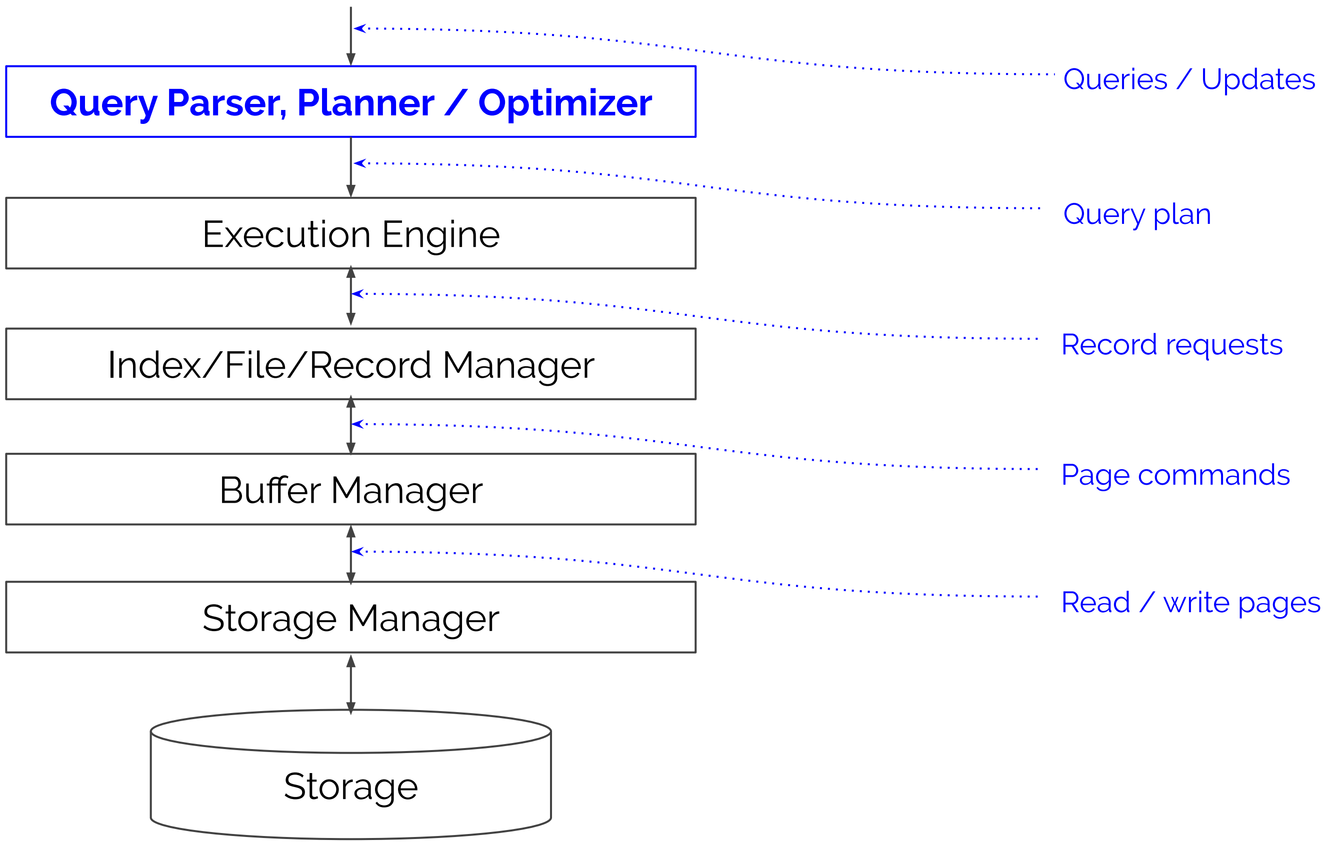
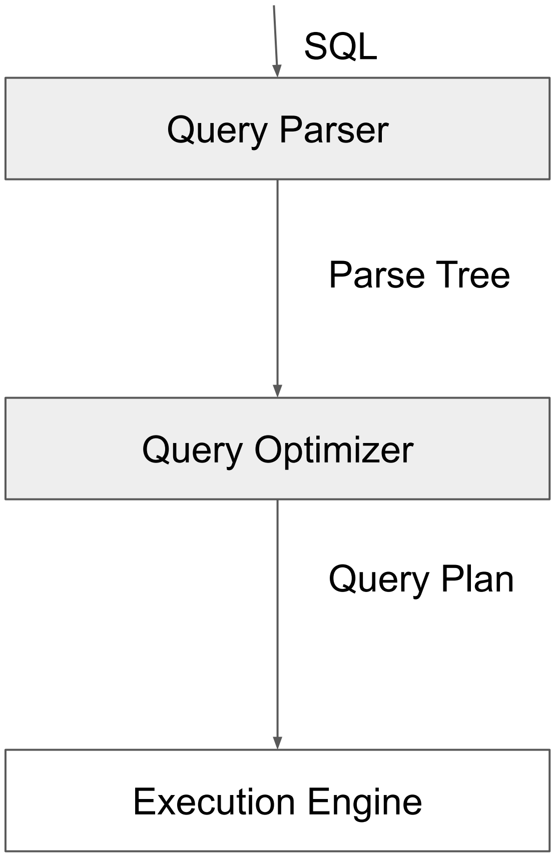
# Query Planning, Index

