Databases
Lectures 1 and 2

Timothy G. Griffin

Computer Laboratory
University of Cambridge, UK

Databases, Lent 2009
Re-ordered Syllabus

Note: All lecture slides have been written from scratch for Lent 2009 — please help me find the typos!

Lecture 01  **Basic Concepts.** Relations, attributes, tuples, and relational schema. Tables in SQL.

Lecture 02  **Query languages.** Relational algebra, relational calculi (tuple and domain). Examples of SQL constructs that mix and match these models.

Lecture 03  **More on SQL.** Null values (and three-valued logic). Inner and Outer Joins. Views and integrity constraints.

Lecture 04  **Database updates.** Basic ACID properties. Serializability in multi-user database context. 2-phase commits. Locking vs. update throughput.
Re-ordered Syllabus

Lecture 05  **Redundancy is a Bad Thing.** Update anomalies. More redundancy implies more locking. Capturing redundancy with functional and multivalued dependencies.

Lecture 06  **Analysis of Redundancy.** Implied functional dependencies, logical closure. Reasoning about functional dependencies.


Lecture 08  **Decomposition algorithms.** Decomposition examples. Multivalued dependencies and Fourth normal form.
Re-ordered Syllabus

Lecture 09  **Multisets, grouping, and aggregates.** Bag (multiset) algebra. Aggregates and grouping examples in SQL. More problems with null values.

Lecture 10  **Redundancy is a Good Thing!** The main issue: query response vs. update throughput. Indices are derived data! Selective de-normalization. Materialized views. The extreme case: “read only” database, data warehousing, data-cubes, and OLAP vs OLTP.

Lecture 11  **Entity-Relationship Modeling.** High-level modeling. Entities and relationships. Representation in relational model. Reverse engineering as a common application.

Lecture 12  **What is a DBMS?** Different levels of abstraction, data independence. Other data models (Object-Oriented databases, Nested Relations). XML as a universal data exchange language.
# Recommended Reading

## Textbooks

<table>
<thead>
<tr>
<th>Code</th>
<th>Author</th>
<th>Title</th>
<th>Publisher</th>
<th>Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2004</td>
<td>Date, C.J.</td>
<td>An introduction to database systems</td>
<td>Addison-Wesley (8th ed.)</td>
<td></td>
</tr>
</tbody>
</table>
### Research Papers (Google for them)

<table>
<thead>
<tr>
<th>Year</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1970</td>
<td>E.F. Codd, (1970)</td>
<td>&quot;A Relational Model of Data for Large Shared Data Banks&quot;. Communications of the ACM.</td>
</tr>
</tbody>
</table>
Lecture 01: Relations and Tables

Lecture Outline

- Relations, attributes, tuples, and relational schema
- Representation in SQL: Tables, columns, rows (records)
- Important: users should be able to create and manipulate relations (tables) without regard to implementation details!
Edgar F. Codd

- The problem: in 1970 you could not write a database application without knowing a great deal about the low-level physical implementation of the data.
- Codd’s radical idea [C1970]: give users a model of data and a language for manipulating that data which is completely independent of the details of its physical representation/implementation.
- This decouples development of Database Management Systems (DBMSs) from the development of database applications (at least in an idealized world).
Let’s start with mathematical relations

Suppose that $S_1$ and $S_2$ are sets. The Cartesian product, $S_1 \times S_2$, is the set

$$S_1 \times S_2 = \{(s_1, s_2) \mid s_1 \in S_1, \ s_2 \in S_2\}$$

A (binary) relation over $S_1 \times S_2$ is any set $r$ with

$$r \subseteq S_1 \times S_2.$$

In a similar way, if we have $n$ sets,

$$S_1, \ S_2, \ldots, S_n,$$

then an $n$-ary relation $r$ is a set

$$r \subseteq S_1 \times S_2 \times \cdots \times S_n = \{(s_1, s_2, \ldots, s_n) \mid s_i \in S_i\}$$
Did you notice the prestidigitation?

