appear once in all fragments
  – $\forall 1 \leq i \neq j \leq n$: $R_i \cap R_j = \emptyset$

• **Horizontal Reconstruction Rule**
  – Primary keys must remain unique
  – Foreign keys must be preserved
2.1 Fragmentation

- **Vertical Partitioning**
  - Relation is split vertically; attributes are distributed
  - Primary key attributes are replicated

<table>
<thead>
<tr>
<th>C_ID</th>
<th>CName</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Athens</td>
<td>Democracy</td>
</tr>
<tr>
<td>C2</td>
<td>Sparta</td>
<td>Monarchy</td>
</tr>
<tr>
<td>C3</td>
<td>Thebes</td>
<td>Oligarchy</td>
</tr>
<tr>
<td>C4</td>
<td>Corinth</td>
<td>Democracy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C_ID</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Attica</td>
</tr>
<tr>
<td>C2</td>
<td>Laconia</td>
</tr>
<tr>
<td>C3</td>
<td>Boeotia</td>
</tr>
<tr>
<td>C4</td>
<td>Arcadia</td>
</tr>
</tbody>
</table>
2.1 Fragmentation

- **Vertical Completeness Rule**
  - Every attribute of $R$ shows up in one of the fragments

- **Vertical Disjointness Rule**
  - Each non-primary-key attribute appears only once in all fragments
  - The primary key attributes are part of all fragments of a given relation

- **Vertical Reconstruction Rule**
  - Joins must be complete, i.e. $R = R_1 \Join R_2 \Join \cdots \Join R_n$
2.1 Fragmentation

- Hybrid Partitioning
  - Use Vertical and Horizontal Partitioning

\[ R = (\text{Cities}_{1,1} \bowtie \text{Cities}_{2,1}) \cup (\text{Cities}_{1,2} \bowtie \text{Cities}_{2,2}) \]
2.1 Fragmentation

- **Allocation of Fragments**
  - When allocating the fragments to nodes, it must be decided whether and which fragments should be replicated

- **Major advantages of Replication**
  - Read-only queries can be executed in parallel with reduced communication costs
  - Resilience vs. system failures

- **Major disadvantages of Replication**
  - High update costs, potential problems with consistency
  - Increased demand of storage capacity
2.1 Fragmentation

- Usually, **partial replication** is used in DDBMS
  - Which fragments to replicate usually depends on
    - Query / Update ratio
    - The actual query load, i.e. which fragments are actually needed by common queries

<table>
<thead>
<tr>
<th></th>
<th>Full Replication</th>
<th>Partial Replication</th>
<th>Partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Processing</td>
<td>Easy</td>
<td>Difficult</td>
<td>Difficult</td>
</tr>
<tr>
<td>Directory Management</td>
<td>Easy</td>
<td>Difficult</td>
<td>Difficult</td>
</tr>
<tr>
<td>Concurrency Control</td>
<td>Moderate</td>
<td>Difficult</td>
<td>Easy</td>
</tr>
<tr>
<td>Reliability</td>
<td>Very High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Parallel Query Potential</td>
<td>Very High</td>
<td>High</td>
<td>Moderate</td>
</tr>
<tr>
<td>Parallel Update Potential</td>
<td>Very Low</td>
<td>Moderate</td>
<td>High</td>
</tr>
<tr>
<td>Applicability</td>
<td>Possibly</td>
<td>Realistic</td>
<td>Possibly</td>
</tr>
<tr>
<td>Storage Efficiency</td>
<td>Very Low</td>
<td>Moderate</td>
<td>Very High</td>
</tr>
</tbody>
</table>
2.2 Partitioning Techniques

• How can we decide how to perform the fragmentation?
  
