Experimental Finance

IEOR Department

Mike Lipkin, Alexander Stanton
Housekeeping

- Lab/home connectivity?
- Problem Set 2 due next week – groups allowed!
- Send in your groups with names and UNI IDs due next week
Outline

- Tables
- SQL Query Syntax
- Joins
- Views
- Indices
Some Basic Database Design Background

Four Basic Building Blocks:

• **Tables:**
  Information is split into multiple tables in order to minimize redundant information storage and to maximize speed of data return

• **Indices and Keys:**
  Organizing indices in an efficient manner so that queries return data quickly

• **Views:**
  Aggregation of table information into one virtual table

• **Stored procedures and Functions:**
  Similar to functions in C – can pass data by reference or by value. Have the added capability of returning a query result.
Table Structures

Tables are normalized and split up for the following reasons:

• Reduce storage by eliminating redundancy
• Querying smaller datasets is always faster
• Data management and storage for real-time systems: Data of the same type (i.e. same columns) can be split into multiple tables and stored on different drive clusters with their own controllers for simultaneous access.
## Tables and Normalization

<table>
<thead>
<tr>
<th>Ticker</th>
<th>date</th>
<th>closePrice</th>
<th>volume</th>
<th>strike</th>
<th>expiration</th>
<th>CallBestBid</th>
<th>CallBestOffer</th>
<th>Call Volume</th>
<th>PutBestBid</th>
<th>PutBestOffer</th>
<th>PutVolume</th>
<th>Delta etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSFT</td>
<td>11/12/2005</td>
<td>24.83</td>
<td>$2032620</td>
<td>20</td>
<td>5/21/2005</td>
<td>5</td>
<td>5.1</td>
<td>0</td>
<td>0</td>
<td>0.05</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MSFT</td>
<td>11/12/2005</td>
<td>24.83</td>
<td>$2032620</td>
<td>25</td>
<td>5/21/2005</td>
<td>0.2</td>
<td>0.25</td>
<td>5</td>
<td>0.2</td>
<td>0.3</td>
<td>3208</td>
<td></td>
</tr>
<tr>
<td>MSFT</td>
<td>11/12/2005</td>
<td>24.83</td>
<td>$2032620</td>
<td>30</td>
<td>5/21/2005</td>
<td>0</td>
<td>0.05</td>
<td>0</td>
<td>5</td>
<td>5.1</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>MSFT</td>
<td>11/12/2005</td>
<td>24.83</td>
<td>$2032620</td>
<td>30</td>
<td>5/18/2005</td>
<td>5.1</td>
<td>0</td>
<td>0</td>
<td>0.05</td>
<td>0</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>MSFT</td>
<td>11/12/2005</td>
<td>24.83</td>
<td>$2032620</td>
<td>20</td>
<td>5/18/2005</td>
<td>0</td>
<td>0.05</td>
<td>0</td>
<td>5</td>
<td>5.1</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>MSFT</td>
<td>11/12/2005</td>
<td>24.83</td>
<td>$2032620</td>
<td>20</td>
<td>7/16/2005</td>
<td>5.1</td>
<td>5.2</td>
<td>17</td>
<td>0</td>
<td>0.4</td>
<td>1082</td>
<td></td>
</tr>
<tr>
<td>MSFT</td>
<td>11/12/2005</td>
<td>24.83</td>
<td>$2032620</td>
<td>25</td>
<td>7/16/2005</td>
<td>0.8</td>
<td>0.85</td>
<td>1427</td>
<td>0.7</td>
<td>0.75</td>
<td>2891</td>
<td></td>
</tr>
</tbody>
</table>

**Normalization**

### securityID - Ticker

<table>
<thead>
<tr>
<th>securityID</th>
<th>Date</th>
<th>closePrice</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11/12/2005</td>
<td>24.83</td>
<td>$2032620</td>
</tr>
<tr>
<td>1</td>
<td>11/12/2005</td>
<td>24.83</td>
<td>$2032620</td>
</tr>
<tr>
<td>2</td>
<td>11/12/2005</td>
<td>14.2</td>
<td>$2233100</td>
</tr>
<tr>
<td>2</td>
<td>11/13/2005</td>
<td>14.1</td>
<td>$2113433</td>
</tr>
</tbody>
</table>

