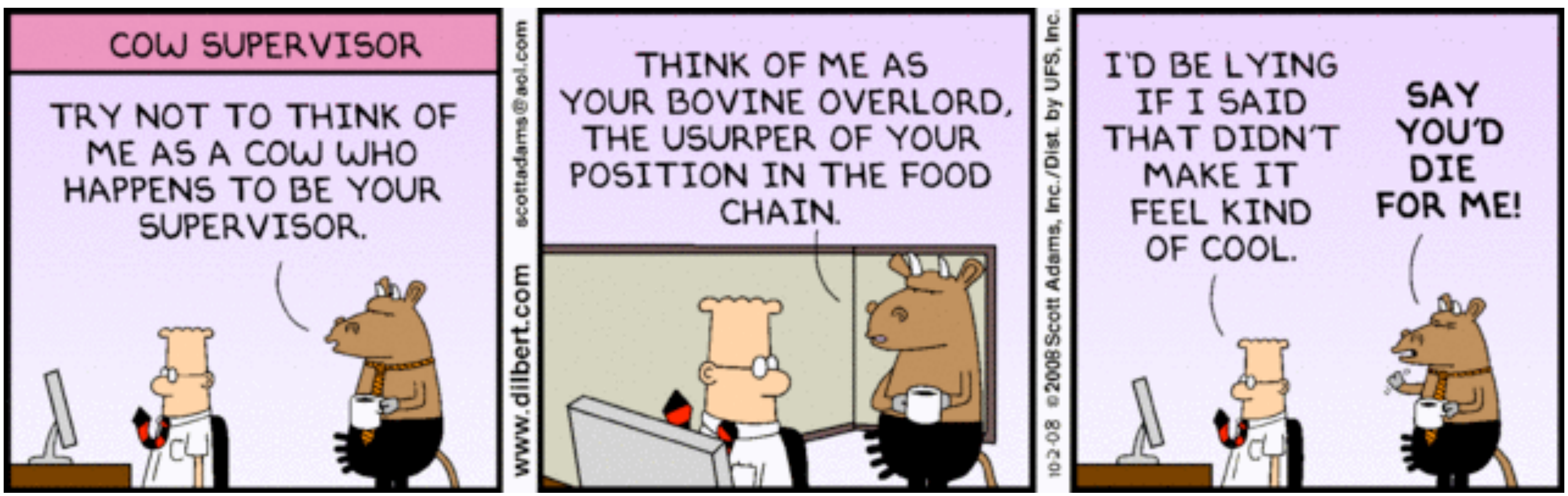
Database Design and Tuning

Chapter 20



Overview

- 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.

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?
- 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 ...

Index Selection for Joins

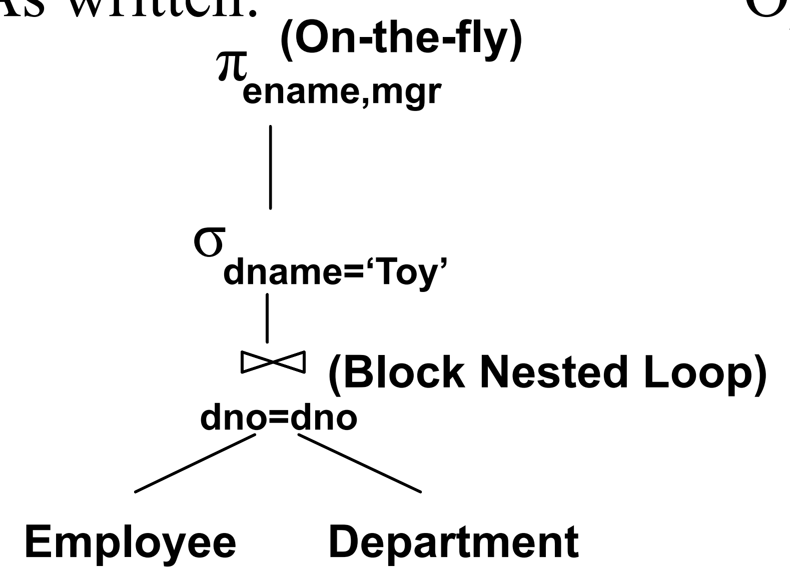
- When considering a join condition:
  - Hash index on inner relation is very good for Index Nested Loops.
    - Should be clustered if join column is not a key for inner, and inner tuples need to be retrieved.
    - Clustering less important if join is on key
  - *Clustered* B+ tree on join column(s) good for Sort-Merge. (saves a sort on one relation)

(We discussed indexes for single-table queries in Chapter 8.)