  – Actually, **fragmentation** and **allocation** can be described as complex **minimization problems**
  
  – **Minimize the execution cost of the applications using the DDBMS**
    
    • Minimalize query and update costs
    • Maximize possible degree of parallelization
      – All data needed to answer a query / perform an update should be located on the same node
      – Minimize communication costs
    • Respect additional **durability** constraints?
– **Qualitative** and **quantitative** information on data and applications is usually needed

  • **Qualitative** information (e.g. schema, query predicates) usually used for **fragmentation**

  • **Quantitative** information (e.g. query load) usually used for **allocation**

• Main goal is to partition in such a way that all applications perform **optimally**
  – e.g. application queries are executed fast and in parallel
2.2 Partitioning Techniques

• An intuitive heuristic is that fragments should contain “chunks” of data which are accessed as a unit by an application / query
  
  – For horizontal partitioning, these chunks may be defined by query predicates
    
    • Basically, Boolean expressions for tuple selection
2.2 Partitioning Techniques

• If the fragments are defined badly, system performance may suffer severely
  – What that means may differ from application to application…

• Possible design goals for a good fragmentation and allocation:
  – Optimization for parallel access
    • Distribute tuples such that a query poses a balanced workload to all nodes; subresults are shipped to query node
  – Optimization for low communication costs
    • A query should create workload only on one single node to avoid result shipping
2.2 Partitioning Techniques

– Optimization for high availability
  
  • Usually, includes high **geographic distribution** and **replication**
  
  • Data should be available at several sites such that, no matter where the query originated, **there is a site nearby** which can process the query
  
  • Data should be **resilient to failure**, e.g. if nodes fail, there should be no data loss
  
  • Especially, system should be **resilient to site failure**
    
    – e.g. even the loss of a data center should have no fatal impact
2.2 Partitioning Techniques

• Types of simple *partitioning*
  – Primary Horizontal Partitioning
    • *Horizontal* partitioning is given by predicates defined on just the current relation
  – Derived Horizontal Partitioning
    • *Horizontal* partitioning is given by predicates defined on another relation
  – Vertical Partitioning
    • Partition the attributes of a relation
  – Hybrid approaches
• **Primary Horizontal Partitioning**
  – The fragments of $R$ are determined by a selection operation using a given set of selection predicates
  – $R_i = \sigma_{P_i}(R), \ 1 \leq i \leq n$

• How can good predicates for defining a horizontal partitioning be found?
  – Manually
  – Automatically at design time
  – Automatically during runtime
2.2 Horizontal Partitioning

• **Manual horizontal partitioning**
  – Database administrator uses *semantic knowledge* of data usage
    • e.g. in DDBMS of a big banking company, partition data by regions as most payment transactions are between parties of the same region

• **Automatics horizontal partitioning at design time**
  – Commonly used approach in enterprise databases
  – *Anticipate information demand and frequency* of common user queries
  – Derive an “optimal” fragmentation such that the estimated overall performance is maximal
    • Often, just *hash partitioning* is used as this results in a simple implementation and predictable performance

• What happens if usage of system changes over time?
2.2 Horizontal Partitioning

• **Automatically** during runtime
  – The system *automatically* decides on a good partitioning scheme
    • Usually, by relying on *usage statistics*
  – No *administrative* input necessary
    • Allows for *elasticity* (nodes may be added and removed without administrative overhead)
  – Good runtime fragmentation is **difficult**
    • Often found in cloud *storage systems* which have *simpler data* and query requirements
    • Will be presented in the cloud part of the lecture
2.2 Horizontal Partitioning

• A simple scheme for automatic horizontal design time partitioning
  – Define the user queries and the frequency of those queries
  – Rewrite the selection condition of the query in disjunctive normal form
    • Disjunction of conjunctions
    • Every Boolean expression can be rewritten
  – Extract the set of all minterms
    • minterms are the terms of the conditions in disjunctive normal form only using negation and conjunction
2.2 Horizontal Partitioning

- **Example:** *minterms*
  
  - $q_1$: `SELECT type FROM entities WHERE name = V`
  
  - $q_2$: `SELECT type FROM artifacts WHERE grandiosity NOT BETWEEN $V_1$ AND $V_2$

