Ken's SQL Syntax Refresher

Select

```
SELECT [ * | ALL | DISTINCT column1, column2  ]
FROM table1 [ , table2 ];

SELECT [ * | ALL | DISTINCT column1, column2  ]
FROM table1 [ , table2 ]
WHERE [ condition1 | expression1 ] [ AND condition2 | expression2 ];

SELECT [ * | ALL | DISTINCT column1, column2  ]
FROM table1 [ , table2 ]
WHERE [ condition1 | expression1 ] [ AND condition2 | expression2 ]
ORDER BY column1| integer [ ASC | DEC ]

SELECT * FROM aTable;

SELECT DISTINCT * FROM aTable ORDER BY id ASC;

SELECT col1,col2 FROM aTable WHERE dateCol > (now() – 1 day);

SELECT DISTINCT COUNT(product_id) FROM productsTable;

SELECT city, AGV(salary) FROM employeeTbl
WHERE city <> 'Greenwood'
GROUP BY city
HAVING AGV(salary) > 20000
ORDER BY 2
```

Where clause Conjunctive Operators

AND, OR

```
SELECT P.product_name, O.orderDate, O.quantity
FROM products P, orders O
WHERE P.serial_number = O.serial_number AND P.vendor_number = vendor_number;

SELECT emp_id, salary
FROM employeeTbl
WHERE salary IS NOT NULL OR
pay_rate IS NOT NULL
```

Compund Queries

UNION, UNION ALL, EXCEPT | MINUS, INTERSECT

The previous OR is the same as:

```
SELECT emp_id, salary
FROM employeeTbl
WHERE salary IS NOT NULL
UNION                          Union is Distinct = no duplicated rows (vs Union All)
SELECT emp_id, salary
FROM employeeTbl
WHERE pay_rate IS NOT NULL
```

```
SELECT * FROM Orders
WHERE Quantity BETWEEN 1 AND 100 EXCEPT   <= Oracle uses MINUS
SELECT * FROM Orders WHERE Quantity BETWEEN 50 AND 75;
```

Where clause operators

=, <>, >, <, >=, <=, BETWEEN, LIKE, NOT LIKE, IN, NOT IN, IS NULL, IS NOT NULL, EXISTS, NOT EXISTS

```
WHERE salary LIKE '200%'     means match on wildcard 200*
WHERE salary LIKE '_00'      means match on wildcard ?00
```

```
SELECT a.FirstName, a.LastName FROM Person.Contact AS a
WHERE EXISTS (
  SELECT * FROM HumanResources.Employee AS b
  WHERE a.ContactId =     b.ContactID AND a.LastName = 'Johnson');
- is the same as -
SELECT a.FirstName, a.LastName FROM Person.Contact AS a
WHERE a.LastName IN (
SELECT a.LastName FROM HumanResources.Employee AS b
WHERE a.ContactId = b.ContactID AND a.LastName = 'Johnson');
```

Summary operators

COUNT, SUM, AVG, MAX, MIN

```
SELECT SUM(salary) as sum_salary FROM employeeTbl
SELECT COUNT(employeeId) as emp_count FROM employeeTbl
SELECT MAX(job_number) as max_job FROM aTable
SELECT MAX(job_number) as max_job, MIN(job_number) as min_job  FROM aTable
```

Subquery with SELECT

```
SELECT E.emp_id, EP.pay_rate
FROM employee_tbl E, employee_pay_tbl EP
```

```
WHERE E.emp_id = EP.emp_id
AND EP.pay_rate > (SELECT pay_rate FROM employee_tbl WHERE emp_id =
'12345')
```