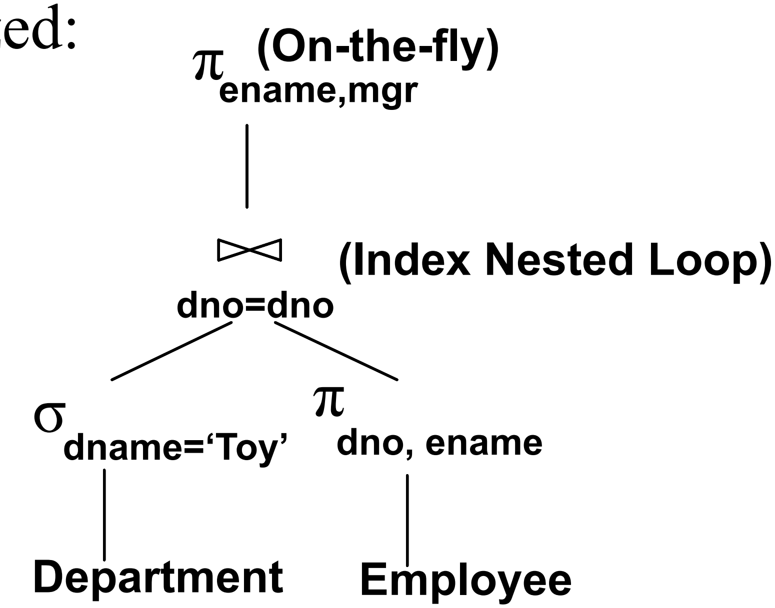
Example 1 – Optimize Query

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

As written:



Optimized:



Example 2 – Create Index

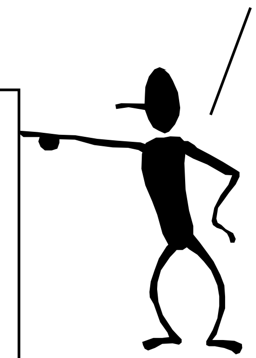
```
SELECT  E.ename, D.mgr
FROM  Emp E, Dept D
WHERE  D.dname='Toy' AND E.dno=D.dno
```

- 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.

Example 3 – More precise SQL

```
SELECT  E.ename, D.mgr
FROM  Emp E, Dept D
WHERE  E.sal >= 10000 AND E.sal <= 20000
  AND E.hobby='Stamps' AND E.dno=D.dno
```

Use of the BETWEEN operator is recommended; it allows the optimizer to recognize both parts of a range selection



```
SELECT  E.ename, D.mgr
FROM  Emp E, Dept D
WHERE  E.sal BETWEEN 10000 AND 20000
  AND E.hobby='Stamps' AND E.dno=D.dno
```

Example 3 – Sometimes Unclear

```
SELECT  E.ename, D.mgr
FROM  Emp E, Dept D
WHERE  E.sal BETWEEN 10000 AND 20000
  AND E.hobby='Stamps' AND E.dno=D.dno
```

- Clearly, Emp should be the outer relation.
  - 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!*

Clustering and Joins

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

- Clustering is especially important when accessing inner tuples in INL.
  - Should make index on *E.dno* clustered.
- Suppose that the WHERE clause is instead:

  WHERE E.hobby=‘Stamps AND E.dno=D.dno

  - If many employees collect stamps, Sort-Merge join may be worth considering. A *clustered* index on D.dno would help.
- *Summary*: Clustering is useful whenever many tuples are to be retrieved.

