UMBC

AN HONORS UNIVERSITY IN MARYLAND

CMSC 461, Database Management Systems
Spring 2018

Lecture 19 – Query Processing Part 1

These slides are based on “Database System Concepts” 6th edition book (whereas some quotes and figures are used from the book) and are a modified version of the slides which accompany the book (http://codex.cs.yale.edu/avi/db-book/db6/slide-dir/index.html), in addition to the 2009/2012 CMSC 461 slides by Dr. Kalpakis

Dr. Jennifer Sleeman

https://www.csee.umbc.edu/~jsleem1/courses/461/spr18

Logistics

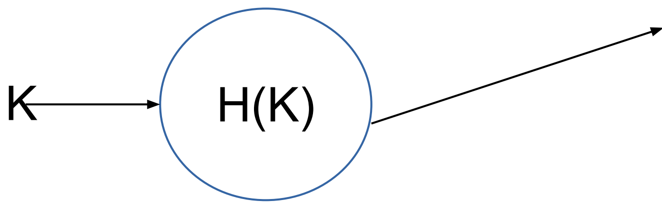
- Homework #4 due 4/9/2018
- Homework #5 due 4/18/2018
- Phase 4 due 4/23/2018

Lecture Outline

- ***Review Hashing***
- Overview of Query Processing
- Selection

Hashing Review

Based on some input key, the address of the bucket is returned



bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Why is this better than sequential file organization?

