Design of a Distributed Database
or: where to store the data

- Requirements and influences
- horizontal fragmentation
  - primary
  - derived
- Vertical fragmentation
- Allocation

Introduction

- How should data be distributed over locations?

**Assumptions**
- Peer-to-Peer Database servers
- DBS software supports flexible allocation and transparent query / update processing
- No replication so far
- Criterion for data placement: basically performance read / write ratio, communication cost, cost of storing at $S_i$

Based on section 5 of Özsu / Valduriez and H. Garcia-Molina, Stanford

Framework for Distribution

Access pattern behavior

dynamic

static

Data and function distribution?

Level of sharing

Level of knowledge

Does DB usage change over time?

Partial information

Complete information

What does the designer know about DB usage (static)?
- nothing: random distribution does not make sense
- Complete knowledge: unrealistic

Top Down vs Bottom up

- Top down design
  - Design database schema from scratch
  - Design schema fragmentation
  - Allocate fragments to servers

  Ideal case. Typically legacy data have to be incorporated

- Bottom up
  - Federated or Multi database
  - Design issue: which data visible globally?
    Defined by export schema
The ideal picture

Global Schema

Fragmentation schema

Allocation schema

local schema

local DB

Server S₁

... local schema

local DB

Server Sₙ

Related issues, treated separately

Clients see the global schema

Autonomy: local client

Clients see the global schema

Redundant Allocation
Redundant allocation = Replication

- Non-replicated partitioned:
  - each fragment resides at only one site

- Replicated
  - fully replicated: each fragment at each site?
    Distribute relations, not fragments
  - partially replicated: each fragment at some of the sites

- Rule of thumb:
  \[
  \text{If } \frac{\# \text{ read-only queries}}{\# \text{ update queries}} > 1
  \]
  replication is advantageous,
  otherwise replication may cause problems

Fragmentation types

- Horizontal fragmentation ("partitioning" in Oracle)
  - Partition of relation \( R \): disjoint cover of \( R \)
    i.e. disjoint subsets \( R_i \) of \( R \), \( R = \bigcup R_i \)
    math speaking: partition of \( R \)

- Vertical fragmentation
  - Split \( R \) attributes according to common access pattern
  - Issue: \( R \) must be reconstructed in a lossless way
  - Trivial if each fragment contains key ⇔ normalization theory

- Hybrid fragmentation
  - No relevance in practice
Horizontal fragmentation

- Common horizontal partitioning techniques
  - Round robin
  - Hash partitioning
  - Range partitioning

  Used primarily for parallel databases

Round robin partitioning

<table>
<thead>
<tr>
<th>R</th>
<th>S₀</th>
<th>S₁</th>
<th>S₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>→</td>
<td>t₁</td>
<td></td>
</tr>
<tr>
<td>t₂</td>
<td>→</td>
<td>t₂</td>
<td>t₃</td>
</tr>
<tr>
<td>t₃</td>
<td>→</td>
<td></td>
<td>t₃</td>
</tr>
<tr>
<td>t₄</td>
<td>→</td>
<td>t₄</td>
<td>t₅</td>
</tr>
<tr>
<td>...</td>
<td>→</td>
<td></td>
<td>t₅</td>
</tr>
</tbody>
</table>

- Evenly distributes data
- Good for scanning full relation
- Bad for point or range queries

- Better alternative today: use RAID disk organization
### Hash partitioning

<table>
<thead>
<tr>
<th>R</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>→</td>
<td>h(k1)=2</td>
<td>t1</td>
</tr>
<tr>
<td>t2</td>
<td>→</td>
<td>h(k2)=0</td>
<td>t2</td>
</tr>
<tr>
<td>t3</td>
<td>→</td>
<td>h(k3)=0</td>
<td>t3</td>
</tr>
<tr>
<td>t4</td>
<td>→</td>
<td>h(k4)=1</td>
<td>t4</td>
</tr>
</tbody>
</table>

- Good for point queries on hashed attribute
- Ditto for joins
  - see hash clusters in Oracle
- Not for range queries
- Not for point queries not on key
- If hash function good, even distribution

### Range partitioning

<table>
<thead>
<tr>
<th>R</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1: A=5</td>
<td>partitioning vector</td>
<td>t1</td>
<td></td>
</tr>
<tr>
<td>t2: A=8</td>
<td>4 7</td>
<td>t2</td>
<td></td>
</tr>
<tr>
<td>t3: A=2</td>
<td>V0 V1</td>
<td>t3</td>
<td></td>
</tr>
<tr>
<td>t4: A=3</td>
<td></td>
<td>t4</td>
<td></td>
</tr>
</tbody>
</table>

- Good for *some* range queries on A
- Need to select good vector: else unbalance
  - data skew
  - execution skew
- Option in Oracle 9i (for parallel DB)
Horizontal Partitioning: Example

Employee relation $E(\#, \text{name, loc, sal, \ldots})$

40% of queries:

Qa: select * from $E$
where loc=S1
and...

Qb: select * from $E$
where loc=S2
and...

