CS 61:
Database Systems

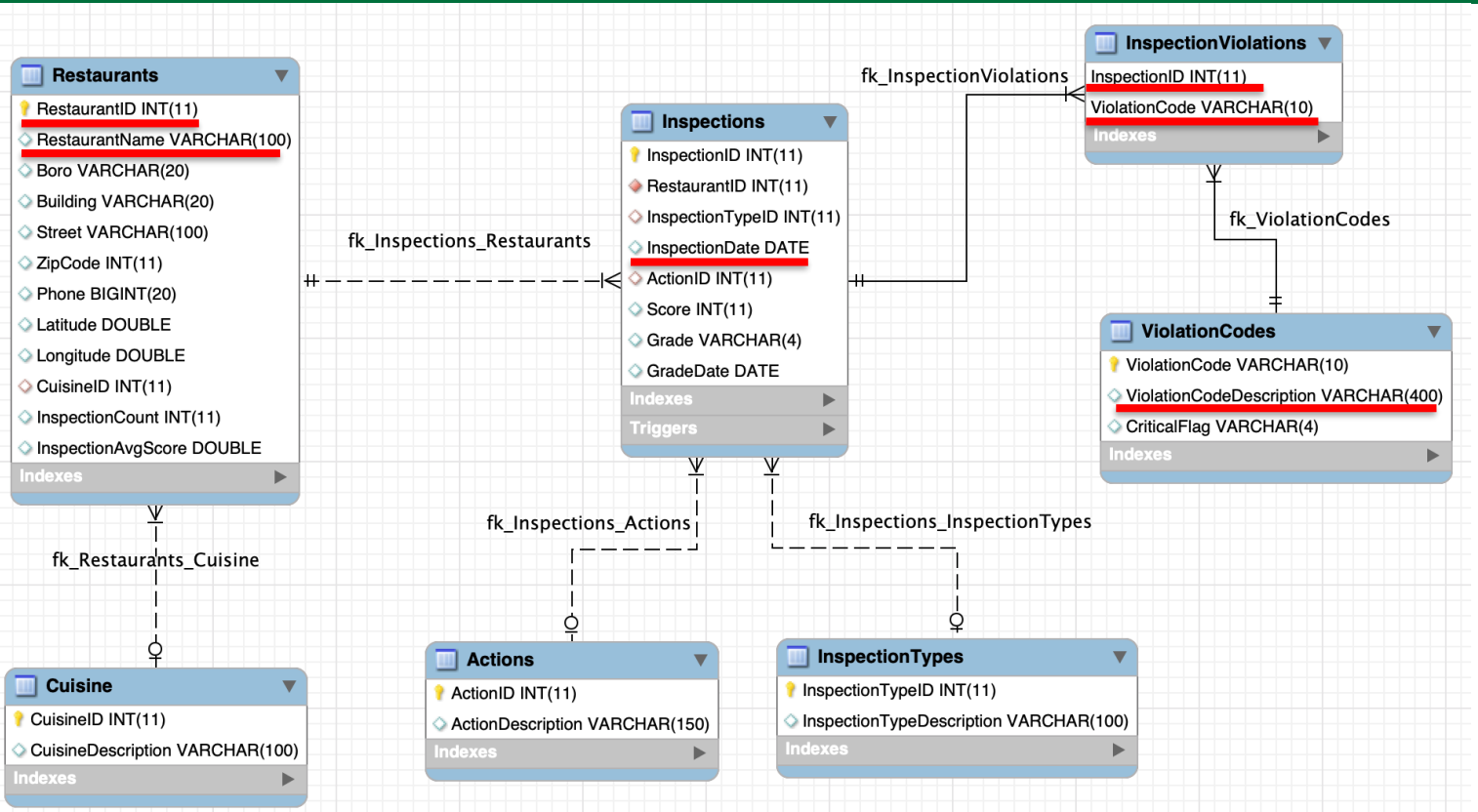
Query optimization

Adapted from Silberschatz, Korth, and Sundarshan unless otherwise noted
Augmented by Claude Opus

Agenda

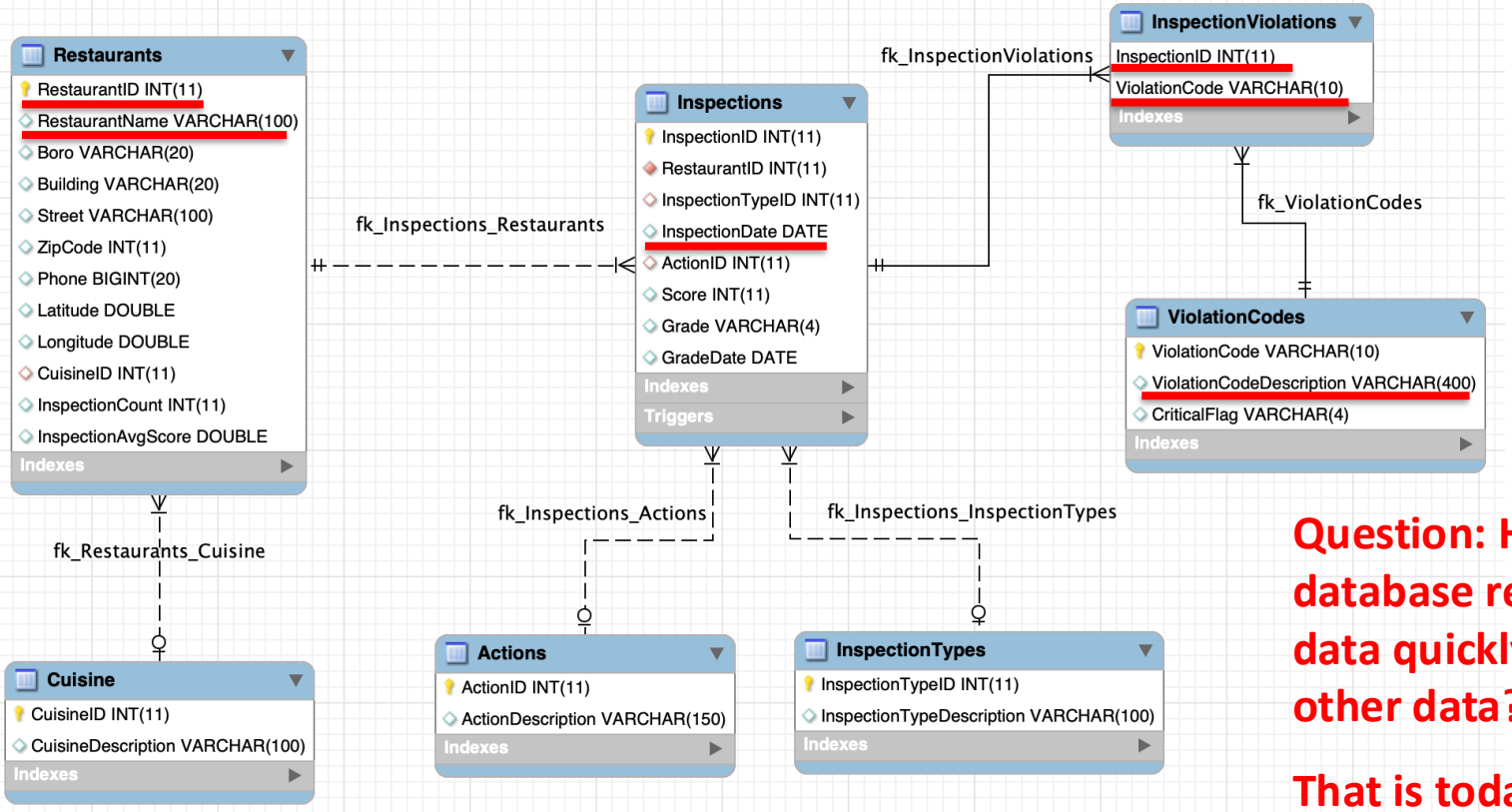
1. Query processing
2. Explain (yourself)
3. Tips for fast queries

Normalized databases generally means more tables which means more joins




```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```

Normalized databases generally means more tables which means more joins



Question: How does the database return the desired data quickly among all the other data?

That is today's topic

Restau...	RestaurantName	InspectionDa...	ViolationCodeDescription
30075445	MORRIS PARK BAKE SHOP	2023-02-03	Cold food item held above 41° F (smoked fish a...
30075445	MORRIS PARK BAKE SHOP	2023-02-03	Non-food contact surface improperly constructe...
30075445	MORRIS PARK BAKE SHOP	2023-08-22	Evidence of mice or live mice present in facility'...
30075445	MORRIS PARK BAKE SHOP	2023-08-22	Establishment is not free of harborage or conditi...
30075445	MORRIS PARK BAKE SHOP	2023-08-22	Pesticide not properly labeled or used by unlice...
30075445	MORRIS PARK BAKE SHOP	2024-11-08	Food, supplies, and equipment not protected fro...
30075445	MORRIS PARK BAKE SHOP	2024-11-08	Pesticide not properly labeled or used by unlice...
30075445	MORRIS PARK BAKE SHOP	2024-11-08	Non-food contact surface improperly constructe...

3 inspections
8 violations

Goal is fast queries; we trust the database to determine how to run queries quickly

Input: SQL query — describes *what* you want

Output: Physical execution plan — describes *how* to compute result

Goal: Minimize cost (typically I/O + CPU time)

Method:

- Search the space of equivalent plans
- Estimate cost of each
- Pick the best

We trust the database to answer query quickly

This trust is what makes the optimizer one of the most important, and most complex, part of any DBMS

Key insight: SQL doesn't say *how* — that's the optimizer's job

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

CPU: as fast as possible

RAM: as much as possible
Why?

- 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
Why?

- 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

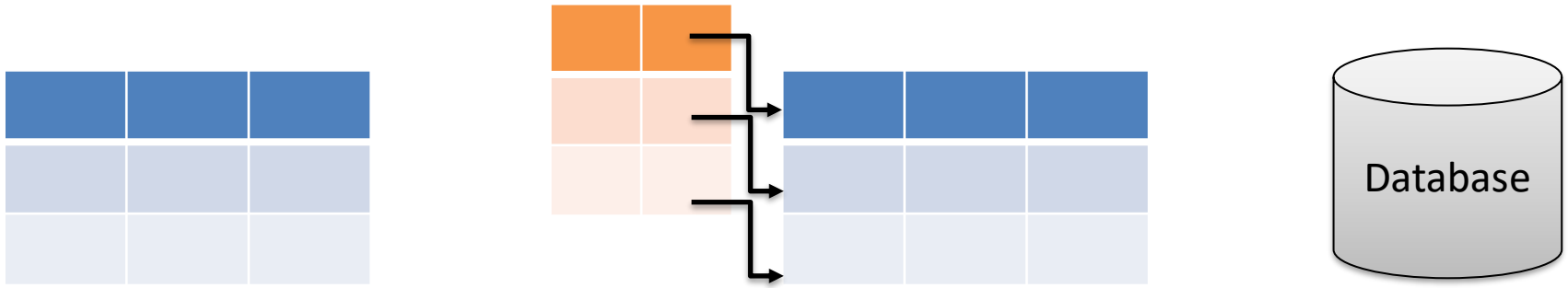
Cache
- Store optimized query plan in cache
- If command submitted again, skip prior steps (already done)

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

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

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

Optimizer approaches: Logical optimization and physical optimization

Logical optimizer:

- Transform SQL command into query tree using algebraic equivalence rules
- "This tree means the same things as that tree, but is simpler/cheaper"
- Rule-based, mostly deterministic

Physical 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)

Relational algebra refresher

Operator	Symbol	SQL
Selection	$\sigma_{predicate}(R)$	WHERE *predicate*
Projection	$\pi_{cols}(R)$	SELECT *cols*
Join	$R \bowtie_{\theta} S$	JOIN … ON θ
Cartesian product	R X S	FROM R, S
Group/Aggregate	γ	GROUP BY
Sort	τ	ORDER BY

On joins, database do not compute the cartesian product (unless they must)

A join ($\bowtie$) is logically a cartesian product (X) followed by a select (σ)

$R \bowtie_\theta S \equiv \sigma_\theta(R \times S)$

But computing a cartesian product would be expensive indeed!

Databases do not compute a cartesian product, then filter, unless they must

Nested Loop Join