### securityID - Date

<table>
<thead>
<tr>
<th>securityID</th>
<th>date</th>
<th>strike</th>
<th>expiration</th>
<th>callPut</th>
<th>bestBid</th>
<th>bestOffer</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11/12/2005</td>
<td>20</td>
<td>5/21/2005</td>
<td>call</td>
<td>5</td>
<td>5.1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>11/12/2005</td>
<td>20</td>
<td>5/21/2005</td>
<td>put</td>
<td>0</td>
<td>0.05</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>11/12/2005</td>
<td>25</td>
<td>5/21/2005</td>
<td>call</td>
<td>0.2</td>
<td>0.25</td>
<td>7021</td>
</tr>
<tr>
<td>1</td>
<td>11/12/2005</td>
<td>25</td>
<td>5/21/2005</td>
<td>put</td>
<td>0.2</td>
<td>0.3</td>
<td>3208</td>
</tr>
<tr>
<td>2</td>
<td>11/12/2005</td>
<td>10</td>
<td>5/21/2005</td>
<td>call</td>
<td>5</td>
<td>5.1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>11/12/2005</td>
<td>10</td>
<td>5/21/2005</td>
<td>put</td>
<td>0</td>
<td>0.05</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>11/12/2005</td>
<td>15</td>
<td>5/21/2005</td>
<td>call</td>
<td>0.2</td>
<td>0.25</td>
<td>7021</td>
</tr>
<tr>
<td>2</td>
<td>11/12/2005</td>
<td>15</td>
<td>5/21/2005</td>
<td>put</td>
<td>0.2</td>
<td>0.3</td>
<td>3208</td>
</tr>
</tbody>
</table>
Database Keys

- Tables must have primary keys so that each row is uniquely identifiable.
- A primary key may consist of one or more columns

<table>
<thead>
<tr>
<th>SECURITY</th>
<th>SECURITY_PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>securityID</td>
<td>Date</td>
</tr>
<tr>
<td>1</td>
<td>11/12/2005</td>
</tr>
<tr>
<td>1</td>
<td>11/13/2005</td>
</tr>
<tr>
<td>2</td>
<td>11/12/2005</td>
</tr>
</tbody>
</table>

Key: SecurityID

Key: SecurityID, Date

<table>
<thead>
<tr>
<th>OPTION_PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>securityID</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Key: SecurityID, Date, Expiration, Strike, callPut
• Keys can be based on columns that contain information, or not.
  
  e.g. Mudd-303
  Social security number (area number, group number, serial)
  ISBN number for books (language identifier)
  Credit cards (3=amex, 5=mastercard)
  incremental ID or random GUID

• In finance, the most frequent key is the ticker symbol. Is this a good key choice?
Database Keys - continued

• Keys can be based on columns that contain information, or not.
  
  e.g. Mudd-303  
  Social security number (area number, group number, serial)  
  ISBN number for books (language identifier)  
  Credit cards (3=amex, 5=mastercard)  
  incremental ID or random GUID

• In finance, the most frequent key is the ticker symbol. Is this a good key choice?
  
  No. Any data that has a remote possibility of changing is **never** a good candidate for a primary key

• Historical data would require time-consuming updates when a corporate name change occurs

• Existing queries and stored procedures might break if keys are referenced in any way (say as part of temporary or result tables)
Database Keys

• Additional reasons not to use data elements as primary keys:
  – Possibility of nulls
    (options may need to be created in a database before they have an official exchange ID or value)
  – Changes in data structures and naming conventions
  – Varying length data and size limitations
Database Keys

- Additional reasons not to use data elements as primary keys:
  - Possibility of nulls
    (options may need to be created in a database before they have an official exchange ID or value)
  - Changes in data structures and naming conventions
  - Varying length data and size limitations

A good financial example of a really bad key choice:

**IBMBL.X 60 Feb 2007 Call**  
**Last:** 37.50  
**OI:** 1451

*After IBMBL.X expires, it is re-used.*

This causes major headaches during analysis, especially when the information to reconstruct the option is not available
Database Keys

• This was finally fixed in 2009:

IBM160323C00172500

• The key is unique for the lifetime of the option, and more importantly before the options is born and after it has expired.
  – Notice that the new format limits you to strikes of no more than $99,999 dollars.
  – Maybe this is short-sighted in an inflation-driven economy.
  – They added weeklies, what about hourlies?

