Common MySQL Queries

Extending Chapter 9 of Get it Done with MySQL 5&6

TreeView

Aggregates

- Basic aggregation
- Aggregates across multiple joins
- Aggregates excluding leaders
- Aggregates of specified size
- Avoiding repeat aggregation
- Cascading aggregates
- Cross-aggregates
- Group data by datetime periods
- League table
- Sales commissions: double aggregation
- Show only one child row per parent row
- Skip repeating values
- Within-group aggregates
- Within-group aggregates with a wrinkle
- Within-group quotas (Top N per group)

Aggregates and Statistics

- Average the top 50% values per group
- Averages from bands of values
- Count unique values of one column
- Median
- Mode
- Rank order

Data comparison

- Backslashes in data
- Compare data in two tables
- Show rows where column value changed

Date and time

- Age in years
- Appointments available
- Count business days between two dates
- Count Tuesdays between two dates
- Date of first Friday of next month
- Date of Monday in a given week of the year
- Datetime difference
- Find available reservation periods
- Find overlapping periods
- Find sequenced duplicates
- Is a given booking period available?
- Julian date
- Last business day before a reference date
- Make a calendar table
- Sum for time periods
- Sum time values
- The date of next Thursday

Schedules

- Game schedule
- Pivot table schedule

DDL

- Add auto-incrementing primary key to a table
- Auto-increment: reset next value
- Change or drop a foreign key
- Compare structures of two tables
- Compare two databases
- Find child tables
- Find parent tables
- Find primary key of a table
- Find the size of all databases on the server
- List differences between two databases
- List users of a database
- Rename Database
- Show Create Trigger
- Show Tables

Frequencies

- Display column values which occur N times
- Display every Nth row

Graphs and Hierarchies

- Trees, networks and parts explosions in MySQL
- Dijkstra�s shortest path algorithm

JOIN

- Approximate joins
- Cascading JOINs
- Data-driven joins
- Full Outer Join
- Intersection and difference
- Many-to-many joins
- What else did buyers of X buy?

Join or subquery?

- Parents without children
- Parties who have contracts with one another
- The unbearable slowness of IN()
- The [Not] Exists query pattern
- What exams did a student not register for?

NULLs

- List NULLs at end of query output
- Parents with and without children

Ordering resultsets

- Next row
- Order by a column containing digits and letters
- Order by month name
- Suppress repeating ordering values

Pagination

- Pagination

Pivot tables

- Automate the writing of pivot table queries
- Column value associations
- Group column statistics in rows
- Pivot table using math tricks
- Pivot table with CONCAT
- Pivot table without GROUP_CONCAT

Relational division

- All possible recipes with given ingredients
- Parties with candidates in all districts
- Who makes all the parts for a given assembly?

Sequences

- Find blocks of unused numbers
- Find missing numbers in a sequence
- Find next highest value of each column value
- Find previous and next values in a sequence
- Find sequence starts and ends
- Find specific sequences
- Gaps in a time series
- Make values of a column sequential
- Track stepwise project completion
- Winning Streaks

Spherical geometry

- Great circle distance

Statistics without aggregates

- Moving average
- Multiple sums across a join
- Percentiles
- Random row selection
- Running Sum
- Sum across categories
- Top ten

Stored procedures

- A cursor if necessary, but not necessarily a cursor
- Emulate sp_exec
- Variable-length argument for query IN() clause

Strings

- Count delimited substrings
- Count substrings
- Levenshtein distance
- Proper case
- Strip HTML tags

Basic aggregation

This is the simplest grouping query pattern. For column *foo*, display the first (smallest), last (largest) or average value of column *bar*.

```
SELECT foo, MIN(bar) AS bar
FROM tbl
GROUP BY foo
```

To return the highest value, and order top to bottom by that value:

```
SELECT foo, MAX(bar) AS Count
FROM tbl
GROUP BY foo
ORDER BY Count DESC;
```

Ditto for AVG(), COUNT() etc. It is easily extended for multiple grouping column expressions.

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Aggregates across multiple joins

Given a parent table and two child tables, a query which sums values in both child tables, grouping on a parent table column, returns sums which are exactly twice as large as they should be. In this recent example from the MySQL General Discussion list:

```
CREATE TABLE packageItem (
 packageItemID INT,
 packageItemName CHAR(20),
 packageItemPrice DECIMAL(10,2)
);
INSERT INTO packageItem VALUES(1,'Delta Hotel',100.00);

CREATE TABLE packageCredit (
 packageCreditID INT,
 packageCreditItemID INT,
 packageItemType CHAR(10),
 packageCreditAmount DECIMAL(10,2)
);
INSERT INTO packageCredit VALUES
(1,1,'Deposit',25.00),
(2,1,'Balance',92.00);

CREATE TABLE packageItemTax (
 packageItemTaxID INT,
 packageItemTaxItemID INT,
 packageItemTaxName CHAR(5),
 packageItemTaxAmount DECIMAL(10,2)
);
INSERT INTO packageItemTax VALUES
(1,1,'GST',7.00),
(2,1,'HST',10.00);
```

The query ...

```
SELECT
  i.packageItemID,
  i.packageItemName,
  i.packageItemPrice,
  SUM(t.packageItemTaxAmount) as Tax,
  SUM(c.packageCreditAmount) as Credit
FROM packageItem i
LEFT JOIN packageCredit c ON i.packageItemID=c.packageCreditItemID
LEFT JOIN packageItemTax t ON i.packageItemID=t.packageItemTaxItemID
```

```
GROUP BY i.packageItemID;
```

returns ...

```
+---------------+-----------------+------------------+-------+--------+
| packageItemID | packageItemName | packageItemPrice | Tax   | Credit |
+---------------+-----------------+------------------+-------+--------+
|             1 | Delta Hotel     |           100.00 | 34.00 | 234.00 |
+---------------+-----------------+------------------+-------+--------+
```

With three child tables, the sums are tripled. Why? Because the query aggregates across each join.How then to get the correct results? With correlated subqueries:

```
SELECT
  packageItemID,
  SUM(packageItemPrice) AS PriceSum,
  ( SELECT SUM(c.packageCreditAmount)
    FROM packageCredit c
    WHERE c.packageCreditItemID = packageItemID
  ) AS CreditSum,
  ( SELECT SUM(t.packageItemTaxAmount)
    FROM packageItemTax t
    WHERE t.packageItemTaxItemID = packageItemID
  ) AS TaxSum
FROM packageItem
GROUP BY packageItemID;
+---------------+----------+-----------+--------+
| packageItemID | PriceSum | CreditSum | TaxSum |
+---------------+----------+-----------+--------+
|             1 |   100.00 |    117.00 |  17.00 |
+---------------+----------+-----------+--------+
```

If subqueries are unavailable or too slow, replace them with temp tables.

Back to the top *Browse the book* *Buy the book* *Feedback*

Aggregates excluding leaders

You have a table of grouped ranks ...

```
DROP TABLE IF EXISTS grps,ranks;
CREATE TABLE grps (grp int);
INSERT INTO grps VALUES(1),(2),(3),(4);
CREATE TABLE ranks(grp int,rank int);
INSERT INTO ranks VALUES(1, 4 ),(1, 7 ),(1, 9 ),(2, 2 ),(2, 3 ),(2, 5 ),(2, 6 ),(2, 8 ),(3, 1 ),(4,11 ),(4,12 ),(4,13 );
```

and you wish to list ranks by group *omitting the leading rank in each group*. The simplest query for group leaders is ...

```
SELECT grp, MIN(rank) as top
FROM ranks r2
GROUP BY grp
+------+------+
| grp  | top  |
+------+------+
|    1 |    4 |
|    2 |    2 |
|    3 |    1 |
|    4 |   11 |
+------+------+
```

The simplest way to get a result that omits these is an *exclusion join* from the ranks table to the above result:

```
SELECT r1.grp, r1.rank
FROM ranks r1
```

```
LEFT JOIN (
  SELECT grp, MIN(rank) as top
  FROM ranks r2
  GROUP BY grp
) AS r2 ON r1.grp=r2.grp AND r1.rank = r2.top
WHERE r2.grp IS NULL
ORDER BY grp, rank;
+------+------+
| grp  | rank |
+------+------+
|    1 |    7 |
|    1 |    9 |
|    2 |    3 |
|    2 |    5 |
|    2 |    6 |
|    2 |    8 |
|    4 |   12 |
|    4 |   13 |
+------+------+
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Aggregates of specified size

Find the values of a table column c1 for which there are a specified number of listed values in another column c2.

To get an overview of the values of c2 for each value of c1:

```
SELECT
  c1,
  GROUP_CONCAT(c2 ORDER BY c2) AS 'C2 values'
FROM table
GROUP BY c1;
```

To retrieve a list of c1 values for which there exist specific values in another column c2, you need an `IN` clause specifying the c2 values and a `HAVING` clause specifying the required number of items in the list ...

```
SELECT c1
FROM table
WHERE c2 IN (1,2,3,4)
GROUP BY c1
HAVING COUNT(c2)=4;
```

This is easy to generalise to multiple column expressions, and a HAVING clause specifying any number of items from the IN list.

To list `c1` values that have *exactly* one instance of each `c2` value, add `DISTINCT` to the count criterion:

```
SELECT c1
FROM table
WHERE c2 IN (1,2,3,4)
GROUP BY c1
HAVING COUNT(DISTINCT c2)=4;
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Avoiding repeat aggregation

In a good introductory tutorial on MySQL subqueries, Jeremy Cole developed a triply nested query to retrieve the name, population, head of state, and number of official languages in countries with the most official languages. The query uses two tables from the MySQL `world` database:

```
CREATE TABLE country (
  Code char(3) NOT NULL DEFAULT '' PRIMARY KEY,
  Name char(52) NOT NULL DEFAULT '',
  Population int(11) NOT NULL DEFAULT '0',
  HeadOfState char(60) DEFAULT NULL,
  ... other columns ...
);
CREATE TABLE countrylanguage (
  CountryCode char(3) NOT NULL DEFAULT '' PRIMARY KEY,
  Language char(30) NOT NULL DEFAULT '',
  IsOfficial enum('T','F') NOT NULL DEFAULT 'F',
  Percentage float(4,1) NOT NULL DEFAULT '0.0'
);
```

The query needs to aggregate language counts by country twice: once for all language counts by country, and once again to identify countries with the highest number of languages:

```
SELECT name, population, headofstate, top.num
FROM Country
JOIN (
  SELECT countrycode, COUNT(*) AS num
  FROM CountryLanguage
  WHERE isofficial='T'
  GROUP BY countrycode
  HAVING num = (
    SELECT MAX(summary.nr_official_languages)
    FROM  (
      SELECT countrycode, COUNT(*) AS nr_official_languages
      FROM CountryLanguage
      WHERE isofficial='T'
      GROUP BY countrycode
    ) AS summary
  )
) as top ON Country.code=top.countrycode;
+--------------+------------+-------------+-----+
| name         | population | headofstate | num |
+--------------+------------+-------------+-----+
| Switzerland  |    7160400 | Adolf Ogi   |   4 |
| South Africa |   40377000 | Thabo Mbeki |   4 |
+--------------+------------+-------------+-----+
```

In addition, one of the nested subqueries is buried in a `HAVING` clause. This is fine with small tables, but if the table being aggregated is very large and the aggregation is complex, performance may be unsatisfactory. Substituting a temporary table for the double nesting can improve performance in two ways:

- the aggregation needs to be done just once
- we can use an exclusion join, which is usually faster than a `HAVING` clause, to find countries with the maximum counts:

```
DROP TABLE IF EXISTS top;
CREATE TABLE top ENGINE=MEMORY
  SELECT countrycode, COUNT(*) AS num
  FROM CountryLanguage l1
  WHERE isofficial='T'
  GROUP BY countrycode;

SELECT name,population,headofstate,t3.num
FROM country c
JOIN (
  SELECT t1.countrycode, t1.num
  FROM top t1
  LEFT JOIN top t2 ON t1.num < t2.num
  WHERE t2.countrycode IS NULL
) AS t3 ON c.code=t3.countrycode;
+--------------+------------+-------------+-----+
| name         | population | headofstate | num |
```

```
+--------------+------------+------------+-----+
| Switzerland  |    7160400 | Adolf Ogi  |   4 |
| South Africa |   40377000 | Thabo Mbeki |   4 |
+--------------+------------+------------+-----+
DROP TABLE top;
```

You notice that we haven't actually used a `TEMPORARY` table? Indeed we haven't, because of the MySQL limitation that temporary tables cannot be referenced multiple times in a query. Until that's lifted, we get almost as much speed improvement from using a `MEMORY` table as a temporary table.

Back to the top	Browse the book	Buy the book	Feedback

Cascading aggregates

When you have parent-child-grandchild tables, eg `companies, users, actions`, and your query requirement is for per-parent aggregates from the child table and per-child aggregates from the grandchild table, then cascading joins yield spuriously multiplied counts, and correlated subqueries fail because the second correlated subquery cannot find a visible joining column.

One solution is to use derived tables. Assuming ...

```
CREATE TABLE companies (id int, name char(10));
CREATE TABLE users (id INT,companyid INT);
CREATE TABLE actions (id INT, userid INT, date DATE);
```

then...

- Join `companies` & `users` once to establish a derived company-user table.
- Join them a second time, this time aggregating on `users.id` to generate user counts per company.
- Join the first derived table to the actions table, aggregating on `actions.id` to report actions per user per company:

Here is the SQL:

```
SELECT cu1.cid, cu1.cname, cu2.cid, cu2.uCnt, ua.aCnt
FROM (
  SELECT c.id AS cid, c.name AS cname, u1.id AS uid
  FROM companies c
  INNER JOIN users u1 ON u1.companyid=c.id
) AS cu1
INNER JOIN (
  SELECT c.id AS cid, COUNT(u2.id) AS uCnt
  FROM companies c
  INNER JOIN users u2 ON u2.companyid=c.id
  GROUP BY c.id
) AS cu2 ON cu1.cid=cu2.cid
INNER JOIN (
  SELECT u3.id AS uid, COUNT(a.id) AS aCnt
  FROM users u3
  INNER JOIN actions a ON a.userid=u3.id
  GROUP BY u3.id
) AS ua ON ua.uid=cu1.uid;
```

Back to the top	Browse the book	Buy the book	Feedback

Cross-aggregates

Given the table authorbook(authid INT, bookid INT), what query finds the books who have authors with more than one book in the table?

Even one level of recursion can induce a mild trance. Escape the trance by taking the problem one step at a time. First write the query that finds the authors with multiple books. Then join an outer query to that on authorid, and have the outer query select bookid:

```
SELECT a1.bookid
FROM authorbook a1
INNER JOIN (
  SELECT authid,count(bookid)
  FROM authorbook a2
  GROUP BY authid
  HAVING COUNT(bookid)>1
) AS a3 ON a1.authid=a3.authid;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Group data by datetime periods

To group rows by a time period whose length in minutes divides evenly into 60, use this formula:

```
GROUP BY ((60/periodMinutes) * HOUR( thistime ) + FLOOR( MINUTE( thistime ) / periodMinutes ))
```

where `thistime` is the `TIME` column and `periodMinutes` is the period length in minutes. So to group by 15-min periods, write ...

```
SELECT ...
GROUP BY ( 4 * HOUR( thistime ) + FLOOR( MINUTE( thistime ) / 15 ))
...
```

The same logic works for months ...

```
GROUP BY ((12/periodMonths) * YEAR( thisdate ) + FLOOR( MONTH( thisdate ) / periodMonths ))
```

It could be made to work for weeks with a function that maps the results of WEEK() to the range 1...52.

When the desired grouping period is a value returned by a MySQL date-time function, matters become simpler: just group by the desired value. Thus to group by weeks, write ..

```
SELECT ...
GROUP BY WEEK( datecol)
...
```

If there is no MySQL date-time function that returns the desired grouping period, you will need to write your own stored function.

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

League table

Here is a simple soccer league table setup that was developed in the MySQL Forum by J Williams and a contributor named "Laptop Alias". The `teams` table tracks team ID and name, the `games` table tracks home and away team IDs and goal totals for each game. The query for standings is built by aggregating a `UNION` of home team and away team game results:

```
DROP TABLE IF EXISTS teams, games;
CREATE TABLE teams(id int primary key auto_increment,tname char(32));
CREATE TABLE games(id int primary key auto_increment, date datetime,
                   hteam int, ateam int, hscore tinyint,ascore tinyint);

INSERT INTO teams VALUES(1,'Wanderers'),(2,'Spurs'),(3,'Celtics'),(4,'Saxons');
INSERT INTO games VALUES
(1,'2008-1-1 20:00:00',1,2,1,0),(2,'2008-1-1 20:00:00',3,4,0,2),
(3,'2008-1-8 20:00:00',1,3,1,1),(4,'2008-1-8 20:00:00',2,4,2,1);
SELECT * FROM teams;
+----+-----------+
| id | tname     |
+----+-----------+
|  1 | Wanderers |
```

```
|  2 | Spurs     |
|  3 | Celtics   |
|  4 | Saxons    |
+----+-----------+
SELECT * FROM games;
+----+---------------------+-------+-------+--------+--------+
| id | date                | hteam | ateam | hscore | ascore |
+----+---------------------+-------+-------+--------+--------+
|  1 | 2008-01-01 20:00:00 |     1 |     2 |      1 |      0 |
|  2 | 2008-01-01 20:00:00 |     3 |     4 |      0 |      2 |
|  3 | 2008-01-08 20:00:00 |     1 |     3 |      1 |      1 |
|  4 | 2008-01-08 20:00:00 |     2 |     4 |      2 |      1 |
+----+---------------------+-------+-------+--------+--------+

-- Standings query:
SELECT
  tname AS Team, Sum(P) AS P,Sum(W) AS W,Sum(D) AS D,Sum(L) AS L,
  SUM(F) as F,SUM(A) AS A,SUM(GD) AS GD,SUM(Pts) AS Pts
FROM(
  SELECT
    hteam Team,
    1 P,
    IF(hscore > ascore,1,0) W,
    IF(hscore = ascore,1,0) D,
    IF(hscore < ascore,1,0) L,
    hscore F,
    ascore A,
    hscore-ascore GD,
    CASE WHEN hscore > ascore THEN 3 WHEN hscore = ascore THEN 1 ELSE 0 END PTS
  FROM games
  UNION ALL
  SELECT
    ateam,
    1,
    IF(hscore < ascore,1,0),
    IF(hscore = ascore,1,0),
    IF(hscore > ascore,1,0),
    ascore,
    hscore,
    ascore-hscore GD,
    CASE WHEN hscore < ascore THEN 3 WHEN hscore = ascore THEN 1 ELSE 0 END
  FROM games
) as tot
JOIN teams t ON tot.Team=t.id
GROUP BY Team
ORDER BY SUM(Pts) DESC ;
+-----------+------+------+------+------+------+------+------+------+
| Team      | P    | W    | D    | L    | F    | A    | GD   | Pts  |
+-----------+------+------+------+------+------+------+------+------+
| Wanderers |    2 |    1 |    1 |    0 |    2 |    1 |    1 |    4 |
| Spurs     |    2 |    1 |    0 |    1 |    2 |    2 |    0 |    3 |
| Saxons    |    2 |    1 |    0 |    1 |    3 |    2 |    1 |    3 |
| Celtics   |    2 |    0 |    1 |    1 |    1 |    3 |   -2 |    1 |
+-----------+------+------+------+------+------+------+------+------+
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Sales commissions: double aggregation

Employees' sales commission rates increase as sales totals increase, according to specified bands of sales total amounts—like a graduated income tax in reverse. How do we automate sales commission calculations?

```
DROP TABLE IF EXISTS tblsales, tblcom;
CREATE TABLE tblsales(employeeID int,sales int);
INSERT INTO tblsales VALUES(1,2),(1,5),(1,7),(2,9),(2,15),(2,12);
```

```
SELECT * FROM tblsales;
+------------+-------+
| employeeID | sales |
+------------+-------+
|          1 |     2 |
|          1 |     5 |
|          1 |     7 |
|          2 |     9 |
|          2 |    15 |
|          2 |    12 |
+------------+-------+
CREATE TABLE tblcom(
  comstart DECIMAL(6,2),
  commend DECIMAL(6,2),
  comfactor DECIMAL(6,2),
  pct INT
);
INSERT INTO tblcom VALUES
(1.00,10.00,0.10,10),(11.00,20.00,0.20,20),(21.00,30.00,0.30,30),(31.00,40.00,0.40,40);
SELECT * FROM tblcom;
+----------+---------+-----------+------+
| comstart | commend | comfactor | pct  |
+----------+---------+-----------+------+
|     1.00 |   10.00 |      0.10 |   10 |
|    11.00 |   20.00 |      0.20 |   20 |
|    21.00 |   30.00 |      0.30 |   30 |
|    31.00 |   40.00 |      0.40 |   40 |
+----------+---------+-----------+------+
```

The first problem is to work out how commission ranges map to sales totals to determine base amounts for calculation of each part-commission. We assume the ranges are inclusive, ie a range that starts at 1 euro is meant to include that first euro:

- if amt < comstart, base amount = 0
- if amt <= commend, base amount = amt-comstart+1
- if amt > commend, base amount = commend - comstart+1

This is a nested IF():

```
IF(s.amt<c.comstart,0,IF(s.amt<=c.commend,s.amt-c.comstart,c.commend-c.comstart))
```

The second problem is how to apply every commission range row to every employee sales sum. That's a `CROSS JOIN` between aggregated sales and commissions:

```
SELECT *
FROM (
  SELECT employeeID,SUM(sales) AS amt
  FROM tblSales
  GROUP BY employeeID
) AS s
JOIN tblcom
ORDER BY s.employeeID;
+------------+------+----------+---------+-----------+------+
| employeeID | amt  | comstart | commend | comfactor | pct  |
+------------+------+----------+---------+-----------+------+
|          1 |   14 |     1.00 |   10.00 |      0.10 |   10 |
|          1 |   14 |    11.00 |   20.00 |      0.20 |   20 |
|          1 |   14 |    21.00 |   30.00 |      0.30 |   30 |
|          1 |   14 |    31.00 |   40.00 |      0.40 |   40 |
|          2 |   36 |    31.00 |   40.00 |      0.40 |   40 |
|          2 |   36 |     1.00 |   10.00 |      0.10 |   10 |
|          2 |   36 |    11.00 |   20.00 |      0.20 |   20 |
|          2 |   36 |    21.00 |   30.00 |      0.30 |   30 |
+------------+------+----------+---------+-----------+------+
```

Now check how the formula applies on every commission band for every sales total:

```
SELECT
  s.employeeID,s.amt,c.comstart,c.commend,
  IF(s.amt<=c.comstart,0,
     IF(s.amt<c.commend,s.amt-c.comstart+1,c.commend-c.comstart+1)
  ) AS base,
  c.comFactor AS ComPct,
  IF(s.amt<=c.comstart,0,
    IF(s.amt<c.commend,s.amt-c.comstart+1,c.commend-c.comstart+1)
  ) * comFactor AS Comm
FROM (
  SELECT employeeID,SUM(sales) AS amt
  FROM tblSales
  GROUP BY employeeID
) AS s
JOIN tblcom c
ORDER BY s.employeeID,comstart;
+------------+------+----------+---------+-------+--------+--------+
| employeeID | amt  | comstart | commend | base  | ComPct | Comm   |
+------------+------+----------+---------+-------+--------+--------+
|          1 |   14 |     1.00 |   10.00 | 10.00 |   0.10 | 1.0000 |
|          1 |   14 |    11.00 |   20.00 |  4.00 |   0.20 | 0.8000 |
|          1 |   14 |    21.00 |   30.00 |  0.00 |   0.30 | 0.0000 |
|          1 |   14 |    31.00 |   40.00 |  0.00 |   0.40 | 0.0000 |
|          2 |   36 |     1.00 |   10.00 | 10.00 |   0.10 | 1.0000 |
|          2 |   36 |    11.00 |   20.00 | 10.00 |   0.20 | 2.0000 |
|          2 |   36 |    21.00 |   30.00 | 10.00 |   0.30 | 3.0000 |
|          2 |   36 |    31.00 |   40.00 |  6.00 |   0.40 | 2.4000 |
+------------+------+----------+---------+-------+--------+--------+
```

Finally, `SUM` the formula results to aggregate commissions on aggregated sales:

```
SELECT
  s.employeeID,
  s.amt,
  SUM(IF(s.amt<=c.comstart,0,
        IF(s.amt<=c.commend,s.amt-c.comstart+1,c.commend-c.comstart+1)) * c.pct/100
  ) AS Comm
FROM (
  SELECT employeeID,SUM(sales) AS amt
  FROM tblSales
  GROUP BY employeeID
) AS s
JOIN tblcom c
GROUP BY s.employeeID;
+------------+------+----------+
| employeeID | amt  | Comm     |
+------------+------+----------+
|          1 |   14 | 1.800000 |
|          2 |   36 | 8.400000 |
+------------+------+----------+
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Show only one child row per parent row

Given tables `parent(id int not null primary key, etc...)` and `child (id int not null primary key, pid int not null references parent (id), etc...)`, how do we write a query that retrieves only one child row per `pid` even when the child table has multiple matching rows? MySQL permits use of `GROUP BY` even when the `SELECT` list specifies no aggregate function, so this will work:

```
select p.id, c.id
from parent p
join child c on p.id=c.pid
group by p.id;
```

But is it accurate? No, because it displays only the first c.pid value it happens to find. For further discussion see Within-group aggregates.

Back to the top	Browse the book	Buy the book	Feedback

Skip repeating values

You want to report all unique values of a column and skip all rows repeating any of these values.

```
SELECT col
FROM foo
GROUP BY col
```

Back to the top	Browse the book	Buy the book	Feedback

Within-group aggregates

You have a `products` table with columns `item, supplier, price`. Multiple suppliers offer various prices for the same item. You need to find the supplier with the lowest price for each item.

```
DROP TABLE IF EXISTS products;
CREATE TABLE products(item int,supplier int,price decimal(6,2));
INSERT INTO products VALUES(1,1,10),(1,2,15),(2,2,20),(2,1,21),(2,2,18);
SELECT * FROM products;
+------+----------+-------+
| item | supplier | price |
+------+----------+-------+
|    1 |        1 | 10.00 |
|    1 |        2 | 15.00 |
|    2 |        2 | 20.00 |
|    2 |        1 | 21.00 |
|    2 |        2 | 18.00 |
+------+----------+-------+
```

Your first thought may be to `GROUP BY item`, but that is not guaranteed to return the correct `supplier` value for each minimum item price. Grouping by both `item` and `supplier` will return more information than you want. Nor can you write `WHERE price=MIN(...)` because the query engine will evaluate the `WHERE` clause before it knows the `MIN` value.

This is the problem of *aggregating within aggregates*. It is sometimes called the 'groupwise aggregates' problem, but the term 'groupwise' is ambiguous at best, so we think better names for it are *subaggregates*, *inner aggregates*, or *within-group aggregates*.

It's easy to show that the within-group aggregates problem is a form of the problem of returning values from non-grouping columns in an aggregate query. Suppose you write ...

