Relational Databases

Sam Madden

Key ideas:

Declarative programming

Transactions

Database

Structured data collection

Records

Relationships

Database management system (DBMS)

Why? 1) Widely used

2) Several "big ideas"

Record oriented "data model"

Explicit Model of Data

Declarative language

Consistent

Atomic & Isolated

Durable

What is a database?

- persistent collection of structured data
    - typically organized as "records"
    - and relationships between records

A Database management system (DBMS) is a piece of software to access and manipulate a database

Why should you care:

- Databases are ubiquitous
    - Almost all websites use them
    - Amazon, Google, your bank, etc
    - Many organizations use them internally (e.g., UCB payroll/account/etc.)

- Databases provide a convenient way to encapsulate an application's state as a collection of records
    - Often much easier to think of state as records than files (closer to representation used in most programs)

Explicit model of data provides several attractive properties

Can look at data and see names and types of fields

Can share data between applications easily

Can evolve representation of data over time

Enforces that data maintains certain consistency properties

- High level "declarative" language
    - Say what I want, not how to do it
    - Query optimizer that systematically determines how to execute a query efficiently

- Allows concurrent access from multiple users while ensuring correct behavior ("Atomicity" , "Isolation")

- Updates are stored persistently on disk; strong guarantees in the face of program crashes, etc. ("Durability")

Zoo
- admin interface
  - edit
  - add animal
- public
  - pictures + maps
- zookeeper
  - feeding

1K animals, 5K pages, 10 admins, 200 keepers

ZooFS: store each page in a text file
ZooFS Ops:
- move each snake to a new bldg
  - custom code, consistency issues
- multiple simultaneous admins
  - serial equivalence
- system crashes
  - pages in uncertain state
- hungriest animal
  - custom code, slow

suppose i am creating a web site that stores information about a zoo.
has :
- admin interface that allows me to add new animals, edit animals
- public interface that allows me to look at pictures and maps
- zookeeper interface that allows keepers to find which animals need to be fed

why not just use a file system? what does a database give the developer?

1,000 animals, 5,000 pages, 10 admins, 200 zookeepers, 10,000 hits per day
why not just create a separate set of pages for each animal, store it in FS
(one page for zookeepers, one page for public)

Operations

- suppose move all the snakes to a new building
  - database => queries

- suppose multiple admins try to edit the same page at the same time
  - need some kind of locking
  - database => ("concurrency control; serial equivalence")

- suppose the system crashes mid-update
  - pages might be in uncertain states
  - database provides
    - transactions + recovery
    - groups of actions that happen atomically -- "all or nothing"

- suppose i want to find the animal that was fed the longest ago
  - have to write a complex program
  - could be very slow if it has to read and search all of the pages

- suppose i to add a new field, or share with someone else

Databases address all of these issues.

Modeling

Features to capture

How to (logically) represent data

<u>Features</u>:

Animals: name, age species, cage

Cages: feedtime, bldgs

Keepers: ...

Data Model: logical structures used for data

Relational (tables):

animals

name	age	species	cageno
stoica	0.5	shrew	1
sam	3	salama	2
sally	1	student	1

cages

no	feedtime	bdlg
1	1:30	1
2	2:30	2

schema: tables, fields, names, types

keeps

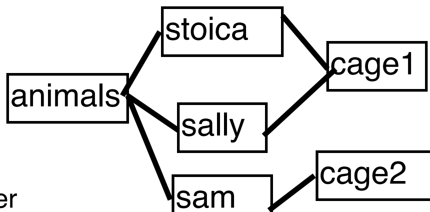
keeper	cage
1	1
1	2
2	1

keepers

keeper	name
1	jenny
2	joe

cage 1
stoica
shrew
.5
sam
salamander
1

cage 2...



stoica isa shrew
stioica livesin cage1
sam isa salamander...

Lets look at how data might be structured in database

<u>What features of our zoo do we want to capture?</u>

each animal has a name, age, species, and is in a cage
each cage gets fed at a particular time, is in a particular building
each cage is kept by many keepers
each keeper keeps many cages
each animal is in one cage, each cage has many animals
“data model” --> “schema”

Relational data model -- tables that represent entities and their properties

Translates into tables by taking all of the one-to-one relations and putting them in table named for object.

This tabular approach is called the **<u>relational</u>** model.

Why relational?

because each record is a relation between fields (“keys” capture relations)

For many to one relations, need to store a reference to other table

Many to many relations require a mapping table

<u>Alternatives</u>

hierarchy, network, triplets

<u>Why might I prefer one representation over the other? Are they equivalent?</u>

Think about writing a program that manipulates these structures

Think about expressing certain complex relationships in some of these models?