What do we really mean by this notation?

\[ S_1 \times S_2 \times \cdots \times S_n \]

Does it represent \( n - 1 \) applications of a binary operator \( \times \)? NO!

If we wanted to be extremely careful we might write something like

\[ \times (S_1, S_2, \ldots, S_n) \]

We perform this kind of sleight of hand very often. Here’s an example from OCaml:

```ocaml
let flatten_left : (('a * 'b) * 'c) -> ('a * 'b * 'c) = function p ->
    (fst (fst p), snd (fst p), snd p)
```

Perhaps if we had the option of writing \( \ast ('a, 'b, 'c) \) it would make this implicit flattening more obvious.
Mathematical vs. database relations

Suppose we have an $n$-tuple $t \in S_1 \times S_2 \times \cdots \times S_n$. Extracting the $i$-th component of $t$, say as $\pi_i(t)$, feels a bit low-level.

Solution: (1) Associate a name, $A_i$ (called an attribute name) with each domain $S_i$. (2) Instead of tuples, use records — sets of pairs each associating an attribute name $A_i$ with a value in domain $S_i$.

A database relation $R$ over the schema $A_1 : S_1 \times A_2 : S_2 \times \cdots \times A_n : S_n$ is a finite set

$$R \subseteq \{\{(A_1, s_1), (A_2, s_2), \ldots, (A_n, s_n)\} \mid s_i \in S_i\}$$
A relational schema

**Students**\( (\text{name}: \text{string}, \text{sid}: \text{string}, \text{age} : \text{integer}) \)

A relational instance of this schema

\[
\text{Students} = \left\{ \begin{array}{l}
(\text{name}, \text{Fatima}), (\text{sid}, \text{fm21}), (\text{age}, 20) \\
(\text{name}, \text{Eva}), (\text{sid}, \text{ev77}), (\text{age}, 18) \\
(\text{name}, \text{James}), (\text{sid}, \text{jj25}), (\text{age}, 19) \end{array} \right\}
\]

A tabular presentation

<table>
<thead>
<tr>
<th>name</th>
<th>sid</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fatima</td>
<td>fm21</td>
<td>20</td>
</tr>
<tr>
<td>Eva</td>
<td>ev77</td>
<td>18</td>
</tr>
<tr>
<td>James</td>
<td>jj25</td>
<td>19</td>
</tr>
</tbody>
</table>
Creating Tables in SQL

create table Students
  (sid varchar(10),
   name varchar(50),
   age int);

-- insert record with attribute names
insert into Students set
  name = 'Fatima', age = 20, sid = 'fm21';

-- or insert records with values in same order
-- as in create table
insert into Students values
  ('jj25', 'James', 19),
  ('ev77', 'Eva', 18);
Listing a Table in SQL

```sql
-- list by attribute order of create table
mysql> select * from Students;
+-----------------------------+
<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>ev77</td>
<td>Eva</td>
<td>18</td>
</tr>
<tr>
<td>fm21</td>
<td>Fatima</td>
<td>20</td>
</tr>
<tr>
<td>jj25</td>
<td>James</td>
<td>19</td>
</tr>
</tbody>
</table>
+-----------------------------+
3 rows in set (0.00 sec)
```
Listing a Table in SQL

-- list by specified attribute order
mysql> select name, age, sid from Students;
+--------+------+------+
| name   | age  | sid  |
+--------+------+------+
| Eva    | 18   | ev77 |
| Fatima | 20   | fm21 |
| James  | 19   | jj25 |
+--------+------+------+
3 rows in set (0.00 sec)
Keys in SQL

A key is a set of attributes that will uniquely identify any record (row) in a table. We will get more precise in Lecture 06.

```sql
-- with this create table
create table Students
    (sid varchar(10),
     name varchar(50),
     age int,
     primary key (sid));

-- if we try to insert this (fourth) student ...
mysql> insert into Students set
    name = 'Flavia', age = 23, sid = 'fm21';

ERROR 1062 (23000): Duplicate entry 'fm21' for key 'PRIMARY'
```
Put all information in one big table?