```
SELECT item,supplier,MIN(price)
FROM products
GROUP BY item;
```

Will this reliably return the correct supplier per item? No. *Unless there is exactly one supplier per item, the supplier value returned will be arbitrary.* To retrieve the correct supplier for each item, you need more logic.

The simplest and often best-performing solution to the within-aggregates problem is an *outer self exclusion join*...

```
SELECT p1.item,p1.supplier,p1.price
FROM products AS p1
LEFT JOIN products AS p2 ON p1.item  = p2.item AND p1.price > p2.price
WHERE p2.id IS NULL;
```

...because in the resultset built by joining on left item = right item and left price > right price, the left-sided rows for which

there is no greater right-sided price are precisely the per-item rows with the smallest prices.

You can also accomplish this by building a table of aggregated minimum prices. Before MySQL 4.1, it has to be a temporary table:

```
CREATE TEMPORARY TABLE tmp (
  item INT,
  minprice DECIMAL DEFAULT 0.0
);
LOCK TABLES products READ;
INSERT INTO tmp
  SELECT item, MIN(price)
  FROM products
  GROUP BY item;
```

to which you then join the products table:

```
SELECT products.item, supplier, products.price
FROM products
JOIN tmp ON products.item = tmp.item
WHERE products.price=tmp.minprice;
UNLOCK TABLES;
DROP TABLE tmp;
```

From MySQL 4.1 on, the temporary table can be a correlated subquery. This is the most intuitively obvious syntax for the problem. Often it's also the *slowest* solution:

```
SELECT item, supplier, price
FROM products AS p1
WHERE price = (
  SELECT MIN(p2.price)
  FROM products AS p2
  WHERE p1.item = p2.item
);
```

Another solution, sometimes the fastest of all, is to move the aggregating subquery from the `WHERE` clause to the `FROM` clause:

```
SELECT p.item, p.supplier, p.price
FROM products AS p
JOIN (
  SELECT item, MIN(price) AS minprice
  FROM products
  GROUP BY item
) AS pm ON p.item = pm.item AND p.price = pm.minprice;
```

But these examples involve small tables. Run `EXPLAIN EXTENDED` on the query to see dangers lurking in the weeds:

```
+----+-------------+------------+------+---------------+------+---------+------+------+----------+----------------------
| id | select_type | table      | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra
+----+-------------+------------+------+---------------+------+---------+------+------+----------+----------------------
|  1 | PRIMARY     | <derived2> | ALL  | NULL          | NULL | NULL    | NULL |    2 |   100.00 |
|  1 | PRIMARY     | p          | ALL  | NULL          | NULL | NULL    | NULL |    5 |   100.00 | Using where; Using joi
|  2 | DERIVED     | products   | ALL  | NULL          | NULL | NULL    | NULL |    5 |   100.00 | Using temporary; Using
+----+-------------+------------+------+---------------+------+---------+------+------+----------+----------------------
```

Yikes. Filesorts are going to be slow with large tables. What happens when we give the table a primary key `(item,supplier,price)`?

```
+----+-------------+------------+-------+---------------+---------+---------+---------+------+----------+----------------
| id | select_type | table      | type  | possible_keys | key     | key_len | ref     | rows | filtered | Extra
+----+-------------+------------+-------+---------------+---------+---------+---------+------+----------+----------------
|  1 | PRIMARY     | <derived2> | ALL   | NULL          | NULL    | NULL    | NULL    |    2 |   100.00 |
|  1 | PRIMARY     | p          | ref   | PRIMARY       | PRIMARY | 4       | pm.item |    2 |   100.00 | Using where; Us
|  2 | DERIVED     | products   | index | NULL          | PRIMARY | 11      | NULL    |    5 |   100.00 | Using index
+----+-------------+------------+-------+---------------+---------+---------+---------+------+----------+----------------
```

That's more like it.

Try all solutions to find which is fastest for your version of the problem.

To find more than one value per group, you might think the `LIMIT` clause would work, but `LIMIT` is limited in subqueries. See Within-group quotas.

Back to the top *Browse the book* *Buy the book* *Feedback*

Within-group aggregates with a wrinkle

We have a wages table holding wage rates by waiter and startdate, and a tips table which tracks hours worked and tips received per waiter per day. The requirement is to report wages and concurrent tips per waiter per day.

```
DROP TABLE IF EXISTS wages,tips;
CREATE TABLE wages( id int, waiter int, start date, rate decimal(6,2));
INSERT INTO wages VALUES
( 1, 4, '2005-01-01', 5.00 ),
( 2, 4, '2005-03-01', 6.00 ),
( 3, 5, '2007-01-05', 7.00 ),
( 4, 5, '2008-03-20', 8.00 ),
( 5, 5, '2008-04-01', 9.00 );
CREATE TABLE tips(
  id int,
  date date,
  waiter int,
  hours_worked smallint,
  tabs smallint,
  tips decimal(6,2)
);
INSERT INTO tips VALUES
( 1, '2008-02-29', 4, 6.50, 21, 65.25 ),
( 2, '2008-03-06', 5, 6.00, 15, 51.75 ),
( 3, '2008-03-21', 4, 2.50, 5, 17.85 ),
( 4, '2008-03-22', 5, 5.25, 10, 39.00 );
SELECT * FROM wages;
+------+--------+------------+------+
| id   | waiter | start      | rate |
+------+--------+------------+------+
|    1 |      4 | 2005-01-01 | 5.00 |
|    2 |      4 | 2005-03-01 | 6.00 |
|    3 |      5 | 2007-01-05 | 7.00 |
|    4 |      5 | 2008-03-20 | 8.00 |
|    5 |      5 | 2008-04-01 | 9.00 |
+------+--------+------------+------+
SELECT * FROM tips;
+------+------------+--------+--------------+------+-------+
| id   | date       | waiter | hours_worked | tabs | tips  |
+------+------------+--------+--------------+------+-------+
|    1 | 2008-02-29 |      4 |            7 |   21 | 65.25 |
|    2 | 2008-03-06 |      5 |            6 |   15 | 51.75 |
|    3 | 2008-03-21 |      4 |            3 |    5 | 17.85 |
|    4 | 2008-03-22 |      5 |            5 |   10 | 39.00 |
+------+------------+--------+--------------+------+-------+
```

For the above dataset, the result which correctly matches wages and tips would be:

```
+------+------------+------+------+-------+------+--------+------+------------+
| tid  | Date       | Hrs  | tabs | tips  | wid  | waiter | rate | start      |
+------+------------+------+------+-------+------+--------+------+------------+
|    1 | 2008-02-29 |    7 |   21 | 65.25 |    2 |      4 | 6.00 | 2005-03-01 |
|    2 | 2008-03-06 |    6 |   15 | 51.75 |    3 |      5 | 7.00 | 2007-01-05 |
|    3 | 2008-03-21 |    3 |    5 | 17.85 |    2 |      4 | 6.00 | 2005-03-01 |
|    4 | 2008-03-22 |    5 |   10 | 39.00 |    4 |      5 | 8.00 | 2008-03-20 |
```

```
+------+------------+------+------+-------+------+--------+------+------------+
```

Why is this different from an ordinary within-groups aggregate? The correct wage rate for a tips row is not the wages row for that waiter with the latest date; it is the wages row *having the latest date before the date in the given tips row*.

One way to proceed is to build a temporary table from a join of wages to tips on waiter and `wages.start < tips.date`, then exclusion-join that result to itself to remove all rows except those with the latest wage rate per tips row. A two-step:

```
-- wages-tips join removing later wage changes:
DROP TABLE IF EXISTS tmp;
CREATE TABLE tmp
SELECT
  t.id AS tid, t.date AS Date, t.hours_worked AS Hrs,t.tabs,t.tips,
  w.id AS wid, w.waiter, w.rate, w.start
FROM tips t
JOIN wages w ON w.waiter=t.waiter AND w.start<=t.date;
-- self-exclusion join to remove obsolete wage rows:
SELECT t1.*
FROM tmp t1
LEFT JOIN tmp t2 ON t1.tid=t2.tid and t1.start<t2.start
WHERE t2.waiter is null
ORDER BY t1.Date;
+------+------------+------+------+-------+------+--------+------+------------+
| tid  | Date       | Hrs  | tabs | tips  | wid  | waiter | rate | start      |
+------+------------+------+------+-------+------+--------+------+------------+
|    1 | 2008-02-29 |    7 |   21 | 65.25 |    2 |      4 | 6.00 | 2005-03-01 |
|    2 | 2008-03-06 |    6 |   15 | 51.75 |    3 |      5 | 7.00 | 2007-01-05 |
|    3 | 2008-03-21 |    3 |    5 | 17.85 |    2 |      4 | 6.00 | 2005-03-01 |
|    4 | 2008-03-22 |    5 |   10 | 39.00 |    4 |      5 | 8.00 | 2008-03-20 |
+------+------------+------+------+-------+------+--------+------+------------+
DROP TABLE tmp;
```

That's fine, but can we skip the temp table? Yes—by adding the condition *wages.start* <= *tips.date* to each side of the exclusion join:

```
SELECT
  t.id AS tid, t.date, t.hours_worked AS Hrs,t.tabs,t.tips,
  w.id AS wid, w.waiter, w.rate, w.start
FROM tips t
JOIN wages w ON w.waiter=t.waiter AND w.start <= t.date
LEFT JOIN wages w2 ON w.waiter=w2.waiter AND w2.start<=t.date AND w.start<w2.start
WHERE w2.id IS NULL
ORDER BY t.date;
```

Much simpler, and it gives the same result as the two-step.

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Within-group quotas (Top N per group)

A table has multiple rows per key value, and you need to retrieve, say, the first or earliest two rows per key.

If the groups are fairly small, this can be done efficiently with a self-join and counts. For example the following table (based on a tip by Rudy Limeback) has three small data groups:

```
DROP TABLE IF EXISTS test;
CREATE TABLE test (
  id INT,
  entrydate DATE
);
INSERT INTO test VALUES
( 1, '2007-5-01' ),
( 1, '2007-5-02' ),
( 1, '2007-5-03' ),
```

```
( 1, '2007-5-04' ),
( 1, '2007-5-05' ),
( 1, '2007-5-06' ),
( 2, '2007-6-01' ),
( 2, '2007-6-02' ),
( 2, '2007-6-03' ),
( 2, '2007-6-04' ),
( 3, '2007-7-01' ),
( 3, '2007-7-02' ),
( 3, '2007-7-03' );
```

The first two rows per `ID` are the rows which, for a given `ID`, have two or fewer rows with earlier dates. If we use an *inequality join* with the `COUNT(*)` function to find the earlier rows per `ID` ...

```
SELECT t1.id, t1.entrydate, COUNT(*) AS earlier
FROM test AS t1
JOIN test AS t2 ON t1.id=t2.id AND t1.entrydate >= t2.entrydate
GROUP BY t1.id, t1.entrydate
+------+------------+---------+
| id   | entrydate  | earlier |
+------+------------+---------+
|    1 | 2007-05-01 |       1 |
|    1 | 2007-05-02 |       2 |
|    1 | 2007-05-03 |       3 |
|    1 | 2007-05-04 |       4 |
|    1 | 2007-05-05 |       5 |
|    1 | 2007-05-06 |       6 |
|    2 | 2007-06-01 |       1 |
|    2 | 2007-06-02 |       2 |
|    2 | 2007-06-03 |       3 |
|    2 | 2007-06-04 |       4 |
|    3 | 2007-07-01 |       1 |
|    3 | 2007-07-02 |       2 |
|    3 | 2007-07-03 |       3 |
+------+------------+---------+
```

... then we get our result immediately by removing rows where the 'earlier' count exceeds 2:

```
SELECT t1.id, t1.entrydate, count(*) AS earlier
FROM test AS t1
JOIN test AS t2 ON t1.id=t2.id AND t1.entrydate >= t2.entrydate
GROUP BY t1.id, t1.entrydate
HAVING earlier <= 2;
+------+------------+---------+
| id   | entrydate  | earlier |
+------+------------+---------+
|    1 | 2007-05-01 |       1 |
|    1 | 2007-05-02 |       2 |
|    2 | 2007-06-01 |       1 |
|    2 | 2007-06-02 |       2 |
|    3 | 2007-07-01 |       1 |
|    3 | 2007-07-02 |       2 |
+------+------------+---------+
```

This works beautifully with smallish aggregates. But the query algorithm compares every within-group row to every other within-group row. As the size N of a group increases, execution time increases by N*N. If the query takes one minute for groups of 1,000, it will take 16 minutes for groups of 4,000, and more than four hours for groups for 16,000. *The solution does not scale.*

What to do? Forget `GROUP BY`! Manually assemble the desired query results in a temporary table from simple indexed queries, in this case, two rows per `ID`:

```
DROP TEMPORARY TABLE IF EXISTS earliers;
CREATE TEMPORARY TABLE earliers( id INT, entrydate DATE);
INSERT INTO earliers
  SELECT id,entrydate FROM test WHERE id=1 ORDER BY entrydate LIMIT 2;
```

```
INSERT INTO earliers
  SELECT id,entrydate FROM test WHERE id=2 ORDER BY entrydate LIMIT 2;
INSERT INTO earliers
  SELECT id,entrydate FROM test WHERE id=3 ORDER BY entrydate LIMIT 2;
```

You need one `INSERT` statement per grouping value. To print the result, just query the `earliers` table:

```
SELECT * FROM earliers
ORDER BY id, entrydate;
+------+------------+
| id   | entrydate  |
+------+------------+
|    1 | 2007-05-01 |
|    1 | 2007-05-02 |
|    2 | 2007-06-01 |
|    2 | 2007-06-02 |
|    3 | 2007-07-01 |
|    3 | 2007-07-02 |
+------+------------+
DROP TEMPORARY TABLE earliers;
```

Most useful reports run again and again. If that's the case for yours, automate it in a stored procedure: using a cursor and a prepared statement, auto-generate an `INSERT` statement for every grouping value, and return the result:

```
DROP PROCEDURE IF EXISTS listearliers;
DELIMITER |
CREATE PROCEDURE listearliers()
BEGIN
  DECLARE curdone, vid INT DEFAULT 0;
  DECLARE idcur CURSOR FOR SELECT DISTINCT id FROM test;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET curdone = 1;
  DROP TEMPORARY TABLE IF EXISTS earliers;
  CREATE TEMPORARY TABLE earliers( id INT, entrydate DATE);
  SET @sql = 'INSERT INTO earliers SELECT id,entrydate FROM test WHERE id=? ORDER BY  entrydate LIMIT 2';
  OPEN idcur;
  REPEAT
    FETCH idcur INTO vid;
    IF NOT curdone THEN
      BEGIN
        SET @vid = vid;
        PREPARE stmt FROM @sql;
        EXECUTE stmt USING @vid;
        DROP PREPARE stmt;
      END;
    END IF;
  UNTIL curdone END REPEAT;
  CLOSE idcur;
  SELECT * FROM earliers ORDER BY id,entrydate;
  DROP TEMPORARY TABLE earliers;
END;
|
DELIMITER ;
CALL listearliers();
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Average the top 50% values per group

Each row of a `games` table records one game score for a team:

```
DROP TABLE IF EXISTS games;
CREATE TABLE games(id INT, teamID INT, score INT);
INSERT INTO games VALUES
  (1,1,3),(2,1,4),(3,1,5),(4,1,6),(5,2,6),
  (6,2,7),(7,2,8),(8,2,7),(9,2,6),(10,2,7);
```

How would we write a query that returns the average of the top 50% of scores per team?

The per-team *median* value is its middle value--lower than the highest 50% and higher than the lowest 50% of values for that team--so a shortcut is to query the team medians, then aggregate on a join that selects per-team scores above the medians.

How to find per-team medians? If a resultset has an odd number of rows, at least one row has the true median score. If it has an even number of rows, the median score is an average of two central values. The following query adapts Joe Celko's formula in "SQL for Smarties" averaging "low" and "high" medians:

```
DROP TABLE IF EXISTS medians;
CREATE TABLE medians
SELECT p1.teamid, AVG(P1.score) AS median
FROM games AS P1, games AS P2
WHERE p1.teamid=p2.teamid
GROUP BY p1.teamid
HAVING (
  SUM(CASE WHEN P2.score <= P1.score THEN 1 ELSE 0 END) >= ((COUNT(*) + 1) / 2)
  AND
  SUM(CASE WHEN P2.score >= P1.score THEN 1 ELSE 0 END) >= (COUNT(*)/2 + 1)
)
OR (
  SUM(CASE WHEN P2.score >= P1.score THEN 1 ELSE 0 END) >= ((COUNT(*) + 1) / 2)
  AND
  SUM(CASE WHEN P2.score <= P1.score THEN 1 ELSE 0 END) >= (COUNT(*)/2 + 1)
);
+--------+--------+
| teamid | median |
+--------+--------+
|      1 | 4.5000 |
|      2 | 6.8333 |
+--------+--------+
```

Now join `games` to `medians` accepting only top-half values:

```
SELECT g.teamid, AVG(g.score) AS Top50Avg
FROM games g
JOIN medians m ON g.teamid = m.teamid AND g.score >= m.median
GROUP BY g.teamid
ORDER BY Top50Avg DESC;
+--------+----------+
| teamid | Top50Avg |
+--------+----------+
|      2 |   7.2500 |
|      1 |   5.5000 |
+--------+----------+
DROP TABLE medians;
```

Yes, all the logic can be moved into one query:

```
SELECT g.teamid, AVG(g.score) AS Top50Avg
FROM games g
JOIN (
  SELECT p1.teamid, AVG(P1.score) AS median
  FROM games AS P1, games AS P2
  WHERE p1.teamid=p2.teamid
  GROUP BY p1.teamid
  HAVING (
    SUM(CASE WHEN P2.score <= P1.score THEN 1 ELSE 0 END) >= ((COUNT(*) + 1) / 2)
    AND
    SUM(CASE WHEN P2.score >= P1.score THEN 1 ELSE 0 END) >= (COUNT(*)/2 + 1)
  )
  OR (
    SUM(CASE WHEN P2.score >= P1.score THEN 1 ELSE 0 END) >= ((COUNT(*) + 1) / 2)
    AND
    SUM(CASE WHEN P2.score <= P1.score THEN 1 ELSE 0 END) >= (COUNT(*)/2 + 1)
```

```
  )
) AS m ON g.teamid = m.teamid AND g.score >= m.median
GROUP BY g.teamid
ORDER BY Top50Avg DESC;
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Averages from bands of values

To count and average scores in bands of 10, ie 0-9,10-19 etc:

```
create table scores(score int);
insert into scores values(5),(15),(25),(35);

SELECT 10  * FLOOR( score / 10  ) AS  Bottom,
       10  * FLOOR( score / 10  ) + 9 AS Top,
       Count( score ),
       Avg( score )
FROM scores
GROUP BY 10  * FLOOR( score / 10  );
+--------+------+---------------+-------------+
| Bottom | Top  | Count( score ) | Avg( score ) |
+--------+------+---------------+-------------+
|      0 |    9 |              1 |      5.0000 |
|     10 |   19 |              1 |     15.0000 |
|     20 |   29 |              1 |     25.0000 |
|     30 |   39 |              1 |     35.0000 |
+--------+------+---------------+-------------+
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Count unique values of one column

```
SELECT col_name, COUNT(*) AS frequency
FROM tbl_name
GROUP by col_name
ORDER BY frequency DESC;
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Median

Statistically, the median is the middle value--the value that is smaller than that found in half of all remaining rows, and larger than that found in the other half:

```
SELECT l1.hours As Median
FROM BulbLife As l1, bulbLife AS l2
GROUP BY l1.Hours
HAVING SUM(CASE WHEN l2.hours <= l1.hours THEN 1 ELSE 0 END) >= (COUNT(*)+1) / 2
   AND SUM(CASE WHEN l2.hours >= l1.hours THEN 1 ELSE 0 END) >= (COUNT(*)/2) + 1;
```

An anonymous reader pointed out that this will cost O(N*N), ie it does not scale, so we posted a MySQL implementation of Torben Mogenson's algorithm for calculating the median (http://ndevilla.free.fr/median/median/node20.html), which is said to be amongst the fastest. It also proved too slow. Now Joe Wynne has offered an algorithm which appears to be correct, and which does scale. Here it is as a MySQL stored procedure:

```
DROP PROCEDURE IF EXISTS Median;
DELIMITER |
```

```
CREATE PROCEDURE Median( tbl CHAR(64), col CHAR(64), OUT res DOUBLE )
BEGIN
  DECLARE arg CHAR(64);
  SET @sql = CONCAT( 'SELECT ((COUNT(*))/2) INTO @c FROM ', tbl );
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
  DROP PREPARE stmt;
  SET @a = CONVERT(FLOOR(@c), SIGNED);
  IF @a = @c THEN
    BEGIN
      SET @a = @a-1;
      SET @b = 2;
      SET arg = CONCAT( 'AVG(', col, ')' );
    END;
  ELSE
    BEGIN
      SET @b = 1;
      SET arg = col;
    END;
  END IF;
  SET @sql = CONCAT('SELECT ', arg, ' INTO @res FROM (SELECT ', col, ' FROM ', tbl,
                  ' ORDER BY ', col, ' LIMIT ?,?) as tmp');
  PREPARE stmt FROM @sql;
  EXECUTE stmt USING @a, @b;
  DROP PREPARE stmt;
  SET res=@res;
END;
|
DELIMITER ;
```

Why don't we make it a function? Because MySQL functions do not (yet?) allow dynamic SQL.

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Mode

Statistically, the mode is the most frequently occurring value. Given tables parent(id int) and child(pid int, cid int), where child.pid references parent.id as a foreign key, what query finds the parent.id most often represented in the child id, that is, the modal count of child.pid?

```
SELECT pid, COUNT(*) AS frequency
FROM child
GROUP BY pid
ORDER BY frequency DESC
LIMIT 1;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Rank order

Without MSSQL's `RANK()` aggregate function, how do we display rank order in a MySQL query, for example from a table like this?

```
CREATE TABLE votes( name CHAR(10), votes INT );
INSERT INTO votes VALUES
  ('Smith',10),('Jones',15),('White',20),('Black',40),('Green',50),('Brown',20);
```

The query is a two-step:
1. Join the table to itself on the value to be ranked, handling ties
2. Group and order the result of the self-join on rank:

```
SELECT v1.name, v1.votes, COUNT(v2.votes) AS Rank
```

```
FROM votes v1
JOIN votes v2 ON v1.votes < v2.votes OR (v1.votes=v2.votes and v1.name = v2.name)
GROUP BY v1.name, v1.votes
ORDER BY v1.votes DESC, v1.name DESC;
+-------+-------+------+
| name  | votes | Rank |
+-------+-------+------+
| Green |    50 |    1 |
| Black |    40 |    2 |
| White |    20 |    3 |
| Brown |    20 |    3 |
| Jones |    15 |    5 |
| Smith |    10 |    6 |
+-------+-------+------+
```

Back to the top | *Browse the book* | *Buy the book* | *Feedback*

Backslashes in data

Backslashes multiply weirdly:

```
SELECT 'a\b' RLIKE 'a\b';
```

returns 1, as does...

```
SELECT 'a\\b' RLIKE 'a\\\\b';
```

because in a pair of backslashes, the second is not escaped by the first, so to compare two literals you double each backslash in the `RLIKE` argument. But if you are querying a table for such a string from the MySQL client, this doubling happens twice--once in the client, and once in the database--so to find a *column* value matching `'a\\b'`, you need to write...

```
SELECT desc FROM xxx WHERE desc RLIKE 'aa\\\\\\\\bb';
```

That's *eight* backslashes to match two!

Back to the top | *Browse the book* | *Buy the book* | *Feedback*

Compare data in two tables

This query UNIONs queries for matching column names from two tables, and keeps just those rows which occur once in the union. Those are the rows unique to one table or the other. Customise your column list { id, col1, col2, col3 ...} as desired.

```
SELECT
  MIN(TableName) as TableName, id, col1, col2, col3, ...
FROM (
  SELECT 'Table a' as TableName, a.id, a.col1, a.col2, a.col3, ...
  FROM a
  UNION ALL
  SELECT 'Table b' as TableName, b.id, b.col1, b.col2, b.col3, ...
  FROM b
) AS tmp
GROUP BY id, col1, col2, col3, ...
HAVING COUNT(*) = 1
ORDER BY ID;
```

Back to the top | *Browse the book* | *Buy the book* | *Feedback*

Show rows where column value changed

SQL is set-oriented, but it can solve row-by-row problems. Suppose you need to retrieve only the rows that differ from

immediatel previous rows given some ordering spec:

```
drop table if exists t;
create table t (
  p char(3),
  d date
);
insert into t values
('50%','2008-05-01'),
('30%','2008-05-02'),
('30%','2008-05-03'),
('50%','2008-05-04'),
('50%','2008-05-05'),
('20%','2008-05-06'),
('20%','2008-05-07'),
('50%','2008-05-08'),
('70%','2008-05-09'),
('70%','2008-05-10');
select * from t order by d;
+------+------------+
| p    | d          |
+------+------------+
| 50%  | 2008-05-01 | *
| 30%  | 2008-05-02 | *
| 30%  | 2008-05-03 |
| 50%  | 2008-05-04 | *
| 50%  | 2008-05-05 |
| 20%  | 2008-05-06 | *
| 20%  | 2008-05-07 |
| 50%  | 2008-05-08 | *
| 70%  | 2008-05-09 | *
| 70%  | 2008-05-10 |
+------+------------+
```

We want to retrieve only rows whose `p` values differ from immediately previous values (marked by * above). As with running sums. we get the desired listing by tracking row-to-row value changes with user variables:

```
set @p='';
set @d='';
select p 'Pct Changed',d Date from (
  select
    p,
    if( p<>@p, d, @d ) as d,
    @p:=p,
    @d:=d
  from t
  order by d
) as t
group by d;
+-------------+------------+
| Pct Changed | Date       |
+-------------+------------+
| 50%         | 2008-05-01 |
| 30%         | 2008-05-02 |
| 50%         | 2008-05-04 |
| 20%         | 2008-05-06 |
| 50%         | 2008-05-08 |
| 70%         | 2008-05-09 |
+-------------+------------+
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Age in years

Given a birthdate in @dob, here is a simple formula for age in years:

```
DATE_FORMAT(FROM_DAYS(TO_DAYS(now()) - TO_DAYS(@dob)), '%Y') + 0;
```

and here is one for age in years to two decimal places (ignoring day of month):

```
ROUND((((YEAR(now()) - YEAR(@dob)))*12 + (((MONTH(now()) - MONTH(@dob)))))/12, 2)
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Appointments available

Given a clinic of physicians, patients and appointments, how to find an available appointment time for a given physician?

