SQL Beyond the Basics

Jerome Hughes
Freelance Developer
Omni Power i Day of Education 2010

Beyond the Basics

- What basics?
  - Just the most basic of queries
  - The sort of statement usually issued "ad-hoc" and left behind
    - SELECT * FROM library/file WHERE field = 'value'
    - SELECT * FROM collection/table WHERE column = 'value'

What's Good about these basics?

- Fill a need when created
- Way easier than writing an equivalent program

What's not so good about these basics?

- Scroll away upward, using STRSQL
- Always starting over, more trouble to find it
- So, how can we move beyond this?

SQL strategies

- To become fluent in a new language, it's best to build on what's been learned beforehand
- To build on what's been learned beforehand, it's easiest if your previous attempts are easily available for review
- So... save your queries, giving them names and descriptions, adding comments and managing them just like you do your RPG programs

Manage your queries!

- The first and often only tool most IBM i programmers used for SQL was STRSQL, which is cumbersome for this, so it's less than "natural" for most
- There are many ways to accomplish this, but it's important to choose one or more and...

Capture your work!

- Stop letting your simpler queries scroll up into oblivion!
- Keep ‘em around to start with when building more complex queries

That simple query...

- Type it faster than find it!
  - But is it, really? Is it adjusted and rerun, and what about next time?
- SELECT orddat, order, amount FROM orders WHERE orddat BETWEEN 20100101 and 20100227
  - What's here that might be reusable?

... is the seed of something less simple!

- Save it by subject, with a name like OrderDateRange
- Then it can be found and adapted to a new name and purpose
- SELECT orddat, count(*), sum(amount) FROM orders WHERE orddat BETWEEN 20070101 and 20070227 GROUP BY orddat ORDER BY orddat

Recipes you can mix together!

- In source containers named for subject and/or purpose, drop any misfires, copy forward successes, commenting!
- Keep what works around for quick reuse and adaptation
- SQL queries (on their own) are a non-procedural language where you specify what you want and let the database figure out how to retrieve it

Starting today!

- Becoming adept with applying more complicated SQL queries is good preparation for learning to write programs that make use of SQL queries
- When writing SQL programs, they'll be stored in source files, so why wait, accelerate!

Simple SELECT statement

SELECT * FROM jhughes/orders

- save it as “orders”
- then comment, copy and adapt

```
-- SELECT * FROM jhughes/orders
SELECT orddat, order, amount
FROM jhughes/orders
WHERE orddat BETWEEN 20070101 AND 20070227
ORDER BY orddat, order
```

Format for you!

- follow consistent case rules for readability break lines when new SQL keywords encountered
- comment lines to "turn them off/on" (cut/paste, editor cmds)
- copy and adjust lines to adapt statements

SELECT with JOIN

```
SELECT a.orddat, a.order, a.custid, b.item, b.quantity, b.price, b.quantity *
b.price lineExt FROM jhughes/orders a

JOIN jhughes/orderLines b

ON a.order = b.order

WHERE orddat BETWEEN 20070101 AND 20070227

ORDER BY orddat, order
```

- correlations “a” and “b” qualify fields to files
- calculation name can be overridden after declaration
- give it a name like “orderLines”

EXCEPTION JOIN

SELECT a.order, a.line, a.quantity, a.item, a.price

FROM jhughes/orderLines a

EXCEPTION join jhughes/order b

ON a.order = b.order

- returns order lines without a matching order header

plain JOIN ?

SELECT a.orddat, a.order, a.custid, b.item, b.quantity, b.price, b.quantity * b.price lineExt

FROM jhughes/orders a

JOIN jhughes/orderLines b

ON a.order = b.order

- regular JOIN does not return order without lines

LEFT OUTER JOIN

```
SELECT a.orddat, a.order, a.custid,
b.item, b.quantity, b.price, b.quantity * b.price lineExt
FROM jhughes/orders a
LEFT OUTER JOIN jhughes/orderLines b
ON a.order = b.order
```

- returns order without lines, with null values in line fields

GROUP BY for summaries

SELECT item, sum(quantity)

FROM jhughes/orderLines

GROUP BY item

ORDER BY item

- summarizes by GROUP BY column(s)
- so all other columns must be aggregated or errors will occur

GROUP BY strategies

- start with grouped columns, add aggregated columns
- start with grouping and ordering alike
- min() or max() can get only value when all in a group match, like an order line item description might here
- other aggregators include avg(), count(*)

