CPSC 421
Database Management Systems

Lecture 19:
Physical Database Design
Concurrency Control and Recovery

* Some material adapted from R. Ramakrishnan, L. Delcambre, and B. Ludaescher

Agenda

• Physical Database Design
• Transactions (as time allows)
What is “Physical” Design?

- After we:
  - Collect requirements
  - Create ER diagram (conceptual design)
  - Create (a normalized) schema
  - Create view definitions
  - Define queries, etc.

- We (often) need to worry about performance:
  - Choose indexes
  - Decide on what attributes should be clustered
  - Refine “external” schema

Determine the “workload”

- For physical design, we need the anticipated database “workload”
  - The most important queries and how often they occur
  - The most important updates and how often they occur
  - The performance requirements for the queries and updates

  These all depend on the requirements of the application!
Determine the “workload”

• For example:
  – The DB is only updated once a year, but with 1,000’s of queries per day (e.g., product catalog)
  – Alternatively, equal mix of updates and queries (typical)
  – Or, 45% of operations are Q1, 25% are Q2, 5% are Q3, 10% are U1, 15% are U2 (detailed)
  – And, Q1 must run in under 1 second, Q2 under 5 seconds, U1 under 2 seconds (very detailed)

Decisions, decisions

• Which indexes should we create for our workload?
  – Which relations should have indexes?
  – What field(s) should be the search key?
  – Should we build several indexes?

  What are the tradeoffs?

• For each index, what kind should it be?
  – Clustered? Unclustered?
  – Hash? B+ Tree?

• Should we change the external schema?
  – Alternative normalized schemas? (different BCNF decompositions)
  – Should we “denormalize” (i.e., add redundancy …)
  – Horizontal partitioning, materialized views …
Index Selection for Joins

• When considering a join condition:
  – Hash index on inner relation can be good for Index Nested Loops
    • This can be done even if there is a B+ Tree on the key
    • Should be clustered if join column is not the key for inner relation
  – Clustered B+ Tree on join column(s) good for Sort-Merge
    • What is the data skew though on the inner relation?

Example 1

Hash index on $D.dname$
  – Given this, index on $D.dno$ is not needed
  – We “lose” $D.dno$ index after pushing select

Why make Dept the outer relation?
  – If Dept is inner, we would have to save selection result to disk!

```sql
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE D.dname="Toy" AND E.dno = D.dno;
```
Example 1

Hash index on $D.dname$
- Given this, index on $D.dno$ is not needed
- We “lose” $D.dno$ index after pushing select

Hash index on $E.dno$
- allows matching (inner) Emp tuples for each selected (outer) Dept tuple

Example 1

If query included “$E.age = 25$”
- Can retrieve Emp tuples using index on $E.age$ (push select)
- Join with Dept tuples satisfying $D.dname$ selection (push select)
- With $E.age$ index … this query provides much less motivation for adding $E.dno$ index!

What does this depend upon?
- Selectivities!
- In this case, $E.age$ is an equality
Example 2

Emp should be the outer relation

- Suggests that we build a hash index on D.dno … why?

What index should we build on Emp?

- B+ Tree on E.sal could be used
- Or, hash index on E.hobby
- Only one of these is needed … the other can be done “on the fly”
- As a rule of thumb, equality is more selective than ranges

SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE E.sal > 10000 AND
    E.sal < 20000 AND
    E.hobby="stamps" AND
    E.dno = D.dno;

Physical Design and Query Optimization

• Both of these examples indicate that …

• … our choice of indexes is guided by the plan(s) we expect the optimizer to consider

Physical design requires understanding query optimization!
Tuning the External Schema

The external schema should be guided by

– the workload
– and redundancy issues

• “Denormalization”
  – BCNF decomposition removes redundancy by “separating” attributes … requiring joins to “bring them together”
  – Joins are expensive!
  – This is why we might want to undo decompositions

Tuning the External Schema

• We may also want to further decompose relations
  – Makes relations smaller
  – Sometimes called “partitioning”

• “Horizontal” partitions
  – For example
    • Employees partitioned into SalesEmployees and ManagerEmployees
  – The partitions represent a unique “selection” over the original table

Note: Decomposition applies “projections”
Tuning the External Schema

- We may also introduce new keys and “vertically” partition
  - Reduce the number of columns in a table by creating a new key
  - \texttt{Student(sid, name, age, credits, major, \{other attributes\})}
  - Becomes:
    - \texttt{Student(sid, name, age, credits, major, oid),}
    - \texttt{StudentMisc(oid, \{other attributes\})}
  - Can be “recovered” through Joins

Tuning Queries

*If a query runs slower than expected*

- Check if an index needs to be re-build
- Or if statistics are too old

- The DBMS may not be executing the plan you had in mind
  - Weaknesses:
    - Selections involving null values
    - Selections involving arithmetic or string expressions
    - Selections involving OR conditions
    - Lack of evaluation features, like index-only strategies, lack of certain join methods, or poor estimation

