CS 61:
Database Systems

Query optimization

Adapted from Silberschatz, Korth, and Sundarshan unless otherwise noted

Agenda

1. Query processing
2. Tips for fast queries
3. Explain (yourself)

Three typical non-database bottlenecks to performance: CPU, Ram, network I/O

CPU: as fast as possible

RAM: as much as possible

- Cache queries and data in memory
- Less query processing and paging to disk

Network:

- You don't want this

Three typical non-database bottlenecks to performance: CPU, Ram, network I/O

CPU: as fast as possible

RAM: as much as possible

- Cache queries and data in memory
- Less query processing and paging to disk

Network:

- Fast network
- Fast disk (SAN)

The query optimizer chooses the best execution plan for a given query

High-level overview of SQL execution process

Parse
- Check syntax
- Check table and columns exist

Compile
- Convert query to machine code

Optimize
- Choose optimal execution plan

Cache
- Store optimized query plan in cache
- If command submitted again, skip prior steps (already done)

Replace placeholders
- Prepared statement are not complete statements
- Have placeholders for some values
- But, format of command is set now
- Placeholders filled with literal values
- Place holder data doesn't change command format

Execute
- Query is executed
- Data is fetched from disk and returned to user

Adapted from: http://javabypatel.blogspot.com/2015/09/how-prepared-statement-in-java-prevents-sql-injection.html

The query optimizer chooses the best execution plan for a given query

High-level overview of SQL execution process

Parse
- Check syntax
- Check table and columns exist

Compile
- Convert query to machine code

Optimize
- Choose optimal execution plan

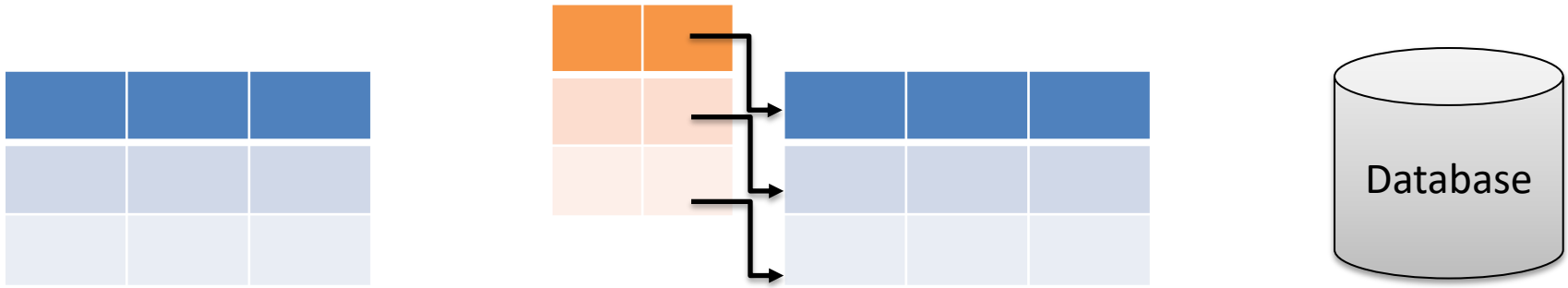
Cache
- Store optimized query plan in cache
- If command submitted again, skip prior steps (already done)

Replace placeholders
- Prepared statement are not complete statements
- Have placeholders for some values
- But, format of command is set now
- Placeholders filled with literal values
- Place holder data doesn't change command format

Execute
- Query is executed
- Data is fetched from disk and returned to user

Adapted from: http://javabypatel.blogspot.com/2015/09/how-prepared-statement-in-java-prevents-sql-injection.html

The database keep statistics to help the optimizer make smart decisions



Tables

- Number of rows/disk blocks used
- Number of columns in each row
- Min/Max value in each column
- Which columns have indexes

Indexes

- Number and name of columns in the index key
- Number of distinct key values in the index key
- Histogram of key values in an index
- Number of disk blocks used by the index

Environment

- Logical and physical disk block size
- Location and size of data files
- CPU speed
- Disk throughput speed
- RAM available

