Experimental Finance

IEOR

Mike Lipkin, Alexander Stanton
Outline

- Problem set notes
- SQL statements continued
- Functions
- Stored Procedures
- Input/Output parameters
- Variable Declarations
- Cursors
• Don't use STD_OPTION_PRICE for historical research unless there is a good reason – this is an interpolated, standardized table and whether it is accurate or not is in the eye of the beholder
Pinning

- What is the purpose of using the following calculations for the pinning questions:
  - Open Interest
  - Average daily volume
  - Implied Volatility

- What statistic are we tracking and what might affect them?
- What biases are we trying to remove?
- What are we normalizing and why?

These are all questions you should be asking yourselves, and us. This is a laboratory course! We are dealing with real-world numbers that are sometimes inaccurate, missing or plain wrong.
SQL Joins

Round vs. Truncate

SELECT CAST(25.75 AS DECIMAL(7,0)) AS roundedValue
SELECT CONVERT(25.75 AS DECIMAL(7,0)) AS roundedValue
### Round vs. Truncate

SELECT CAST(25.75 AS DECIMAL(7,0)) AS roundedValue

SELECT CONVERT(25.75 AS DECIMAL(7,0)) AS roundedValue

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>numeric</td>
<td>numeric</td>
<td>Round</td>
</tr>
<tr>
<td>numeric</td>
<td>int</td>
<td>Truncate</td>
</tr>
<tr>
<td>numeric</td>
<td>money</td>
<td>Round</td>
</tr>
<tr>
<td>money</td>
<td>int</td>
<td>Round</td>
</tr>
<tr>
<td>money</td>
<td>numeric</td>
<td>Round</td>
</tr>
<tr>
<td>float</td>
<td>int</td>
<td>Truncate</td>
</tr>
<tr>
<td>float</td>
<td>numeric</td>
<td>Round</td>
</tr>
<tr>
<td>float</td>
<td>datetime</td>
<td>Round</td>
</tr>
<tr>
<td>datetime</td>
<td>int</td>
<td>Round</td>
</tr>
</tbody>
</table>

CONVERT is SQL Server specific, CAST is ANSI.
JOIN Performance

SELECT ticker, closePrice, dbo.mbbo(bestBid, bestAsk)
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 o.date = sp.date
SELECT ticker, closePrice, dbo.mbbo(bestBid, bestAsk)
FROM security s
INNER JOIN Security_Price sp
    ON s.SecurityID = sp.SecurityID
INNER JOIN Option_Price_View o
    ON o.SecurityID = sp.SecurityID
    AND o.date = sp.date

• Always JOIN the least number of tables together
• Optimize for index use
SELECT O.SecurityID, count(Strike) 
FROM Option_Price_View O 
INNER JOIN Security_Price S 
ON S.SecurityID = O.SecurityID 
AND DATEPART(dw, O.Expiration) - 1 = DATEPART(dw, s.Date) 
WHERE O.SecurityID in (103125, 107525) 
AND O.Date BETWEEN '1996-01-01' AND '2005-01-01' 
AND S.ClosePrice BETWEEN dbo.formatstrike(strike)-0.15 
AND dbo.formatstrike(strike)+0.15 
AND ImpliedVolatility > 0 
GROUP BY O.SecurityID
JOIN Performance

```sql
SELECT O.SecurityID, count(Strike)
FROM Option_Price_View O
INNER JOIN Security_Price S
ON S.SecurityID = O.SecurityID
AND DATEPART(dw,O.Expiration) - 1 = DATEPART(dw, s.Date)
WHERE O.SecurityID in (103125, 107525)
AND O.Date BETWEEN '1996-01-01' AND '2005-01-01'
AND S.ClosePrice BETWEEN dbo.formatstrike(strike) - 0.15
AND dbo.formatstrike(strike) + 0.15
AND ImpliedVolatility > 0
AND DATEPART(dw,O.Expiration) - 1 = DATEPART(dw, s.Date)
GROUP BY O.SecurityID
```