CSC 365

# DBMS System Components



# Query Parser, Optimizer



# Query Parser

Two main parts

- Lexical Parser
  - Parse the entire query into tokens
- Grammar Rule Module
  - Apply grammar rules
  - Produces a parse tree

# Query Parser

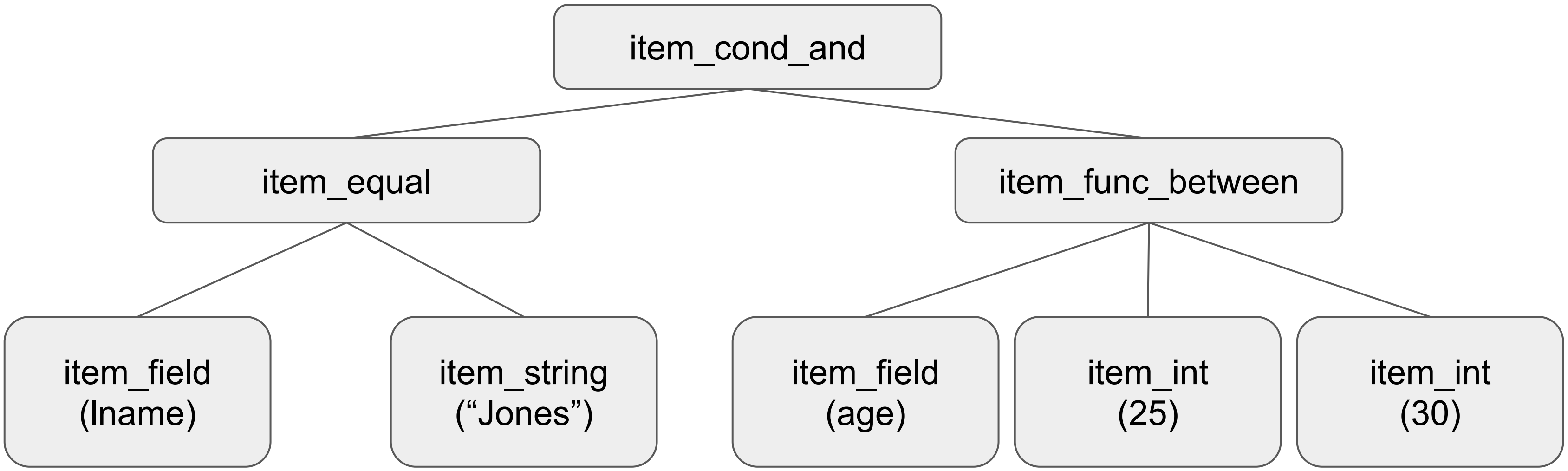
Two main objectives

- The parser must be lightening fast.
- The generated parse tree must provide the information to the optimizer in a way that permits it to access the data efficiently.

