CMU SCS

Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 - DB Applications

C. Faloutsos – A. Pavlo

Lecture#18: Physical Database Design

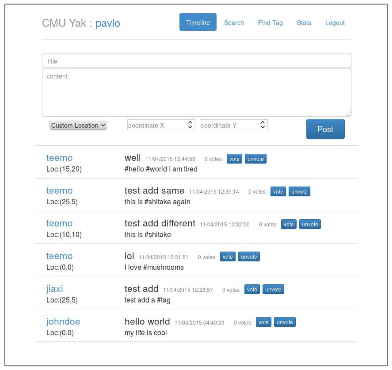
CMU SCS

Administrivia

- HW6 is due **right now**.
- HW7 is out today
  – Phase 1: **Wed Nov 11th**
  – Phase 2: **Mon Nov 30th**
- Recitations (WEH 5302):
  – Tue Nov 10th
  – Tue Nov 17th

CMU SCS

HW7: CMU "YikYak"



- PHP Web Application
- Postgres Database
- **Phase 1:** Design Spec
- **Phase 2:** Implementation

CMU SCS

Last Class

- Decomposition
  – Lossless
  – Dependency Preserving
- Normal Forms
  – 3NF
  – BCNF

Today's Class

- Introduction
- Index Selection
- Denormalization
- Decomposition
- Partitioning
- Advanced Topics

Introduction

- After ER design, schema refinement, and the view definitions, we have a conceptual and external schema for our database.
- The next step is to create the physical design of the database.

Physical Database Design

- Physical design is tightly linked to query optimization
  - Query optimization is usually a "top down" concept.
  - But in this lecture we'll discuss this from the "bottom up"

Physical Database Design

- It is important to understand the application's workload:
  - What kind of queries/updates does it execute?
  - How fast is the database growing?
  - What is the desired performance metric?

Understanding Queries

- 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?

Understanding Updates

- For each update in the workload:
  - Which attributes are involved in predicates?
  - How selective are these conditions likely to be?
  - What types of update operations and what attributes do they affect?
  - How often are records inserted/updated/deleted?

Consequences

- Changing a database's design does not magically make every query run faster.
  - May require you to modify your queries and/or application logic.
- APIs hide implementation details and can help prevent upstream apps from breaking when things change.

General DBA Advice

- Modifying the physical design of a database is expensive.
- DBA's usually do this when the application demand is low
  - Typically Sunday mornings.
  - May have to do it whenever the application changes.

Today's Class

- Introduction
- Index Selection
- Denormalization
- Decomposition
- Partitioning
- Advanced Topics

Index Selection

- Which relations should have indexes?
- What attributes(s) or expressions should be the search key?
- What order to use for attributes in index?
- How many indexes should we create?
- For each index, what kind of an index should it be?

Example #1

```
CREATE TABLE users (
    userID INT,
    servID VARCHAR,
    data VARCHAR,
    updated DATETIME,
    PRIMARY KEY (userId)
);
```

```
CREATE TABLE locations (
    locationID INT,
    servID VARCHAR,
    coordX FLOAT,
    coordY FLOAT
);
```

```
SELECT U.*, L.coordX, L.coordY
  FROM users AS U INNER JOIN locations AS L
    ON (U.servID = L.servID)
 WHERE U.userID > $1
   AND EXTRACT(dow FROM U.updated) = 2;
```


Example #1: Join Clause


```
CREATE TABLE users (
    userID INT,
    servID VARCHAR,
    data VARCHAR,
    updated DATETIME,
    PRIMARY KEY (userId)
);
CREATE TABLE locations (
    locationID INT,
    servID VARCHAR,
    coordX FLOAT,
    coordY FLOAT
);
```


- Examine the attributes in the join clause
  - Is there an index?
  - What is the cardinality of the attributes?


```
SELECT U.*, L.coordX, L.coordY
  FROM users AS U INNER JOIN locations AS L
    ON (U.servID = L.servID)
 WHERE U.userID > $1
   AND EXTRACT(dow FROM U.updated) = 2;
```

Example #1: Where Clause

```
CREATE TABLE users (
    userID INT,
    servID VARCHAR,
    data VARCHAR,
    updated DATETIME,
    PRIMARY KEY (userId)
);
```

```
CREATE TABLE locations (
    locationID INT,
    servID VARCHAR,
    coordX FLOAT,
    coordY FLOAT
);
```

- Examine the attributes in the where clause
  - Is there an index?
  - How are they be accessed?

```
SELECT U.*, L.coordX, L.coordY
  FROM users AS U INNER JOIN locations AS L
    ON (U.servID = L.servID)
 WHERE U.userID > $1
   AND EXTRACT(dow FROM U.updated) = 2;
```