Based on and image from "Database System Concepts" book and slides, 6th edition

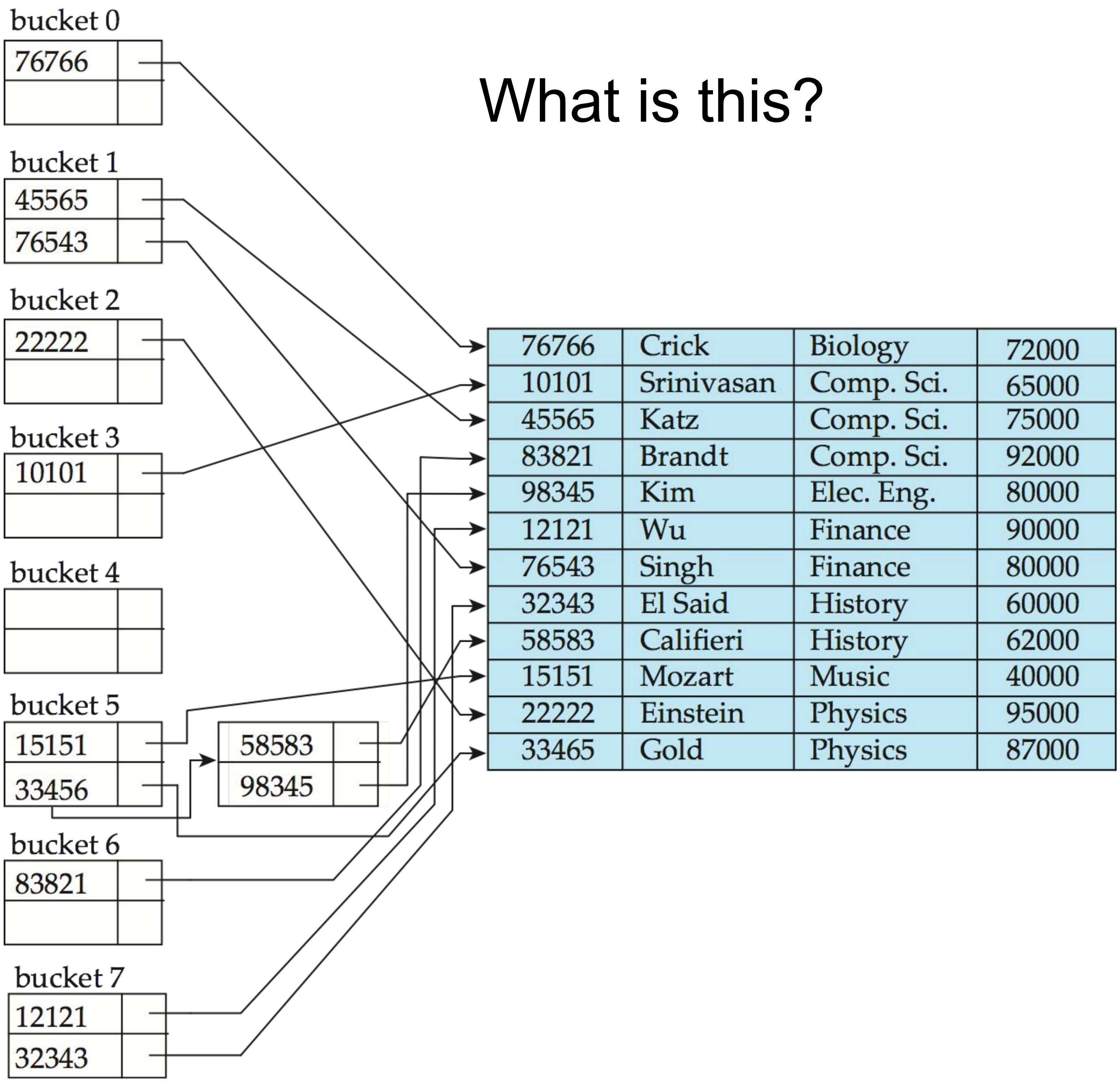
Hashing Review

- What is a bucket?
- Can the hash function return the same bucket for two different keys?
- What is bucket overflow?
- What is one way to handle bucket overflow?
- What properties should our hash function have?

Based on and image from "Database System Concepts" book and slides, 6th edition

Hashing Review

What is this?



Hashing Review

Why is static hashing deficient?

Hashing Review

Why is static hashing deficient?

Fixed set of buckets

When database grows have to use overflow buckets

If space is allocated for future growth, large amount of space wasted

Indexing In-Class

Simple Vs. Unique

```
CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...);
```

```
CREATE INDEX index_name
ON table_name ( column1, column2,...);
```

Indexing In-Class

Altering the index:

```
ALTER table_name ADD PRIMARY KEY (column1, column2,...)

ALTER table_name ADD UNIQUE index_name (column1, column2,...)

ALTER table_name ADD index_name (column1, column2,...)

ALTER table_name ADD FULLTEXT index_name (column1, column2,...)
```

Indexing In-Class

Showing the index:

```
SHOW INDEX FROM table_name;
```

Indexing In-Class

Go to:

https://relational.fit.cvut.cz/dataset/IMDb

We are going to play with this database.

Indexing In-Class

Issue the following commands at the command line:

mysql -h relational.fit.cvut.cz -u guest -p

(where password is ‘relational’)

use imdb_ijs;

Indexing In-Class

```
mysql> show tables;
+--------------------+
| Tables_in_imdb_ijs |
+--------------------+
| actors             |
| directors          |
| directors_genres   |
| movies             |
| movies_directors   |
| movies_genres      |
| roles              |
+--------------------+
7 rows in set (0.11 sec)
```

Indexing In-Class

```
mysql> describe movies;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| id    | int(11)      | NO   | PRI | 0       |       |
| name  | varchar(100) | YES  | MUL | NULL    |       |
| year  | int(11)      | YES  |     | NULL    |       |
| rank  | float        | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
4 rows in set (0.16 sec)
```

Indexing In-Class

```
mysql> describe actors;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| id         | int(11)      | NO   | PRI | 0       |       |
| first_name | varchar(100) | YES  | MUL | NULL    |       |
| last_name  | varchar(100) | YES  | MUL | NULL    |       |
| gender     | char(1)      | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
4 rows in set (0.19 sec)
```

Indexing In-Class

```
mysql> describe roles;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| actor_id | int(11)      | NO   | PRI | NULL    |       |
| movie_id | int(11)      | NO   | PRI | NULL    |       |
| role     | varchar(100) | NO   | PRI | NULL    |       |
+----------+--------------+------+-----+---------+-------+
3 rows in set (0.14 sec)
```

Indexing In-Class

Noticed we gave -h relational.fit.cvut.cz command to mysql which means we are connecting to a remote database

Indexing In-Class

Let's create a local version of this database:

We are going to use mysqldump to do it.

mysqldump -h relational.fit.cvut.cz imdb_ijs -u guest -p > myimdb.sql

Indexing In-Class

Let's create a local version of this database:

```
CREATE DATABASE imdb;
```

```
GRANT ALL ON imdb.* TO root@'localhost';
```

```
mysql -u root -p imdb < myimdb.sql
```

Indexing In-Class

Log in to mysql this time as local user on the localhost:

```
mysql -u root -p
```

```
use imdb;
```

Indexing In-Class

Issue the following query:

```
Select * from actors limit 500;
```

Look at total time to execute.

Indexing In-Class

Log in to mysql this time as local user:

```
mysql -u root -p
```

```
use imdb;
```

Issue the following query:

```
Select * from movies limit 500;
```

Look at total time to execute.

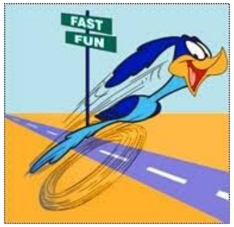
Indexing In-Class

```
SHOW INDEX FROM actors;
```

What do you see?