# An Example Parse Tree for Typical WHERE clause

SELECT COUNT(*) FROM customer

WHERE lname='Jones' AND age BETWEEN 25 AND 30;



# Query Optimizer

The optimizer is the set of routines which decide what execution path the DBMS should take for queries.

# Query Optimizer

# Query Optimizer


Consider the following query:

```
SELECT c.first_name,c.last_name,c.phone,p.name,p.price
    FROM customer c,orders o, product p
    WHERE c.id = o.customer_id AND o.product_id = p.id
            AND o.payment_status = 'FAILED'
    ORDER BY c.last_name,c.first_name
```

# Naive Approach

- For each record in customer as c
  - For each record in orders as o
    - For each record in product as p
      - Check if the combination of c, o, p matches the WHERE clause

10,000 10,000 10,000

**10,000 x 10,000 x 10,000 = 1 trillion!**

# An Optimization Example

With a processor capable of examining 1 million records per second, the query would take 1 million seconds, or more than 11 days.

# Query Optimizer

# Query Optimizer


Consider the following query:

```
SELECT c.first_name,c.last_name,c.phone,p.name,p.price
    FROM customer c,orders o, product p
    WHERE c.id = o.customer_id AND o.product_id = p.id
            AND o.payment_status = 'FAILED'
    ORDER BY c.last_name,c.first_name
```

# An Optimization Example

Let’s take an advantage of the fact that

- we have keys on customer.id, orders.payment_status, and product.id, and that
- The keys customer.id and product.id are unique.

Then, we can potentially eliminate a lot of records on orders.payment_status.

# An Optimization Example

1. SELECT orders o WHERE o.payment_status = 'FAILED'
2. For each matching record of orders,
   a. Select a record out of customer using the key on id
   b. Select a record out of product using the key on id

Even if the step 1 returns 10,000 records,
we only have to examine 10,000 records,

Which is much less than 1 trillion!

# An Optimization Example

- The use of keys may increase the amount of time needed to create each combination, this overhead in the end was worthwhile.

# An Optimization Example

- The use of keys may increase the amount of time needed to create each combination, this overhead in the end was worthwhile.
- According to the standard MySQL optimizer cost estimate model, each key access takes three times as long for the same table as the scan access.

# An Optimization Example

- Suppose that the naïve approach for creating a record combination costs 1+1+1=3

# An Optimization Example

- Suppose that the naïve approach for creating a record combination costs 1+1+1=3
- Then, the improved approach for the same operation costs three times more: 3+3+3=9.

# An Optimization Example

- Suppose that the naïve approach for creating a record combination costs 1+1+1=3
- Then, the improved approach for the same operation costs three times more: 3+3+3=9.
- With a processor capable of examining 1 million records per second, we can now process only 333,333 combinations per second, instead of 1 million.

# An Optimization Example

However, we now need to process no more than 10,000 of them, and the optimized query should take less than 0.03 seconds, down from 11 days.

# JOIN ORDER is important for Optimization!

# Important Tasks of MySQL Optimizer 1/2

- **Determine which keys can be used to retrieve the records from tables, and choose the best one for each table.**

# Important Tasks of MySQL Optimizer 1/2

- Determine which keys can be used to retrieve the records from tables, and choose the best one for each table.
- **For each table, decide whether a table scan is better than reading on a key. If there are a lot of records that match the key value, the advantages of the key are reduced and the table scan becomes faster.**

# Important Tasks of MySQL Optimizer 1/2

- Determine which keys can be used to retrieve the records from tables, and choose the best one for each table.
- For each table, decide whether a table scan is better than reading on a key. If there are a lot of records that match the key value, the advantages of the key are reduced and the table scan becomes faster.
- **Determine the order in which tables should be joined when more than one table is present in the query.**