The good news for you is that you still get to play with the old broken format when you are dealing with historical data
Surrogate Keys

- **Surrogate keys** are used to manage relationships between data where defining data is ambiguous or likely to change.

<table>
<thead>
<tr>
<th>SECURITY</th>
<th>SECURITY_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>securityID, Ticker</td>
<td>securityID, Date, Ticker</td>
</tr>
<tr>
<td>103049 C</td>
<td>103049 1/1/1996 TRV</td>
</tr>
<tr>
<td>111019 TRV</td>
<td>103049 10/8/1998 CCI</td>
</tr>
<tr>
<td>103545 CCI</td>
<td>103049 12/8/1998 C</td>
</tr>
</tbody>
</table>

**securityID stays constant with stock and option prices for the same company**

<table>
<thead>
<tr>
<th>SECURITY_PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>securityID, Date, closePrice, volume</td>
</tr>
<tr>
<td>103049, 11/12/2005, 24.83, 82032620</td>
</tr>
<tr>
<td>103049, 11/13/2005, 24.92, 79032620</td>
</tr>
<tr>
<td>103049, 11/14/2005, 14.2, 2233100</td>
</tr>
<tr>
<td>103049, 11/15/2005, 14.1, 2113433</td>
</tr>
</tbody>
</table>
What really happened?

- Citibank used to be **CCI** before it merged with **TRV**
- **C** merged with Daimler to become **DCX**
- When Citibank merged with Traveler’s Group it stayed **CCI** for two months before changing to **C**, about one month after Chrysler merged with Daimler.
- **C** was **Chrysler** – it is now **Citibank**
- **TRV** is now **Thousand Trails Inc.**
- **CCI** is now **Crown Castle Intl.**
• Transact SQL is the database language – pronounced ‘Sequel’
• Most databases use SQL to manipulate data
  – Microsoft SQL
  – Oracle
  – DB2
  – Postgres
  – MySQL

• There are varying formats for SQL, but differences are minimal:
  e.g. return the top 10 rows:
  – SELECT ticker FROM security limit 10; (MySQL)
  – SELECT TOP 10 ticker FROM security; (MS-SQL)
  – SELECT ticker FROM security WHERE ROWNUM < 10; (Oracle)
**Basic Format**

SELECT <columns>
FROM <table>
WHERE <condition>
ORDER BY <columns>

**Example:**

```
SELECT securityID FROM security WHERE ticker= 'MSFT'
```

**Result:**

```
1051334
```

The keyword DISTINCT can be used to remove duplicates:

```
SELECT DISTINCT securityID FROM security_price
WHERE (date range)
```
WHERE Clause

- WHERE clause can use several operators:
  - >
  - <
  - =
  - AND, OR, LIKE, BETWEEN, IS NULL, IN, NOT

  e.g. SELECT securityID, date, closePrice
       FROM security_price
       WHERE close_price > 200 AND
            date BETWEEN '2005-01-01' AND '2005-01-03'
       ORDER BY securityID, date

  Returns all closePrices for stocks that have a price greater than $200, ordered first by ticker, then by date
GROUP BY Clause

- GROUP BY clauses create groupings when using aggregate functions such as:

  ```
  AVG(), SUM(), COUNT(), MAX(), MIN()
  ```

- Example SQL query:

  ```sql
  SELECT ticker, (weekNumber), MAX(closePrice) as maxClosePrice
  FROM security_price
  WHERE date BETWEEN '2005-01-01' AND '2005-01-03'
  GROUP BY ticker, (weekNumber)
  ORDER BY ticker, (weekNumber)
  ```

  Instructs SQL to return the maximum close price for each week for each ticker (i.e. “grouped by ticker and week”)