This is a variant of the [Not] Exists query pattern. Though we can write it with subqueries, performance will be crisper with a join. But finding data that is not there requires a join to data which *is* there. So in addition to tables for appointments, doctors and patients, we need a table of all possible appointment datetimes. Here's a schema illustrating the idea ...

```
CREATE TABLE a_dt (         -- POSSIBLE APPOINTMENT DATES AND TIMES
  d DATE,
  t TIME
);
CREATE TABLE a_drs (        -- DOCTORS
  did INT                   -- doctor id
);
CREATE TABLE a_pts (        -- PATIENTS
  pid INT
);
CREATE TABLE a_appts (      -- APPOINTMENTS
  aid INT,                  -- appt id
  did INT,                  -- doctor id
  pid INT,                  -- patient id
  d DATE,
  t TIME
);
```

Now we can apply the [Not] Exists query pattern. To find free appointment datetimes for a given doctor in a given datetime range, we left join possible appointments to existing appointments on date and time and doctor, add Where conditions for desired appointment datetimes, and finally add a Where condition that the appointment slot be null, i.e. free...

```
SELECT d.did, a.d, a.t
FROM a_dt AS a
LEFT JOIN a_appts AS ap USING (d,t)
LEFT JOIN a_drs AS d
   ON a.d = ap.d
  AND a.t = ap.t
  AND ap.did = d.did
  AND ap.did = 1
WHERE a.d BETWEEN desired_start_date AND desired_end_date
  AND a.t BETWEEN desired_start_time AND desired_end_time
  AND ap.aid IS NULL;
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Count business days between two dates

Given a table named `dates` with date columns `d1,d2`, if the two dates are in the same year, the solution is simply the date difference in days minus the date difference in weeks:

```
SELECT d1, d2, DATEDIFF(d2, d1) - (WEEK(d2) - WEEK(d1)) * 2 AS BizDays
FROM dates
ORDER BY d1, d2;
```

For dates that span different years, week numbers won't work. The answer the number of raw days, minus twice the number of whole weeks (because there are two weekend days/week), minus the number of weekend days in any remainder part-week. This algorithm works when the start and stop dates are themselves business days (but needs refinement to work when passed weekend dates--anybody want to try?):

```
SET @d1='2007-1-1';
SET @d2='2007-3-31';
SET @dow1 = DAYOFWEEK(@d1);
SET @dow2 = DAYOFWEEK(@d2);
SET @days = DATEDIFF(@d2,@d1);
SET @wknddays = 2 * FLOOR( @days / 7 ) +
                IF( @dow1 = 1 AND @dow2 > 1, 1,
                    IF( @dow1 = 7 AND @dow2 = 1, 1,
                        IF( @dow1 > 1 AND @dow1 > @dow2, 2,
                            IF( @dow1 < 7 AND @dow2 = 7, 1, 0 )
                          )
                      )
                  );
SELECT FLOOR(@days - @wkndDays) AS BizDays;
```

The algorithm is easily encapsulated in a function:

```
DROP FUNCTION IF EXISTS BizDayDiff;
DELIMITER |
CREATE FUNCTION BizDayDiff( d1 DATE, d2 DATE )
RETURNS INT
DETERMINISTIC
BEGIN
  DECLARE dow1, dow2, days, wknddays INT;
  SET dow1 = DAYOFWEEK(d1);
  SET dow2 = DAYOFWEEK(d2);
  SET days = DATEDIFF(d2,d1);
  SET wknddays = 2 * FLOOR( days / 7 ) +
                IF( dow1 = 1 AND dow2 > 1, 1,
                    IF( dow1 = 7 AND dow2 = 1, 1,
                        IF( dow1 > 1 AND dow1 > dow2, 2,
                            IF( dow1 < 7 AND dow2 = 7, 1, 0 )
                          )
                      )
                  );
  RETURN FLOOR(days - wkndDays);
END;
|
DELIMITER ;
```

To include time in the difference, you would probably adopt the convention of returning a string like `N days hh:mm:ss` where `N` is the date difference calculated above, minus one if the time portion of `d1` is later than that of `d2`:

```
DROP FUNCTION IF EXISTS BizDateTimeDiff;
DELIMITER |
CREATE FUNCTION BizDateTimeDiff( d1 DATETIME, d2 DATETIME )
RETURNS CHAR(30)
DETERMINISTIC
BEGIN
  DECLARE dow1, dow2, days, wknddays INT;
  DECLARE tdiff CHAR(10);
  SET dow1 = DAYOFWEEK(d1);
  SET dow2 = DAYOFWEEK(d2);
  SET tdiff = TIMEDIFF( TIME(d2), TIME(d1) );
  SET days = DATEDIFF(d2,d1);
  SET wknddays = 2 * FLOOR( days / 7 ) +
                IF( dow1 = 1 AND dow2 > 1, 1,
                    IF( dow1 = 7 AND dow2 = 1, 1,
                        IF( dow1 > 1 AND dow1 > dow2, 2,
                            IF( dow1 < 7 AND dow2 = 7, 1, 0 )
                          )
```

```
                        )
                    );
  SET days = FLOOR(days - wkndDays) - IF( ASCII(tdiff) = 45, 1, 0 );
  SET tdiff = IF( ASCII(tdiff) = 45, TIMEDIFF( '24:00:00', SUBSTRING(tdiff,2)), TIMEDIFF( tdiff, '00:00:00' ));
  RETURN CONCAT( days, ' days ', tdiff );
END;
|
DELIMITER ;
SELECT BizDateTimeDiff( '2007-1-1 00:00:00', '2007-3-31 00:00:00' ) AS dtdiff;
+------------------+
| dtdiff           |
+------------------+
| 64 days 00:00:00 |
+------------------+
SELECT BizDateTimeDiff( '2007-1-1 11:00:00', '2007-3-31 00:00:00' ) AS dtdiff;
+------------------+
| dtdiff           |
+------------------+
| 63 days 13:00:00 |
+------------------+
SELECT BizDateTimeDiff( '2007-1-1 12:00:00', '2007-3-31 13:00:00' ) AS dtdiff;
+------------------+
| dtdiff           |
+------------------+
| 64 days 01:00:00 |
+------------------+
SELECT BizDateTimeDiff( '2007-1-1 00:00:00', '2007-3-31 11:00:00' ) AS dtdiff;
+------------------+
| dtdiff           |
+------------------+
| 64 days 11:00:00 |
+------------------+
```

To factor in national and religious holidays, you need a *holidays* table and a stored procedure that adds in the number of holidays between *d1* and *d2*.

Back to the top *Browse the book* *Buy the book* *Feedback*

Count Tuesdays between two dates

Date arithmetic is deceptively hard. One way to appreciate the difficulties is to read Chapter 21 in our book. Another is to try to calculate the number of Tuesdays (or another weekday) between two dates. It's not a back-of-the-napkin problem.

An earlier formula we had for this problem sometimes gave incorrect results. As a debugging aid, we wrote a brute force calculator for the problem:

```
SET GLOBAL log_bin_trust_function_creators=1;
DROP FUNCTION IF EXISTS DayCount;
DELIMITER |

CREATE FUNCTION DayCount( d1 DATE, d2 DATE, daynum SMALLINT ) RETURNS INT
BEGIN
  DECLARE days INT DEFAULT 0;
  IF D1 IS NOT NULL AND D2 IS NOT NULL THEN
    WHILE D1 <= d2 DO
      BEGIN
        IF DAYOFWEEK(d1) = daynum THEN
          SET days=days+1;
        END IF;
        SET d1 = ADDDATE(d1, INTERVAL 1 DAY);
      END;
    END WHILE;
  END IF;
  RETURN days;
END;
|
```

```
DELIMITER ;
select
  daycount('2008-3-16','2008-3-28',7)
  AS 'Sats from 2008-3-16 through 2008-3-28';
```

You would not want to use that function on long date spans in a big table, but it will do for testing.

Now, how to count the number of *Tuesdays*, say, between two dates? The basic logic is:

1. Count *weeks* between the two dates.

2. If beginning and ending weekdays are the same, then if they're Tuesday, the answer is *weeks*+1, otherwise it's just *weeks*.

3. Otherwise, if the beginning weekday <= the ending weekday, then if Tuesday is between them, the answer is *weeks*+1, otherwise it's just *weeks*.

4. Otherwise the ending weekday is less than the starting weekday; if Tuesday >= the starting weekday or <= the ending weekdayy, the answer is *weeks*+1, otherwise it's just *weeks*.

For a convenient datasource, we'll use the two date columns *orderdate* and *shippeddate* in the *orders* table of the NorthWind database, and we'll use our brute force function DayCount() to check results:

```
SET @day = 3;
SELECT
  DATE_FORMAT(orderdate,'%y%m%d') AS OrdDt,
  DATE_FORMAT(shippeddate,'%y%m%d') AS ShipDt,
  LEFT(DAYNAME(orderdate),3) AS D1,
  LEFT(DAYNAME(shippeddate),3) AS D2,
  @dow1 := DAYOFWEEK(orderdate) AS 'dw1',
  @dow2 := DAYOFWEEK(shippeddate) AS 'dw2',
  @days := DATEDIFF(shippeddate,orderdate) AS Days,
  @wks  := FLOOR( @days / 7 ) AS Wks,
  FLOOR( IF( @dow1 = @dow2, IF( @day = @dow1, @wks+1, @wks),
           IF( @dow1 < @dow2, IF( @day BETWEEN @dow1 AND @dow2, @wks+1, @wks),
               IF( @day >= @dow1 OR @day <= @dow2, @wks+1, @wks )
             )
         )
     ) AS Res,
  DayCount(DATE(orderdate),DATE(shippeddate),@day) AS Chk
FROM orders
HAVING !ISNULL(res-chk) AND res-chk <> 0;
Empty set (0.00 sec)
```

No errors. We get the same result for @day = 1, 2, 4, 5, 6 and 7.

But the formula is buried in the specifics of one table, so abstract it to a reusable function:

```
DROP FUNCTION IF EXISTS NamedDaysBetween;
DELIMITER |
CREATE FUNCTION NamedDaysBetween( d1 DATE, d2 DATE, daynum SMALLINT )
RETURNS INT
BEGIN
  DECLARE dow1, dow2, wks, days INT;
  IF !ISNULL(d1) AND !ISNULL(d2) THEN
    SET dow1 = DAYOFWEEK( d1 );
    SET dow2 = DAYOFWEEK( d2 );
    SET days = DATEDIFF( d2, d1 );
    SET wks  = FLOOR( days / 7 );
    SET days = IF( dow1 = dow2, IF( daynum = dow1, wks+1, wks),
                 IF( dow1 < dow2, IF( daynum BETWEEN dow1 AND dow2, wks+1, wks),
                     IF( daynum >= dow1 OR daynum <= dow2, wks+1, wks )
                   )
               );
  END IF;
  RETURN days;
```

```
END;
|
DELIMITER ;
```

Again check it against lots of date value pairs:

```
SELECT
  nameddaysbetween(orderdate,shippeddate,3) - daynamecount(orderdate,shippeddate,3)
  AS diff
FROM orders
HAVING !ISNULL(diff) AND diff <> 0;
Empty set (0.00 sec)
```

Back to the top	Browse the book	Buy the book	Feedback

Date of first Friday of next month

Assuming a calendar table `calendar(date DATE)` with one row per date through the relevant period...

```
SET @d = NOW();
SELECT MIN(date) AS 1stFridayOfMonth
FROM calendar
WHERE YEAR(date) = IF( MONTH(@d) = 12, 1+YEAR(@d), YEAR(@d) )
  AND MONTH(date) = IF( MONTH(@d) = 12, 1, MONTH(@d) + 1 )
  AND WEEKDAY(date)=4;
```

Back to the top	Browse the book	Buy the book	Feedback

Date of Monday in a given week of the year

The week number of a given date @d, when weeks are defined as starting on Mondays and when we agree to number weeks of the year from 1 through 53, is given by WEEK(@d, 2). Here is a way to get the date of Monday in that week:

```
set @d='2008-1-31';
select makedate( left(yearweek(@d),4),week( @d, 2 ) * 7 ) as 1stdayOfWeek;
+--------------+
| 1stdayOfWeek |
+--------------+
| 2008-01-28   |
+--------------+
set @d='2008-7-15';
select makedate( left(yearweek(@d),4),week( @d, 2 ) * 7 ) as 1stdayOfWeek;
+--------------+
| 1stdayOfWeek |
+--------------+
| 2008-07-14   |
+--------------+
```

Back to the top	Browse the book	Buy the book	Feedback

Datetime difference

Find the difference between two datetime values in seconds, minutes, hours or days. If dt1 and dt2 are datetime values of the form 'yyyy-mm-dd hh:mm:ss', the number of seconds between dt1 and dt2 is

```
UNIX_TIMESTAMP( dt2 ) - UNIX_TIMESTAMP( dt1 )
```

To get the number of minutes divide by 60, for the number of hours divide by 3600, and for the number of days, divide by 3600 * 24.

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Find available reservation periods

Given a bookings table where each row specifies one reservation period for one property, find the unbooked periods for a given property:

```
CREATE TABLE bookings( ID int, propertyID int, startDate date, endDate date );
INSERT INTO bookings VALUES
  (1,1,'2007-1-1','2007-1.15'),
  (2,1,'2007-1-20','2007-1.31'),
  (3,1,'2007-2-10','2007-2-17');
SELECT * FROM bookings;
+------+------------+------------+------------+
| ID   | propertyID | startDate  | endDate    |
+------+------------+------------+------------+
|    1 |          1 | 2007-01-01 | 2007-01-15 |
|    2 |          1 | 2007-01-20 | 2007-01-31 |
|    3 |          1 | 2007-02-10 | 2007-02-17 |
+------+------------+------------+------------+
```

Reservation systems usually adopt the *closed-open* convention of representing when reservations begin and end. For example, if you book a hotel room for 22 May through 24 May, the hotel expects you to stay overnight on 22 May and 23 May, but not on 24 May. Apart from that difference, this is the same pattern as *Finding missing numbers in a sequence.*

```
SELECT
  a.enddate AS 'Available From',
  Min(b.startdate) AS 'To'
FROM bookings AS a
JOIN bookings AS b ON a.propertyID=b.propertyID AND a.enddate < b.startdate
WHERE a.propertyID=1
GROUP BY a.enddate
HAVING a.enddate < MIN(b.startdate);
+----------------+------------+
| Available From | To         |
+----------------+------------+
| 2007-01-15     | 2007-01-20 |
| 2007-01-31     | 2007-02-10 |
+----------------+------------+
```

This query cannot see reservation dates earlier than the first existing reservation date, or later than the last. Usually, you would want a calendar table to provide those limits, but you can fake them with a union. If the allowable reservation period is 1 Dec 2006 through 1 Jul 2007, union the left side of the join with a made-up row for 1 Dec 2006, and union the right side of the join with a made-up row for 1 Jul 2007:

```
SELECT
  a.enddate AS 'Available From',
  Min(b.startdate) AS 'To'
FROM (
  SELECT 0,1 as propertyID,'2006-12-01' as startdate,'2006-12-01' as enddate
  UNION
  SELECT * FROM bookings
) AS a
JOIN (
  SELECT * FROM bookings
  UNION
  SELECT 0,1,'2007-07-01' as startdate,'2007-07-02' as enddate
) AS b ON  a.propertyID=b.propertyID AND a.enddate < b.startdate
WHERE a.propertyID=1
GROUP BY a.enddate
HAVING a.enddate < MIN(b.startdate);
+----------------+------------+
| Available From | To         |
+----------------+------------+
```

```
| 2006-12-01     | 2007-01-01 |
| 2007-01-15     | 2007-01-20 |
| 2007-01-31     | 2007-02-10 |
| 2007-02-17     | 2007-07-01 |
+----------------+------------+
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Find overlapping periods

You have a table of visits, and you would like to display the time periods during which there were visit time overlaps.

```
drop table if exists visits;
create table visits(id int primary key,start datetime,end datetime);
insert into visits values
(1, '2008-09-01 15:01', '2008-09-01 15:04'),
(2, '2008-09-01 15:02', '2008-09-01 15:09'),
(3, '2008-09-01 15:12', '2008-09-01 15:15'),
(4, '2008-09-01 16:11', '2008-09-01 16:23'),
(5, '2008-09-01 16:19', '2008-09-01 16:25'),
(6, '2008-09-01 17:52', '2008-09-01 17:59'),
(7, '2008-09-01 18:18', '2008-09-01 18:22');
```

There are two overlap periods in this data:

```
1     |-----|
2        |-----|
3                 |--|
4                       |-----|
5                          |-----|
6                                  |---|
7                                        |---|
```

One solution is to use a View to identify starting and stopping events, then define an Overlaps View:

```
CREATE VIEW events AS
SELECT start AS time, 1 AS value, id FROM visits
UNION
SELECT end AS time, -1 AS value, id FROM visits;

CREATE VIEW overlaps AS
SELECT time t, (SELECT SUM(value) FROM events WHERE time <=t ) as visitcount
FROM events;

SELECT t, visitcount
FROM overlaps
WHERE visitcount=(SELECT MAX(visitcount) FROM overlaps);
+---------------------+------------+
| t                   | visitcount |
+---------------------+------------+
| 2008-09-01 15:02:00 |          2 |
| 2008-09-01 16:19:00 |          2 |
+---------------------+------------+
```

But that doesn't show us when overlap periods end. There is a fuller and more straightforward solution: join `visits` to itself on the criteria that

(i) the first of each joined pair of visits started no earlier than the second,
(ii) the first visit started before the second ended, and
(iii) the second visit started before the first ended:

```
SELECT v1.id,v1.start,v2.id,v2.end
FROM visits v1
JOIN visits v2 ON v1.id <> v2.id and v1.start >= v2.start and v1.start < v2.end and v2.start < v1.end;
```

```
+------+---------------------+------+---------------------+
| id   | start               | id   | end                 |
+------+---------------------+------+---------------------+
| 2    | 2008-09-01 15:02:00 | 1    | 2008-09-01 15:04:00 |
| 5    | 2008-09-01 16:19:00 | 4    | 2008-09-01 16:23:00 |
+------+---------------------+------+---------------------+
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Find sequenced duplicates

A table that tracks time periods may require period uniqueness. That means it has no sequenced duplicates.

If a table has columns processID, start_date and end_date, those three columns are period unique if there exists no pair of rows with the same processID and overlapping start_date and end_date values. If there is such a pair of rows, the table exhibits *sequenced duplication*.

Another way of saying it: if an instant is the smallest datetime unit of start_date and end_date columns, then if there are no sequenced duplicates, there is exactly one processID value at any instant.

Here is a query to find sequenced duplicates for those columns:

```
SELECT t.processid
FROM tbl t
WHERE EXISTS (
  SELECT * FROM tbl AS t3
  WHERE t3.processid IS NULL
)
OR EXISTS (
  SELECT * FROM tbl AS t1
  WHERE 1 < (
    SELECT COUNT(processid)
    FROM tbl AS t2
    WHERE t1.processid = t2.processid
      AND t1.start_date < t2.end_date
      AND t2.start_date < t1.end_date
  )
);
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Is a given booking period available?

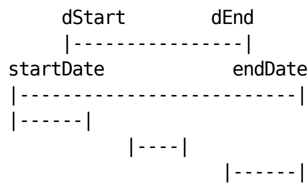
You rent vacation properties, tracking bookings with a table like this:

```
CREATE TABLE bookings( ID int, propertyID int, startDate date, endDate date );
INSERT INTO bookings VALUES (1,1,'2007-1-1','2007-1.15'),(2,1,'2007-1-20','2007-1.31');
SELECT * FROM bookings;
+------+------------+------------+------------+
| ID   | propertyID | startDate  | endDate    |
+------+------------+------------+------------+
|    1 |          1 | 2007-01-01 | 2007-01-15 |
|    2 |          1 | 2007-01-20 | 2007-01-31 |
+------+------------+------------+------------+
```

You need a query indicating whether a given property is available for a given period of time.

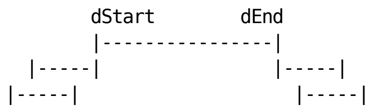
Hotels & property renters usually adopt what is called the 'closed-open' convention for bookings, eg a booking from 22 May through 24 May means you sleep there the nights of 22 and 23 May. To show that property P is available for the desired closed-open period dStart to dEnd, you need to prove there is no booked period for P that overlaps dStart through dEnd.

Until you're used to thinking about periods, it's easier to analyse graphically. There are four ways a booked reservation can overlap the desired date range ...



```
             dStart        dEnd
             |----------------|
        startDate            endDate
        |-------------------------|
        |------|
                   |----|
                            |------|
```

but there are just two ways a booked reservation can *not* overlap:



```
             dStart        dEnd
             |----------------|
       |-----|                |-----|
     |-----|                    |-----|
```

So the period dStart through dEnd is available if there is no row where ...

```
!(endDate <= dStart OR startDate >= dEnd)
```

or equivalently ...

```
endDate > dStart AND startDate < dEnd
```

Here is a simple stored procedure for testing the query:

```
DROP PROCEDURE IF EXISTS isavailable;
DELIMITER |
CREATE PROCEDURE isavailable( iProperty int, dStart date, dEnd date )
SELECT IF( COUNT(1),'No','Yes' ) AS Available
FROM bookings
WHERE propertyID = iProperty
  AND startDate < dEnd
  AND endDate > dStart;
|
DELIMITER ;

CALL isavailable(1,'2006-12-27','2007-1-20');
+-----------+
| Available |
+-----------+
| No        |
+-----------+

CALL isavailable(1,'2007-1-10' ,'2007-1-16');
+-----------+
| Available |
+-----------+
| No        |
+-----------+

CALL isavailable(1,'2007-1-16' ,'2007-1-17');
+-----------+
| Available |
+-----------+
| Yes       |
+-----------+

CALL isavailable(1,'2007-1-22' ,'2007-1-23');
+-----------+
| Available |
+-----------+
| No        |
+-----------+
```

```
CALL isavailable(1,'2007-1-22' ,'2007-2-2');
+-----------+
| Available |
+-----------+
| No        |
+-----------+

CALL isavailable(1,'2007-2-1' ,'2007-2-2');
+-----------+
| Available |
+-----------+
| Yes       |
+-----------+

CALL isavailable(1,'2006-12-1' ,'2007-2-1');
+-----------+
| Available |
+-----------+
| No        |
+-----------+
1 row in set (0.00 sec)
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Julian date

Unix_Timestamp(datetimevalue) / (60*60*24)) + 2440587.5

Back to the top *Browse the book* *Buy the book* *Feedback*

Last business day before a reference date

Given a date value in *datetimecol* ...

```
SELECT
  @refday := datetimecol,
  @dow := DAYOFWEEK(@refday) AS DOW,
  @subtract := IF( @dow = 1, 2, IF( @dow = 2, 3, 1 )) AS MINUS,
  @refday - INTERVAL @subtract DAY AS LastBizDay
FROM ... etc
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Make a calendar table

You need a calendar table for joins to datetime data in other tables:

```
create table calendar (
  dt datetime primary key
);
```

An elegant method of generating any desired number of sequential values, posted by Giuseppe Maxia on his blog, is ...

- Create three dummy rows in a View.
- Cross join them to make 10 dummy rows.
- Cross join those to make 100, 1,000 or however many you need.

So to give the calendar table a million rows at one-hour intervals starting on 1 Jan 1970:

```
create view v3 as select 1 n union all select 1 union all select 1;
```

```
create view v as select 1 n from v3 a, v3 b union all select 1;
set @n = 0;
insert into calendar select '1970-1-1 00:00:00'+interval @n:=@n+1 hour
from v a, v b, v c, v d, v e, v;
```

If your MySQL version does not support Views, or if you prefer to avoid writing the View joins, make a table of integers 1 through 10...

```
create table ints(i int);
insert into ints values(1),(2),(3),(4),(5),(6),(7),(8),(9),(10);
```

and join them as we did with the Views above.

Here is a variation on Giuseppe's method (written about also by Baron Schwartz and Rob Wultsch) to make a table of a thousand dates starting today:

```
CREATE TABLE digits(i INT PRIMARY KEY);
INSERT INTO digits VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9);

SELECT CURDATE() + INTERVAL t.i*10 + u.i DAY AS Date
FROM digits AS t
JOIN digits AS u
JOIN digits AS v
WHERE ( t.i*100 + u.i*10 + v.i ) < 1000;
```

You can use such a `SELECT` as a View, as an inline derived table, or as input to a `CREATE TABLE` statement.

A slightly more elaborate method, giving the calendar table an auto-increment key that can also be used as a surrogate for datetime interval calculations:

```
  CREATE TABLE calendar (
    id INT AUTO_INCREMENT PRIMARY KEY,
    date DATE,
    UNIQUE days (date)
  );
```

Calculate the number of days needed in the calendar, eg

```
  SELECT DATEDIFF('2010-12-31','1989-12-31');   # 7670, or 21*365 plus 5
```

Find a table with that many rows, 7670 in this case. Add a row to the calendar table for every day in the range:

```
  INSERT INTO calendar (id)
  SELECT NULL FROM [name of table with 7670  rows] LIMIT 4018;
```

Populate the date column by incrementing the starting date:

```
  UPDATE calendar SET date = ADDDATE('1989-12-31',id);
```

The calendar table now has one row for each day from 1990-01-01 through 2010-12-31. Keep the auto_increment ID column for quick day counts in the range, or drop the column if you don't need that.