- Join conditions should be based on indexed fields whenever possible
- Calculations that use both JOIN tables should be part of the WHERE clause and not the JOIN condition (in general)
JOIN condition performance – good practices

- Restriction on fields in JOIN conditions should be indexed
- Calculations/Selects are allowed when they do not depend on both JOIN tables:

```sql
SELECT O.SecurityID, count(Strike)
FROM Option_Price_View O
INNER JOIN Security_Price S
ON S.SecurityID = O.SecurityID
AND S.securityID IN (20234,23456)
AND S.volume < 50000
WHERE O.Date between '1996-01-01' and '2005-01-01' 
    AND S.ClosePrice BETWEEN dbo.formatstrike(strike)-0.15
    AND dbo.formatstrike(strike)+0.15
    AND ImpliedVolatility > 0
GROUP BY O.SecurityID
```
JOIN Condition Performance

- Sub queries should be “joinTable/joinRow independent”

```sql
SELECT O.SecurityID, count(Strike)
FROM Option_Price_View O
INNER JOIN Security_Price S
ON S.SecurityID = O.SecurityID
AND S.securityID IN (20234,23456)
AND S.date IN (SELECT date FROM myDateRange)
AND S.date IN (SELECT date FROM option_price_view o1
WHERE o1.volume>5000)
WHERE O.Date between '1996-01-01' and '2005-01-01'
    AND S.ClosePrice between dbo.formatstrike(strike)-0.15
    AND dbo.formatstrike(strike)+0.15
    AND ImpliedVolatility > 0
GROUP BY O.SecurityID
```
Surrogate Keys

- Highly useful for ordering results. E.g.

(assume myTable exists with an identity column defined as myID=IDENTITY(int,1,1))

```
SELECT strike, date, expiration [..]
INTO myTable
FROM [..] ORDER BY strike [ASC/DESC dep. on In/Out of the money]
```

```
SELECT strike, vol, price etc.
FROM myTable T1 INNER JOIN ON T1.myID=T2.myID+1
```

This creates a clean series, i.e. we have removed the dependency on fixed increment dollar values, dates, binning intervals, etc. etc.
CAREFUL when you implicitly create a table:

SELECT ID=IDENTITY(INT,1,1), strike,date,expiration [..] INTO myTable
FROM [..] ORDER BY strike [ASC/DESC dep. on In/Out of the money]

UNPREDICTABLE RESULTS – the ORDER BY clause is not guaranteed to be followed during the insert due to parallel processing and physical page locks.

NEVER USE THIS!!!

Unless you really don’t care about the order
And even then it is a bad habit
“You may notice in certain cases that the IDENTITY value that SQL Server generates is not in the same order as the ORDER BY column in the query.

In other cases, you may notice that the IDENTITY value that SQL Server generates is in exactly the same order as the ORDER BY column in the query. However, this is coincidental and is not the guaranteed order you will receive every time the query is run.”

-- MSDN (Microsoft’s Developer Network)
• Create a table of expirations

create table expirations(
    id INT NOT NULL IDENTITY(1,1),
    securityID INT NOT NULL,
    expiration DATETIME NOT NULL);

INSERT INTO expirations (securityID,expiration)
SELECT distinct securityID, expiration
FROM option_price_view opv
ORDER BY securityID, expiration
• Add previous and next expiration columns:

```sql
alter table expirations
add prevExpiration datetime not null

alter table expirations
add nextExpiration datetime not null
```
• Add previous and next expiration columns:

```sql
alter table expirations
add prevExpiration datetime not null
```

```sql
alter table expirations
add nextExpiration datetime not null
```

**NOT NULL DOESN'T WORK**
Useful Table Example

Alter table expirations
add prevExpiration datetime NULL

alter table expirations
add nextExpiration datetime NOT NULL
DEFAULT '1900-01-01 00:00:00'