Adapted from Coronel and Morris

Optimizer approaches: rule-based and cost-based

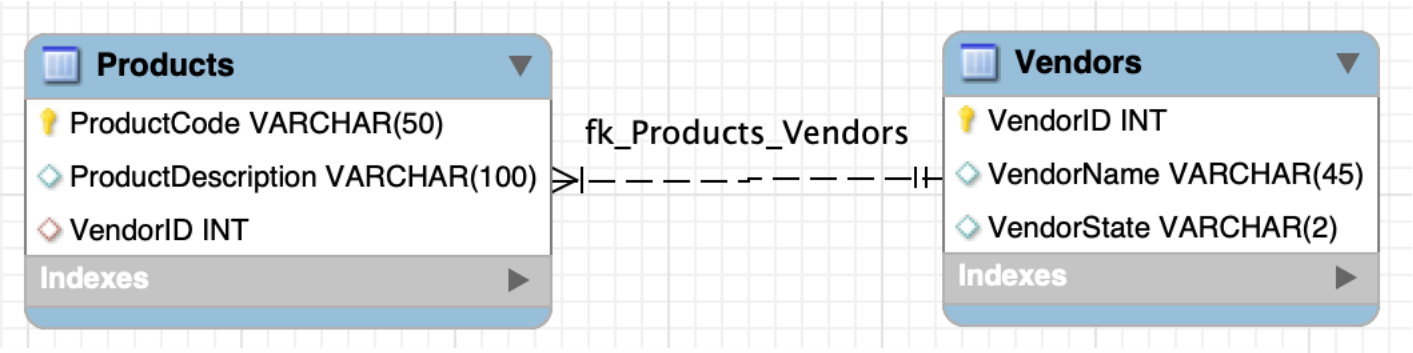
Rule-based optimizer:

- Uses preset rules and cost points to determine the best approach to execute a query
- Rules assign a fixed cost to each SQL operation

Cost-based optimizer:

- Uses algorithms based on statistics about objects being accessed to determine the best approach to execute a query
- Adds up the total SQL operation cost
  - I/O costs
  - Processing costs
  - Resource costs (RAM and temporary space)

Cost-based example: multiple ways to execute the same query




```
3     -- Find products produced by vendors in New Hampshire
4     SELECT ProductCode, ProductDescription, VendorName, VendorState
5     FROM Products p, Vendors v
6     WHERE p.VendorID = v.VendorID
7     AND v.VendorState = 'NH';
```

Want data from both tables, so will require a JOIN

Optimizer	Products table	Vendors table
Knows	7,000 rows	300 rows
Estimates	NH products: 1,000	NH vendors: 10

Optimizer tries to determine best way to execute query

- Book gives detailed analysis of cost
- I will focus on I/O operations
- Two ways this query could be executed

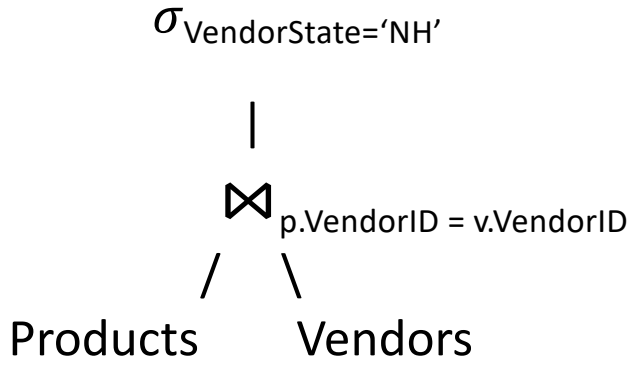
Adapted from Coronel and Morris

Two options to execute the query

Two options

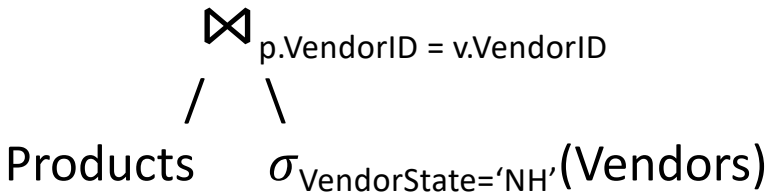
```
3     -- Find products produced by vendors in New Hampshire
4     SELECT ProductCode, ProductDescription, VendorName, VendorState
5     FROM Products p, Vendors v
6     WHERE p.VendorID = v.VendorID
7     AND v.VendorState = 'NH';
```

