Although the whole of this life were said to be nothing but a dream and the physical world nothing but a phantasm, I should call this dream or phantasm real enough, if, using reason well, we were never deceived by it.

Baron Gottfried Wilhelm von Leibniz

Review - Normal Forms

- Redundancy can cause problems
  - Insert, Update, Delete anomalies
  - Functional Dependencies indicate possible redundancy
  - Decomposition can remove redundancy

- Given FDs, can determine form of schema
  - BCNF: no redundancy
  - 3NF: some redundancy possible

Review: Normal Forms

- Decomposition
  - lossless-join mandatory
  - for each FD in relation R X \rightarrow Y, if X \rightarrow Y is empty, (R - Y), XY) is lossless
  - dependency preserving decomposition is nice
  - can always decompose to BCNF, but may not preserve dependencies
  - can always decompose to 3NF and preserve dependencies

Introduction

- After ER design, schema refinement, and the definition of views, we have the conceptual and external schemas for our database.
- The next step is to choose indexes, make clustering decisions, and to refine the conceptual and external schemas (if necessary) to meet performance goals.
- We must begin by understanding the workload:
  - The most important queries and how often they arise.
  - The most important updates and how often they arise.
  - The desired performance for these queries and updates.

Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions?
  - How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions?
  - How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Creating an ISUD Chart

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Frequency</th>
<th>Table</th>
<th>Employee Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Payroll Run</td>
<td>monthly</td>
<td>100%</td>
<td>S</td>
</tr>
<tr>
<td>Add Emps</td>
<td>daily</td>
<td>0.1%</td>
<td>I</td>
</tr>
<tr>
<td>Delete Emps</td>
<td>daily</td>
<td>0.1%</td>
<td>D</td>
</tr>
<tr>
<td>Give Raises</td>
<td>monthly</td>
<td>10%</td>
<td>U</td>
</tr>
</tbody>
</table>
### Decisions to Make

- **What indexes should we create?**
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- **For each index, what kind of an index should it be?**
  - Clustered? Hash/tree? Dynamic/static? Dense/sparse?
- **Should we make changes to the conceptual schema?**
  - Consider alternative normalized schemas? (Remember, there are many choices in decomposing into BCNF, etc.)
  - Should we `undo` some decomposition steps and settle for a lower normal form? *(Denormalization.)*
  - Horizontal partitioning, replication, views ...

### Tuning the Conceptual Schema

- **Choice of conceptual schema should be guided by workload, in addition to redundancy issues:**
  - We may settle for a 3NF schema rather than BCNF.
  - Workload may influence choice we make in decomposing a relation into 3NF or BCNF.
  - We may further decompose a BCNF schema!
  - We might *denormalize* (i.e., undo a decomposition step), or we might add fields to a relation.
  - We might consider *horizontal decompositions.*
- **If such changes are made after a database in use, called *schema evolution*; might mask changes by defining *views.***

### Example Schemas

- **Contracts (Cid, Sid, Jid, Did, Pid, Qty, Val)**
- Depts (Did, Budget, Report)
- Suppliers (Sid, Address)
- Parts (Pid, Cost)
- Projects (Jid, Mgr)

- **We will concentrate on Contracts, denoted as CSJDPQV.** The following ICS are given to hold:
  - JP ⊇ C, SD ⊇ P, C is the primary key.
    - What are the candidate keys for CSJDPQV?
    - What normal form is this relation schema in?

### Settling for 3NF vs BCNF

- **CSJDPQV** can be decomposed into **SDP and CSJDQV,** and both relations are in **BCNF.** *(Which FD suggests that we do this?)*
  - Lossless decomposition, but not dependency-preserving.
  - Adding CJP makes it dependency-preserving as well.
- **Suppose that this query is very important:**
  - Find the number of copies Q of part P ordered in contract C.
  - Requires a join on the decomposed schema, but can be answered by a scan of the original relation CSJDPQV.
  - Could lead us to settle for the 3NF schema CSJDPQV.

