SQL Joins
and Query Optimization

How it works

by Brian Gallagher
www.BrianGallagher.com

Why JOIN?

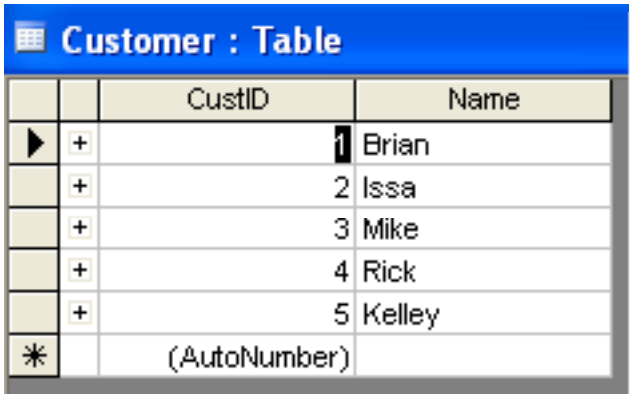
- Combine data from multiple tables
- Reduces the number of queries needed
- Moves processing burden to database server
- Improves data integrity (over combining data in program code)

Types of JOINs

- INNER
  - the most common, return all rows with matching records
  - `SELECT * FROM T1 INNER JOIN T2 ON T1.fld1 = T2.fld2`
- LEFT or LEFT OUTER
  - return all rows on the left (first) table and right (second)
  - If no matching record on the right side, NULL-values for each field are returned
  - `SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.fld1 = T2.fld2`
- FULL or FULL OUTER
  - return all rows on the left (first) table and right (second)
  - If no matching record on the left or right side, NULL-values for each field are returned
  - `SELECT * FROM T1 FULL OUTER JOIN T2 ON T1.fld1 = T2.fld2`
- CROSS
  - the least common, return all possible row combinations
  - `SELECT * FROM T1, T2`
- RIGHT or RIGHT OUTER
  - **Don't use them without a very good reason.**
  - They do not add any functionality over a LEFT JOIN and make code more confusing
  - Works the same as the LEFT and LEFT OUTER JOINs, but the second table is the one which all rows are returned from. These two queries are functionally identical:
  - `SELECT * FROM T1 LEFT JOIN T2 ON T1.fld1 = T2.fld2`
  - `SELECT * FROM T2 RIGHT JOIN T1 ON T1.fld1 = T2.fld2`

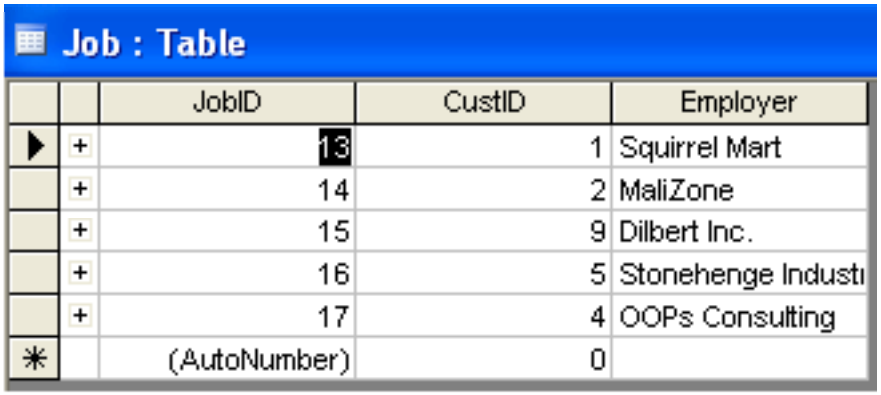
Sample Test Data

- Sample Customer and Job records



Customer : Table

		CustID	Name
▶	+	1	Brian
	+	2	Issa
	+	3	Mike
	+	4	Rick
	+	5	Kelley
✳		(AutoNumber)	



Job : Table

		JobID	CustID	Employer
▶	+	13	1	Squirrel Mart
	+	14	2	MaliZone
	+	15	9	Dilbert Inc.
	+	16	5	Stonehenge Industı
	+	17	4	OOPs Consulting
✳		(AutoNumber)	0	

INNER JOIN

Customer : Table

	CustID	Name
+	1	Brian
+	2	Issa
+	4	Rick
+	5	Kelley
+	6	Mike

Job : Table

	JobID	CustID	Employer
+	13	1	Squirrel Mart
+	14	2	MaliZone
+	15	9	Dilbert Inc
+	16	5	Stonehenge Ltd
+	17	4	OOPs Consulting

Tables to be joined

Inner Join

```
SELECT *
FROM Customer C, Job J
ON C.CustID = J.CustID
```

SQL command being executed

Customer : Table

	CustID	Name
+	1	Brian
+	2	Issa
+	4	Rick
+	5	Kelley
+	6	Mike

Job : Table

	JobID	CustID	Employer
+	13	1	Squirrel Mart
+	14	2	MaliZone
+	15	9	Dilbert Inc
+	16	5	Stonehenge Ltd
+	17	4	OOPs Consulting