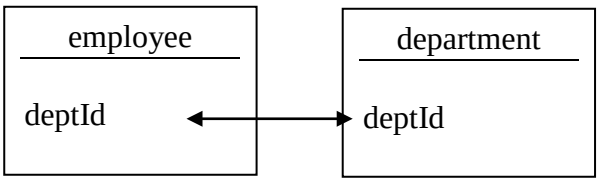
Efficent Queries

1. Query the smallest table first
2. Use the most restrictive query first
3. Use IN rather than OR

	Use this one:
SELECT emp_id, last_name, first_name FROM employee_tbl WHERE city = 'Indianapolis' OR city = 'Brownsburg' OR city = 'Greenfield'	SELECT emp_id, last_name, first_name FROM employee_tbl WHERE city IN ('Indianapolis', 'Brownsburg', 'Greenfield')

Join

```
SELECT field-list
FROM  table-1
        { INNER | LEFT OUTER | RIGHT OUTER } JOIN table-2
ON table-1.field-1 { = | < | > | <= | >= |<> } table-2.field.2
[ WHERE  selection-criteria ]
[ ORDER BY field-list ]
```




```
SELECT * FROM employee, department
WHERE employee.deptId = department.deptId
```

- is equivalent to –

```
SELECT * FROM employee
INNER JOIN department ON employee.deptId = department.deptId
```

- is equivalent to (equi join) -

```
SELECT * FROM employee
INNER JOIN department USING (deptId)
```

Natural Join: only has one column with the matches from both tables:

```
SELECT * FROM employee NATURAL JOIN department
```

Join on more than two tables

```
SELECT Vendors
        INNER JOIN Invoices
                ON Vendors.VendorID = Invoices.VendorID
        INNER JOIN InvoiceLineItems
```

ON Invoices.InvoiceID = InvoiceLineItems.InvoiceID

There no SELECT DISTINCT on Joins, use a "Nautral Join" A Natural Join is a special equi(equals)-join returns the columns and rows in common between multiple tables

SELECT * FROM employee NATURAL JOIN department

SQL Server supports additional *cross joins* and *full outer joins*

SELECT * FROM employee CROSS JOIN department
same as:
SELECT * FROM employee, department;

Outer Joins

returns blank (or non existant) rows in the other table

SELECT E.id P.id FROM employeeTable E, employeePay P, WHERE E.id = P.id(+);

SELECT * FROM employee
LEFT OUTER JOIN department
ON employee.DepartmentID = department.DepartmentID

Insert

INSERT INTO *table-name* [(*field-list*)]
VALUES (*value-list*)

INSERT INTO aTable (col1, col2, col2) VALUES ('value1', 'value2', NULL);

Subquery with Insert

INSERT INTO table-name (*field-columns)*
SELECT-statement *<= for the values to put into the field-columns*

INSERT INTO aTable (col1, col2, col3)
SELECT FROM anotherTable col1,col2,col3 WHERE dateCol > (now() – 1 day);

INSERT INTO InvoiceArchive
SELECT * FROM Invoices
WHERE InvoiceTotal – PaymentTotal – CreditTotal = 0

Update

UPDATE *table-name*
SET *expression-1* [*,expression-2*] …
WHERE *selection-criteria*

UPDATE ordersTable
SET quantity = 5
WHERE order_no = '12345';

Subquery with Update

UPDATE employee_pay_tbl
SET pay_rate = pay_rate * 1.1
WHERE emp_id IN
(SELECT emp_id FROM employee_tbl WHERE city = 'Indianapolis')

Delete

DELETE FROM *table-name*
WHERE *selection-criteria*

DELETE FROM orders_table
WHERE order_no = '12345' AND date < (now() – 30 days);

Subquery with Delete

DELETE FROM employee_pay_tbl
WHERE emp_id = (SELECT emp_id FROM employee_tbl WHERE last_name = 'Freed' AND first_name = 'Ken')

Views

A View is a predefined query that's stored in a database

CREATE VIEW VendorsMin AS
SELECT VendorName, VandorStatus
FROM Vendors

Use the above view just like it was a table:

SELECT * FROM VendorsMin …

Stored Procedures

A **Stored Procedure** is one or more SQL statements that

- can accept input parameters and
- that have been compiled and stored within the database.