### Denormalization

- **Suppose that the following query is important:**
  - *Is the value of a contract less than the budget of the department?*
- **To speed up this query, we might add a field budget B to Contracts.**
  - This introduces the FD D ⊇ B wrt Contracts.
  - Thus, Contracts is no longer in 3NF.
- **Might choose to modify Contracts thus if the query is sufficiently important, and we cannot obtain adequate performance otherwise (i.e., by adding indexes or by choosing an alternative 3NF schema.)*

### Horizontal Decompositions

- **Def. of decomposition: Relation is replaced by collection of relations that are projections.** Most important case.
- **Sometimes, might want to replace relation by a collection of relations that are selections.**
  - Each new relation has same schema as original, but subset of rows.
  - Collectively, new relations contain all rows of the original.
  - Typically, the new relations are disjoint.
Horizontal Decompositions (Contd.)

- Suppose that contracts with value > 10000 are subject to different rules.
  - So queries on Contracts will often say WHERE val>10000.
- One approach: clustered B+ tree index on the val field.
- Second approach: replace contracts by two new relations, LargeContracts and SmallContracts, with the same attributes (CSJDQVP).
  - Performs like index on such queries, but no index overhead.
  - Can build clustered indexes on other attributes, in addition!

Masking Conceptual Schema Changes

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

- **Horizontal Decomposition** from above
- Masked by a view.
  - NOTE: queries with condition val>10000 must be asked wrt LargeContracts for efficiency: so some users may have to be aware of change.
  - I.e. the users who were having performance problems
  - Arguably that’s OK – they wanted a solution!

Now, About Indexes

- One approach:
  - Consider most important queries in turn.
  - Consider best plan using the current indexes, and see if better plan is possible with an additional index.
  - If so, create it.
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: indexes can make queries go faster, updates slower. Require disk space, too.

Issues to Consider in Index Selection

- Attributes mentioned in a WHERE clause are candidates for index search keys.
  - Exact match condition suggests hash index.
    - Note: rather inflexible, supported by few systems in practice.
    - Hence B+-trees might still be a good choice.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries, although it can help on equality queries as well in the presence of duplicates.
  - Try to choose indexes that benefit as many queries as possible.
  - **NOTE:** only one index can be clustered per relation!
    - So choose it based on important queries that benefit the most from clustering!!

Issues in Index Selection (Contd.)

- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - If range selections are involved, order of attributes should be carefully chosen to match the range ordering.
  - Such indexes can sometimes enable index-only strategies for important queries.
    - For index-only strategies, clustering is not important!
- When considering a join condition:
  - Hash index on inner is very good for Index Nested Loops.
    - Should be clustered if join column is not key for inner, and inner tuples need to be retrieved.
  - Clumped B+ tree on join column(s) good for Sort-Merge.

Example 1

```sql
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE E.dno=D.dno AND D.dname='Toy'
```

- Hash index on D.dname supports 'Toy' selection.
  - Given this, index on D.dno is not needed.
- Hash index on E.dno allows us to get matching (inner) Emp tuples for each selected (outer) Dept tuple.
- What if WHERE included: ```'... AND E.age=25'``` ?
  - Could retrieve Emp tuples using index on E.age, then join with Dept tuples satisfying dname selection. Comparable to strategy that used E.dno index.
  - So, if E.age index is already created, this query provides much less motivation for adding an E.dno index.
Clustering and Joins

- All selections are on Emp so it should be the outer relation in any Index NL join.
  - Suggests that we build a hash index on D.dno.
- What index should we build on Emp?
  - B+ tree on E.sal could be used, OR an index on E.hobby could be used. Only one of these is needed, and which is better depends upon the selectivity of the conditions.
  - As a rule of thumb, equality selections more selective than range selections.
- As both examples indicate, our choice of indexes is guided by the plan(s) that we expect an optimizer to consider for a query. Have to understand optimizers!

Index-Only Plans

- A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.
  - SELECT D.mgr
    FROM Dept D, Emp E
    WHERE D.dno=E.dno
  - SELECT D.mgr, E.ename
    FROM Dept D, Emp E
    WHERE D.dno=E.dno
  - SELECT E.dno, COUNT(*)
    FROM Emp E
    GROUP BY E.dno
  - SELECT E.dno, MIN(E.sal)
    FROM Emp E
    GROUP BY E.dno
  - SELECT AVG(E.sal)
    FROM Emp E
    WHERE E.age >= 25 AND E.sal BETWEEN 3000 AND 5000

Examples of Clustering

- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - Is the index clustered?
- Consider the GROUP BY query.
  - If many tuples have E.age > 10, using E.age index and sorting the retrieved tuples may be costly.
  - Clustered E.dno index may be better!
- Equality queries and duplicates:
  - Clustering on E.hobby helps!

