Announcements

- Homework info:
  - HW #7: Due on Tuesday, May 29 (5 PM)
  - Then just one more HW remains! (“NoSQL”)

- Today’s plan:
  - Today: Physical DB design (e.g., use of indexes)
  - Next up: NoSQL & Big Data *(a la AsterixDB)*
    - Not in the textbook, so...
    - *See (and really do) the NoSQL-related materials linked to the course Wiki page’s syllabus!*
    - In particular, *definitely* read the SQL++ tutorial by Don Chamberlin (“Father of SQL”) that’s attached now.

Overview

- After ER design, schema refinement, and the definition of views, we have the *conceptual* and *external* schemas for our database.

- Next step is to choose *indexes*, make *clustering decisions*, and *refine* the conceptual and external *schemas* (if needed) to meet *performance goals*.

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

- **Primary vs. secondary**: If index search key contains the primary key, this is called the primary index.
  - Unique index: Search key contains a candidate key.
- **Clustered vs. unclustered**: If the order of data records is the same as, or `close to`, the order of stored data records, we have 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 (Reminder)

(Read each page once.) (Read more pages – and repeatedly!)
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 means we must understand and see how a DBMS evaluates its queries. *(Query execution 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).
  - Range query → B+ tree index.
    - Clustering especially useful for range queries, but can also help with **equality queries with duplicate values** (non-key field index).
- **Multi-attribute** search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes matters for range queries.
  - Such indexes can sometimes enable **index-only strategies** for important queries (e.g., aggregates / grouped aggregates).
    - **Note:** 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**

---

**Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke**

11

---

**Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke**

12
Composite Search Keys

- To retrieve Emp records with $age=30$ AND $sal=4000$, an index on $<age,sal>$ would be better than an index only on $age$ or an index only on $sal$.
  - *Note:* Choice of index key is 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! Picture the index…)*

- 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.)

- SELECT $E.dno$, COUNT(*)
  FROM Emp $E$
  GROUP BY $E.dno$

- SELECT $E.dno$, $E.sal$
  FROM Emp $E$
  GROUP BY $E.dno$

- SELECT AVG($E.sal$)
  FROM Emp $E$
  WHERE $E.age=25$ AND $E.sal$ BETWEEN 3000 AND 5000

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke
Some Illustrated Index-Only Plans

**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 key and the 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) fantastic for this!
  - **Hash join (HJ) method:**
    - Indexing not needed (not for the join, anyway).

Note: The index files are each much smaller than the main file!
Example 1

- 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 get 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.

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

Example 2

- 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 consider for a query. ∴ Understand query optimizer!

```
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
```
**Clustering and Joins**

```sql
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:
  ```sql
  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(sid, sname, address, phone, …)
Parts(pid, pname, size, color, listprice, …)
Stock(sid, pid, 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.
    - If sid→sname (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**

<table>
<thead>
<tr>
<th>eno</th>
<th>email</th>
<th>name</th>
<th>addr</th>
<th>salary</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="mailto:joe@aol.com">joe@aol.com</a></td>
<td>Joe</td>
<td>1 Main St.</td>
<td>100000</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td><a href="mailto:sue@gmail.com">sue@gmail.com</a></td>
<td>Sue</td>
<td>10 State St.</td>
<td>125000</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td><a href="mailto:zack@fb.com">zack@fb.com</a></td>
<td>Zack</td>
<td>100 Wall St.</td>
<td>2500000</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>

(Vertical partitioning: \(\times\))

<table>
<thead>
<tr>
<th>eno</th>
<th>email</th>
<th>name</th>
<th>addr</th>
<th>salary</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="mailto:joe@aol.com">joe@aol.com</a></td>
<td>Joe</td>
<td>1 Main St.</td>
<td>100000</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td><a href="mailto:sue@gmail.com">sue@gmail.com</a></td>
<td>Sue</td>
<td>10 State St.</td>
<td>125000</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td><a href="mailto:zack@fb.com">zack@fb.com</a></td>
<td>Zack</td>
<td>100 Wall St.</td>
<td>2500000</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>

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

---

**Horizontal Decompositions**

- Prior def’n. of decomposition: Relation replaced by a set of joinable relations that are *projections*. (That’s the most important and the most common case.)
- 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.)
  - Original relation is the UNION (ALL) of the new ones (i.e., rather than the JOIN of the new ones).
Horizontal Decompositions (Cont’d.)

- 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 a clustered B+ tree index on the $val$ field of Contracts.
- Another approach could be to replace Contracts by two new relations, LargeContracts & SmallContracts, with the same attributes (e.g., CSJDPQV).
  - Performs like index on such queries, but no index overhead.
  - Can build clustered indexes on other attributes, in addition!

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 $val > 10000$ must be written against LargeContracts* for fast execution; users concerned with performance must be aware of this change.
  (*The DBMS isn’t aware of the two tables’ value constraints.*)
Horizontal Partitioning Example

<table>
<thead>
<tr>
<th>eno</th>
<th>email</th>
<th>name</th>
<th>addr</th>
<th>salary</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="mailto:joe@aol.com">joe@aol.com</a></td>
<td>Joe</td>
<td>1 Main St.</td>
<td>100000</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td><a href="mailto:sue@gmail.com">sue@gmail.com</a></td>
<td>Sue</td>
<td>10 State St.</td>
<td>125000</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td><a href="mailto:zack@fb.com">zack@fb.com</a></td>
<td>Zack</td>
<td>100 Wall St.</td>
<td>250000</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>

(Horizontal partitioning: U)

**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 more advanced evaluation features like 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 utter 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 might not 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, we’ll go back and forth between tasks to refine a DB design; decisions 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.
  - Importance of dependency-preservation depends on the dependency to be preserved and the cost of the IC check.
    - Can add a relation for dep-preservation (e.g., the CJP example); or else, can check dependency using a join (e.g., using a trigger).

- 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...?