Experimental Finance Mike Lipkin, Alexander Stanton Page 19
GROUP BY Clause Using a Count

```
SELECT securityID, date, count(*)
FROM option_price_view (where MSFT etc.)
GROUP BY securityID, date
ORDER BY securityID, date
```

Desire the number of different strike dollar values across all series per day
GROUP BY Clause Using a Count

```
SELECT securityID, date, count(*)
FROM option_price_view (where MSFT etc.)
GROUP BY securityID, date
ORDER BY securityID, date
```

Desire the number of different strike dollar values across all series per day

What might be wrong here?
Almost Fixed Count

```sql
SELECT securityID, date, count(*)
FROM option_price
WHERE callPut = "call" AND (MSFT etc.)
GROUP BY securityID, date
ORDER BY securityID, date
```

This SQL query selects the securityID, date, and counts the number of strikes across all expirations per day per ticker.
Almost Fixed Count

```
SELECT securityID, date, count(*)
FROM option_price
WHERE callPut = "call" AND (MSFT etc.)
GROUP BY securityID, date
ORDER BY securityID, date
```

Returns the number of strikes across all expirations per day per ticker.

still not quite right…
### Fixed Count

```
SELECT securityID, date, count(DISTINCT strike)
FROM option_price
WHERE callPut = 'call' AND (MSFT etc.)
GROUP BY securityID, date
ORDER BY securityID, date
```

This SQL statement selects the number of distinct strikes for each expiration per day per ticker. It filters the data to include only calls and then groups the results by security ID and date, ordering them by security ID and date.

The resulting table shows the count of distinct strikes for each security ID and date:

<table>
<thead>
<tr>
<th>securityID</th>
<th>date</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11/12/2005</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>11/12/2005</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>11/12/2005</td>
<td>14</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>12/13/2005</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>12/13/2005</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>12/13/2005</td>
<td>14</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Returns the number of distinct strikes across all expirations per day per security ID.
IN Clause

- IN Clause:

```sql
SELECT ticker, date, closePrice
FROM security_price
WHERE ticker IN ( 'MSFT', 'MT', 'INTL', 'KO' ) AND
date BETWEEN '2005-01-01' AND '2005-01-03'
ORDER BY date, ticker
```

Returns all closePrices for the four stocks in the date range, ordered first by date, then ticker.
Sub Queries:

```
SELECT ticker, date, closePrice
FROM security_price
WHERE ticker IN (SELECT ticker FROM myTickerTable)
AND date BETWEEN '2005-01-01' AND '2005-01-03'
ORDER BY date, ticker
```

- Very useful speed up strategy for repeated querying:

  Instead of joining two large tables to find the intersection, and running this repeatedly, create a small table of the intersection and then use either a JOIN or a sub-query using IN for further queries.
SQL JOINS
JOINS

• Joins allow combining of data from disjoint tables based on keys
• Allow conditional selection of data from one table that matches (or doesn’t match) conditions in another
• Join conditions can be based on primary keys, unique and non-unique keys, as well as generic data in the table
• Joins can be complicated – they are highly susceptible to design errors, and can lead to exceedingly long query times and incorrect data
### Stock Price

<table>
<thead>
<tr>
<th>securityID</th>
<th>Date</th>
<th>closePrice</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11/12/2005</td>
<td>24.83</td>
<td>82032620</td>
</tr>
<tr>
<td>1</td>
<td>11/13/2005</td>
<td>24.92</td>
<td>79032620</td>
</tr>
<tr>
<td>2</td>
<td>11/12/2005</td>
<td>14.2</td>
<td>2233100</td>
</tr>
<tr>
<td>2</td>
<td>11/13/2005</td>
<td>14.1</td>
<td>2113433</td>
</tr>
</tbody>
</table>