Relational Model

Many possible representations of a given data set

name	age	species	cageno	feedtime	bldg
stoica	13	shrew	1	1:30	VLSB
sam	3	salam	2	2:30	SODA
sally	1	student	1	1:30	VLSB

"Normalization" -- avoid potential inconsistencies

Accessing a database

"names of shrews"

```
for each row r in animals
    if r.species = 'shrew'
        output r.name
```

"selection query"

```
SELECT r.name FROM animals
WHERE r.species = 'shrew'
```

1

caged in bldg 2

```
for each row r1 in animals
    for each row r2 in cages
      if r1.bldg = r2.no and r2.bldg = VLSB
        output r1
```

join operator (join)

SQL:

```
SELECT r FROM animals AS r1, cages AS r2
WHERE r1.bldg = r2.no AND r2.bldg = VLSB
```

2

avg bear age

```
SELECT AVG(age)  FROM animals
WHERE type = 'bear'
```

INSERT, DELETE, UPDATE

3

Also that there are many possible relations for a given set of data
(example with joined column)

Rules for choosing the best set of relations for a given data set
"schema normalization"

Also that the logical model of tables doesn't say how they are physically arranged on disk
->Physical data model -- will come back to this

For now, use a physical representation similar to the logical representation -- e.g., rows in a file.

what kind of operations might i want to perform on a relation?

SQL -- structured query language; **show slide**

1) find the names of animals that are shrews.

2) find the animals in a cage in VLSB. (you guys) -- "join"

3) find the average age of the bears.

add a a new snake named bill
INSERT
delete barney
DELETE
move the snakes to a new cage
UPDATE

Show additional queries

Under the covers -- Declarative queries:
multiple equivalent procedural plans

sorted animals on type => binary search

+ search performance
- update performance

indices: map from (value) -> (record list)

Query optimization

Declarative query -> physical execution plan

DBMS chooses the execution plan

Cost model

Data Independence

Physical

Logical
Can change/evolve schema over time
Create "views" that look like old schema

View

Replace age with birthday

```
create view animals as
(select  name, now() - bday as age,
           species
from new_animals)
```

Declarative:
Notice, however, that our procedural programs are not the only way to compute the answers to these queries!

<u>When could I do something besides the procedural programs shown above?</u>

For example, if we store animals in animal type order, we can use binary search to find the animals of a particular type quickly.

<u>Is there a cost to doing this?</u>

Have to store in sorted order (more expensive inserts)

Lots of other possibilities -- e.g., can have hash table (index) that maps from type -> records

<u>Query optimization</u> -- Depending on physical representation of data, and type of query, DBMS selects what it believes to be the *best* plan. Uses a *cost model* to estimate how long different plans will take to run.

Optimization selects which implementation of each operation to use, as well as order of individual operations -- e.g., can move selection below join.

In *declarative* programming, the physical representation -- e.g., the layout in memory or on disk -- is different than the logical representation the user's programs interact with. Optimizer's job is to implement the logical query effectively on physical representation.

in standard *imperative* programming, logical and physical representation are typically more closely aligned.

E.g. can represent store the table in sorted order, or not. Repr is not exposed in SQL, or app!

Decoupling of logical model from physical representation is known as "physical data independence"
Can store the data in different ways on disk, don't have to change program

Also talk about logical data independence
Can change the logical schema, and can avoid changing program
Views

Transactions -- Atomic actions
begin //T1
M = read sam feedtime
S = read sal feedtime
change sam feedtime to S
change sal feedtime to M
end
Intermediate state is never visible
"All or nothing"
Recoverable

Concurrency control

begin //T2
M = read sam feedtime
change sal feedtime to M
end

name	feedtime
sam	2
sal	1

Valid outcomes: If not careful, could get

n	f	n	f	n	f
sam	1	sam	2	sam	1
sal	1	sal	2	sal	2

Serial equivalence
Locking protocol -- *run by the DBMS*
acquire locks on objects before using them,
releases locks at end of transaction

Summary : Database systems provide
- relational model of data
- declarative query language
- automatic optimization
- transactions
  - atomicity
    - serial equivalence
  - durability & recoverability

Next 2 lectures + CS186

Third big idea in databases (besides data modeling, optimization + data independence): Transactions

Powerful way to handle concurrent access to the database

name	feedtime
sam	2
sally	1

Allow a user to group operations into *atomic sections*:
begin //T1
M = read sam feedtime
S = read sal feedtime
change sam feedtime to S <--- external xaction cant see this until after "end"
change sal feedtime to M
end
Another concurrent user can't see intermediate state

"All or nothing" -- xaction may fail (because it violates some constraint, for example), but if it does all its effects are undone

If xaction succeeds, its effects are permanent and on disk

Even if system crashes in the middle of a query -- need some way to ensure that partial state isn't reflected on disk -- *recovery*

Transactions may run concurrently, but effect is indistinguishable from running in some serial order -- *Serial Equivalence*

begin //T2
M = read sam feedtime
change sal feedtime to M
end

Valid outcomes: If not careful, could get

n	f	n	f	n	f
-	-	-	-	-	-
sam	1	sam	2	sam	1
sal	1	sam	2	sal	2

Under the covers, how does the system achieve serializabiliy? Run only one transaction at a time?
Bad idea -- no concurrency, can't take advantage of multiple CPUs, can't mask disk stalls

Idea -- use automatic locking protocol