GROUP BY for ranking

```
SELECT item, sum(quantity) qtySold
FROM jhughes/orderLines
GROUP BY item
ORDER BY qtySold DESC
```

- show biggest sellers first with DESC

HAVING groups like...

```
SELECT item, sum(quantity) qtySold
FROM jhughes/orderLines
GROUP BY item
HAVING qtySold > 500
ORDER BY qtySold DESC
```

- establish a floor on the summary with HAVING
- like a post-GROUPing WHERE clause

WHERE clause variations

- BETWEEN column/value AND column/value
- IN(value, value, value...)
- IN(SELECT column FROM table WHERE...)
- LIKE 'string%' (% = any number of chars)
- LIKE 'string_' (_ = any one character)
- IS NULL (not "= NULL")
- NOT flips any of these

Counting...

```
SELECT state, count(*)
FROM jhughes/orders
WHERE orddate between 20070101 AND 20070227
GROUP BY state
ORDER BY count(*) DESC
```

- shows distribution of records by code

... and Sampling

SELECT *

FROM jhughes/orders a

WHERE MOD(RRN(a),100)=0

- shows every 100th row (assuming even distribution)

UNION combines multiple SELECTs

```
SELECT class, item FROM jhughes/itemClassA
WHERE class LIKE 'A%'
UNION
SELECT class, item FROM jhughes/itemClassB
WHERE class LIKE 'B%'
ORDER BY class, item
```

- returns records from both SELECTS in one result

UNION details

- each SELECT gets its own WHERE clause
- one ORDER BY clause for entire construct
- field list types must match across SELECTs
- first SELECT determines naming
- fields can be sourced from anywhere

CASE gets procedural in SELECT

```
SELECT key, name,
CASE code
WHEN 'A' THEN 'After'
WHEN 'B' THEN 'Before'
END
FROM table...
```

- allows expansions of codes, etc.
- can also be done with conditionals, like...

CASE with conditions

```
SELECT key, name,
CASE WHEN colval < 10 THEN 'singledigit'
     WHEN colval < 100 THEN 'doubledigits'
     ELSE 'hundredsormore' END
FROM table...
```

- allows labeling of ranges, etc.

Subqueries for selection

```
SELECT order, amount
FROM orders
WHERE amount >
  (SELECT AVG(amount) FROM orders)
```

- returns above average orders

More Subquery Selection

SELECT customer, order, amount

FROM orders

WHERE customer IN

(SELECT customers FROM topcusts)

- selects only orders from customers listed in topcusts

Finding unattached lines

```
SELECT a.*
FROM orderLines a
WHERE NOT a.order IN
  (SELECT b.order FROM orders b)
```

- shows only those lines not attached to orders

... and lineless orders

```
SELECT a.*
FROM orders a
WHERE NOT a.order IN
  (SELECT DISTINCT b.order FROM orderLines
b)
```

- shows only those orders without lines

Queries that change things

- When moving beyond SELECT to UPDATE and DELETE
- Test WHERE clauses first using SELECTs
- Prove your test results are satisfactory first!
  - Run to a file, then query to prove it matches

Be sure beforehand!

- When ready, run first on test copy of data
- After testing, back up what will change first!
- Create and document your plan with comments/notes
  - Verify your plan, and protect your assets!

Updating related records

- Use caution, do it on test copies first, etc.
- Here's the template for doing one...
  - keys must specify unique records
  - may be compound keys concatenated
  - use CHAR(column) to concatenate numeric keys

Related record template

- UPDATE tablea a SET a.updatecolumn =
  (SELECT b.value FROM tableb b
  WHERE b.key = a.key)
- WHERE a.key =
- (SELECT b.key FROM tableb b
  WHERE b.key = a.key)

Trying it out on a customer table

- cusmas and cusnew were the same
- both are keyed by cusnbr
- changes made to cusnew's cuscls are needed in cusmas
- start by SELECTing the target set

Trying it out on customer table II

```
SELECT a.cusnbr, a.cusnam, a.cuscls, b.cuscls
FROM cusmas a
JOIN cusnew b
ON a.cusnbr = b.cusnbr
WHERE a.cuscls <> b.cuscls
```

- shows record key, description, old and new classes
- proves correct records will be adjusted as desired

Before you go do it

- Be sure you have the right records verified
- Make a copy of the data
- Run it against that copy
- Check that it worked
- Make a backup of what you're going to change
- Be careful and document, then read what was written!