Values that do not match join condition (will be excluded)

Matching values exist in both tables

Inner Join : Select Query

	C.CustID	Name	JobID	J.CustID	Employer
	1	Brian	13	1	Squirrel Mart
	2	Issa	14	2	MaliZone
	4	Rick	17	4	OOPs Consulting
	5	Kelley	16	5	Stonehenge Ltd

No matching value in other table, match to NULL instead

NULL value; No matching value was found

LEFT (OUTER) JOIN

Customer : Table

	CustID	Name
+	1	Brian
+	2	Issa
+	4	Rick
+	5	Kelley
+	6	Mike

Job : Table

	JobID	CustID	Employer
+	13	1	Squirrel Mart
+	14	2	MaliZone
+	15	9	Dilbert Inc
+	16	5	Stonehenge Ltd
+	17	4	OOPs Consulting

Tables to be joined

Left [outer] Join

```
SELECT *
FROM Customer C LEFT JOIN Job J
ON C.CustID = J.CustID
```

SQL command being executed

Customer : Table

	CustID	Name
+	1	Brian
+	2	Issa
+	4	Rick
+	5	Kelley
+	6	Mike

Job : Table

	JobID	CustID	Employer
+	13	1	Squirrel Mart
+	14	2	MaliZone
+	15	9	Dilbert Inc
+	16	5	Stonehenge Ltd
+	17	4	OOPs Consulting

Values that do not match join condition (will be excluded)

Matching values exist in both tables

Left Join : Select Query

	C.CustID	Name	JobID	J.CustID	Employer
	1	Brian	13	1	Squirrel Mart
	2	Issa	14	2	MaliZone
	4	Rick	17	4	OOPs Consulting
	5	Kelley	16	5	Stonehenge Ltd
	6	Mike			

No matching value in other table, match to NULL instead

NULL value; No matching value was found

One-To-Many LEFT JOIN

Customer : Table

	CustID	Name
+	1	Brian
+	2	Issa
+	4	Rick
+	5	Kelley
+	6	Mike

Job2 : Table

JobID	CustID	Employer
13	1	Squirrel Mart
14	2	MaliZone
15	9	Dilbert Inc
16	5	Stonehenge Ltd
17	4	OOPs Consulting
18	1	Possum Hut
19	1	Lizard Land

Tables to be joined

```
SELECT *
FROM Customer C RIGHT JOIN Job2 J2
ON C.CustID = J2.CustID
```

SQL command being executed

Customer : Table

	CustID	Name
+	1	Brian
+	2	Issa
+	4	Rick
+	5	Kelley
+	6	Mike

Job2 : Table

JobID	CustID	Employer
13	1	Squirrel Mart
14	2	MaliZone
15	9	Dilbert Inc
16	5	Stonehenge Ltd
17	4	OOPs Consulting
18	1	Possum Hut
19	1	Lizard Land

Values that do not match join condition (will be excluded)

Matching values exist in both tables

Left Join One-to-Many : Select Query

C.CustID	Name	JobID	J2.CustID	Employer
1	Brian	13	1	Squirrel Mart
1	Brian	18	1	Possum Hut
1	Brian	19	1	Lizard Land
2	Issa	14	2	MaliZone
4	Rick	17	4	OOPs Consulting
5	Kelley	16	5	Stonehenge Ltd
6	Mike			

No matching value in other table, match to NULL instead

NULL value; No matching value was found

RIGHT (OUTER) JOIN

Customer : Table

	CustID	Name
+	1	Brian
+	2	Issa
+	4	Rick
+	5	Kelley
+	6	Mike

Job : Table

	JobID	CustID	Employer
+	13	1	Squirrel Mart
+	14	2	MaliZone
+	15	9	Dilbert Inc
+	16	5	Stonehenge Ltd
+	17	4	OOPs Consulting

Tables to be joined

Right [outer] Join

```
SELECT *
FROM Customer C RIGHT JOIN Job J
ON C.CustID = J.CustID
```

SQL command being executed

Customer : Table

	CustID	Name
+	1	Brian
+	2	Issa
+	4	Rick
+	5	Kelley
+	6	Mike

Job : Table

	JobID	CustID	Employer
+	13	1	Squirrel Mart
+	14	2	MaliZone
+	15	9	Dilbert Inc
+	16	5	Stonehenge Ltd
+	17	4	OOPs Consulting

Values that do not match join condition (will be excluded)

Matching values exist in both tables

Right Join : Select Query

C.CustID	Name	JobID	J.CustID	Employer
1	Brian	13	1	Squirrel Mart
2	Issa	14	2	MaliZone
4	Rick	17	4	OOPs Consulting
5	Kelley	16	5	Stonehenge Ltd
		15	9	Dilbert Inc