<table>
<thead>
<tr>
<th></th>
<th>securityID</th>
<th>expiration</th>
<th>prevExpiration</th>
<th>nextExpiration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1996-01-20 00:00:00</td>
<td>NULL</td>
<td>1996-01-20 00:00:00</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1996-02-17 00:00:00</td>
<td>NULL</td>
<td>1996-01-20 00:00:00</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1996-03-16 00:00:00</td>
<td>NULL</td>
<td>1996-01-20 00:00:00</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1996-05-18 00:00:00</td>
<td>NULL</td>
<td>1996-01-20 00:00:00</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1996-08-17 00:00:00</td>
<td>NULL</td>
<td>1996-01-20 00:00:00</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>1996-01-20 00:00:00</td>
<td>NULL</td>
<td>1996-01-20 00:00:00</td>
</tr>
</tbody>
</table>
update expirations set prevExpiration = (select expiration from expirations e where e.id = expirations.id - 1)

update expirations set nextExpiration = (select expiration from expirations e where e.id = expirations.id + 1)
update expirations set prevExpiration = (select expiration from expirations e where e.id = expirations.id-1)

update expirations set nextExpiration =(select expiration from expirations e where e.id = expirations.id+1)

Msg 515, Level 16, State 2, Line 1
Cannot insert the value NULL into column 'nextExpiration', table 'IVY.dbo.expirations'; column does not allow nulls. UPDATE fails. The statement has been terminated.
useful table example

update expirations set nextExpiration = (select expiration from expirations e where e.id = expirations.id+1)

works:

update expirations set nextExpiration = (select expiration from expirations e where e.id = expirations.id+1) where id < (select MAX(id) from expirations)
Useful Table Example

• Some cleanup:

```
delete from expirations where prevExpiration is null or nextExpiration is null
```

```
select e1.securityID, e1.id, e1.expiration as frontMonth, e2.id, e2.expiration as roughlySixMonth from expirations e1 inner join expirations e2 on e1.securityID=e2.securityID and e2.id=e1.id+4 order by securityID,e1.id
```
Adjustment Factors

• Cumulative Total Adjustment Factor
  – Includes stock splits but not dividend adjustments

• Cumulative Total Return Factor
  – Includes both stock splits and dividend adjustments
  – Same procedure as for Adjustment Factor

• For options, the Adjustment Factor incorporates any distributions that affect the number of contracts held.
  (see the DISTRIBUTION table and DistributionType field)
Adjustment Factors

• To compute a normalized price, use closePrice*adjustmentFactor

  – Cumulative adjustment factor since the time the stock was listed

    MSFT, 2005-06-30, AdjF = 16
    split 1 time, 16:1 Or
    split 2 times, 8:1, 2:1 Or
    split 3 times, 4:1, 2:1, 2:1 Or
    split 4 times, 2:1, 2:1, 2:1, 2:1 Or

    (cannot distinguish without looking at the history)

    SELECT DISTINCT sp.adjustmentFactor
    FROM security_price sp
    WHERE securityID=107525

    1.0
    2.0
    4.0
    8.0
    16.0
Adjustment Factors

SELECT DISTINCT min(date) as date, 
    sp.adjustmentFactor as adjF 
FROM security_price sp 
WHERE securityID=107525 
GROUP BY sp.adjustmentFactor 
ORDER BY date

**Price adjusted:**

SELECT DISTINCT date, closePrice*sp.adjustmentFactor as adjPrice 
FROM security_price sp 
WHERE securityID=107525 
    AND date< '1997-05-01' 
ORDER BY date

<table>
<thead>
<tr>
<th>Date</th>
<th>AdjF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996-01-02</td>
<td>1.0</td>
</tr>
<tr>
<td>1996-12-09</td>
<td>2.0</td>
</tr>
<tr>
<td>1998-02-23</td>
<td>4.0</td>
</tr>
<tr>
<td>2003-02-18</td>
<td>8.0</td>
</tr>
</tbody>
</table>
Adjustment Factors

