Relational Databases

Firefox browsing history is stored as a database on disk

Subversion stores your source code in a database

Embedded database is an alternative to saving and loading a file format

Instead of saving Java heap objects to a file with a textual format like XML, you can store the data in a database instead

Benefits of Using a Database

Persistence
- Databases are persistent by default – updates to the database are immediately stored on disk
- Usually robust to program crashes and hardware reboots
- Contrast with objects in the Java heap, which disappear on a crash

Query performance
- Databases build and maintain indexes to answer complex queries quickly, e.g. “find books written by Stephen King in 2004”

Concurrency
- Databases provide an effective synchronization mechanism, transactions, that allows safe concurrent updates to a pile of relational data

Relational Databases

A relational database is a set of named tables
- A table has a fixed set of named columns (aka fields or attributes) and a varying set of unnamed rows (aka records or tuples)

<table>
<thead>
<tr>
<th>Person</th>
<th>3 columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name</td>
<td>LastName</td>
</tr>
<tr>
<td>Daniel</td>
<td>Jackson</td>
</tr>
<tr>
<td>Rob</td>
<td>Miller</td>
</tr>
</tbody>
</table>

- Each cell in the table stores a value of a primitive data type
  - e.g. string, integer, date, time
  - object references are represented by integer IDs

A table represents a relation
- In general, a mathematical relation is a set of n-tuples (a binary relation is a special case, which is a set of pairs)
**Example**

An object model we want to store in a database

- **Song** relation
  - `songId` column is the object's address in memory, and other columns are fields of the object
  - The `id` column is usually automatically generated by the database system so that all songs have a unique ID
  - Analogy: Java's `new` operator automatically generates a fresh address

**Pure Relational View**

One table per binary relation

<table>
<thead>
<tr>
<th>songId</th>
<th>songName</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mr. Brightside</td>
<td>4:17</td>
</tr>
<tr>
<td>2</td>
<td>Somebody Told Me</td>
<td>5:57</td>
</tr>
<tr>
<td>3</td>
<td>Girlfriend</td>
<td>3:42</td>
</tr>
</tbody>
</table>

**lyric** relation

<table>
<thead>
<tr>
<th>songId</th>
<th>lyric</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Hey, hey, you, you, I don't like your girlfriend...</td>
</tr>
</tbody>
</table>

**albumTracks** relation

<table>
<thead>
<tr>
<th>albumId</th>
<th>albumName</th>
<th>track1</th>
<th>track2</th>
<th>track3</th>
<th>track4</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Hot Fuss</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>102</td>
<td>The Best Damn Thing</td>
<td>101</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**Class/Relation View**

Often all the exactly-one (!) relations for a class are combined into a single table

<table>
<thead>
<tr>
<th>Song relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>songId</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

**Bad Designs**

Relations with other multiplicities (+, *, ?) generally should not be combined

- Otherwise, ? relation would force columns to have empty cells
- Multiplicity + and * would force columns to become arrays
- Sometimes this is done anyway for performance reasons, just like nulls are sometimes useful for Java field values

<table>
<thead>
<tr>
<th>albumName</th>
<th>albumTracks</th>
<th>albumId</th>
<th>albumName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hot Fuss</td>
<td>101</td>
<td>1</td>
<td>Hot Fuss</td>
</tr>
<tr>
<td>The Best Damn Thing</td>
<td>102</td>
<td>2</td>
<td>The Best Damn Thing</td>
</tr>
</tbody>
</table>

© Robert Miller 2009
Querying a Relational Database

SQL (“Structured Query Language”)

- SQL is a standard language for querying (and mutating) a relational database
- Most database systems support some flavor of SQL
- SQL’s SELECT statement offers a compact language for retrieving subsets of relational data
  - Find all songs longer than 5 minutes
    ```sql
    SELECT songName FROM Song WHERE duration > 300
    ```
- If you know nothing else about SQL, you should know about SELECT
  - Note that SQL is case-insensitive, so SELECT and select are the same, as are songName and songname

Relational Algebra

SELECT is based on a few simple operations that can be performed on relations

- Each operation takes one or more relations and produces a relation
- PROJECT filters the columns
- SELECT filters the rows
- PRODUCT adjoining columns from two relations
- RENAME renames columns
- A relation is a set of rows, so the usual set operations also apply
- UNION
- INTERSECTION
- DIFFERENCE

Projection

Projection keeps a set of named columns and discards the rest

```sql
SELECT songId, duration
FROM   Song
```

<table>
<thead>
<tr>
<th>songId</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4:17</td>
</tr>
<tr>
<td>2</td>
<td>4:57</td>
</tr>
<tr>
<td>3</td>
<td>5:57</td>
</tr>
</tbody>
</table>

Selection

Selection keeps the subset of rows that match a predicate and discards the rest

```sql
SELECT * 
FROM   Song
WHERE duration > 300
```

<table>
<thead>
<tr>
<th>songId</th>
<th>songName</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Somebody Told Me</td>
<td>5:57</td>
</tr>
</tbody>
</table>

- Like filtering on the rows
**Product**

**Cartesian product**
- The Cartesian product of two relations R1 and R2 is the result of concatenating each row in R1 with all rows in R2.

```sql
SELECT * FROM Song, Album
```