No matching value in other table, match to NULL instead

NULL value; No matching value was found

FULL (OUTER) JOIN

Customer : Table

CustID	Name
1	Brian
2	Issa
4	Rick
5	Kelley
6	Mike

Job : Table

JobID	CustID	Employer
13	1	Squirrel Mart
14	2	MaliZone
15	9	Dilbert Inc
16	5	Stonehenge Ltd
17	4	OOPs Consulting

Tables to be joined

Full (outer) Join

```
SELECT *
FROM Customer C FULL JOIN Job J
ON C.CustID = J.CustID
```

SQL command being executed

Customer : Table

CustID	Name
1	Brian
2	Issa
4	Rick
5	Kelley
6	Mike

Job : Table

JobID	CustID	Employer
13	1	Squirrel Mart
14	2	MaliZone
15	9	Dilbert Inc
16	5	Stonehenge Ltd
17	4	OOPs Consulting

Values that do not match join condition (will be excluded)

Matching values exist in both tables

Full Join : Union Query

C.CustID	Name	JobID	J.CustID	Employer
		15	9	Dilbert Inc
1	Brian	13	1	Squirrel Mart
2	Issa	14	2	MaliZone
4	Rick	17	4	OOPs Consulting
5	Kelley	16	5	Stonehenge Ltd
6	Mike			

No matching value in other table, match to NULL instead

NULL value; No matching value was found

FULL (OUTER) JOIN

- FULL JOIN is not supported in MS-Access
- Can be emulated using UNION queries

- They are rarely used

CROSS JOINs

The Join you never use…

except every time

Making it Big

- A CROSS JOIN combines every row on the left table with every row in the right table
- Resulting recordset will be the total of both row counts multiplied together
  - Left row has 10,000 records
  - Right row has 30,000 records
  - Resulting recordset has 300,000,000 records
- If no JOIN condition is specified, a CROSS JOIN will be executed

Joins each record to the other table

Customer : Table

	CustID	Name
+	1	Brian
+	2	Issa
+	4	Rick
+	5	Kelley
+	6	Mike

Job : Table

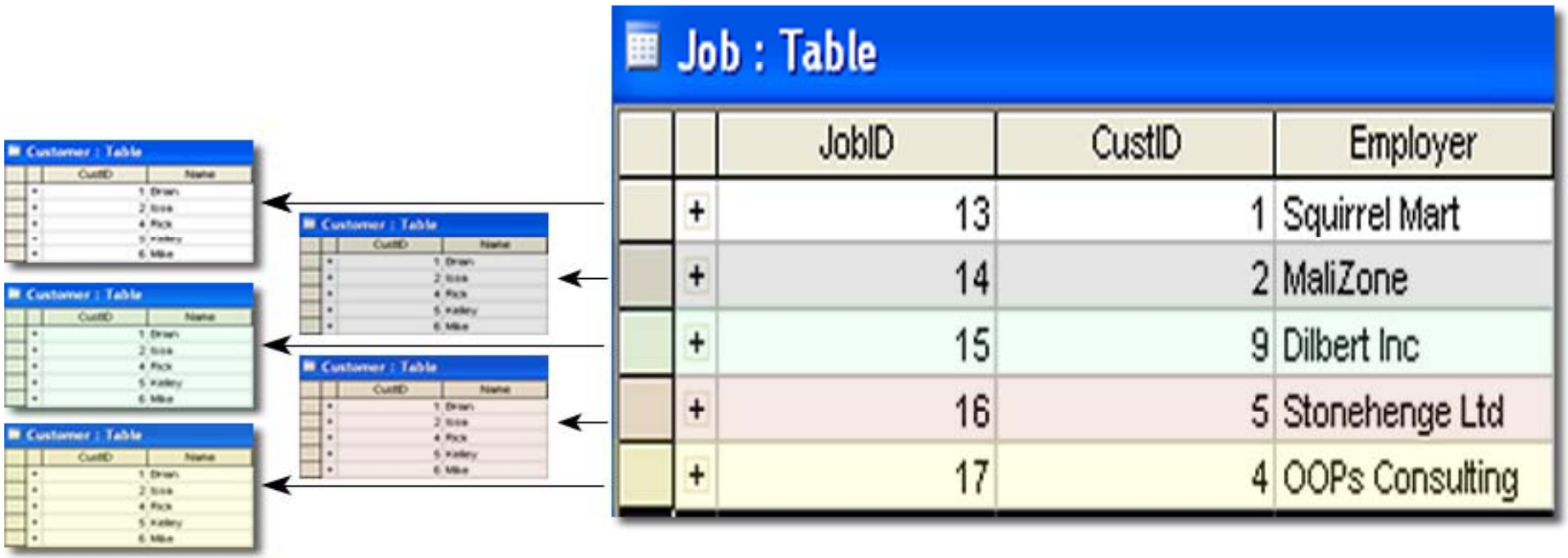
	JobID	CustID	Employer
+	13	1	Squirrel Mart
+	14	2	MaliZone
+	15	9	Dilbert Inc
+	16	5	Stonehenge Ltd
+	17	4	OOPs Consulting

