Introduction to Data Management

Lecture #22
(Physical DB Design II and NoSQL Background)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

- Homework info:
  - HW #7: Due on Friday (at 5 PM).
  - Then one final HW left (“NoSQL”).
    - Try AsterixDB installation before Friday if you want!
- The plan:
  - Today: Physical DB design wrap-up.
  - Indexing quiz in discussions this week.
  - Then: NoSQL & Big Data (a la AsterixDB)
    - Not in book: See paper linked to wiki syllabus!
### Some Index-Only Plan Examples

- **SELECT** `sal, age` FROM `Emp` WHERE `age > 11` AND `age <= 13`;
- `(SELECT * FROM Emp)`
- **SELECT** `age, COUNT(*)` FROM `Emp` GROUP BY `age`;
- **SELECT** `DISTINCT sal` FROM `Emp`;
- **SELECT** `sal, AVG(age)` FROM `Emp` GROUP BY `sal`;
- **Note:** The index files are each much smaller than the main file!

### Decomposition of a BCNF Relation

- Suppose that we choose \{ SDP, CSJDQV \}. This is in BCNF, so there is no “reason” to decompose further.
- However, suppose that two queries are very frequent:
  - *Find the contracts C held by suppliers S.*
  - *Find the contracts C that departments D are involved in.*
- Decomposing CSJDQV further into CS, CD and CJQV could speed up these queries. (“Vertical partitioning.”)
- On the other hand, the query below would now be slower (as it would need to do a join: CS ⨝ CJQV):
  - *Find the total value V of all contracts C held by supplier S.*
A **Vertical** Partitioning Example

<table>
<thead>
<tr>
<th>eno</th>
<th>email</th>
<th>name</th>
<th>addr</th>
<th>salary</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="mailto:joe@aol.com">joe@aol.com</a></td>
<td>Joe</td>
<td>1 Main St.</td>
<td>100000</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td><a href="mailto:sue@gmail.com">sue@gmail.com</a></td>
<td>Sue</td>
<td>10 State St.</td>
<td>125000</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td><a href="mailto:zack@fb.com">zack@fb.com</a></td>
<td>Zack</td>
<td>100 Wall St.</td>
<td>2500000</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>

(Vertical partitioning: $\times$)

<table>
<thead>
<tr>
<th>eno</th>
<th>email</th>
<th>name</th>
<th>addr</th>
<th>eno</th>
<th>salary</th>
<th>age</th>
<th>eno</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="mailto:joe@aol.com">joe@aol.com</a></td>
<td>Joe</td>
<td>1 Main St.</td>
<td>1</td>
<td>100000</td>
<td>25</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td><a href="mailto:sue@gmail.com">sue@gmail.com</a></td>
<td>Sue</td>
<td>10 State St.</td>
<td>2</td>
<td>125000</td>
<td>28</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td><a href="mailto:zack@fb.com">zack@fb.com</a></td>
<td>Zack</td>
<td>100 Wall St.</td>
<td>3</td>
<td>2500000</td>
<td>40</td>
<td>3</td>
<td>30</td>
</tr>
</tbody>
</table>

**Horizontal Decompositions**

- Prior def’n. of decomposition: Relation replaced by a set of joinable relations that are *projections*. (This is the most important and most common case.)
- Occasionally, we may want to instead replace a relation by a set of relations that are *selections*.
  - Each new relation has same schema as the original, but only a subset of the rows.
  - Collectively, the new relations contain all rows of the original. (Typically, the new relations are *disjoint*.)
  - Original relation is the UNION (ALL) of the new ones (i.e., rather than the JOIN of the new ones).
**Horizontal Decompositions (Cont’d.)**

- Suppose contracts with values over 10000 are subject to different rules. (This means queries on Contracts will frequently contain the condition $val > 10000$.)
- One approach to deal with this would be a clustered B+ tree index on the $val$ field of Contracts.
- Another approach is to replace Contracts by two new relations, LargeContracts & SmallContracts, with the same attributes (CSJDPQV).
  - Performs like index on such queries, but no index overhead.
  - Can build clustered indexes on other attributes, in addition!