### Option Price

<table>
<thead>
<tr>
<th>securityID</th>
<th>date</th>
<th>strike</th>
<th>expiration</th>
<th>callPut</th>
<th>bestBid</th>
<th>bestOffer</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11/12/2005</td>
<td>20</td>
<td>5/21/2005 call</td>
<td>5</td>
<td>5.1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11/12/2005</td>
<td>20</td>
<td>5/21/2005 put</td>
<td>0</td>
<td>0.05</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11/12/2005</td>
<td>15</td>
<td>5/21/2005 call</td>
<td>0.2</td>
<td>0.25</td>
<td>7021</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11/12/2005</td>
<td>15</td>
<td>5/21/2005 put</td>
<td>0.2</td>
<td>0.3</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11/13/2005</td>
<td>20</td>
<td>5/21/2005 call</td>
<td>5</td>
<td>5.1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11/13/2005</td>
<td>20</td>
<td>5/21/2005 put</td>
<td>0</td>
<td>0.05</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11/13/2005</td>
<td>15</td>
<td>5/21/2005 call</td>
<td>0.2</td>
<td>0.25</td>
<td>7021</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11/13/2005</td>
<td>15</td>
<td>5/21/2005 put</td>
<td>0.2</td>
<td>0.3</td>
<td>200</td>
<td></td>
</tr>
</tbody>
</table>

Stock & Option Data
Basic Format

SELECT <columns>
FROM <table1>
INNER JOIN <table2> ON <condition>
WHERE <condition>
ORDER BY <columns>

• Returns a Cartesian product of table1 and table2, N x M rows
• <Condition> can have multiple components
• **Beware!!** Joins where the condition is not precisely matched can cause large record sets of garbage data
• We will see later that the usage of NULL is very powerful
INNER JOINs

• JOIN tables based on one or more conditions
• Result is an intersection of both tables where JOIN condition is satisfied

```
SELECT ticker, date, closePrice
FROM security s
INNER JOIN security_price sp ON s.securityID=sp.securityID
```
INNER JOINS

SELECT securityID, sp.date, sp.closePrice, o.strike, o.closePrice
FROM security_price sp
INNER JOIN option_price o ON o.securityID=sp.securityID

What is wrong here?
Cartesian Products

- JOIN conditions result in $N_{\text{match}} \times M_{\text{match}}$ Rows

It is critical to get them right:

```sql
SELECT ticker, sp.date, sp.closePrice, o.strike, o.closePrice
FROM security_price sp
INNER JOIN option_price o
ON o.securityID=sp.securityID AND o.date=sp.date
```
• Note that this is (semantically) not quite the same as:

```
SELECT ticker, sp.date, sp.closePrice, o.strike, o.closePrice
FROM security_price sp INNER JOIN option_price o
ON o.securityID=sp.securityID
WHERE o.date=s.date
```

WHY?
JOIN Conditions - continued

• Note that this is (semantically) not quite the same as:

```
SELECT ticker, sp.date, sp.closePrice, o.strike, o.closePrice
FROM security_price sp INNER JOIN option_price o
ON o.securityID=sp.securityID
WHERE o.date=s.date
```

WHY?

• The general processing order is:
  – **FIRST**: Joins are evaluated
  – **SECOND**: results restricted by WHERE clause

• The query above creates much more data, most of which is false, and then trims out the false results. **Highly inefficient**.

  (NOTE: some SQL engines are good at parsing and optimizing, but this is fundamental knowledge you should have. The DB might get it wrong so the more you know the better)
OUTER JOINS

- OUTER JOINS select all data from one table, and only data from the second table where the JOIN condition is matched:

  ```sql
  SELECT ticker, date, closePrice
  FROM security s
  LEFT OUTER JOIN security_price sp ON
  s.securityID=sp.securityID
  ```

- Joins are evaluated first, then the results are restricted according to the WHERE clause

- Can use LEFT OUTER JOIN and RIGHT OUTER JOIN to specify which table will be returned in full

- Columns values from the second table that don’t match are filled with a NULL
OUTER JOINS

```sql
SELECT ticker, date, closePrice
FROM security s
LEFT OUTER JOIN security_price sp ON s.securityID=sp.securityID
```
• OUTER JOINS are very powerful in conjunction with the NULL value. Example:

SELECT TOP 1 ticker
FROM security s
LEFT OUTER JOIN security_price sp ON
s.securityID=sp.securityID
WHERE sp.date=NULL