Tables to be joined

Left [outer] Join

```
SELECT *
FROM Customer C, Job J
ON C.CustID = J.CustID
```

SQL command being executed



CROSS JOIN's resulting set of all record combinations

Job : Table

	JobID	CustID	Employer
+	13	1	Squirrel Mart
+	14	2	MaliZone
+	15	9	Dilbert Inc
+	16	5	Stonehenge Ltd
+	17	4	OOPs Consulting

Cross Join : Select Query

C.CustID	Name	JobID	J.CustID	Employer
1	Brian	13	1	Squirrel Mart
2	Issa	13	1	Squirrel Mart
4	Rick	13	1	Squirrel Mart
5	Kelley	13	1	Squirrel Mart
6	Mike	13	1	Squirrel Mart
1	Brian	14	2	MaliZone
2	Issa	14	2	MaliZone
4	Rick	14	2	MaliZone
5	Kelley	14	2	MaliZone
6	Mike	14	2	MaliZone
1	Brian	15	9	Dilbert Inc
2	Issa	15	9	Dilbert Inc
4	Rick	15	9	Dilbert Inc
5	Kelley	15	9	Dilbert Inc
6	Mike	15	9	Dilbert Inc
1	Brian	16	5	Stonehenge Ltd
2	Issa	16	5	Stonehenge Ltd
4	Rick	16	5	Stonehenge Ltd
5	Kelley	16	5	Stonehenge Ltd
6	Mike	16	5	Stonehenge Ltd
1	Brian	17	4	OOPs Consulting
2	Issa	17	4	OOPs Consulting
4	Rick	17	4	OOPs Consulting
5	Kelley	17	4	OOPs Consulting
6	Mike	17	4	OOPs Consulting

How **Cross** Joins Make **Inner** Joins

- Tables are joined using a CROSS JOIN
- JOIN criteria is evaluated, removing all records which don't match the "ON" condition

```
SELECT *
FROM Customers C
INNER JOIN Job J
ON C.CustID = J.CustID
```

- The remaining rows are the recordset returned for the INNER JOIN

Using “ON” criteria to select records

Inner Join

```
SELECT *
FROM Customer C INNER JOIN Job J
ON C.CustID = J.CustID
```

Cross Join : Select Query

C.CustID	Name	JobID	J.CustID	Employer
1	Brian	13	1	Squirrel Mart
2	Issa	13	1	Squirrel Mart
4	Rick	13	1	Squirrel Mart
5	Kelley	13	1	Squirrel Mart
6	Mike	13	1	Squirrel Mart
1	Brian	14	2	MaliZone
2	Issa	14	2	MaliZone
4	Rick	14	2	MaliZone
5	Kelley	14	2	MaliZone
6	Mike	14	2	MaliZone
1	Brian	15	9	Dilbert Inc
2	Issa	15	9	Dilbert Inc
4	Rick	15	9	Dilbert Inc
5	Kelley	15	9	Dilbert Inc
6	Mike	15	9	Dilbert Inc
1	Brian	16	5	Stonehenge Ltd
2	Issa	16	5	Stonehenge Ltd
4	Rick	16	5	Stonehenge Ltd
5	Kelley	16	5	Stonehenge Ltd
6	Mike	16	5	Stonehenge Ltd
1	Brian	17	4	OOPs Consulting
2	Issa	17	4	OOPs Consulting
4	Rick	17	4	OOPs Consulting
5	Kelley	17	4	OOPs Consulting
6	Mike	17	4	OOPs Consulting

Inner Join : Select Query

C.CustID	Name	JobID	J.CustID	Employer
1	Brian	13	1	Squirrel Mart
2	Issa	14	2	MaliZone
4	Rick	17	4	OOPs Consulting
5	Kelley	16	5	Stonehenge Ltd

One-to-Many CROSS JOIN

```
SELECT *
FROM Customer C, Job2 J2
```

Customer : Table

	CustID	Name
+	1	Brian
+	2	Issa
+	4	Rick
+	5	Kelley
+	6	Mike

Job2 : Table

JobID	CustID	Employer
13	1	Squirrel Mart
14	2	MaliZone
15	9	Dilbert Inc
16	5	Stonehenge Ltd
17	4	OOPs Consulting
18	1	Possum Hut
19	1	Lizard Land

One to Many: Cross Join to Inner Join

```
SELECT *
FROM Customer C INNER JOIN Job2 J2
ON C.CustID = J2.CustID
```

