Database Tuning and Physical Design (cont'd)

Spring 2018

School of Computer Science
University of Waterloo

Databases CS348

Physical Database Design and Tuning

Physical Design The process of selecting a physical schema (collection of data structures) to implement the conceptual schema

Tuning Periodically adjusting the physical and/or conceptual schema of a working system to adapt to changing requirements and/or performance characteristics

Good design and tuning requires understanding the database **workload**.

Workload Modeling

Definition (Workload Description)

A *workload description* contains

- the most important queries and their frequency
- the most important updates and their frequency
- the desired performance goal for each query or update

- For each query:
  - Which relations are accessed?
  - Which attributes are retrieved?
  - Which attributes occur in selection/join conditions? How *selective* is each condition?
- For each update:
  - Type of update and relations/attributes affected.
  - Which attributes occur in selection/join conditions? How *selective* is each condition?

Database Tuning

Goal

Make a set of applications execute "as fast as possible".

- optimize for response time?
- optimize for overall throughput?

How can we affect performance (as DBAs)?

- make queries run faster (data structures, clustering, replication)
- make updates run faster (locality of data items)
- minimize congestion due to concurrency

The Physical Schema

- A storage strategy is chosen for each relation
  - Possible storage options:
    - Unsorted (heap) file
    - Sorted file
    - Hash file

- Indexes are then added
  - Speed up queries
  - Extra update overhead
  - Possible index types:
    - B-trees (actually, B+-trees)
    - R trees
    - Hash tables
    - ISAM, VSAM
    - ...

A Table Scan

```
Select *
From Employee
Where Lastname = 'Smith'
```

- To answer this query, the DBMS must search the blocks of the database file to check for matching tuples.
- If no indexes exist for Lastname (and the file is unsorted with respect to Lastname), all blocks of the file must be scanned.

Creating Indexes

```
CREATE INDEX LastnameIndex
ON Employee(Lastname) [CLUSTER]

DROP INDEX LastnameIndex
```

Primary effects of `LastnameIndex`:

- Substantially reduce execution time for selections that specify conditions involving `Lastname`
- Increase execution time for insertions
- Increase or decrease execution time for updates or deletions of tuples from `Employee`
- Increase the amount of space required to represent `Employee`

Clustering vs. Non-Clustering Indexes

- An index on attribute *A* of a relation is a **clustering** index if tuples in the relation with similar values for *A* are stored together in the same block.
- Other indices are **non-clustering** (or secondary) indices.

Note

A relation may have at most one clustering index, and any number of non-clustering indices.

Co-Clustering Relations

Definition (Co-Clustering)

Two relations are **co-clustered** if their tuples are interleaved within the same file

- Co-clustering is useful for storing hierarchical data (1:N relationships)
- Effects on performance:
  - Can speed up joins, particularly foreign-key joins
  - Sequential scans of either relation become slower

Range Queries

- B-trees can also help for **range queries**:

  ```
  SELECT *
  FROM R
  WHERE A ≥ c
  ```

- If a B-tree is defined on *A*, we can use it to find the tuples for which $A = c$. Using the forward pointers in the leaf blocks, we can then find tuples for which $A > c$.

Multi-Attribute Indices

- It is possible to create an index on several attributes of the same relation. For example:

```
CREATE INDEX NameIndex
ON Employee(Lastname,Firstname)
```

- The order in which the attributes appear is important. In this index, tuples (or tuple pointers) are organized first by `Lastname`. Tuples with a common surname are then organized by `Firstname`.

Using Multi-Attribute Indices

- The `NameIndex` index would be useful for these queries:

```
SELECT *
FROM Employee
WHERE Lastname = 'Smith'
```

```
SELECT *
FROM Employee
WHERE Lastname = 'Smith'
AND Firstname = 'John'
```

- It would be *very* useful for these queries:

```
SELECT Firstname
FROM Employee
WHERE Lastname = 'Smith'
```

```
SELECT Firstname, Lastname
FROM Employee
```

- It would not be useful at all for this query:

```
SELECT *
FROM Employee
WHERE Firstname = 'John'
```

Query Plan Tools (EXPLAIN)

How do we know what plan is used (and what the estimated cost is)?
⇒ `db2expln` and `dynexpln` tools

```
select name from author, wrote where aid=author
```

(without index)

```
Estimated Cost        = 50
Estimated Cardinality = 120

Optimizer Plan:

     RETURN
       |
     MSJOIN
    /      \
 TBSCAN  TBSCAN
   |       |
  SORT    SORT
   |       |
 TBSCAN  TBSCAN
   |       |
 Table:  Table:
 AUTHOR  WROTE
```

(index on `wrote(author)`)

```
Estimated Cost        = 25
Estimated Cardinality = 120

Optimizer Plan:

       RETURN
         |
       MSJOIN
      /      \
 TBSCAN       IXSCAN
   |         /     \
  SORT    Index:  Table:
   |      AW      WROTE
 TBSCAN
   |
 Table:
 AUTHOR
```

More complex Designs

Multi-attribute Indices

complex search/join conditions (in lexicographical order!)
index-only plans (several *clustered indices*)

Join Indices

allow replacing joins by index lookups

Materialized Views

allow replacing subqueries by index lookups

Problem 1

How does the *query optimizer* know if/where to use such indices/views?