To make the calendar table a diary, make the period one leap year, add month, day and text columns, update month and day values with MONTH(date) and DAYOFMONTH(date) respectively, and if the diary is to be used from year to year, drop the date field.
[Based on a *builder.com* SQL Tip by Arthur Fuller and a MySQL list tip by Michael Stassen]

To automate all this, write a stored procedure, for example:

```
CREATE TABLE times (
  date_hour DATETIME,
  KEY ( date_hour )
);
```

```
DROP PROCEDURE IF EXISTS timespopulate;
DELIMITER |
CREATE PROCEDURE timespopulate( startdate DATETIME, num INT )
BEGIN
  DECLARE ctr INT DEFAULT 0;
  WHILE ctr < num DO
    BEGIN
      INSERT INTO times VALUES ( DATE_ADD( startdate, INTERVAL ctr HOUR) );
      SET ctr = ctr + 1;
    END;
  END WHILE;
END;
|
DELIMITER ;
CALL timespopulate( '2007-1-1, 31*24 );
```

Or, you can have the sproc do your counting:

```
DROP PROCEDURE IF EXISTS calendar;
DELIMITER |
CREATE PROCEDURE calendar( pstart datetime, pstop datetime, pminutes int )
DETERMINISTIC
BEGIN
  DECLARE thisdate datetime;
  DROP TABLE IF EXISTS cal;
  CREATE TABLE cal( dt datetime );
  SET thisdate=pstart;
  INSERT INTO cal VALUES(pstart);
  WHILE thisdate < pstop u
    SET thisdate = adddate( thisdate, INTERVAL pminutes MINUTE );
    INSERT INTO cal VALUES( thisdate );
  END WHILE;
END |
DELIMITER ;
-- make cal for 2007, 20-min intervals:
CALL calendar('2007-1-1 00:00:00', '2007-2-1 00:00:00', 20);
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Sum for time periods

A table tracks attendance at some location:

```
drop table if exists t;
create table t(interval_id int,start datetime,end datetime, att int);
insert into t values
(1,'2007-01-01 08:00:00','2007-01-01 12:00:00',5 ),
(2,'2007-01-01 13:00:00','2007-01-01 17:00:00',10),
(3,'2007-01-01 10:00:00','2007-01-01 15:00:00',15),
(4,'2007-01-01 14:00:00','2007-03-07 19:00:00',20);
select * from t;
+-------------+---------------------+---------------------+------+
| interval_id | start               | end                 | att  |
+-------------+---------------------+---------------------+------+
|           1 | 2007-01-01 08:00:00 | 2007-01-01 12:00:00 |    5 |
|           2 | 2007-01-01 13:00:00 | 2007-01-01 17:00:00 |   10 |
|           3 | 2007-01-01 10:00:00 | 2007-01-01 15:00:00 |   15 |
|           4 | 2007-01-01 14:00:00 | 2007-03-07 19:00:00 |   20 |
+-------------+---------------------+---------------------+------+
```

In this table, `att` is a *delta*: it tracks *entrances*. Actual attendance is `SUM(att)` at any given moment. For example, if the attendance sum is `x` at a given moment, then after we add a row with `att=y` and start/end datetimes embracing that moment, attendance will be `x+y`. So to retrieve total attendance at 01330h on 1 Jan 2007, we write:

```
SELECT SUM(att)
```

```
FROM t
WHERE t.start <= '2007-01-01 13:30:00' AND t.end >= '2007-01-01 13:30:00';
+----------+
| SUM(att) |
+----------+
|       25 |
+----------+
```

Then how would we extract maximum attendance during a given period, for example, maximum attendance between 1300h and 1700h?

SQL does not deal efficiently with time. Some SQL dialects offer time series enhancements to the language; MySQL does not.

And, querying time series data for aggregate statistics gets complicated very quickly.

It gets a bit simpler with a calendar table that has a row for every possible datetime value. For our example, assume a granularity of one hour and a query period of one day. Naturally a real system would require a range of dates and perhaps a finer time granularity:

```
create table cal(id int,dt datetime);
insert into cal values(1,'2007-1-1 01:00:00');
insert into cal values(2,'2007-1-1 02:00:00');
insert into cal values(3,'2007-1-1 03:00:00');
insert into cal values(4,'2007-1-1 04:00:00');
insert into cal values(5,'2007-1-1 05:00:00');
insert into cal values(6,'2007-1-1 06:00:00');
insert into cal values(7,'2007-1-1 07:00:00');
insert into cal values(8,'2007-1-1 08:00:00');
insert into cal values(9,'2007-1-1 09:00:00');
insert into cal values(10,'2007-1-1 10:00:00');
insert into cal values(11,'2007-1-1 11:00:00');
insert into cal values(12,'2007-1-1 12:00:00');
insert into cal values(13,'2007-1-1 13:00:00');
insert into cal values(14,'2007-1-1 14:00:00');
insert into cal values(15,'2007-1-1 15:00:00');
insert into cal values(16,'2007-1-1 16:00:00');
insert into cal values(17,'2007-1-1 17:00:00');
insert into cal values(18,'2007-1-1 18:00:00');
insert into cal values(19,'2007-1-1 19:00:00');
insert into cal values(20,'2007-1-1 20:00:00');
insert into cal values(21,'2007-1-1 21:00:00');
insert into cal values(22,'2007-1-1 22:00:00');
insert into cal values(23,'2007-1-1 23:00:00');
insert into cal values(24,'2007-1-1 24:00:00');
```

To accumulate the maximum attendance sum, collect target values for defined periods in an inner query, and sum them from the outer query:

```
SELECT SUM( att )
FROM (
  SELECT
    t.start AS PeriodStart,
    t.end AS PeriodEnd,
    MIN(cal.dt) + INTERVAL 1 HOUR AS CountBegin,
    MAX(cal.dt) AS CountEnd,
    t.att
  FROM t
  JOIN cal ON cal.dt >= t.start AND cal.dt < t.end
  GROUP BY PeriodStart, PeriodEnd
  HAVING CountBegin < '2007-01-01 17:00:00' AND CountEnd > '2007-01-01 11:00:00'
) AS periods;
+------------+
| SUM( att ) |
+------------+
|         45 |
```

```
+------------+
```

If the data is more complicated, eg if we also need to track exits, the period logic needs refinement but the principle remains the same.

Back to the top *Browse the book* *Buy the book* *Feedback*

Sum time values

```
SELECT SEC_TO_TIME( SUM( TIME_TO_SEC( time_col ))) AS total_time
FROM tbl;
```

Summing values like '12:65:23' produces meaningless results.

Back to the top *Browse the book* *Buy the book* *Feedback*

The date of next Thursday

Given a date and its weekday number (1=Sunday, ..., 7=Saturday), there are three possibilities:
1. Today is Thursday: thebn next Thursday is 7 days from nopw.
2. Today is before Thuirsday in the week: then next Thursday is (5 minus today's weekday number) from now.
3. Today is after Thursday in the week: then next Thursday is today's weekday number-1 days from today.

```
set @d=curdate();
set @n = dayofweek(curdate());
select
  @d:=adddate(curdate(),0) as date,
  @n:=dayofweek(adddate(curdate(),0)) as weekday,
  adddate(@d,if(@n=5,7,if(@n<5,5-@n,@n-1) ) ) as thurs;
+------------+---------+------------+
| date       | weekday | nextthurs  |
+------------+---------+------------+
| 2008-03-10 |       2 | 2008-03-13 |
+------------+---------+------------+
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Game schedule

List all possible home-away encounters of teams listed in a table.

```
SELECT t1.name AS Visiting,
       t2.name AS Home
FROM teams AS t1
STRAIGHT_JOIN teams AS t2
WHERE t1.ID <> t2.ID;
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Pivot table schedule

You have a schedule table (period, day, subject, room) with a primary key *period,day* to avoid duplicate bookings. You wish to display the schedule as periods, subjects and rooms in rows, and days of the week in columns.

```
SELECT
  period,
  MAX(IF(day=1, CONCAT(subject,' ',room), '')) AS Mon,
  MAX(IF(day=2, CONCAT(subject,' ',room), '')) AS Tue,
  MAX(IF(day=3, CONCAT(subject,' ',room), '')) AS Wed,
```

```
  MAX(IF(day=4, CONCAT(subject,' ',room), '')) AS Thu,
  MAX(IF(day=5, CONCAT(subject,' ',room), '')) AS Fri
FROM schedule
GROUP BY period
```

MAX() chooses existing over blank entries, and GROUP BY lines everything up on the same row.

Back to the top *Browse the book* *Buy the book* *Feedback*

Add auto-incrementing primary key to a table

The steps are: (i) recreate the table, populating a new column from an incrementing user variable, then (ii) alter the table to add auto_increment and primary key properties to the new column. So given table t with columns named `dt` and `observed`...

```
DROP TABLE IF EXISTS t2;
SET @id=0;
CREATE TABLE t2
  SELECT @id:=@id+1 AS id, dt, observed FROM t ORDER BY dt;
ALTER TABLE t2
  MODIFY id INT AUTO_INCREMENT PRIMARY KEY;
DROP TABLE t;
RENAME TABLE t2 TO t;
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Auto-increment: reset next value

ALTER TABLE tbl SET AUTO_INCREMENT=val;

Back to the top *Browse the book* *Buy the book* *Feedback*

Change or drop a foreign key

To change a foreign key, first drop it, then declare the new, revised foreign key. The syntax for declaring a foreign key is ...

```
[CONSTRAINT [constraint_name]]
FOREIGN KEY [key_name] (keycol_name,...) reference_definition
```

and the syntax for dropping one is ...

```
DROP FOREIGN KEY constraint_name
```

Notice that you can omit the `CONSTRAINT` when you declare a foreign key, but the *only* way to `DROP` a foreign key is to reference it by the `constraint_name` which you probably never specified!

There should be a circle of hell reserved for designers who build inconsistencies like this into their tools. The only way round this one is to run `SHOW CREATE TABLE` to find out what the foreign key's `constraint_name` is, so you can write the `DROP` statement. Here is a wee test case:

```
drop table if exists a,b;
create table a(i int primary key)engine=innodb;
create table b(i int,foreign key(i) references a(i)) engine=innodb;
show create table\G

CREATE TABLE `b` (
  `i` int(11) DEFAULT NULL,
  KEY `i` (`i`),
  CONSTRAINT `b_ibfk_1` FOREIGN KEY (`i`) REFERENCES `a` (`i`)
```

```
) ENGINE=InnoDB DEFAULT CHARSET=latin1

-- drop and recreate the FK:
alter table b drop foreign key b_ibfk_1;
alter table b add foreign key(i) references a(i) on update cascade;
show create table b\G

Create Table: CREATE TABLE `b` (
  `i` int(11) DEFAULT NULL,
  KEY `i` (`i`),
  CONSTRAINT `b_ibfk_1` FOREIGN KEY (`i`) REFERENCES `a` (`i`) ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1

drop table a,b;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Compare structures of two tables

To compare columns by name and ordinal position in tables test.t1 and test.t2:

```
SELECT
  MIN(TableName) AS 'Table',
  column_name AS 'Column',
  ordinal_position AS 'Position'
FROM (
  SELECT
    't1' as TableName,
    column_name,
    ordinal_position
  FROM information_schema.columns AS i1
  WHERE table_schema='test' AND table_name='t1'
  UNION ALL
  SELECT
    't2' as TableName,
    column_name,
    ordinal_position
  FROM information_schema.columns AS i2
  WHERE table_schema='test' AND table_name='t2'
) AS tmp
GROUP BY column_name
HAVING COUNT(*) = 1
ORDER BY ordinal_position;
```

For MySQL 5.0.2 or later here is a query that lists all table structure differences between any two tables. It selects all information_schema.columns rows for one table, does the same for the second table, UNIONs these two queries, then uses HAVING to pick only those rows where the COUNT(*) in the union is 1—that is, where any column of one table differs from its mate.

To avoid having to cut and paste database and table names, save it as a stored procedure in any database (other than information_schema):

```
DROP PROCEDURE IF EXISTS CompareTableStructs;
-- Uncomment if MySQL version is 5.0.6-5.0.15:
-- SET GLOBAL log_bin_trust_routine_creators=TRUE;
DELIMITER |
CREATE PROCEDURE CompareTableStructs (
  IN db1 CHAR(64), IN tbl1 CHAR(64), IN db2 CHAR(64), IN tbl2 CHAR(64)
)
SELECT
  MIN(TableName) AS TableName,
  column_name,
  ordinal_position,
  column_default,
  is_nullable,
```

```
  data_type,
  character_maximum_length,
  numeric_precision,
  numeric_scale,
  character_set_name,
  collation_name,
  column_type,
  column_key,
  extra,
  privileges,
  column_comment
FROM (
  SELECT
    tbl1 as TableName,
    column_name,
    ordinal_position,
    column_default,
    is_nullable,
    data_type,
    character_maximum_length,
    numeric_precision,
    numeric_scale,
    character_set_name,
    collation_name,
    column_type,
    column_key,
    extra,
    privileges,
    column_comment
  FROM information_schema.columns AS i1
  WHERE table_schema=db1 AND table_name=tbl1
  UNION ALL
  SELECT
    tbl2 as TableName,
    column_name,
    ordinal_position,
    column_default,
    is_nullable,
    data_type,
    character_maximum_length,
    numeric_precision,
    numeric_scale,
    character_set_name,
    collation_name,
    column_type,
    column_key,
    extra,
    privileges,
    column_comment
  FROM information_schema.columns AS i2
  WHERE table_schema=db2 AND table_name=tbl2
) AS tmp
GROUP BY column_name
HAVING COUNT(*) = 1
ORDER BY column_name ;
|
DELIMITER ;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Compare two databases

One of EF Codd's rules for relational databases is the no-back-door rule: all info about tables should be accessible only by a query on tables. Since version 5, the MySQL implementation of `information_schema` (I_S) helps meet Codd's requirement. I_S supplies metadata in tables, so it's the first place to look for how to compare the structures of two databases.

Elsewhere on this page there is a simple query template for comparing data in two structurally similar tables:

```
SELECT MIN(TableName) as TableName, id, col1, col2, col3, ...
FROM (
  SELECT 'Table a' as TableName, a.id, a.col1, a.col2, a.col3, ...
  FROM a
  UNION ALL
  SELECT 'Table b' as TableName, b.id, b.col1, b.col2, b.col3, ...
  FROM b
) AS tmp
GROUP BY id, col1, col2, col3, ...
HAVING COUNT(*) = 1
ORDER BY ID;
```

To apply this logic to the comparison of two database structures:

- write temp tables collecting desired I_S metadata on each database
- map the compare-data query template to those two metadata tables

This logic is easiest to re-use when it is parameterised in a stored procedure, in a *system* database:

```
USE sys;
DROP PROCEDURE IF EXISTS CompareDBs;
DELIMITER |
CREATE PROCEDURE CompareDBs( vdb1 VARCHAR(64), vdb2 VARCHAR(64) )
BEGIN

  DROP TEMPORARY TABLE IF EXISTS desc1,desc2;
  CREATE TEMPORARY TABLE desc1
  SELECT
    t1.table_schema,
    t1.table_name,
    t1.table_type,
    t1.engine,
    c1.column_name,
    c1.ordinal_position,
    c1.column_type,
    c1.column_default,
    c1.is_nullable,
    c1.column_key
  FROM information_schema.tables t1
  JOIN information_schema.columns c1 USING (table_schema,table_name)
  WHERE t1.table_schema=vdb1
  ORDER BY t1.table_name,c1.column_name;

  CREATE TEMPORARY TABLE desc2
  SELECT
    t1.table_schema,
    t1.table_name,
    t1.table_type,
    t1.engine,
    c1.column_name,
    c1.ordinal_position,
    c1.column_type,
    c1.column_default,
    c1.is_nullable,
    c1.column_key
  FROM information_schema.tables t1
  JOIN information_schema.columns c1 USING (table_schema,table_name)
  WHERE t1.table_schema=vdb2
  ORDER BY t1.table_name,c1.column_name;

  SELECT
    TableName,column_name,MIN(SchemaName),table_type,engine,
    ordinal_position,column_type,column_default,is_nullable,column_key
  FROM (
```

```
    SELECT
      a.table_schema AS SchemaName,a.table_name AS TableName,a.table_type,a.engine,
      a.column_name,a.ordinal_position,a.column_type,a.column_default,a.is_nullable,a.column_key
    FROM desc1 a
    UNION ALL
    SELECT
      b.table_schema AS SchemaName,b.table_name AS TableName,b.table_type,b.engine,
      b.column_name,b.ordinal_position,b.column_type,b.column_default,b.is_nullable,b.column_key
    FROM desc2 b
  ) AS tmp
  GROUP BY TableName,table_type,engine,column_name,ordinal_position,column_type,column_default,is_nullable,column_key
  HAVING COUNT(*) = 1
  ORDER BY TableName,column_name,SchemaName;

  DROP TEMPORARY TABLE desc1, desc2;

END |
DELIMITER ;
```

Call it as follows:

```
CALL compareDBs('db1','db2');
```

`MEMORY` tables would it speed it up, but unfortunately MySQL `MEMORY` tables do not support `BLOB/TEXT` columns.

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Find child tables

Starting with MySQL 5, you can find all tables which contain foreign key references to a given table with this `information_schema` query:

```
SELECT c.table_schema,u.table_name,u.column_name,u.referenced_column_name
FROM information_schema.table_constraints AS c
INNER JOIN information_schema.key_column_usage AS u
USING( constraint_schema, constraint_name )
WHERE c.constraint_type = 'FOREIGN KEY'
 AND u.referenced_table_schema='db'
 AND u.referenced_table_name = 'table'
ORDER BY c.table_schema,u.table_name;
```

Unfortunately it performs slowly, not because of how it is written, but because of how MySQL has implemented `information_schema`. Indeed a bug report page is devoted to the issue. It says MySQL AB will not be speeding up `information_schema` query performance any time soon.

In theUsual we recently solved this problem by writing a PHP function that queries `information_schema` if that is required, but by default parses the results of iterative `SHOW TABLES` commands. The `SHOW TABLES` method logic is simple, and will port readily to another application language. It runs 10-50 times faster than the equivalent `information_schema` query.

The following version looks for child tables in one database; it slows down a bit when modified to search all server DBs, but even then it is much faster than its `information_schema` equivalent. It assumes an available connection object `$conn`:

```
function childtables( $db, $table, $via_infoschema=FALSE ) {
  GLOBAL $conn;
  $ret = array();
  if( $via_infoschema ) {
    $res = mysql_query( childtablesqry( $db, $table )) || die( mysql_error() );
    if( !is_bool( $res ))
      while( $row = mysql_fetch_row( $res ))
        $ret[] = $row;
  }
  else {
    $tables = array();
    $res = mysql_query( "SHOW TABLES" );
```

```
    while( $row = mysql_fetch_row( $res )) $tables[] = $row[0];
    $res = mysql_query( "SELECT LOCATE('ANSI_QUOTES', @@sql_mode)" );
    $ansi_quotes = $res ? mysql_result( $res, 0 ) : 0;
    $q = $ansi_quotes ? '"' : "`";
    $sref = ' REFERENCES ' . $q . $table . $q . ' (' . $q;
    foreach( $tables as $referringtbl ) {
      $res = mysql_query( "SHOW CREATE TABLE $referringtbl" );
      $row = mysql_fetch_row( $res );
      if(( $startref = stripos( $row[1], $sref )) > 0 ) {
        $endref = strpos( $row[1], $q, $startref + strlen( $sref ));
        $referencedcol = substr( $row[1], $startref+strlen($sref),
                                 $endref-$startref-strlen($sref) );
        $endkey = $startref;
        while( substr( $row[1], $endkey, 1 ) <> $q ) $endkey--;
        $startkey = --$endkey;
        while( substr( $row[1], $startkey, 1 ) <> $q ) $startkey--;
        $referencingcol = substr( $row[1], $startkey+1, $endkey - $startkey );
        $ret[] = array( $db, $referringtbl, $referencingcol, $referencedcol );
      }
    }
  }
  return $ret;
}

function childtablesqry( $db, $table ) {
  return "SELECT c.table_schema,u.table_name,u.column_name,u.referenced_column_name " .
         "FROM information_schema.table_constraints AS c " .
         "INNER JOIN information_schema.key_column_usage AS u " .
         "USING( constraint_schema, constraint_name ) " .
         "WHERE c.constraint_type = 'FOREIGN KEY' " .
         "AND u.referenced_table_schema='$db' " .
         "AND u.referenced_table_name = '$table' " .
         "ORDER BY c.table_schema,u.table_name";
}
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Find parent tables

List tables which are referenced by foreign key constraints in a given table.This is a simple query on two information_schema tables: table_constraints and key_column_usage. It is easy to parameterise, so we show it in stored procedures. The first sproc lists all foreign key references in a database. The second lists all foreign key references for a table.

```
CREATE PROCEDURE ListParentsForDb( pdb CHAR(64) )
BEGIN
  SELECT
   u.table_schema AS 'Schema',
   u.table_name AS 'Table',
   u.column_name AS 'Key',
   u.referenced_table_schema AS 'Parent Schema',
   u.referenced_table_name AS 'Parent table',
   u.referenced_column_name AS 'Parent key'
  FROM information_schema.table_constraints AS c
  INNER JOIN information_schema.key_column_usage AS u
  USING( constraint_schema, constraint_name )
  WHERE c.constraint_type = 'FOREIGN KEY'
    AND c.table_schema = pdb
  ORDER BY u.table_schema,u.table_name,u.column_name;
END;

CREATE PROCEDURE ListParentsForTable( pdb CHAR(64), ptable CHAR(64) )
BEGIN
  SELECT
   u.table_schema AS 'Schema',
   u.table_name AS 'Table',
```

```
   u.column_name AS 'Key',
   u.referenced_table_schema AS 'Parent Schema',
   u.referenced_table_name AS 'Parent table',
   u.referenced_column_name AS 'Parent key'
  FROM information_schema.table_constraints AS c
  INNER JOIN information_schema.key_column_usage AS u
  USING( constraint_schema, constraint_name )
  WHERE c.constraint_type = 'FOREIGN KEY'
    AND c.table_schema = pdb
    AND u.referenced_table_name = ptable
  ORDER BY u.table_schema,u.table_name,u.column_name;
END;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Find primary key of a table

To retrieve primary keys of db.tbl...

```
SELECT k.column_name
FROM information_schema.table_constraints t
JOIN information_schema.key_column_usage k
USING (constraint_name,table_schema,table_name)
WHERE t.constraint_type='PRIMARY KEY'
  AND t.table_schema='db'
  AND t.table_name='tbl'
```

For pre-5 versions of MySQL:

```
SHOW INDEX FROM tbl
WHERE key_name='primary';
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Find the size of all databases on the server

This is based on a query Mark Leith posted to the MySQL General Discussion list.

```
CREATE VIEW dbsize AS
SELECT
  s.schema_name AS 'Schema',
  IFNULL(ROUND((SUM(t.data_length)+SUM(t.index_length)) /1024/1024,2),0.00)
  AS 'Total Mb',
  IFNULL(ROUND(((SUM(t.data_length) +
  SUM(t.index_length))-SUM(t.data_free))/1024/1024,2),0.00) AS 'Mb Used',
  IFNULL(ROUND(SUM(data_free)/1024/1024,2),0.00) AS 'Mb Free',
  IFNULL(ROUND((((SUM(t.data_length)+SUM(t.index_length))-SUM(t.data_free))
  /((SUM(t.data_length)+SUM(t.index_length)))*100),2),0) AS 'Pct Used',
  COUNT(table_name) AS Tables
FROM INFORMATION_SCHEMA.SCHEMATA s
LEFT JOIN INFORMATION_SCHEMA.TABLES t ON s.schema_name = t.table_schema
GROUP BY s.schema_name
WITH ROLLUP;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

List differences between two databases

Given two databases named @db1 and @db2:

```
SELECT
  MIN(table_name) as TableName,
  table_catalog,table_schema,table_name,column_name,
  ordinal_position,column_default,is_nullable,
  data_type,character_maximum_length,character_octet_length,
  numeric_precision,numeric_scale,character_set_name,
  collation_name,column_type,column_key,
  extra,privileges,column_comment
FROM (
  SELECT 'Table a' as TableName,
  table_catalog,table_schema,table_name,column_name,
  ordinal_position,column_default,is_nullable,
  data_type,character_maximum_length,character_octet_length,
  numeric_precision,numeric_scale,character_set_name,
  collation_name,column_type,column_key,
  extra,privileges,column_comment
  FROM information_schema.columns c1
  WHERE table_schema=@db1
  UNION ALL
  SELECT 'Table a' as TableName,
  table_catalog,table_schema,table_name,column_name,
  ordinal_position,column_default,is_nullable,
  data_type,character_maximum_length,character_octet_length,
  numeric_precision,numeric_scale,character_set_name,
  collation_name,column_type,column_key,
  extra,privileges,column_comment
  FROM information_schema.columns c2
  WHERE table_schema=@db2
) AS tmp
GROUP BY tablename,
  table_catalog,table_schema,column_name,
  ordinal_position,column_default,is_nullable,
  data_type,character_maximum_length,character_octet_length,
  numeric_precision,numeric_scale,character_set_name,
  collation_name,column_type,column_key,
  extra,privileges,column_comment
HAVING COUNT(*) = 1
ORDER BY tablename,column_name;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

List users of a database

```
DROP PROCEDURE IF EXISTS ListDbUsers;
DELIMITER |
CREATE PROCEDURE ListDbUsers( dbname CHAR(64) )
  SELECT host,user
  FROM mysql.user
  WHERE Select_priv = 'Y'
       OR Insert_priv = 'Y'
     OR Update_priv = 'Y'
     OR Delete_priv = 'Y'
     OR Create_priv = 'Y'
     OR Drop_priv = 'Y'
     OR Reload_priv = 'Y'
     OR Shutdown_priv = 'Y'
     OR Process_priv = 'Y'
     OR File_priv = 'Y'
     OR Grant_priv = 'Y'
     OR References_priv = 'Y'
     OR Index_priv = 'Y'
     OR Alter_priv = 'Y'
     OR Show_db_priv = 'Y'
     OR Super_priv = 'Y'
     OR Create_tmp_table_priv = 'Y'
```

```
     OR Lock_tables_priv = 'Y'
     OR Execute_priv = 'Y'
     OR Repl_slave_priv = 'Y'
     OR Repl_client_priv = 'Y'
     OR Create_view_priv = 'Y'
     OR Show_view_priv = 'Y'
     OR Create_routine_priv = 'Y'
     OR Alter_routine_priv = 'Y'
     OR Create_user_priv = 'Y'
     OR Event_priv = 'Y'
     OR Trigger_priv = 'Y'
  UNION
  SELECT host,user
  FROM mysql.db
  WHERE db=dbname
    AND (
     Select_priv = 'Y'
     OR Insert_priv = 'Y'
     OR Update_priv = 'Y'
     OR Delete_priv = 'Y'
     OR Create_priv = 'Y'
     OR Drop_priv = 'Y'
     OR Grant_priv = 'Y'
     OR References_priv = 'Y'
     OR Index_priv = 'Y'
     OR Alter_priv = 'Y'
     OR Create_tmp_table_priv = 'Y'
     OR Lock_tables_priv = 'Y'
     OR Create_view_priv = 'Y'
     OR Show_view_priv = 'Y'
     OR Create_routine_priv = 'Y'
     OR Alter_routine_priv = 'Y'
     OR Execute_priv = 'Y'
     OR Event_priv = 'Y'
     OR Trigger_priv = 'Y'
  )
  UNION
  SELECT host,user
  FROM mysql.tables_priv
  WHERE db=dbname
  UNION
  SELECT host,user
  FROM mysql.columns_priv
  WHERE db=dbname;
|
DELIMITER ;
CALL ListDbUsers( 'test' );
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Rename Database

It's sometimes necessary to rename a database. MySQL 5.0 has no command for it. Simply bringing down the server to rename a database directory is *not* safe. MySQL 5.1.7 introduced a `RENAME DATABASE` command, but the command left several unchanged database objects behind, and was found to lose data, so it was dropped in 5.1.23.