Select * from actors where first_name like 'Ko%';

Look at total time to execute.

1302 rows in set (0.01 sec) 

Indexing In-Class

Now remove the index.

```
SHOW INDEX FROM actors;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
actors	0	PRIMARY	1	id	A	818795	NULL	NULL		BTREE		
actors	1	actors_first_name	1	first_name	A	90311	NULL	NULL	YES	BTREE		
actors	1	actors_last_name	1	last_name	A	283192	NULL	NULL	YES	BTREE		

3 rows in set (0.00 sec)

Indexing In-Class

Now remove the index.

```
ALTER TABLE actors DROP INDEX
actors_first_name;
```

Indexing In-Class

After removing the index verify it is gone.

```
SHOW INDEX FROM actors;
```

```
+--------+------------+-----------------+-------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---
------------+
| Table  | Non_unique | Key_name        | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment |
Index_comment |
+--------+------------+-----------------+-------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---
------------+
| actors |          0 | PRIMARY         |            1 | id          | A         |      818795 |     NULL | NULL   |      | BTREE      |         |
|
| actors |          1 | actors_last_name |            1 | last_name   | A         |      283192 |     NULL | NULL   | YES  | BTREE      |         |
|
+--------+------------+-----------------+-------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---
------------+
2 rows in set (0.00 sec)
```

Indexing In-Class

Now run the query again.

Select * from actors where first_name like 'Ko%';

Look at total time to execute.

1302 rows in set (*0.21* sec)

Indexing In-Class

According to the MySQL documentation:

B-Tree Index Characteristics

A B-tree index can be used for column comparisons in expressions that use the =, >, >=, <, <=, or BETWEEN operators. The index also can be used for LIKE comparisons if the argument to LIKE is a constant string that does not start with a wildcard character. For example, the following SELECT statements use indexes:

```
SELECT * FROM tbl_name WHERE key_col LIKE 'Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE 'Pat%_ck%';
```

In the first statement, only rows with `'Patrick' <= key_col < 'Patricl'` are considered. In the second statement, only rows with `'Pat' <= key_col < 'Pau'` are considered.

The following SELECT statements do not use indexes:

```
SELECT * FROM tbl_name WHERE key_col LIKE '%Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE other_col;
```

In the first statement, the LIKE value begins with a wildcard character. In the second statement, the LIKE value is not a constant.

Source: https://dev.mysql.com/doc/refman/5.5/en/index-btree-hash.html

Indexing In-Class

Let’s test this to verify it is true:

Notice we don’t have our index.

```
SHOW INDEX FROM actors;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
actors	0	PRIMARY	1	id	A	818795	NULL	NULL		BTREE		
actors	1	actors_last_name	1	last_name	A	283192	NULL	NULL	YES	BTREE		

2 rows in set (0.00 sec)

Indexing In-Class

Let’s test this to verify it is true:

Select * from actors where first_name like '%ok%' or first_name like '%ri%';

Look at total time to execute.

83531 rows in set (0.37 sec)

Indexing In-Class

Let's add the index again.

ALTER TABLE actors ADD INDEX actors_first_name (first_name);

```
SHOW INDEX FROM actors;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
actors	0	PRIMARY	1	id	A	818795	NULL	NULL		BTREE		
actors	1	actors_first_name	1	first_name	A	90311	NULL	NULL	YES	BTREE		
actors	1	actors_last_name	1	last_name	A	283192	NULL	NULL	YES	BTREE		

3 rows in set (0.00 sec)

Indexing In-Class

Let's test this to verify it is true:

Select * from actors where first_name like '%ok%' or first_name like '%ri%';

Look at total time to execute.

83531 rows in set (0.34 sec)

Indexing In-Class

MySQL documentation on hashing indexing:

Hash Index Characteristics

Hash indexes have somewhat different characteristics from those just discussed:

- They are used only for equality comparisons that use the **=** or **<=>** operators (but are *very* fast). They are not used for comparison operators such as **<** that find a range of values. Systems that rely on this type of single-value lookup are known as "key-value stores"; to use MySQL for such applications, use hash indexes wherever possible.
- The optimizer cannot use a hash index to speed up `ORDER BY` operations. (This type of index cannot be used to search for the next entry in order.)
- MySQL cannot determine approximately how many rows there are between two values (this is used by the range optimizer to decide which index to use). This may affect some queries if you change a `MyISAM` or `InnoDB` table to a hash-indexed `MEMORY` table.
- Only whole keys can be used to search for a row. (With a B-tree index, any leftmost prefix of the key can be used to find rows.)

Source: https://dev.mysql.com/doc/refman/5.5/en/index-btree-hash.html

Indexing In-Class

Remove the index.