Problem 2

Balance between *cost of rematerialization* and *savings for queries*.

Physical Design Guidelines

1. Don't index unless the performance increase outweighs the update overhead
2. Attributes mentioned in WHERE clauses are candidates for index search keys
3. Multi-attribute search keys should be considered when
   - a WHERE clause contains several conditions; or
   - it enables index-only plans
4. Choose indexes that benefit as many queries as possible
5. Each relation can have at most one clustering scheme; therefore choose it wisely
   - Target important queries that would benefit the most
     - Range queries benefit the most from clustering
     - Join queries benefit the most from co-clustering
   - A multi-attribute index that enables an index-only plan does not benefit from being clustered

Index Selection and Tools

Idea

Convert *physical design* into another *optimization problem*

- generate the space of all possible physical designs
- pick the best one *based on a given WORKLOAD*

Workload

An abstraction of applications executed against a database:

- list of queries
- list of updates
- frequencies/probabilities of the above
- sequencing constraints
- ...

Index Selection and Tools Example

```
rees$ db2advis -d cs338
               -s "select name from author,wrote where aid=author"

Calculating initial cost (without recommmended indexes) [25.390385]
Initial set of proposed indexes is ready.
Found maximum set of [2] recommended indexes
Cost of workload with all indexes included [0.364030] timerons
total disk space needed for initial set [   0.014] MB
total disk space constrained to         [  -1.000] MB
  2  indexes in current solution
 [ 25.3904] timerons  (without indexes)
 [  0.3640] timerons  (with current solution)
 [%98.57] improvement

 Trying variations of the solution set.
--
-- execution finished at timestamp 2006-11-23-12.25.24.205770
--
-- LIST OF RECOMMENDED INDEXES
-- ===========================
-- index[1],    0.009MB
   CREATE INDEX WIZ8 ON "DAVID   "."AUTHOR" ("AID" ASC, "NAME" ASC) ;
-- index[2],    0.005MB
   CREATE INDEX AW ON "DAVID   "."WROTE" ("AUTHOR" ASC) ;
-- ===========================
Index Advisor tool is finished.
```

Schema Tuning and Normal Forms

So far we only *added data structures* to improve performance.

what to do if this isn't enough?

Changes to the *conceptual design*

Goals:

- avoid expensive operations in query execution (joins)
- retrieve *related data* in fewer operations

Techniques:

- alternative normalization/weaker normal form
- co-clustering of relations (if available)/denormalization
- vertical/horizontal partitioning of data (and views)
- avoiding concurrency hot-spots

Tuning the Conceptual Schema

Suppose that after tuning the physical schema, the system still does not meet the performance goals!

- Adjustments can be made to the conceptual schema:
  - Re-normalization
  - Denormalization
  - Partitioning

Warning

Unlike changes to the physical schema, changes to the conceptual schema of an operational system—called *schema evolution*—often can't be completely masked from end users and their applications.

Denormalization

Normalization is the process of decomposing schemas to reduce redundancy

Denormalization is the process of merging schemas to intentionally *increase* redundancy

In general, redundancy *increases update overhead* (due to change anomalies) but *decreases query overhead*.

The appropriate choice of normal form depends heavily upon the workload.

Partitioning

- Very large tables can be a source of performance bottlenecks
- *Partitioning* a table means splitting it into multiple tables for the purpose of reducing I/O cost or lock contention
  1. Horizontal Partitioning
     - Each partition has all the original columns and a subset of the original rows
     - Tuples are assigned to a partition based upon a (usually natural) criteria
     - Often used to separate operational from archival data
  2. Vertical Partitioning
     - Each partition has a subset of the original columns and all the original rows
     - Typically used to separate frequently-used columns from each other (concurrency *hot-spots*) or from infrequently-used columns

Tuning Queries

- Changes to the physical or conceptual schemas impacts *all* queries and updates in the workload.
- Sometimes desirable to target performance of specific queries or applications
- Guidelines for tuning queries:
  1. Sorting is expensive. Avoid unnecessary uses of ORDER BY, DISTINCT, or GROUP BY.
  2. Whenever possible, replace subqueries with joins
  3. Whenever possible, replace correlated subqueries with uncorrelated subqueries
  4. Use vendor-supplied tools to examine generated plan. Update and/or create statistics if poor plan is due to poor cost estimation.

Tuning Applications

Guidelines for tuning applications:

1. Minimize communication costs
   - Return the fewest columns and rows necessary
   - Update multiple rows with a WHERE clause rather than a cursor
2. Minimize lock contention and hot-spots
   - Delay updates as long as possible
   - Delay operations on hot-spots as long as possible
   - Shorten or split transactions as much as possible
   - Perform insertions/updates/deletions in batches
   - Consider lower isolation levels

Summary

Physical design has *enormous impact* on performance

- Decisions based on *understanding* what the DBMS is doing
  $\Rightarrow$ query execution, transaction processing, and query optimization
- Modern systems provide tools for DBAs (EXPLAIN)
- VERY active area of research