Stored Procedures can improve db performance since they're compiled and optimized the first time they're executed

Their alternative is to execute Transact-SQL programs stored locally on client computers

Lots of places that use SQL Server use stored procs for database I/O. While it gets better performance & security, it can be a real pain to figure out what is going on if there is data issue. The SQL server profiler can be a useful utility for troubleshooting though.

- The biggest complaint with Stored Procedures is when there is a series of transactions that were fired off as a bunch of stored procedures, it can be difficult and time consuming to replicate the events that led up to where the problem is occurring, depending on how many stored procs are called.

A **Trigger** is a special type of stored procedure that's *fired* when record are inserted, updated or deleted from a table.

- Triggers can be used to enforce Referential Integrity.
- Triggers can be used to check the validity of data updated or inserted into a table

```
CREATE PROCEDURE VendorByState @State Char As
        SELECT VendorName, VendorState, VendorPhone
        FROM Vendors
        WHERE VendorState = @State
        ORDER BY VendorName
```

Using the above stored procedure:

Execute VendorsByState for State = "CA"

Some Common Queries

CREATE DATABASE aDataBase;

```
CREATE TABLE testTable (id INT NOT NULL AUTO_INCRMENT,
PRIMARY_KEY(id), testField VARCHAR(50);
```

GRANT `insert,update,select on *.* to aTable@localhost identified by 'aPassWord';`

INSERT INTO testTable (testField) VALUES ('first message');

DELETE FROM testTable WHERE id=2;

UPDATE members set expire_ts='2009-09-09 12:00:00' where id=26;

ALTER TABLE some_table ADD expired TYPE boolean; // add a column

ALTER TABLE (aTableName) ADD <colname> <coltype>;

show tables;
show tables like 'journal%'
describe service_attr;

MISC

Database Normalization

1st normal form:

No repeated data. Create separate tables

2nd normal form:

Data depends on the WHOLE key (e.g., in case you concatenate two attributes together to make a unique key

e.g.: person,skill <= address *violates since address has nothing to do with skill*

3rd normal form:

Attributes don't depend on an attribute which depends on the whole key,

e.g.: tournament,year <= winner <= winner's birthday

Referential Integrity:

Means that the records with foreign keys always have records with a matching primary key in another table.

- This means that you can't add a record with a foreign key value if the primary key record doesn't already exist (in another table).
- It also means that you can't delete a primary key record without deleting related foreign key records.

One way to enforce Referential Integrity is to define the primary and foreign key relationship within the database.

- SQL Server calls this **Foreign Key Constraints.**
  - When you do this, you're using what SQL Server calls **Declarative Referential Integrity (DRI)** = Referential Integrity is enforce automatically by the DBMS
- SQl Server doesn't provide for *cascading* changes and deletes to related tables.
  - If you want to be able to cascade changes and deletes with SQL Server, you can use Triggers to enforce referential integrity instead of Foreign Key Constraints,

    i.e. Triggers let you *cascade* changes from the Primary Keys in one table to the Foreign Keys in a related table.

A 1-1 table relationship means you can just put it all in one table.

- Sometimes you'd do this for secure columns, or too many columns

Handling Lists of Things in Tables

Wrong way to design:

firstName	lastName	Instrument1	Instrument2	Instrument3
Angie	Beltran	Guitar	Sax	
Laura	Chow	Sax	Clarinet	Piano
Debbie	Moss	Drums	Guitar	
Raul	Garcia	Guitar	Piano	Drums

Right way to design (repeat some info in the rows), which is a one-to-many relationship:

firstName	lastName	Instrument
Angie	Beltran	Guitar
Angie	Beltran	Sax
Debbie	Moss	Drums
Debbie	Moss	Guitar
Raul	Garcia	Guitar
Raul	Garcia	Piano
Raul	Garcia	Drums