Suppose we want to add information about college membership to our Student database. We could add an additional attribute for the college.

<table>
<thead>
<tr>
<th>StudentsWithCollege</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Eva</td>
</tr>
<tr>
<td>Fatima</td>
</tr>
<tr>
<td>James</td>
</tr>
</tbody>
</table>
Put logically independent data in distinct tables?

Students: +------------------------------------------+
<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>sid</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eva</td>
<td>18</td>
<td>ev77</td>
<td>k</td>
</tr>
<tr>
<td>Fatima</td>
<td>20</td>
<td>fm21</td>
<td>cl</td>
</tr>
<tr>
<td>James</td>
<td>19</td>
<td>jj25</td>
<td>cl</td>
</tr>
</tbody>
</table>
+------------------------------------------+

Colleges: +------------------------------------------+
<table>
<thead>
<tr>
<th>cid</th>
<th>college_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>King’s</td>
</tr>
<tr>
<td>cl</td>
<td>Clare</td>
</tr>
<tr>
<td>sid</td>
<td>Sidney Sussex</td>
</tr>
<tr>
<td>q</td>
<td>Queens’</td>
</tr>
<tr>
<td>...</td>
<td>.....</td>
</tr>
</tbody>
</table>

But how do we put them back together again?
The main themes of these lectures

- We will focus on databases from the perspective of an application writer.
  - We will not be looking at implementation details.
- The main question is this:
  - What criteria can we use to assess the quality of a database application?
- We will see that there is an inherent tradeoff between query response time and (concurrent) update throughput.
- Understanding this tradeoff will involve a careful analysis of the data redundancy implied by a database schema design.
Lecture 02: Relational Expressions

Outline
- Database query languages
- The Relational Algebra
- The Relational Calculi (tuple and domain)
- SQL
What is a (relational) database query language?

Input: a collection of relation instances

\[ R_1, R_2, \ldots, R_k \]

Output: a single relation instance

\[ Q(R_1, R_2, \ldots, R_k) \]

How can we express \( Q \)?

In order to meet Codd’s goals we want a query language that is high-level and independent of physical data representation.

There are many possibilities ...
\[
Q ::= R \quad \text{base relation} \\
| \sigma_p(Q) \quad \text{selection} \\
| \pi_X(Q) \quad \text{projection} \\
| Q \times Q \quad \text{product} \\
| Q - Q \quad \text{difference} \\
| Q \cup Q \quad \text{union} \\
| Q \cap Q \quad \text{intersection} \\
| \rho_M(Q) \quad \text{renaming}
\]

- \( p \) is a simple boolean predicate over attributes values.
- \( X = \{A_1, A_2, \ldots, A_k\} \) is a set of attributes.
- \( M = \{A_1 \mapsto B_1, A_2 \mapsto B_2, \ldots, A_k \mapsto B_k\} \) is a renaming map.
Relational Calculi

The Tuple Relational Calculus (TRC)

\[ Q = \{ t \mid P(t) \} \]

The Domain Relational Calculus (DRC)

\[ Q = \{(A_1 = v_1, A_2 = v_2, \ldots, A_k = v_k) \mid P(v_1, v_2, \ldots, v_k)\} \]
The SQL standard

- Origins at IBM in early 1970’s.
- SQL has grown and grown through many rounds of standardization:
  - ANSI: SQL-86
- SQL is made up of many sub-languages:
  - Query Language
  - Data Definition Language
  - System Administration Language
  - ...
### Selection

The selection operation, denoted by $Q(R)$, is defined as:

- **RA**  $Q = \sigma_{A>12}(R)$
- **TRC**  $Q = \{ t \mid t \in R \land t.A > 12 \}$
- **DRC**  $Q = \{ \{(A, a), (B, b), (C, c), (D, d)\} \mid \{(A, a), (B, b), (C, c), (D, d)\} \in R \land a > 12 \}$
- **SQL**  `select * from R where R.A > 12`
Projection

\[ R \]

\[
\begin{array}{cccc}
A & B & C & D \\
20 & 10 & 0 & 55 \\
11 & 10 & 0 & 7 \\
4 & 99 & 17 & 2 \\
77 & 25 & 4 & 0 \\
\end{array}
\]

\[ Q(R) \]

\[
\begin{array}{cc}
B & C \\
10 & 0 \\
99 & 17 \\
25 & 4 \\
\end{array}
\]

- RA  \( Q = \pi_{B,C}(R) \)
- TRC  \( Q = \{ t \mid \exists u \in R \land t.[B, C] = u.[B, C] \} \)
- DRC  \( Q = \{ \{(B, b), (C, c)\} \mid \exists\{(A, a), (B, b), (C, c), (D, d)\} \in R \} \)
- SQL  \( \text{select distinct } B, C \text{ from } R \)
Why the distinct in the SQL?

The SQL query

```
select B, C from R
```

will produce a bag (multiset)!

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>0</td>
<td>55</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>99</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>77</td>
<td>25</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

$$R \quad \quad \quad Q(R)$$

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>99</td>
<td>17</td>
</tr>
<tr>
<td>25</td>
<td>4</td>
</tr>
</tbody>
</table>

SQL is actually based on multisets, not sets. We will look into this more in Lecture 09.
### Renaming

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>0</td>
<td>55</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>99</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>77</td>
<td>25</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>E</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>0</td>
<td>55</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>99</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>77</td>
<td>25</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
R = \rho_{B \rightarrow E, \; D \rightarrow F}(R)
\]

\[
TRC \quad Q = \{ t \mid \exists u \in R \land t.A = u.A \land t.E = u.E \land t.C = u.C \land t.F = u.D \}\]

\[
DRC \quad Q = \{\{(A, \; a), \; (E, \; b), \; (C, \; c), \; (F, \; d)\} \mid \exists\{(A, \; a), \; (B, \; b), \; (C, \; c), \; (D, \; d)\} \in R\}\}
\]

**SQL**

```
select A, B as E, C, D as F from R
```
Note the automatic flattening

$RA \quad Q = R \times S$

$TRC \quad Q = \{ t \mid \exists u \in R, v \in S, t.[A, B] = u.[A, B] \land t.[C, D] = v.[C, D] \}$

$DRC \quad Q = \{ \{(A, a), (B, b), (C, c), (D, d)\} \mid \{(A, a), (B, b)\} \in R \land \{(C, c), (D, d)\} \in S \}$

$SQL \quad \text{select } A, B, C, D \text{ from } R, S$
Union

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>Q(R, S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>99</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>77</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
</tr>
</tbody>
</table>

\[
RA \quad Q = R \cup S
\]

\[
TRC \quad Q = \{ t \mid t \in R \lor t \in S \}
\]

\[
DRC \quad Q = \{ \{(A, a), (B, b)\} \mid \{(A, a), (B, b)\} \in R \lor \{(A, a), (B, b)\} \in S \}
\]

\[
SQL \quad (\text{select } * \text{ from } R) \text{ union } (\text{select } * \text{ from } S)
\]
Intersection

\[
\begin{array}{c|c}
R & S \\
\hline
A & B \\
20 & 10 \\
11 & 10 \\
4 & 99 \\
\end{array}
\quad
to
\quad
\begin{array}{c|c}
Q(R) & \\
\hline
A & B \\
20 & 10 \\
20 & 10 \\
77 & 1000 \\
\end{array}
\]