Example #1: Output Clause


```
CREATE TABLE users (
    userID INT,
    servID VARCHAR,
    data VARCHAR,
    updated DATETIME,
    PRIMARY KEY (userId)
);
```

```
CREATE TABLE locations (
    locationID INT,
    servID VARCHAR,
    coordX FLOAT,
    coordY FLOAT
);
```


- Examine the query's output clause
  - What attributes from what tables are needed?


```
SELECT U.*, L.coordX, L.coordY
  FROM users AS U INNER JOIN locations AS L
    ON (U.servID = L.servID)
 WHERE U.userID > $1
   AND EXTRACT(dow FROM U.updated) = 2;
```


Example #1: Summary


```
CREATE TABLE users (
    userID INT,
    servID VARCHAR,
    data VARCHAR,
    updated DATETIME,
    PRIMARY KEY (userId)
);
```

```
CREATE TABLE locations (
    locationID INT,
    servID VARCHAR,
    coordX FLOAT,
    coordY FLOAT
);
```


- **Join**: `U.servID`, `L.servID`
- **Where**: `U.userID`, `U.updated`
- **Output**: `U.userID, U.servID, U.data, U.updated`, `L.coordX`, `L.coordY`


```
SELECT U.*, L.coordX, L.coordY
  FROM users AS U INNER JOIN locations AS L
    ON (U.servID = L.servID)
 WHERE U.userID > $1
   AND EXTRACT(dow FROM U.updated) = 2;
```


Index Selection


```
CREATE TABLE users (
    userID INT,
    servID VARCHAR,
    data VARCHAR,
    updated DATETIME,
    PRIMARY KEY (userId)
);
```

```
CREATE TABLE locations (
    locationID INT,
    servID VARCHAR,
    coordX FLOAT,
    coordY FLOAT
);
```


- We already have an index on `U.userID`.
  - Why?
- What if we created separate indexes for `U.servID` and `L.servID`?

```
CREATE INDEX idx_u_servID ON users (servID);
```

```
CREATE INDEX idx_l_servID ON locations (servID);
```

Index Selection (2)

```
CREATE TABLE users (
    userID INT,
    servID VARCHAR,
    data VARCHAR,
    updated DATETIME,
    PRIMARY KEY (userId)
);
```

```
CREATE TABLE locations (
    locationID INT,
    servID VARCHAR,
    coordX FLOAT,
    coordY FLOAT
);
```

- We still have to look up **U.updated**.
- What if we created another index?
- This doesn't help our query. Why?

```
SELECT U.*, L.coordX, L.coordY
  FROM users AS U INNER JOIN locations AS L
    ON (U.servID = L.servID)
 WHERE U.userID > $1
   AND EXTRACT(dow FROM U.updated) = 2;
```

```
CREATE INDEX idx_u_updated ON users (
                    EXTRACT(dow FROM updated));
```


Index Selection (3)


```
CREATE TABLE users (
    userID INT,
    servID VARCHAR,
    data VARCHAR,
    updated DATETIME,
    PRIMARY KEY (userId)
);
```

```
CREATE TABLE locations (
    locationID INT,
    servID VARCHAR,
    coordX FLOAT,
    coordY FLOAT
);
```


- The query outputs **L.coordX** and **L.coordX**.
- This means that we have to fetch the location record.
- We can create a covering index.

```
CREATE INDEX idx_u_servID ON users (servID);
```

```
CREATE INDEX idx_l_servID ON locations (
                    servID, coordX, coordY);
```

```
CREATE INDEX idx_u_updated ON users (
```

```
CREATE INDEX idx_l_servID ON locations (servID)
                    INCLUDE (coordX, coordY);
```

Only MSSQL

Index Selection (4)


```
CREATE TABLE users (
    userID INT,
    servID VARCHAR,
    data VARCHAR,
    updated DATETIME,
    PRIMARY KEY (userId)
);
```

```
CREATE TABLE locations (
    locationID INT,
    servID VARCHAR,
    coordX FLOAT,
    coordY FLOAT
);
```


- Can we do any better?
- Is the index **U.servID** necessary?
- Create a partial index

Repeat for the other six days of the week!

```
CREATE INDEX idx_u_servID ON users (servID)
            WHERE EXTRACT(dow FROM updated) = 2;
```

```
CREATE INDEX idx_l_servID ON locations (
                    servID, coordX, coordY);
```

```
CREATE INDEX idx_u_updated ON users (
                    EXTRACT(dow FROM updated));
```


Index Selection (5)


```
CREATE TABLE users (
    userID INT,
    servID VARCHAR,
    data VARCHAR,
    updated DATETIME,
    PRIMARY KEY (userId)
);
```

```
CREATE TABLE locations (
    locationID INT,
    servID VARCHAR,
    coordX FLOAT,
    coordY FLOAT
);
```


