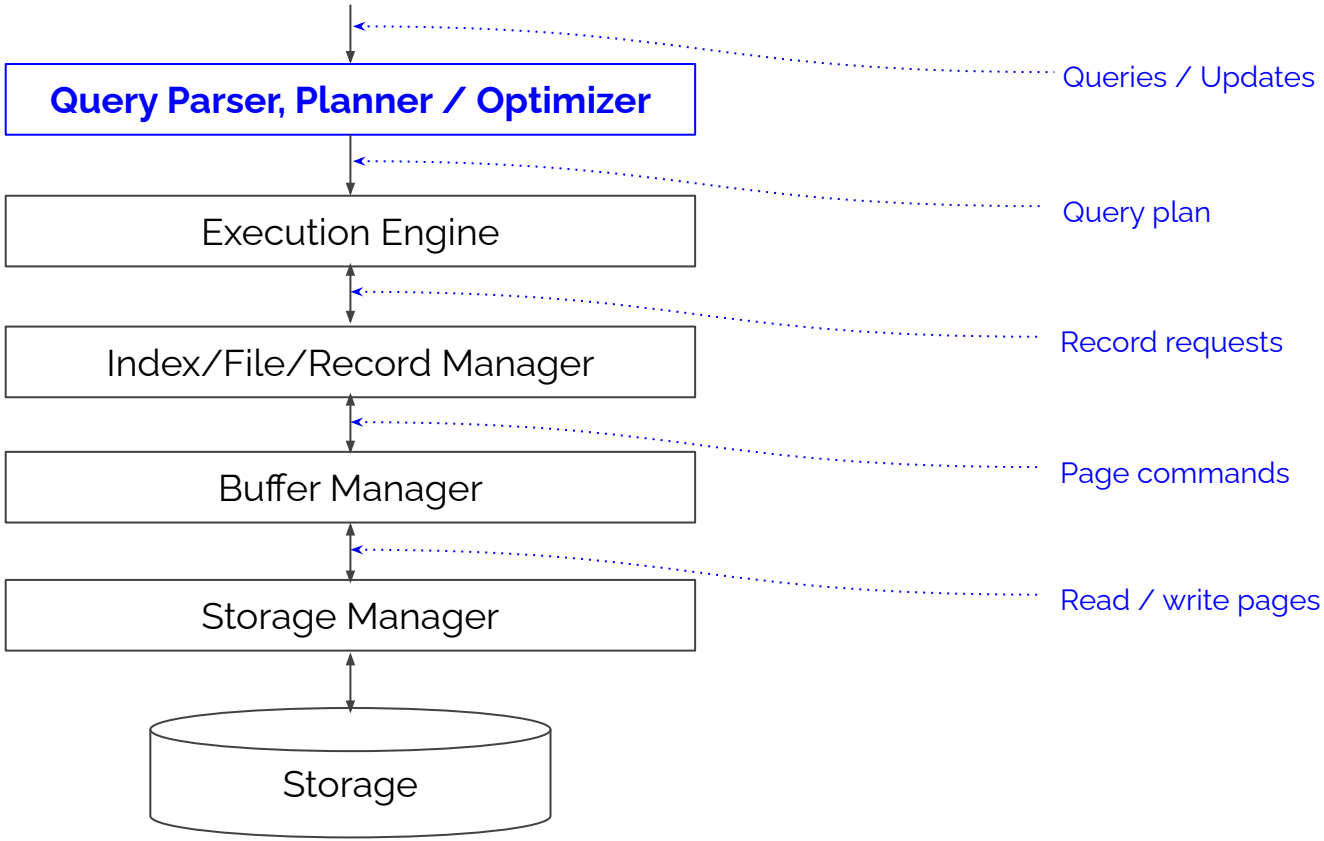
CSC 365

Introduction to Database Systems

DBMS System Components

CAL POLY



Relational Algebra / Logical Query Plan

We can represent a query as a tree of relational algebra operators. We have also seen that a single query may have multiple (equivalent) representations, based on transformation rules

- Join ordering
- Algebraic laws

Consider four tables: A, B, C, D.

What are the possible ways to perform a natural join between these four tables?

Join Order - Dynamic Programming

Suppose we want to join $R_1 \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n$

We construct a table with an entry for each of the $2^n - 1$ subsets of one or more of the *n* tables.

Table Entries: join expression and cost of computing the join. In the simple case: the cost is the sum of the sizes of intermediate results.
Base Case: Entry for each relation R, a cost of zero and a formula that is just R itself.
Solution: The expression that joins all tables with the lowest cost.

Join Order - Dynamic Programming

1) Consider individual tables

2) Consider pairs of joined tables, find least cost way to join, record in table.

3) Consider all combinations of three tables, find least cost way to join.

...

Continue until you have considered all combinations of *n* tables. Choose the join expression with the lowest cost (cost function, in simple case, could be the sum of *predicted* intermediate result sizes.)

Join Order - Example

CAL POLY

R(A, B) S(B, C) T(C, D)

R has 3 tuples, S has 6 tuples, T has 5 tuples

R.B is a foreign key referencing S.B, S.C is a foreign key referencing T.C

R ⋈ S ⋈ T

Subset of Relations	{R}	{S}	{T}	**{R,S}**	{S,T}	{R,T}	{R,S,T}
Size	3	6	5	**3**	6	15	3
Cost (sum of *intermediate* sizes)	**0**	**0**	**0**	**0**	**0**	**0**	3
Best Plan	R	S	T	**R ⋈ S**	S ⋈ T	R ⋈ T	(R ⋈ S) ⋈ T

Cost computed as zero when there are no intermediate results

Since there are no common attributes, R ⋈ T = R x T

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. histogram)
- Table access methods available (full scan, index scan, range scan, etc.)