```
ALTER TABLE actors DROP INDEX
actors_first_name;
```

Indexing In-Class

After removing the index verify it is gone.

```
SHOW INDEX FROM actors;
```

```
+--------+-----------+-----------------+-------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---
------------+
| Table  | Non_unique | Key_name        | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment |
Index_comment |
+--------+-----------+-----------------+-------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---
------------+
| actors |          0 | PRIMARY         |            1 | id          | A         |      818795 |     NULL | NULL   |      | BTREE      |         |
|
| actors |          1 | actors_last_name |            1 | last_name   | A         |      283192 |     NULL | NULL   | YES  | BTREE      |         |
|
+--------+-----------+-----------------+-------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---
------------+
2 rows in set (0.00 sec)
```

Indexing In-Class

Let’s add the index as a hash.

ALTER TABLE actors ADD INDEX actors_first_name (first_name) USING HASH;

```
SHOW INDEX FROM actors;
```

```
+--------+------------+-------------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+--
-------------+
| Table  | Non_unique | Key_name          | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment |
Index_comment |
+--------+------------+-------------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+--
-------------+
| actors |          0 | PRIMARY           |            1 | id          | A         |      818795 |     NULL | NULL   |      | BTREE      |         |
|
| actors |          1 | actors_first_name |            1 | first_name  | A         |       90311 |     NULL | NULL   | YES  | BTREE      |         |
|
| actors |          1 | actors_last_name  |            1 | last_name   | A         |      283192 |     NULL | NULL   | YES  | BTREE      |         |
|
+--------+------------+-------------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+--
-------------+
3 rows in set (0.00 sec)
```

Indexing In-Class

Rerun the queries:

Select * from actors where first_name like 'Ko%';

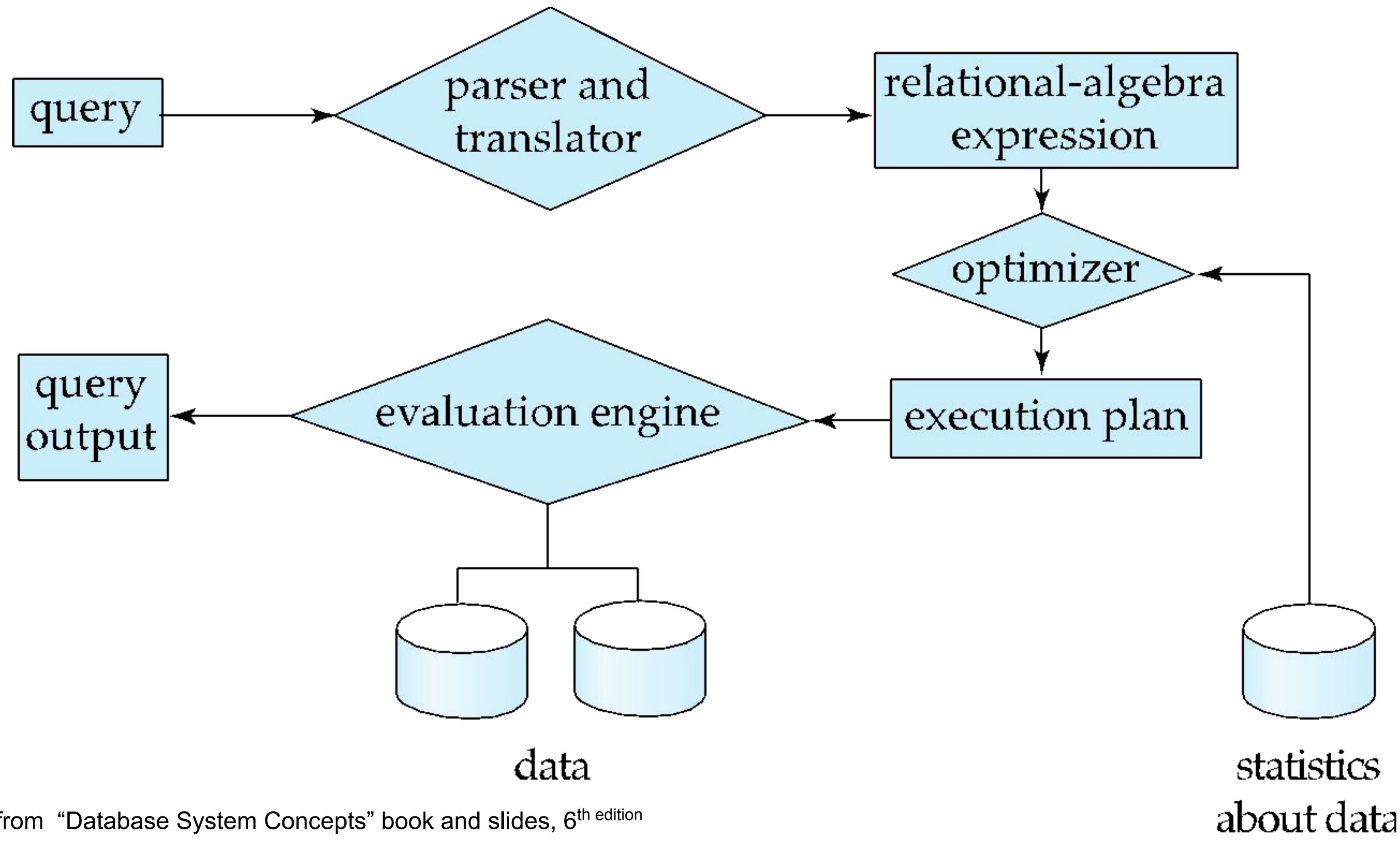
Any difference? Why?

Lecture Outline

- Review Hashing
- ***Overview of Query Processing***
- Selection

Overview of Query Processing

- Parsing and translation
- Optimization
- Evaluation



Based on and image from "Database System Concepts" book and slides, 6th edition

Basic Steps in Query Processing

- *Parsing*
  - check syntax
  - verify relations exist
- *Translation*
  - translate the query into its internal form
  - translate internal form into relational algebra
- *Evaluation*
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query

Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions

$$\sigma_{salary<75000}(\Pi_{salary}(instructor)) \text{ is equivalent to}$$
$$\Pi_{salary}(\sigma_{salary<75000}(instructor))$$

- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.

Basic Steps in Query Processing : Optimization

- Annotated expression specifying detailed evaluation strategy is called an evaluation-plan.