- **Conditions in disjunctive normal form:**
  
  - $q_1$: `name = V`
  
  - $q_2$: $\neg (\text{grandiosity} \geq V_1 \land \text{grandiosity} \leq V_2)$
    
    $\equiv \neg \text{grandiosity} \geq V_1 \lor \neg \text{grandiosity} \leq V_2$

- *minterms* = \{`name = V, $\neg \text{grandiosity} \geq V_1, \neg \text{grandiosity} \leq V_2$\}
• Simple **automatic horizontal partitioning** (cont.)
  
  – After the set of minterms is established, estimate the **selectivity** of each **minterm**
    
    • **Selectivity**
      – How many tuples are probably selected by each minterm?
      – How high is the probability of a tuple being selected by a minterm?

  – Now, find **minimal** and **complete** sets of minterms for defining fragments
    
    • **Minimal**: At least one query accesses a fragment
    • **Complete**: The access probability for each tuple within a fragment is similar
2.2 Horizontal Partitioning

• Example: Partition cities
  – Use minterm-sets: \{\text{location}=\text{Boetia}\}, \{\text{Location}=\text{Peloponnesse}\}, \{\text{Location}=\text{Attica}\}\}
  – Query: \text{SELECT * FROM cities WHERE c\_id=V}

• Complete and minimal
  – Each fragment could be accessed
  – Within each fragment, probability for selecting a tuple is similar

<table>
<thead>
<tr>
<th>C_ID</th>
<th>CName</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>Thebes</td>
<td>Oligarchy</td>
<td>Boeotia</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C_ID</th>
<th>CName</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>Sparta</td>
<td>Monarchy</td>
<td>Peloponnesse</td>
</tr>
<tr>
<td>C4</td>
<td>Corinth</td>
<td>Democracy</td>
<td>Peloponnesse</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C_ID</th>
<th>CName</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Athens</td>
<td>Democracy</td>
<td>Attica</td>
</tr>
</tbody>
</table>
2.2 Horizontal Partitioning

• Example: Partition cities
  – Use minterm-sets: \{\{location=Boetia\},
    \{Location=Peloponnesse\}, \{Location=Attica\}\}
  – Query: SELECT * FROM cities WHERE c_id < c4
    • Not complete
      – C2 has higher probability (1.0) than C4 (0.0)

<table>
<thead>
<tr>
<th>C_ID</th>
<th>CName</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>Thebes</td>
<td>Oligarchy</td>
<td>Boeotia</td>
</tr>
<tr>
<td>C1</td>
<td>Athens</td>
<td>Democracy</td>
<td>Attica</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C_ID</th>
<th>CName</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>Sparta</td>
<td>Monarchy</td>
<td>Peloponnesse</td>
</tr>
<tr>
<td>C4</td>
<td>Corinth</td>
<td>Democracy</td>
<td>Peloponnesse</td>
</tr>
</tbody>
</table>
Naively, the algorithm iteratively generates all possible fragmentation and keeps the best one

- **Optimization problem**
- \(2^n\) possible fragmentations; \(n\) number of minterm predicates
- Algorithm aims at distributing data equally wrt. to the queries
  - Good idea when data shipping is cheap and high parallelism necessary – bad idea otherwise
- You may refer to the Tamer Özsu textbook for more detail…
• Up to now, we just investigated a single relation. What happens if multiple relations are involved?
• Foreign Key Dependencies in **Join Link Graphs**
  – Each link defines a **foreign key dependency**
    • The source of a link is called **owner relation**
    • The target of a link is called **member relation**
2.2 Horizontal Partitioning