Result:

<table>
<thead>
<tr>
<th>ticker</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAH</td>
</tr>
</tbody>
</table>
Unfortunate Syntax

- Table1 LEFT JOIN table 2 ON table1.ID = table2.ID
- Table1 LEFT OUTER JOIN Table2 ON table1.ID = table2.ID
- Table1, Table2 WHERE table1.ID = table2.ID
  Bad Syntax – confuses WHERE clause
- Table1 OUTER JOIN Table2 ON table1.ID = table2.ID
  Bad Syntax – only time the order of the tables matters

Use Standard ANSI-92 Notation:

Table1 LEFT OUTER JOIN Table2 ON table1.ID = table2.ID
FULL JOINS and CROSS JOINS

- FULL JOINS evaluate both tables and return all rows from both tables

- CROSS JOINS return the cartesian product of two tables where there is no common key (i.e., no “ON” clause)
UNIONs join tables with identical tables structures:

- **SELECT * FROM OPTION_PRICE_2005_01**
- **UNION ALL**
- **SELECT * FROM OPTION_PRICE_2005_02**

<table>
<thead>
<tr>
<th>securityID</th>
<th>Date</th>
<th>closePrice</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11/12/2005</td>
<td>24.83</td>
<td>82032620</td>
</tr>
<tr>
<td>1</td>
<td>11/13/2005</td>
<td>24.92</td>
<td>79032620</td>
</tr>
<tr>
<td>2</td>
<td>11/12/2005</td>
<td>14.2</td>
<td>2233100</td>
</tr>
<tr>
<td>2</td>
<td>11/13/2005</td>
<td>14.1</td>
<td>2113433</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>securityID</th>
<th>Date</th>
<th>closePrice</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11/12/2006</td>
<td>23.2</td>
<td>46476467</td>
</tr>
<tr>
<td>1</td>
<td>11/13/2006</td>
<td>23.4</td>
<td>47643763</td>
</tr>
<tr>
<td>2</td>
<td>11/12/2006</td>
<td>9</td>
<td>2342343</td>
</tr>
<tr>
<td>2</td>
<td>11/13/2006</td>
<td>8.8</td>
<td>2232332</td>
</tr>
</tbody>
</table>
Self JOINS

- SQL processing is entirely row based
- There is no knowledge of previous or future row values
- Use JOINS to compare different rows in the same table

```
SELECT s.ticker, sp1.date as date1, sp1.closePrice as price1,
sp2.date as date2, sp2.closePrice as price2
FROM security s
INNER JOIN security_price sp1 on sp1.ID = s.ID
LEFT JOIN security_price sp2 on sp2.ID = sp1.ID AND
sp2.date = sp1.date - 1
WHERE ticker = 'MSFT' AND
sp1.date between '2004-01-07' and '2004-01-12'
```

Why the LEFT JOIN?
### Self JOINS

- **Why would this query be useful?**

<table>
<thead>
<tr>
<th>ticker</th>
<th>date</th>
<th>closePrice</th>
<th>date</th>
<th>closePrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSFT</td>
<td>1/8/2004</td>
<td>28.16</td>
<td>1/7/2004</td>
<td>28.21</td>
</tr>
<tr>
<td>MSFT</td>
<td>1/12/2004</td>
<td>27.57</td>
<td><strong>null</strong></td>
<td><strong>null</strong></td>
</tr>
<tr>
<td>MSFT</td>
<td>1/13/2004</td>
<td>27.43</td>
<td>1/12/2004</td>
<td>27.57</td>
</tr>
</tbody>
</table>
Self JOINS

Why would this query be useful?

