Relational databases and SQL

Matthew J. Graham
CACR

Methods of Computational Science
Caltech, 29 January 2009

matthew graham

relational model

- Proposed by E. F. Codd in 1969
- An attribute is an ordered pair of attribute name and type (domain) name
- An attribute value is a specific valid value for the attribute type
- A tuple is an unordered set of attribute values identified by their names
- A relation is defined as an unordered set of n-tuples

databases

- A relation consists of a heading (a set of attributes) and a body (n-tuples)
- A relvar is a named variable of some specific relation type and is always associated with some relation of that type
- A relational database is a set of relvars and the result of any query is a relation
- A table is an accepted representation of a relation:
  attribute => column, tuple => row

structured query language

- Appeared in 1974 from IBM
- First standard published in 1986; most recent in 2006
- SQL92 is taken to be default standard
- Different flavours:

Microsoft/Sybase	Transact-SQL
MySQL	MySQL
Oracle	PL/SQL
PostgreSQL	PL/pgSQL

create

CREATE DATABASE *databaseName*
CREATE TABLE *tableName* (name1 type1, name2 type2, . . .)

```
CREATE TABLE star (name varchar(20), ra float, dec float, vmag float)
```

Data types:

- boolean, bit, tinyint, smallint, int, bigint;
- real/float, double, decimal;
- char, varchar, text, binary, blob, longblob;
- date, time, datetime, timestamp

```
CREATE TABLE star (name varchar(20) not null, ra float default 0, ...)
```

keys

```
CREATE TABLE star (name varchar(20), ra float, dec float, vmag float,
    CONSTRAINT PRIMARY KEY (name))
```

- A primary key is a unique identifier for a row and is automatically not null

```
CREATE TABLE star (name varchar(20), ..., stellarType varchar(8),
    CONSTRAINT stellarType_fk FOREIGN KEY (stellarType)
    REFERENCES stellarTypes(id))
```

- A foreign key is a referential constraint between two tables identifying a column in one table that refers to a column in another table.

insert

INSERT INTO *tableName* VALUES(val1, val2, . . .)

```
INSERT INTO star VALUES('Sirius', 101.287, -16.716, -1.47)

INSERT INTO star(name, vmag) VALUES('Canopus', -0.72)

INSERT INTO star
    SELECT ...
```

delete

DELETE FROM *tableName* WHERE *condition*
TRUNCATE TABLE *tableName*
DROP TABLE *tableName*

```
DELETE FROM star WHERE name = 'Canopus'

DELETE FROM star WHERE name LIKE 'C_n%'

DELETE FROM star WHERE vmag > 0 OR dec < 0

DELETE FROM star WHERE vmag BETWEEN 0 and 5
```

update

UPDATE *tableName* SET *columnName* = val1 WHERE *condition*

```
UPDATE star SET vmag = vmag + 0.5

UPDATE star SET vmag = -1.47 WHERE name LIKE 'Sirius'
```

select

SELECT *selectionList* FROM *tableList* WHERE *condition* ORDER BY *criteria*

```
SELECT name, constellation FROM star WHERE dec > 0
    ORDER by vmag

SELECT * FROM star WHERE ra BETWEEN 0 AND 90

SELECT DISTINCT constellation FROM star

SELECT name FROM star LIMIT 5
    ORDER BY vmag
```

joins

- Inner join: combining related rows

```
SELECT * FROM star s INNER JOIN stellarTypes t ON s.stellarType = t.id

SELECT * FROM star s, stellarTypes t WHERE s.stellarType = t.id
```

- Outer join: each row does not need a matching row

```
SELECT * from star s LEFT OUTER JOIN stellarTypes t ON s.stellarType = t.id

SELECT * from star s RIGHT OUTER JOIN stellarTypes t ON s.stellarType = t.id

SELECT * from star s FULL OUTER JOIN stellarTypes t ON s.stellarType = t.id
```

aggregate functions

COUNT, AVG, MIN, MAX, SUM

```
SELECT COUNT(*) FROM star

SELECT AVG(vmag) FROM star

SELECT stellarType, MIN(vmag), MAX(vmag) FROM star
    GROUP BY stellarType

SELECT stellarType, AVG(vmag), COUNT(id) FROM star
    GROUP BY stellarType
    HAVING vmag > 14
```

views

CREATE VIEW *viewName* AS …

```
CREATE VIEW region1View AS
    SELECT * FROM star WHERE ra BETWEEN 150 AND 170
        AND dec BETWEEN -10 AND 10

SELECT id FROM region1View WHERE vmag < 10

CREATE VIEW region2View AS
    SELECT * FROM star s, stellarTypes t WHERE s.stellarType = t.id
        AND ra BETWEEN 150 AND 170 AND dec BETWEEN -10 AND 10

SELECT id FROM regionView2 WHERE vmag < 10 and stellarType LIKE 'A%'
```

indexes

CREATE INDEX *indexName* ON *tableName*(*columns*)

```
CREATE INDEX vmagIndex ON star(vmag)
```