• Derived Horizontal Partitioning
  – Given: relation $R$ and relation $S$
    • There is a foreign key link from $S$ to $R$
      – $S$ is owner, $R$ is member
    • $S$ is already partitioned in $S_1, S_2, \ldots, S_n$
  – Relation $R$ is partitioned with respect to the fragments of $F$
    • Use semijoin operator!
    • $R_i := R \bowtie S_i, 1 \leq i \leq n$
2.2 Horizontal Partitioning

**Example:**

- Let the relation $EntityTypes$ be partitioned as:
  
  $EntityTypes_1 = \sigma_{Type=God \lor Type=Titan} EntityTypes$
  
  $EntityTypes_2 = \sigma_{Type=Monster \lor Type=Hero} EntityTypes$

<table>
<thead>
<tr>
<th>Type</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>God</td>
<td>Immortal</td>
</tr>
<tr>
<td>Titan</td>
<td>Tough</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hero</td>
<td>Mortal</td>
</tr>
<tr>
<td>Monster</td>
<td>Ugly</td>
</tr>
</tbody>
</table>

- *Entity* is thus partitioned to:

<table>
<thead>
<tr>
<th>E_ID</th>
<th>EName</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>E3</td>
<td>Hercules</td>
<td>Hero</td>
</tr>
<tr>
<td>E4</td>
<td>Hydra</td>
<td>Monster</td>
</tr>
<tr>
<td>E7</td>
<td>Theseus</td>
<td>Hero</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E_ID</th>
<th>EName</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>Poseidon</td>
<td>God</td>
</tr>
<tr>
<td>E2</td>
<td>Hyperion</td>
<td>Titan</td>
</tr>
<tr>
<td>E5</td>
<td>Mnemosyne</td>
<td>Titan</td>
</tr>
<tr>
<td>E6</td>
<td>Athena</td>
<td>God</td>
</tr>
<tr>
<td>E8</td>
<td>Cronus</td>
<td>Titan</td>
</tr>
</tbody>
</table>
2.2 Horizontal Partitioning

• Considerations
  – This type of partitioning may easily lead to incomplete fragmentations
    • *NULL* values in the member relation, e.g. \{E9, Pegasus, NULL\}
  – A relation might be member of multiple dependency links. Which candidate fragmentation to choose?
    • Fragmentation with better join characteristics
    • Fragmentation used in more applications
      – Simple, just count…
2.2 Horizontal Partitioning

• “Fragmentation with better join characteristics”
  – The **join performance** in a DDBMS benefits when
    • The relations or fragments to be joined are **small** (few tuples)
    • Joins should be executed on a **single node**
      – But sometimes, the opposite is desirable…
  – Informal approach: **Fragment Join Graphs**
    • Each fragment is a node in this graph
    • If a join between two fragments might produce a non-empty result, the corresponding nodes are connected
    • The fewer links there are, the better the fragmentation is
• Good: **Simple Join Graph**

![Diagram of simple join graph with nodes R1, R2, R3, R4 connected to nodes S1, S2, S3, S4. The diagram illustrates the partitioning of R and S into fragments to optimize join operations.]
2.2 Horizontal Partitioning

- **Bad:** Total Join Graph

![Diagram of Total Join Graph]

- Fragments of $R$:
  - $R_1$
  - $R_2$
  - $R_3$
  - $R_4$

- Fragments of $S$:
  - $S_1$
  - $S_2$
  - $S_3$
  - $S_4$
• Okay: **Partitioned Join Graph**
2.2 Horizontal Partitioning

- Hash-Based Partitioning
  - Hash-based **partitioning** is especially important for peer-to-peer systems
    - So will cover it in detail in the P2P part of the lecture
  - **Base idea:**
    - Hash function creates hash for tuple
    - Each node is responsible for a given hash range
2.2 Horizontal Partitioning

• Hash Partitioning: **Pro**
  – **Automatic Partitioning**
  – Easy to implement
  – No semantic knowledge necessary
  – Easy load balancing
  – Stable performance without nasty surprises