Price adjusted to “real” price wrt max(date):

SELECT DISTINCT date,  
‘adjPrice’ = closePrice*sp.adjustmentFactor/(SELECT sp1.adjustmentFactor  
FROM security_price sp1 WHERE  
sp.securityID=sp1.securityID AND  
sp.date= ‘1997-05-01’ )  
FROM security_price sp  
WHERE  
securityID=107525  
AND date< ‘1997-05-01’  
ORDER BY date

Use of variables is important here!
Use of variables

DECLARE @myDate AS DATETIME
DECLARE @mySID AS INT

SELECT @myDate=‘1997-05-01’
SELECT @mySID=107525

SELECT DISTINCT date,
    ‘adjPrice’ =
        closePrice*sp.adjustmentFactor/(SELECT sp1.adjustmentFactor
            FROM security_price sp1
            WHERE sp.securityID=@mySID AND
                sp.date=@myDate)
FROM security_price sp
WHERE securityID=@mySID AND
    date<@myDate
ORDER BY date

Notice the join is no longer dependent on the outer query – big benefit!
Functions and Stored Procedures
What Are They For?

• Run quick checks/fixes to data directly on the server
• Running processes on the server that are too complex to encapsulate in a single query
• Network bandwidth conservation
• Query auto-parameterization (similar to an API)
• Encapsulation of business rules and policies
• Consistent, safe parameter modification
What Can They Do?

• Perform a calculation based on input variables
• Return multiple values
• Return datasets
• Create/update/modify tables
• Offer procedural statements:
  – While loops
  – If statements
  – Cursors
Functions

- Return a single value
- Used in queries/stored procedures
- Called using similar syntax to C:
  
  \[
  \text{myFunction}(\text{param1})
  \]

- Used inline in SQL statements, e.g.

  \[
  \text{SELECT dbo.mbbo(bestBid, bestAsk) FROM } \ldots
  \]
CREATE FUNCTION isWeekDay() AS
RETURNS bit
BEGIN
    DECLARE @var bit
    IF DATEPART(dw, GETDATE()) < 5 SELECT @var=1
    ELSE SELECT @var=0
    RETURN @var
END
GO

SELECT ivyuser.isWeekDay()
Stored Procedures

- SPs manipulate data efficiently on the server
- very useful for calculations where
  - The result is a small dataset
  - The datasets required to do the calculation are large, or too large, to perform on the client

There are trade-offs:

Too much CPU usage on the server will slow it down

centralized servers/mainframes vs. distributed computing is continuously debated. There is no right answer – the choice is highly application-specific
CREATE PROCEDURE isWeekDay
AS
BEGIN
    DECLARE @var bit
    IF DATEPART(dw,GETDATE()) < 5 SELECT @var=1
    ELSE SELECT @var=0
    RETURN @var
END
GO

-----------------------------
declare @v bit
exec @v=isWeekDay
SELECT @v
Calling Stored Procedures

EXEC ivyuser.getYieldCurve ‘MSFT’, ‘2002-02-02’

• Stored procedures cannot be called inline in a SQL query

• Unlike Functions, stored procedures are called by separating parameters with a comma - NOT parentheses. (Why you ask? No reason...)

• Stored procedures can return multiple values, and are generally used to compute several steps of complexity in a program
The FUNCTION with GETDATE() will NOT WORK.

- Functions are compiled more strictly for speed.
- Certain variable functions are not allowed within functions – in this case getDate will fail so the only way to do this would be with a stored procedure.
- Functions are best used when computing a value based on inputs, or performing a single result query within the database based on inputs.