```
for each row r in R (outer):
   for each row s in S (inner):
      if r.x = s.x:
         output (r, s)
```

Cost

|R| × |S| evaluations + scan costs

Use this if you do not have a better option (or both tables as tiny)

Index Nested Loop Join

```
for each row r in R (outer):
   use index on S.x to find matches
   for each r.x = s.x:
      output (r, s)
```

Cost

|R| × (index lookup cost on S)

Requires

Index on join attribute in S

Hash Join

```
build: for each row r in R:
                  hashtable.add(r.x)
probe:  for each row s in S:
          for each match hashtable[s.x]:
            output (r, s)
```

Cost

R| + |S| (one pass over each table)

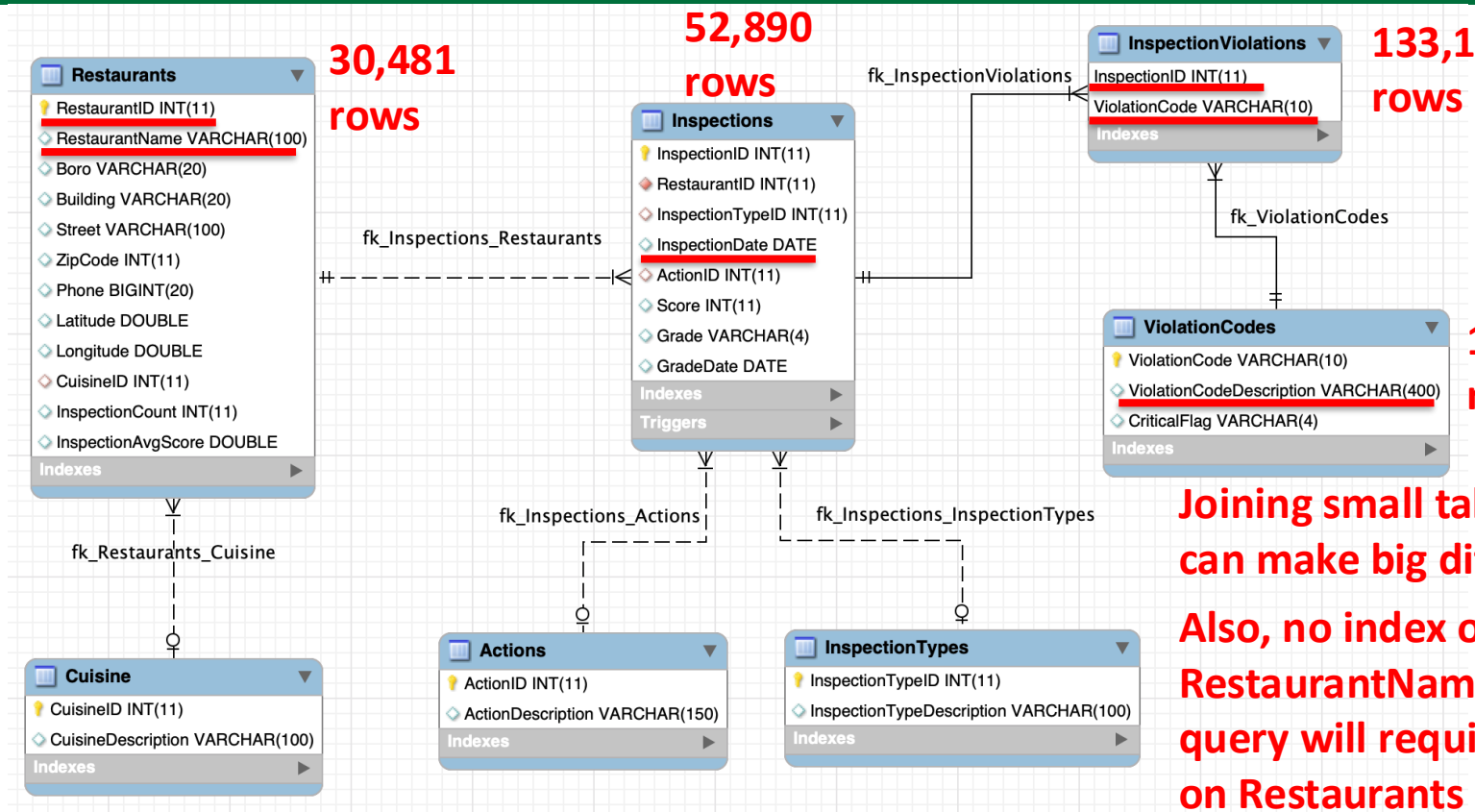
Requires

Assume R smaller than S (reverse otherwise and build on S, probe on R)

Equi-join (works only on =)

CROSS JOIN requires a cartesian product

Our example query



Joining small tables first vs. last can make big difference

Also, no index on RestaurantName means this query will require a table scan on Restaurants

```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```

This query could go wrong if not properly analyzed

Plan A (naïve)

- Join all 4 tables together
- Cartesian product would produce around 3.2 x 10^{16} rows
- Filter down to 133,186 rows afterward

Slightly smarter
Use joins and avoid full cartesian products

Estimate hundreds of thousands of row operations (plus a giant sort)

Plan B (smart)

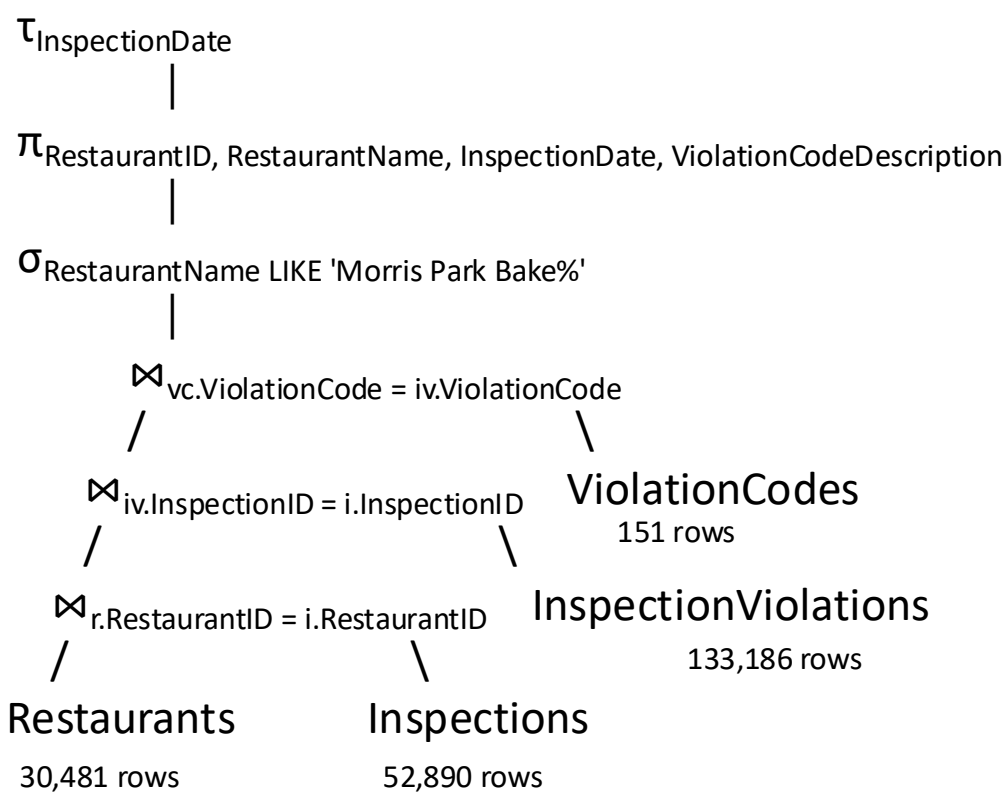
- Filter Restaurants by name first (find maybe ~3 matches),
- Then walk outward through indexes to find their inspections, violations, and descriptions

Estimate a few hundred row operations

Both return identical results. The optimizer's job is to pick B

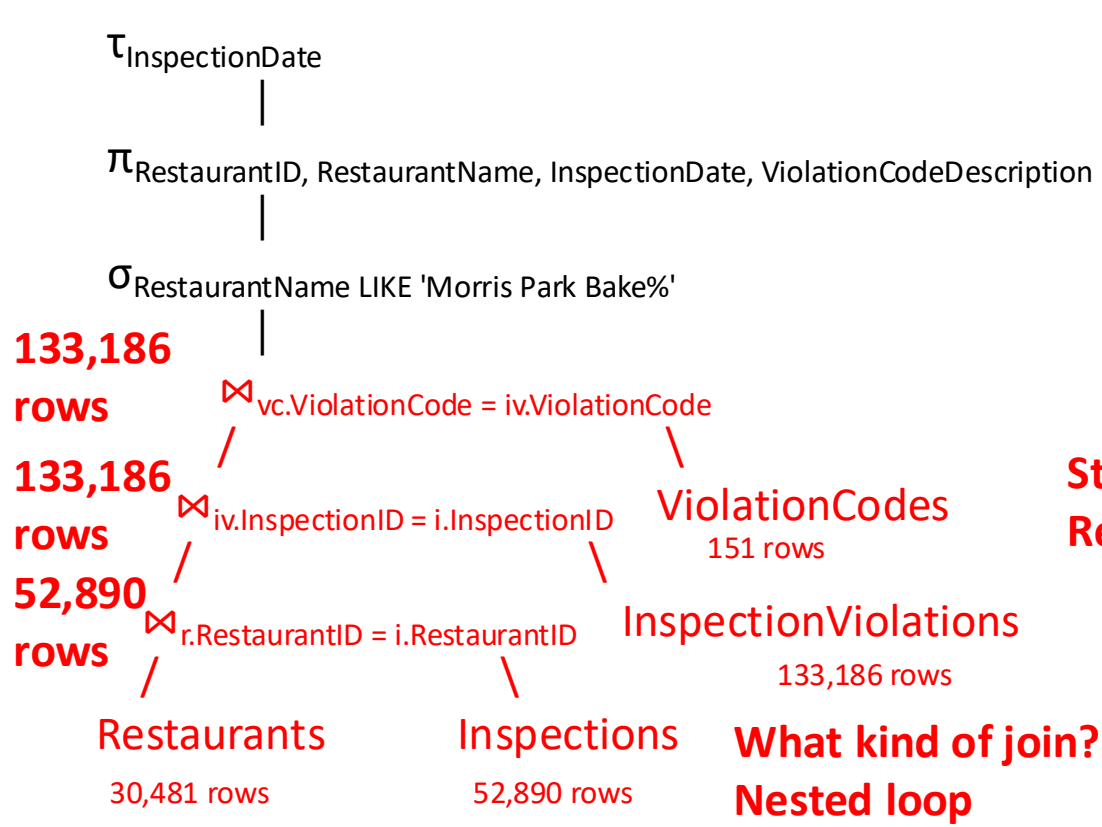
The naïve translation from SQL just follows commands as written

```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```



The naïve translation from SQL just follows commands as written

