2.0 Sharing Architectures

- 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 main memory?"

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
    - 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

2.0 Sharing Architectures

- 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 architecture again become popular 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
  — 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
• Also used for multi-core servers

2.0 Sharing Architectures

• 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 enterprise applications...
  — Nowadays, most notable application for shared-nothing architectures are web-age applications
    • e.g. Google, Yahoo, etc...

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>Inter-Node: Okay</td>
</tr>
<tr>
<td></td>
<td>Intra-Node: Very Good</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

<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>Inter-Node: Okay</td>
</tr>
<tr>
<td></td>
<td>Intra-Node: Very Good</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>
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 $F_R = \{R_1, R_2, R_3, \ldots, R_n\}$
- 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**

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

**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.:
      - Fragment $t$<br>city_population $\geq$ 100000<br>fragment $t$<city_population $< 100000$
  - **Value Lists**
    - Provide the attribute values for each partition
      - Fragment $t$<city_location $\in$ {Attica, Laconia}
        - Fragment $t$<city_location $\in$ {Boeotia, Arcadia}
  - **Hash Values**
    - Provide a hash function on tuples. Each fragment is responsible for a given hash range,
      - Fragment $0 \leq h(t) \leq 50$
      - Fragment $50 < h(t) \leq 100$

**2.1 Fragmentation**

- Several major issues arise:
  - **Disjointness Rule**
  - **Completeness Rule**
  - **Reconstruction Rule**

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes

**2.1 Fragmentation**

- Allocation to nodes
2.1 Fragmentation

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

- **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

### Vertical Partitioning

- **Vertical Completeness Rule**
  - Every attribute of $R$ shows up in one of the fragments
  - $R = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$

- **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. $R_i \bowtie R_j = R_i \bowtie R_j$

### Hybrid Partitioning

- Use Vertical and Horizontal Partitioning

### 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

### 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

### Query Processing

<table>
<thead>
<tr>
<th>Processing</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>Consistency 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
    • Minimize 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!

• 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

• 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

• 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
2.2 Horizontal Partitioning

- **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 during runtime

- **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 the cloud part of the lecture

- **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 = \( V \)
    * \( q_2: \) (grandiosity \( \geq V_1 \) \& grandiosity \( \leq V_2 \)) \equiv \neg \text{grandiosity} \geq V_1 \text{ \&} \neg \text{grandiosity} \leq V_2
  - minterms = \{ name = \( V \), \neg \text{grandiosity} \geq V_1 \text{ \&} \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

- **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

- Example: Partition cities
  - Use minterm-sets: \(\{\text{location}=\text{Boetia}\}, \{\text{Location}=\text{Peloponnesse}\}, \{\text{Location}=\text{Attica}\}\)
  - Query: \(\text{SELECT} \ast \text{FROM cities WHERE c_id}=V\)
    - Complete and minimal
      - Each fragment could be accessed
      - Within each fragment, probability for selecting a tuple is similar

2.2 Horizontal Partitioning

- 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...

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: Partition cities
  - Use minterm-sets: \(\{\text{location}=\text{Boetia}\}, \{\text{Location}=\text{Peloponnesse}\}, \{\text{Location}=\text{Attica}\}\)
  - Query: \(\text{SELECT} \ast \text{FROM cities WHERE c_id}=V\)
    - Not complete
      - \(C_2\) has higher probability (1.0) than \(C_4\) (0.0)

2.2 Horizontal Partitioning

- 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

- Example:
  - Let the relation \(\text{EntityTypes}\) be partitioned as
    \(\text{EntityTypes}_1 = \sigma_{\text{Type}=\text{God}} \text{EntityType}\)
    \(\text{EntityTypes}_2 = \sigma_{\text{Type}=\text{Monster}} \text{EntityType}\)
  - Entity is thus partitioned to

---

**Entity**

<table>
<thead>
<tr>
<th>E_ID</th>
<th>Name</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>Menoephe</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>

**Entity**

<table>
<thead>
<tr>
<th>E_ID</th>
<th>Type</th>
<th>Characteristic</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>
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
  - 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

2.2 Horizontal Partitioning

2.2 Horizontal Partitioning

2.2 Partitioning Techniques

Good: Simple Join Graph

Bad: Total Join Graph

Okay: Partitioned Join Graph

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

"Purple Rain"
### 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

- **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”
    \[ q_1 = \text{SELECT type FROM cities WHERE c_id = V} \]
  - “Find all cities in a given area”
    \[ q_2 = \text{SELECT name FROM cities WHERE location = V} \]
  - “How many democratic cities are in an area?”
    \[ q_3 = \text{SELECT count(type) FROM cities WHERE location = V and type = "Democracy"} \]
  - Keep in mind!
    - In most DBs, 20% of all queries produce 80% of all load ⇒ just using the most important queries is OK

\[ \begin{array}{c|cccc|c|c|c}
   & A_1 & A_2 & A_3 & A_4 & C_{id} & \text{Name} & \text{Type} \\
\hline
C_1 & Athens & Democracy & Attica & & \text{Thebes} & \text{Arcadia} & \text{Monarchy} & \text{Lacconia} \\
C_2 & Sparta & & & & \text{Thebes} & \text{Arcadia} & \text{Oligarchy} & \text{Boeotia} \\
C_3 & & & & & Corinth & Democracy & \text{Arcadia} & \\
C_4 & & & & & & & & \\
\end{array} \]