Multi-Attribute Index Keys

- To retrieve Emp records with age=30 AND sal=4000, an index on <age,sal> would be better than an index on age or an index on sal.
  - Such indexes also called composite or concatenated indexes.
  - Choice of index key orthogonal to clustering etc.
- If condition is: 20<age<30 AND 3000<sal<5000:
  - Clustered tree index on <age,sal> or <sal,age> is best.
- If condition is: age=30 AND 3000<sal<5000:
  - Clustered <age,sal> index much better than <sal,age> index!
- Composite indexes are larger, updated more often.

Points to Remember

- Database design consists of several tasks: requirements analysis, conceptual design, schema refinement, physical design and tuning.
  - In general, have to go back and forth between these tasks to refine a database design, and decisions in one task can influence the choices in another task.
- Understanding the nature of the workload for the application, and the performance goals, is essential to developing a good design.
  - What are the important queries and updates? What attributes/relations are involved?
Points to Remember

- Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision; only one index on a given relation can be clustered.
  - Order of fields in composite index key can be important.
- Static indexes may have to be periodically re-built.
- Statistics have to be periodically updated.

Index Tuning “Wizards”

- Both IBM’s DB2 and MS SQL Server have automated index advisors
  - Some info in Section 20.6 of the book
- Basic idea:
  - They take a workload of queries
  - Possibly based on logging what's been going on
  - They use the optimizer cost metrics to estimate the cost of the workload over different choices of sets of indexes
  - Enormous # of different choices of sets of indexes:
    - Heuristics to help this go faster

Tuning Queries and Views

- If a query runs slower than expected, check if an index needs to be re-built, or if 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 (bad selectivity estimates)
  - Selections involving arithmetic or string expressions (ditto)
  - Selections involving OR conditions (ditto)
  - Complex, correlated subqueries
  - Lack of evaluation features like index-only strategies or certain join methods or poor size estimation.
- Check the plan that is being used! Then adjust the choice of indexes or rewrite the query/view.
  - E.g. check via POSTGRES "Explain" command
  - Some systems rewrite for you under the covers (e.g. DB2)
    - Can be confusing and/or helpful!

More Guidelines for Query Tuning

- Minimize the use of DISTINCT: don’t need it if duplicates are acceptable, or if answer contains a key.
- Minimize the use of GROUP BY and HAVING:

  ```
  SELECT MIN (E.age) FROM Employee E 
  GROUP BY E.dno 
  HAVING E.dno=102
  ```

  - Consider DBMS use of index when writing arithmetic expressions: \( E.age=2*D.age \) will benefit from index on \( E.age \), but might not benefit from index on \( D.age! \)

Guidelines for Query Tuning (Contd.)

- Avoid using intermediate relations:

  ```
  SELECT E.dno, AVG(E.sal) FROM Emp E, Dept D WHERE E.dno=D.dno 
  AND D.mgname='Joe'
  GROUP BY E.dno 
  ```

  ```
  SELECT T.dno, AVG(T.sal) FROM Temp T 
  GROUP BY T.dno
  ```

  - Does not materialize the intermediate rel Temp.
  - If there is a dense B+ tree index on \(<dno, sal>\), an index-only plan can be used to avoid retrieving Emp tuples in the second query!

Summary of Database Tuning

- The conceptual schema should be refined by considering performance criteria and workload:
  - May choose 3NF or lower normal form over BCNF.
  - May choose among alternative decompositions into BCNF (or 3NF) based upon the workload.
  - May denormalize, or undo some decompositions.
  - May decompose a BCNF relation further!
  - May choose a horizontal decomposition of a relation.
  - Importance of dependency-preservation based upon the dependency to be preserved, and the cost of the IC check.
    - Can add a relation to ensure dep-preservation (for 3NF, not BCNF); or else, can check dependency using a join.
Summary (Contd.)

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

• System may still not find a good plan:
  – Only left-deep plans considered!
  – Null values, arithmetic conditions, string expressions, the use of ORs, etc. can confuse an optimizer.

• So, may have to rewrite the query/view:
  – Avoid nested queries, temporary relations, complex conditions, and operations like DISTINCT and GROUP BY.