1) JOIN first, then SELECT NH



Optimizer must choose which approach is better

2) SELECT NH first, then JOIN



Adapted from Coronel and Morris

Option 1: JOIN first, then SELECT

1) JOIN first, then SELECT NH

```
3     -- Find products produced by vendors in New Hampshire
4 •   SELECT ProductCode, ProductDescription, VendorName, VendorState
5     FROM Products p, Vendors v
6     WHERE p.VendorID = v.VendorID
7     AND v.VendorState = 'NH';
```

Products: 7,000 rows
NH products: 1,000

Vendors: 300 rows
NH vendors: 10

$\sigma_{\text{VendorState='NH'}}(\sigma_{\text{p.VendorID = v.VendorID}}(\text{Products X Vendors}))$

Step	Operations	Read I/O Ops	Write I/O Ops	Total I/O Ops
1	Cartesian Product (Product x Vendor)	7,000 + 300 = 7,300	2,100,000	2,107,300
2	Select rows from Step 1 with same vendor codes	2,100,000	7,000	2,107,000
3	Select rows from Step 2 with State = NH	7,000	1,000	8,000
	Total	2,114,300	2,108,000	**4,222,300**

Remember from Relational Algebra, a JOIN is a Cartesian Product followed by a SELECT

Adapted from Coronel and Morris

Option 2: SELECT first, then JOIN

2) SELECT NH first, then JOIN

```
3     -- Find products produced by vendors in New Hampshire
4 •   SELECT ProductCode, ProductDescription, VendorName, VendorState
5     FROM Products p, Vendors v
6     WHERE p.VendorID = v.VendorID
7     AND v.VendorState = 'NH';
```

This example considers only I/O cost, the book is more precise

Products: 7,000 rows
NH products: 1,000

Vendors: 300 rows
NH vendors: 10

$\sigma_{p.VendorID = v.VendorID}(Products \times \sigma_{VendorState='NH'}(Vendors))$

Step	Operations	Read I/O Ops	Write I/O Ops	Total I/O Ops
1	Select rows in Vendor with State = 'NH'	300	10	310
2	Cartesian product Products x Step 1	7,000 + 10 = 7,010	70,000	77,010
3	Select rows in Step 2 with same vendor codes	70,000	1,000	71,000
	Total	77,310	71,010	**148,320**

Option 1: 4,222,300

Option 2: 148,320

Optimizer picks Option 2 as execution plan (28 times smaller)

Adapted from Coronel and Morris

Agenda

1. Query processing
2. Tips for fast queries
3. Explain (yourself)

Majority of performance problems are related to poorly written SQL code

A carefully written query almost always outperforms a poorly written query

- When possible, use simple columns or literals as operands; try to avoid using conditional expressions with functions
- Numeric field comparisons are faster than character, date, and NULL comparisons
- Equality comparisons are faster than inequality comparisons
- When using multiple AND conditions, write the condition most likely to be false first (take advantage of short circuiting)
- When using multiple OR conditions, write the condition most likely to be true first (short circuiting again)
- Avoid the use of NOT logical operator (NOT Price>10 becomes Price <= 10)
- For text matching, use 'A%' not '%A%' if possible
- Consider your index use!