# Important Tasks of MySQL Optimizer 1/2

- Determine which keys can be used to retrieve the records from tables, and choose the best one for each table.
- For each table, decide whether a table scan is better than reading on a key. If there are a lot of records that match the key value, the advantages of the key are reduced and the table scan becomes faster.
- Determine the order in which tables should be joined when more than one table is present in the query.
- **Rewrite the WHERE clause to eliminate dead code, reducing the unnecessary computations and other optimizations to open the way for using keys.**

# Important Tasks of a Typical Optimizer 2/2

- **Eliminate unused tables from the join.**

# Important Tasks of a Typical Optimizer 2/2

- Eliminate unused tables from the join.
- **Determine whether keys can be used for ORDER BY and GROUP BY.**

# Important Tasks of a Typical Optimizer 2/2

- Eliminate unused tables from the join.
- Determine whether keys can be used for ORDER BY and GROUP BY.
- **Attempt to replace an outer join with an inner join.**

# Important Tasks of a Typical Optimizer 2/2

- Eliminate unused tables from the join.
- Determine whether keys can be used for ORDER BY and GROUP BY.
- Attempt to replace an outer join with an inner join.
- **Attempt to simplify subqueries, as well as determine to what extent their results can be cached.**

# MySQL Query Optimizer

- Every query is treated as a set of JOINs.
- The term JOIN is used more broadly here:
  - A query on only one table is a degenerate JOIN

# MySQL Query Optimizer

1. The optimizer determines the best join order.
2. The optimizer does a nested loop to accomplish the join.

The optimizer selects the record access methods and puts the tables in an order it believes would minimize the cost.

# Optimizing WHERE Clause

The optimizer performs optimization of conditions in WHERE clause.

For example, in MySQL:

- Removal of unnecessary parentheses
- Constant folding
    - `(a<b AND b=c) AND a=5`
    - `-> b>5 AND b=c AND a=5`
- Constant condition removal
    - `(b>=5 AND b=5) OR (b=6 AND 5=5) OR (b=7 AND 5=6)`
    - `-> b=5 OR b=6`
- Early detection of invalid constant expressions.

# The Problem of Query Optimization

- Find the best access paths for each table
- Find the best join order
- Do all of the above in a short amount of time.

# Cost-Based Optimizer

A **cost-based optimizer** estimates cost of each query plan based on the *current state* of the database. This takes into account statistics maintained internally by the database, such as:

- Number of rows in each table
- Distribution of column values (ie. represented as a histogram)
- Table access methods available (full scan, index scan, range scan, etc.)

# Cost Model

The **cost model** used by the database is an important component of cost-based optimization.

Traditional, disk-based, databases typically use a cost model focused on I/O, based on the assumption that CPU usage will be negligible compared to the cost of accessing data on disk.

Main memory databases must take into account CPU and RAM access time.

# Peeking at RDBMS Query Plans

Most RDBMSs include commands or tools to analyze query plans

In the MySQL command line, we use EXPLAIN. We can prefix a statement with EXPLAIN to see detail about the query execution plan.

```
EXPLAIN SELECT * FROM Student WHERE MajorCode = 'CSC'
```

# MySQL EXPLAIN

EXPLAIN displays information from the query optimizer about the execution plan. In other words: *how* MySQL would process the statement.

[Full documentation]()

| Column(s) in EXPLAIN Output | Brief Description |
|---|---|
| id | The `SELECT` identifier (table or alias) |
| select_type | Type of `SELECT (SIMPLE, PRIMARY / SUBQUERY, UNION, etc.)` |
| table / partitions | Underlying table and table partitions (if any) |
| type | Physical join type (ALL / full table scan, range, index) |
| possible_keys / key / ken_len / ref | Detail about indexes: available indexes, the chosen index, and key length (if any), columns referenced by the chosen. key/index |
| rows / filtered | Estimated number of rows to be examined, and percent that will be filtered by table conditions |
| extra | Additional information about this line in the query plan |