Two sites: S1, S2
Distribution according to loc-attribute

Qa $\rightarrow$ S1 $\leftarrow$ S2 $\rightarrow$ Qb

Not a big deal, common sense is ok

---

Horizontal partitioning of more than one relation

- Given Relations
  employees $E$, schedule $S$, projects $P$

- Start fragmentation of one relation
  Primary partitioning (fragmentation)

- Fragmentation of other relations dependent:
  Derived partitioning (fragmentation)

Intuitive reason: If $E$ fragments will join with particular $S$ tuples ($E.\# = S.\#$), should be allocated to the same server
Primary horizontal partitioning

Partition defined by elementary (simple) predicates
p1, p2, ..., pn

Two separate issues:
  ▸ How to check
    ▪ cover property ("completeness")
    ▪ disjointness ("non redundant")
    of fragments?
  ▸ Which simple predicates should be taken for partitioning?
    ▪ Common sense: those occurring often in queries
      e.g. location (loc) in example

Horizontal partitioning

▸ Elementary (simple) predicates
  Given relation R(a1,...,an).
  Pj : ai = val
  is called simple predicate
  Returns a boolean value for each t ∈ R

Let PR = {Pj | Pj used for partitioning}

Example: PR = {loc =’Munic’, loc = ’New York’, loc = ’Berlin’,
              sal <= 100000}

This PR does not cover R:
(4711, Meier, Wien, 200000,...) not in any segment
Horizontal partitioning using minterms

- Min terms $M$ of $PR$:
  
  $$M = \big\{ \bigwedge \Pi^*, \Pi \in PR\big\}$$
  
  $\Pi^* = \Pi$ or $\neg \Pi$

- Convention for negation of predicates using comparison:
  
  $\Pi : P < val$ then $\neg \Pi : P \geq val$

- Obvious: $M$ defines a (complete, disjoint) partition of $R$

<table>
<thead>
<tr>
<th>Sal &gt; 200000</th>
<th>Sal \leq 200000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loc = 'B' &amp; sal &gt; 200000</td>
<td>Loc = 'B' &amp; sal \leq 200000</td>
</tr>
<tr>
<td>Loc &lt;&gt; 'B' &amp; sal &gt; 200000</td>
<td>Loc &lt;&gt; 'B' &amp; sal \leq 200000</td>
</tr>
</tbody>
</table>

---

Horizontal partitioning using minterms

- Partitioning
  
  Input: Set $PR$ of application dependend simple predicates
  
  - Calculate set $M$ of minterms
  - Eliminate "impossible" (contradictory) minterms
    
    e.g. $loc = 'Munic' \land loc = 'Berlin'$
  
  - Resulting $PT(E) \subseteq M$ defines the horizontal fragmentation

- How to define $PR$?
  
  - Use knowledge about application
  - Use at least one predicate set which covers relation
    
    $loc = 'A'$, $loc = 'B'$, $loc = 'C'$ are not necessarily exhaustive
  
  - Use predicates occurring frequently in queries
  - Access probability of $P$ and $\neg P$ should be about the same (?)
  
  - Use predicates reflecting "natural distribution" of data ("loc=..")
**Derived horizontal fragmentation (partitioning)**

| E(#, name, loc, sal...) | 1:n | S(#, p#, from, to) | m:n | P(p#, ..) |

Should $S$ be partitioned in the same way?

E.g. $S.# < 5000$

No: horrible performance, if join between $E$ and $S$ is a frequent operation

Goal: make joins as local as possible

**Derived horizontal fragmentation using semi-join**

$S \bowtie E = \{ s \mid s \in S, \exists e \in E: s.a = e.a, a \text{ join attribute} \}$

Those tuples of $S$ are qualified, which have a join partner in $E$

- Given partition of $E$ $\text{PT}(E) = \{E_1, \ldots, E_k\}$
  
  $\text{PT}(S) = \{ S_i \mid S_i = S \bowtie E_i \}$

- $E$ owner relation
- $S$ Member relation

Example

$E(#) , name, loc, sal...$ $1:n$ $S(#, p#, from, to) m:n P(p#, ..)$

$S$ is partitioned into $\{S_i\}$ such that for each $S_i$ there is an $E_j$ from $E$'s partition with the same values in attribute $#$. 
Derived partition

Not true in general!

a) Suppose lack of referential integrity between E and S i.e. more s.# values than e.# values
   ⇒ no fragment for tuples with "dangling pointers (#-values)"

b) S.# is not key in E: tuples of S might be in different fragments Si
   ⇒ technique is useful only if join attribute is foreign key in S

This is the important case.