It seems a natural for a stored procedure using dynamic (prepared) statements. `PREPARE` supports `CREATE | RENAME TABLE`. As precautions:

- Before calling the sproc, the new database must have been created.
- The procedure refuses to rename the *mysql* database.
- The old database is left behind, minus what was moved.

```
DROP PROCEDURE IF EXISTS RenameDatabase;
DELIMITER |
CREATE PROCEDURE RenameDatabase(
  IN oldname CHAR (64), IN newname CHAR(64)
)
BEGIN
  DECLARE version CHAR(32);
  DECLARE sname CHAR(64) DEFAULT NULL;
  DECLARE rows INT DEFAULT 1;
  DECLARE changed INT DEFAULT 0;
  IF STRCMP( oldname, 'mysql' ) <> 0 THEN
    REPEAT
      SELECT table_name INTO sname
      FROM information_schema.tables AS t
      WHERE t.table_type='BASE TABLE'
        AND t.table_schema = oldname
      LIMIT 1;
      SET rows = FOUND_ROWS();
      IF rows = 1 THEN
        SET @scmd = CONCAT( 'RENAME TABLE ', oldname, '.', sname,
                           ' TO ', newname, '.', sname );
        PREPARE cmd FROM @scmd;
        EXECUTE cmd;
        DEALLOCATE PREPARE cmd;
        SET changed = 1;
      END IF;
    UNTIL rows = 0 END REPEAT;
    IF changed > 0 THEN
      SET @scmd = CONCAT( "UPDATE mysql.db SET Db = '",
                          newname,
                          "' WHERE Db = '", oldname, "'" );
      PREPARE cmd FROM @scmd;
      EXECUTE cmd;
      DROP PREPARE cmd;
      SET @scmd = CONCAT( "UPDATE mysql.proc SET Db = '",
                          newname,
                          "' WHERE Db = '", oldname, "'" );
      PREPARE cmd FROM @scmd;
      EXECUTE cmd;
      DROP PREPARE cmd;
      SELECT version() INTO version;
      IF version >= '5.1.7' THEN
        SET @scmd = CONCAT( "UPDATE mysql.event SET db = '",
                            newname,
                            "' WHERE db = '", oldname, "'" );
        PREPARE cmd FROM @scmd;
        EXECUTE cmd;
        DROP PREPARE cmd;
      END IF;
      SET @scmd = CONCAT( "UPDATE mysql.columns_priv SET Db = '",
                          newname,
                          "' WHERE Db = '", oldname, "'" );
      PREPARE cmd FROM @scmd;
      EXECUTE cmd;
      DROP PREPARE cmd;
      FLUSH PRIVILEGES;
    END IF;
  END IF;
END;
|
DELIMITER ;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Show Create Trigger

As yet, MySQL has no Show Create Trigger command.Here is a stored procedure which behaves as a Show Create Trigger command should, listing all Triggers defined on one table in one database

```
SET GLOBAL log_bin_trust_routine_creators=TRUE;
DROP PROCEDURE IF EXISTS ShowCreateTrigger;
DELIMITER |
CREATE PROCEDURE ShowCreateTrigger( IN db CHAR(64), IN tbl CHAR(64) )
  BEGIN
    SELECT
      CONCAT(
        'CREATE TRIGGER ',trigger_name, CHAR(10),
        action_timing,' ', event_manipulation, CHAR(10),
        'ON ',event_object_schema,'.',event_object_table, CHAR(10),
        'FOR EACH ROW', CHAR(10),
        action_statement, CHAR(10)
      ) AS 'Triggers'
    FROM information_schema.triggers
    WHERE event_object_schema = db
      AND event_object_table = tbl;
  END;
|
DELIMITER ;
```

and here is a stored proc which lists all triggers in a database:

```
SET GLOBAL log_bin_trust_routine_creators=TRUE;
DROP PROCEDURE IF EXISTS ListTriggers;
DELIMITER |
CREATE PROCEDURE ListTriggers( IN db CHAR(64) )
  BEGIN
    SELECT
      trigger_name AS 'Trigger',
      event_object_table AS 'Table'
    FROM information_schema.triggers
    WHERE event_object_schema = db;
  END;
|
DELIMITER ;
```

If you have a collection of generic stored procs like these, it's most convenient to keep them in one place for easy accessibility. We keep ours in a *sys* database.

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Show Tables

The MySQL `SHOW TABLES` command is fine, but sometimes we want a little more information.

This simple stored procedure lists the table name, engine type, version, collation and rowcount for every table in a database. (Individual databases come and go, so we keep all such database-wide stored routines in a 'system' database.)

```
DROP PROCEDURE IF EXISTS ShowTables;
DELIMITER |
CREATE PROCEDURE ShowTables( IN dbname CHAR(64) )
BEGIN
  SELECT
    table_name,
    engine,
    version,
    table_collation AS collation,
    table_rows AS rows
  FROM information_schema.tables
  WHERE table_schema=dbname;
END;
|
```

```
DELIMITER ;
```

Back to the top | Browse the book | Buy the book | Feedback

Display column values which occur N times

```
SELECT id
FROM tbl
GROUP BY id
HAVING COUNT(*) = N;
```

Change the HAVING condition to >1 to list duplicate values, etc.

Back to the top | Browse the book | Buy the book | Feedback

Display every Nth row

```
MySQL earlier than version 4.1:

SELECT id
FROM tbl
GROUP BY id
HAVING MOD(id,N) = 0;

Since MySQL 4.1:

SELECT *
FROM tbl
WHERE ( id, 0 ) IN (
  SELECT id, MOD( id, N ) FROM tbl
);

SELECT *
FROM (
 SELECT id FROM tbl
) AS tmp
WHERE MOD( tmp.id, N ) = 0;
```

Back to the top | Browse the book | Buy the book | Feedback

Trees, networks and parts explosions in MySQL

http://www.artfulsoftware.com/mysqlbook/sampler/mysqled1ch20.html

Back to the top | Browse the book | Buy the book | Feedback

Dijkstra�s shortest path algorithm

Given a table of source-to-destination paths, each of whose nodes references a row in a nodes table, how do we find the shortest path from one node to another?

One answer is Dijkstra's algorithm (http://en.wikipedia.org/wiki/Dijkstra's_algorithm). Peter Larsson has posted a SQL Server implementation of it on the SQL Team Forum. Here is a MySQL implementation.

The DDL:

```
DROP TABLE IF EXISTS dijnodes,dijpaths;
CREATE TABLE dijnodes (
  nodeID int PRIMARY KEY AUTO_INCREMENT NOT NULL,
  nodename varchar (20) NOT NULL,
  cost int NULL,
  pathID int NULL,
  calculated tinyint NOT NULL
);

CREATE TABLE dijpaths (
  pathID int PRIMARY KEY AUTO_INCREMENT,
  fromNodeID int NOT NULL ,
  toNodeID int NOT NULL ,
  cost int NOT NULL
);
```

Here is a stored procedure to populate valid nodes and paths:

```
DROP PROCEDURE IF EXISTS dijAddPath;
DELIMITER |
CREATE PROCEDURE dijAddPath(
  pFromNodeName VARCHAR(20), pToNodeName VARCHAR(20), pCost INT
)
BEGIN
  DECLARE vFromNodeID, vToNodeID, vPathID INT;
  SET vFromNodeID = ( SELECT NodeID FROM dijnodes WHERE NodeName = pFromNodeName );
  IF vFromNodeID IS NULL THEN
    BEGIN
      INSERT INTO dijnodes (NodeName,Calculated) VALUES (pFromNodeName,0);
      SET vFromNodeID = LAST_INSERT_ID();
    END;
  END IF;
  SET vToNodeID = ( SELECT NodeID FROM dijnodes WHERE NodeName = pToNodeName );
  IF vToNodeID IS NULL THEN
    BEGIN
      INSERT INTO dijnodes(NodeName, Calculated)
      VALUES(pToNodeName,0);
      SET vToNodeID = LAST_INSERT_ID();
    END;
  END IF;
  SET vPathID = ( SELECT PathID FROM dijpaths
                  WHERE FromNodeID = vFromNodeID AND ToNodeID = vToNodeID
                );
  IF vPathID IS NULL THEN
    INSERT INTO dijpaths(FromNodeID,ToNodeID,Cost)
    VALUES(vFromNodeID,vToNodeID,pCost);
  ELSE
    UPDATE dijpaths SET Cost = pCost
    WHERE FromNodeID = vFromNodeID AND ToNodeID = vToNodeID;
  END IF;
END;
|
DELIMITER ;
```

Use dijAddpath() to populate the tables:

```
call dijaddpath( 'a', 'b',  4 );
call dijaddpath( 'a', 'd',  1 );
call dijaddpath( 'b', 'a', 74 );
call dijaddpath( 'b', 'c',  2 );
call dijaddpath( 'b', 'e', 12 );
call dijaddpath( 'c', 'b', 12 );
call dijaddpath( 'c', 'f', 74 );
call dijaddpath( 'c', 'j', 12 );
call dijaddpath( 'd', 'e', 32 );
call dijaddpath( 'd', 'g', 22 );
call dijaddpath( 'e', 'd', 66 );
call dijaddpath( 'e', 'f', 76 );
```

```
call dijaddpath( 'e', 'h', 33 );
call dijaddpath( 'f', 'i', 11 );
call dijaddpath( 'f', 'j', 21 );
call dijaddpath( 'g', 'd', 12 );
call dijaddpath( 'g', 'h', 10 );
call dijaddpath( 'h', 'g',  2 );
call dijaddpath( 'h', 'i', 72 );
call dijaddpath( 'i', 'f', 31 );
call dijaddpath( 'i', 'j',  7 );
call dijaddpath( 'i', 'h', 18 );
call dijaddpath( 'j', 'f',  8 );

SELECT * FROM dijnodes;
+--------+----------+------+--------+------------+
| nodeID | nodename | cost | pathID | calculated |
+--------+----------+------+--------+------------+
|      1 | a        | NULL |   NULL |          0 |
|      2 | b        | NULL |   NULL |          0 |
|      3 | d        | NULL |   NULL |          0 |
|      4 | c        | NULL |   NULL |          0 |
|      5 | e        | NULL |   NULL |          0 |
|      6 | f        | NULL |   NULL |          0 |
|      7 | j        | NULL |   NULL |          0 |
|      8 | g        | NULL |   NULL |          0 |
|      9 | h        | NULL |   NULL |          0 |
|     10 | i        | NULL |   NULL |          0 |
+--------+----------+------+--------+------------+
SELECT * FROM dijpaths;
+--------+------------+----------+------+
| pathID | fromNodeID | toNodeID | cost |
+--------+------------+----------+------+
|      1 |          1 |        2 |    4 |
|      2 |          1 |        3 |    1 |
|      3 |          2 |        1 |   74 |
|      4 |          2 |        4 |    2 |
|      5 |          2 |        5 |   12 |
|      6 |          4 |        2 |   12 |
|      7 |          4 |        6 |   74 |
|      8 |          4 |        7 |   12 |
|      9 |          3 |        5 |   32 |
|     10 |          3 |        8 |   22 |
|     11 |          5 |        3 |   66 |
|     12 |          5 |        6 |   76 |
|     13 |          5 |        9 |   33 |
|     14 |          6 |       10 |   11 |
|     15 |          6 |        7 |   21 |
|     16 |          8 |        3 |   12 |
|     17 |          8 |        9 |   10 |
|     18 |          9 |        8 |    2 |
|     19 |          9 |       10 |   72 |
|     20 |         10 |        6 |   31 |
|     21 |         10 |        7 |    7 |
|     22 |         10 |        9 |   18 |
|     23 |          7 |        6 |    8 |
+--------+------------+----------+------+
```

Now for the stored procedure, a 6-step:

- null out path columns in the nodes table
- find the nodeIDs referenced by input params
- loop through all uncalculated one-step paths, calculating costs in each
- if a node remains uncalculated, the graph is invalid, so quit
- write the path sequence to a temporary table
- query the temp table to show the result

```
DROP PROCEDURE IF EXISTS dijResolve;
DELIMITER |
```

```
CREATE PROCEDURE dijResolve( pFromNodeName VARCHAR(20), pToNodeName VARCHAR(20) )
BEGIN
  DECLARE vFromNodeID, vToNodeID, vNodeID, vCost, vPathID INT;
  DECLARE vFromNodeName, vToNodeName VARCHAR(20);
  -- null out path info in the nodes table
  UPDATE dijnodes SET PathID = NULL,Cost = NULL,Calculated = 0;
  -- find nodeIDs referenced by input params
  SET vFromNodeID = ( SELECT NodeID FROM dijnodes WHERE NodeName = pFromNodeName );
  IF vFromNodeID IS NULL THEN
    SELECT CONCAT('From node name ', pFromNodeName, ' not found.' );
  ELSE
    BEGIN
      -- start at src node
      SET vNodeID = vFromNodeID;
      SET vToNodeID = ( SELECT NodeID FROM dijnodes WHERE NodeName = pToNodeName );
      IF vToNodeID IS NULL THEN
        SELECT CONCAT('From node name ', pToNodeName, ' not found.' );
      ELSE
        BEGIN
          -- calculate path costs till all are done
          UPDATE dijnodes SET Cost=0 WHERE NodeID = vFromNodeID;
          WHILE vNodeID IS NOT NULL DO
            BEGIN
              UPDATE
                dijnodes AS src
                JOIN dijpaths AS paths ON paths.FromNodeID = src.NodeID
                JOIN dijnodes AS dest ON dest.NodeID = Paths.ToNodeID
              SET dest.Cost = CASE
                                WHEN dest.Cost IS NULL THEN src.Cost + Paths.Cost
                                WHEN src.Cost + Paths.Cost < dest.Cost THEN src.Cost + Paths.Cost
                                ELSE dest.Cost
                              END,
                  dest.PathID = Paths.PathID
              WHERE
                src.NodeID = vNodeID
                AND (dest.Cost IS NULL OR src.Cost + Paths.Cost < dest.Cost)
                AND dest.Calculated = 0;

              UPDATE dijnodes SET Calculated = 1 WHERE NodeID = vNodeID;

              SET vNodeID = ( SELECT nodeID FROM dijnodes
                              WHERE Calculated = 0 AND Cost IS NOT NULL
                              ORDER BY Cost LIMIT 1
                            );
            END;
          END WHILE;
        END;
      END IF;
    END;
  END IF;
  IF EXISTS( SELECT 1 FROM dijnodes WHERE NodeID = vToNodeID AND Cost IS NULL ) THEN
    -- problem,  cannot proceed
    SELECT CONCAT( 'Node ',vNodeID, ' missed.' );
  ELSE
    BEGIN
      -- write itinerary to map table
      DROP TEMPORARY TABLE IF EXISTS map;
      CREATE TEMPORARY TABLE map (
        RowID INT PRIMARY KEY AUTO_INCREMENT,
        FromNodeName VARCHAR(20),
        ToNodeName VARCHAR(20),
        Cost INT
      ) ENGINE=MEMORY;
      WHILE vFromNodeID <> vToNodeID DO
        BEGIN
          SELECT
            src.NodeName,dest.NodeName,dest.Cost,dest.PathID
            INTO vFromNodeName, vToNodeName, vCost, vPathID
```

```
          FROM
            dijnodes AS dest
            JOIN dijpaths AS Paths ON Paths.PathID = dest.PathID
            JOIN dijnodes AS src ON src.NodeID = Paths.FromNodeID
          WHERE dest.NodeID = vToNodeID;

          INSERT INTO Map(FromNodeName,ToNodeName,Cost) VALUES(vFromNodeName,vToNodeName,vCost);

          SET vToNodeID = (SELECT FromNodeID FROM dijPaths WHERE PathID = vPathID);
        END;
      END WHILE;
      SELECT FromNodeName,ToNodeName,Cost FROM Map ORDER BY RowID DESC;
      DROP TEMPORARY TABLE Map;
    END;
  END IF;
END;
|
DELIMITER ;
CALL dijResolve( 'a','i');
+--------------+------------+------+
| FromNodeName | ToNodeName | Cost |
+--------------+------------+------+
| a            | b          |    4 |
| b            | c          |    6 |
| c            | j          |   18 |
| j            | f          |   26 |
| f            | i          |   37 |
+--------------+------------+------+
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Approximate joins

There are two main ways to reconcile payments against charges:

- *Open Item*: match payments against individual charges, typically by carrying the charge number in the payments table ,
- *Statement*: list and sum all charges and all payments, and show the difference as the outstanding balance.

The Open Item method needs a foolproof way to match payments to charges, but what if the customer neglected to return a copy of the invoice, or to write the invoice number on the cheque? Reconciliation staff spend much of their time resolving such problems.

Can we help? Yes! It won't be entirely foolproof, but it will drastically cut down the onerous work of reconciliation.

Here is DDL for a test case:

```
CREATE SCHEMA approx;
USE approx;
CREATE TABLE charges (
  ID INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
  custID INT UNSIGNED,
  amount DECIMAL(10,2) NOT NULL
);
CREATE TABLE payments (
  ID INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
  custID INT UNSIGNED,
  amount DECIMAL( 10,2) NOT NULL
);
```

Both tables carry a `custID` column to identify whose charge or payment it is, but there is no foreign key linking payments to specific charges--that is the link we are going to approximate.

Now populate the tables with a few rows of sample charges and payments for customer #1, ensuring that you have a variety of payments � some that match the charge exactly, some that are close but not enough, and some that are slight overpayments.

```
INSERT INTO approx.charges VALUES
(NULL,1,100),(NULL,1,12),(NULL,1,56),(NULL,1,43),(NULL,1,59),(NULL,1,998);
INSERT INTO approx.payments VALUES
(NULL,1,99),(NULL,1,62),(NULL,1,40),(NULL,1,50),(NULL,1,12),(NULL,1,1000);

SELECT * FROM charges;
+----+--------+--------+
| ID | custID | amount |
+----+--------+--------+
|  1 |      1 | 100.00 |
|  2 |      1 |  12.00 |
|  3 |      1 |  56.00 |
|  4 |      1 |  43.00 |
|  5 |      1 |  59.00 |
|  6 |      1 | 998.00 |
+----+--------+--------+
SELECT * FROM payments;
+----+--------+---------+
| ID | custID | amount  |
+----+--------+---------+
|  1 |      1 |   99.00 |
|  2 |      1 |   62.00 |
|  3 |      1 |   40.00 |
|  4 |      1 |   50.00 |
|  5 |      1 |   12.00 |
|  6 |      1 | 1000.00 |
+----+--------+---------+
```

The first thing to do is define an approximation threshold: how close must the amount paid be to the amount charged before we conclude that the amounts are related? For this example we define the proximity threshold as 2. In a real-world example, it might be 10, or 50, or perhaps percentage of the charge. It all depends on the nature of the organisation and the typical total purchase. A house builder may make frequent purchases valued at $1000 and more. You scale the threshold to the typical situation.

Since the amount paid might be more or less or even equal to the amount charged, to link a payment to a charge we need not an *equi-join* but a *theta-join* that tests a range both below and above the charge amount. That might suggest a `BETWEEN` clause. Here is a better idea: use the `ABS()` function:

```
SET  @proximity = 2;   -- change this value to suit your situation
SELECT
  c.ID AS ChargeNo,
  c.Amount AS Charge,
  p.ID AS PaymentNo,
  p.Amount AS Payment
FROM charges c
JOIN payments p
  ON c.custID = p.custID
  AND ABS(c.amount - p.amount) <= @proximity
WHERE c.custID = 1;
```

Before you run this query, look at the data to anticipate the result.

Here it is:

```
+----------+--------+-----------+---------+
| ChargeNo | Charge | PaymentNo | Payment |
+----------+--------+-----------+---------+
|        1 | 100.00 |         1 |   99.00 |
|        2 |  12.00 |         5 |   12.00 |
|        6 | 998.00 |         6 | 1000.00 |
+----------+--------+-----------+---------+
```

The solution is correct, as far as it goes, but it doesn�t go far enough. We correctly identified the three situations: underpayment, exact payment and overpayment, but we suppressed all charges that don�t have a matching payment. Reconciliation staff are probably interested in a bigger picture of the situation. Fix this by changing the `INNER JOIN` to a `LEFT JOIN`:

```
SET @proximity = 2;
SELECT
  c.ID AS ChargeNo,
  c.amount AS Charge,
  p.ID AS PaymentNo,
  p.amount AS Payment
FROM
  charges c
LEFT JOIN payments p
  ON c.custID = p.custID
  AND ABS(c.amount - p.amount) <= @proximity
WHERE c.custID = 1;
+----------+--------+-----------+---------+
| ChargeNo | Charge | PaymentNo | Payment |
+----------+--------+-----------+---------+
|        1 | 100.00 |         1 |   99.00 |
|        2 |  12.00 |         5 |   12.00 |
|        3 |  56.00 |      NULL |    NULL |
|        4 |  43.00 |      NULL |    NULL |
|        5 |  59.00 |      NULL |    NULL |
|        6 | 998.00 |         6 | 1000.00 |
+----------+--------+-----------+---------+
```

Much better! The reconciliation people now know that three charges have no matching payment.

What if the customer mistakenly pays for something twice? Add a row to the Payments table with an amount of $1000, then re-run the last query:

```
+----------+--------+-----------+---------+
| ChargeNo | Charge | PaymentNo | Payment |
+----------+--------+-----------+---------+
|        1 | 100.00 |         1 |   99.00 |
|        2 |  12.00 |         5 |   12.00 |
|        3 |  56.00 |      NULL |    NULL |
|        4 |  43.00 |      NULL |    NULL |
|        5 |  59.00 |      NULL |    NULL |
|        6 | 998.00 |         6 | 1000.00 |
|        6 | 998.00 |         7 | 1000.00 |
+----------+--------+-----------+---------+
```

How convenient! We can see at once that charge number 6 was paid for twice.

Somebody in the reconciliation department owes you lunch.

