SQL is Not Computational

- SQL is NOT a computational language
- All computable problems can be solved in a non-procedural language like SQL
  - Primitive recursive functions are equivalent to push down stack automata --- big fat hairy deal!
- SQL works best with tables only
- All relations are shown as columns and values
  - don’t make assumptions about the physical storage
  - it will change from product to product, release to release
Auxiliary Tables -1

- It is not an entity or a relationship in the data model
- It is a “look up” table for a function used by the database
- If you showed it on an E-R diagram, it would connect to almost every table in the Schema
- You use them to get around the weak computational power of SQL
- You use them to “fake” some procedural code
- They take advantage of parallelism in database machines
- They guarantee all host programs use the same calculations
We will discuss a few examples
- And we will not go into much detail
- There are a lot more ways to use an auxiliary table than just these

Think about using auxiliary tables when the calculations in the problem gets to be horribly complicated

Think about using auxiliary tables when you need to move the calculations from one host language to another
The Gregorian Calendar (properly known as the Common Era calendar) is irregular.

Fiscal year calendars are worse!
- There are over 150 different standard fiscal years in the GAAP
- Fiscal weeks can vary from 3 to 12 days

Lunar calendars (Moslem, Jewish and Chinese) are hard to map to a Solar calendar.

Christians have Easter.

Various civil holidays move around by decree.

Emergency days can be declared.
Calendars -2

- What is January 31 + 1 month?
  - Feb 28 (end of next month)
  - Feb 29 (leapyear, end of next month)
  - Mar 01 (30 days - bond trading rule)
  - Mar 02 (31 days - why not? This is the outer limit)

- Adding months is very irregular
  - Why Standard SQL avoids it and starts with days are the largest temporal unit in the math

- In SQL-92, you can only add days without errors
  - No two vendors agree
  - No two industries agreed either
Example: stock trades must close in three (3) business days after execution
  - Business days do not include weekends (Saturday, Sunday)
  - Business days do not include holidays, known in advance
  - Business days do not include emergency closing

Stock market holidays can change during the year (wars, presidential decrees, halt in trading in a stock, etc.)
Solution: a Calendar table for data

CREATE TABLE Calendar
(cal_date DATE NOT NULL PRIMARY KEY,
fiscal_year INTEGER NOT NULL,
fiscal_month INTEGER NOT NULL,
next_month INTEGER NOT NULL,
holiday CHAR(1) NOT NULL
CHECK holiday IN ('y', 'n'),
dayofweek INTEGER NOT NULL
CHECK(dayofweek BETWEEN 0 AND 6), ...);
Use Joins to get to Calendar table:

… Mytable.mydate = Calendar.cal_date

Use ranges in tables for events:

CREATE TABLE ShoppingSeasons
(season CHAR(25) NOT NULL PRIMARY KEY,
 start_date DATE NOT NULL,
 end_date DATE , -- null means still open ...
);

INSERT INTO ShoppingSeasons
• The range trick uses NULL as a marker for “eternity” or “until now”
• The BETWEEN predicate has to be aware of the NULL:

…WHERE mydate
    BETWEEN start_date
    AND COALESCE(end_date, mydate)

Or …WHERE mydate
    BETWEEN start_date
    AND COALESCE(end_date, CURRENT_TIMESTAMP)
The range tables and Calendar table can be used together. Example: The rates for a rental vary by time periods.

```sql
CREATE TABLE WidgetRental
(
  start_date DATE NOT NULL UNIQUE,
  end_date DATE NOT NULL UNIQUE,
  rental_price DECIMAL(12,4) NOT NULL,
  CHECK(start_date <= end_date))
```

Customers get a widget over a period of time that might not match a unique WidgetRental row.
The trick is to use the Calendar table to get the days in the intersection of the range of rental rates and the customer’s usage

```
SELECT :start_rental, :end_rental, SUM(price)
FROM Calendar AS C1,
    WidgetRentals AS W1
WHERE C1.cal_date BETWEEN :start_rental
    AND :end_rental
    AND C1.cal_date BETWEEN W1.start_date
    AND W1.end_date
```
• Sequence table replaces a loop to put multiple columns in one column