• Hash Partitioning: **Con**
  – Does not use semantic knowledge
    • Performance may be suboptimal
  – Ignores actual query load
    • Again: Performance may be suboptimal
2.2 Vertical Partitioning

- **Vertical Partitioning**
  - In vertical partitioning, fragments contain a subset of the attributes of \( R \) as well as the primary key attributes
  - Fragmentation problem is more complex than horizontal fragmentation

- More different fragmentations are possible
  - Horizontal: \( 2^n \) fragmentation; \( n \) number of minterm predicates
  - Vertical: \( B(m) \); \( m \) is number of non-primary key attributes; \( B(m) \) is \( m \)th Bell number; e.g. \( B(10) \approx 10^5, B(15) \approx 10^9, B(30) \approx 10^{23} \)

- **Heuristic** approach necessary!
2.2 Vertical Partitioning

– **Grouping Heuristics**
  
  • Create a fragment for each non-primary-key attribute
  
  • Join the fragments according a given heuristic until some criterion is fulfilled

– **Splitting Heuristics**

  • Start a fragment with the whole relation
  
  • Use heuristic to find beneficial splits until some criterion is fulfilled
  
  • Usually shows better performance
2.2 Vertical Partitioning

- Vertical partitioning can formally be defined using the projection operator
  - Set of fragments of R
    \[ F_R = \{R_1, R_2, R_3, \ldots, R_n\} \]
  - Attribute sets of each fragment
    \[ A_R = \{A_1, A_2, A_3, \ldots, A_n\} \]
    - with \( \forall 1 \leq i \neq j \leq n: A_j \cap A_i = \emptyset \)
  - Fragments
    \[ R_i = \pi_{A_i} R, \quad 1 \leq i \leq n \]
• Again, the idea is to group attributes into fragments which are “usually accessed together” by applications

• How could such an fragmentation be performed?
  – **Statistics needed!**
  • Which queries are executed by applications?
  • Which **attributes** are accessed by queries?
  • Which queries are executed **how often**?
  • Which attributes are queried together?
  
  – **Cluster** attributes such that related attributes are in the same fragment
2.2 Vertical Partitioning

• A simple technique to capture the necessary information are
  
  – **Attribute Usage Matrix**
    • Which queries use which attribute?
  
  – **Attribute Affinity Matrix**
    • How closely related are attributes?
    • Results from Usage Matrix and query statistics
### 2.2 Vertical Partitioning

- Building the attribute usage matrix:
  - **Elicit typical application queries**
    - “Find political type of a city given its id”
    - \( q_1 = \text{SELECT type FROM cities WHERE c_id = V } \)

<table>
<thead>
<tr>
<th>C_ID</th>
<th>CName</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Athens</td>
<td>Democracy</td>
<td>Attica</td>
</tr>
<tr>
<td>C2</td>
<td>Sparta</td>
<td>Monarchy</td>
<td>Laconia</td>
</tr>
<tr>
<td>C3</td>
<td>Thebes</td>
<td>Oligarchy</td>
<td>Boeotia</td>
</tr>
<tr>
<td>C4</td>
<td>Corinth</td>
<td>Democracy</td>
<td>Arcadia</td>
</tr>
</tbody>
</table>
2.2 Vertical Partitioning

– “Find names and type of all cities”
  • \( q_2 = \text{SELECT } \text{cname}, \text{type } \text{FROM } \text{cities} \)

– “Find all cities in a given area”
  • \( q_3 = \text{SELECT } \text{cname } \text{FROM cities WHERE } \text{location} = V \)

– “How many democratic cities are in an area?”
  • \( q_4 = \text{SELECT count(type) FROM cities WHERE location} = V \text{ and type} = \text{“Democracy”} \)

• Keep in mind!
  – In most DBs, 20% of all queries produce 80% of all load → just using the most important queries is OK
2.2 Vertical Partitioning

- Which attributes are used in which query?
  - Constructing the **Attribute Usage Matrix** (use)