```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```



What kind of join?
Nested loop

Step 1-3: join all four tables
Result: 133,186 rows

The naïve translation from SQL just follows commands as written

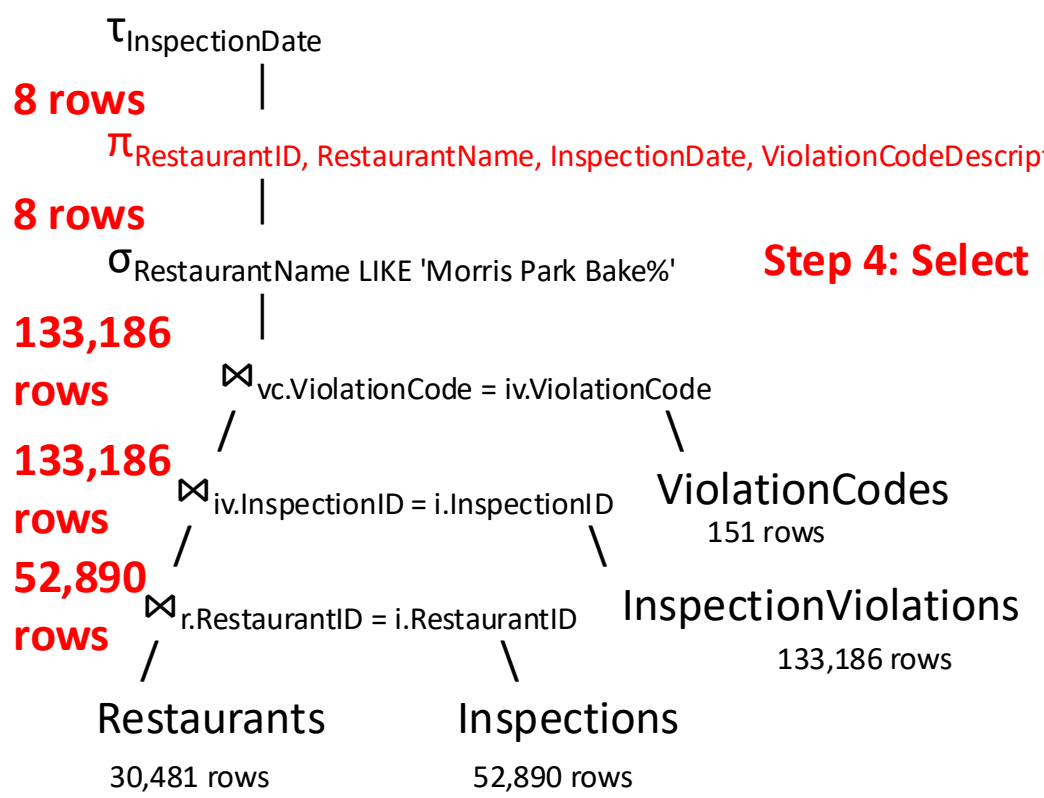
```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```

Step 4: Select rows

Step 1-3: join all four tables
Result: 133,186 rows

The naïve translation from SQL just follows commands as written

```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```



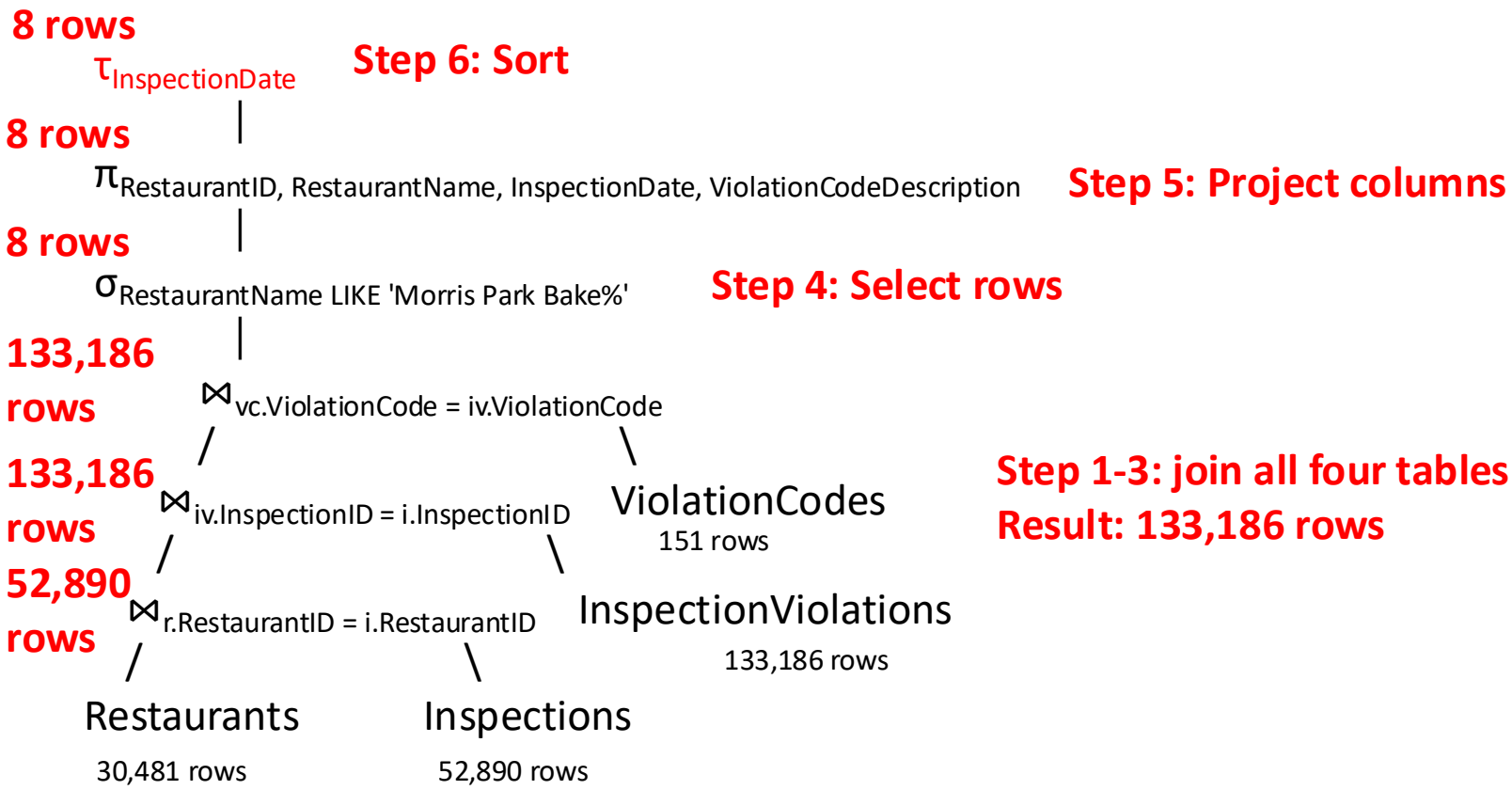
Step 5: Project columns

Step 4: Select rows

Step 1-3: join all four tables
Result: 133,186 rows

The naïve translation from SQL just follows commands as written

```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```



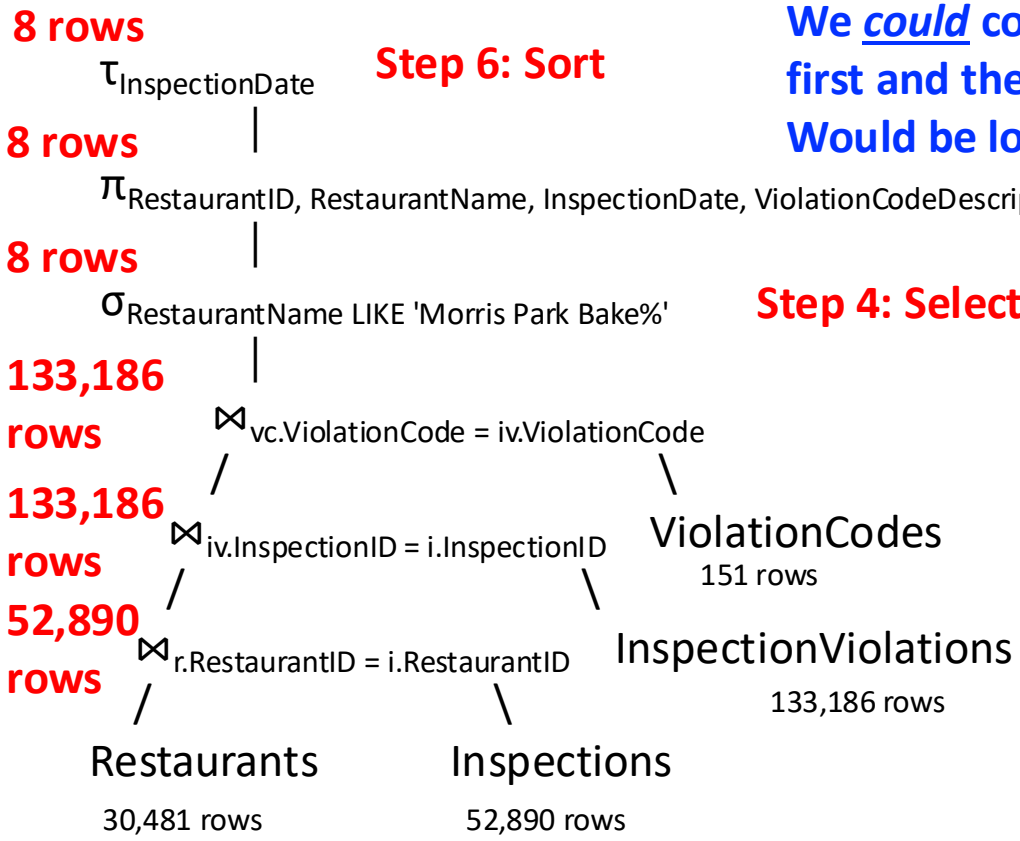
The naïve translation from SQL just follows commands as written