Cross Join - 1 to Many : Select Query

C.CustID	Name	JobID	J2.CustID	Employer
1	Brian	13	1	Squirrel Mart
2	Issa	13	1	Squirrel Mart
4	Rick	13	1	Squirrel Mart
5	Kelley	13	1	Squirrel Mart
6	Mike	13	1	Squirrel Mart
1	Brian	14	2	MaliZone
2	Issa	14	2	MaliZone
4	Rick	14	2	MaliZone
5	Kelley	14	2	MaliZone
6	Mike	14	2	MaliZone
1	Brian	15	9	Dilbert Inc
2	Issa	15	9	Dilbert Inc
4	Rick	15	9	Dilbert Inc
5	Kelley	15	9	Dilbert Inc
6	Mike	15	9	Dilbert Inc
1	Brian	16	5	Stonehenge Ltd
2	Issa	16	5	Stonehenge Ltd
4	Rick	16	5	Stonehenge Ltd
5	Kelley	16	5	Stonehenge Ltd
6	Mike	16	5	Stonehenge Ltd
1	Brian	17	4	OOPs Consulting
2	Issa	17	4	OOPs Consulting
4	Rick	17	4	OOPs Consulting
5	Kelley	17	4	OOPs Consulting
6	Mike	17	4	OOPs Consulting
1	Brian	18	1	Possum Hut
2	Issa	18	1	Possum Hut
4	Rick	18	1	Possum Hut
5	Kelley	18	1	Possum Hut
6	Mike	18	1	Possum Hut
1	Brian	19	1	Lizard Land
2	Issa	19	1	Lizard Land
4	Rick	19	1	Lizard Land
5	Kelley	19	1	Lizard Land
6	Mike	19	1	Lizard Land

Inner Join - 1 to Many : Select Query

C.CustID	Name	JobID	J2.CustID	Employer
1	Brian	13	1	Squirrel Mart
1	Brian	18	1	Possum Hut
1	Brian	19	1	Lizard Land
2	Issa	14	2	MaliZone
4	Rick	17	4	OOPs Consulting
5	Kelley	16	5	Stonehenge Ltd

Unpredictable Record Orders

- Databases are not required to return records in any particular order
- Records may be returned by the order they are stored on disk or may be unordered, depending on how the query engine handles data
- If you need data in a particular order, use an ORDER BY clause
- Some databases may return data presorted by Primary key or Clustered index
  - DO NOT DEPEND on this behavior
  - It is not reliably portable across different databases
  - Do not make a program rely on a behavior not specified by the program – bad programming style

Indexes

Unique, clustered, etc.

What are Indexes

- Indexes are a pre-sorted list of database field values
- This list speeds up queries A LOT
  - An index is much smaller than the full recordset
  - An index tracks only the fields specified when created

Indexes and Performance

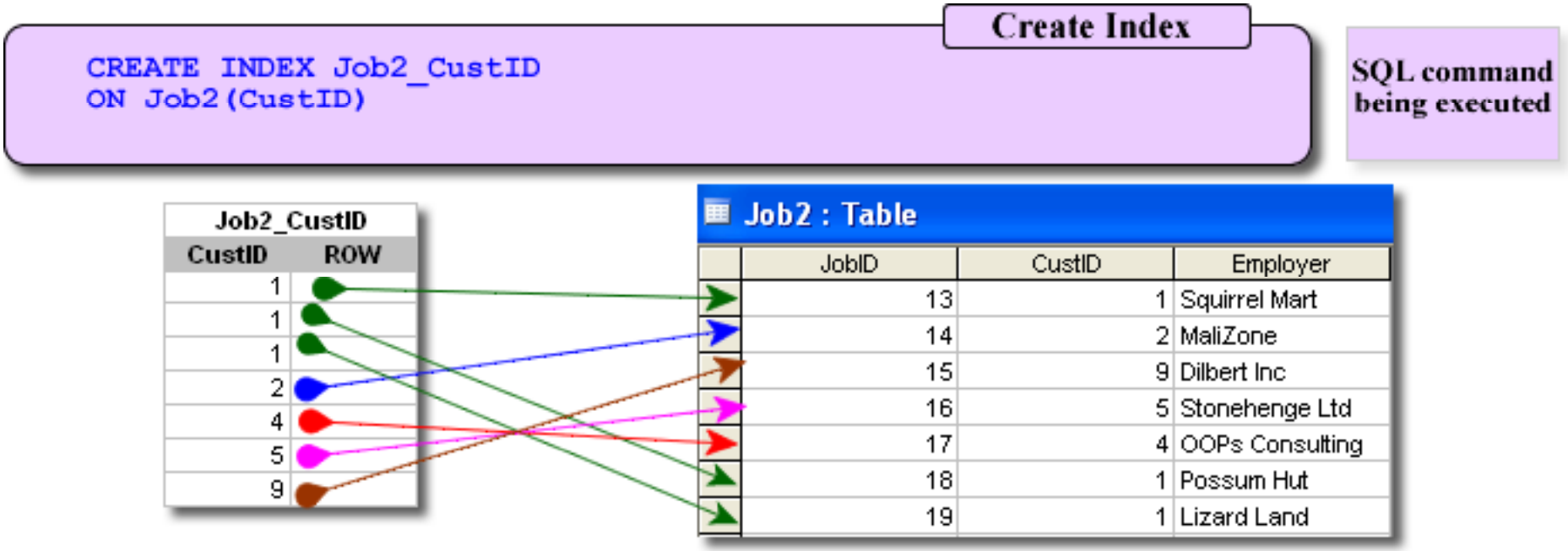
- Indexes GREATLY speed up queries on the fields being indexed
- Indexes SLIGHTLY slow down INSERTS, UPDATES and DELETES
  - The indexes need to be updated anytime a record changes
  - This adds a small bit of overhead to the system
- The increase in query speed usually greatly offsets the slight extra load on INSERTS, UPDATES and DELETES
  - Most database use is typically reading (instead of writing)