Back to the top *Browse the book* *Buy the book* *Feedback*

Cascading JOINs

Show parents, children and grandchildren including parents without children

```
SELECT parent.id AS ParentID,
       IFNULL(child.parent_id,') AS ChildParentID,
       IFNULL(child.id,') AS ChildID,
       IFNULL(grandchild.child_id,') AS GrandchildChildID
FROM parent
  LEFT JOIN child ON parent.id=child.parent_id
  LEFT JOIN grandchild ON child.id=grandchild.child_id;
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Data-driven joins

Data-driven table relationships are hard to maintain, but sometimes they cannot be avoided. How do we build joins for them? One way is to use a `CASE` statement in the `SELECT` list to handle the joining possibilities. In this example, the `parent.linktable` column determines the name of the table where a particular parent row's data is. The method is fine when the number of child tables is small:

```
USE test;
DROP TABLE IF EXISTS parent, child1, child2;

CREATE TABLE parent (
  id INT UNSIGNED PRIMARY KEY,
  linktable CHAR(64) NOT NULL
);
INSERT INTO parent VALUES (1, 'child1'), (2, 'child2');

CREATE TABLE child1 (
  id INT UNSIGNED PRIMARY KEY,
  data CHAR(10)
);
INSERT INTO child1 VALUES (1, 'abc');

CREATE TABLE child2 (
  id INT UNSIGNED PRIMARY KEY,
  data CHAR(10)
);
INSERT INTO child2 VALUES (2, 'def');
```

To retrieve all child data for all parents, include in the `SELECT` list a `CASE` statement which handles all child table possibilities:

```
SELECT
    p.id,
    p.linktable,
    CASE linktable
      WHEN 'child1' THEN c1.data
      WHEN 'child2' THEN c2.data
      ELSE 'Error'
    END AS Data
FROM parent AS p
LEFT JOIN child1 AS c1 ON p.id=c1.id
LEFT JOIN child2 AS c2 ON p.id=c2.id;
+----+-----------+------+
| id | linktable | Data |
+----+-----------+------+
|  1 | child1    | abc  |
|  2 | child2    | def  |
+----+-----------+------+
```

When the number of child tables is too large for a convenient `CASE` statement, `PREPARE` the query in a stored procedure.

(Based on a MySQL Forum post by Felix Geerinckx)

Back to the top *Browse the book* *Buy the book* *Feedback*

Full Outer Join

A `FULL OUTER` join between tables a and b retrieves:

- all rows from a, with matching rows or nulls from b, and
- all rows from b, with matching rows or nulls from a

so for these tables:

```
DROP TABLE IF EXISTS a,b;
CREATE TABLE a(id int,name char(1));
CREATE TABLE b(id int,name char(1));
INSERT INTO a VALUES(1,'a'),(2,'b');
INSERT INTO b VALUES(2,'b'),(3,'c');
SELECT * FROM a;
+------+------+
| id   | name |
+------+------+
|    1 | a    |
|    2 | b    |
+------+------+
SELECT * FROM b;
+------+------+
| id   | name |
+------+------+
|    2 | b    |
|    3 | c    |
+------+------+
```

a full outer join returns:

```
+------+------+------+------+
| id   | name | id   | name |
+------+------+------+------+
|    1 | a    | NULL | NULL |
|    2 | b    |    2 | b    |
| NULL | NULL |    3 | c    |
+------+------+------+------+
```

MySQL does not support `FULL OUTER JOIN`. How to emulate it? If the joining keys of each table are unique, you can just `UNION` left and right joins:

```
SELECT * FROM a LEFT JOIN b ON a.id=b.id
UNION
SELECT * FROM a RIGHT JOIN b ON a.id=b.id;
```

But suppose the tables to be joined have duplicate rows, and you wish your result to preserve them. For example, add a duplicate row to table a:

```
INSERT INTO a VALUES(1,'a');
```

Now `UNION` removes the duplicate row you want preserved in the result. How to get back the desired duplicates? A `FULL OUTER JOIN` consists of:

- an `INNER JOIN` between a and b to catch row matches between a and b,
- a `LEFT EXCLUSION JOIN` from a to b to catch rows that are in a and not in b,
- a `RIGHT EXCLUSION JOIN` from b to a to catch rows in b that are not in a.

In SQL:

```
SELECT * FROM a INNER JOIN b ON a.id=b.id
UNION ALL
SELECT * FROM a LEFT JOIN b ON a.id=b.id WHERE b.id IS NULL
UNION ALL
SELECT * FROM a RIGHT JOIN b ON a.id=b.id WHERE a.id IS NULL
```

But the first two joins—the inner join, and the left exclusion join—are logically equivalent to a left outer join, so we can

write:

```
SELECT * FROM a LEFT JOIN b ON a.id=b.id
UNION ALL
SELECT * FROM a RIGHT JOIN b ON a.id=b.id WHERE a.id IS NULL;
+------+------+------+------+
| id   | name | id   | name |
+------+------+------+------+
|    1 | a    | NULL | NULL |
|    2 | b    |    2 | b    |
|    1 | a    | NULL | NULL |
| NULL | NULL |    3 | c    |
+------+------+------+------+
```

Why doesn't MySQL implement `FULL OUTER JOIN` syntax for this? We don't know.

Back to the top | *Browse the book* | *Buy the book* | *Feedback*

Intersection and difference

MySQL implements `UNION`, but does not directly implement `INTERSECTION` or `DIFFERENCE`.

`INTERSECTION` is just an `INNER JOIN` on all columns:

```
drop table if exists a,b;
create table(i int,j int);
create table b like a;
insert into a values(1,1),(2,2);
insert into a values(1,1),(3,3);
select * from a join b using(a,b);
+------+------+
| i    | j    |
+------+------+
|    1 |    1 |
+------+------+
```

Get the `DIFFERENCE` between tables a and b by `UNION`ing exclusion joins from a to b, and from b to a:

```
select * from a left join b using(i,j) where b.i is null
union
select * from b left join a using(i,j) where a.i is null;
+------+------+
| i    | j    |
+------+------+
|    2 |    2 |
|    3 |    3 |
+------+------+
```

Back to the top | *Browse the book* | *Buy the book* | *Feedback*

Many-to-many joins

We have a collection of articles and users' scores of them. How to report statistics on these scores? We need three tables--one for articles, one for users, and a *bridge* table where each row represents one score on one article by one user:

```
DROP TABLE IF EXISTS art_articles;
CREATE TABLE art_articles (
  ID INT AUTO_INCREMENT PRIMARY KEY,
  title CHAR(30),
  txt TEXT,
  UNIQUE KEY (title)
);
INSERT INTO art_articles VALUES (1,'abc',''),(2,'def',''),(3,'ghi',''),(4,'jkl','');
```

```
DROP TABLE IF EXISTS art_users;
CREATE TABLE art_users(
  ID INT AUTO_INCREMENT PRIMARY KEY,
  name CHAR(20)
);
INSERT INTO art_users VALUES (1,'A'),(2,'B');

DROP TABLE IF EXISTS art_scores;
CREATE TABLE art_scores (
  id INT AUTO_INCREMENT PRIMARY KEY,
  articleID INT NOT NULL, -- references article.articleID
  userID INT NOT NULL,    -- references user.userID
  score DECIMAL(6,2)
);
INSERT INTO art_scores VALUES (1,1,1,80),(2,1,2,90),(3,2,2,60);

-- find average score for article titled 'abc'
SELECT a.title, AVG( s.score ) AS Art1Avg
FROM art_articles a
JOIN art_scores s ON a.id=s.articleID
WHERE a.title='abc'
GROUP BY a.title;

-- find average score submitted by user 1
SELECT u.name, AVG( s.score ) AS User1Avg
FROM art_users u
JOIN art_scores s ON u.id=s.userID
WHERE u.id = 1
GROUP BY u.name;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

What else did buyers of X buy?

We often wish to know which purchases are associated with which other purchases, ie "people who bought this item also bought ..." In a real-world database, the table that summarises this information might be a View that encapsulates joins from customers to orders to orderitems to products, perhaps scoped on a recent date range. Here we'll ignore all such detail, focussing only on the logic of three typical problems of this type:

```
DROP TABLE IF EXISTS userpurchases;
CREATE TABLE userpurchases(
  custID INT UNSIGNED,
  prodID INT UNSIGNED
);
INSERT INTO userpurchases
VALUES (1,1),(1,2),(2,4),(3,1),(3,2),(4,2),(4,3),(5,1),(5,2),(5,3);
SELECT custID, GROUP_CONCAT(prodID ORDER BY prodID) AS PurchaseList
FROM userpurchases
GROUP BY custID;
+--------+--------------+
| custID | PurchaseList |
+--------+--------------+
|      1 | 1,2          |
|      2 | 4            |
|      3 | 1,2          |
|      4 | 2,3          |
|      5 | 1,2,3        |
+--------+--------------+
```

The basic idea is to self-join on the product ID as often as need be to get the answer. For example, to list all products bought by customers who'd already bought at least one other product, join userpurchases to itself on matching custIDs and non-matching prodIDs:

```
SELECT DISTINCT p2.prodid
FROM userpurchases p1
JOIN userpurchases p2 ON p1.custID = p2.custID AND p1.prodID <> p2.prodID;
+--------+
| prodid |
+--------+
|      1 |
|      2 |
|      3 |
+--------+
```

To find what else buyers of product 1 bought, copy the above join and group by customer ID:

```
SELECT p1.custID,GROUP_CONCAT(p2.prodid) as 'Buyers of #1 Also bought'
FROM userpurchases p1
JOIN userpurchases p2 ON p1.custID=p2.custID AND p1.prodID <> p2.prodID
WHERE p1.prodID = 1
GROUP BY p1.custID;
+--------+--------------------------+
| custID | Buyers of #1 Also bought |
+--------+--------------------------+
|      1 | 2                        |
|      3 | 2                        |
|      5 | 2,3                      |
+--------+--------------------------+
```

What customers bought both product 1 and product 2?

```
SELECT DISTINCT p1.custID
FROM userpurchases p1
JOIN userpurchases p2 ON p1.custID=p2.custID AND p1.prodID=1 AND p2.prodID=2
+--------+
| custID |
+--------+
|      1 |
|      3 |
|      5 |
+--------+
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Join or subquery?

Usually, a `JOIN` is faster than an uncorrelated subquery. For example in the sakila test database, customer is a parent of rental (via customer_id) which in turn is a parent of payment (via rental_id). The subquery version of a query for whether a customer has made payments and rentals...

```
SELECT DISTINCT c.customer_id
FROM customer c
WHERE c.customer_id IN (
   SELECT r.customer_id
   FROM rental r
   JOIN payment p USING (rental_id)
   WHERE c.customer_id = 599;
);
```

is eight times slower than the join version...

```
SELECT DISTINCT c.customer_id
FROM customer c
JOIN rental r USING (customer_id)
JOIN payment p USING (rental_id)
WHERE c.customer_id = 599;
```

Running `EXPLAIN` on the two queries reveals why: the subquery version has to read most customer rows, while the join version proceeds inside out and discovers it needs to read just one customer row.

Back to the top *Browse the book* *Buy the book* *Feedback*

Parents without children

Given tables `parent(id INT), child(id INT,parent_id INT)`, how to find parents that have no children? This is a simple version of the Not Exists query pattern, which can be written as an *exclusion join*...

```
SELECT parent.id
FROM parent
LEFT JOIN child ON parent.id = child.parent_id
WHERE child.parent_id IS NULL;
```

or with a `NOT EXISTS` subquery, which is logically equivalent to the exclusion join, but usually much slower:

```
SELECT parent.id AS ParentID
FROM parent
WHERE NOT EXISTS (
  SELECT parent.id
  FROM parent
  JOIN child ON parent.ID = child.parent_id
);
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Parties who have contracts with one another

You have a parties table that holds info on peoples' names etc, and a contracts table where each row has clientID and contractorID value pointing at a parties.partyID value--that is, each contracts row points at two parties rows. You want to list the names of all contractors and their clients.

```
SELECT clientpartyID,
       pCli.name AS Client,
       contractorpartyID,
       pCon.name AS Contractor
FROM contracts
  INNER JOIN parties AS pCli
    ON contracts.clientpartyID = pCli.partyID
  INNER JOIN parties AS pCon
    ON contracts.contractorpartyID = pCon.partyID;
```

Back to the top *Browse the book* *Buy the book* *Feedback*

The unbearable slowness of IN()

`IN()` subqueries can be impossibly slow. This query for orders from NorthWind customers who have made multiple orders...

```
SELECT orderID, discount
FROM orderdetails
WHERE orderID IN (
  SELECT orderID FROM orderdetails
  GROUP BY orderID
  HAVING COUNT(orderID)>1
);
```

takes half again as long to execute as this logically equivalent query using a correlated subquery in an `EXISTS()` clause ...

```
SELECT od.orderID, od.discount
```

```
FROM orderdetails od
WHERE EXISTS (
  SELECT orderID FROM orderdetails
  WHERE orderID = od.orderID
  GROUP BY orderID
  HAVING COUNT(orderID)>1
);
```

and is *50 times slower* than this logically equivalent query which moves the subquery to the `FROM` clause:

```
SELECT o.orderID, discount
FROM orderdetails AS o
INNER JOIN (
  SELECT orderID
  FROM orderdetails
  GROUP BY orderID
  HAVING COUNT(1) > 1
) AS t ON o.orderID=t.orderID;
```

Why? Both the `IN()` and `EXISTS()` queries have to execute a table scan for each row in the table. Performance degrades as the square of the number of rows. The `JOIN` version builds its derived table on one table scan, and quickly picks off its resultset from that.

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

The [Not] Exists query pattern

Given a table `employee( employeeID INT, mgr_employeeID INT, salary DECIMAL(10,2))`, find the managers who earn less than at least one of their subordinates.

We can write this query directly from the logic of its spec...

```
SELECT DISTINCT employeeID
FROM employee AS e
WHERE EXISTS (
  SELECT employeeID
  FROM employee AS m
  WHERE m.mgr_employeeID = e.employeeID AND e.salary > m.salary
);
```

...but a JOIN (or decorrelated) version of the logic is usually much faster. This query pattern is simple:

- Inner join the table (t1) to itself (t2) on the grouping key.
- Add the condition on which you wish to find existing rows to the Join clause.

```
SELECT DISTINCT m.employeeID
FROM employee AS e
INNER JOIN employee AS m ON e.Mgr = m.employeeID AND e.salary > m.salary;
```

The correlated subquery version of the Not Exists pattern merely inserts a strategic NOT:

```
SELECT DISTINCT employeeID
FROM employee AS e
WHERE NOT EXISTS (
  SELECT employeeID
  FROM employee AS m
  WHERE m.Mgr = e.employeeID AND e.salary > m.salary
);
```

The decorrelated version of the *Not* Exists version of this query uses an *exclusion join*--a `LEFT JOIN` with an `IS NULL` condition imposed on the right side of the join:

- *Left* join the table (t1) to itself (t2) on the grouping key.

- Add the condition on which you wish to find existing rows to the Join clause.
- For the condition on which you wish to find missing rows in t2,
- (a) add the t2 value condition to the Join clause, and
- (b) add 't2 grouping key is null' to the Where clause:

```
SELECT DISTINCT m.employeeID
FROM employee AS e
LEFT JOIN employee AS m ON e.Mgr = m.employeeID AND e.salary > m.salary
WHERE m.employeeID IS NULL;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

What exams did a student not register for?

We have a `students` table, an `exams` table, and a `registrations` (bridge) table that records which students are registered for which exams. How do we find the exams for which a particular student is *not* registered?

```
CREATE TABLE students (
  sid int(10) unsigned PRIMARY KEY auto_increment,
  firstname varchar(45) NOT NULL default '',
  lastname varchar(45) NOT NULL default ''
);
INSERT INTO students VALUES
(1, 'Jack', 'Malone'),(2, 'Hiro', 'Nakamura'),(3, 'Bree', 'Van de Kamp'),
(4, 'Susan', 'Mayer'),(5, 'Matt', 'Parkman'),(6, 'Claire', 'Bennet');

CREATE TABLE exams (
  eid int(10) unsigned PRIMARY KEY auto_increment,
  exam_name varchar(45) NOT NULL default '',
  active smallint(5) unsigned NOT NULL default '0'
);
INSERT INTO exams VALUES
(1, 'Javascript Expert', 1),(2, 'Lost Survival Course', 0),(3, 'Zend PHP Certification', 1);
(4, 'Superhero Advanced Skills', 1),(5, 'Desperation Certificate', 1);

CREATE TABLE registrations (
  registration_id int(11) PRIMARY KEY auto_increment,
  eid int(10) unsigned NOT NULL default '0',
  sid int(10) unsigned NOT NULL default '0',
  registration_date datetime NOT NULL default '0000-00-00 00:00:00'
);
INSERT INTO registrations (registration_id, eid, sid, registration_date) VALUES
(1, 5, 14, '2007-10-25 00:00:00'),(2, 5, 3, '0000-00-00 00:00:00'),
(3, 5, 4, '2007-10-23 00:00:00'),(4, 4, 2, '2007-10-16 00:00:00'),
(5, 4, 5, '2007-10-22 00:00:00'),(6, 4, 6, '2007-10-23 00:00:00'),
(7, 5, 2, '2007-10-23 00:00:00');
```

It's a variation of the [Not] Exists query pattern, but there is a wrinkle: we might expect to join registrations to students to get student info into the result, yet the registrations table will be the object of the exclusion join, so how do we retrieve the required student info? One solution is to left join exams to (students left join registrations), permitting us to retrieve the required student info while imposing the `registration_id IS NULL` condition on registrations to show only the exams he is not taking:

```
SELECT e.exam_name
FROM exams e
LEFT JOIN (
  students s
  LEFT JOIN registrations r
        ON s.sid=r.sid AND s.firstname='Hiro' AND s.lastname='Nakamura'
) ON e.eid=r.eid
WHERE r.registration_id IS NULL;
+-----------------------+
| exam_name             |
```

```
+------------------------+
| Javascript Expert      |
| Lost Survival Course   |
| Zend PHP Certification |
+------------------------+
```

We can prove that this logic is correct by modifying the query to show Hiro's registration or not for all exams:

```
SELECT e.exam_name,IF(s.sid IS NULL, 'No', 'Yes') AS 'Hiro registered'
FROM exams e
LEFT JOIN (
  students s
  LEFT JOIN registrations r
    ON s.sid=r.sid
    AND s.names='Hiro'
    AND s.surnames='Nakamura'
) ON e.eid=r.eid;
+---------------------------+-----------------+
| exam_name                 | Hiro registered |
+---------------------------+-----------------+
| Javascript Expert         | No              |
| Lost Survival Course      | No              |
| Zend PHP Certification    | No              |
| Superhero Advanced Skills | Yes             |
| Desperation Certificate   | Yes             |
+---------------------------+-----------------+
```

Thanks to Pascal Mitride for the example.

Back to the top *Browse the book* *Buy the book* *Feedback*

List NULLs at end of query output

If ordering by `col`...

```
... ORDER BY IF(col IS NULL, 1, 0 ), col ...
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Parents with and without children

You have parties and contracts tables. Every contracts row has a contractorpartyID value which references a row in parties, and a clientpartyID value which also references a row in parties. How to list all parties and their contracts, showing blanks as empty strings rather than NULLs?

```
SELECT parties.partyID,
       IFNULL(contractorpartyID,'') AS contractor,
       IFNULL(clientpartyID,'') AS client
FROM parties
LEFT JOIN contractor_client ON partyID=contractorpartyID
ORDER BY partyID;
+---------+------------+--------+
| partyID | contractor | client |
+---------+------------+--------+
|       1 |            |        |
|       2 | 2          | 1      |
|       3 |            |        |
+---------+------------+--------+
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Next row

You have a table of names, you have retrieved a row with name $name, and you want the row for the next name in name order. MySQL LIMIT syntax makes this very easy:

```
SELECT *
FROM tbl
WHERE name > $name
ORDER BY name
LIMIT 1
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Order by a column containing digits and letters

To have column values 1abc,10abc,8abc appear in the expected order 1abc,8abc,10abc, take advantage of a trick built into MySQL string parsing ...

```
SELECT '3xyz'+0;
+----------+
| '3xyz'+0 |
+----------+
|        3 |
+----------+
```

to write ...

```
SELECT ...
...
ORDER BY colname+0, colname;
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Order by month name

The MySQL FIELD(str,str1,str2,...,strN) function returns 1 if str=str1, 2 if str=str2, etc., so ...

```
SELECT .
ORDER BY FIELD(month,'JAN','FEB','MAR',...,'NOV','DEC') .
```

will order query output from a legacy table in month-number order.

Back to the top *Browse the book* *Buy the book* *Feedback*

Suppress repeating ordering values

You have tables tracking authors and their books, for example:

```
CREATE TABLE author (
  id int(4) NOT NULL auto_increment PRIMARY KEY,
  name text NOT NULL
);
INSERT INTO author (id, name)
VALUES (1,'Brad Phillips'),(2,'Don Charles'),(3,'Kur Silver');
CREATE TABLE book (
  id int(4) NOT NULL auto_increment PRIMARY KEY,
  name text NOT NULL
);
INSERT INTO book (id, name)
VALUES (1,'MySQL in a bucket '),(2,'Databases for Delinquents'),
```

```
      (3,'Design Patterns'),(4,'PHP Professional'),(5,'Java Script Programming');
CREATE TABLE book_author (
  book_id int(4) NOT NULL default '0',
  author_id int(4) NOT NULL default '0'
);
INSERT INTO book_author (book_id, author_id)
VALUES (1,1), (1,2), (2,3), (4,1), (3,1), (5,2);
```

You want to list authors' books while suppressing repeating authors' names. A simple solution is to use MySQL's extremely useful GROUP_CONCAT() function to group books by author:

```
SELECT
  a.name AS Author,
  GROUP_CONCAT(b.name ORDER BY b.name) AS Books
FROM book_author AS ba
JOIN book AS b ON ba.book_id=b.id
JOIN author AS a ON ba.author_id=a.id
GROUP BY a.name;
```

For a neater-looking result:
1. Retrieve authors and their books.
2. Order them
3. Use a variable to remember and suppress repeating author names:

```
SET @last='';
SELECT
  IF(r.author=@last,'',@last:=r.author) AS Author,
  r.book AS Book
FROM (
  SELECT DISTINCT a.name AS author,b.name AS book
  FROM book_author AS ba
  JOIN book AS b ON ba.book_id=b.id
  JOIN author AS a ON ba.author_id=a.id
  ORDER BY a.name,b.name
) AS r;
+---------------+--------------------------+
| author        | book                     |
+---------------+--------------------------+
| Brad Phillips | Design Patterns          |
|               | MySQL in a bucket        |
|               | PHP Professional         |
| Don Charles   | Java Script Programming  |
|               | MySQL in a bucket        |
| Kur Silver    | Databases for Delinquents |
+---------------+--------------------------+
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Pagination

Suppose you have a phone book of names, addresses, etc. You are looking at page 100 and want to now see page 99 ... How do you do this knowing only what page you are on and the name at the top of the page?

You don't need the first name on the page. Assuming...

- 1-based page numbers
- you are on page P
- each page shows N rows

then a SELECT statement that reproduces your page, assuming no rows have changed, ends with
`LIMIT ((P-1)*N, N)`
so you could retrieve page 99 with a SELECT statement which ends with
`LIMIT (98*N,N).`

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Automate the writing of pivot table queries

You have a `sales` table listing product, salesperson and amount:

```
DROP TABLE IF EXISTS sales;
CREATE TABLE sales (
  id int(11) default NULL,
  product char(5) default NULL,
  salesperson char(5) default NULL,
  amount decimal(10,2) default NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
INSERT INTO sales VALUES
  (1,'radio','bob','100.00'),
  (2,'radio','sam','100.00'),
  (3,'radio','sam','100.00'),
  (4,'tv','bob','200.00'),
  (5,'tv','sam','300.00'),
  (6,'radio','bob','100.00');
SELECT * FROM sales;
+------+---------+-------------+--------+
| id   | product | salesperson | amount |
+------+---------+-------------+--------+
|    1 | radio   | bob         | 100.00 |
|    2 | radio   | sam         | 100.00 |
|    3 | radio   | sam         | 100.00 |
|    4 | tv      | bob         | 200.00 |
|    5 | tv      | sam         | 300.00 |
|    6 | radio   | bob         | 100.00 |
+------+---------+-------------+--------+
```

If you are asked to tabulate sales amount against salesperson and product, you write a pivot table query:

```
SELECT
  product,
  SUM( CASE salesperson WHEN 'bob' THEN amount ELSE 0 END ) AS 'Bob',
  SUM( CASE salesperson WHEN 'sam' THEN amount ELSE 0 END ) AS 'Sam',
  SUM( amount ) AS Total
FROM sales
GROUP BY product WITH ROLLUP;
+---------+--------+--------+--------+
| product | Bob    | Sam    | Total  |
+---------+--------+--------+--------+
| radio   | 200.00 | 200.00 | 400.00 |
| tv      | 200.00 | 300.00 | 500.00 |
| NULL    | 400.00 | 500.00 | 900.00 |
+---------+--------+--------+--------+
```

The query generates one product per row and one column per salesperson. The pivoting `CASE` expressions assign values of `sales.amount` to the matching salesperson's column. For two products and two salespersons, it's a snap once you've done it a few times. When there are dozens of products and salespersons, though, writing the query becomes tiresome and error-prone.

Some years ago Giuseppe Maxia published a little query that automates writing the pivot expressions. His idea was to embed the syntax for lines like the `SUM( CASE ...)` lines above in a query for the `DISTINCT` values. At the time Giuseppe was writing, MySQL did not support stored procedures. Now that it does, we can further generalise Giuseppe's idea by parameterising it in a stored procedure.

Admittedly, it's a little daunting. To write a query with variable names rather than the usual literal table and column names, we have to write `PREPARE` statements. What we propose to do here is to write SQL that writes `PREPARE` statements.

Code which writes code which writes code. Not a job for the back of a napkin.

It's easy enough to write the sproc shell. We keep generic queries in a sys database, so the routine needs parameters specifying database, table, pivot column and (in some cases) the aggregating column. Then what? What worked for us was to proceed from back to front:

- Write the pivot expressions for a specific case.
- Write the PREPARE statement that generates those expressions.
- Parameterise the result of #2.
- Put the result of #3 in an sproc.

Further complicating matters, we soon found that different summary aggregations, for example COUNT and SUM, require different sprocs. Here is the routine for generating COUNT pivot expressions:

```
USE sys;
DROP PROCEDURE IF EXISTS writecountpivot;
DELIMITER |
CREATE PROCEDURE writecountpivot( db CHAR(64), tbl CHAR(64), col CHAR(64) )
BEGIN
  DECLARE datadelim CHAR(1) DEFAULT '"';
  DECLARE singlequote CHAR(1) DEFAULT CHAR(39);
  DECLARE comma CHAR(1) DEFAULT ',';
  SET @sqlmode = (SELECT @@sql_mode);
  SET @@sql_mode='';
  SET @sql = CONCAT( 'SELECT DISTINCT CONCAT(', singlequote,
                     ',SUM(IF(', col, ' = ', datadelim, singlequote, comma,
                     col, comma, singlequote, datadelim, comma, '1,0)) AS `',
                     singlequote, comma, col, comma, singlequote, '`', singlequote,
                     ') AS countpivotarg FROM ', db, '.', tbl,
                     ' WHERE ', col, ' IS NOT NULL' );
  -- UNCOMMENT TO SEE THE MIDLEVEL CODE:
  -- SELECT @sql;
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
  DROP PREPARE stmt;
  SET @@sql_mode=@sqlmode;
END;
|
DELIMITER ;
CALL sys.writecountpivot('test','sales','salesperson');
```

This generates the SQL ...

```
SELECT DISTINCT
  CONCAT(',SUM(IF(salesperson = "',salesperson,'",1,0)) AS `',salesperson,'`')
  AS countpivotarg
FROM test.sales
WHERE salesperson IS NOT NULL |
```

and returns...

```
+--------------------------------------------+
| countpivotarg                              |
+--------------------------------------------+
| ,SUM(IF(salesperson = "bob",1,0)) AS `bob` |
| ,SUM(IF(salesperson = "sam",1,0)) AS `sam` |
+--------------------------------------------+
```

which we plug into ...

```
SELECT
  product
  ,SUM(IF(salesperson = "bob",1,0)) AS `bob`
  ,SUM(IF(salesperson = "sam",1,0)) AS `sam`
  ,COUNT(*) AS Total
FROM test.sales
GROUP BY product WITH ROLLUP;
```

```
+---------+------+------+-------+
| product | bob  | sam  | Total |
+---------+------+------+-------+
| radio   |    2 |    2 |     4 |
| tv      |    1 |    1 |     2 |
| NULL    |    3 |    3 |     6 |
+---------+------+------+-------+
```

Not overwhelming for two columns, very convenient if there are 20. (Yes, it could also be written with `COUNT( ... 1, NULL)`).

One point to notice is that the two levels of code generation create quotemark nesting problems. To make the double quotemark "" available for data value delimiting, we turn off `ANSI_QUOTES` during code generation, and put it back afterwards.