- Should we make the index on users a covering index?
  - What if **U.data** is large?
  - Should **userID** come before **servID**?
  - Do we still need the primary key index?

```
CREATE INDEX idx_u_everything ON users
                        (userID, servID)
        WHERE EXTRACT(dow FROM updated) = 2;
```

Other Index Decisions

- What type of index to use?
  - B+Tree, Hash table, Bitmap, R-Tree, Full Text

Today's Class

- Introduction
- Index Selection
- Denormalization
- Decomposition
- Partitioning
- Advanced Topics

Denormalization

- Joins can be expensive, so it might be better to denormalize two tables back into one.
- This is goes against all of the BCNF goodness that we talked about it.
  - But we have bills to pay, so this is an example where reality conflicts with the theory…

Game Example #1

```
CREATE TABLE players (
   playerID INT PRIMARY KEY,
   ⋮
);
```

```
CREATE TABLE prefs (
   playerID INT PRIMARY KEY
                REFERENCES player (playerID),
   data VARBINARY
);
```

Game Example #1

- Get player preferences (1:1)

```
SELECT P1.*, P2.*
  FROM players AS P1 INNER JOIN prefs AS P2
    ON P1.playerID = P2.playerID
 WHERE P1.playerID = $1
```


Game Example #1

```
CREATE TABLE players (
   playerID INT PRIMARY KEY,
   data VARBINARY,
   ⋮
);
```

Denormalize into parent table

```
CREATE TABLE prefs (
   data VARBINARY
);
```

```
SELECT P1.* FROM players AS P1
 WHERE P1.playerID = $1
```


Denormalization (1:*n*)

- It's harder to denormalize tables with a 1:*n* relationship.
  - Why?

- Example: There are multiple "game" instances in our application that players participate in. We need to keep track of each player's score per game.

Game Example #2

```
CREATE TABLE players (
   playerID INT PRIMARY KEY,
   ⋮
);
```

```
CREATE TABLE games (
   gameID INT PRIMARY KEY,
   ⋮
);
```

```
CREATE TABLE scores (
   gameID INT REFERENCES games (gameID),
   playerID INT REFERENCES players (playerID),
   score INT,
   PRIMARY KEY (gameID, playerID)
);
```

Game Example #2

- Get the list of playerIDs for a particular game sorted by their score.

```
SELECT S.gameID, S.score, G.*, P.*
  FROM players AS P, games AS G, scores AS S
 WHERE G.gameID = $1
   AND G.gameID = S.gameID
   AND S.playerID = P.playerID
 ORDER BY S.score DESC
```


Game Example #2

```
CREATE TABLE players (
   playerID INT PRIMARY KEY,
   ⋮
);
```

```
CREATE TABLE games (
   gameID INT PRIMARY KEY,
   playerID1 INT REFERENCES players (playerID),
   score1 INT,
   playerID2 INT REFERENCES players (playerID),
   score2 INT,
   playerID3 INT REFERENCES players (playerID),
   score3 INT,
   ⋮
);
```


Arrays

- Denormalize 1:*n* relationships by storing multiple values in a single attribute.
  - Oracle: **VARRAY**
  - Postgres: **ARRAY**
  - DB2/MSSQL/SQLite: UDTs
  - MySQL: Fake it with **VARBINARY**
- Requires you to modify your application to manage these arrays.
  - DBMS will not enforce foreign key constraints.

Game Example #2

```
CREATE TABLE players (
   playerID INT PRIMARY KEY,
   ⋮
);
```

```
CREATE TABLE games (
   gameID INT PRIMARY KEY,
   playerIDs INT[],
   scores INT[],
   ⋮
);
```

```
SELECT P.*, G.*
  G.scores[idx(G.playerIDs, P.playerID)] AS score
  FROM players AS P JOIN games AS G
    ON P.playerID = ANY(G.playerIDs)
 WHERE G.gameID = $1
 ORDER BY score DESC;
```

Not a standard SQL function
See: https://wiki.postgresql.org/wiki/Array_Index

Game Example #2

```
CREATE TABLE players (
   playerID INT PRIMARY KEY,
   ⋮
);
```

```
CREATE TABLE games (
   gameID INT PRIMARY KEY,
   playerScores INT[][], -- (playerId, score)
   ⋮
);
```

No easy way to query this in pure SQL…

Today's Class

- Introduction
- Index Selection
- Denormalization
- Decomposition
- Partitioning
- Advanced Topics

Decomposition

- Split physical tables up to reduce the overhead of reading data during query execution.
  - **Vertical**: Break off attributes from a table.
  - **Horizontal**: Split data from a single table into multiple tables.
- This is an application of normalization.

