CS 348: Intro to Database Management

Michael Noukhovitch

Winter 2015, University of Waterloo

Notes written from Grant Weddel's lectures.

Contents

1 Introduction

1.1 DBMS

1.1.1 Definitions

Database: a large and persistent collection of data
DBMS: a program that manages details for storage and access to a db
Schema: a description of the data interface to the database
to abstract common functions and create a uniform interface we need:

- **data model**: all data stored uniformally
- **access control**: authorization to modify/view
- **concurrency control**: multiple applications can access at same time
- **database recovery**: nothing is lost
- **database maintenance**

1.1.2 Three-Level Schema

external schema: what the app and user see
conceptual schema: description of the logical structure of the data
physical schema: description of physical aspects (storage algorithms . . .)

DBMS allows the data to be stored via the physical schema, reasoned via the conceptual schema, and accessed via the external schema.

1.1.3 Interfacing

Interfacing to DBMS, we can interact with it through:
Data Definition Language: specifies schemas

- may be different for each schema
- the **data dictionary** (or **catalog**) stores the information

Data Manipulation Language: specifies queries and updates *(e.g SQL)*

- navigational (procedural)
- non-navigational (declarative)

1.1.4 DBAs

Database administrators are responsible for:

- managing conceptual schema
- assisting with app view integration
- monitoring and tuning DBMS performance

- defining internal schema
- loading and reformating DB
- security and reliability

1.2 Big Ideas

There are three big ideas which have influenced the creation and development of databases

1.2.1 Quantification

Database queries can be described by relational algebra as quantifiers

1.2.2 Data Independence

Data Independence: allow each schema to be independant of the others

- **physical independance**: application immune to changes in storage structure
- **logical independence**: application immune to changes in data organization

1.2.3 Transaction

Transaction: an application-specified atomic and durable unit of work
ACID: transaction properties ensured by the DBMS

- **atomic**: a transaction cannot be split up
- **consistency**: each transaction preserves consistency
- **isolated**: concurrent transaction don't interfere with each other
- **durable**: once completed, changes are permanent

2 Relational Model

2.1 Definitions

Relational model: all information is organized in (flat) relations

- powerful and declarative query language
- semantic integrity constraints (using first order logic)
- data independence

2.2 Properties

- based on finite set theory
  - attribute ordering *not strictly necessary*
  - tuples identified by attribute values
  - instance has set semantics *no ordering, no duplicates*
- all attribute values are atomic
- **degree**: number of attributes in schema
- **cardinality**: number of tuples in instance

We can algebraically define databases as a finite set of relation schemas

2.3 Relations vs SQL Tables

SQL has extensions on top of the relational model:

1. semantics of instances:
   - relations are **sets** of tuples
   - tables are **multisets** (bags) of tuples
2. unknown values: SQL includes `Null`

3 Relation Algebra

3.1 Primary Operators

- **Relation Name**: R
- **Selection**: $\sigma_{condition}(E)$ satisfies some condition
- **Projection**: $\pi_{attributes}(E)$ only includes these attributes
- **Rename**: $\rho(R(\bar{F}), E)$ (where $\bar{F}$ is a list of $oldname \mapsto newname$)
- **Product**: $E_1 \times E_2$

3.2 Joins

- **Conditional Join**: $E_1 \bowtie_{condition} E_2$
- **Natural Join**: $E_1 \bowtie E_2$ common attributes

3.3 Set Operators

Schemas R and S must be **union compatible**: have same number (and type) of fields

- **Union**: $R \cup S$
- **Difference**: $R - S$
- **Intersection**: $R \cap S$
- **Division**: $R \,/\, S$ (*opposite of* $\times$)

4 SQL

4.1 SQL Standard

Data Manipulation Language : query and modify tables

Data Definition Language : create tables and enforce access/security

Example 4.1. Basic query block

```
select attribute-list
from relation-list
[where condition]
```

4.2 DML

4.2.1 Null

A necessary evil that indicates unknown or missing data

- test using `is (not)NULL`
- expressions with `NULL` e.g. `x + NULL = NULL`
- `where` treats `NULL` like `False`