Tuning the Conceptual Schema

- The choice of conceptual schema should be guided by the workload, in addition to redundancy issues:
  - We may settle for a 3NF schema rather than BCNF.
  - Workload may influence the 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 is in use, called *schema evolution*; might want to mask some of these changes from applications 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 FDs 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. (recall SD → P drives this decomposition)
  - Lossless decomposition, but not dependency-preserving.
  - Adding CJP makes it dependency-preserving, at the cost of redundancy.
- 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.
- We might choose to modify Contracts thusly 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 CSJDQV; lossless-join but not dep-preserving.
  - SDP, CSJDQV 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.sid=C.sid AND P.did=C.did
GROUP BY C.jid, P.pid
HAVING COUNT (C.cid) > 1))
```

Choice of Decompositions (Contd.)

- The following ICs were given to hold:
  JP $\rightarrow$ C, SD $\rightarrow$ P, C is the primary key.
- Suppose that, in addition, a given supplier always charges the same price for a given part: SPQ $\rightarrow$ 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, CSJDQ.
  - This gives us the lossless-join decomp: SPQV, SDP, CSJDQ.
  - To preserve JP $\rightarrow$ C, we can add CJP, as before.
- Choice: { SPQV, SDP, CSJDQ } or { SDP, CSJDQV }?

Decomposition of a BCNF Relation

- Suppose that we choose { SDP, CSJDQV }. 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 contracts with supplier S.*
  - *Find the contracts made by department D.*
- Decomposing CSJDQV 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 *val > 10000*.
- One way to deal with this is to build a clustered B+ tree index on the *val* 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

```
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 *val>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.

SQLite: Behind the Curtains

- Access to the DBMS's query execution plan

```
$ sqlite3 origMovies.db
SQLite version 3.6.20
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> EXPLAIN QUERY PLAN SELECT COUNT(*)
   ...> FROM Customers C, Rentals R
   ...> WHERE R.cardNo=C.cardNo AND movieID=15922;
0|0|TABLE Customers AS C
1|1|TABLE Rentals AS R WITH INDEX sqlite_autoindex_Rentals_1
sqlite> EXPLAIN QUERY PLAN SELECT COUNT(*)
   ...> FROM Movies M, Rentals R
   ...> WHERE R.movieID=M.movieID and M.Title='The Graduate';
0|1|TABLE Rentals AS R
1|0|TABLE Movies AS M USING PRIMARY KEY
```

SQLite3: Use of Indices

- See the DBMS's query execution plan

```
$ sqlite3 movies.db
:
index|cardNo_movieId|Rentals|544569|CREATE INDEX cardNo_movieId on Rentals(cardNo,movieId)
index|Cust_cardNo|Customers|697242|CREATE INDEX Cust_cardNo on Customers(cardNo)
index|Mov_Id|Movies|703747|CREATE INDEX Mov_Id on Movies(movieId)
index|Inx_movieID|Rentals|703948|CREATE INDEX Inx_movieID on Rentals(movieId)
:
sqlite> EXPLAIN QUERY PLAN SELECT COUNT(*)
  ...> FROM Rentals R, Customers C
  ...> WHERE R.cardNo=C.cardNo AND movieID=15922;
0|0|TABLE Rentals AS R WITH INDEX Inx_movieID
1|1|TABLE Customers AS C USING PRIMARY KEY
sqlite> EXPLAIN QUERY PLAN SELECT COUNT(*)
   ...> FROM Movies M, Rentals R
   ...> WHERE R.movieID=M.movieID and M.Title='The Graduate';
 0|0|TABLE Movies AS M
 1|1|TABLE Rentals AS R WITH INDEX Inx_movieID
```

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.sname IN
      (SELECT Y.sname
         FROM YoungSailors Y)
```

=

```
SELECT DISTINCT S.*
  FROM Sailors S,
       YoungSailors Y
 WHERE S.sname = Y.sname
```

- *Not always possible ...*

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

≠

```
SELECT S.*
  FROM Sailors S,
       YoungSailors Y
 WHERE S.sname = Y.sname
```

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.*
- <u>Good news</u>: 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.age)
FROM  Employee E
GROUP BY  E.dno
HAVING  E.dno=102
```

```
SELECT  MIN (E.age)
FROM  Employee E
WHERE  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.mgrname='Joe'
GROUP BY  E.dno
```

vs.

```
SELECT  *  INTO  Temp
FROM  Emp E, Dept D
WHERE  E.dno=D.dno
   AND  D.mgrname='Joe'
```

and

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

- Does not materialize the intermediate reln 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

- 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?

Summary

- 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.