Compute the log return:

SELECT s.ticker, sp1.date, (sp1.close-sp2.close)/sp2.close AS logReturn
Comparing Stock Price Differences

```sql
select
    sp1.date,
    s1.ticker,
    sp1.closePrice,
    s2.ticker,
    sp2.closePrice,
    (sp1.closePrice-sp2.closePrice) as diff
from ivydb..security_price sp1
inner join ivydb..security_price sp2 on sp1.date=sp2.date
inner join ivydb..security s1 on s1.securityID=sp1.securityID and s1.ticker='MOT'
inner join ivydb..security s2 on s2.securityID=sp2.securityID and s2.ticker='NOK'
where sp1.date>'2004-01-01'
```
Example: Price Differences

MOT vs. NOK price difference 1/2004-5/2005
VIEWS

• Views are based on SELECT statements and offer a way to:
  
  – Encapsulate often repeated joins
    (e.g. security -> security_price -> option_price_view)
  
  – Reduce statement size for readability and to reduce errors
  
  – Encapsulate complex statements and force re-use of standard queries (once you solve the weekend problem, encapsulate it in a view and forget about the details)

• Views are identical to tables in terms of querying. Each base table can and will be evaluated based on its own indices
Large Queries

```sql
SELECT sp.date, sp.closePrice, dbo.formatstrike(strike) as strike, expiration, callput,
datediff(day,sp.date,expiration) as daysToExpiration,
CAST(datediff(day,sp.date,expiration) as float)/360 as interestDays, dbo.mbbo(bestBid,bestOffer) as mbbo,
PPop = (case WHEN dbo.formatstrike(strike)<sp.closePrice THEN dbo.mbbo(bestBid,bestOffer) ELSE
dbo.mbbo(bestBid,bestOffer)-(dbo.formatstrike(strike)-sp.closeprice) END),
CPOP = (case WHEN dbo.formatstrike(strike)>sp.closePrice THEN dbo.mbbo(bestBid,bestOffer) ELSE
dbo.mbbo(bestBid,bestOffer)-(sp.closeprice-dbo.formatstrike(strike)) END)
FROM security s
INNER JOIN security_price sp ON s.securityID=sp.securityID
INNER JOIN option_price_view o ON o.securityID=s.securityID AND sp.date=o.date
INNER JOIN option_price_view o2 ON o2.securityID=s2.securityID AND sp.date=o2.date-1
WHERE
sp.date>='2001-01-01 00:00:00' AND
sp.date<='2005-01-01 00:00:00' AND
Ticker IN (SELECT ticker FROM portfolioTickers pt INNER JOIN option_price_view o3 ON pt.securityID=03.securityID WHERE putCall='P') AND
abs(sp.closePrice-dbo.formatstrike(strike))<=7.5
ORDER BY sp.date, strike, expiration, callput
```
• Views are generally non-updateable, however updatable views can be created as well

• IVY has a large UNION query for option prices

  OPTION_PRICE_VIEW

• Views can be indexed (allows precompilation)

CREATE VIEW vLogReturns AS
SELECT {} FROM {} INNER JOIN {} INNER JOIN {} INNER JOIN {} INNER JOIN {} INNER JOIN {} INNER JOIN {} WHERE {} Etc.

SELECT ticker, date, logReturn FROM vLogReturns
ORDER BY ticker, date
SQL is full of Gotchas

What is wrong here (this is an Oracle example):

```
SELECT t.*
FROM trades t
WHERE ROWNUM < 6
ORDER BY date DESC
```
What is wrong here (this is an Oracle example):

```sql
SELECT t.*
FROM trades t
WHERE ROWNUM < 6
ORDER BY date DESC
```

Corrected:

```sql
SELECT * FROM
  (SELECT t.*
   FROM trades t
   ORDER BY date DESC)
WHERE ROWNUM < 6
```
Indices
Indices

- Indices are added to tables to quickly access results when asking for a subset.
- Primary keys are inherently indexed.

```
<table>
<thead>
<tr>
<th>securityID</th>
<th>Date</th>
<th>closePrice</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11/12/2005</td>
<td>24.83</td>
<td>82032620</td>
</tr>
<tr>
<td>1</td>
<td>11/13/2005</td>
<td>24.92</td>
<td>79032620</td>
</tr>
<tr>
<td>2</td>
<td>11/12/2005</td>
<td>14.2</td>
<td>2233100</td>
</tr>
</tbody>
</table>
```

E.g. “SELECT all option data for Microsoft on 2005-05-12” will be an indexed query (securityID is the primary key, and date is indexed by IVY).