Adapted from Coronel and Morris

Consider your index use

Index considerations

Indices speed up reads, but slow down writes

- Reads need only scan rows meeting criteria, not full table scan
- Writes must update tables as well as (possibly) index

Impractical to put index on every attribute

- Take up too much memory
- Performance hit

Considerations for indices:

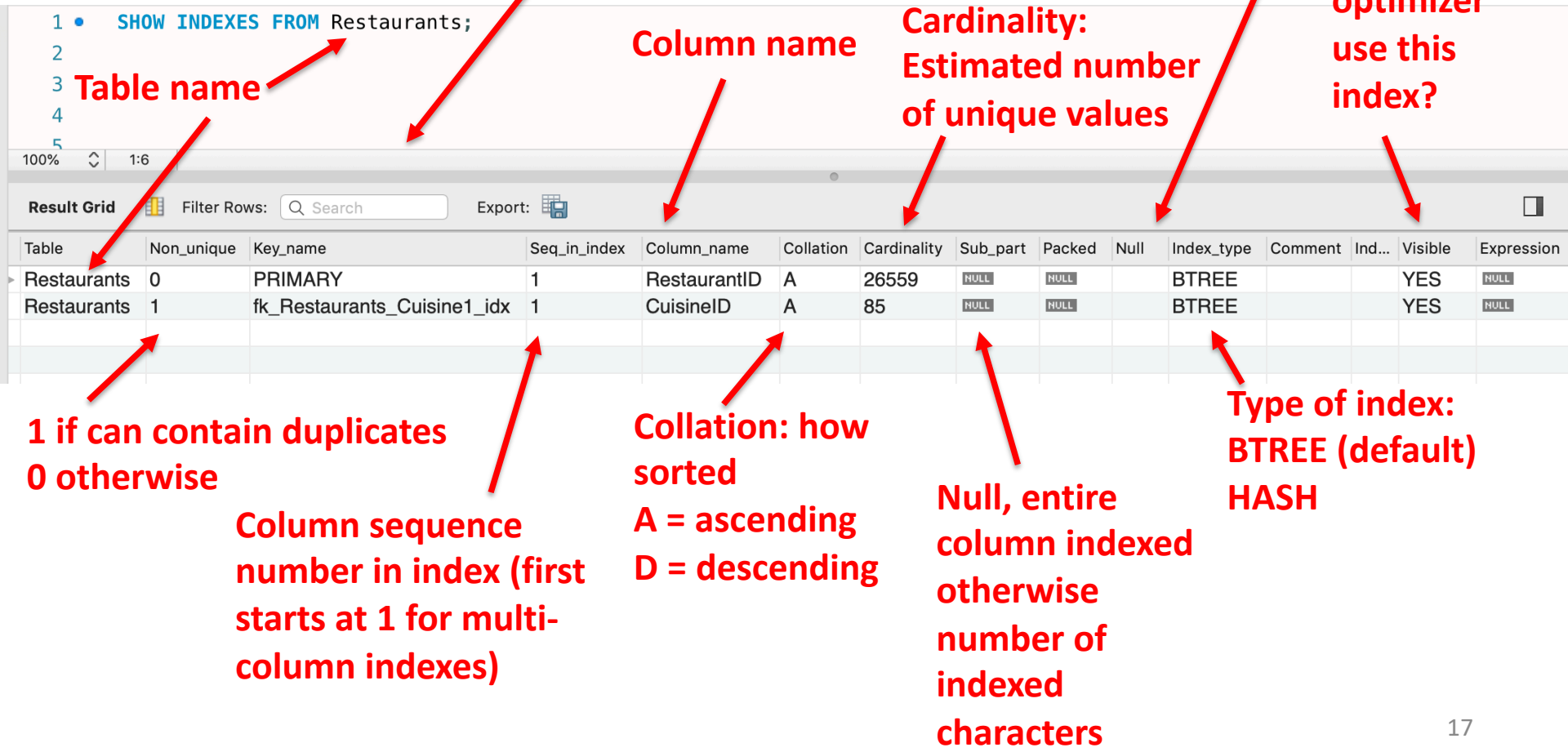
- Use when attribute used in WHERE, HAVING, ORDER BY, or GROUP BY clauses of frequently run queries
- Do not use on small tables
- Do not use with low cardinality (small number of unique values)
- Declare PK and FK so optimizer can use indexes on JOINs (automatically done by MySQL)
- Declare indices for non-prime attributes used in JOINs
- Drop infrequently used indices