```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
 ORDER BY InspectionDate;
```

Filter at Step 4 sits *above* all three joins

We *could* compute a giant 4-way cartesian product first and then filter

Would be logically correct, but catastrophically slow



Step 1-3: join all four tables

Result: 133,186 rows

The optimizer's job is to transform this tree into something more efficient

The naïve translation from SQL just follows commands as written

Assume database

1. Reads needed tables
2. Processes rows
3. Writes result back to temporary table

We can do better!

Estimated I/O operations

Step	Op	Read I/O	Write I/O	Total
1	Rest ⋈ Insp	83,371	52,890	136,261
2	1 ⋈ InspViol	186,076	133,186	319,262
3	2 ⋈ ViolCode	133,337	133,186	266,523
4	Select	133,186	8	133,202
5	Project	8	8	16
6	Sort	8	8	16
	Total	**535,986**	**319,286**	**855,272**

These are I/O operations only, does not include row processing which can be significant!

We will see soon, sometimes row processing takes more time than I/O

Equivalence Rule 1: Do selection early (selection push down)

```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```

This is *the* canonical query optimization
Every row you avoid early is a row you skip in every subsequent operation

Rule: If a selection only mentions columns from one side of a join, push it down:

$\sigma_p(R \bowtie S) \equiv \sigma_p(R) \bowtie S$ *(when p only references R)*

For our query:

`WHERE RestaurantName LIKE 'Morris Park Bake%'` only references `Restaurants`

Before:

$\sigma_{\text{name LIKE ...}}$(Restaurants ⋈ Inspections ⋈ InspectionViolations ⋈ ViolationCodes)

After:

$\sigma_{\text{name LIKE ...}}$(Restaurants) ⋈ Inspections ⋈ InspectionViolations ⋈ ViolationCodes

Why it helps: Filter 30,481 restaurants down to 1 *before* joining
The other 30,480 restaurants never touch the following joins

Equivalence Rule 2: Projection Pushdown

```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
 ORDER BY InspectionDate;
```

Rule: Project away unneeded columns as soon as possible

For our query, we only output 4 columns:

`r.RestaurantID, r.RestaurantName, i.InspectionDate, vc.ViolationCodeDescription`

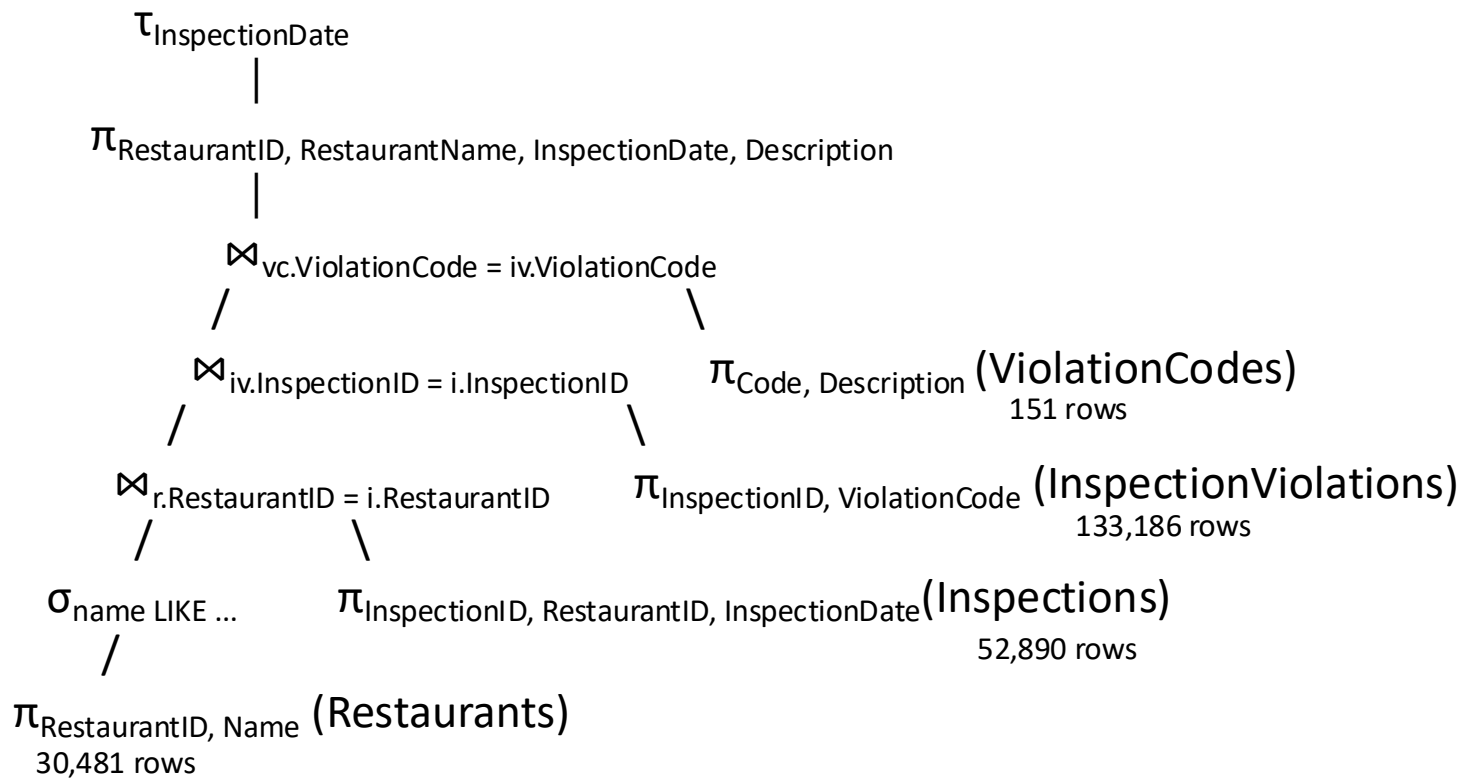
We can drop unnecessary columns early

Why it helps:

- In MySQL rows live in **16 KB pages**
- Smaller tuples -> more tuples per page -> fewer page reads
- Reduces memory pressure in joins and sorts

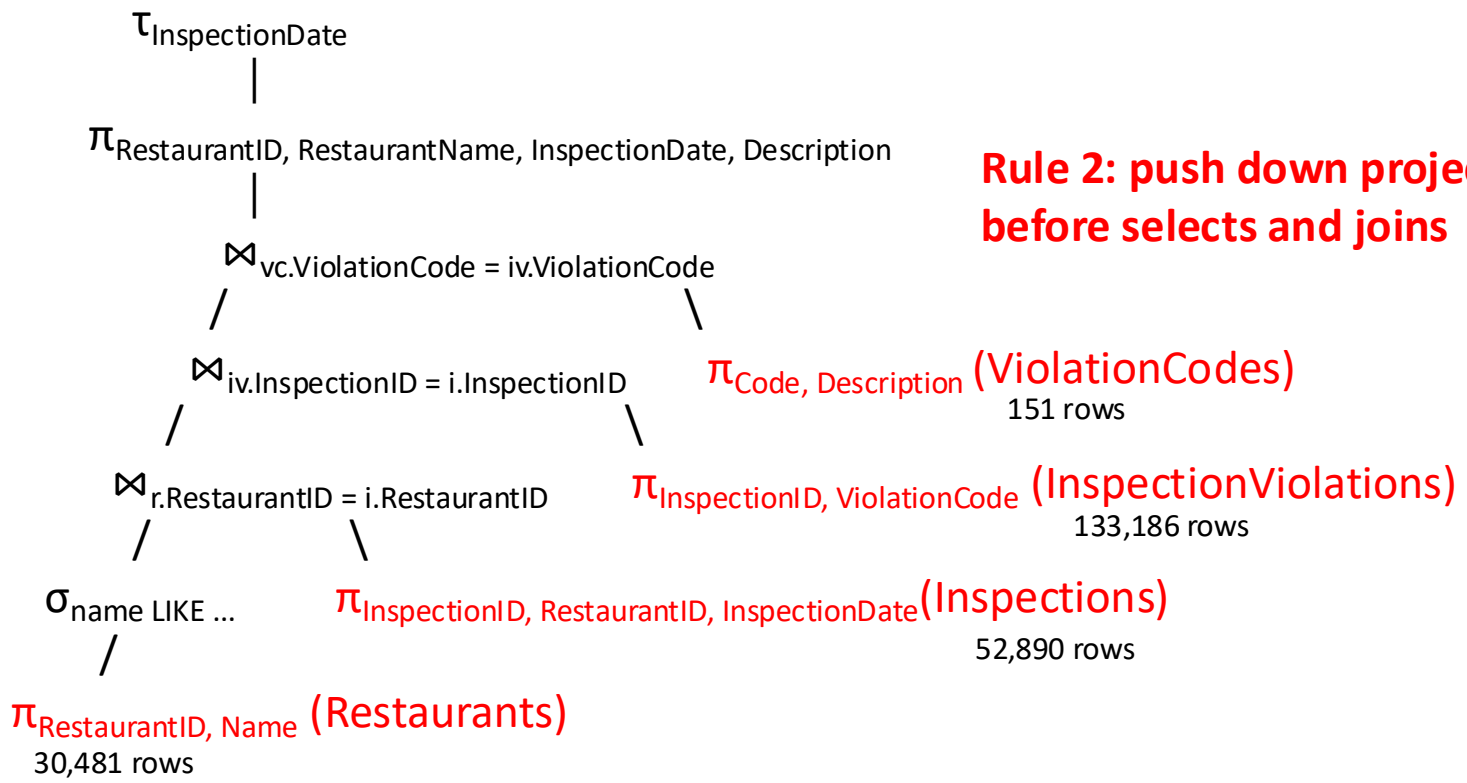
Equivalent query but now optimized

```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```



Equivalent query but now optimized

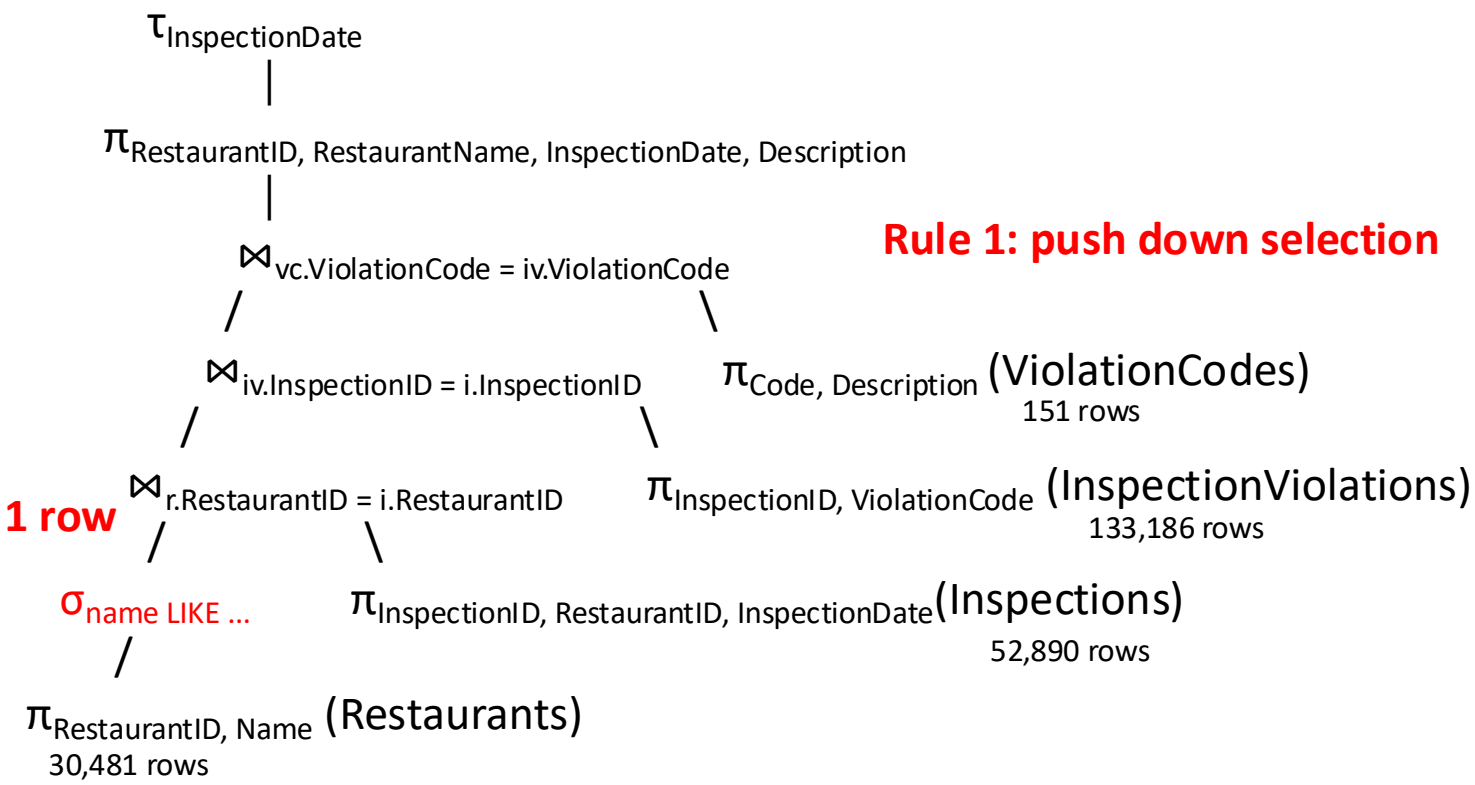
```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```