4.2.2 Subquery

`where` supports predicates as part of its clause

Example 4.2. select all employees with the highest salary

```
select empno, lastname
from employee
where salary >= all
    ( select salary
     from employee )
```

4.2.3 Ordering

No ordering can be assumed unless you use `order by`

4.2.4 Grouping

`group by` allows you to aggregate results

Example 4.3. for each dept, list number of employees and combined salary

```
select deptno, deptname, sum(salary) as totalsalary,
    count(*)as employees
from department d, employee e
where e.workdept = d.deptno
group by deptno, deptname
```

`having` is like `where` for groups

Example 4.4. list average salary for each dept >= 4 people

```
select deptno, deptname, avg(salary) as MeanSalary
    count(*)as employees
from department d, employee e
where e.workdept = d.deptno
group by deptno, deptname
having count(*) >= 4
```

4.3 DDL

4.3.1 Table

create : creates a table

alter : change the table

drop : delete the table

Example 4.5. create table

```
create table Employee (
EmpNo char(6),
FirstName varchar(12),
HireDate date
)
```

4.3.2 Data Types

- integer
- decimal(p,q)
- float(p)
- char(n)
- varchar(n): variable length
- date
- time
- timestamp: date + time
- year/month interval
- day/time interval

4.3.3 Constraints

- not NULL
- primary key
- unique
- foriegn key
- column or tuple check

Example 4.6. add a start date that must come before hire date

```
alter table Employee
add column StartDate date
add constraint hire_before_start
    check (HireDate <= StartDate);
```

4.3.4 Triggers

trigger: procedure executed by the db in response to table change

- event
- condition
- action

```
create trigger log_addr
after update of addr, phone on person
referencing OLD as o NEW as n
for each row
mode DB2SQL
when (o.status = 'VIP' or n.status = 'VIP')
    insert into VIPaddrhist(pid, oldaddr, oldphone,
        newaddr, newphone, user, modtime)
    values (o.pid, o.addr, o.phone,
        n.addr, n.phone, user, current timestamp)
```

5 Views

5.1 Definition

View: a relation whose instance is determined by other relations

- **Virtual**: views not stored, used only for querying
- **Materialized**: query for view is executed and view is stored

```
create [materialized] view <name>
    as query
```

Example 5.1. Manufacturing projects view

```
create view ManufacturingProjects as
    ( select projno, projname, firstname, lastname
      from project, employee
      where respemp = empno and deptno = 'D21' )
```

5.2 Updating

Changes to a view schema propogate back to instances of relations in conceptual schema, so to avoid ambiguity a view is updateable if:

- the query references exactly one table
- the query only outputs simple attributes
- there is **no** grouping/aggregation/distinct
- there are no nested queries
- there are no set operations

Materialized views also have to be update with periodically to account for base table changes

6 Application Development

6.1 Static Embedded SQL

Embed SQL into C with `EXEC SQL` and suffixing with `;`, using host variables to send and recieve values from DB

Example 6.1. Host variables in C

```
EXEC SQL BEGIN DECLARE SECTION;
char deptno[4];
char deptname[30];
char mgrno[7];
char admrdept[4];
char location[17];
EXEC SQL END DECLARE SECTION;

/ * program assigns values to variables * /

EXEC SQL INSERT INTO
Department(deptno,deptname,mgrno,admrdept,location)
VALUES
(:deptno,:deptname,:mgrno,:admrdept,:location);
```

indicator variables are flags used to handle host variables that might recieve `NULL`

Example 6.2. Indicator variables

```
int PrintEmployeePhone( char employeenum[] ) {
EXEC SQL BEGIN DECLARE SECTION;
    char empno[7];
    char phonenum[5];
    short int phoneind;
EXEC SQL END DECLARE SECTION;
    strcpy(empno,employeenum);
EXEC SQL
    SELECT phoneno INTO :phonenum :phoneind
    FROM employee WHERE empno = :empno;
if( SQLCODE < 0) { return( -1 ); } /* error */
else if(SQLCODE&=& 100){printf("no such employee\n");}
else if (phoneind<0){printf("phone unknown\n");}
else { printf("%s\n",phonenum); }
return( 0 );
}]
```

cursors: pointer-like objects used to iterate when > 1 row is returned. Can be before the first tuple, on a tuple, after the last tuple.

