Introduction
- We will be talking at length about “database design”
  - Conceptual Schema: info to capture, tables, columns, views, etc.
  - Physical Schema: indexes, clustering, etc.
- Physical design linked tightly to query optimization
  - So we’ll study this “bottom up”
  - But note: DB design is usually “top-down”
    - Conceptual then physical. Then iterate.
- 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.

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

Creating an ISUD Chart

Insert, Select, Update, Delete Frequencies

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Frequency</th>
<th>% table</th>
<th>Employee Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Payroll Run</td>
<td>monthly</td>
<td>100</td>
<td>S</td>
</tr>
<tr>
<td>Add Emps</td>
<td>daily</td>
<td>0.1</td>
<td>I</td>
</tr>
<tr>
<td>Delete Emps</td>
<td>daily</td>
<td>0.1</td>
<td>D</td>
</tr>
<tr>
<td>Give Raises</td>
<td>monthly</td>
<td>10</td>
<td>U</td>
</tr>
</tbody>
</table>

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? Dynamic/static?
- Should we make changes to the conceptual schema?
  - More on this later...
- Horizontal partitioning, replication, views ...

Index Selection
- One approach:
  - Consider most important queries in turn.
  - Consider best plan using the current indexes, and see if better plan is possible with an additional index.
  - If so, create it.
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: indexes can make queries go faster, updates slower. Require disk space, too.
**Issues to Consider in Index Selection**

- Attributes mentioned in a WHERE clause are candidates for index search keys.
  - Range conditions are sensitive to clustering
  - Exact match conditions don't require clustering
    - Or do they???: :-(
- Try to choose indexes that benefit as many queries as possible.
- NOTE: only one index can be clustered per relation!
- Notice that only one index can be clustered per relation!
- So choose it based on important queries that benefit the most from clustering!!

**Clustering and Joins**

- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - If range selections are involved, order of attributes should be carefully chosen to match the range ordering.
  - Such indexes can sometimes enable index-only strategies for important queries.
    - For index-only strategies, clustering is not important!
- When considering a join condition:
  - B+ tree on inner is very good for Index Nested Loops.
    - Should be clustered if join column is not key for inner, and inner tuples need to be retrieved.
  - Clustered B+ tree on join column(s) good for Sort-Merge.

**Examples of Clustering**

- B+ tree index on D.dname supports 'Toy' selection.
  - Given this, index on D.dno is not needed.
- B+ tree 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.

**Issues in Index Selection (Contd.)**

- All selections are on Emp so it should be the outer relation in any Index NL join.
  - Suggests that we build a B+ tree index on D.dno.
- What index should we build on Emp?
  - B+ tree on E.dno 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!

**Example 1**

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

**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
```

**Examples**

- B+ tree index on E.Age can be used to get qualifying tuples.
  - How selective is the condition?
  - Is the index clustered?
- Consider the GROUP BY query.
  - If many tuples have $E.Age > 10$, using $E.Age$ index and sorting the retrieved tuples may be costly.
  - Clustered $E.dno$ index may be better!
- Equality queries and duplicates:
  - Clustering on $E.hobby$ helps!

**Summary**

- 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.
Multi-Attribute Index 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.
  - Such indexes also called composite or concatenated indexes.
  - Choice of index key orthogonal to clustering etc.
• If condition is: 20<age<30 AND 3000<sal<5000:
  - Clustered 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!
• Composite indexes are larger, updated more often.

Index-Only Plans

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

Horizontal Decompositions

• Usual Def. of decomposition: Relation is replaced by collection of relations that are projections. Most important case.
  - We will talk about this at length as part of Conceptual DB Design.
• Sometimes, might want to replace relation by a collection of relations that are selections.
  - Each new relation has same schema as original, but subset of rows.
  - Collectively, new relations contain all rows of the original.
  - Typically, the new relations are disjoint.

Horizontal Decompositions (Contd.)

• Contracts (Cid, Sid, Jid, Did, Pid, Qty, Val)
  - Suppose that contracts with value > 10000 are subject to different rules.
  - So queries on Contracts will often say WHERE val>10000.
• One approach: clustered B+ tree index on the val field.
• Second approach: 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

• Horizontal Decomposition from above
• Masked by a view.
  - NOTE: queries with condition val>10000 must be asked wrt LargeContracts for efficiency: so some users may have to be aware of change.
  - I.e. the users who were having performance problems
  - Arguably that's OK -- they wanted a solution!

Index Tuning "Wizards"

• Both IBM's DB2 and MS SQL Server have automated index advisors
  - Some info in Section 20.6 of the book
• Basic idea:
  - They take a workload of queries
    - Possibly based on logging what's been going on
  - They use the optimizer cost metrics to estimate the cost of the workload over different choices of sets of indexes
  - Enormous # of different choices of sets of indexes:
    - Heuristics to help this go faster
Tuning Queries and Views

- If a query runs slower than expected, check if an index needs to be re-clustered, 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 (bad selectivity estimates)
  - Selections involving arithmetic or string expressions (ditto)
  - Selections involving OR conditions (ditto)
  - Complex subqueries (more on this later)
  - 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.
  - E.g. check via POSTGRES "Explain" command
  - Some systems rewrite for you under the covers (e.g. DB2)
  - Can be confusing and/or helpful!

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
```

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

```
SELECT E.dno, AVG(E.sal) 
FROM Emp E, Dept D 
WHERE E.dno=D.dno 
AND D.mgrname='Joe'
GROUP BY E.dno
```

```
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!

Points to Remember

- 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?

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

```
SELECT E.dno, AVG(E.sal) 
FROM Emp E, Dept D 
WHERE E.dno=D.dno 
AND D.mgrname='Joe'
GROUP BY E.dno
```

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

- Over time, indexes have to be fine-tuned (dropped, created, re-clustered, ...) 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?
  - Null values, arithmetic conditions, string expressions, the use of ORs, nested queries, 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.

Points to Remember (Contd.)

- Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision; only one index on a given relation can be clustered!
  - Order of fields in composite index key can be important.
- Static indexes may have to be periodically re-built.
- Statistics have to be periodically updated.