# Using EXPLAIN to understand the Optimizer

```
SELECT m.title, g.genre
FROM Movies m, Genre g, StarsIn s
WHERE m.mid = g.mid
 AND g.genre = 'Sci-Fi'
 AND m.mid = s.mid
 AND s.sname = 'Will Smith'
```

| title | genre |
|---|---|
| Men in Black 3 | Sci-Fi |
| Wild Wild West | Sci-Fi |
| Suicide Squad | Sci-Fi |
| I Am Legend | Sci-Fi |
| Men in Black II | Sci-Fi |
| After Earth | Sci-Fi |
| I, Robot | Sci-Fi |
| Men in Black | Sci-Fi |

8 rows in set (0.03 sec)

# Using EXPLAIN to understand the Optimizer

```
EXPLAIN SELECT m.title, g.genre
FROM Movies m, Genre g, StarsIn s
WHERE m.mid = g.mid
 AND g.genre = 'Sci-Fi'
 AND m.mid = s.mid
 AND s.sname = 'Will Smith' \G
```

The \G switch tells MySQL to display the result set vertically.

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 52
          ref: const
         rows: 12
     filtered: 100.00
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: tkuboi.s.mid
         rows: 1
     filtered: 100.00
        Extra: NULL
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 56
          ref: tkuboi.s.mid,const
         rows: 1
     filtered: 100.00
        Extra: Using index
```

# Using EXPLAIN to understand the Optimizer

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 52
          ref: const
         rows: 12
     filtered: 100.00
        Extra: Using index
```

StarsIn table will be examined first.

# Using EXPLAIN to understand the Optimizer

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 52
          ref: const
         rows: 12
     filtered: 100.00
        Extra: Using index
```

The query does not use UNION or subqueries.

# Using EXPLAIN to understand the Optimizer

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 52
          ref: const
         rows: 12
     filtered: 100.00
        Extra: Using index
```

There is an option to read either Primary Key or mid.

# Using EXPLAIN to understand the Optimizer

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 52
          ref: const
         rows: 12
     filtered: 100.00
        Extra: Using index
```

The Primary Key is chosen.

# Using EXPLAIN to understand the Optimizer

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 52
          ref: const
         rows: 12
     filtered: 100.00
        Extra: Using index
```

The result may contain multiple records.

# Using EXPLAIN to understand the Optimizer

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 52
          ref: const
         rows: 12
     filtered: 100.00
        Extra: Using index
```

The first 52 bytes of the sname field will be used.

# Using EXPLAIN to understand the Optimizer

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 52
          ref: const
         rows: 12
     filtered: 100.00
        Extra: Using index
```

The key value used is a constant supplied in the WHERE clause.

# Using EXPLAIN to understand the Optimizer

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 52
          ref: const
         rows: 12
     filtered: 100.00
        Extra: Using index
```

The estimated number of records retrieved is 12.

# Using EXPLAIN to understand the Optimizer

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 52
          ref: const
         rows: 12
     filtered: 100.00
        Extra: Using index
```

The optimizer has decided that reading only the value of index without reading the entire tuple is sufficient.

# Using EXPLAIN to understand the Optimizer

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 52
          ref: const
         rows: 12
     filtered: 100.00
        Extra: Using index
```

An estimated percentage of table rows that will be filtered by the table condition.

# Using EXPLAIN to understand the Optimizer

```
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: tkuboi.s.mid
         rows: 1
     filtered: 100.00
        Extra: NULL
```

Movies table will be examined second.

# Using EXPLAIN to understand the Optimizer

```
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: tkuboi.s.mid
         rows: 1
     filtered: 100.00
        Extra: NULL
```

There is only one possible key to be used.

# Using EXPLAIN to understand the Optimizer

```
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: tkuboi.s.mid
         rows: 1
     filtered: 100.00
        Extra: NULL
```

The Primary Key will be used and one record per given value is possible.

# Using EXPLAIN to understand the Optimizer

```
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: tkuboi.s.mid
         rows: 1
     filtered: 100.00
        Extra: NULL
```

The value of the mid read previously from the StarsIn table is used for the key lookup.

# Using EXPLAIN to understand the Optimizer

```
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 56
          ref: tkuboi.s.mid,const
         rows: 1
     filtered: 100.00
        Extra: Using index
```

Genre table will be examined last.

# Let’s force optimizer to use the specified join order