Rule 2: push down projection before selects and joins

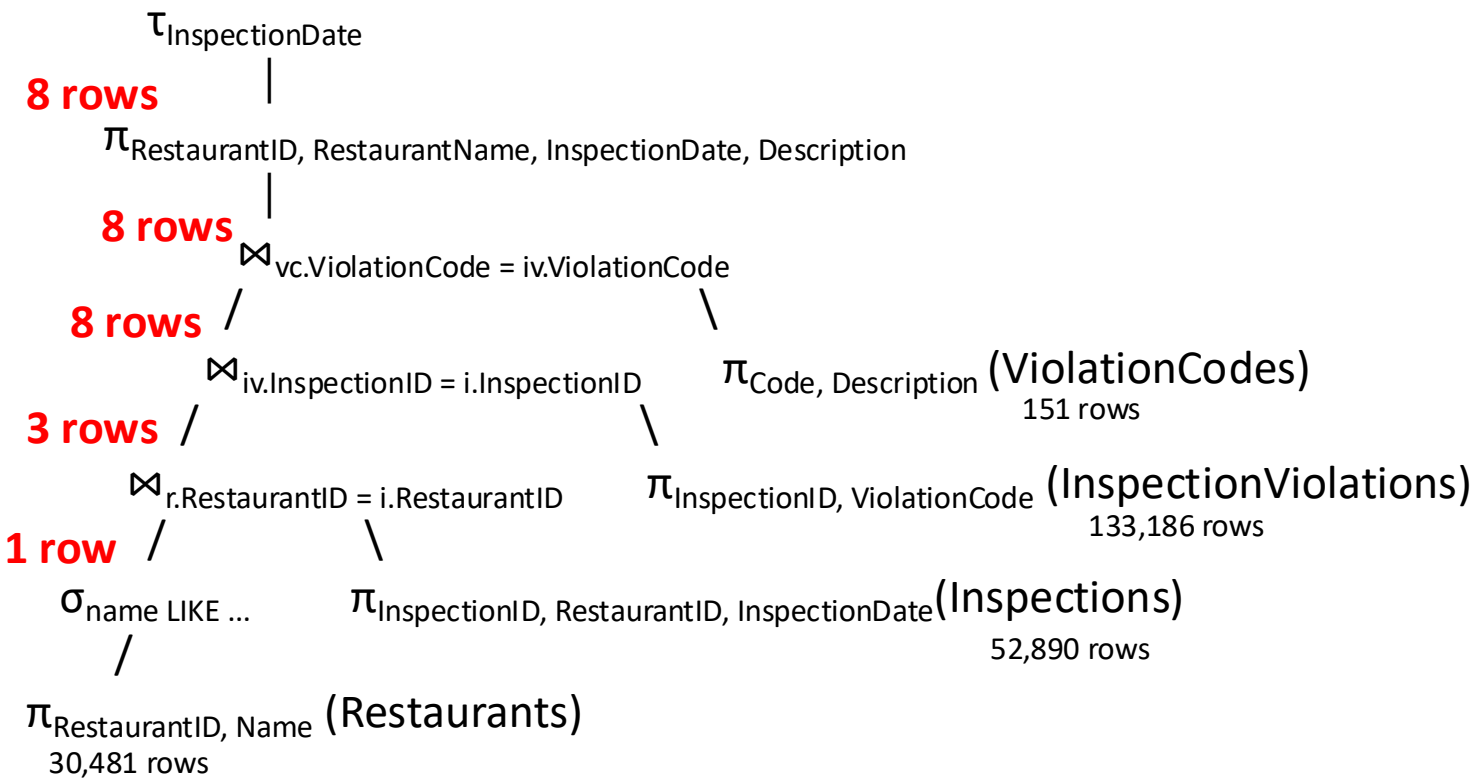
Equivalent query but now optimized

```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```

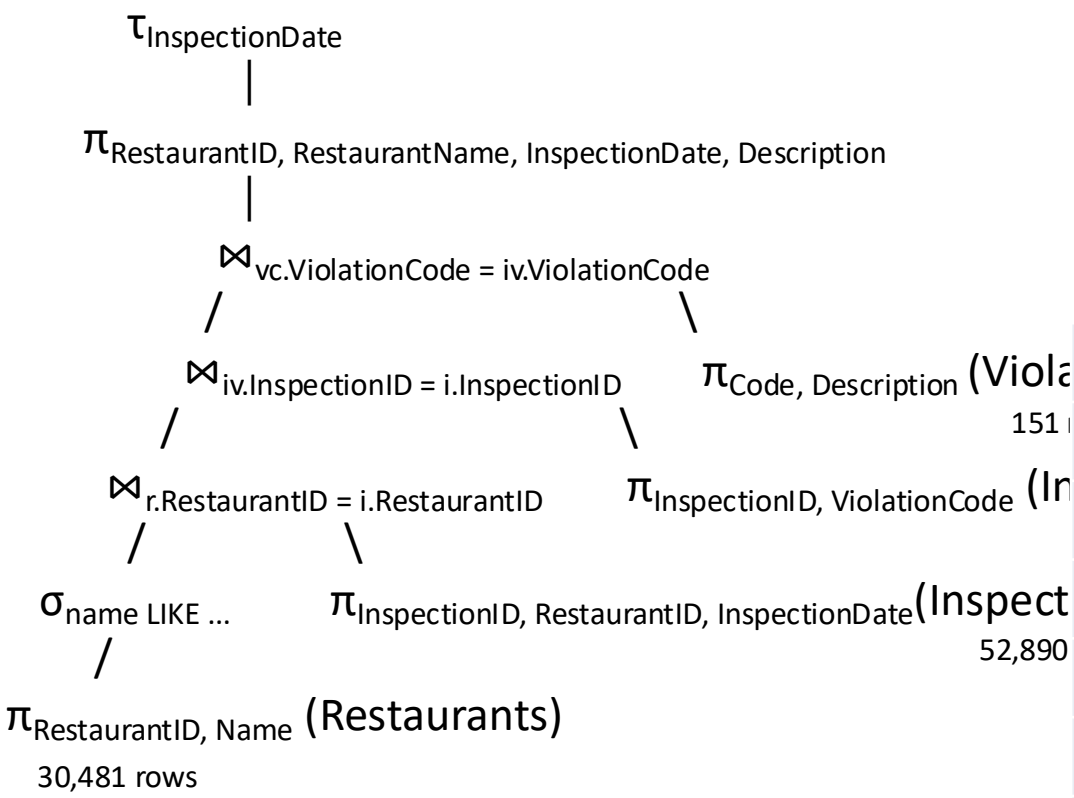


Equivalent query but now optimized

```
SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
 ORDER BY InspectionDate;
```



Equivalent query but now optimized



Assume database

1. Reads needed tables
2. Processes rows
3. Writes result back to temporary table

Estimated I/O operations

Step	Op	Read I/O	Write I/O	Total
1	Proj/Select Rest	30,481	1	30,482
2	1 ⋈ Proj Insp	52,891	3	52,894
3	2 ⋈ Proj InspViol	133,187	8	133,195
4	3 ⋈ Proj ViolCodes	152	8	160
5	Proj	8	8	16
6	Sort	8	8	16
	Total	**216,727**	**36**	**216,763**

Compared with total 855,272 for the naïve query
Plus, lower row processing costs (fewer attributes)

MySQL also considers other optimizations

- **Join Commutativity and Associativity:** ordering of joins change output
- **Constant folding:** `WHERE 1=1 AND name LIKE 'X%'` -> `WHERE name LIKE 'X%'`
- **Subquery -> semi-join:** `WHERE x IN (SELECT ...)` rewritten as a join
- **Outer join -> inner join:** when WHERE filters out NULLs anyway
- **View merging:** inline view definitions into the outer query
- **ORDER BY elimination:** if an index already provides the order
  - If we had an index on `InspectionDate`
  - MySQL could skip the sort step in our query due to index
- **DISTINCT elimination:** if columns include a unique key skip DISTINCT
- **Trivial condition removal:** `WHERE FALSE` -> return empty without scanning

Database creates estimates of tables to help estimate costs

ANALYZE TABLE Restaurants;

1. Samples some pages from each index on the table
2. Counts distinct values within those samples
3. Extrapolates to estimate cardinality for the whole table
4. Writes the results to `mysql.innodb_table_stats` and `mysql.innodb_index_stats`
5. Invalidates cached query plans that referenced this table

It usually takes seconds, even on large tables, because it samples rather than scans

Runs automatically when more than 10% of rows change

Database analyzes tables by sampling them to help estimate costs

Sample tables to get estimates

