Introduction to Data Management

Lecture #21
(Physical DB Design)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

- Homework info:
  - HW #7: Due this coming Friday (5 PM).
    - Be sure to drop indexes when done with each section!
  - Then just one more HW remains (on “NoSQL”).
- This week’s plan:
  - Today: Physical DB design (e.g., use of indexes).
  - Indexing quiz in discussions this week.
  - Next up: NoSQL & Big Data (a la AsterixDB)
    - Not in book – see paper linked to wiki syllabus!
- Any lingering questions on indexing...?
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 should start by understanding the workload:
  - Most important queries and how often they arise.
  - Most important updates and how often they arise.
  - Desired performance goals for those queries/updates?

Decisions to Be Made Include...

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be their search keys? Should we build several indexes?
- For each index, what kind of an index should it be?
  - B+ tree? Hashed? Clustered? Un-clustered?
- Should we make changes to the conceptual schema?
  - Consider alternative normalized schemas? (There are multiple choices when decomposing into BCNF, etc.)
  - Should we ``undo'' some decomposition steps and settle for a lower normal form? (“Denormalization.”)
  - Horizontal partitioning, materialized views, replication, ...
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? (And how selective are these conditions expected to be?)

- For each update in the workload:
  - Which attributes are involved in selection/join conditions? (And how selective are these conditions likely to be?)
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Index Classification (Review)

- **Primary vs. secondary**: If search key contains the primary key, then called the primary index.
  - **Unique** index: Search key contains a candidate key.
- **Clustered vs. unclustered**: If order of data records is the same as, or `close to`, the order of stored data records, then called a clustered index.
  - A table can be clustered on at most one search key.
  - Cost of retrieving data records via an index varies greatly based on whether index is clustered or not!
Clustered vs. Unclustered Indexes  
(Review)

Choice of Indexes (Cont’d.)

- **One approach:** Consider the most important queries in turn. Consider the best query plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
  - This implies that we must understand and see how a DBMS evaluates its queries. *(Query evaluation plans.)*
  - Let’s start by discussing simple 1-table queries!
- Before creating an index, must also consider its impact on updates in the workload.
  - **Trade-off:** Indexes can make queries go faster, but updates will become slower. *(Indexes require disk space, too.)*
Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition: hashed index (or B+ tree if not available).
  - Range query: B+ tree index.
    - Clustering especially useful for range queries; also helps equality queries with duplicates (non-key field index).
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes important for range queries.
  - Such indexes can sometimes enable index-only strategies for important queries (e.g., aggregates / grouped aggregates).
    - For index-only strategies, clustering isn’t important!
- Choose indexes that benefit as many queries as possible.
  - Only one index can be clustered per relation, so choose it based on important queries that can benefit the most from clustering.

Examples of Clustered Indexes

- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - Should the index be clustered?

- Consider the GROUP BY query.
  - If most tuples have E.age > 10, using E.age index and sorting the retrieved tuples may be costly.
  - Clustered E.dno index may win!

- Equality queries & duplicates:
  - Clustering on E.hobby helps!
Indexes with Composite Search Keys

- **Composite Search Keys**: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - (age=20 AND sal=75)
  - Range query: Some field value is a range, not a constant. E.g. again wrt <sal,age> index:
    - age=20; or (age=20 AND sal > 10)
- Data entries in index sorted by search key to support such range queries.
  - **Lexicographic order**

Various possible composite key indexes using lexicographic order.

Composite Search 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.
  - Note: Choice of index key orthogonal to clustering.
- If condition is: **20<age<30 AND 3000<sal<5000**:
  - Clustered B+ 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! (Think about why!)
- Composite indexes are larger; updated more often.
**Index-Only Query Plans**

- Some queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

- (Sometimes called a “covering index” for the given query.)

  - \[ \text{SELECT } E.\text{dno}, \text{COUNT}(\ast) \]
  - \[ \text{FROM } \text{Emp } E \]
  - \[ \text{GROUP BY } E.\text{dno} \]

  - \[ \text{SELECT } E.\text{dno}, \text{MIN}(E.\text{sal}) \]
  - \[ \text{FROM } \text{Emp } E \]
  - \[ \text{GROUP BY } E.\text{dno} \]

  - \[ \text{SELECT } \text{AVG}(E.\text{sal}) \]
  - \[ \text{FROM } \text{Emp } E \]
  - \[ \text{WHERE } E.\text{age}=25 \text{ AND } E.\text{sal} \text{ BETWEEN } 3000 \text{ AND } 5000 \]

**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 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 key and inner tuples need to be retrieved.
  - **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) good for this!*
  - **Hash join (HJ) method:**
    - Indexing not needed (not just 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.
- 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 the $E.dno$ index.
  - So, if $E.age$ index is 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 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 consider 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 a value or 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 settle for a 3NF 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

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 were given to hold:
  - JP → C
  - SD → P
  - C → CSJDPQV.
  - Currently this relation is in 3NF....

Settling for 3NF vs BCNF

- CSJDPQV can be decomposed into SDP and CSJDQV, and both relations would then be in BCNF
  - Lossless decomposition, not dependency-preserving.
  - Adding CJP would make it dependency-preserving too.
- But – suppose that this query is very important:
  - Find the # 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 (redundant) field budget B to Contracts.
  - This would introduce the FD \( D \rightarrow B \) w.r.t. Contracts.
  - Thus, Contracts would no longer even be in 3NF...!
- Might choose to modify Contracts this way if the query is sufficiently important and we can’t obtain adequate performance otherwise (i.e., by adding indexes or finding an alternative 3NF schema.)

Decomposition of a BCNF Relation

- Suppose that we choose \{SDP, CSJDQV\}. This is in BCNF, and there is no reason to decompose further.
- However, suppose that these queries are frequent:
  - Find the contracts held by suppliers S.
  - Find the contracts 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 following query would now be slower:
  - Find the total value of all contracts held by supplier S.
To Be Continued...