1. declare the cursor
2. open the cursor
3. fetch tuples using the cursor
4. close the cursor

6.2 Dynamic Embedded SQL

When tables, columns, predicates are not known at the time application is written

1. `PREPARE`: prepare statement for execution
2. `EXECUTE`: execute the statement
3. **placeholder**: appears instead of literals, host variables replace during execution
4. **descriptor**: used to determine input and output numbers and types with `DESCRIBE`

SQLJ: allows embedding SQL into Java, with runtime established via JDBC connection

Example 6.3. Host variables and placeholders

```
EXEC SQL BEGIN DECLARE SECTION;
char s[100] = ‘‘INSERT INTO employee VALUES (?, ?, ... )’’;
char empno[7];
char firstname[13];
...
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE S1 FROM :s;
strcpy(empno,‘‘000111’’);
strcpy(firstname,‘‘Ken’’);
...
EXEC SQL EXECUTE S1 USING :empno, :firstname, ... ;
```

6.3 Call Level Interface

A vendor-neutral ISO-standard programming interface for SQL systems. As opposed to embedded SQL, does not need to be recompiled to access different DBMS and can access multiple at the same time

- queries represented as strings in the application
- queries prepared then executed
- app won't know numbers and types, descriptor areas hold metadata and actual data
- describing a query makes DBMS place type info into descriptor area which app can read

6.4 Stored Procedure

Allows to execute application logic directly inside DBMS process

- minimize data transfer costs
- centralize application code
- logical independence

Example 6.4.
```
CREATE FUNCTION sumSalaries(dept CHAR(3))
    RETURNS DECIMAL(9,2)
LANGUAGE SQL
RETURN
    SELECT sum(salary)
    FROM employee
    WHERE workdept = dept
```

7 Data Modelling

7.1 Basic ER Modelling

Used for database design, described in terms of:

Entity : a distinguishable object (*e.g.* student)

Attributes : describes properties of entities (*e.g.* studentName)

Relationship : representation of two related entities (*e.g.* student in course)

Role : the function of an entity set in a relationship

7.2 Constraints in ER Models

- primary key
- relationship types (*e.g.* N:1 or N:N)
- existence dependencies: a *subordinate* entity depends on a *dominant* entity

  weak entity set : entity set containing subordinate entities
  strong entity set : entity set without subordinate entities
- cardinality constraints

7.3 Extensions

- structured attributes:
  - composite attributes: composed of a fixed number of attributes
  - multi-valued attributes: set-valued attributes
- aggregation: relationships can be viewed as higher-level attributes
- specialization: a more specialized"child" entity (*e.g.* graduate student)
- generalization: a more general "parent" entity (*e.g.* vehicle → car, truck)
- disjointness: specialized entity sets are disjoint but can have entities in common

7.4 Design Considerations

- attribute or entity set: separate object → entity set
- entity set or relationship set
- degrees of relationship
- extended features?

8 Mapping ER to Relational Tables

8.1 Main Ideas

- entity set maps to new table
- attribute maps to new column
- relationship set maps to new columns or new table
- specialization maps to new tables for specialized entity
- generalization maps to tables for specialized entities only

8.2 Entity Sets

- strong entity sets: map primary key of set to primary key of table
- weak entity sets: table should include attributes of identifying relationship

8.3 Relationship Sets

- if relationship is identifying for weak entity set, do nothing
- if we can deduce cardinality constraints, then to one entity table add columns:
  - attributes of relationship set
  - primary key of related entity sets
- otherwise create a table for this relationship

9 ER Schema Refinement

9.1 Design Principles

- relations should have semantic unity
- repetition should be avoided
- avoid null values
- avoid spurious joins

9.2 Functional Dependencies

We want to express that some attributes uniquely determine other attributes, so if $X \to Y$ then for any tuples t, u where $t[X] = u[X]$, it is true that $t[Y] = u[Y]$

superkey : set of attributes such that no two tuples have the same values for them

candidate key : a minimal superkey

primary key : a candidate key chosen by the DBA

9.2.1 Closures

If we have a set of functional dependencies F, then the **functional closure** of them, $F+$, is all the dependencies that can be derived from F.