Agenda

1. Query processing
2. Tips for fast queries
3. Explain (yourself)

Often indexes can increase SQL read performance significantly

Show indexes



Adapted from: https://dev.mysql.com/doc/mysql-infoschema-excerpt/5.7/en/statistics-table.html

Indices can be created on multiple attributes

Optimizer can use index on left most prefix

- **Can use on Boro**
- **Can use on Boro and ZipCode**
- **Cannot use on just ZipCode (left most not met)**

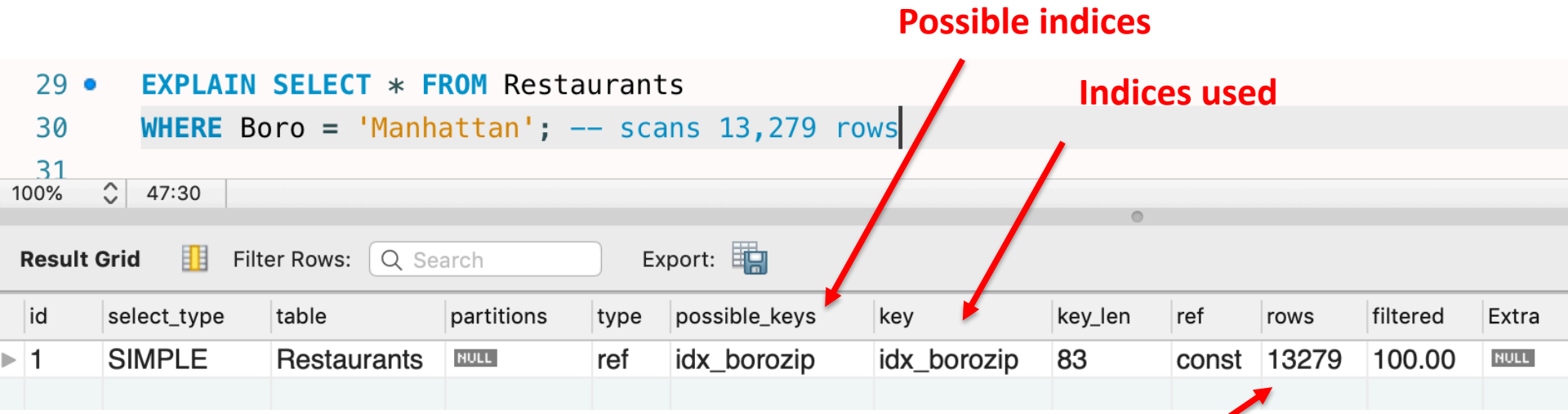
```
16    -- indicies can be created on multiple columns
17    -- called a composite index
18 •  CREATE INDEX idx_borozip ON Restaurants(Boro,ZipCode);
19 •  SHOW INDEXES FROM Restaurants;
20
```

100% 26:19

Result Grid Filter Rows: Search Export:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Ind...	Visible	Expression
Restaurants	0	PRIMARY	1	RestaurantID	A	26559	NULL	NULL		BTREE			YES	NULL
Restaurants	1	fk_Restaurants_Cuisine1_idx	1	CuisineID	A	85	NULL	NULL		BTREE			YES	NULL
Restaurants	1	idx_borozip	1	Boro	A	6	NULL	NULL	YES	BTREE			YES	NULL
Restaurants	1	idx_borozip	2	ZipCode	A	233	NULL	NULL	YES	BTREE			YES	NULL

EXPLAIN tells you how MySQL is using indices



There are 26,573 rows in Restaurants table

Using index, execution plan only estimates scanning 13,279 rows; does not do a full table scan

But there are only 10,649 rows in Manhattan

MySQL uses estimates from table statistics to guess how many rows it will need to process