  - can use an index on salary to find instructors with salary < 75000,
  - or can perform complete relation scan and discard instructors with salary 75000

Based on and image from “Database System Concepts” book and slides, 6th edition

Basic Steps in Query Processing : Optimization

- *Evaluation primitive* – relational algebra operation annotated
- *Query evaluation plan* – Sequence of operations to be used for evaluating query
- *Query execution engine* -
  - Accepts plan
  - Executes plan
  - Returns a result

Basic Steps in Query Processing: Optimization

- Different evaluation plans – different costs
- Database system must construct most efficient query evaluation plan
- *Query Optimization* – Chooses evaluation plan with lowest cost
  - Cost is estimated using statistical information from the database catalog
    - number of tuples in each relation, size of tuples, etc.
- Once lowest cost plan chosen, query is evaluated and records returned

Basic Steps in Query Processing: Optimization

- We will study
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
- Read Chapter 14 (will not be covered in this class)
  - How to optimize queries, how to find an evaluation plan with lowest estimated cost

Measures of Query Cost

- Estimate cost of individual operations then combine for query evaluation plan cost
- Cost is generally measured as <u>total elapsed time</u> for answering query
  - Many factors contribute to time cost
    - *disk accesses*
    - *CPU*
    - or even network *communication*

Measures of Query Cost

- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - Cost to write a block is greater than cost to read a block
      - data is read back after being written to ensure that the write was successful

Measures of Query Cost

- For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for b block transfers plus S seeks

$$b * t_T + S * t_S$$

- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae

Measures of Query Cost

- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation

Measures of Query Cost

- Response Time – time it takes to execute the plan
  - Hard to estimate due to:
    - Dependence on contents of buffer when query execution begins
    - Dependence on how distributed in a multi-disk configuration
- Optimizers try to minimize resource consumption vs. response time

Profiling In-class

Let’s look at how MySQL profiles our queries:

```
SET profiling = 1;
```

```
Execute the following queries:
```

```
Select * from actors where first_name like 'Ko%'
or first_name like 'Wr%';
```

```
select a.first_name, a.last_name, r.role, m.name,
m.year from actors a, roles r, movies m where
a.id=r.actor_id and m.id=r.movie_id limit 500;
```

Read more `https://dev.mysql.com/doc/refman/5.7/en/show-profiles.html`

Profiling In-class

Let’s look at how MySQL profiles our queries:

```
Run the follow commands:
```

```
SHOW PROFILES;
```

Read more `https://dev.mysql.com/doc/refman/5.7/en/show-profiles.html`

Profiling In-class

You should see something like this:

```
mysql> SHOW PROFILES;
+----------+------------+-------------------------------------------------------------------------------------------
-----------------------------------------------+
| Query_ID | Duration   | Query
|
+----------+------------+-------------------------------------------------------------------------------------------
-----------------------------------------------+
|        1 | 0.00880925 | Select * from actors where first_name like 'Ko%' or first_name like 'Wr%'
|
|        2 | 0.00449275 | select a.first_name, a.last_name, r.role, m.name, m.year from actors a, roles r, movies m where
a.id=r.actor_id and m.id=r.movie_id limit 500 |
+----------+------------+-------------------------------------------------------------------------------------------
-----------------------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

Read more `https://dev.mysql.com/doc/refman/5.7/en/show-profiles.html`

Profiling In-class

Let's look at how MySQL profiles our queries:

```
SHOW PROFILE FOR QUERY 1;
```

```
+----------------------+----------+
| Status               | Duration |
+----------------------+----------+
| starting             | 0.000134 |
| checking permissions | 0.000018 |
| Opening tables       | 0.000032 |
| init                 | 0.000064 |
| System lock          | 0.000020 |
| optimizing           | 0.000022 |
| statistics           | 0.000263 |
| preparing            | 0.000037 |
| executing            | 0.000007 |
| Sending data         | 0.008110 |
| end                  | 0.000018 |
| query end            | 0.000019 |
| closing tables       | 0.000014 |
| freeing items        | 0.000029 |
| cleaning up          | 0.000024 |
+----------------------+----------+
15 rows in set, 1 warning (0.00 sec)
```

Read more `https://dev.mysql.com/doc/refman/5.7/en/show-profiles.html`

Profiling In-class

Let's look at how MySQL profiles our queries:

```
SHOW PROFILE FOR QUERY 2;
```

```
+----------------------+----------+
| Status               | Duration |
+----------------------+----------+
| starting             | 0.000133 |
| checking permissions | 0.000010 |
| checking permissions | 0.000005 |
| checking permissions | 0.000008 |
| Opening tables       | 0.000035 |
| init                 | 0.000044 |
| System lock          | 0.000028 |
| optimizing           | 0.000023 |
| statistics           | 0.000091 |
| preparing            | 0.000029 |
| executing            | 0.000007 |
| Sending data         | 0.003984 |
| end                  | 0.000013 |
| query end            | 0.000015 |
| closing tables       | 0.000015 |
| freeing items        | 0.000031 |
| cleaning up          | 0.000024 |
+----------------------+----------+
17 rows in set, 1 warning (0.00 sec)
```

Read more `https://dev.mysql.com/doc/refman/5.7/en/show-profiles.html`

Explain In-class

Let's look at how MySQL Explain can be used:

```
mysql> explain actors;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| id         | int(11)      | NO   | PRI | 0       |       |
| first_name | varchar(100) | YES  | MUL | NULL    |       |
| last_name  | varchar(100) | YES  | MUL | NULL    |       |
| gender     | char(1)      | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql> explain movies;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| id    | int(11)      | NO   | PRI | 0       |       |
| name  | varchar(100) | YES  | MUL | NULL    |       |
| year  | int(11)      | YES  |     | NULL    |       |
| rank  | float        | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql> explain roles;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| actor_id | int(11)      | NO   | PRI | NULL    |       |
| movie_id | int(11)      | NO   | PRI | NULL    |       |
| role     | varchar(100) | NO   | PRI | NULL    |       |
+----------+--------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

Read more `https://dev.mysql.com/doc/refman/5.7/en/using-explain.html`

Explain In-class

Let’s look at how MySQL Explain can be used:

```
mysql> explain Select * from actors where first_name like 'Ko%' or first_name like 'Wr%';
+----+-------------+--------+------------+-------+-------------------+-------------------+---------+------+------+----------+----------
-------------+
| id | select_type | table  | partitions | type  | possible_keys     | key               | key_len | ref  | rows | filtered | Extra
|
+----+-------------+--------+------------+-------+-------------------+-------------------+---------+------+------+----------+----------
-------------+
|  1 | SIMPLE      | actors | NULL       | range | actors_first_name | actors_first_name | 303     | NULL | 1322 |   100.00 | Using
index condition |
+----+-------------+--------+------------+-------+-------------------+-------------------+---------+------+------+----------+----------
-------------+
1 row in set, 1 warning (0.00 sec)