Types of Indexes

- Simple Index
  - Indexes the field specified
  - Can have multiple simple indexes
  - Speeds up queries using the indexed field
- Composite Index
  - Indexes combinations of fields
  - Can have multiple composite indexes
  - Speeds up queries using the specified combination of fields

Simple Indexes

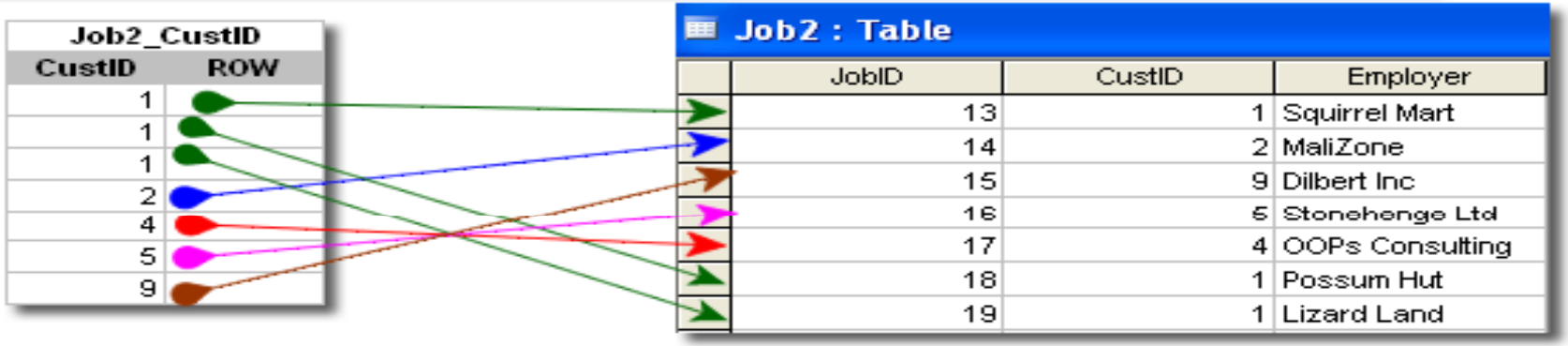
- An Index contains a list of all field values and pointers to the record with that value



How Indexes help SELECTs

Create Index

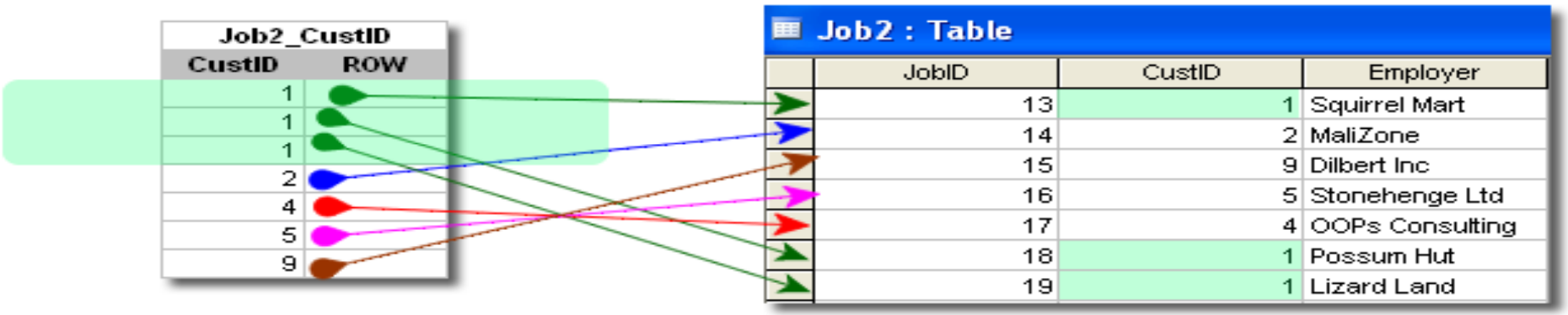
```
CREATE INDEX Job2_CustID
ON Job2(CustID)
```



Select

```
SELECT * FROM Job2
WHERE CustID = 1
```

Indexes are consulted first to see what rows to return.
In this case, the database only needs to read **three index records** to find out which fields in Job2 match.
Without an index, the database would have to read **every record** in the table (called a Table Scan)



Composite Indexes

- Useful only when you will be querying multiple fields simultaneously
- Most selective (resulting in the fewest records matching) fields should be listed first
- Slight performance gain over multiple Simple Indexes when used correctly
- No performance gain (possibly even a performance loss) when used incorrectly.