- \[ use(q_i, A_j) = \begin{cases} 
1 & \text{iff } q_i \text{ uses } A_j \\
0 & \text{otherwise} 
\end{cases} \]

- Resulting usage matrix:

\[
use = \begin{bmatrix}
q_1 & A_1 & A_2 & A_3 & A_4 \\
q_2 & 1 & 0 & 1 & 0 \\
q_3 & 0 & 1 & 1 & 0 \\
q_4 & 0 & 0 & 1 & 1
\end{bmatrix}
\]
2.2 Vertical Partitioning

• Which attributes belong together?
  – Constructing the **Attribute Affinity Matrix** (aff)
  – Base idea: Create a weighted attribute-attribute matrix from the **query statistic**
    • Each cell describes how often one attribute is used alongside another
    • Query Statistic Vector (which query is executed how often)
      \( qstat = \{45, 5, 75, 3\} \)
      – i.e. \( q_2 \) has been executed 5 times
2.2 Vertical Partitioning

- **Attribute Affinity Matrix**
  - Compute cell $aff(A_i, A_j)$
    - Count how often all queries are executed which use attribute $A_i$ as well as attribute $A_j$
      - from attribute usage matrix
    - $aff(A_i, A_j) = \sum_{k:(use(q_k,A_i)=1 \land use(q_k,A_j)=1)} qstat(q_k)$
  - Example:
    - $q_1$ is the only query to access $A_1$ and $A_3$ at the same time, thus $aff(A_1, A_3) = qstat(q_1) = 45$
    - $A_2$ is accessed by $q_2$ and $q_3$, thus $aff(A_2, A_2) = qstat(q_2) + qstat(q_3) = 80$
2.2 Vertical Partitioning

- **Attribute Affinity Matrix** (Example cont.)

\[
\text{aff} = \begin{bmatrix}
A_1 & A_2 & A_3 & A_4 \\
A_1 & 45 & 0 & 45 & 0 \\
A_2 & 0 & 80 & 5 & 75 \\
A_3 & 45 & 5 & 53 & 3 \\
A_4 & 0 & 75 & 3 & 78 \\
\end{bmatrix}
\]
After the attribute affinity matrix is obtained, the matrix can be clustered

- Reorganize columns and rows such that similar elements are close together
- Use some clustering algorithm
  - e.g. complicated bond energy algorithm
- Result: Clustered Affinity Matrix
  - Detour: See Özsu-Valduriez book
2.2 Vertical Partitioning

- Clustered Attribute Affinity Matrix
  – (Example cont.)

\[
\text{aff} = \begin{bmatrix}
A_1 & A_3 & A_2 & A_4 \\
A_1 & 45 & 45 & 0 & 0 \\
A_3 & 45 & 53 & 5 & 3 \\
A_2 & 0 & 5 & 53 & 75 \\
A_4 & 0 & 3 & 75 & 78 \\
\end{bmatrix}
\]
2.2 Vertical Partitioning

• Resulting Partitioning:

<table>
<thead>
<tr>
<th>C_ID</th>
<th>CName</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Athens</td>
<td>Attica</td>
</tr>
<tr>
<td>C2</td>
<td>Sparta</td>
<td>Laconia</td>
</tr>
<tr>
<td>C3</td>
<td>Thebes</td>
<td>Boeotia</td>
</tr>
<tr>
<td>C4</td>
<td>Corinth</td>
<td>Arcadia</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C_ID</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Democracy</td>
</tr>
<tr>
<td>C2</td>
<td>Monarchy</td>
</tr>
<tr>
<td>C3</td>
<td>Oligarchy</td>
</tr>
<tr>
<td>C4</td>
<td>Democracy</td>
</tr>
</tbody>
</table>
After the partitioning has been decided, the fragments have to be allocated at different sites of a computer network

- Different allocation strategies
- Focus on high availability
- Performance gain vs. replication

History: file allocation problem in networks

- Wesley Chu: "Optimal File Allocation in Multiple Computer Systems", IEEE Transactions on Computers, 1969
2.3 Where is my data?