---

**Joins**

**A join is a special case of Cartesian product**
- When the two relations share a column, we only want to concatenate rows that have the same value for that column.

```sql
SELECT * FROM Song, AlbumTracks WHERE Song.songId = AlbumTracks.songId
```

---

**Question**

**How do I get a list of songName, albumName pairs?**

```sql
SELECT songName, albumName FROM Song, AlbumTracks, Album WHERE Song.songId = AlbumTracks.songId AND AlbumTracks.albumId = Album.albumId
```

---

**Other Set Operations**

**Union, intersection, difference of relations**
- Find songs longer than 5 minutes or containing “midnight” in the lyric

```sql
SELECT songId FROM Song WHERE duration > 300 UNION SELECT songId FROM Lyrics WHERE lyric LIKE '%midnight%'
```

- Find songs longer than 5 minutes for which we have the lyrics

```sql
SELECT songId FROM Song WHERE duration > 300 INTERSECT SELECT songId FROM Lyrics
```

- Find albums that don’t have any tracks

```sql
SELECT albumId FROM Album EXCEPT SELECT albumId FROM AlbumTracks
```

- These operations are rarely used in practice, because select predicates can usually do the job, and database systems are good at optimizing SELECT.
Aggregate Functions

Accumulating a column of data into a single value

- How long is the album Thriller?

SELECT SUM(duration)
FROM Song, Album, AlbumTracks
WHERE Song.songId = AlbumTracks.songId
AND Album.albumId = AlbumTracks.albumId
AND Album.albumName = "Thriller"

Other aggregate functions

- AVG
- COUNT
- MAX
- MIN

Grouping

GROUP BY computes aggregate functions on subsets of the tuples

- How long is each album?

SELECT albumName, SUM(duration)
FROM Song, Album, AlbumTracks
WHERE Song.songId = AlbumTracks.songId
AND Album.albumId = AlbumTracks.albumId
GROUP BY albumName

<table>
<thead>
<tr>
<th>albumName</th>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hot Fuss</td>
<td>10:14</td>
</tr>
<tr>
<td>The Best Damn Thing</td>
<td>3:47</td>
</tr>
</tbody>
</table>

Exercise

Write SELECT statements for the following queries

- Find the name of the album with the song named “Girlfriend”

- Find names of albums for which we have lyrics (for at least one song)

- List all albums, showing album name and number of songs

<table>
<thead>
<tr>
<th>songId</th>
<th>songName</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mr. Brightside</td>
<td>4:17</td>
</tr>
<tr>
<td>2</td>
<td>Somebody Told Me</td>
<td>5:57</td>
</tr>
<tr>
<td>3</td>
<td>Girlfriend</td>
<td>3:24</td>
</tr>
</tbody>
</table>

Mutating the Database

Insert a row

INSERT INTO Song
VALUES (4, "Thriller", 6:02)

Update rows

UPDATE Song
SET songName = "Smile Like You Mean It", duration = 4:57
WHERE songId = 1

Delete rows

DELETE FROM Song
WHERE songName = "Girlfriend"
Concurrency in Databases

**Transactions allow concurrent database modifications**
- A transaction is a block of SQL statements that need to execute together.

**Transactions implement ACID semantics**
- Atomicity – either the full effects of a transaction are recorded or no trace of it will be found.
- Consistency – a transaction is recorded only if it preserves invariants.
  - e.g., every AlbumTrack row must contain an albumId that exists in Album and a songId that exists in Song.
- Isolation – if two transactions operate on the same data, the outcome will always be the same as executing them sequentially one after the other.
- Durability – if the transaction completes, its effects will never be lost.

Transaction Example

**Transfer money between bank accounts**

```sql
BEGIN TRANSACTION
SELECT balance FROM Account WHERE accountId = 1
and put it in local variable balance1
SELECT balance FROM Account WHERE accountId = 2
and call it balance2
balance1 -= 100
balance2 += 100
UPDATE Account SET balance=
balance1 WHERE accountId = 1
UPDATE Account SET balance=balance2 WHERE accountId = 2
COMMIT
```

Transactions vs. Locks

**Transaction is tentative until successful commit**
- COMMIT fails if a simultaneous transaction changed the same rows and managed to commit first.
- If commit fails, the transaction is rolled back – i.e., it has no effect on the database.
- Your program can retry the transaction if the commit failed.

**Database handles low-level concurrency mechanisms**
- e.g., it may lock the rows touched, or detect conflicts at commit time.

**Transactions are widely considered easier to program**
- locking discipline and granularity (database, table, row) is managed by the database implementer.
- programmer just has to think about which statements need to execute in isolation, without acquiring or releasing locks.
- active research on transactional memory is trying to bring the notion of transactions to the shared memory paradigm (like Java objects).

Summary

**Relations as database tables**
- Relational database is a relation-centric implementation of an object model.

**Normal form**
- All rows are unique, no entries can be null.

**Relational algebra for querying**
- Project, select, and join operators combine relations.
- SQL `select` statement uses all three operators.

**Transactions support concurrency**
- Widely considered easier than locks.