Composite Index

Create Index

```
CREATE INDEX Job2_CustID
ON Job2(CustID)
```

Products_Mfr_Sku

Manufacturer	SKU	ROW
Sparkleys	A10	
Sparkleys	A11	
B52	A10	
B52	A11	
B52	B11	

Products : Table

ItemID	Manufacturer	SKU	Desc	Price
1	Sparkleys	A10	Sparkler, Red	10
2	Sparkleys	A11	Sparkler, Blue	10
3	B52	A10	Rock Lobster	30
4	B52	B11	Love Shack	500
5	B52	A11	Tin Roof Rusted	25

Select

```
SELECT * FROM Products
WHERE Manufacturer = "B52" AND SKU="B11"
```

Products_Mfr_Sku

Manufacturer	SKU	ROW
Sparkleys	A10	
Sparkleys	A11	
B52	A10	
B52	A11	
B52	B11	

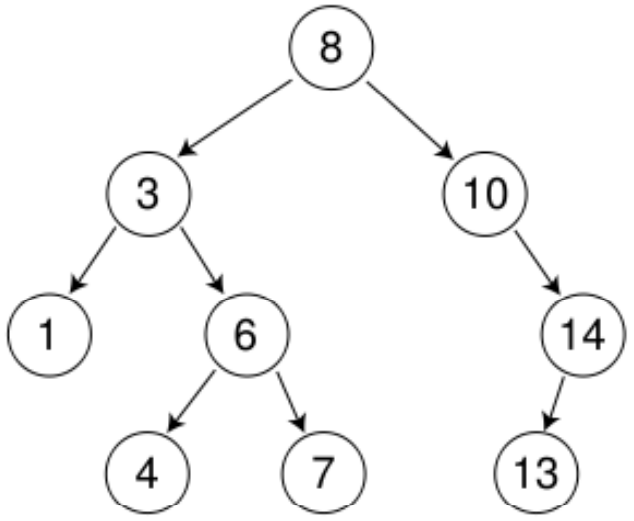
Products : Table

ItemID	Manufacturer	SKU	Desc	Price
1	Sparkleys	A10	Sparkler, Red	10
2	Sparkleys	A11	Sparkler, Blue	10
3	B52	A10	Rock Lobster	30
4	B52	B11	Love Shack	500
5	B52	A11	Tin Roof Rusted	25

Searching Indexes

- There are high-performance algorithms for searching pre-sorted data
- Index data stored internally as binary trees (typically)
- Searching through binary tree much faster than reading through table

1
2
3
4
5
6
7
8
9
10
11
12
13
14



Binary Tree of sorted data.
Start at top. If that is the value you are looking for, stop.
If number you are looking for is lower, look on left side.
If number you are looking for is higher, look on right side.

Design Strategy

Make it work

Make it fast

Make it pretty

Focus on what’s needed

- Make your query as specific as possible
  - Makes joins simpler (and therefore faster)
  - Returns no unwanted data
  - Increases response time
  - Reduces network and server load
- Put as much as possible in the WHERE clause
- Join your fields properly

Maximum Selectivity / Specificity

- If differents parts of the WHERE clause will result in smaller data sets than other ones, list them the ones that will most greatly reduce the data first
- List the others in the same order

Optimizing Queries

Faster is Better

Horizontal Partitioning

- Avoid extremely “wide” records (records with lots of fields)
- Split data into two tables with a common ID field
- All the record data is rarely used in all queries, so it reduces traffic and speeds up processing and disk reads

Add Indexes

- Make sure all fields searched on regularly are indexed

Define Clustered Index

- The most-likely criteria to be sorted on should be defined as your clustered index
- Clustered index defines the order the records will physically be stored on disk

Normalize Data

- No redundant data
- Link related data
- Don’t go crazy

Denormalize Data

- If joining more 4 tables or more in a single query, consider de-normalizing data (combining data from external tables back into the main table)
  - Can return faster query results
  - Risks include having to manage duplicate data in database and larger disk usage

Size records to fit database page size

- SQL Server 7.0, 2000, and 2005 data pages are 8K (8192 bytes) in size.
  - Of the 8192 available bytes in each data page, only 8060 bytes are used to store a row. The rest is used for overhead.
  - If you have a row that was 4031 bytes long, then only one row could fit into a data page, and the remaining 4029 bytes left in the page would be empty
    - Making each record 1 byte shorter would halve the disk access required to read the table

Use a Primary Key

- Make sure you have a primary key defined
- Ideally, it should be a single unique field
  - You can define composite keys, but they are generally not needed if the database is designed properly
- Most tables should have a unique *TableName*ID field
  - This allows you to identify any row by a single unique value

Use an IDENTITY Column

- If there is:
  - no Primary Key on the table
  - no unique index on the table
- Then
  - Add an IDENTITY (unique value) column to the table
  - Optionally, index the IDENTITY column if you will query it regularly