`SUM` pivot queries need different syntax:

```
USE sys;
DROP PROCEDURE IF EXISTS writesumpivot;
DELIMITER |
CREATE PROCEDURE writesumpivot( db CHAR(64), tbl CHAR(64), pivotcol CHAR(64), sumcol CHAR(64) )
BEGIN
  DECLARE datadelim CHAR(1) DEFAULT '"';
  DECLARE comma CHAR(1) DEFAULT ',';
  DECLARE singlequote CHAR(1) DEFAULT CHAR(39);
  SET @sqlmode = (SELECT @@sql_mode);
  SET @@sql_mode='';
  SET @sql = CONCAT( 'SELECT DISTINCT CONCAT(', singlequote,
                     ',SUM(IF(', pivotcol, ' = ', datadelim, singlequote, comma,
                     pivotcol, comma, singlequote, datadelim, comma, sumcol, ',0)) AS `',
                     singlequote, comma, pivotcol, comma, singlequote, '`', singlequote,
                     ') AS sumpivotarg FROM ', db, '.', tbl,
                     ' WHERE ', pivotcol, ' IS NOT NULL' );
   -- UNCOMMENT TO SEE THE MIDLEVEL SQL:
   -- SELECT @sql;
   PREPARE stmt FROM @sql;
   EXECUTE stmt;
   DROP PREPARE stmt;
   SET @@sql_mode=@sqlmode;
END;
|
DELIMITER ;
CALL writesumpivot('test','sales','salesperson','amount');
+-----------------------------------------------+
| sumpivotarg                                   |
+-----------------------------------------------+
| ,SUM(IF(salesperson = "bob",amount,0)) AS `bob` |
| ,SUM(IF(salesperson = "sam",amount,0)) AS `sam` |
+-----------------------------------------------+
```

which forms the guts of our report query:

```
SELECT
 product
 ,SUM(IF(salesperson = "bob",amount,0)) AS `bob`
 ,SUM(IF(salesperson = "sam",amount,0)) AS `sam`
 ,SUM(amount) AS Total
FROM test.sales
GROUP BY product;
+---------+--------+--------+--------+
| product | bob    | sam    | Total  |
+---------+--------+--------+--------+
| radio   | 200.00 | 200.00 | 400.00 |
| tv      | 200.00 | 300.00 | 500.00 |
+---------+--------+--------+--------+
```

There are higher levels of generality beckoning---say, a routine that generates a complete pivot table query, not just the pivot expressions.

Column value associations

From table `tbl( lastname CHAR(20),firstname CHAR(20))`, you want to list every lastname in the table, each on one line, with a comma-separated list of every firstname it is associated with.

The simplest solution is to create the pivot table with a self-join and `GROUP_CONCAT()`:

```
SELECT
  t1.lastname,
  GROUP_CONCAT(t2.firstname ORDER BY t2.firstname SEPARATOR ', ')
FROM tbl AS t1 INNER JOIN tbl AS t2 USING (id)
GROUP BY t1.lastname;
```

This is easily generalised to n-tuples. For a personnel table ...

```
CREATE TABLE tbl(id INT,colID INT,value CHAR(20));
INSERT INTO tbl VALUES
  (1,1,'Sampo'),(1,2,'Kallinen'),(1,3,'Office Manager'),
  (2,1,'Jakko'),(2,2,'Salovaara'),(2,3,'Vice President');
```

where colID=1,2,3 mean, respectively, first name,last name,job title...

```
ID colID value
1   1   1 Sampo
1   2   1 Kallinen
1   3   1 Office Manager
2   1   2 Jakko
2   2   2 Salovaara
2   3   2 Vice President
```

you can pivot first name, last name and job title on ID with this query:

```
SELECT
  id,
  GROUP_CONCAT(if(colID = 1, value, NULL)) AS 'First Name',
  GROUP_CONCAT(if(colID = 2, value, NULL)) AS 'Last Name',
  GROUP_CONCAT(if(colID = 3, value, NULL)) AS 'Job Title'
FROM tbl
GROUP BY id;
```


Group column statistics in rows

A *pivot* (or *crosstab*, or *contingency*) table aggregates sets of column values into rows of statistics, and *pivots* target value statistics on partitioning criteria defined by any available data.

Spreadsheet applications have intuitive point-and-click interfaces for generating pivot tables. RDBMSs generally do not. The task looks difficult in SQL, though, only until you have coded a few.

If you ported the Microsoft sample database `Northwind` to your MySQL database (as described in chapter 11 of [Get It Done with MySQL](#)), you can execute this example step by step. Even if you haven't ported Northwind, the example is easy to follow.

Amongst the tables in the Northwind database are:

```
employees(employeeID, lastname, firstname, ...)
orders(orderID, customerId, employeeID, orderdate, ...)
```

There are nine employees, and 803 orders dated from 1996 through 1998. Each order points to an employeeID. Suppose we wish to report counts of orders taken by employees *pivoted on year*--how would we proceed?

We do the pivot table 3-step:

1. *Write the basic aggregating query*, a `GROUP BY` query to aggregate the data on desired variables (in this case, employee and year).

2. *Write the pivoting query* as an outer query that creates a column for each pivot value (year) from #1 written as an inner subquery. (Sometimes it is more efficient to write the results of #1 to a temp table and write #2 to refer to the temp table.)

3. *Fix a `ROLLUP` display glitch* by encapsulating #2 in a new outer query that labels the `ROLLUP` row meaningfully.

Here are the three steps in more detail:

1. Group the joined counts by the two criteria, employee and order year, yielding one result row per employee per year:

```
SELECT
  CONCAT(firstname,' ',lastname) AS 'Employee',
  YEAR(OrderDate) AS col,
  COUNT(*) AS Data
FROM Employees e
JOIN Orders o ON e.EmployeeID = o.EmployeeID
GROUP BY e.employeeID, YEAR(o.OrderDate);
+------------------+------+------+
| Employee         | col  | Data |
+------------------+------+------+
| Nancy Davolio    | 1996 |   26 |
| Nancy Davolio    | 1997 |   55 |
| Nancy Davolio    | 1998 |   42 |
| Andrew Fuller    | 1996 |   16 |
| Andrew Fuller    | 1997 |   41 |
| Andrew Fuller    | 1998 |   39 |
| Janet Leverling  | 1996 |   18 |
| Janet Leverling  | 1997 |   71 |
| Janet Leverling  | 1998 |   38 |
| Margaret Peacock | 1996 |   31 |
| Margaret Peacock | 1997 |   81 |
| Margaret Peacock | 1998 |   44 |
| Steven Buchanan  | 1996 |   11 |
| Steven Buchanan  | 1997 |   18 |
| Steven Buchanan  | 1998 |   13 |
| Michael Suyama   | 1996 |   15 |
| Michael Suyama   | 1997 |   33 |
| Michael Suyama   | 1998 |   19 |
| Robert King      | 1996 |   11 |
| Robert King      | 1997 |   36 |
| Robert King      | 1998 |   25 |
| Laura Callahan   | 1996 |   19 |
| Laura Callahan   | 1997 |   54 |
| Laura Callahan   | 1998 |   31 |
| Anne Dodsworth   | 1996 |    5 |
| Anne Dodsworth   | 1997 |   19 |
| Anne Dodsworth   | 1998 |   19 |
+------------------+------+------+
```

Nine employees for three years yield 27 aggregated rows.

2. We want one summary row per employee, and one count column for each year when an employee took an order. We *pivot* the rows of the above resultset on year by querying the above resultset, defining a colunn for every `year` found, for example:

```
  SUM( CASE col WHEN '1996' THEN data ELSE 0 END ) AS '1996',
```

grouping the result by `row WITH ROLLUP` to provide a row of column sums at the bottom. This gives the following query:

```
SELECT
  Employee,
  SUM( CASE col WHEN '1996' THEN data ELSE 0 END ) AS '1996',
  SUM( CASE col WHEN '1997' THEN data ELSE 0 END ) AS '1997',
  SUM( CASE col WHEN '1998' THEN data ELSE 0 END ) AS '1998',
  SUM( data ) AS Total    -- sums across years by employee
FROM (
  SELECT                  -- the query from step #1
    CONCAT(firstname,' ',lastname) AS 'Employee',
    YEAR(OrderDate) AS 'col',
    COUNT(*) AS Data
  FROM Employees e
  JOIN Orders o ON e.EmployeeID = o.EmployeeID
  GROUP BY e.employeeID, YEAR(o.OrderDate)
) AS stats
GROUP BY employee WITH ROLLUP;
+------------------+------+------+------+-------+
| Employee         | 1996 | 1997 | 1998 | Total |
+------------------+------+------+------+-------+
| Andrew Fuller    |   16 |   41 |   39 |    96 |
| Anne Dodsworth   |    5 |   19 |   19 |    43 |
| Janet Leverling  |   18 |   71 |   38 |   127 |
| Laura Callahan   |   19 |   54 |   31 |   104 |
| Margaret Peacock |   31 |   81 |   44 |   156 |
| Michael Suyama   |   15 |   33 |   19 |    67 |
| Nancy Davolio    |   26 |   55 |   42 |   123 |
| Robert King      |   11 |   36 |   25 |    72 |
| Steven Buchanan  |   11 |   18 |   13 |    42 |
| NULL             |  152 |  408 |  270 |   830 |
+------------------+------+------+------+-------+
```

3. The result of #2 is correct except that sums ought not to be reported as NULL! We fix that bit of weirdness by writing query #2 as a derived table, and having the new outer query alias the yearly sums row:

```
SELECT
  IFNULL( employee, 'SUMS') AS Employee, 1996, 1997, 1998, Total
FROM (
  SELECT
    Employee,
    SUM( CASE col WHEN '1996' THEN data ELSE 0 END ) AS '1996',
    SUM( CASE col WHEN '1997' THEN data ELSE 0 END ) AS '1997',
    SUM( CASE col WHEN '1998' THEN data ELSE 0 END ) AS '1998',
    SUM( data ) AS Total
  FROM (
    SELECT
      CONCAT(firstname,' ',lastname) AS 'Employee',
      YEAR(OrderDate) AS 'col',
      COUNT(*) AS Data
    FROM Employees e
    JOIN Orders o ON e.EmployeeID = o.EmployeeID
    GROUP BY e.employeeID, YEAR(o.OrderDate)
  ) AS stats
  GROUP BY employee WITH ROLLUP
) AS stats2;
+------------------+------+------+------+-------+
| Employee         | 1996 | 1997 | 1998 | Total |
+------------------+------+------+------+-------+
| Andrew Fuller    | 1996 | 1997 | 1998 |    96 |
| Anne Dodsworth   | 1996 | 1997 | 1998 |    43 |
| Janet Leverling  | 1996 | 1997 | 1998 |   127 |
| Laura Callahan   | 1996 | 1997 | 1998 |   104 |
| Margaret Peacock | 1996 | 1997 | 1998 |   156 |
| Michael Suyama   | 1996 | 1997 | 1998 |    67 |
| Nancy Davolio    | 1996 | 1997 | 1998 |   123 |
| Robert King      | 1996 | 1997 | 1998 |    72 |
| Steven Buchanan  | 1996 | 1997 | 1998 |    42 |
```

```
| Sums             | 1996 | 1997 | 1998 |   830 |
+------------------+------+------+------+-------+
```

With multiple statistics and pivot layers, a pivot table query can get complex, but following this 3-step will keep things clear.

Back to the top *Browse the book* *Buy the book* *Feedback*

Pivot table using math tricks

http://en.wikibooks.org/wiki/Programming:MySQL/Pivot_table

Back to the top *Browse the book* *Buy the book* *Feedback*

Pivot table with CONCAT

Here is a MySQL pivot table query for room bookings by weekday:

```
SELECT slot
  , max(if(day=1, concat(subject,' ',room), '')) as day1
  , max(if(day=2, concat(subject,' ',room), '')) as day2
  , max(if(day=3, concat(subject,' ',room), '')) as day3
  , max(if(day=4, concat(subject,' ',room), '')) as day4
  , max(if(day=5, concat(subject,' ',room), '')) as day5
from schedule
group by slot
```

MAX(...) decides between an entry and a blank (the entry will win if one exists) while the group by lines everything up on the same row. Friendly caution: If more than one entry exists for the same day and time, you will only see the one that is alphabetically "greater".

To see how many classes are scheduled by day for each slot (to check for conflicts) try:

```
SELECT slot
  , sum(if(day=1,1,0)) as day1
  , sum(if(day=2,1,0)) as day2
  , sum(if(day=3,1,0)) as day3
  , sum(if(day=4,1,0)) as day4
  , sum(if(day=5,1,0)) as day5
from schedule
group by slot
```

There is a pattern:

- Columns you want as "row headers" are listed both in the SELECT _and_ in the GROUP BY clauses
- Values you want as columns are selectively chosen by IF() functions to return something or nothing so that one of the aggregate functions (MIN, MAX, SUM,AVG, etc) can condense those rows and columns into single values

What to count in any column can be as complex as you like. If you don't like the IF() function you can use CASE statements or anything else to help you provide the correct set of values for the aggregate function you are using on that column. You can mix and match aggregate functions to be what you need. Let's say you join the appropriate tables together to form a data set that looks like:

```
day, slot, subject, student, grade
```

and you wanted to build a chart showing the statistics of grades vs. days and subject. You could use this type of query to work that out:

```
SELECT day, subject
  , AVG(grade) as average
  , MIN(grade) as lowest
  , MAX(grade) as highest
```

```
from <necessary tables>
group by day, subject
```

Now while one needn't choose values for columns, to "pivot" that table by days (each column representing statistics for just one day) change the query to:

```
SELECT subject
  , AVG(IF(day=1, grade,null)) as D1_average
  , MIN(IF(day=1, grade, null)) as D1_lowest
  , MAX(IF(day=1,grade,null)) as D1_highest
  , AVG(IF(day=2, grade,null)) as D2_average
  , MIN(IF(day=2, grade, null)) as D2_lowest
  , MAX(IF(day=2,grade,null)) as D2_highest
  , .... (repeat for rest of the days)
FROM <necessary tables>
GROUP BY day, subject
```

The IF ... NULL test prevents AVG() from counting all other grades for the same subject from different days. The same trick works for MIN and MAX functions.

Back to the top *Browse the book* *Buy the book* *Feedback*

Pivot table without GROUP_CONCAT

Data designs often require flexibility in numbers and names of data points per instance row: instead of saving all the data points belonging to a key value in a single row, you save each data point as a name-value pair in its own row.

Thus given table user_class(user_id INT, class_id CHAR(20), class_value CHAR(20)) with these rows:

```
user_id  class_id   class_value
1        firstname  Rogier
1        lastname   Marat
2        firstname  Jean
2        lastname   Smith
```

and you wish a resultset that links first names to last names for each ID...

```
user_id  firstname  lastname
1        Rogier     Marat
2        Jean       Smith
```

the following query accomplishes the required pivot via an INNER SELF-JOIN:

```
SELECT
  u1.user_ID,
  class_value AS firstname,
  u2.lastname
FROM user_class AS u1
INNER JOIN (
  SELECT
    user_ID,
    class_value AS lastname
  FROM user_class
  WHERE class_id='lastname'
) AS u2
ON u1.user_ID=u2.user_ID AND u1.class_id='firstname'
```

Back to the top *Browse the book* *Buy the book* *Feedback*

All possible recipes with given ingredients

We have tables for recipes (r), ingredients (i), and recipe-ingredient combinations (ri). The `ri` table implements a many-many

relationship between recipes and ingredients, where each row specifies one ingredient for one recipe. A recipe can use any number of ingredients, and an ingredient can occur in any number of recipes:

```
drop table if exists r,i,ri;
create table r(id int);
insert into r values(1),(2);
create table i(id int);
insert into i values(1),(2),(3);
create table ri(rid int,iid int);
insert into ri values (1,1),(1,2),(2,1),(2,4),(3,5);
select * from r;
+------+
| id   |
+------+
|    1 |
|    2 |
+------+
select * from i;
+------+
| id   |
+------+
|    1 |
|    2 |
|    3 |
+------+
select * from ri;
+------+------+
| rid  | iid  |
+------+------+
|    1 |    1 |
|    1 |    2 |
|    2 |    1 |
|    2 |    4 |
|    3 |    5 |
+------+------+
```

Given our ingredients, what recipes can we make? Inspection shows the answer will be recipe #1.

SQL has no universal quantifier, so how do we proceed? 'All A is B' is logically equivalent to the double negative 'there is no A that is not B', so we can reformulate the requirement ...

list the recipes for which we have all ingredients

into terms SQL can handle ...

list the recipes for which there is no ingredient we do not have

A double negative, so a double query. One inner query, one outer. Tackle the inner one first: find the recipes for which we are missing an ingredient.

That's a straight exclusion join, i.e., a left join on ingredient from 'required' to 'available', plus a where clause that restricts the resultset to nulls on the right ('available') side of the join:

```
SELECT DISTINCT rid
FROM ri
LEFT JOIN i ON ri.iid=i.id
WHERE i.id IS NULL;
+------+
| rid  |
+------+
|    2 |
|    3 |
+------+
```

Now the outer query has to find the recipes which are not in this list. That's another exclusion join, this time from recipes to

the above derived table:

```
SELECT r.id
FROM r
LEFT JOIN (
  SELECT DISTINCT rid
  FROM ri
  LEFT JOIN i ON ri.iid=i.id
  WHERE i.id IS NULL
) AS rno ON r.id = rno.rid
WHERE rno.rid IS NULL;
+------+
| id   |
+------+
|    1 |
+------+
```

It's an example of *relational division*, one of Codd's eight basic relational operations. Dividing a divisor table into a dividend table yields a quotient or results table:

dividend � divisor = quotient

As in arithmetic, multiplication reverses it:

*divisor * quotient = dividend*

```
                         +-----------+
     +-----+    +------+    | table AxB |
     |  A  |    |  B   |    +-----+-----+
     +-----+    +------+    |key_a|key_b|
     |key_a|    |key_b |    +-----+-----+
     +-----+    +------+    |  2  |  1  |
     |  2  |    |  1   |    |  2  |  7  |
     |  4  |    |  7   |    |  2  |  3  |
     +-----+    |  3   |    |  4  |  1  |
                +------+    |  4  |  7  |
                            |  4  |  3  |
                            +-----+-----+
```

When we multiply (`CROSS JOIN`) tables A and B to yield AxB, AxB gets a row combining every row of A with every row of B, and all the columns from A and B. When we reverse that operation, dividing AxB by B, we get back A by listing distinct B values associated with A values in AxB.

Back to the top *Browse the book* *Buy the book* *Feedback*

Parties with candidates in all districts

You have candidates (name, district, party), parties (party), and districts (district)tables. Every candidate belongs to a party. Which parties have candidates in all districts?

SQL has no universal quantifier so the solution is to ask for the parties for which there is no district for which there is no candidate. It's a form of relational division. We obtain the desired result (the quotient) by dividing the list of districts (the dividend) by the list of parties (the divisor).

```
SELECT DISTINCT party FROM parties
WHERE NOT EXISTS (
   SELECT * FROM districts
   WHERE NOT EXISTS (
      SELECT * FROM candidates
      WHERE candidates.party=parties.party
      AND candidates.district=districts.district
   )
);
```

One way to get this done without subqueries in the WHERE clause is to JOIN and GROUP BY, then check the party count with a HAVING subquery:

```
SELECT party
FROM candidates
INNER JOIN districts USING (district)
GROUP BY party
HAVING COUNT(party) >= (SELECT COUNT(party) FROM parties);
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Who makes all the parts for a given assembly?

One way to arrive at the answer is by asking: What are the assembly-supplier pairs such that no part of the assembly is not made by the supplier? That's relational division again, formulated for two tables by Stephen Todd. Given `assemblyparts(assembly,part)` and `partsuppliers(part,supplier)` tables, here is a query that Joe Celko credits to Pierre Mullin.

```
SELECT DISTINCT
  AP1.assembly,
  SP1.supplier
FROM AssemblyParts AS AP1, PartSuppliers AS SP1
WHERE NOT EXISTS (
  SELECT *
  FROM AssemblyParts AS AP2
  WHERE AP2.assembly = AP1.assembly
  AND NOT EXISTS (
    SELECT SP2.part
    FROM PartSuppliers AS SP2
    WHERE SP2.part = AP2.part AND SP2.supplier = SP1.supplier
  )
);
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Find blocks of unused numbers

In a table of sequential numbers, some are used and some are not. Find the blocks, if any, of unused IDs:

```
DROP TABLE IF EXISTS tbl;
CREATE TABLE tbl(id INT,used BOOL);
INSERT INTO tbl VALUES(1,1),(2,0),(3,0),(4,1),(5,0),(6,0);
SELECT * FROM tbl;
+------+------+
| id   | used |
+------+------+
|    1 |    1 |
|    2 |    0 |
|    3 |    0 |
|    4 |    1 |
|    5 |    0 |
|    6 |    0 |
+------+------+
```

To list blocks of unused numbers, join the table to itself three times. Call these four representations of the table a, b, c, d, and condition each join on `used=0` on both sides:

- do an exclusion join from a to b on a.id immediately preceding b.id; thus a.id values are unused id values having no immediate unused predecessor,
- left join a to c on a.id < c.id, to remove backward sequences and force a stop at the last value,
- do an exclusion join from c to d on c.id immediately succeeding d.id; thus `MIN(c.id)` is the first unused id in the current

block without an unused successor,

and `GROUP BY a.id`. In the result, `a.id` begins a missing block and `MIN(c.id)` ends a missing block:

```
SELECT a.id AS Start, MIN( c.id ) AS End
FROM tbl AS a
LEFT JOIN tbl AS b ON a.id=b.id + 1 AND a.used=0 AND b.used=0
LEFT JOIN tbl AS c ON a.id<c.id AND a.used=0 AND c.used=0
LEFT JOIN tbl AS d ON c.id=d.id-1 AND c.used=0 AND d.used=0
WHERE b.id IS NULL
  AND c.id IS NOT NULL
  AND d.id IS NULL
GROUP BY a.id;
+-------+------+
| Start | End  |
+-------+------+
|     2 |    3 |
|     5 |    6 |
+-------+------+
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Find missing numbers in a sequence

You have a table `tbl(id int)` with values (1,2,4,18,19,20,21), and you wish to find the first missing number in its sequence of id values:

```
SELECT t1.id+1 AS Missing
FROM tbl AS t1
LEFT JOIN tbl AS t2 ON t1.id+1 = t2.id
WHERE t2.id IS NULL
ORDER BY id LIMIT 1;
+---------+
| Missing |
+---------+
|       3 |
+---------+
```

For all the gaps, including gaps of more than 1 value, you need something a little more baroque...

```
SELECT
  a.id+1 AS 'Missing From',
  MIN(b.id) - 1 AS 'To'
FROM tbl AS a, tbl AS b
WHERE a.id < b.id
GROUP BY a.id
HAVING a.id < MIN(b.id) - 1;
+--------------+------+
| Missing From | To   |
+--------------+------+
|            3 |    3 |
|            5 |   17 |
+--------------+------+
```

We often need such lists, so the query is a natural for a stored procedure that finds missing sequence values in any table:

```
DROP PROCEDURE IF EXISTS MissingInSeq;
DELIMITER |
CREATE PROCEDURE MissingInSeq( db VARCHAR(64), tbl VARCHAR(64), col VARCHAR(64) )
BEGIN
  SET @sql = CONCAT( "SELECT  a.", col,
                     "+1 AS 'Missing From',",
                     "MIN(b.",
                     col,
```

```
                     ") - 1 AS 'To' FROM ",
                     db,
                     ".",
                     tbl,
                     " AS a,",
                     db,
                     ".",
                     tbl,
                     " AS b WHERE a.",
                     col,
                     " < b.",
                     col,
                     " GROUP BY a.",
                     col,
                     " HAVING a.",
                     col,
                     " < MIN(b.",
                     col,
                     ") - 1"
                   );
  -- SELECT @sql;
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
  DROP PREPARE stmt;
END;
|
DELIMITER ;
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Find next highest value of each column value

Sometimes we need all next values of a column:

```
drop table if exists t;
create table t(id int);
insert into t values(2),(4),(6),(8),(10);
select x.aid as id,x.bid as nextvalue
from (
  select a.id as aid,b.id as bid
  from t a
  join t b on a.id<b.id
) x
left join (
  select a.id as aid,b.id as bid
  from t a
  join t b on a.id<b.id
) y on x.aid=y.aid and x.bid>y.bid
where y.bid is null
order by x.aid,x.bid;
+------+-----------+
| id   | nextvalue |
+------+-----------+
|    2 |         4 |
|    4 |         6 |
|    6 |         8 |
|    8 |        10 |
+------+-----------+
```

Modify the algorithm to suit for next lowest &c.