```
39 •  ANALYZE TABLE Restaurants, Inspections, InspectionViolations, ViolationCodes;
40
41 •  SELECT database_name, table_name, n_rows, clustered_index_size
42    FROM mysql.innodb_table_stats
43    WHERE database_name = DATABASE();
44
```

See estimates in mysql.innodb_table_stats table

100% 25:33

Result Grid | Filter Rows: Search | Edit: | Export/Import:

database_name	table_name	n_rows	clustered_index_size
nyc_data	actions	5	1
nyc_data	cuisine	90	1
nyc_data	employees	5	1
nyc_data	inspections	53001	225
nyc_data	inspectiontypes	34	1
nyc_data	inspectionviolations	133497	289
nyc_data	restaurant_inspections	281096	6569
nyc_data	restaurants	30352	289
nyc_data	users	5	1
nyc_data	violationcodes	151	4
NULL	NULL	NULL	NULL

Estimated number of rows and pages

Database will assess different costs to different actions to estimate query costs

```
37 •  SELECT cost_name, default_value
38        FROM mysql.engine_cost
39    UNION
40    SELECT cost_name, default_value
41        FROM mysql.server_cost;
```

100% 13:30

Result Grid Filter Rows: Search

cost_name	default_value
io_block_read_cost	1
memory_block_read_cost	0.25
disk_temptable_create_cost	20
disk_temptable_row_cost	0.5
key_compare_cost	0.05
memory_temptable_create_cost	1
memory_temptable_row_cost	0.1
row_evaluate_cost	0.1

With estimates of table characteristics and costs for actions, database can estimate query costs

Default costs used to estimate query costs

Database Administrators (DBA) tweak these costs

We will come back to this table

Agenda

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

Often indexes can increase SQL read performance significantly

Show indexes

Tables with a Primary Key always have an index on that key

Key name
Primary key always called PRIMARY

YES if column can be NULL

Table name

Column name

Cardinality: Estimated number of unique values

Can the optimizer use this index?

```
65    -- we can see existing indicies with SHOW INDEXES
66 •  SHOW INDEXES FROM Restaurants;
67
```

100% 39:61

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	Index_comment	Visible	Expre
Restaurants	0	PRIMARY	1	RestaurantID	A	30296	NULL	NULL		BTREE			YES	NULL
Restaurants	1	fk_Restaurant_Cuisine1	1	CuisineID	A	84	NULL	NULL	YES	BTREE			YES	NULL

1 if can contain duplicates
0 otherwise

Column sequence number in index (first starts at 1 for multi-column indexes)

Collation: how sorted
A = ascending
D = descending

Null, entire column indexed otherwise number of indexed characters (often used for text)

Type of index: BTREE (default) HASH

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

Sometimes the Primary Key index doesn't help

```
63 •  SELECT *
64        FROM Restaurants
65        WHERE Boro = 'Manhattan';
66
67
```

100% 5:64

Restaurants has index on RestaurantID and CuisineID, but not boro

Must scan all rows to find Boro = 'Manhattan'

Result Grid | Filter Rows: Search | Edit: | Export/Import: | Fetch rows:

Restau...	RestaurantName	Boro	Building	Street	ZipCode	Phone	Latitude	Longitude	InspectionAvg	Ir
30191841	D.J. REYNOLDS	Manhattan	351	WEST 57 STREET	10019	2122452912	40.76732571155	-73.98431042362	18.4	1
40359480	1 EAST 66TH STREET KITCHEN	Manhattan	1	EAST 66 STREET	10065	2128793900	40.76854691423	-73.96958057845	9	4
40362264	P & S DELI GROCERY	Manhattan	730	COLUMBUS AVENUE	10025	2129323030	40.79262063589	-73.96770961573	12.5455	1
40362274	ANGELIKA FILM CENTER	Manhattan	18	WEST HOUSTON STREET	10012	2129952570	40.72574361744	-73.99747810759	10.2	5
40363298	CAFE METRO	Manhattan	625	8 AVENUE	10018	2127149342	40.75618542308	-73.99056473379	18.9091	1
40364179	MISS MAIME'S SPOONBREAD TOO	Manhattan	364	WEST 110 STREET	10025	2128656744	40.80137126967	-73.96015994361	20.4	5
40364347	METROPOLITAN CLUB	Manhattan	1	EAST 60 STREET	10022	2128387400	40.76479552185	-73.97230783434	10.7857	1
40364373	ISLE OF CAPRI	Manhattan	1028	3 AVENUE	10065	2127581902	40.76270773611	-73.96574957601	17.625	8
40364389	OLD TOWN BAR	Manhattan	45	EAST 18 STREET	10003	2125296732	40.73759507351	-73.98964720852	10.5556	9
40364439	SEVILLA	Manhattan	62	CHARLES STREET	10014	2129293189	40.73490837812	-74.00297328507	31.6364	1
40364443	CRIMINAL COURT BLDG CAFETERIA	Manhattan	100	CENTRE STREET	10013	2129621037	40.71607656565	-74.00142489623	17.0769	1

EXPLAIN tells you how the database plans to run the query


```
EXPLAIN SELECT *
    FROM Restaurants
    WHERE Boro = 'Manhattan';
```

```
-> Filter: (restaurants.Boro = 'Manhattan')  (cost=3107 rows=3035)
    -> Table scan on Restaurants  (cost=3107 rows=30352)
```

Read from inner/bottom outward
Database is telling us it will have to scan all rows in Restaurant (a table scan)
There is no way to know which rows are for Manhattan, there is no index on Boro
Database *estimates* there are 30,352 rows (but there are really 30,481, so pretty close)
Cost estimate: Total I/O Cost + Total Processing Cost
Restaurants uses 289 disk blocks

EXPLAIN tells you how the database plans to run the query

Add EXPLAIN before SQL command

```
EXPLAIN SELECT *
    FROM Restaurant
    WHERE Boro = 'M
```

```
-> Filter: (restau
    -> Table scan
```

Read from inner/botto
Database is telling us it
There is no way to kno
Database *estimates* the
Cost estimate: Total I/C
Restaurants uses 2

```
41 •  SELECT database_name, table_name, n_rows, clustered_index_size
42    FROM mysql.innodb_table_stats
43    WHERE database_name = DATABASE();
44
```

100% 25:33

Result Grid | Filter Rows: Search | Edit: | Export/Import:

database_name	table_name	n_rows	clustered_index_size
nyc_data	actions	5	1
nyc_data	cuisine	90	1
nyc_data	employees	5	1
nyc_data	inspections	53001	225
nyc_data	inspectiontypes	34	1
nyc_data	inspectionviolations	133497	289
nyc_data	restaurant_inspections	281096	6569
nyc_data	restaurants	30352	289
nyc_data	users	5	1
nyc_data	violationcodes	151	4
NULL	NULL	NULL	NULL

EXPLAIN tells you how the database plans to run the query


```
EXPLAIN SELECT *
    FROM Restaurants
    WHERE Boro = 'Manhattan';
```

```
-> Filter: (restaurants.Boro = 'Manhattan')  (cost=3107 rows=3035)
    -> Table scan on Restaurants  (cost=3107 rows=30352)
```

Read from inner/bottom outward
Database is telling us it will have to scan all rows in Restaurant (a table scan)
There is no way to know which rows are for Manhattan, there is no index on Boro
Database *estimates* there are 30,352 rows (but there are really 30,481, so pretty close)
Cost estimate: Total I/O Cost + Total Processing Cost
- **Restaurants uses 289 disk blocks**
- **Database charges 0.25 for each memory_block_read_cost**

EXPLAIN tells you how the database plans to run the query


```
EXPLAIN SELECT *
    FROM Restaurants
    WHERE Boro = 'Manhattan';
```

```
-> Filter: (restaurants.Boro = 'Manhattan')  (cost=3107 rows=3035)
    -> Table scan on Restaurants  (cost=3107 rows=30352)
```

Read from inner/bottom outward
Database is telling us it will have to scan all rows in Restaurant (a table scan)
There is no way to know which rows are for Manhattan, there is no index on Boro
Database *estimates* there are 30,352 rows (but there are really 30,481, so pretty close)
Cost estimate: Total I/O Cost + Total Processing Cost
- **Restaurants uses 289 disk blocks**
- **Database charges 0.25 for each memory_block_read_cost**
- **Charges 0.1 for row_evaluate_cost**

EXPLAIN tells you how the database plans to run the query

Add EXPLAIN before SQL command

```
EXPLAIN SELECT *
    FROM Restaurants
    WHERE Boro = 'Manhattan';
```

```
-> Filter: (restaurants.Boro = 'Ma
    -> Table scan on Restaurants
```

Read from inner/bottom outward

Database is telling us it will have to scan all r

There is no way to know which rows are for

Database *estimates* there are 30,352 rows (I

Cost estimate: Total I/O Cost + Total Processi

- Restaurants uses 289 disk blocks
- Database charges 0.25 for each memor
- Charges 0.1 for row_evaluate_cost on e

```
37 •  SELECT cost_name, default_value
38        FROM mysql.engine_cost
39    UNION
40    SELECT cost_name, default_value
41        FROM mysql.server_cost;
```

100% 13:30

Result Grid Filter Rows: Search

cost_name	default_value
io_block_read_cost	1
memory_block_read_cost	0.25
disk_temptable_create_cost	20
disk_temptable_row_cost	0.5
key_compare_cost	0.05
memory_temptable_create_cost	1
memory_temptable_row_cost	0.1
row_evaluate_cost	0.1

EXPLAIN tells you how the database plans to run the query


```
EXPLAIN SELECT *
    FROM Restaurants
    WHERE Boro = 'Manhattan';
```

```
-> Filter: (restaurants.Boro = 'Manhattan')  (cost=3107 rows=3035)
    -> Table scan on Restaurants  (cost=3107 rows=30352)
```

Read from inner/bottom outward
Database is telling us it will have to scan all rows in Restaurant (a table scan)
There is no way to know which rows are for Manhattan, there is no index on Boro
Database *estimates* there are 30,352 rows (but there are really 30,481, so pretty close)
Cost estimate: Total I/O Cost + Total Processing Cost

- **Restaurants uses 289 disk blocks**
- **Database charges 0.25 for each memory_block_read_cost**
- **Charges 0.1 for row_evaluate_cost on each of the estimated 30,352 rows**
- **Total cost: 289 * 0.25 + 30,352 * 0.10 = 72.25 + 3035.2 = 3107.45**

EXPLAIN tells you how the database plans to run the query

Add EXPLAIN before SQL command