2.2 Vertical Partitioning

- Which attributes are used in which query?
  - Constructing the Attribute Usage Matrix (use)
    \[ \text{use} (q, A_j) = \begin{cases} 
      1 & \text{if } q \text{ uses } A_j \\
      0 & \text{otherwise} 
    \end{cases} \]
  - Resulting usage matrix:
    \[ \text{Affinity Matrix} = \begin{bmatrix} 
      1 & 0 & 1 & 0 \\
      0 & 1 & 1 & 0 \\
      0 & 1 & 0 & 1 \\
      0 & 0 & 1 & 1 
    \end{bmatrix} \]

2.2 Vertical Partitioning

- Attribute Affinity Matrix
  - Compute cell \( \text{aff} (A_i, A_j) \)
    - Count how often all queries are executed which use attribute \( A_i \) as well as attribute \( A_j \)
    - from attribute usage matrix
    \[ \text{aff} (A_i, A_j) = \sum_k \text{use}(q_k, A_i) \text{use}(q_k, A_j) \]
  - Example:
    - \( q_1 \) is the only query to access \( A_1 \) and \( A_3 \) at the same time, thus \( \text{aff} (A_1, A_3) = q_{\text{stat}}(q_1) = 45 \)
    - \( A_2 \) is accessed by \( q_2 \) and \( q_4 \), thus \( \text{aff} (A_2, A_3) = q_{\text{stat}}(q_2) + q_{\text{stat}}(q_4) = 80 \)

\[ \begin{array}{c|cccc|c}
   & A_1 & A_2 & A_3 & A_4 \\
\hline
A_1 & 45 & 0 & 45 & 0 \\
A_2 & 0 & 80 & 5 & 75 \\
A_3 & 45 & 5 & 53 & 3 \\
A_4 & 0 & 75 & 3 & 78 \\
\end{array} \]
2.2 Vertical Partitioning

• 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

• 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>

2.3 Data Allocation

• Some common practical assumptions
  – Generic packet-switched networks have a known data transmission rate (e.g., 100 mb/s)
  – Shortest distances between sites are picked and no protocol overhead is assumed
  – Ideal network propagation delay is the speed of light, but usually a lower speed of 200,000 km/s is assumed to allow for cable degradation
  – Network contention is ignored

• 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
2.3 Allocation Problem

- Find the **optimal distribution** of fragments on sites
  - Optimality can be defined with respect to minimal cost under performance constraints
- The **cost function** consists of:
  - the cost of storing each fragment at a site
  - the cost of querying/updating a fragment at a site (CPU and I/O)
  - the cost of data communication

2.3 A Note on Replication

- **Trade-Offs** in database distribution due to data replication

2.3 Allocation Strategies

- **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
- **Common techniques**
  - Non-redundant 'best fit' method
  - 'All beneficial sites' method
  - Progressive table allocation method

2.3 Best Fit Method

- The **non-redundant best fit method** determines a single site for most beneficial allocation
  - Benefit is interpreted as total query and update references
  - Fragment Ri is placed at site Sj, where the number of local query and update references is maximized
    - Group fragment accesses (query/update) by sites
    - Choose the site for allocation with maximum number
    - If there are several equivalent options, always choose the site that hosts the smallest number of fragments

2.3 Best Fit Method

- **Example**

<table>
<thead>
<tr>
<th>Fragment</th>
<th>Site</th>
<th># Accesses (r/w)</th>
<th>Typical Frequency</th>
<th>Total local references</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>S1</td>
<td>2</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>3</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>R2</td>
<td>S1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>3</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

- **Allocation decision**
  - Allocate fragment R1 to site S1
  - Allocate fragment R2 to site S1
  - Allocate fragment R1 to site S2 to gain robustness
2.3 Best Fit Method

- The **best fit method** is easy to compute
  - But lacks accuracy since references do not take I/O times, total block accesses, etc. into account
  - No data replication is performed
- How can we extend this method to consider real read/write and network costs and care for replication?

---

2.3 All Beneficial Sites

- The **all beneficial sites' methods** introduces a degree of **redundancy** for improved availability
  - Select all sites for allocation where the benefit is greater than the cost for one additional copy of a fragment
    - Exhaustively enumerate the total cost for initial allocations
    - Compute total cost after replication of some fragment
    - Decide for replication only if total costs are lower
  - Can also be used for a non-redundant allocation

---

2.3 Progressive Allocation

- The **progressive fragment allocation** method is a practical extension of the all beneficial sites method
  - The **first copy** is always allocated based on the maximum value of benefits minus costs
  - The **next allocation decision** is based on the location of the first copy and the maximum value of benefit minus costs for the remaining sites
  - Continue until no benefit exceeds costs for any site

---

Architectures

- There are 3 major architectures for DDBMS
  - **Share-Everything Architecture**
    - Nodes share main memory
    - Suitable for tightly coupled high performance highly parallel DDBMS
    - Weaknesses wrt. scalability and reliability
  - **Shared-Disk Architecture**
    - Nodes have access to same secondary storage (usually SAN)
    - Strengths wrt. complex data and transactions
    - Common in enterprise level DDBMS
  - **Share-Nothing Architecture**
    - Nodes share nothing and only communicate over network
    - Common for web-age DDBMS and the cloud
    - Strength wrt. scalability and elasticity
**Fragmentation**

- 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?

**Outlook**

- **Next Lecture:** Distributed Querying
  - Distributed Catalogs & Statistics
  - Query and Result shipping
  - Distributed Query Evaluation
  - Distributed Optimization

**Fragmentation**

- In general, fragmentation and allocation are **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!