Back to the top *Browse the book* *Buy the book* *Feedback*

Find previous and next values in a sequence

Here is an algorithm by Baron Schwartz (xaprb.com) for retrieving the previous and next column values in a sequence, given a particular column value `thisvalue`. The previous value is the maximum value less than `thisvalue`, and the next value is the minimum value greater than `thisvalue`:

```
SELECT
  IF(col > thisvalue,'next','prev') AS Direction,
  IF(col > thisvalue,MIN(col),MAX(col)) AS 'Prev/Next'
FROM tablename
WHERE col <> thisvalue
GROUP BY SIGN(col - thisvalue);
```

So, to find the previous and next order ids in the Northwind database table orders (nwib.orders), starting from order number 10800:

```
SELECT
  IF(orderid > 10800,'next','prev') AS Direction,
  IF(orderid > 10800,MIN(orderid),MAX(orderid)) AS 'Prev/Next'
FROM nwib.orders
WHERE orderid <> 10800
GROUP BY SIGN(orderid - 10800);
+-----------+-----------+
| Direction | Prev/Next |
+-----------+-----------+
| prev      | 10799     |
| next      | 10801     |
+-----------+-----------+
```

This is a natural for a stored procedure:

```
DROP PROCEDURE IF EXISTS PrevNext;
DELIMITER |
CREATE PROCEDURE PrevNext(
  IN db CHAR(64), IN tbl CHAR(64), IN col CHAR(64), IN seq INT
)
BEGIN
  IF db IS NULL OR db = '' THEN
    SET db = SCHEMA();
  END IF;
  SET @sql = CONCAT( "SELECT ",
                     " IF(", col, " > ", seq,",'next','prev') AS Direction,",
                     " IF(", col, " > ", seq, ",MIN(", col, "),MAX(", col, ")) AS 'Prev/Next'",
                     " FROM ", db, ".", tbl,
                     " WHERE ", col, " <> ", seq,
                     " GROUP BY SIGN(", col, " - ", seq, ")" );
  PREPARE stmt FROM @sql;
  EXECUTE  stmt;
  DEALLOCATE PREPARE stmt;
END;
|
DELIMITER ;
```

Or, it can be embedded in the `FROM` clause of another query, for example ...

```
SELECT o2.OrderID,o2.Value,o.customerid
FROM orders o
JOIN (
  SELECT 'This' AS 'OrderId', 10800 AS 'Value'
  UNION
  SELECT
    IF( orderid > 10800, 'Next', 'Prev') AS Which,
    IF( orderid > 10800, MIN(orderid), MAX(orderid )) AS 'Value'
  FROM orders
  WHERE orderid <> 10800
  GROUP BY SIGN( orderid - 10800 )
) AS o2 ON o.orderid=o2.value
ORDER BY o.orderid;
+---------+-------+------------+
```

```
| OrderID | Value | customerid |
+---------+-------+------------+
| Prev    | 10799 | KOENE      |
| This    | 10800 | SEVES      |
| Next    | 10801 | BOLID      |
+---------+-------+------------+
```

Back to the top	Browse the book	Buy the book	Feedback

Find sequence starts and ends

Find the first and last values of column value sequences. Assuming table `tbl` and numeric column `id`, an exclusion join on the previous sequential value finds the first value of each sequence, and the minimum next value on a left join and an exclusion join on the previous sequential value finds the last of each sequence:

```
SELECT
  a.id AS Start,
  MIN( c.id ) AS End
FROM tbl AS a
LEFT OUTER JOIN tbl AS b ON a.id = b.id + 1
LEFT OUTER JOIN tbl AS c ON a.id < c.id
LEFT OUTER JOIN tbl AS d ON c.id = d.id - 1
WHERE b.id IS NULL
  AND c.id IS NOT NULL
  AND d.id IS NULL
GROUP BY a.id;
```

A variant on the problem: some IDs are used and some are not; find blocks of unused IDs:

```
DROP TABLE IF EXISTS tbl;
CREATE TABLE tbl(id INT,used BOOL);
INSERT INTO tbl VALUES(1,1),(2,0),(3,0),(4,1),(5,0),(6,0);
SELECT a.id AS Start, MIN( c.id ) AS End
FROM tbl AS a
LEFT JOIN tbl AS b ON a.id=b.id + 1 AND a.used=0 AND b.used=0
LEFT JOIN tbl AS c ON a.id<c.id AND a.used=0 AND c.used=0
LEFT JOIN tbl AS d ON c.id=d.id-1 AND c.used=0 AND d.used=0
WHERE b.id IS NULL
  AND c.id IS NOT NULL
  AND d.id IS NULL
GROUP BY a.id;
+-------+------+
| Start | End  |
+-------+------+
|     2 |    3 |
|     5 |    6 |
+-------+------+
```

Back to the top	Browse the book	Buy the book	Feedback

Find specific sequences

You have a table which tracks hits on named web pages...

```
CREATE TABLE hits (
  id INT NOT NULL DEFAULT 0,
  page CHAR(128) DEFAULT '',
  time TIMESTAMP NOT NULL DEFAULT 0,
  PRIMARY KEY(id, time)
)
```

where id is unique to a session. Here is a bit of sample data:

```
INSERT INTO hits VALUES
  (1, 'A', TIMESTAMPADD(SECOND,10,NOW())),
  (1, 'B', TIMESTAMPADD(SECOND,20,NOW())),
  (2, 'A', TIMESTAMPADD(SECOND,40,NOW())),
  (1, 'A', TIMESTAMPADD(SECOND,50,NOW())),
  (1, 'C', TIMESTAMPADD(SECOND,60,NOW())),
  (3, 'A', TIMESTAMPADD(SECOND,110,NOW())),
  (3, 'A', TIMESTAMPADD(SECOND,120,NOW())),
  (3, 'C', TIMESTAMPADD(SECOND,130,NOW())),
  (2, 'C', TIMESTAMPADD(SECOND,90,NOW())),
  (2, 'A', TIMESTAMPADD(SECOND,100,NOW()));
```

You desire a count of the number of sessions where a user moved from one particular page directly to another, for example from 'A' to 'C'.

To find the next hit in a given session, scope on id, order by time, and limit the output to one row. Then simply count the rows meeting the page criteria:

```
SELECT
  COUNT(DISTINCT h1.id) AS 'Moves from A to C'
FROM hits AS h1
WHERE
  h1.page = 'A'
  AND 'C' = (
    SELECT h2.page
    FROM hits AS h2
    WHERE h2.id = h1.id
      AND h2.time > h1.time
    ORDER BY h2.time LIMIT 1
  );
 -------------------
| Moves from A to C |
 -------------------
|                 3 |
 -------------------
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Gaps in a time series

Advanced time series analysis generally requires custom software, but straightforward SQL queries can answer simple time series questions. You have a jobtimes table with columns ID, job, machine, start_time, and stop_time. You wish to know which machines have had gaps between activity periods.Here is a query that shows the start times following breaks in activity for a given machine.

```
SELECT
  id,
  machine AS thismachine,
  start_time AS StartAfterGap
FROM jobtimes
WHERE id > 1 AND NOT EXISTS (
  SELECT stop_time
  FROM jobtimes
  WHERE machine=thismachine
    AND start_time < StartAfterGap
    AND stop_time >= StartAfterGap
)
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Make values of a column sequential

You have a table *tbl* with an integer primary key column *keycol* which is not a key in another table, and which you wish to make perfectly sequential starting with 1.

```
SET @i=0;
UPDATE tbl SET keycol=(@i:=@i+1);
```


Track stepwise project completion

A master table has one row for each project, and the number of sequential steps required to complete each project. A detail table has one row per project per completed step:

```
DROP TABLE IF EXISTS t1 ;
CREATE TABLE t1 (
  id INT, projectname CHAR(2), projectsteps INT
);
INSERT INTO t1 VALUES
(1, 'xx', 3),
(2, 'yy', 3),
(3, 'zz', 5);

DROP TABLE IF EXISTS t2;
CREATE TABLE t2 (
  id INT, projectID INT, xid INT
);
INSERT INTO t2 VALUES
(1, 1, 1),
(2, 1, 2),
(3, 2, 1),
(4, 1, 3),
(5, 3, 2),
(6, 1, 2),
(7, 2, 1),
(8, 2, 1);
```

The requirement is for a query which, for every project, reports 'OK' if there is at least one detail row for every project step, or otherwise reports the number of the last sequential completed step:

Here is one way to build such a query:

1. Join t1 to t2 on projectID.

2. Left Join t2 to itself on projectID and integer succession.

3. Add a `WHERE` condition which turns the left self-join into an exclusion join that finds the first missing sequential xid value.

4. To the `SELECT` list add this item:

```
  IF( a.xid < p.projectstep,a.xid,'OK' ) AS StepState
```

so when the first sequential missing xid is not less than the number of project steps, display 'Ok', otherwise display the xid value before the first missing xid value.

5. Remove dupes with a `GROUP BY` clause.

```
SELECT
  p.projectname,p.projectsteps,a.xid,
  IF(a.xid < p.projectsteps, a.xid, 'OK') AS CompletionState
```

```
FROM t1 p
JOIN t2 a ON p.id = a.projectID
LEFT JOIN t2 AS b ON a.projectID = b.projectID AND a.xid+1 = b.xid
WHERE b.xid IS NULL
GROUP BY p.projectname;
+-------------+--------------+------+-----------------+
| projectname | projectsteps | xid  | CompletionState |
+-------------+--------------+------+-----------------+
| xx          |            3 |    3 | OK              |
| yy          |            3 |    1 | 1               |
| zz          |            5 |    2 | 2               |
+-------------+--------------+------+-----------------+
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Winning Streaks

Given a table of IDs and won-lost results, how do we find the longest winning streak?

```
drop table if exists results;
create table results(id int,result char(1));
insert into results values
(1,'w'),(2,'l'),(3,'l'),(4,'w'),(5,'w'),(6,'w'),(7,'l'),(8,'w'),(9,'w');
select * from results;
+------+--------+
| id   | result |
+------+--------+
|    1 | w      |
|    2 | l      |
|    3 | l      |
|    4 | w      |
|    5 | w      |
|    6 | w      |
|    7 | l      |
|    8 | w      |
|    9 | w      |
+------+--------+
```

We can find streaks of two with a left join on a.id=b.id+1. To count streak lengths, initialise a counter to 0 then increment it for every hit:

```
set @count=0;
select a.id, a.result, b.result, @count := IF(a.result = b.result, @count + 1, 1) as Streak
from results a
left join results b on a.id = b.id + 1
where a.result = 'w';
```

The longest winning streak is the longest such streak found:

```
set @count=0;
select MAX(@count:=IF(a.result = b.result, @count + 1, 1)) as LongestStreak
from results a
left join results b on a.id = b.id + 1
where a.result = 'w';
+---------------+
| LongestStreak |
+---------------+
|             3 |
+---------------+
```

(From a response by Jon Roshko to a question by Ed Ball on the MySQL Newbie Forum)

Back to the top *Browse the book* *Buy the book* *Feedback*

Great circle distance

Find the distance in kilometres between two points on the surface of the earth. This is just the sort of problem stored functions were made for. For a first order approximation, ignore deviations of the earth's surface from the perfectly spherical. Then the distance in radians is given by a number of trigonometric formulas. ACOS and COS behave reasonably:

```
            COS(lat1-lat2)*(1+COS(lon1-lon2)) - COS(lat1+lat2)*(1-COS(lon1-lon2))
rads = ACOS( --------------------------------------------------------------------- )
                                              2
```

We need to convert degrees latitude and longitude to radians, and we need to know the length in km of one radian on the earth's surface, which is 6378.388. The function:

```
set log_bin_trust_function_creators=TRUE;

DROP FUNCTION IF EXISTS GeoDistKM;
DELIMITER |
CREATE FUNCTION GeoDistKM( lat1 FLOAT, lon1 FLOAT, lat2 FLOAT, lon2 FLOAT ) RETURNS float
BEGIN
  DECLARE pi, q1, q2, q3 FLOAT;
  DECLARE rads FLOAT DEFAULT 0;
  SET pi = PI();
  SET lat1 = lat1 * pi / 180;
  SET lon1 = lon1 * pi / 180;
  SET lat2 = lat2 * pi / 180;
  SET lon2 = lon2 * pi / 180;
  SET q1 = COS(lon1-lon2);
  SET q2 = COS(lat1-lat2);
  SET q3 = COS(lat1+lat2);
  SET rads = ACOS( 0.5*((1.0+q1)*q2 - (1.0-q1)*q3) );
  RETURN 6378.388 * rads;
END;
|
DELIMITER ;

-- toronto to montreal (505km):
select geodistkm(43.6667,-79.4167,45.5000,-73.5833);
+----------------------------------------------+
| geodistkm(43.6667,-79.4167,45.5000,-73.5833) |
+----------------------------------------------+
|                            505.38836669921875 |
+----------------------------------------------+
```

(Setting `log_bin_trust_function_creators` is the most convenient way to step round determinacy conventions implemented since 5.0.6.)

Back to the top *Browse the book* *Buy the book* *Feedback*

Moving average

Given a table of dates and daily values, retrieve their moving 5-day average:

```
DROP TABLE IF EXISTS t;
CREATE TABLE t (dt DATE, qty INT);
INSERT INTO t VALUES ('2007-1-1',5),
                     ('2007-1-2',6),
                     ('2007-1-3',7),
                     ('2007-1-4',8),
                     ('2007-1-5',9),
                     ('2007-1-6',10),
                     ('2007-1-7',11),
                     ('2007-1-8',12),
                     ('2007-1-9',13);
SELECT
  t1.dt,
```

```
  ( SELECT SUM(t2.qty) / COUNT(t2.qty)
    FROM t AS t2
    WHERE DATEDIFF(t1.dt, t2.dt) BETWEEN 0 AND 4
  ) AS '5dayMovingAvg'
FROM t AS t1
ORDER BY t1.dt;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Multiple sums across a join

You have a *parties* table that holds info on peoples' names etc, and a *contracts* table in which each row defines one contract, identifying a client as clientpartyID and a contractor as contractorpartyID, each of these a foreign key referencing parties.partyID. You want a list of parties showing how many contracts they have participated in as client, and how many they've participated in as contractor.

```
SELECT
  p.partyID,
  p.name,
  (SELECT COUNT(*) FROM contractor_client c1 WHERE c1.clientpartyID = p.partyID )
  AS ClientDeals,
  (SELECT COUNT(*) FROM contractor_client c2 WHERE c2.contractorpartyID = p.partyID)
  AS ContractorDeals
FROM parties p
ORDER BY partyID;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Percentiles

Assuming a table tbl(id INT, val FLOAT), retrieve a percentile ranking of all vals.

```
SELECT
  a.id ,
  ROUND( 100.0 * ( SELECT COUNT(*) FROM tbl AS b WHERE b.val <= a.val ) / total.cnt, 1 )
  AS percentile
FROM child a
CROSS JOIN (
  SELECT COUNT(*) AS cnt
  FROM tbl
) AS total
ORDER BY percentile DESC;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Random row selection

When your web page loads it is to provide a randomly selected Murphy's Law from your murphy table (id int, text law)):

```
SELECT law
FROM murphy
ORDER BY RAND()
LIMIT 1;
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Running Sum

Calculate and display a per-row cumulative sum of column values

```
SET @total=0;
SELECT value, @total:=@total+value AS RunningSum
FROM tbl
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*

Sum across categories

You often need to sum across several categories to total customer purchase amounts, salesperson sales amounts, political party election spending, etc.

For this example assume three tables: candidates, parties and ridings. You want to get the total amount spent in all ridings by every party in one output row. Here is the schema:

```
CREATE TABLE candidates (
  id int(11) NOT NULL default '0',
  `name` char(10) ,
  riding char(12) ,
  party char(12) ,
  amt_spent decimal(10,0) NOT NULL default '0',
  PRIMARY KEY  (id)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

INSERT INTO candidates
  VALUES (1,'Anne Jones','Essex','Liberal','5000'),
         (2,'Mary Smith','Malton','Liberal','7000'),
         (3,'Sara Black','Riverdale','Liberal','15000'),
         (4,'Paul Jones','Essex','Socialist','3000'),
         (5,'Ed While','Essex','Conservative','10000'),
         (6,'Jim kelly','Malton','Liberal','9000'),
         (7,'Fred Price','Riverdale','Socialist','4000');

CREATE TABLE ridings (
  riding char(10) NOT NULL default '',
  PRIMARY KEY  (riding)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

INSERT INTO ridings VALUES ('Essex'),('Malton'),('Riverdale');

CREATE TABLE parties (
  party char(12) NOT NULL default '',
  PRIMARY KEY  (party)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

INSERT INTO parties VALUES ('Conservative'),('Liberal'),('Socialist');
```

And here is the query that does it:

```
SELECT
  SUM(amt_spent) AS Total,
  (SUM(amt_spent)-SUM(CASE WHEN data.party='Conservative' THEN NULL ELSE amt_spent END)) AS Cons,
  (SUM(amt_spent)-SUM(CASE WHEN data.party='Liberal' THEN NULL ELSE amt_spent END)) AS Lib,
  (SUM(amt_spent)-SUM(CASE WHEN data.party='Socialist' THEN NULL ELSE amt_spent END)) AS Soc
FROM
  (SELECT * FROM candidates
  INNER JOIN parties ON candidates.party=parties.party
  INNER JOIN ridings ON candidates.riding=ridings.riding) AS data

 ------- ------- ------- ------
| Total | Cons  | Lib   | Soc  |
 ------- ------- ------- ------
| 53000 | 10000 | 36000 | 7000 |
```

```
 ------- ------- ------- ------
```

Back to the top	Browse the book	Buy the book	Feedback

Top ten

We often want to know the top 1, 2, 10 or whatever values from a query. This is dead simple in MySQL. However many JOINs and WHEREs the query has, simply ORDER BY the column(s) whose highest values are sought, and LIMIT the resultset:

```
  SELECT (somecolumn), (othercolumns) ...
  FROM (some tables) ...
  ORDER BY somecolumn DESC
  LIMIT 10;
```

Back to the top	Browse the book	Buy the book	Feedback

A cursor if necessary, but not necessarily a cursor

You have `photos (id INT, photo BLOB, tally INT)` and `votes(id INT, userID INT, photoID INT)` tables. You wish to update `photos.tally` values from counts per photo in the `votes` table. You can use a cursor to walk the photos table, updating the tally as you go:

```
DROP TABLE IF EXISTS photos;
CREATE TABLE photos (id INT, photo BLOB, tally INT);
INSERT INTO photos VALUES(1,'',0),(2,'',0);
DROP TABLE IF EXISTS VOTES;
CREATE TABLE VOTES( userID INT, photoID INT);
INSERT INTO votes VALUES (1,1),(2,1),(2,2);

DROP PROCEDURE IF EXISTS updatetallies;
DELIMITER //
CREATE PROCEDURE updatetallies()
BEGIN
  DECLARE done INT DEFAULT 0;
  DECLARE pid INT;
  DECLARE cur1 CURSOR FOR SELECT id FROM photos;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
  OPEN cur1;
  FETCH cur1 INTO pid;
  WHILE done = 0 DO
    UPDATE photos
      SET tally = (SELECT COUNT(*) FROM votes WHERE photoid = pid )
      WHERE id = pid;
    FETCH cur1 INTO pid;
  END WHILE;
  CLOSE cur1;
  SELECT id,tally FROM photos;
END //
DELIMITER ;
CALL updatetallies();
+------+-------+
| id   | tally |
+------+-------+
|    1 |     2 |
|    2 |     1 |
+------+-------+
```

but a simple join does exactly the same job at much less cost:

```
UPDATE photos
SET tally = (
```

```
  SELECT COUNT(*) FROM votes WHERE votes.photoid = photos.id
);
```

Before you burden your app with a cursor, see if you can simplify the processing to a straightforward join.

Back to the top *Browse the book* *Buy the book* *Feedback*

Emulate sp_exec

Sometimes it is desirable to call multiple stored procedures in one command. In SQL Server this can be done with `sp_exec`. In MySQL we can easily write such an sproc that calls as many sprocs as we please, for example...

```
USE sys;
DROP PROCEDURE IF EXISTS sp_exec;
DELIMITER |
CREATE PROCEDURE sp_exec( p1 CHAR(64), p2 CHAR(64) )
BEGIN
  -- permit doublequotes to delimit data
  SET @sqlmode=(SELECT @@sql_mode);
  SET @@sql_mode='';
  SET @sql = CONCAT( "CALL ", p1 );
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
  DROP PREPARE stmt;
  SET @sql = CONCAT( "CALL ", p2 );
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
  DROP PREPARE stmt;
  SET @@sql_mode=@sqlmode;
END;
|
DELIMITER ;
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Variable-length argument for query IN() clause

To have an sproc accept a variable-length parameter list for an `IN(...)` clause in a query, code the sproc to `PREPARE` the query statement:

```
DROP PROCEDURE IF EXISTS passInParam;
DELIMITER |
CREATE PROCEDURE passInParam( IN qry VARCHAR(100), IN param VARCHAR(1000) )
BEGIN
  SET @qry = CONCAT( qry, param, ')' );
  PREPARE stmt FROM @qry;
  EXECUTE stmt;
  DROP PREPARE stmt;
END;
|
DELIMITER ;
```

For this example, the query string should be of the form:

```
SELECT ... FROM ... WHERE ... IN (
```

but so long as it has those elements, it can be as complex as you like. When you call the sproc:
1. Quote each argument with a *pair* of single quotes,
2. Separate these quoted arguments with commas,
3. Surround the whole `param` string with another set of single quotes:

```
CALL passInParam( 'SELECT * FROM tbl WHERE colval IN (', ('''abc'',''def'',''ghi''' );
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Count delimited substrings

Here is a function to count substrings delimited by a constant delimiting string:

```
DROP FUNCTION IF EXISTS strcount;
SET GLOBAL log_bin_trust_function_creators=1;
DELIMITER |
CREATE FUNCTION strCount( pDelim VARCHAR(32), pStr TEXT) RETURNS int(11)
BEGIN
  DECLARE n INT DEFAULT 0;
  DECLARE pos INT DEFAULT 1;
  DECLARE strRemain TEXT;
  SET strRemain = pStr;
  SET pos = LOCATE( pDelim, strRemain );
  WHILE pos != 0 DO
    SET n = n + 1;
    SET pos = LOCATE( pDelim, strRemain );
    SET strRemain = SUBSTRING( strRemain, pos+1 );
  END WHILE;
RETURN n;
END |
DELIMITER ;

-- example call:
SET @str  = "The quick brown fox jumped over the lazy dog";
SET @delim = " ";
SELECT strCount(@delim,@str);
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Count substrings

To count instances of a search string in a target string ...

- in the target string, replace the search string with a single character,
- subtract the length of the modified target string from the length of the original target string,
- divide that by the length of the search string:

```
SET @str  = "The quick brown fox jumped over the lazy dog";
SET @find = "the";
SELECT ROUND(((LENGTH(@str) - LENGTH(REPLACE(LCASE(@str), @find, '')))/LENGTH(@find)),0)
AS COUNT;
+-------+
| COUNT |
+-------+
|     2 |
+-------+
```

Note that REPLACE() does a case-sensitive search; to get a case-insensitive result you must coerce target and search strings to one case.

To remove decimals from the result:

```
SELECT CAST((LENGTH(@str) - LENGTH(REPLACE(LCASE(@str)), @find, '')))/LENGTH(@find) AS SIGNED) AS COUNT;
```

Back to the top *Browse the book* *Buy the book* *Feedback*

Levenshtein distance

The Levenshtein distance between two strings is the minimum number of operations needed to transform one string into the other, where an operation may be insertion, deletion or substitution of one character. Jason Rust published this MySQL algorithm for it at http://www.codejanitor.com/wp/.

```
CREATE FUNCTION levenshtein( s1 VARCHAR(255), s2 VARCHAR(255) )
  RETURNS INT
  DETERMINISTIC
  BEGIN
    DECLARE s1_len, s2_len, i, j, c, c_temp, cost INT;
    DECLARE s1_char CHAR;
    -- max strlen=255
    DECLARE cv0, cv1 VARBINARY(256);
    SET s1_len = CHAR_LENGTH(s1), s2_len = CHAR_LENGTH(s2), cv1 = 0x00, j = 1, i = 1, c = 0;
    IF s1 = s2 THEN
      RETURN 0;
    ELSEIF s1_len = 0 THEN
      RETURN s2_len;
    ELSEIF s2_len = 0 THEN
      RETURN s1_len;
    ELSE
      WHILE j <= s2_len DO
        SET cv1 = CONCAT(cv1, UNHEX(HEX(j))), j = j + 1;
      END WHILE;
      WHILE i <= s1_len DO
        SET s1_char = SUBSTRING(s1, i, 1), c = i, cv0 = UNHEX(HEX(i)), j = 1;
        WHILE j <= s2_len DO
          SET c = c + 1;
          IF s1_char = SUBSTRING(s2, j, 1) THEN
            SET cost = 0; ELSE SET cost = 1;
          END IF;
          SET c_temp = CONV(HEX(SUBSTRING(cv1, j, 1)), 16, 10) + cost;
          IF c > c_temp THEN SET c = c_temp; END IF;
            SET c_temp = CONV(HEX(SUBSTRING(cv1, j+1, 1)), 16, 10) + 1;
            IF c > c_temp THEN
              SET c = c_temp;
            END IF;
            SET cv0 = CONCAT(cv0, UNHEX(HEX(c))), j = j + 1;
        END WHILE;
        SET cv1 = cv0, i = i + 1;
      END WHILE;
    END IF;
    RETURN c;
  END;
```

Helper function:

```
CREATE FUNCTION levenshtein_ratio( s1 VARCHAR(255), s2 VARCHAR(255) )
  RETURNS INT
  DETERMINISTIC
  BEGIN
    DECLARE s1_len, s2_len, max_len INT;
    SET s1_len = LENGTH(s1), s2_len = LENGTH(s2);
    IF s1_len > s2_len THEN
      SET max_len = s1_len;
    ELSE
      SET max_len = s2_len;
    END IF;
    RETURN ROUND((1 - LEVENSHTEIN(s1, s2) / max_len) * 100);
  END;
```

Back to the top	Browse the book	Buy the book	Feedback

Proper case

The basic idea is...

- lower-case the string
- upper-case the first character if it is a-z, and any other a-z character that follows a punctuation character

Here is the function. To make it work with strings long than 128 characters, change its input and return declarations accordingly:

```
DROP FUNCTION IF EXISTS proper;
SET GLOBAL  log_bin_trust_function_creators=TRUE;
DELIMITER |
CREATE FUNCTION proper( str VARCHAR(128) )
RETURNS VARCHAR(128)
BEGIN
  DECLARE c CHAR(1);
  DECLARE s VARCHAR(128);
  DECLARE i INT DEFAULT 1;
  DECLARE bool INT DEFAULT 1;
  DECLARE punct CHAR(17) DEFAULT ' ()[]{},.-_!@;:?/';
  SET s = LCASE( str );
  WHILE i < LENGTH( str ) DO
    BEGIN
      SET c = SUBSTRING( s, i, 1 );
      IF LOCATE( c, punct ) > 0 THEN
        SET bool = 1;
      ELSEIF bool=1 THEN
        BEGIN
          IF c >= 'a' AND c <= 'z' THEN
            BEGIN
              SET s = CONCAT(LEFT(s,i-1),UCASE(c),SUBSTRING(s,i+1));
              SET bool = 0;
            END;
          ELSEIF c >= '0' AND c <= '9' THEN
            SET bool = 0;
          END IF;
        END;
      END IF;
      SET i = i+1;
    END;
  END WHILE;
  RETURN s;
END;
|
DELIMITER ;
```

Back to the top	Browse the book	Buy the book	Feedback

Strip HTML tags

Ported from a T-SQL function by Robert Davis:

```
SET GLOBAL log_bin_trust_function_creators=1;
DROP FUNCTION IF EXISTS fnStripTags;
DELIMITER |
CREATE FUNCTION fnStripTags( Dirty varchar(4000) )
RETURNS varchar(4000)
DETERMINISTIC
BEGIN
  DECLARE iStart, iEnd, iLength int;
  WHILE Locate( '<', Dirty ) > 0 And Locate( '>', Dirty, Locate( '<', Dirty )) > 0 DO
```

```
    BEGIN
      SET iStart = Locate( '<', Dirty ), iEnd = Locate( '>', Dirty, Locate('<', Dirty ));
      SET iLength = ( iEnd - iStart) + 1;
      IF iLength > 0 THEN
        BEGIN
          SET Dirty = Insert( Dirty, iStart, iLength, '');
        END;
      END IF;
    END;
  END WHILE;
  RETURN Dirty;
END;
|
DELIMITER ;
```

SELECT fnStripTags('this is **a test**, nothing more') AS Test;

```
+-----------------------------+
| Test                        |
+-----------------------------+
| this is a test, nothing more |
+-----------------------------+
```

Back to the top	*Browse the book*	*Buy the book*	*Feedback*