Scalar Subselect

```
UPDATE cusmas a SET a.cuscls =

 (SELECT b.cuscls FROM cusnew b

  WHERE b.cusnbr = a.cusnbr)
WHERE a.cusnbr =

 (SELECT b.cusnbr FROM cusnew b

  WHERE b.cusnbr = a.cusnbr)
```

- prove this works on a subset and it will save a lot of time on processing a large table which needs updating by key!

Embedding SQL in RPG programs

- All SQL statements must be delimited by

  /EXEC SQL and /END-EXEC statements
- Source is compiled with CRTSQLRPG command
- SQL statements are first evaluated by SQL precompiler
- At execution time, errors are returned in SQLCOD
  - don't define this, it will just be there

Opening access path

- First declare a cursor to manage the path

  C/EXEC SQL

  C DECLARE CURSOR c1 FOR SELECT * FROM table1

  C/END-EXEC

  - execution of this code establishes the access path

Retrieving a row

- "Read" a record from the path with a FETCH statement

  C/EXEC SQL

  C FETCH c1 INTO :dsname

  C/END-EXEC

- :dsname is a data structure field for SELECT clause record image
  - use an external DS to pull in columns
  - access data structure subfields to use data

Check state

- SQLSTT & SQLCOD are automatically included
  - don't need to be defined
- 0 = ok, other codes denote EOF, errors

```
SQLSTT  IFEQ 0
            EXSR PROCESS
            ENDIF
```

... and close

- close the path with a CLOSE cursor-name

```
C/EXEC SQL
C CLOSE c1
C/END EXEC
```

SQL Trigger example 1.1

```
CREATE TRIGGER new_hire
 AFTER  INSERT ON employee
 FOR EACH ROW MODE DB2SQL
   UPDATE company_stats SET nbemp = nbemp + 1
```

SQL Trigger example 1.2

```
CREATE TRIGGER former_employee
 AFTER  DELETE ON employee
 FOR EACH ROW MODE DB2SQL
 BEGIN ATOMIC
  UPDATE company_stats SET nbemp = nbemp - 1;
END
```

- 1.1 & 1.2 together keep a count of employees updated

SQL Trigger example 2.1

```
CREATE TRIGGER reorder
 AFTER UPDATE OF onhand, max_stocked ON parts
 REFERENCING NEW_TABLE AS ntable
 FOR EACH STATEMENT MODE DB2SQL
 BEGIN ATOMIC
  (SELECT ...)
 END
```

SQL Trigger example 2.2

```
(...BEGIN ATOMIC...)
SELECT issue_ship_request(max_stocked - on_hand,
 partno)
FROM ntable
WHERE on_hand < 0.10 * max_stocked;
END
```

- only runs once per statement, finds rows where stock is low and runs UDF issue_ship_request for each one

SQL Trigger example 3.1

```
CREATE TRIGGER sal_adj
 AFTER UPDATE OF salary ON employee
 REFERENCING OLD AS old_emp
                    NEW AS new_emp
 FOR EACH ROW MODE DB2SQL
 WHEN (new_emp.salary > old_emp.salary * 1.2))
 BEGIN ATOMIC
  (do something)
END
```

SQL Trigger example 3.2

```
(BEGIN ATOMIC...)
    SIGNAL SQLSTATE '75001' ('Invalid Salary Increase
    exceeds 20%');
END
```

- throws SQL exception to requestor

SQL Stored Procedure example

```
CREATE PROCEDURE update_salary_1
 (IN employee_number CHAR(10),
  IN rate DECIMAL(6,2))
  LANGUAGE SQL MODIFIES SQL DATA
  UPDATE corpdata.employee
    SET salary = salary * rate
      WHERE empno = employee_number
```

- declares parameters and runs statement with columns & parms

SQL Stored Procedure components

- expand beyond single statement with SQL control statements
  - CALL, CASE, FOR, IF, ITERATE, LEAVE, LOOP, REPEAT, RETURN, WHILE
- run from client or with SQL CALL from another procedure
- see Robert Andrews in Session 5!

Get help with it...

- check out http://www.midrange.com
- email me directly at...
  - jromeh@aol.com or jromeh@comcast.net
- will be glad to help when there's time
- it's always good to have a sounding board
  - thanks to the many folks who have served in this role for me!