Move **TEXT**, **NTEXT**, and **IMAGE** data into table

- These types normally stored outside the table (uses a pointer to the data in the field)
- If these types will be searched frequently, consider moving them into the database table's storage with:

```
sp_tableoption 'tablename', 'text in row', 'on'
```

or

```
sp_tableoption 'tablename', 'text in row', 'size'
```

Consider Replication in advance

- If Replication is to be used, factor the decision into the original design of the database

Use Built-in Referential Integrity

- Use foreign keys and validation constraints built into the database
- Don’t manage it in the application

- Benefits:
  - Faster execution
  - Can’t mess it up with application errors

Re-test when changing servers

- Retest and rebenchmark applications when moving to a new server, such as:
  - Development
  - Staging
  - Production
- Problems often caused by:
  - More rows in test data on servers “closer” to production
  - Server configurations different
  - Tables not indexed the same way

Constraints are Fast

- Constraints on fields are faster than
  - Triggers
  - Rules
  - Defaults

Don’t duplicate effort

- Don’t check for the same thing twice
  - (duh, but it happens)
  - Don’t use a trigger and a constraint to do the same thing
  - Same for constraints and defaults
  - Same for constraints and rules

Limit records to be JOINed

- Use WHERE clause to minimize rows to be JOINed.
  - Particularly in the OUTER table of an OUTER JOIN

Index Foreign Keys

- Fields in a table that are a foreign key are not indexed automatically just because they are a foreign key.
- Add these indexes manually

Minimize Duplicate JOIN field data

- JOINs are slower when there are few different keys in the joining table

JOIN on unique indexed fields

- JOINs will perform fastest when joining indexed fields with no duplicate data

JOIN Numeric Fields

- JOINs on numeric fields perform much faster than JOINs on other datatype fields.

JOIN the exact same datatypes

- JOINs should be to the exact same datatype for best performance
  - Same type of field
  - Same field length
  - Same encoding (ASCII vs. Unicode, etc)

Use ANSI JOIN syntax

- Improves readability
- Less likely to cause programmer errors
- No Aliases:

```
SELECT fname, lname, department
  FROM names INNER JOIN departments ON
  names.employeeid = departments.employeeid
```

- Microsoft syntax example:

```
SELECT fname, lname, department
  FROM names, departments
  WHERE names.employeeid = departments.employeeid
```

- Code is more portable between databases

Use Table Aliases

- Shortens code
- Makes it easier to follow, especially with long queries
- Identifies which table each field is coming from
- No table aliases:

```
SELECT fname, lname, department
  FROM names INNER JOIN departments ON
  names.employeeid = departments.employeeid
```

- Microsoft syntax example:

```
SELECT N.fname, N.lname, D.department
  FROM names N INNER JOIN departments D ON
  N.employeeid = D.employeeid
```

Don’t use SELECT *

- Requires additional parsing on the server to extract field names
- Returns unneccesary data (unless you are actually using *every* field)
- Returns duplicate data on JOINs (do you really need *two* copies of the RecordID field?)
- Can cause errors (*in some databases*) if JOINed tables have fields with the same name

Store in separate files in filegroup

- For very large joins placing tables to be joined in separate physical files within the same filegroup can improve performance
- SQL Server can spawn separate threads for processing each file

Don’t use CROSS JOINs

- Unless actually needed (rarely) do not use a CROSS JOIN (returns all combinations of records on both sides of JOIN)
- People sometimes will do a CROSS JOIN and then use DISTINCT and GROUP BY to eliminate all the duplication
  - Don’t do that.

JOINs vs. Subquery

- Depending on the specifics, either could be faster. Write both and test performance to be sure which is best.
- JOIN:

```
SELECT a.*
  FROM Table1 a INNER JOIN Table2 b
  ON a.Table1ID = b.Table1ID
  WHERE b.Table2ID = 'SomeValue'
```

- Subquery:

```
SELECT a.* FROM Table1 a WHERE Table1ID IN
  (SELECT Table1ID FROM Table2 WHERE Table2ID =
  'SomeValue')
```

Avoid Subqueries unless needed

- Avoid subqueries unless actually required
- Most subqueries can be expressed as JOINs
- Subqueries are generally harder to read and understand (for humans) and, therefore, maintain

Use an Indexed View (Enterprise 2000 and later only)

- An Indexed View maintains an updated record of how the tables are joined via a clustered index
- This slows INSERTs, UPDATEs and DELETEs a bit, so consider the tradeoff for the faster queries

Use Database's Performance Optimization Tools

- Most major databases have methods for monitoring and optimizing database performance
- Google "*databaseName* optimizing" for tons of links

Avoid DISTINCT when possible

- DISTINCT clauses are frequently (and usually unintentionally) used to hide an incorrect JOIN
- Properly normalized database will not frequently need DISTINCT clauses
- Look for incorrect JOINs or create more explicit WHERE clauses to avoid needing DISTINCT
- Of course, use it when appropriate

The End

(for now)

Google: Query Optimization

for lots more information