• Given this table

  CREATE TABLE Foobar
  (keycol CHAR(10) NOT NULL PRIMARY KEY,
   x1 INTEGER NOT NULL,
   x2 INTEGER NOT NULL,
   x3 INTEGER NOT NULL,
   x4 INTEGER NOT NULL);

• Produce a table or View like this:

  CREATE TABLE Flatbar
  (keycol CHAR(10) NOT NULL,
   x INTEGER NOT NULL);
CREATE VIEW Flatbar (keycol, x) AS
SELECT keycol,
    CASE S1.seq
        WHEN 1 THEN F1.x1
        WHEN 2 THEN F1.x2
        WHEN 3 THEN F1.x3
        WHEN 4 THEN F1.x4
        ELSE NULL END
FROM Foobar AS F1
CROSS JOIN
    (SELECT seq
     FROM Sequence AS S1
     WHERE S1.seq BETWEEN 1 AND 4);

In SQL-92 you can also use VALUES (1), (2), (3), (4)
The net present value (NPV) calculation tells you the present value of a series of cashflows (CF) made to you over time (t) from an initial investment (t0) for a given rate of return (r).

It is a polynomial equation:

\[
NPV = \sum_{t=0}^{T} \frac{CF_t}{(1+r)^t} = CF_0 + \frac{CF_1}{(1+r)^1} + \frac{CF_2}{(1+r)^2} + \ldots + \frac{CF_T}{(1+r)^T}
\]
CREATE TABLE CashFlows
    (project_id CHAR(15) NOT NULL,
     time_period INTEGER NOT NULL,
     CHECK (time_period >= 0),
     amount DECIMAL(12,4) NOT NULL,
     PRIMARY KEY (project_id, time_period));

CREATE TABLE Rates
    (rate DECIMAL(6,4) NOT NULL PRIMARY KEY);
Use a VIEW to review all the projects at once.

CREATE VIEW NPV_by_Rate(project_id, rate, npv) AS
  SELECT CF.project_id, R1.rate, SUM(amount)
    / ( POWER (1.00 + R1.rate), time_period))
FROM CashFlows AS CF, Rates AS R1
GROUP BY R1.rate, CF.project_id;
... or set the amount to 1 and store the multiplier for the (rate, time_period) in a table:

```
INSERT INTO NPV_Multipliers (time_period, rate, npv_multiplier)
SELECT seq, rate,
    SUM(1.00/( (1.00 + R1.rate), seq))
FROM Sequence, Rates AS R1
WHERE seq <= :upper_limit
GROUP BY seq, R1.rate;
```
The IRR is the interest rate where the NPV of the project equals zero. It tells you what you're making on this particular project.

Assuming we use the VIEW, the IRR is now the single query:

SELECT 'Acme', rate AS irr, npv
FROM NPV_by_Rate
WHERE ABS(npv) = (SELECT MIN(ABS(npv))
FROM NPV_by_Rate)
AND project_id = 'Acme';
• Warning: The previous query does not work in all cases!

• IRR can have many zeros because NPV is a polynomial of an unknown degree.

• Moral to the story: be careful about using auxiliary tables for inverses of functions
• Use spreadsheets for calendar function creation - they have good date and fiscal functions.

• Use Fortran or a math package for scientific functions

• Test the auxiliary table against the formula or other source
  - SQL might not doing rounding and truncation as expected.

• Use NULLs for answers out of range
  - Or where the function is undefined

• You can use interpolation techniques with the auxiliary table
  - Look up the techniques in engineering and finance books written before the pocket calculator was
Hints - 2

- Trick for Sequence table:
  
  ```sql
  SELECT 100* D1.i + 10*D2.i + D3.i 
  FROM Digits AS D1, Digits AS D2, Digits AS D3
  WHERE (100* D1.i + 10*D2.i + D3.i ) > 0;
  ```

- Digits is a one column auxiliary table of the numbers zero through nine

- Technique can be expanded to any size integer then limited to a range in the WHERE clause.