For example: `SELECT expiration, isWeekDay(expiration) from option_price_view`
• DROP FUNCTION mbbo
• DROP PROCEDURE isWeekDay
• Shortcut: Use “ALTER PROCEDURE isWeekDay” instead of DROP/CREATE
• ALTER also preserves permissions
Input Parameters

- Used to return data based on input variables:

```
CREATE PROCEDURE mbbo
@bid DECIMAL(10,2),
@ask DECIMAL(10,2)
AS
BEGIN
    RETURN (@bid+@ask)/2.0
END
GO

EXEC @result = ivyuser.mbbo 0.1, 0.20
```
CREATE PROCEDURE isWeekDay
@date DATETIME
AS
BEGIN
    IF DATEPART(dw, @date) < 5 RETURN 1
    ELSE RETURN 0
END
GO

Other Examples of Functions/Stored Procedures:

daysToExpiration(date)
daysFromExpiration(date)
percentBadVols(ticker,expiration,strike,C/P,date1,date2)
abnormalOptions(ticker,date1,date2) returns TRUE/FALSE
or better yet a list of issues (abnormal settlements,dupes)
Output Parameters

- Keyword OUTPUT
- Used when multiple values are returned:

```sql
CREATE PROCEDURE rounded
    @myNumber DECIMAL(10,2),
    @rounded INTEGER OUTPUT,
    @floor INTEGER OUTPUT,
    @ceiling INTEGER OUTPUT
AS
BEGIN
    SELECT @rounded = CAST(ROUND(@myNumber,2) as INTEGER)
    SELECT @floor = CEILING(@myNumber)
    SELECT @ceiling = FLOOR(@myNumber)
END
GO

EXEC ivyuser.rounded 12.32, @round OUTPUT, @up OUTPUT, @down OUTPUT

answer: @round=12  @up=13  @down=12
```
CREATE PROCEDURE getPrice
@ticker VARCHAR(10),
@myDate DATETIME
AS
BEGIN
    DECLARE @retPrice DECIMAL(10,2)
    SELECT @retPrice = price
    FROM security_price
    INNER JOIN security on sp.securityID=s.securityID
    WHERE date = @myDate AND ticker=@ticker

    RETURN @retPrice
END
GO
Returning Queries

CREATE PROCEDURE getNYSETickersByVolume
    @date DATETIME,
    @volume INT
AS
BEGIN
    IF (@volume IS NULL) SELECT @volume=0
    SELECT s.ticker, s.securityID
    FROM security s INNER JOIN security_price sp
    ON s.securityID=sp.securityID
    WHERE exchangeFlags='2' AND sp.volume>@volume
        AND sp.date=@date ORDER BY ticker
END
GO

EXEC ivyuser.getGetNYSETickersByVolume '2005-01-05', 100000

<table>
<thead>
<tr>
<th>Ticker</th>
<th>SecurityID</th>
</tr>
</thead>
<tbody>
<tr>
<td>XYZ</td>
<td>230000</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EXEC getNYSETickersByVolume Dt, null

<table>
<thead>
<tr>
<th>Ticker</th>
<th>SecurityID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Begin/End Keywords

• Used to group and delineate sections within a stored procedure when the section contains multiple lines:

  – Procedure declarations
  – If statements
  – While statements

If @price > @lastPrice
BEGIN
  SELECT @price = @lastPrice
  SELECT @counter = @counter+1
END
ELSE SELECT @counter=10
CREATE PROCEDURE myProcedure
@volume INTEGER
AS
BEGIN
    IF @volume<10,000
    BEGIN
        'Do something specific for illiquid stocks
    END
ELSE
BEGIN
    -- Do something for all other stocks
END
END
GO
WHILE (SELECT AVG(contracts) FROM myTable) < 100
BEGIN
  UPDATE myTable SET contracts = contracts * 2
  IF (SELECT MAX(contracts) FROM myTable) > 300
    BREAK
  ELSE
    CONTINUE
END

• Doubles the quantity of contracts held until the average number is above 100 contracts

• Exits the loop if the maximum number of contracts in any row is above 300
Cursors

- Used to procedurally “walk” through a recordset - allows interaction of variables with row results

- Allows multiple processing steps on individual records

- Big network-related speed benefits:
  - a traditional client/server connection to a database manipulating large groups of records one at a time will be magnitudes slower than executing a stored procedure (assuming the stored procedure is able to perform the calculations)

However, cursors are orders of magnitude slower than direct SQL statements that select/update rows in bulk
Cursors Continued

• Declared the same way as variables

• Use a control loop to move one record at a time

• Drawbacks:
  – SQL statements are usually much faster than cursors
  – Limited function availability in stored procedures
  – Cursors are generally “forward only”
Assignment: count the number of times the closePrice for microsoft increased 10% day-to-day.