- Check the plan that is being used
  - In this case, adjust indexes, or rewrite the query/view
Tuning Queries

• Minimize the use of DISTINCT
  – Don’t need it if duplicates are acceptable
  – Or if answer contains a key
• Minimize use of GROUP BY and HAVING

\[
\begin{align*}
\text{SELECT MIN(E.age)} \\
\text{FROM Emp E} \\
\text{GROUP BY E.dno} \\
\text{HAVING E.dno = 102;}
\end{align*}
\]

\[
\begin{align*}
\text{SELECT MIN(E.age)} \\
\text{FROM Emp E} \\
\text{WHERE E.dno = 102;}
\end{align*}
\]

• Consider DBMS use of index when writing arithmetic expressions
  – \( E\text{.age} = 2 \times D\text{.age} \)
  – Will benefit from index on \( E\text{.age} \), but (might) not from index on \( D\text{.age} \)

Wrap up

• Database design consists of several tasks:
  – Requirements analysis
  – Conceptual design
  – Schema refinement
  – Physical design and tuning
  
  \textit{Usually an iterative process}

• Understanding the application workload and performance requirements is crucial to a “good” design

• Physical design and tuning requires understanding DBMS query optimization approaches
On to Concurrency Control and Recovery …

Basic Database Architecture

DBMS
- Plan Executor
- Operator Evaluator
- File and Access Methods
- Buffer Manager
- Disk Space Manager
- Recovery Manager
- System Catalog
- Index Files
- Data Files
- Transaction Manager
- Lock Manager
- Query Evaluation Engine
- SQL Commands
- Web Forms
- Application Front Ends
- SQL Interface
Concurrency Control and Recovery

- Transactions
  - A way to define a single “all or nothing” set of SQL actions
  - Based on a set of properties (the “ACID” properties)
  - Enable concurrency
  - The basis for crash recovery

Transactions

- A “transaction” is a set of SQL statements (that modify a database) chosen by a user

Transfer $100 from one account to another (MySQL):

START TRANSACTION;
SELECT @A1 := balance FROM Account WHERE acctno=500;
UPDATE Account SET balance := @A1+100 WHERE acctno=500;
SELECT @A2 := balance FROM Account WHERE acctno=501;
UPDATE Account SET balance := @A2-100 WHERE acctno=501;
COMMIT;

- We’re overlooking a lot of details here!!!
Transactions

• A “transaction” is a set of SQL statements (that modify a database) chosen by a user

Transfer $100 from one account to another (generic):

BEGIN TRANSACTION
  READ balance from account 500
  ADD $100 to balance of account 500
  WRITE new balance to account 500
  READ balance in account 501
  VERIFY balance to see if it contains at least $100
    ABORT if balance is less than $100
  SUBTRACT $100 from balance of account 501
  WRITE new balance to account 501
COMMIT TRANSACTION

Transactions

• User (application developer) must
  – Begin transaction
  – Read, write, and modify statements intermixed with other programming language statements (e.g., verify)

• Plus either
  – Commit to indicate successful completion or
  – Abort to indicate that the transaction should be “rolled back” … i.e., erase previous steps of transaction

To ensure database stays in a consistent/correct state

  – The DBMS and programmer must guarantee 4 properties of transactions
  – These are called the “ACID” properties
ACID Properties

Atomicity

Consistency

Isolation

Durability

ACID Properties

Atomicity

A transaction happens in its entirety or not at all

• What if the OS crashed half-way through the transaction ... after $100 was removed from the first account? (good for the bank!)

• The recovery manager of the DBMS must assure that the $100 is deposited back to the first account

• Often called “roll back”
ACID Properties

**Consistency**

*If the DB starts in a consistent state, the transaction will transform it into a consistent state*

- The notion of “consistency” is specific to the application constraints (defined by the user)
- Thus the *programmer* must ensure transactions are consistent
  - The DBMS ensures the transaction is atomic
- E.g., what if the transaction only deposited $100?
  - Probably not consistent according to our example application

---

ACID Properties

**Isolation**

*Each transaction is “isolated” from other transactions … DB state is as if each transaction executed by itself*

- What if another transaction computed the total bank balance after $100 was removed from the first account?
- The *concurrency control* subsystem must
  - ensure that all transactions run in isolation (i.e., don’t mess up other transactions)
  - unless the programmer chooses a less strict level of isolation
  - similar to concurrency control in operating systems
ACID Properties

**Durability**

*If a transaction commits, its changes to the DB state persist (changes are permanent)*

- What if after the commit the OS crashed before the deposit was written to disk?
- The *recovery manager* must assure that the deposit was at least logged (e.g., to make the DB consistent)