Wikipedia Example

```
CREATE TABLE pages (
   pageID INT PRIMARY KEY,
   title VARCHAR UNIQUE,
   latest INT REFERENCES revisions (revID)
   updated DATETIME
);
```

```
CREATE TABLE revisions (
   revID INT PRIMARY KEY,
   pageID INT REFERENCES pages (pageID),
   content TEXT,
   updated DATETIME
);
```

Wikipedia Example

- Load latest revision for page

```
SELECT P.*, R.*
  FROM pages AS P INNER JOIN revisions AS R
    ON P.latest = R.revID
 WHERE P.pageID = $1
```

- Get all revision history for page

```
SELECT R.revID, R.updated, ...
  FROM revisions AS R
 WHERE R.pageID = $1
```


Wikipedia Example

```
CREATE TABLE pages (
   pageID INT PRIMARY KEY,
   title VARCHAR UNIQUE,
   latest INT REFERENCES revisions (revID),
   updated DATETIME
);
```

```
CREATE TABLE rev
   revID INT PRI
   pageID INT REFER      pages (pageID),
   content TEXT,
   updated DATETIME
);
```

Avg. Size of Wikipedia Revision: ~16KB

Vertical Decomposition

- Split out large attributes into a separate table (aka "normalize").
- Trade-offs:
  - Some queries will have to perform a join to get the data that they need.
  - But other queries will read less data.

Vertical Decomposition

```
CREATE TABLE pages (
   pageID INT PRIMARY KEY,
   title VARCHAR UNIQUE,
   latest INT REFERENCES revisions (revID),
   updated DATETIME
);
```

```
CREATE TABLE revisions (
   content TEXT
);
```

```
SELECT P.*, R.*, RD.*
  FROM pages AS P, revisions AS R,
       revData AS RD
 WHERE P.pageID = $1
   AND P.latest = R.revID
   AND R.revID = RD.revID
```

Horizontal Decomposition

- Replace a single table with multiple tables where tuples are assigned to a table based on some condition.
- Can mask the changes to the application using views and triggers.

Horizontal Decomposition

```
CREATE TABLE revisions (
   revID INT PRIMARY KEY,
   pageID INT REFERENCES pages (pageID),
   updated DATETIME
);
```

All new revisions are first added to this table.

```
SELECT R.revID, R.updated, ...
  FROM revisions AS R
 WHERE R.pageID = $1
```

Then a rev… to this tab… longer the latest.

```
CREATE VIEW revData AS
    (SELECT * FROM revDataNew)
  UNION
    (SELECT * FROM revDataOld)
```


Today's Class

- Introduction
- Index Selection
- Denormalization
- Decomposition
- Partitioning
- Advanced Topics

Partitioning

- Split single logical table into disjoint physical segments that are stored/managed separately.
- Ideally partitioning is transparent to the application.
  - The application accesses logical tables and doesn't care how things are stored.
  - Not always true.

Vertical Partitioning

- Store a table's attributes in a separate location (e.g., file, disk volume).
- Have to store tuple information to reconstruct the original record.

Partition #1 | Partition #2

Tuple#1	revID	pageID	updated	content Tuple#1
Tuple#2	revID	pageID	updated	content Tuple#2
Tuple#3	revID	pageID	updated	content Tuple#3
Tuple#4	revID	pageID	updated	content Tuple#4

Horizontal Partitioning

- Divide the tuples of a table up into disjoint segments based on some partitioning key.
  - Hash Partitioning
  - Range Partitioning
  - Predicate Partitioning
- We will cover this more in depth when we talk about distributed databases.

Horizontal Partitioning (Postgres)

```
CREATE TABLE revisions (
   revID INT PRIMARY KEY,
   pageID INT REFERENCES pages (pageID),
   updated DATETIME
);
```

```
CREATE TABLE revData (
   revID INT REFERENCES revisions (revID),
   content TEXT,
   isLatest BOOLEAN DEFAULT true
);
```

Still need triggers to move data between partitions on update.

```
CREATE TABLE revDataNew (
  CHECK (isLatest = true)
) INHERITS revData;
```

```
CREATE TABLE revDataOld (
  CHECK (isLatest = false)
) INHERITS revData;
```


Today's Class

- Introduction
- Index Selection
- Denormalization
- Decomposition
- Partitioning
- Advanced Topics

Caching

- Queries for content that does not change often slow down the database.
- Use external cache to store objects.
  - Memcached, Facebook Tao
  - Application has to maintain consistency.

Auto-Tuning

- Vendors include tools that can help with the physical design process:
  - IBM DB2 Advisor
  - Microsoft AutoAdmin
  - Oracle SQL Tuning Advisor
  - Random MySQL/Postgres tools
- Still a very manual process.
- ***We are working on something better…***

Next Three Weeks

- Database System Internals
  - Concurrency Control
  - Logging & Recovery
  - Distributed DBMSs
  - Column Store DBMSs