Summary: horizontal fragmentation

Type: primary minterm based, derived using semi-join

Properties
  Fragmentation: completeness, disjointness i.e. a partition in the math sense

Predicates:
  use predicates occurring frequently in applications
### Vertical fragmentation

**R with attributes A**

<table>
<thead>
<tr>
<th>A1</th>
<th>A2</th>
<th>( A_i )</th>
<th>A3</th>
<th>A4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
R[A] \Rightarrow R[A_1] \quad ... \quad R[A_n]
\]

with \( A_i \subseteq A \)

**Desirable Properties**

**Completeness:**
\[
\bigcup A_i = A
\]

**Disjointness:**
\[
A_i \cap A_j = \emptyset, \quad i \neq j
\]

**Bad:** \( R \) cannot be reconstructed

**Lossless join:**
\[
\begin{align*}
\forall i \quad R_i &= R \\
&\text{Guaranteed if primary key is in every } A_i
\end{align*}
\]

### Grouping of attributes

**How do we decide what attributes are grouped with which?**

**Example:**
\[
E(\#, NM, LOC, SAL) \quad E_1(\#, NM) \\
E_2(\#, SAL) \quad E_3(\#, SAL)
\]

**Desirable Properties**

- Completeness: \( \bigcup A_i = A \)
- Disjointness: \( A_i \cap A_j = \emptyset, \quad i \neq j \)
- Bad: \( R \) cannot be reconstructed
- Lossless join: \( \forall i \quad R_i = R \)
Grouping of attributes

**Attribute affinity matrix:** Aij measure of co-occurrence of different attributes in the same application

<table>
<thead>
<tr>
<th></th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A2</td>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A3</td>
<td>45</td>
<td>48</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A4</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A5</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>75</td>
<td>-</td>
</tr>
</tbody>
</table>

Use appropriate cluster algorithm

\[ R_1[K,A_1,A_2,A_3] \quad \quad R_2[K,A_4,A_5] \]

Hybrid fragmentation

- Makes sense - in principle
- Neither theoretically solved nor urgently required in practice...

Horizontal partitioning of vertical fragments, some or all
### Allocation

- The problem:

  Given a partitioning of $R = R_1, R_2, \ldots, R_n$, and locations (DB servers) $L_1, \ldots, L_k$:

  Find a cost optimal assignment of $R_i$ to $L_j$, all $i$, whatever "cost optimal" means

- 0/1 optimization problem:

  Find solution of:

  $$\sum c_{ij} \times x_{ij} = \text{min}, \quad x_{ij} = \begin{cases} 1 & \text{if } R_i \text{ stored at } L_j \\ 0 & \text{otherwise} \end{cases}$$

  $c_{ij}$ = cost for assigning $R_i$ at $L_j$

### Intractable?

- Solving the optimization problem is simple, e.g. 10 locations, 100 fragments \(\Rightarrow\) 1000 variables

  \(\Rightarrow\) complexity lies in the estimation of cost

- Issues (some...)

  - What is communication cost?
    - depending on size of result set, relations, ...
  - What is storage capacity, cost at sites?
    - and size of fragments?
  - What is processing power at sites?
  - How to "normalize" cost types
    - (communication, storage, ... )
Allocation as optimization

More issues:
- query processing strategy?
  - How are joins done?
  - Where does query originate?
  - Where are results collected?

Replicates?
- write overhead for replicated data?
- concurrency control?

No replicates ⇔ additional constraints on optimization problem

Optimization of allocation

Further constraint:
- bandwidth, CPU power
- available storage
...

Optimization problem: what to optimize??
- cost? How to define it?
- response time
- throughput
"Simple" approach

- Scenario: read cost $r_{ij}$ for replication of fragment $F$

  \[ r_{i,1} \quad r_{i,2} \quad r_{i,3} \]

  \[ r = \infty \]

  - Read cost: $\sum (t_i \times \text{MIN } r_j)$

    accessing $F$ at $S_j$ from $S_i$, minimal read effort

Simple approach: write

- Write cost – update all replicates of fragment $F$

  \[ \sum \sum X_j u_{ij} w_{ij} \]

  $i$: Originating site of request

  $j$: Site being updated

  $X_j$: 0 if $F$ not stored at $S_j$

  1 if $F$ stored at $S_j$

  $u_{ij}$: Write traffic at $S_i$: how many writes at $S_i$

  $w_{ij}$: Write cost per write:

    Updating fragment $F$ at $S_j$ from $S_i$
Simple approach: storage cost

Storage cost

\[ \sum X_i d_i \]

- \( X_i \): 0 if F not stored at \( S_i \)
- 1 if F stored at \( S_i \)
- \( d_i \): storage cost at \( S_i \)

Target function for allocation of one fragment F:

Minimize

\[ \sum [t_i \times \text{MIN } r_{ij} + \sum x_j \times u_i \times w_{ij} + \sum x_i \times d_i] \]

Summary allocation

- Finding optimal allocation impossible
  … since not all influences can be quantified
   (join algorithm, distribution of values, …)
- Simple allocation model gives some hints
- More important for parallel DB than for distributed

Why?
Summary

- Distribution schema:
  - Fragmentation versus Allocation
- Horizontal partitioning
- Vertical partitioning
- How to find "good" partitionings?
- How to find "good" (not to say optimal ;) allocations

Point for discussion:
does fragmentation / allocation make sense in a parallel environment?
- Think of flexible disk architectures (e.g. RAID)
- Provide the same I/O parallelism as fragmented allocation?