CREATE PROCEDURE get10pctIncreaseCount AS
BEGIN

-- Declare local variables
DECLARE @myPrice DECIMAL(10,2)
DECLARE @lastPrice DECIMAL(10,2)
DECLARE @increaseCount INTEGER

-- Declare cursor for close price of MSFT ordered by date
DECLARE myCursor CURSOR FOR
    SELECT closePrice FROM security_price
    WHERE ticker= ‘MSFT’
    ORDER BY date
Example of Cursor (2 / 2)

-- Initialize @lastPrice and @increaseCount
SELECT @lastPrice = -99
SELECT @increaseCount = 0

-- Open the cursor
OPEN myCursor

-- Fetches the first record and stores the entry into @myPrice
FETCH NEXT FROM myCursor INTO @myPrice
Example of Cursor (3 / 3)

-- Loop through records
WHILE @@FETCH_STATUS = 0
BEGIN

  -- If there is a 10% increase in price, update count
  IF (@myPrice/@lastPrice)>1.1
    SELECT @increaseCount = @increaseCount+1

  -- Update @lastPrice to current price @myPrice
  SELECT @lastPrice = @myPrice

  FETCH NEXT FROM myCursor INTO @myPrice
END

RETURN @increaseCount
END
CREATE PROCEDURE get10pctIncreaseCount AS
BEGIN

-- Declare local variables
DECLARE @myPrice DECIMAL(10,2)
DECLARE @lastPrice DECIMAL(10,2)
DECLARE @increaseCount INTEGER

-- Declare cursor for close price of MSFT ordered by date
DECLARE myCursor CURSOR FOR
    SELECT closePrice FROM security_price
    WHERE ticker= 'MSFT'
    ORDER BY date

-- Initialize @lastPrice and @increaseCount
SELECT @lastPrice = -99
SELECT @increaseCount = 0

-- Open the cursor
OPEN myCursor

-- Fetches the first record and stores the entry into @myPrice
FETCH NEXT FROM myCursor INTO @myPrice

-- Loop through records
WHILE @@FETCH_STATUS = 0 BEGIN
    -- If there is an increase in price, update count
    IF (@myPrice/@lastPrice)>1.1
        SELECT @increaseCount = @increaseCount+1

    -- Update @lastPrice to current price @myPrice
    SELECT @lastPrice = @myPrice

    FETCH NEXT FROM myCursor INTO @myPrice
END

RETURN @increaseCount
END
Example of Cursor (3 / 3)

```sql
select count(*)
from security_price s1
WHERE
(s1.closeprice/
  (select top 1 closePrice FROM security_price s2
   WHERE s1.securityID=s2.securityID and s2.date<s1.date
   ORDER BY s2.date DESC))>1.1
AND s1.securityID=107525
```

34 seconds
Advanced Functionality

• Access DLLs and custom libraries directly from stored procedures (e.g. matlab libraries)
  Functions are called in the same manner as standard SQL functions

• Ability to create forward/backward cursors

• TRIGGERS
  – Allows specific actions to take place when one or more of the following occur:
    • INSERT
    • UPDATE
    • DELETE
Boundary conditions

• Next Time:
  – Two weeks of Michael!

• Then:
  – More Cursors
  – Fact checking and interpretation
  – Real Time vs. batched data

• Use MAX/MIN and spot check for problems (a max return of 50,000% is very impressive, even for Madoff) Kiervel Leeson ...