EXPLAIN tells you how MySQL is using indices

```
29 •  EXPLAIN SELECT * FROM Restaurants
30    WHERE Boro = 'Manhattan'; -- scans 13,279 rows
31
32 •  EXPLAIN SELECT * FROM Restaurants
33    WHERE ZipCode = '10023'; -- scans 26,573 rows
```

100% 1:36

Result Grid Filter Rows: Search Export:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	Restaurants	NULL	ALL	NULL	NULL	NULL	NULL	26559	10.00	Using where

Full table scan if only use ZipCode

ZipCode is the second index, not part of the left most

Remember unique rows (the Cardinality) is MySQL's estimate, may not be exact

Can use ANALYZE TABLE <name> to get updated key distribution and cardinality statistics from random sample (just an estimate, not an exact count)

Optimizer may use selectivity and cardinality to determine where to use index on JOIN operations

EXPLAIN tells you how MySQL is using indices

```
29 •  EXPLAIN SELECT * FROM Restaurants
30    WHERE Boro = 'Manhattan'; -- scans 13,279 rows
31
32 •  EXPLAIN SELECT * FROM Restaurants
33    WHERE ZipCode = '10023'; -- scans 26,573 rows
34
35 •  EXPLAIN SELECT * FROM Restaurants
36    WHERE Boro = 'Manhattan' AND ZipCode = '10023'; -- scans 201 rows
```

100% 1:39

Result Grid Filter Rows: Search Export:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	Restaurants	NULL	ref	idx_borozip	idx_borozip	88	const,const	201	100.00	NULL

Using both index attributes scans of only 201 rows

Can suggest (or force) use of index, even if optimizer chooses otherwise

```
38    -- drop existing index, create a new one based on first 3 characters
39    -- of restaurant name
40 •  DROP INDEX idx_borozip ON Restaurants;
41 •  CREATE INDEX idx_name ON Restaurants(RestaurantName(3));
42    -- SHOW INDEXES FROM Restaurants;
43    -- can hint (or force) query to using index
44 •  EXPLAIN SELECT RestaurantName
45    FROM Restaurants USE INDEX(idx_name) -- can replace USE with FORCE
46    WHERE RestaurantName like 'Tim%';
47
```

Create index based on first three characters of restaurant name

Suggest (with USE) or require (with FORCE) use of index

100% 1:49

Result Grid | Filter Rows: Search | Export:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	Restaurants	NULL	range	idx_name	idx_name	15	NULL	18	100.00	Using where

Only needs to scan 18 rows (not 26,573 rows!)

Explain also shows how multiple tables are accessed in a JOIN

```
116 •  use nyc_inspections;
117 •  explain SELECT RestaurantID, RestaurantName, InspectionDate, Score
118    FROM Inspections JOIN Restaurants USING (RestaurantID)
119    WHERE RestaurantID = 30075445;
120
```

100% 1:122

Result Grid Filter Rows: Search Export:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows
1	SIMPLE	Restaurants	NULL	const	PRIMARY	PRIMARY	4	const	1
1	SIMPLE	Inspections	NULL	ref	fk_Inspections_Restaurants_idx	fk_Inspections_Restaurants_idx	4	const	6

Number of rows read

SIMPLE – no subqueries or UNIONs
PRIMARY – outermost in JOIN
DERIVED – part of subquery within FROM
SUBQUERY – first SELECT in subquery
…
Others, see MySQL documentation

- **const – table has only one matching indexed row**
- **ref – all matching rows of indexed column are read for each combination of rows from previous table**
- **all – table scan!**

…
Others, see MySQL documentation

More info at https://www.sitepoint.com/using-explain-to-write-better-mysql-queries/

Practice: Indices

Download customers_schema.sql from course web page

- Take at the customers table and the fields it contains
- List the indices on this table

Try running the following command:

```
SELECT * FROM Customers
WHERE ContactFirstName like 'A%'
OR ContactLastName LIKE 'A%';
```

Answer these questions:

What does this command do?

What indices does it use?

Try suggesting the query use the composite indices

How do the execution times compare with and without your suggestion

Adaptec from https://www.mysqltutorial.org/mysql-index/mysql-use-index/