Closures can be derived using:

- reflexivity: if $X \subseteq Y$ then $Y \to X$
- augmentation: if $X \to Y$ then $XZ \to YZ$
- transitivity: if $X \to Y$ and $Y \to Z$ then $X \to Z$

Attribute closure, $X+$, is the set of all attributes functionally determined by some attribute X.

9.3 Schema Decomposition

We can decompose a schema R into many schemas $R_1, \ldots, R_n$ kind of like breaking down one big SQL table into smaller ones

Lossless Decomposition : decomposing a schema so that a natural join of subschemas has no extra rows

Dependecy Preservation : is making sure all functionally dependencies are still satisfied within our subschemas. Subschemas that requires fewer checks to ensure dependency preservation are better

9.4 Normal Forms

9.4.1 BCNF

Enforces the idea that independent relationships are stored in separate tables. A schema R is in BCNF if for all $X \to Y$ either

- X is a superkey of R (only appears in one row) or
- $Y \subseteq X$

To get BCNF, decompose original relation by removing a violating $X \to Y$ and making it into it's own relation.

9.4.2 Minimal Cover

To find the candidate key, we need to find the **minimal cover**: the minimal set of dependencies G such that the resulting closure is the same as our original closure $G+ = F+$. For each depenendency $X \to A$:

- every right hand A is a single attribute
- every dependency is necessary
- every left hand side X is minimal

To find the minimal cover:

1. change every $X \to YZ$ to $X \to Y, X \to Z$
2. change $AX \to Y$ to $X \to Y$ if X can determine Y on its own
3. remove $X \to Y$ if X can determine Y without this relation

9.4.3 3NF

Less strict version of BCNF, so that A schema R is in BCNF if for all $X \to Y$ either

- X is a superkey of R (only appears in one row) or
- $Y \subseteq X$ or

- every attribute in $Y - X$ is in the candidate key

Create 3NF in two ways:

- decomposition: decompose similar to BCNF then "repair" by adding all relations in minimal cover that were not satisified
- synthesis: turn every rule of the minimal cover into a relation and (if not already there) add a relation for candidate key

10 Transactions and Concurrency

10.1 Why

Transaction: application specific unit of work, ACID guaranteed

- **atomic**: a transaction cannot be split up
- **consistency**: each transaction preserves consistency
- **isolated**: concurrent transaction don't interfere with each other
- **durable**: once completed, changes are permanent

They help prevent problems from failures, when the system fails during execution, and concurrency

10.2 Serializability

Concurrent transactions must appear to have been executed one at a time, everything in some order.

Equivalence : if two transaction histories are over the same set of transactions and the ordering of each conflict pair is the same

Serializable : a history that has an equivalent serial history (where all of T_i goes before T_j). A history is serializable iff its serialization graph is acyclic

10.3 Transactions in SQL

10.3.1 Abort and Commit

An **active** (started but not finished) transaction can terminate in two ways:

- **abort**: transaction fails and any updates are undone (SQL `rollback work`)
- **commit**: transaction succeeds and updates become durable and visible to other transactions (SQL `commit work`)

10.3.2 Isolation Levels

SQL allows serializability guarantee to be relaxed in four levels:

0. Read Uncommitted: transactions can see uncommitted updates
1. Read Committed: transactions sees only committed changes but non-repeatable reads are possible
2. Repeatable Read: reads are repeatable but "phantoms" are possible
3. Serializability

10.4 Implementing Transactions

10.4.1 Concurrency Control

Guarantees that the execution history has desired properties (*e.g serializability*)

Strict Two-Phase Locking:

- before a transaction may read or write an object, it must have a lock on it
  - shared lock required for read
  - exclusive lock required for write
- only one lock may be held on an object (unless all are shared locks)
- a transaction may not release locks until it commits

Lock conflicts, when two or more transactions request a lock, can be resolved with