Sudarshan Kadambi, Jianjun Chen, Brian F. Cooper, David Lomax, Raghu Ramakrishnan, Adam Silberstein, Erwin Tam, Hector Garcia-Molina; VLDB 2011
2.3 PNUTS

Platform for Nimble Universal Table Storage: “a massively parallel and geographically distributed database system for Yahoo!’s web applications”

- Record level operation
- Indexes and views
- Novel Consistency model
- Structured, flexible schema
- Geographic replication
- Asynchronous operations

Flexible access: hashed or ordered, Indexes, views
Timeline consistency:

Timeline consistency enforced through record mastership. Record mastery ensures that her writes get applied in all regions in the order in which she made them.
2.3 Where is my data?

• Motivation and goals:
  – Legal constraints.
  – Minimum number of copies of critical data.
  – Inter-datacenter bandwidth can be extremely expensive: replication and forward bandwidth.
  – Latency constraints.
• Selective replication mechanism:
  – Finer grained: per-record selective replication policy.
  – Goal: minimize replication costs but:
    • Respecting policy constraints.
2.3 Selective replication

• Policy constraints:
  
  – **MIN_COPIES**: The minimum number of full replicas of the record that must exist.
  
  – **INCL_LIST**: An inclusion list, the locations where a full replica of the record must exist.
  
  – **EXCL_LIST**: An exclusion list, the locations where a full replica of the record cannot exist.
2.3 Selective replication

• Rule 1:

IF

    TABLE_NAME = "Users"

THEN

    SET 'MIN_COPIES' = 2
    CONSTRAINTPRI = 0
2.3 Selective replication

• Rule 2:

IF

TABLE_NAME = "Users" AND
FIELD STR('home location') = 'France'

THEN

SET 'MIN_COPIES' = 3 AND
SET 'EXCL LIST' = 'USWest, USEast'

CONSTRAINT PRI = 1
2.3 Where to place the replicas?

• Static constraint-based data placement
  – Define a function \texttt{choose_replicas}(R,C), based on the values of the fields

• Dynamic placement
  – Make full replica (read pattern) and
  – Demote a full replica (write pattern)
  – Key aspect: retention interval
• **Practice** shows…
  – Sophisticated algorithms are **rarely** needed in real life scenarios
  – In most cases **simple analytical models** are sufficient to support decisions

• **Major factors**
  – Fragmentation schema
  – User queries/updates and their frequencies
  – Network topology, bandwidth and latency
  – The sites’ storage and processing characteristics
• What is the best placement of fragments and/or best number of copies to:
  – minimize query response time
  – maximize throughput
  – minimize “some cost”: communication, storage, updates, at sites…

• Subject to constraints?
  – Available storage
  – Available bandwidth, power,…
  – Keep 90% of response time below \( X \)
  – …
2.3 Allocation Problem

- **Golden Rules**
  - Place data **as close as possible** to where it will be used
  - Use **load balancing** to find a global optimization of system performance
• Data has to be distributed across nodes
• Main concepts:
  – **Fragmentation**: partition all data into smaller fragments / “chunks”
    • How to fragment? How big should fragments be? What should fragments contain?
  – **Allocation**: where should fragments be stored?
    • Distribution and replication
    • Where to put which fragment? Should fragments be replicated? If yes, how often and where?
In general, fragmentation and allocation are an optimization problem which are closely depended on the actual application.

- Focus on high availability?
- Focus on high degree of distribution?
- Focus on low communication costs and locality?
- Minimize or maximize geographic diversity?
- How complex is the data?
- Which queries are used how often?

Many possibilities and decision!
Next Lecture: Distributed Querying

- Distributed Catalogs & Statistics
- Query and Result shipping
- Distributed Query Evaluation
- Distributed Optimization