**RA**  \( Q = R \cap S \)

**TRC**  \( Q = \{ t \mid t \in R \land t \in S \} \)

**DRC**  \( Q = \{(A, a), (B, b)\} \mid \{(A, a), (B, b)\} \in R \land \{(A, a), (B, b)\} \in S \} \)

**SQL**  

\[(select \ast \text{ from } R) \text{ intersect } (select \ast \text{ from } S)\]
Difference

\[
\begin{array}{c|c}
R & S \\
\hline
A & B \\
20 & 10 \\
11 & 10 \\
4 & 99 \\
\end{array}
\quad \Rightarrow \quad
\begin{array}{c|c}
A & B \\
20 & 10 \\
77 & 1000 \\
\end{array}
\quad \Rightarrow \quad
\begin{array}{c|c}
A & B \\
11 & 10 \\
4 & 99 \\
\end{array}
\]

RA \quad Q = R - S

TRC \quad Q = \{ t \mid t \in R \land t \notin S \}

DRC \quad Q = \{ \{ (A, a), (B, b) \} \mid \{ (A, a), (B, b) \} \in R \land \{ (A, a), (B, b) \} \notin S \}

SQL \quad \text{(select * from R) except (select * from S)}
Query Safety

A query like \( Q = \{ t \mid t \in R \land t \notin S \} \) raises some interesting questions. Should we allow the following query?

\[
Q = \{ t \mid t \notin S \}
\]

We want our relations to be finite!

Safety

A (TRC) query

\[
Q = \{ t \mid P(t) \}
\]

is safe if it is always finite for any database instance.

- Problem: query safety is not decidable!
- Solution: define a restricted syntax that guarantees safety.

Safe queries can be represented in the Relational Algebra.
Given $R(\mathbf{X}, \mathbf{Y})$ and $S(\mathbf{Y})$, the division of $R$ by $S$, denoted $R \div S$, is the relation over attributes $\mathbf{X}$ defined as (in the TRC)

$$R \div S \equiv \{ x \mid \forall s \in S, \ x \cup s \in R \}.$$

<table>
<thead>
<tr>
<th>name</th>
<th>award</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fatima</td>
<td>writing</td>
</tr>
<tr>
<td>Fatima</td>
<td>music</td>
</tr>
<tr>
<td>Eva</td>
<td>music</td>
</tr>
<tr>
<td>Eva</td>
<td>writing</td>
</tr>
<tr>
<td>Eva</td>
<td>dance</td>
</tr>
<tr>
<td>James</td>
<td>dance</td>
</tr>
</tbody>
</table>

$$\div$$

<table>
<thead>
<tr>
<th>award</th>
</tr>
</thead>
<tbody>
<tr>
<td>music</td>
</tr>
<tr>
<td>writing</td>
</tr>
<tr>
<td>dance</td>
</tr>
</tbody>
</table>

$$=\quad$$

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eva</td>
</tr>
</tbody>
</table>
Division in the Relational Algebra?

Clearly, $R \div S \subseteq \pi_X(R)$. So $R \div S = \pi_X(R) - C$, where $C$ represents counter examples to the division condition. That is, in the TRC,

$$C = \{ x \mid \exists s \in S, \ x \cup s \not\in R \}.$$  

- $U = \pi_X(R) \times S$ represents all possible $x \cup s$ for $x \in X(R)$ and $s \in S$,
- so $T = U - R$ represents all those $x \cup s$ that are not in $R$,
- so $C = \pi_X(T)$ represents those records $x$ that are counter examples.

Division in RA

$$R \div S \equiv \pi_X(R) - \pi_X((\pi_X(R) \times S) - R)$$
Limitations of simple relational query languages

- The expressive power of RA, TRC, and DRC are essentially the same.
  - None can express the transitive closure of a relation.
- We could extend RA to a more powerful languages (like Datalog).
- SQL has been extended with many features beyond the Relational Algebra.
  - stored procedures
  - recursive queries
  - ability to embed SQL in standard procedural languages