mysql> explain select a.first_name, a.last_name, r.role, m.name, m.year from actors a, roles r, movies m where a.id=r.actor_id and
m.id=r.movie_id limit 500;
+----+-------------+-------+------------+--------+---------------------------+----------+---------+-----------------+--------+---------
-+-------------+
| id | select_type | table | partitions | type   | possible_keys             | key      | key_len | ref             | rows   | filtered
| Extra       |
+----+-------------+-------+------------+--------+---------------------------+----------+---------+-----------------+--------+---------
-+-------------+
|  1 | SIMPLE      | m     | NULL       | ALL    | PRIMARY                   | NULL     | NULL    | NULL            | 392486 |   100.00
| NULL        |
|  1 | SIMPLE      | r     | NULL       | ref    | PRIMARY,actor_id,movie_id | movie_id | 4       | imdb.m.id       |      9 |   100.00
| Using index |
|  1 | SIMPLE      | a     | NULL       | eq_ref | PRIMARY                   | PRIMARY  | 4       | imdb.r.actor_id |      1 |   100.00
| NULL        |
+----+-------------+-------+------------+--------+---------------------------+----------+---------+-----------------+--------+---------
-+-------------+
3 rows in set, 1 warning (0.00 sec)
```

Read more `https://dev.mysql.com/doc/refman/5.7/en/using-explain.html`

Explain In-class

Let's look at how MySQL Explain can be used:

```
explain select actors.first_name, actors.last_name, roles.role,
movies.name, movies.year from actors, roles, movies  where
roles.actor_id=actors.id and roles.movie_id=movies.id and
movies.name like 'S%' and actors.first_name like 'Wr%';
```

```
+----+-------------+--------+------------+--------+--------------------------+-------------------+---------+-------------
--------+------+----------+-----------------------+
| id | select_type | table  | partitions | type   | possible_keys            | key               | key_len | ref
| rows | filtered | Extra                 |
+----+-------------+--------+------------+--------+--------------------------+-------------------+---------+-------------
--------+------+----------+-----------------------+
|  1 | SIMPLE      | actors | NULL       | range  | PRIMARY,actors_first_name | actors_first_name | 303     | NULL
|   20 |   100.00 | Using index condition |
|  1 | SIMPLE      | roles  | NULL       | ref    | PRIMARY,actor_id,movie_id | actor_id          | 4       |
imdb.actors.id      |    4 |   100.00 | Using index           |
|  1 | SIMPLE      | movies | NULL       | eq_ref | PRIMARY,movies_name       | PRIMARY           | 4       |
imdb.roles.movie_id |    1 |    18.07 | Using where           |
+----+-------------+--------+------------+--------+--------------------------+-------------------+---------+-------------
--------+------+----------+-----------------------+
3 rows in set, 1 warning (0.00 sec)
```

Read more `https://dev.mysql.com/doc/refman/5.7/en/using-explain.html`

Explain In-class

Let's look at how MySQL Explain can be used:

```
explain select actors.first_name, actors.last_name, roles.role,
movies.name, movies.year from actors, roles, movies  where
roles.actor_id=actors.id and roles.movie_id=movies.id and
movies.name like 'S%' limit 500;
```

```
+----+-------------+--------+------------+--------+-----------------------------+--------------+---------+-------------------
--+-------+----------+-----------------------+
| id | select_type | table  | partitions | type   | possible_keys               | key          | key_len | ref
| rows  | filtered | Extra                 |
+----+-------------+--------+------------+--------+-----------------------------+--------------+---------+-------------------
--+-------+----------+-----------------------+
|  1 | SIMPLE      | movies | NULL       | range  | PRIMARY,movies_name         | movies_name  | 303     | NULL
| 70086 |   100.00 | Using index condition |
|  1 | SIMPLE      | roles  | NULL       | ref    | PRIMARY,actor_id,movie_id | movie_id     | 4       | imdb.movies.id
|    10 |   100.00 | Using index           |
|  1 | SIMPLE      | actors | NULL       | eq_ref | PRIMARY                     | PRIMARY      | 4       |
imdb.roles.actor_id |     1 |   100.00 | NULL                  |
+----+-------------+--------+------------+--------+-----------------------------+--------------+---------+-------------------
--+-------+----------+-----------------------+
3 rows in set, 1 warning (0.00 sec)
```

Read more `https://dev.mysql.com/doc/refman/5.7/en/using-explain.html`

Explain In-class

Let's look at how MySQL Explain can be used:

```
explain select actors.first_name, actors.last_name, roles.role,
movies.name, movies.year from actors, roles, movies  where
roles.actor_id=actors.id and roles.movie_id=movies.id and
movies.name like 'S%' and actors.first_name like 'S%' limit 500;
```

