MULTENANT-DATABASES

applications
on demand
software as a service

SaaS

Kaushik Mukherjee
23rd Nov 2010
Outline

• Motivation
  – Software as a Service
  – Need for Multi-tenancy
  – Need for Multi-tenant Databases

• Multi-tenant Databases
  – Challenges
  – Design Approaches and tradeoff
  – Experiment

• Discussion

• Conclusion
Today's Software Service Industry

- **On Premise Software**

- **Software as a Service**
Multi-tenancy
Why Multi-tenant databases?

- Consolidating multiple databases onto same operational system
- Reduces Total Cost of Ownership
Challenges in Multi-tenant Database

- **Scalability**
  - Tradeoff between cost handling many tables and cost query rewriting

- **Allow Schema Extensibility**
  - Multiple tenant share tables
  - Need for tenant specific schema extensibility
Design Approaches

- **Private Table**
  - Natural Thing to do - each tenant gets a private schema
  - Low cost on query transformation
  - Less consolidation

- **Extension Table**
  - Split off extensions into separate tables
  - Higher cost on Query transformation
  - Slightly better consolidation
Universal Table

- **Generic Structure with VARCHAR value columns**
  - $n$-th Column of the logical table is mapped to Col-$n$ in the universal table
  - Extensibility

- **Disadvantages**
  - Many *Null* Values
  - Not type safe
  - No Indexing
Pivot Table

<table>
<thead>
<tr>
<th>Account</th>
<th>Name</th>
<th>Hospital</th>
<th>Beds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Acme</td>
<td>St. Mary</td>
<td>135</td>
</tr>
<tr>
<td>2</td>
<td>Gump</td>
<td>State</td>
<td>1042</td>
</tr>
</tbody>
</table>

**Pivot Table Example**: (Two tables shown)

<table>
<thead>
<tr>
<th>Tenant</th>
<th>Table</th>
<th>Col</th>
<th>Row</th>
<th>Str</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Acme</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>St. Mary</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Gump</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>State</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Ball</td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>Big</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tenant</th>
<th>Table</th>
<th>Col</th>
<th>Row</th>
<th>Int</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>135</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1042</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>65</td>
</tr>
</tbody>
</table>
Chunk Table

- **Generic Structure**
  - Suitable if data-set can be partitioned into dense subsets
  - Derived from Pivot table

- **Performance**
  - Fewer joins for reconstruction if densely populated subsets can be extracted
  - Reduced meta-data/data ratio dependent on the chunk size
  - Indexable
Row Fragmentation

- Combine different schema mappings for getting best fit
  - Mixes Extension and Chunk Tables
  - Each fragment can be stored in an optimal schema layout

- Optimal row fragmentation depends on
  - Workload
  - Data distribution
  - Data popularity
Query Transformation

- Reconstructing original tav requires many equi-joins

**Source Query**

```
SELECT Beds
FROM Account17
WHERE Hospital = 'State'
```

- Collect table and column names
  - Account17 : Beds, Hospital

- Obtain chunk tables and meta-data
  - Chunk (int|str)
  - Account17 :
    - Table = 0, Tenant =17
  - Beds, Hospital :
    - Chunk =1
Query Transformation

• **Generate filter query**

```sql
SELECT Str1 as Hospital , Int1 as Beds
FROM Chunk (int|str)
WHERE Tenant = 17 AND Table = 0 AND Chunk = 1
```

• **Replace reference in source query**

```sql
SELECT Beds FROM
(SELECT Str1 as Hospital Int1 as Beds
FROM Chunk (int|str) WHERE Tenant = 17
AND Table =0 AND Chunk =1) As Account17
WHERE Hospital = 'State'
```
Query Transformation

● Structural Changes
  – Additional Nesting
  – Joins
  – Base Table Access

● Impact on Performance
  – Nesting can be flattened by query optimizer
  – Joins are cheaper only if the cost of loading the chunks and applying index supported join are cheaper than loading wider conventional relation
  – Meta data columns in base tables have indexing support
Query Evaluation Experiment

- **Goal**
  - Show if the query transformation can handle issues of scalability
  - Evaluate impact of Join overhead
  - Evaluate impact of meta-data overhead

- **Test Query**

  ```sql
  SELECT p.id, ...
  FROM parent p, child c
  WHERE p.id = c.parent
  AND p.id = ?.
  ```
Query Evaluation Experiments

- **Conventional Schema**
  
  Parent
  
  id col1 col2 ... col90
  
  Child
  
  id Parent col1 col2...col90

- **Chunk Schema**
  
  **ChunkData**
  
  table chunk row int1 int2 int2 date date2
  str1 str2

  **ChunkIndex**
  
  table chunk row int1
Join Overhead Costs

Chunk Table width: ▲ 3
(# columns)
○ 6
▼ 15
▲ 30
● 90

conventional table: □

Q2 scale factor (((# of data columns)/2 in Q2’s SELECT clause)
Discussion

• **Strengths**
  - Chunk tables is a good design for trade-off of extensibility and meta data usage.
  - Chunk tables gives response time improvement over vertical partitioning

• **ShortComings/Future work**
  - No Algorithms to design chunk tables
  - Identifying the chunks is heuristic
  - No comparative experiment done with the other schema mapping techniques proposed in the paper
Conclusion

- Is chunk tables a good approach for designing multi-tenant databases?
- How practical it is for real life systems?
- How do companies like Salesforce.com handle it?
THANK YOU

QUESTIONS ?