```
EXPLAIN SELECT *
    FROM Restaurants
    WHERE Boro = 'Manhattan';
```

```
-> Filter: (restaurants.Boro = 'Manhattan')  (cost=3107 rows=3035)
    -> Table scan on Restaurants  (cost=3107 rows=30352)
```

Read from inner/bottom outward
Database is telling us it will have to scan all rows in Restaurant (a table scan)
There is no way to know which rows are for Manhattan, there is no index on Boro
Database *estimates* there are 30,352 rows (but there are really 30,481, so pretty close)
Cost estimate: Total I/O Cost + Total Processing Cost
- Restaurants uses 289 disk blocks
- Database charges 0.25 for each memory_block_read_cost
- Charges 0.1 for row_evaluate_cost on each of the estimated 30,352 rows
- Total cost: 289 * 0.25 + 30,352 * 0.10 = 72.25 + 3035.2 = 3107.45

EXPLAIN tells you how the database plans to run the query

Add EXPLAIN before SQL command

```
EXPLAIN SELECT *
    FROM Restaurants
    WHERE Boro = 'Manhattan';
```

```
-> Filter: (restaurants.Boro = 'Manhattan')  (cost=3107 rows=3035)
    -> Table scan on Restaurants  (cost=3107 rows=30352)
```

Read from inner/bottom outward
Database is telling us it will have to scan all rows in Restaurant (a table scan)
There is no way to know which rows are for Manhattan, there is no index on Boro
Database *estimates* there are 30,352 rows (but there are really 30,481, so pretty close)
Cost estimate: Total I/O Cost + Total Processing Cost
- Restaurants uses 289 disk blocks
- Database charges 0.25 for each memory_block_read_cost
- Charges 0.1 for row_evaluate_cost on each of the estimated 30,352 rows
- Total cost: 289 * 0.25 + 30,352 * 0.10 = 72.25 + 3035.2 = 3107.45

Note: most of the cost here is evaluating all the rows:
- 72.25 cost of reads vs. 3035.2 cost of row evals

Indices can be created on multiple attributes to reduce table scans

If multiple attributes, optimizer can use index on left most prefixes

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

```
CREATE INDEX idx_borozip ON Restaurants(Boro,ZipCode);
SHOW INDEXES FROM Restaurants;
```

100% 19:67

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	Index_comment	Visible	Exp
Restaurants	0	PRIMARY	1	RestaurantID	A	30296	NULL	NULL		BTREE			YES	NUL
Restaurants	1	fk_Restaurant_Cuisine1	1	CuisineID	A	84	NULL	NULL	YES	BTREE			YES	NUL
Restaurants	1	idx_borozip	1	Boro	A	7	NULL	NULL	YES	BTREE			YES	NUL
Restaurants	1	idx_borozip	2	ZipCode	A	231	NULL	NULL	YES	BTREE			YES	NUL

Index on Boro first, zipcode second

Creates secondary index with B+ tree using keys of (boro, zipcode) -> Primary Key

Search secondary B+ tree for boro (or boro, zipcode) to get Primary Key, then search clustered index to find row data

With an index on boro and zip, queries that involve boro, zip do not search the whole table

Only search the relevant rows for given boro, zip (or just boro)

EXPLAIN shows the secondary index helps reduce cost

```
EXPLAIN SELECT *
    FROM Restaurants
    WHERE Boro = 'Manhattan';
```

MySQL will now use the secondary index we just added to jump straight to rows with boro of Manhattan

```
-> Index lookup on Restaurants using idx_borozip (Boro = 'Manhattan')
(cost=1734 rows=15176)
```

Cost drops to 1734 (previously 3107)
Why not lower?
Secondary indexes give the block containing the Primary Keys for blocks matching 'Manhattan'
Database must go read the blocks containing the Primary Key to get all fields (*) requested
Shown as Index lookup here, sometimes referred to as a *bookmark lookup*

The secondary doesn't help on ZipCode alone

```
EXPLAIN SELECT *
  FROM Restaurants
  WHERE ZipCode = 10023;
```

```
-> Filter: (restaurants.ZipCode = 10023)  (cost=3033 rows=2961)
    -> Table scan on Restaurants  (cost=3033 rows=29610)
```

Full table scan required if using ZipCode in WHERE clause

Why?

Index created on (Boro, Zipcode), not just ZipCode

```
CREATE INDEX idx_borozip ON Restaurants(Boro,ZipCode);
```

Boro,Zipcode index shines using both attributes

```
EXPLAIN SELECT *
  FROM Restaurants
  WHERE ZipCode = 10023 and Boro = 'Manhattan';
```

```
-> Index lookup on Restaurants using idx_borozip (Boro =
'Manhattan', ZipCode = 10023)  (cost=77.7 rows=222)
```

Cost drops to only 77 when using both attributes of the index!
Why?
Database uses secondary index to find only rows in ZipCode 10023 and Boro Manhattan

```
SELECT COUNT(*)
    FROM Restaurants
    WHERE ZipCode = 10023 and Boro = 'Manhattan';
```

COUNT(*)
222

Only 222 rows match, find them quickly from the 30,481 rows in Restaurants thanks to the index

Add an index on the first three letters of RestaurantName

```
-- drop idx_borozip index
DROP INDEX idx_borozip ON Restaurants;

-- looking for restaurant by name (no index on name)
EXPLAIN SELECT RestaurantName
    FROM Restaurants
    WHERE RestaurantName like 'Tim%'; -- scans all rows
```

```
-> Filter: (restaurants.RestaurantName like 'Tim%')  (cost=3033 rows=3290)
    -> Table scan on Restaurants  (cost=3033 rows=29610)
```

Full table scan on RestaurantName
Primary Key index doesn't help
If you are going to query by RestaurantName a lot, consider putting an index on that attribute
Remember, however, indexes speed up database reads, but slow down writes

Add an index on the first three letters of RestaurantName

```
CREATE INDEX idx_name ON Restaurants(RestaurantName(3));
EXPLAIN SELECT RestaurantName
    FROM Restaurants
    WHERE RestaurantName LIKE 'Tim%';
```

```
-> Filter: (restaurants.RestaurantName like 'Tim%')  (cost=11.5 rows=25)
    -> Index range scan on Restaurants using idx_name over (RestaurantName =
'Tim')  (cost=11.5 rows=25)
```

Now only search for table rows where the first three letter ntName are Tim
Turns out there are 25 that match that criteria
Cost is 11.5 vs. table scan 3107
Index helps a lot!

RestaurantName
Tim's Tasty Treats
TIMES SQUARE CAFE
TIMES SQUARE DINER & GRILL
TIM HORTONS
TIME TEQUILA BAR CAFE
TIM HO WAN
TIM HORTONS
TIM HORTONS
TIME AGAIN
TIMES SQUARE MARKET
TIME DELIGHT RESTAURANT
TIMES SQUARE TOWER CAFE
TIMES DELI & CAFE
TIME AND TIDE
TIMUR HOUSE
TIME 4 BAGELS
TIMBUKTU MARKET LLC
TIME OUT MARKET (MANHATTAN) LLC
TIME OUT MARKET (MANHATTAN) LLC
TIME OUT MARKET (MANHATTAN) LLC
TIME OUT MARKET (MANHATTAN) LLC
TIME OUT MARKET (MANHATTAN) LLC
TIME OUT MARKET (MANHATTAN) LLC
TIME OUT MARKET (MANHATTAN) LLC
TIME OUT MARKET (MANHATTAN) LLC

EXPLAIN shows database plan on a complex query

```
EXPLAIN SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
  ORDER BY InspectionDate;
```

Assume no index on RestaurantName

```
-> Sort: i.InspectionDate
  -> Stream results  (cost=15787 rows=18511)
    -> Nested loop inner join  (cost=15787 rows=18511)
      -> Nested loop inner join  (cost=9309 rows=18511)
        -> Nested loop inner join  (cost=5633 rows=7231)
          -> Filter: (r.RestaurantName like 'Morris Park Bake%')  (cost=3102 rows=3366)
            -> Table scan on r  (cost=3102 rows=30296)
          -> Index lookup on i using fk_Inspections_Restaurants_idx (RestaurantID = r.RestaurantID)  (cost=0.537 rows=2.15)
        -> Covering index lookup on iv using fk_InspectionViolations_Inspections1_idx (InspectionID = i.InspectionID)  (cost=0.252 rows=2.56)
      -> Single-row index lookup on vc using PRIMARY (ViolationCode = iv.ViolationCode)  (cost=0.25 rows=1)
```

EXPLAIN shows database plan on a complex query

```
EXPLAIN SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
  ORDER BY InspectionDate;
```

```
-> Sort: i.InspectionDate
  -> Stream results  (cost=15787 rows=18511)
    -> Nested loop inner join  (cost=15787 rows=18511)
      -> Nested loop inner join  (cost=9309 rows=18511)
        -> Nested loop inner join  (cost=5633 rows=7231)
          -> Filter: (r.RestaurantName like 'Morris Park Bake%')  (cost=3102 rows=3366)
            -> Table scan on r  (cost=3102 rows=30296)
          -> Index lookup on i using fk_Inspections_Restaurants_idx (RestaurantID = r.RestaurantID)  (cost=0.537 rows=2.15)
        -> Covering index lookup on iv using fk_InspectionViolations_Inspections1_idx (InspectionID = i.InspectionID)  (cost=0.252 rows=2.56)
      -> Single-row index lookup on vc using PRIMARY (ViolationCode = iv.ViolationCode)  (cost=0.25 rows=1)
```

1) Start at inner/bottom most and read outward

Table scan on Restaurants because no index on Name

EXPLAIN shows database plan on a complex query

```
EXPLAIN SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
  ORDER BY InspectionDate;
```

2) Select push down
Filter rows to only Morris Park Bake%
Database is expecting to find 3366 rows (but find only 1, so following estimates will be much too high!)