- A clustered index is one in which the ordering of data entries is the same as the ordering of data records
- Only one clustered index per table but multiple unclustered indexes
- Typically implemented as B+ trees but alternate types such as bitmap index for high frequency repeated data

spatial indexing

- Find all objects near a particular point
- Remember spherical geometry
- R-tree - multi-dimensional index supported by PostgreSQL and Oracle
- Pixellation algorithms: HTM, HealPix
- Zone approach

zones

- Simple idea often more efficient than technological solution: SQL can evaluate 10^6 spatial distance calculations per sec. per CPU GHz but function calls costs $100\times$ more

- Divide declination into zones of equal height:

```
zone = dec / zoneHeight

create table ZoneIndex ( zone int, objid bigint,
        ra float, dec float, ...,
        primary key (zone, ra, objid))
```

zone crossmatch

```
select z1.objid as objid1, z2.objid as objid2
    into #answer
    from zone z1 join zone z2
    where z1.zone = z2.zone
        and z1.objid <> z2.objid
        and z1.margin = 0
        and ra between ra - @maxAlpha and @ra + @maxAlpha
        and dec beween @dec - @theta and @dec + @theta
        and (cx+@x + cy*@y + cz+@z) > cos(radians(@theta))

insert #answer
    select objid2, objid1 from #answer
```

normalisation

- First normal form: no repeating elements or groups of elements
  table has a unique key (and no nullable columns)

- Second normal form: no columns dependent on only part of the key

Star Name	Constellation	Area

- Third normal form: no columns dependent on other non-key columns

Star Name	Magnitude	Flux

partitions

- Horizontal: different rows in different tables
- Vertical: different columns in different tables (normalisation)
- Range: rows where values in a particular column are inside a certain range
- List: rows where values in a particular column match a list of values
- Hash: rows where a hash function returns a particular value

transactions

BEGIN WORK … COMMIT / ROLLBACK

```
BEGIN WORK
    INSERT INTO star VALUES(...)
COMMIT

BEGIN WORK
    UPDATE star SET vmag = NULL
ROLLBACK
```

cursors

DECLARE *cursorName* CURSOR FOR SELECT ...
OPEN *cursorName*
FETCH *cursorName* INTO ...
CLOSE *cursorName*

- A cursor is a control structure for successive traversal of records in a result set

- Slowest way of accessing data

cursors example

- For each row in the result set, update the relevant stellar model

```
DECLARE @name varchar(20)
DECLARE @mag float
DECLARE starCursor CURSOR FOR
    SELECT name, AVG(vmag) FROM star
        GROUP BY stellarType
OPEN starCursor
   FETCH starCursor INTO @name, @mag
   EXEC updateStellarModel @name, @mag / CALL updateStellarModel(@name, @mag)
CLOSE starCursor
```

triggers

CREATE TRIGGER *triggerName* ON *tableName* ...

> A trigger is procedural code that is automatically executed in response to certain events on a particular table:
>
> - INSERT
> - UPDATE
> - DELETE

```
CREATE TRIGGER starTrigger ON star FOR UPDATE AS
    IF @@ROWCOUNT = 0 RETURN
    IF UPDATE (vmag) EXEC refreshModels
GO
```

stored procedures

CREATE PROCEDURE *procedureName* @param1 type, . . .
  AS ...

```
CREATE PROCEDURE findNearestNeighbour @starName varchar(20) AS
BEGIN
    DECLARE @ra, @dec float
    DECLARE @name varchar(20)
    SELECT @ra = ra, @dec = dec FROM star WHERE name LIKE @starName
    SELECT name FROM getNearestNeighbour(@ra, @dec)
END

EXEC findNearestNeighbour 'Sirius'
```

programming

Java

```
import java.sql.*
...
String dbURL = "jdbc:mysql://127.0.0.1:1234/test";
Connection conn = DriverManager.getConnection(dbUrl, "mjg", "mjg");
Statement stmt = conn.createStatement();
ResultSet res = stmt.executeQuery("SELECT * FROM star");
...
conn.close();
```

programming

Python:

```
import MySQLdb
Con = MySQLdb.connect(host="127.0.0.1", port=1234, user="mjg",
    passwd="mjg", db="test")
Cursor = Con.cursor()
sql = "SELECT * FROM star"
Cursor.execute(sql)
Results = Cursor.fetchall()
...
Con.close()
```

skynodes

- Distributed query across heterogeneous databases using common programmatic interface
- Poll each database to get estimate of cost of query ($QueryCost$)
- Execution plan consists of ordered list of node identifier, node URL and query for that node
- Node receives execution plan: if there are subsequent nodes, submit execution plan to downline node, ingest result table and perform query

examples

- NVO SQL tutorial examples

  `http://www.us-vo.org/summer-school/2006/proceedings/presentations/SQL2006.html`

- SDSS SkyServer SQL examples

  `http://cas.sdss.org/dr6/en/help/docs/realquery.asp`

- CasJobs

  `http://casjobs.sdss.org/CasJobs/`