```
+----+-------------+--------+------------+--------+----------------------------+--------------------+---------+-------------
--------+--------+----------+-----------------------+
| id | select_type | table  | partitions | type   | possible_keys              | key                | key_len | ref
| rows   | filtered | Extra                 |
+----+-------------+--------+------------+--------+----------------------------+--------------------+---------+-------------
--------+--------+----------+-----------------------+
|  1 | SIMPLE      | actors | NULL       | range  | PRIMARY,actors_first_name | actors_first_name | 303     | NULL
| 112800 |   100.00 | Using index condition |
|  1 | SIMPLE      | roles  | NULL       | ref    | PRIMARY,actor_id,movie_id | actor_id           | 4       |
imdb.actors.id      |      4 |   100.00 | Using index           |
|  1 | SIMPLE      | movies | NULL       | eq_ref | PRIMARY,movies_name        | PRIMARY            | 4       |
imdb.roles.movie_id |      1 |    18.07 | Using where           |
+----+-------------+--------+------------+--------+----------------------------+--------------------+---------+-------------
--------+--------+----------+-----------------------+
3 rows in set, 1 warning (0.00 sec)
```

Read more `https://dev.mysql.com/doc/refman/5.7/en/using-explain.html`

Lecture Outline

- Review Hashing and Indexing in Context
- Overview of Query Processing
- ***Selection***

Selection Operation

- ***File scan***
  - Lowest level operator to access data
  - Algorithms to locate/retrieve records
- Assuming tuples are stored in one file for a selection operation
- Let's look at each search algorithm

Selection Operation

- Algorithm **A1** (**linear search**)
  - Scan each file block and test all records to see whether they satisfy the selection condition
  - Initial seek required to access first block

Cost estimate = b_r block transfers + 1 seek

- b_r denotes number of blocks containing records from relation r

If selection is on a key attribute, can stop on finding record

- cost = (b_r /2) block transfers + 1 seek

Selection Operation

- Linear search can be applied to any file regardless of
  - selection condition or
  - ordering of records in the file, or
  - availability of indices
- Slower than other algorithms
- Note: binary search generally does not make sense since data is not stored consecutively
  - except when there is an index available,
  - and binary search requires more seeks than index search

Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index
- **A2** (**primary index, equality on key**)
  - Retrieve a single record that satisfies the corresponding equality condition
  - Equality comparison on key attribute with primary index
  - Use index to retrieve record that satisfies equality condition

$$Cost = (h_i + 1) * (t_T + t_S)$$

  - Can be used B+-Tree file organization to help access performance
    - Problems related to relocation and secondary indices

Based on and image from "Database System Concepts" book and slides, 6th edition

Selections Using Indices

- **A3** (**primary index, equality on nonkey**)
  - Retrieve multiple records using a primary index when selection condition specifies equality comparison on nonkey attribute
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records

$$Cost = h_i * (t_T + t_S) + t_S + t_T * b$$

Selections Using Indices

- **A4** (**secondary index, equality on nonkey**)
  - Retrieve a single record if the search-key is a candidate key

    $$Cost = (h_i + 1) * (t_T + t_S)$$

  - Retrieve multiple records if search-key is not a candidate key
    - each of *n* matching records may be on a different block

    $$\text{Cost} = (h_i + n) * (t_T + t_S)$$

    Can be very expensive!

Based on and image from "Database System Concepts" book and slides, 6th edition

Selections Involving Comparisons

- Selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear search,
  - or by using indices in the following ways:
- **A5** (**primary index, comparison**)
  - Relation is sorted on A
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple > v; do not use index

Selections Involving Comparisons

- **A6** (**secondary index, comparison**)
  - Use secondary ordered index for <,<=,>=,>
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry ≥ *v* and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry > *v*
  - In either case, retrieve records that are pointed to
    - requires an I/O for each record
    - Linear file scan may be cheaper

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta 1} \wedge_{\theta 2} \wedge \ldots_{\theta n}(r)$
- **A7** (**conjunctive selection using one index**)
  - Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta i}(r)$
  - Test other conditions on tuple after fetching it into memory buffer
- **A8** (**conjunctive selection using composite index**)
  - Use appropriate composite (multiple-key) index if available
  - Type of index determines whether A2, A3 or A4 will be used

Implementation of Complex Selections

- **A9** (**conjunctive selection by intersection of identifiers**)
  - Requires indices with record pointers
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory
  - Cost is sum of costs of each index scan + cost of retrieving records in intersection of retrieved lists of pointers , can retrieve records in sorted order

Implementation of Complex Selections

- **A10** (**disjunctive selection by union of identifiers**)
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file

Negation: $\sigma_{\neg\theta}(r)$

- Use linear scan on file
- If very few records satisfy $\neg\theta$, and an index is applicable to θ
  - Find satisfying records using index and fetch from file