Database Tuning

Chapter 16, Part B

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 CSJDPQV, 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 into Contracts.
  - Thus, Contracts is no longer in 3NF.
- We 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.)

Choice of Decompositions

- There are 2 ways to decompose CSJDPQV into BCNF:
  - SDP and CSJDPQV: lossless-join but not dep-preserving.
  - SDP, CSJDPQV and CJP: dep-preserving as well.

- The difference between these is really the cost of enforcing the FD JP → C.
  - 2nd decomposition: Index on JP on relation CJP.
  - 1st:
    CREATE ASSERTION CheckDep
    CHECK (NOT EXISTS (SELECT * FROM PartInfo P, ContractInfo C WHERE P.pid=C.pid AND P did=C did GROUP BY C.pid, P.pid HAVING COUNT (C did) > 1))
Choice of Decompositions (Contd.)
- The following ICs were given to hold: JP → C, SD → P, C is the primary key.
- Suppose that, in addition, a given supplier always charges the same price for a given part: SPQ → V.
- If we decide that we want to decompose CSJDPQV into BCNF, we now have a third choice:
  - Begin by decomposing it into SPQV and CSJDPQ.
  - Then, decompose CSJDPQ (not in 3NF) into SDP, CSJQ.
  - This gives us the lossless-join decomp: SPQV, SDP, CSJQ.
  - To preserve JP → C, we can add CP, as before.
- Choice: {SPQV, SDP, CSJQ} or {SDP, CSJQV} ?

Decomposition of a BCNF Relation
- Suppose that we choose {SDP, CSJQV}. This is in BCNF, and there is no reason to decompose further (assuming that all known ICs are FDs).
- However, suppose that these queries are important:
  - Find the contract held by supplier S.
  - Find the contract that department D is involved in.
- Decomposing CSJQV further into CS, CD and CJQV could speed up these queries. (Why?)
- On the other hand, the following query is slower:
  - Find the total value of all contracts held by supplier S.

Horizontal Decompositions
- Our definition of decomposition: Relation is replaced by a 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 the original, but a subset of the 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. This means that queries on Contracts will often contain the condition vuv > 10000.
- One way to deal with this is to build a clustered B+ tree index on the vuv field of Contracts
- A second approach is to replace contracts by two new relations: LargeContracts and 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 Conceptual Schema Changes
```sql
CREATE VIEW Contracts (cid, sid, jid, did, pid, qty, val)
AS SELECT *
FROM LargeContracts
UNION
SELECT *
FROM SmallContracts
```
- The replacement of Contracts by LargeContracts and SmallContracts can be masked by the view.
- However, queries with the condition vuv > 10000 must be asked wrt LargeContracts for efficient execution: so users concerned with performance have to be aware of the change.

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.
  - Selections involving arithmetic or string expressions.
  - Selections involving OR conditions.
  - 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.
Rewriting SQL Queries

- Complicated by interaction of:
  - NULLs, duplicates, aggregation, subqueries.
- **Guideline**: Use only one "query block", if possible.

```
SELECT DISTINCT * FROM Sailors S
WHERE S.name IN (SELECT Y.name FROM YoungSailors Y)
```

- Not always possible ...

```
SELECT * FROM Sailors S
WHERE S.name IN (SELECT DISTINCT Y.name FROM YoungSailors Y)
WHERE S.name = Y.name
```

Summary on Unnesting Queries

- **DISTINCT** at top level: Can ignore duplicates.
  - Can sometimes infer **DISTINCT** at top level (e.g. subquery clause matches at most one tuple)
- **DISTINCT** in subquery w/o **DISTINCT** at top:
  - Hard to convert.
- Subqueries inside OR: Hard to convert.
- All subqueries: Hard to convert.
  - EXISTS and ANY are just like IN.
- Aggregates in subqueries: Tricky.
- Good news: Some systems now rewrite under the covers (e.g. DB2).

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.id) FROM Employee E
GROUP BY E.id
```

- Consider DBMS use of index when writing arithmetic expressions: Employment will benefit from index on `Employee`, but might not benefit from index on `Employee`!

Guidelines for Query Tuning (Contd.)

- Avoid using intermediate relations:

```
SELECT E.id, AVG(E.salary)
FROM Employee E, Department D
WHERE E.dept = D.dept
AND D.name = 'IT'
```

- Does not materialize the intermediate rel Temp.
- If there is a dense B-tree index on `Emp.dept, salary`, 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!); otherwise, 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.