- blocking: second transaction forced to wait
- pre-emption: first transaction is aborted and gives up the lock

10.4.2 Recovery Management

Guarantees that committed transactions are durable (despite failures) and aborted transactions have no effect by

- rollback of individual transactions
- recovery from system failures

System Failure:

- database server is halted abruptly
- processing current SQL commands is halted abruptly
- connections to clients are broken
- contents of memory buffers are lost
- database files are not damaged

After a system failure, every transaction is either restarted or rolled back and their committed changes are not lost

Logging is the way this is implemented, using a log in persistent storage that writes the following commands *before* updating:

- UNDO information: old versions of objects used to undo changes when that transaction aborts
- REDO information: new versions of objects used to redo changes by a transaction that commits
- BEGIN/COMMIT/ABORT: records whenever a transaction does something

Recovering from a system failure:

1. scan the log from newest to oldest
    - create list of committed transactions
    - undo updates of active and aborted transactions
2. scan the log from oldest to newest
    - redo updates of committed transactions

Rolling back a transaction:

- scan from the newest to the transaction's BEGIN
- undo the transaction's updates

11 Physical Database Design and Tuning

11.1 Introduction

Physical Design : selecting data structures to implement conceptual schema

Tuning : periodically adjusting physical and/or conceptual schema to adapt to changing requirements or performance characteristics

Workload Description :

- most important queries and their frequency
- most important updates and their frequency
- performance goal for each query and update

11.2 Designing and Tuning the Physical Schema

11.2.1 Indexing

A storage strategy is chosen for each relation (*e.g heap*) and then we add indexes:

- substantially reduce selection time for queries with index
- increase insertion time
- increase or decrease update and deletion time
- increase space used to store the table

Create Index:

```
create index LastnameIndex
on Employee(Lastname) [CLUSTER]
```

Remove Index:

```
drop index LastnameIndex
```

Clustering Index : tuples with similar values are stored together in the same block. A relation can have at most one clustering index

Co-Clustering Index : two relations with their tuples interleaved within the same file

- useful for storing heirarchical data (1:N)
- speeds up joins but slows down sequential scans of either

Multi-Attribute Index : index on more than one attribute, sorting first by the first attribute, then by second ...

11.2.2 Guidelines for Physical Design

- index only when performance increase outweighs update overhead
- attributes mentiontioned in `WHERE` clauses can be used for index search keys
- mulit-attribute search keys should be used when:
  - a `WHERE` clause contains multiple conditions
  - it enables index-only plans
- choose indexes that benefit as many queries as possible
- choose clustering scheme wisely, you can only have one
  - range queries benefit the most from clustering
  - join queries benefit the most from co-clustering
  - multi-attribute index for index-only plan doesn't benefit
- there's a DB2 Index Advisor!

11.3 Tuning the Conceptual Schema

Sometimes, even after tuning the physical schema, performance goals are not met

Denormalization: merging schemas to intentionally increase redundancy

- decrease query overhead
- increase update overhead

Partitioning: splitting a large table into multiple tables to reduce I/O costs or lock contention

- **Horizontal Partitioning**: each partition has all the columns and a subset of rows
  - tuples assigned based on a natural criteria
  - often used to separate current from old data
- **Vertical Partitioning**: each partition has all the rows and a subset of the columns
  - used to separate frequently-used columns from each other (to improve concurrency)

11.4 Tuning Queries and Applications

11.4.1 Tuning Queries

- changes to schema impact everything, but sometimes we want to target specific queries
- sorting is expensive, avoid `ORDER BY ...`
- replace subqueries with joins
- replace correlated subqueries with uncorrelated subqueries
- use vendor-supplied tools to examine generated plan, update if your cost estimation sucks

11.4.2 Tuning Applications

- minimize communication costs
  - return fewest columns/rows necessary
  - update multiple rows with `WHERE` instead of cursor
- minimize lock contention and concurrency hot spots
  - delay updates as long as possible
  - delay operations on hot spots as long as possible
  - shorten/split transactions
  - perform insert/update/delete in batches
  - consider lower isolation levels