```
SELECT m.title, g.genre
FROM Movies m
STRAIGHT_JOIN Genre g
STRAIGHT_JOIN StarsIn s
WHERE m.mid = g.mid
 AND g.genre = 'Sci-Fi'
 AND m.mid = s.mid
 AND s.sname = 'Will Smith';
```

```
*************************** 1. row *****
           id: 1
  select_type: SIMPLE
        table: m
   partitions: NULL
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 482
     filtered: 100.00
        Extra: NULL
*************************** 2. row *****
           id: 1
  select_type: SIMPLE
        table: g
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 56
          ref: tkuboi.m.mid,const
         rows: 1
     filtered: 100.00
        Extra: Using index
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 56
          ref: const,tkuboi.m.mid
         rows: 1
     filtered: 100.00
        Extra: Using index
```

Table Scan.

No Key used.

482 records.

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
   partitions: NULL
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 482
     filtered: 100.00
        Extra: NULL
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 56
          ref: tkuboi.m.mid,const
         rows: 1
     filtered: 100.00
        Extra: Using index
```

1 record.

```
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 56
          ref: const,tkuboi.m.mid
         rows: 1
     filtered: 100.00
        Extra: Using index
```

1 record.

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 52
          ref: const
         rows: 12
     filtered: 100.00
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: tkuboi.s.mid
         rows: 1
     filtered: 100.00
        Extra: NULL
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 56
          ref: tkuboi.s.mid,const
         rows: 1
     filtered: 100.00
        Extra: Using index
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
   partitions: NULL
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 482
     filtered: 100.00
        Extra: NULL
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 56
          ref: tkuboi.m.mid,const
         rows: 1
     filtered: 100.00
        Extra: Using index
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY,mid
          key: PRIMARY
      key_len: 56
          ref: const,tkuboi.m.mid
         rows: 1
     filtered: 100.00
        Extra: Using index
```

# Let's force optimizer to use the primary key

