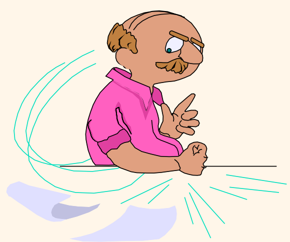
Introduction to Data Management

Lecture #23
(Physical DB Design II)



Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

- Two HW assignments remain:
  - HW #7: Due tomorrow (5 PM)
    - Physical DB design (for MySQL and beyond)
  - HW #8: Due next Thursday, June 6th (5 PM)!
    - NoSQL and **NoLateDay**
- Today's plan :
  - Physical DB design wrap-up
  - Then: **NoSQL & Big Data** (*a la* AsterixDB)
    - *Not* in the book,, so be sure to see the wiki for readings!
    - You can (and should) study ahead, e.g., by going through the Apache AsterixDB SQL++ Primer (Using SQL++).

*Index Selection for **Joins***

- When considering a join condition:
  - Index Nested Loop join (**INLJ**) method:
    - For each outer table tuple, use its join attribute value to probe the inner table for tuples to join (match) it with.
    - Indexing the inner table's join column will help!
    - Good for this index to be *clustered* if the join column is *not* the inner's PK (e.g., FK) and inner tuples need to be fetched.
  - Sort-Merge join (**SMJ**) method:
    - Sort outer and inner tables on join attribute value and then scan them concurrently to match tuples.
    - *Clustered* B+ trees on both join column(s) fantastic for this!
  - Hash join (**HJ**) method:
    - Indexing is not needed (not for the join, anyway).

Example 1

```
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, an index on ***D***.dno is not needed (not useful).
- Hash index on *E.dno* allows us to fetch matching (inner) Emp tuples for each outer Dept tuple.
- What if **WHERE** included: "... AND E.age=25"?
  - Could instead retrieve Emp tuples using index on *E.age*, then join with Dept tuples satisfying *dname* selection. (Comparable to strategy that uses the *E.dno* index.)
  - So, if *E.age* index were already created, this query provides less motivation for adding an *E.dno* index.

Example 2

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 (E) should be the outer relation.
  - Suggests that we build an index (hashed) 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 rough rule of thumb, equality selections tend to be 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 choose for a query. ∴ ***Understand query optimizer!***

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 INLJ (index nested loops join).
  - Should make index on *E.dno* clustered. (Q: See why?)
- Suppose that the WHERE clause were instead:

  WHERE E.hobby= 'Stamps' AND E.dno=D.dno
  - If most 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 for one value or a range of values.

*Tuning the **Conceptual** Schema*

- The choice of conceptual schema should be guided by the workload, in addition to redundancy issues:
  - We may go for a 3NF (or lower!) schema rather than BCNF.
  - Workload may influence the choice we make in decomposing a relation into 3NF or BCNF.
  - We might *denormalize* (i.e., **undo** a decomposition step), or we might add fields to a relation.
  - We might consider *vertical decompositions*.
- If such changes come after a database is in use, it's called *schema evolution*; might want to mask some of the changes from applications by defining *views*.

Some Example Schemas (& Tradeoffs)

Suppliers(<u>sid</u>, <u>sname</u>, address, phone, …)
Parts(<u>pid</u>, pname, size, color, listprice, …)
Stock(<u>sid, pid</u>, price, quantity)

- What if a large fraction of the workload consists of Stock queries that also want suppliers' names?
  - SELECT s.sid, s.*sname*, AVG(t.price) FROM Suppliers s, Stock t WHERE s.sid = t.sid GROUP BY s.sid, s.*sname*;
  - Consider: ALTER TABLE Stock ADD COLUMN sname …;
  - This is denormalization (on purpose, for performance!)
    - If sid→sname and sname→sid, Stock would then be in 3NF.
    - ***Q:*** If sid→sname (but not vice versa), what NF would Stock be in?

Vertical Partitioning

- Consider a table with lots of columns, not all of which are of interest to all queries.
  - *Ex:* Emp(**eno**, email, name, addr, salary, age, dno)
- A given workload might actually turn out to be a "union of sub-workloads" in reality.
  - Employee communications queries
  - Employee compensation queries/analytics
  - Employee department queries/analytics

Vertical Partitioning Example

eno	email	name	addr	salary	age	dno
1	joe@aol.com	Joe	1 Main St.	100000	25	10
2	sue@gmail.com	Sue	10 State St.	125000	28	20
3	zack@fb.com	Zack	100 Wall St.	2500000	40	30

(Vertical partitioning: ⋈)

eno	email	name	addr
1	joe@aol.com	Joe	1 Main St.
2	sue@gmail.com	Sue	10 State St.
3	zack@fb.com	Zack	100 Wall St.

eno	salary	age
1	100000	25
2	125000	28
3	2500000	40

eno	dno
1	10
2	20
3	30

(*In the limit:* We get a column store!)

Horizontal Partitioning

- Occasionally, we may want to instead replace a relation by a set of relations that are *selections*.
  - Each new relation has same schema (columns) 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*.)
  - The original relation is the UNION (ALL) of the new ones (i.e., rather than the JOIN of the new ones).

Horizontal Partitioning Example

eno	email	name	addr	salary	age	dno
1	joe@aol.com	Joe	1 Main St.	100000	25	10
2	sue@gmail.com	Sue	10 State St.	125000	28	20
3	zack@fb.com	Zack	100 Wall St.	2500000	40	30

(Horizontal partitioning: ∪)

eno	email	name	addr	salary	age	dno
1	joe@aol.com	Joe	1 Main St.	100000	25	10
2	sue@gmail.com	Sue	10 State St.	125000	28	20

eno	email	name	addr	salary	age	dno
3	zack@fb.com	Zack	100 Wall St.	2500000	40	30

Potential Horizontal Rationale

- 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 to create a clustered B+ tree index on Contracts(*val*).
- Another approach could be to replace Contracts by two relations, LargeContracts & SmallContracts, with the same attributes.
  - Performs like index but without index overhead.
  - Can then cluster on other (perhaps different!) attributes.

Masking Schema Changes

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

- Replacement of Contracts by LargeContracts and SmallContracts can be masked by this view.
- Note: queries with *val>10000* can be written against LargeContracts* for faster execution; users concerned with performance must be aware of this change.

(*The DBMS is unaware of the two tables' value constraints.)

In General: Tuning 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 arithmetic or LIKE expressions.
  - Selections involving OR conditions.
  - Selections involving null values.
  - Lack of advanced evaluation features like some 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 or view.

Miscellany for Query Tuning

- Minimize the use of **DISTINCT**: Don't use the D-word if duplicates are acceptable or if the answer contains a key.
- Consider the DBMS's use of indexes when writing arithmetic expressions: *E.age = 2*D.age* will benefit from an index on *E.age*, but it probably wouldn't benefit from an index on *D.age*!

Physical DB Design Summary

- End-to-end DB design consists of several tasks: *requirements analysis, conceptual design, schema refinement, physical design* and finally *tuning*.
  - In general, one goes back and forth between tasks to refine a DB design
  - Decisions made in one task can influence choices in another task.
- Understanding the *workload* for the application, and performance goals, is essential to good design.
  - What are the important queries and updates? What attributes/relations are involved?

Summary (Cont'd.)

- 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.

Summary (Cont'd.)

- Over time, indexes may have to be fine-tuned (dropped, created, re-built, ...) for performance.
  - Be sure to examine the query plan(s) used by the system and adjust the choices 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...?