Let’s say, however, that we want to return all days where the volume was above 3,000,000 for any stock.

The database would not be able to use the same index since neither Date nor SecurityID is part of the query restriction.
Table scans

• Every row in the table is traversed and compared to the requirement “volume>3,000,000”, then the list is compiled and returned.

• With 250,000,000 rows this will not even happen overnight

Resolution

• Define a index on the database based on the “volume” column

• This creates an ordered list that can be searched using a B-Tree quickly and efficiently.

It is hardly ever necessary to use a table scan for historical data – be careful!!
The database can only use one index per table operation

Consider:

“SELECT all options FROM optionsTable
WHERE ticker='MSFT' AND optionVolume>3,000,000”

The database has two options:

– Use the primary key index (securityID, date) to select only MSFT data, then do a scan on the subset

– Use the volume index to get all option data that has volume>3,000,000, then scan to return only MSFT data

Which is better?
The Query Optimizer – Query Plans

• **Answer:** *It depends on selectivity*
  
  – Limiting to MSFT means that we can reduce the dataset by a factor of 5,000 using the primary key
  – If MSFT strike volume is only above 10,000 very infrequently, and there are only a few other stocks that have a similarly high volume, the volume index will be more efficient.

• The built-in SQL Query Optimizer decides which index is better to use, based on hit ratios and other statistics

• This is a statistical process and the optimizer may choose a different query plan for the same query depending on “circumstance”

• A small change in the query (e.g. volume > 3,500,000) might cause it to change the query plan and use a different index, yielding completely different execution times
Query Plans are not Voodoo

- Look at the query plan before the query executes:

  set showplan_all [on|off]

- Displays the query plan, i.e. what steps the engine will use to execute the query. The query is not actually executed.

- Check the query will use the indices you expect it to

- If a slightly modified query executes in a vastly different time frame, compare query plans.

- It is possible to give the query optimizer “Hints” (e.g. “TABLE HINT [...] INDEX”) however this is for experts only – the query optimizer is generally better than you.


Do not let queries run out of control!
Data Validation and Cleaning

- Always validate your results
- Spot check by selecting example rows and perform the entire calculations manually
- IVY generally has “good” data, but care must be taken to ensure analysis is not based on inconsistent data, mostly as a result of query selectivity.
- All data sources have problems – don’t forget this
- Check for:
  - abnormal values
  - negative numbers
  - rounding errors
  - GROUP BY queries that aggregate differently sized sets (e.g. 4-day weeks)
Data Validation and Cleaning - continued

• Additional checks:
  – date range issues
  – Missing data
  – Calculated numbers (implied vols, greeks)
  – Assumptions (there is no “50-delta” option)
Useful Functions

- IF THEN ELSE Statements
- CAST (arithmetic) and CONVERT (strings)
- Date Functions:
  - DATEADD
  - DATEDIFF
  - DATEPART
  - MONTH
  - YEAR
- FLOOR, CEILING
Date Gotchas

• Nobody agrees on holidays, especially internationally
  – 4th July
  – Bank Holidays
  – Easter in Gregorian/Julian calendars, Orthodox vs. Catholic
  – Weekends in Islamic countries
  – Germany’s hard-working four-day work-week
  – On July 4th, 2008 they announced that the world's atomic clocks would be adding another second at the end of the year

• The Maritime Freight Industry publishes local holiday schedules for every port in the world

• We aren’t so lucky, and even if we know the dates, interpreting pricing rules for instruments and comparing different markets is exceedingly difficult

• Dates are just an example: Currencies, Lots, Price Ticks all provide ample fun for your weekend coding
ODBC Setup

- Control Panel -> Administrative Tools -> Data Sources (ODBC)
- Add a File DSN, choosing the SQL server driver
- Connect to the Database in the same way as the Query Analyzer
Excel & Data Sources

- In Excel, choose Data->Import External Data->New Database Query
- Choose the DSN you created, and write a query to select results
Next Time

- Creating Tables
- Creating Temporary Tables
- Creating Indices
- Stored Procedures and Functions

Also… you might be interested in the following bed-time reading:

“A Market-Induced Mechanism For Stock Pinning”