```
-> Sort: i.InspectionDate
  -> Stream results  (cost=15787 rows=18511)
    -> Nested loop inner join  (cost=15787 rows=18511)
      -> Nested loop inner join  (cost=9309 rows=18511)
        -> Nested loop inner join  (cost=5633 rows=7231)
          -> Filter: (r.RestaurantName like 'Morris Park Bake%')  (cost=3102 rows=3366)
            -> Table scan on r  (cost=3102 rows=30296)
          -> Index lookup on i using fk_Inspections_Restaurants_idx (RestaurantID = r.RestaurantID)  (cost=0.537 rows=2.15)
        -> Covering index lookup on iv using fk_InspectionViolations_Inspections1_idx (InspectionID = i.InspectionID)  (cost=0.252 rows=2.56)
      -> Single-row index lookup on vc using PRIMARY (ViolationCode = iv.ViolationCode)  (cost=0.25 rows=1)
```

EXPLAIN shows database plan on a complex query

```
EXPLAIN SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
  ORDER BY InspectionDate;
```

3) Use index to look up *Inspections* using FK from Restaurants
This is an Index Nested Loop (INL) from slide 13

```
-> Sort: i.InspectionDate
  -> Stream results  (cost=15787 rows=18511)
    -> Nested loop inner join  (cost=15787 rows=18511)
      -> Nested loop inner join  (cost=9309 rows=18511)
        -> Nested loop inner join  (cost=5633 rows=7231)
          -> Filter: (r.RestaurantName like 'Morris Park Bake%')  (cost=3102 rows=3366)
            -> Table scan on r  (cost=3102 rows=30296)
          -> Index lookup on i using fk_Inspections_Restaurants_idx (RestaurantID = r.RestaurantID)  (cost=0.537 rows=2.15)
        -> Covering index lookup on iv using fk_InspectionViolations_Inspections1_idx (InspectionID = i.InspectionID)  (cost=0.252 rows=2.56)
      -> Single-row index lookup on vc using PRIMARY (ViolationCode = iv.ViolationCode)  (cost=0.25 rows=1)
```

EXPLAIN shows database plan on a complex query

```
EXPLAIN SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
  ORDER BY InspectionDate;
```

4) Cost so far:

3,102 (table scan)

+ 3,366 × 0.537 (INL one probe per match)

+ 7231 * 0.1 (estimated row processing)

≈ 5,633

```
-> Sort: i.InspectionDate
  -> Stream results  (cost=15787 rows=18511)
    -> Nested loop inner join  (cost=15787 rows=18511)
      -> Nested loop inner join  (cost=9309 rows=18511)
        -> Nested loop inner join  (cost=5633 rows=7231)
          -> Filter: (r.RestaurantName like 'Morris Park Bake%')  (cost=3102 rows=3366)
            -> Table scan on r  (cost=3102 rows=30296)
          -> Index lookup on i using fk_Inspections_Restaurants_idx (RestaurantID = r.RestaurantID)  (cost=0.537 rows=2.15)
        -> Covering index lookup on iv using fk_InspectionViolations_Inspections1_idx (InspectionID = i.InspectionID)  (cost=0.252 rows=2.56)
      -> Single-row index lookup on vc using PRIMARY (ViolationCode = iv.ViolationCode)  (cost=0.25 rows=1)
```

EXPLAIN shows database plan on a complex query

```
EXPLAIN SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
  ORDER BY InspectionDate;
```

```
-> Sort: i.InspectionDate
  -> Stream results  (cost=15787 rows=18511)
    -> Nested loop inner join  (cost=15787 rows=18511)
      -> Nested loop inner join  (cost=9309 rows=18511)
        -> Nested loop inner join  (cost=5633 rows=7231)
          -> Filter: (r.RestaurantName like 'Morris Park Bake%')  (cost=3102 rows=3366)
            -> Table scan on r  (cost=3102 rows=30296)
          -> Index lookup on i using fk_Inspections_Restaurants_idx (RestaurantID = r.RestaurantID)  (cost=0.537 rows=2.15)
        -> Covering index lookup on iv using fk_InspectionViolations_Inspections1_idx (InspectionID = i.InspectionID)  (cost=0.252 rows=2.56)
      -> Single-row index lookup on vc using PRIMARY (ViolationCode = iv.ViolationCode)  (cost=0.25 rows=1)
```

5) *Covering index lookup* means the secondary index gives all needed info
No need to look up PK on clustered index after searching secondary index

EXPLAIN shows database plan on a complex query

```
EXPLAIN SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```

```
-> Sort: i.InspectionDate
  -> Stream results  (cost=15787 rows=18511)
    -> Nested loop inner join  (cost=15787 rows=18511)
      -> Nested loop inner join  (cost=9309 rows=18511)
        -> Nested loop inner join  (cost=5633 rows=7231)
          -> Filter: (r.RestaurantName like 'Morris Park Bake%')  (cost=3102 rows=3366)
            -> Table scan on r  (cost=3102 rows=30296)
          -> Index lookup on i using fk_Inspections_Restaurants_idx (RestaurantID = r.RestaurantID)  (cost=0.537 rows=2.15)
        -> Covering index lookup on iv using fk_InspectionViolations_Inspections1_idx (InspectionID = i.InspectionID)  (cost=0.252 rows=2.56)
      -> Single-row index lookup on vc using PRIMARY (ViolationCode = iv.ViolationCode)  (cost=0.25 rows=1)
```

6) Cumulative cost

Estimating 18,511 rows at this point

EXPLAIN shows database plan on a complex query

```
EXPLAIN SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
  ORDER BY InspectionDate;
```

```
-> Sort: i.InspectionDate
  -> Stream results  (cost=15787 rows=18511)
    -> Nested loop inner join  (cost=15787 rows=18511)
      -> Nested loop inner join  (cost=9309 rows=18511)
        -> Nested loop inner join  (cost=5633 rows=7231)
          -> Filter: (r.RestaurantName like 'Morris Park Bake%')  (cost=3102 rows=3366)
            -> Table scan on r  (cost=3102 rows=30296)
          -> Index lookup on i using fk_Inspections_Restaurants_idx (RestaurantID = r.RestaurantID)  (cost=0.537 rows=2.15)
        -> Covering index lookup on iv using fk_InspectionViolations_Inspections1_idx (InspectionID = i.InspectionID)  (cost=0.252 rows=2.56)
      -> Single-row index lookup on vc using PRIMARY (ViolationCode = iv.ViolationCode)  (cost=0.25 rows=1)
```

7) Single-row index lookup

MySQL knows ViolationCodes table Primary Key has only 1 entry per ViolationCode

Look up row data in ViolationCodes for that Primary Key

EXPLAIN shows database plan on a complex query

```
EXPLAIN SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
ORDER BY InspectionDate;
```

```
-> Sort: i.InspectionDate
  -> Stream results  (cost=15787 rows=18511)
    -> Nested loop inner join  (cost=15787 rows=18511)
      -> Nested loop inner join  (cost=9309 rows=18511)
        -> Nested loop inner join  (cost=5633 rows=7231)
          -> Filter: (r.RestaurantName like 'Morris Park Bake%')  (cost=3102 rows=3366)
            -> Table scan on r  (cost=3102 rows=30296)
          -> Index lookup on i using fk_Inspections_Restaurants_idx (RestaurantID = r.RestaurantID)  (cost=0.537 rows=2.15)
        -> Covering index lookup on iv using fk_InspectionViolations_Inspections1_idx (InspectionID = i.InspectionID)  (cost=0.252 rows=2.56)
      -> Single-row index lookup on vc using PRIMARY (ViolationCode = iv.ViolationCode)  (cost=0.25 rows=1)
```

8) Cumulative expected cost 15,787
expected rows = 18,511

EXPLAIN shows database plan on a complex query

```
EXPLAIN SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
  ORDER BY InspectionDate;
```

```
-> Sort: i.InspectionDate
  -> Stream results  (cost=15787 rows=18511)
    -> Nested loop inner join  (cost=15787 rows=18511)
      -> Nested loop inner join  (cost=9309 rows=18511)
        -> Nested loop inner join  (cost=5633 rows=7231)
          -> Filter: (r.RestaurantName like 'Morris Park Bake%')  (cost=3102 rows=3366)
            -> Table scan on r  (cost=3102 rows=30296)
          -> Index lookup on i using fk_Inspections_Restaurants_idx (RestaurantID = r.RestaurantID)  (cost=0.537 rows=2.15)
        -> Covering index lookup on iv using fk_InspectionViolations_Inspections1_idx (InspectionID = i.InspectionID)  (cost=0.252 rows=2.56)
      -> Single-row index lookup on vc using PRIMARY (ViolationCode = iv.ViolationCode)  (cost=0.25 rows=1)
```

9) Stream results (hold in memory, don't write to temp table)
Sort results

Total estimated cost: 15,787
Estimated output rows: 18,511
Join order: Restaurants -> Inspections -> InspectionViolations -> ViolationCodes
Join algorithm at every step: Index Nested Loop
Sort: filesort at the end

Many inaccurate cost estimates due to RestaurantName LIKE, fix with index

```
EXPLAIN SELECT r.RestaurantID, RestaurantName, InspectionDate, ViolationCodeDescription
    FROM Restaurants r JOIN Inspections i ON r.RestaurantID = i.RestaurantID
        JOIN InspectionViolations iv ON iv.InspectionID = i.InspectionID
        JOIN ViolationCodes vc ON vc.ViolationCode = iv.ViolationCode
    WHERE RestaurantName LIKE 'Morris Park Bake%'
  ORDER BY InspectionDate;
```

```
CREATE INDEX idx_name ON Restaurants(RestaurantName);
```

EXPLAIN ANALYZE will give actual run times

```
-> Sort: i.InspectionDate
 -> Stream results  (cost=4.83 rows=5.23)
   -> Nested loop inner join  (cost=4.83 rows=5.23)
     -> Nested loop inner join  (cost=3 rows=5.23)
       -> Nested loop inner join  (cost=1.94 rows=2.1)
         -> Filter: (r.RestaurantName like 'Morris Park Bake%')  (cost=1.21 rows=1)
           -> Covering index range scan on r using idx_name over ('Morris Park Bake' <= RestaurantName <= 'Morris Park Bake')  (cost=1.21 rows=1
         -> Index lookup on i using fk_Inspections_Restaurants_idx (RestaurantID = r.RestaurantID)  (cost=0.734 rows=2.1)
       -> Covering index lookup on iv using fk_InspectionViolations_Inspections1_idx (InspectionID = i.InspectionID)  (cost=0.371 rows=2.5)
     -> Single-row index lookup on vc using PRIMARY (ViolationCode = iv.ViolationCode)  (cost=0.269 rows=1)
```

Index will return one matching RestaurantName
Other estimates more accurate

EXPLAIN is a Database Admin's primary diagnostic tool

Diagnosing a Slow Query (the #1 use case)

1. Slow query gets reported — by a user, a monitoring alert, or the slow query log
2. DBA copies the query and runs EXPLAIN on it
3. Looks for red flags (next slide)
4. Forms a hypothesis: missing index? bad join order? wrong row estimate?
5. Tries a fix (add index, rewrite query, update stats)
6. Runs EXPLAIN again to confirm the plan changed
7. Runs EXPLAIN ANALYZE to confirm actual performance improved

Database Admins look for red flags based on EXPLAIN output

Symptom	What it means	Fix
Table scan	Must read all rows	Add index to WHERE column
Row estimate is huge	Optimizer expects many rows	Investigate selectivity, indexes, or filters
Extra: filesort	Sorting rows	Add an index matching the ORDER BY column
Extra: Using temporary	Write temp table	Often with GROUP BY or DISTINCT
Estimated vs actual rows differ by 10X or more	Bad stats or correlated columns	ANALYZE table, add histograms

Agenda

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

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 indexes for non-prime attributes used in JOINs
- Drop infrequently used indices

Practice: Indices

Download customers_schema.sql from course web page

- Look 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/