**Masking Schema Changes**

```sql
CREATE VIEW Contracts(cid, sid, jid, did, pid, qty, val) AS SELECT * 
FROM LargeContracts 
UNION ALL 
SELECT * 
FROM SmallContracts
```

- The replacement of Contracts by LargeContracts and SmallContracts can be masked by this view.
- However, queries with $val > 10000$ must be run on LargeContracts* for fast execution; users concerned with performance must be aware of this change.
  (*The DBMS isn’t aware of the two tables’ value constraints.*)
A **Horizontal Partitioning Example**

<table>
<thead>
<tr>
<th>eno</th>
<th>email</th>
<th>name</th>
<th>addr</th>
<th>salary</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="mailto:joe@aol.com">joe@aol.com</a></td>
<td>Joe</td>
<td>1 Main St.</td>
<td>100000</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td><a href="mailto:sue@gmail.com">sue@gmail.com</a></td>
<td>Sue</td>
<td>10 State St.</td>
<td>125000</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td><a href="mailto:zack@fb.com">zack@fb.com</a></td>
<td>Zack</td>
<td>100 Wall St.</td>
<td>250000</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>

(Horizontal partitioning: \( \cup \))

<table>
<thead>
<tr>
<th>eno</th>
<th>email</th>
<th>name</th>
<th>addr</th>
<th>salary</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="mailto:joe@aol.com">joe@aol.com</a></td>
<td>Joe</td>
<td>1 Main St.</td>
<td>100000</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td><a href="mailto:sue@gmail.com">sue@gmail.com</a></td>
<td>Sue</td>
<td>10 State St.</td>
<td>125000</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td><a href="mailto:zack@fb.com">zack@fb.com</a></td>
<td>Zack</td>
<td>100 Wall St.</td>
<td>250000</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>

**Tuning Your Queries (and Views)**

- If a query runs slower than expected, see if an index needs to be re-built, or if *table statistics* are too old.
- Sometimes, the DBMS may not be executing the plan you had in mind. Common areas of weakness:
  - Selections involving *null values*.
  - Selections involving *arithmetic or string expressions*.
  - Selections involving *OR* conditions.
  - Lack of more advanced evaluation features like index-only strategies or certain join methods, or poor size estimation.
- Check the query plan! Then adjust the choice of indexes or maybe *rewrite the query/view*. 
**Miscellany for Query Tuning**

- Minimize the use of DISTINCT: don’t need/say it if duplicates are acceptable or answer contains a key.

- Consider the DBMS’s use of indexes when writing arithmetic expressions: \( E\text{.age} = 2 \times D\text{.age} \) will benefit from an index on \( E\text{.age} \), but it might not benefit from an index on \( D\text{.age} \)!

**Physical DB Design Summary**

- DB design consists of several tasks: requirements analysis, conceptual design, schema refinement, physical design and finally tuning.
  - In general, will go back and forth between tasks to refine a DB design; decisions in one task can influence choices in another task.

- Understanding the workload for the application, and the performance goals, is essential to a good design.
  - What are the important queries and updates? What attributes/relations are involved?
The conceptual schema should perhaps be refined by considering performance criteria and workload:
  • May choose 3NF or a lower normal form over BCNF.
  • May choose among several alternative decompositions based on the expected workload.
  • May actually denormalize, or undo, some decompositions.
  • May consider further vertical or horizontal decompositions.
  • Importance of dependency-preservation depends on the dependency to be preserved and the cost of the IC check.
    • Can add a relation for dep-preservation (e.g., the CJP example); or else, can check dependency using a join (e.g., using a trigger).

Over time, the indexes may have to be fine-tuned (dropped, created, re-built, ...) for performance.
  • Should examine the query plan(s) used by the system and adjust the choice of indexes appropriately.

Sometimes the system may still not find a good plan:
  • Null values, arithmetic conditions, string expressions, the use of ORs, etc., can “confuse” some query optimizers.

So, may have to rewrite a particular query or view:
  • Might need to re-examine your complex nested queries, complex conditions, or operations like DISTINCT.

Any lingering questions...?