```
SELECT m.title, g.genre
FROM Movies m FORCE KEY(PRIMARY)
STRAIGHT_JOIN Genre g STRAIGHT_JOIN StarsIn s
WHERE m.mid = g.mid
 AND g.genre = 'Sci-Fi'
 AND m.mid = s.mid
 AND s.sname = 'Will Smith';
```

# Cost : Optimized v.s. Un-Optimized

- Optimized
  - JOIN ORDER: StarsIn, Movies, Genre
  - Examines: 12 x 1 x 1 = 12 records
- Un-Optimized
  - JOIN ORDER: Movies, Genre, StarsIn
  - Examines: 482 x 1 x 1 = 482 records

# MySQL EXPLAIN

Some optimizations are expensive and fairly uncommon. Those may indicate that the query, without it, would be a complete performance disaster.

Example: ***range checked for each record: (index map: N)*** in ***Extra*** field.

It should be considered as an invitation to write a better query.

# Using EXPLAIN to Optimize Queries

Three general options:

1. Revise your SQL query
2. Change structure of underlying data
3. Create or change table **indexes**

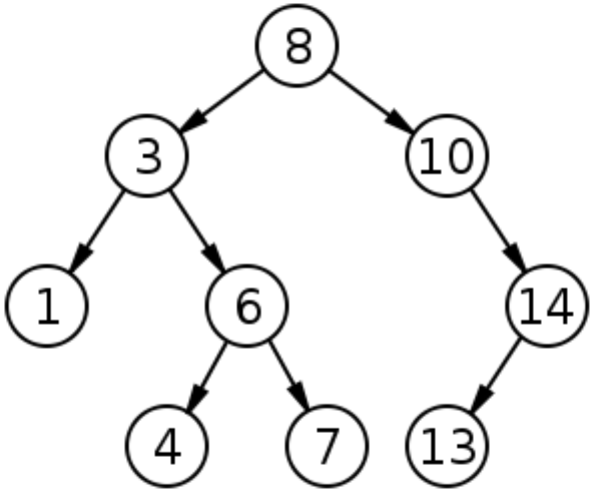
# Indexes

An **index** on a single attribute A or a set of attributes $A_1,\ldots A_n$ is a data structure that allows a database to quickly find tuples that have a certain value for the indexed attribute(s)

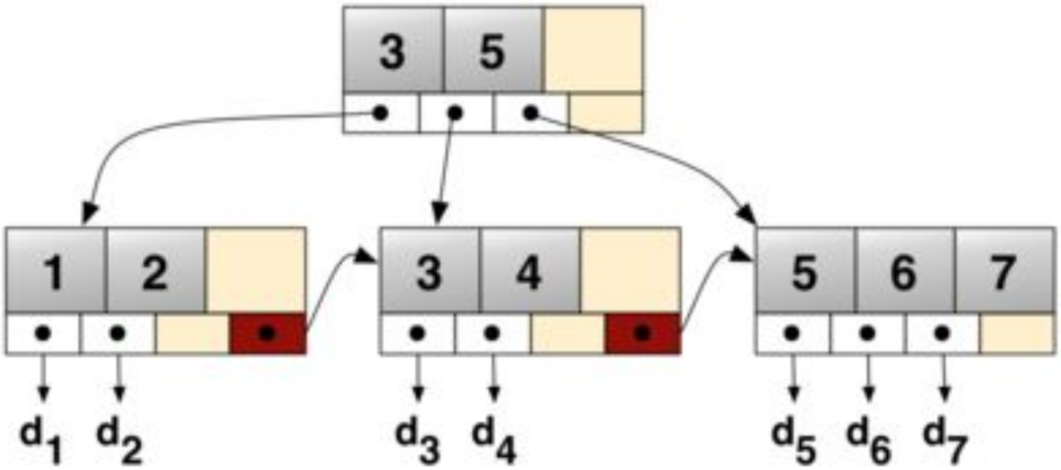
- Indexes can make some operations very efficient:
  - [In]equality conditions
    - WHERE SKU = 'PL122'
    - WHERE Flavor = 'Chocolate' AND Price >= 1.0
  - Range condition,   WHERE mpg BETWEEN 40 AND 55
  - Equijoins

Often implemented using a *B-Tree* data structure (a binary tree in which a node can have more than two children)

# Binary Search Tree vs. B+ Tree




Binary Search Tree




B+ Tree, keys 1-7 point to data items $d_1$ - $d_7$

# Indexes

Every table has a single **primary index** (typically, but not always, defined on the primary key). The primary index controls the physical storage order of records.

In MySQL, primary key will be always indexed.

A table may also have any number of *secondary indexes*

# Index Creation

Most RDBMSs support syntax similar to:

CREATE INDEX <index name> ON <table name> (<column(s)>);

Example:

CREATE INDEX my_date_idx ON bigbank (post_date);

DROP INDEX my_date_idx ON bigbank;

MySQL reference documentation

# Indexes - Tradeoffs

An index can make queries more efficient. Why not index every column in every table?  Because indexes carry costs, namely:

- Indexes consume storage space (possibly in-memory)
- Each index must be maintained as data changes (INSERT, UPDATE, DELETE)

Carefully choose indexes based on known query access patterns and/or profiling results.

# Indexes - Tradeoffs

- Query
  - One extra disk access needed to read a page of the index
  - But that may reduce the number of pages needed to read to answer the query
- Insert / Update / Delete
  - One extra disk access needed to read a page of index
  - Another extra disk access needed to write back the page
  - These are necessary for each index

# Indexes - Tradeoffs

- Suppose the cost of the Query without index is ***CQo***, with index is ***CQi***,
- the cost of Insert/Update without index is ***CIo***, with index is ***CIi***,
- the percentage of the time the Query is done is ***P***,
- the percentage of the time Insert/Update is done is **1 - *P***,
- creating an index pays off if:
  - ***CQo x P + CIo x (1 - P) > CQi x P + CIi x (1 - P)***

# Summary

- Query Optimizer
  - Find the best access paths for each table
  - Find the best join order
- Indexes
  - Primary index
  - Secondary indexes
  - Beware of Trade-offs