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.

Physical Plan

For each logical query plan, one or more **physical plan** exists. A physical plan is expressed in terms of physical operators, which include:

- Sequential (Full Table) Scan
- Index Scan
- Nested Loop Join
- Sort-Merge Join
- Hash Join
- Sort

Logical / Physical Plan

Consider the following query:

```
-- Monday(s) on which more than 2 Apple-flavored pastries were sold (BAKERY)
SELECT R.SaleDate, COUNT(*)
FROM receipts AS R, items AS I, goods AS G
WHERE R.RNumber = I.Receipt AND I.Item = G.GId
AND DAYNAME(R.SaleDate) = 'Monday' AND G.Flavor = 'Apple'
GROUP BY R.SaleDate
HAVING COUNT(*) > 2
ORDER BY R.SaleDate
```

Possible logical query plan(s)? physical plan(s)?

Analyzing RDBMS Query Plans

Most database systems include commands or tools to analyze physica query plans

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

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

MySQL EXPLAIN

CAL POLY

`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

Query Variations

Different query variations have different estimated costs.

Cost depends on:

- Query structure
- Capabilities and configuration of RDBMS optimizer
- Table structures & existing *data*

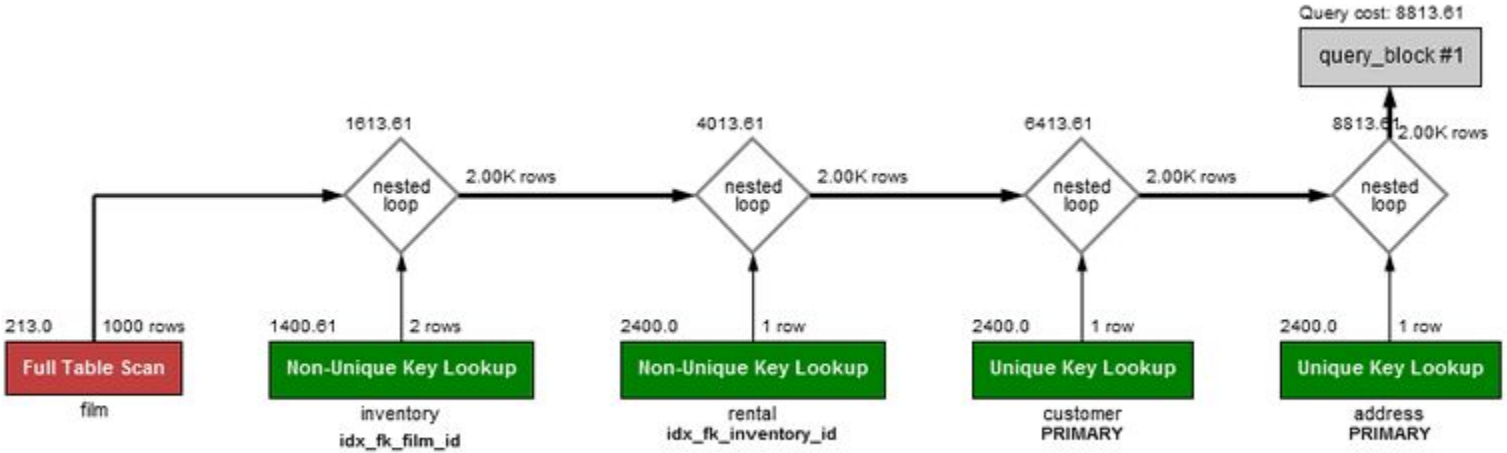
MySQL: EXPLAIN FORMAT=JSON

In addition to the table-based view, MySQL offers a JSON-formatted plan that includes additional detail.

```
EXPLAIN FORMAT=JSON
SELECT * FROM Student WHERE MajorCode = 'CSC';
```

Visual Query Plan

CAL POLY



https://dev.mysql.com/doc/workbench/en/wb-performance-explain.html

Using EXPLAIN to Optimize Queries

Three general options:

1. Revise your SQL query
2. **Create or change table indexes**
3. Change structure of underlying data
   (possibly using a different database system or storage engine)

Indexes

An **index** on a single attribute A or a set of attributes $A_1,...A_n$ 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

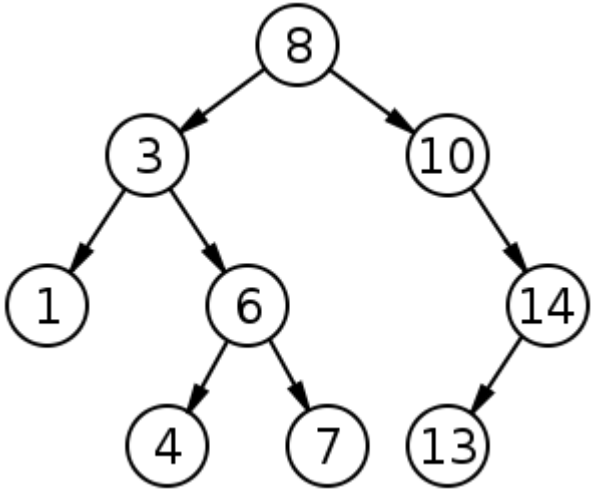
Often implemented using a *B-Tree* data structure

B-Tree / B+ Tree

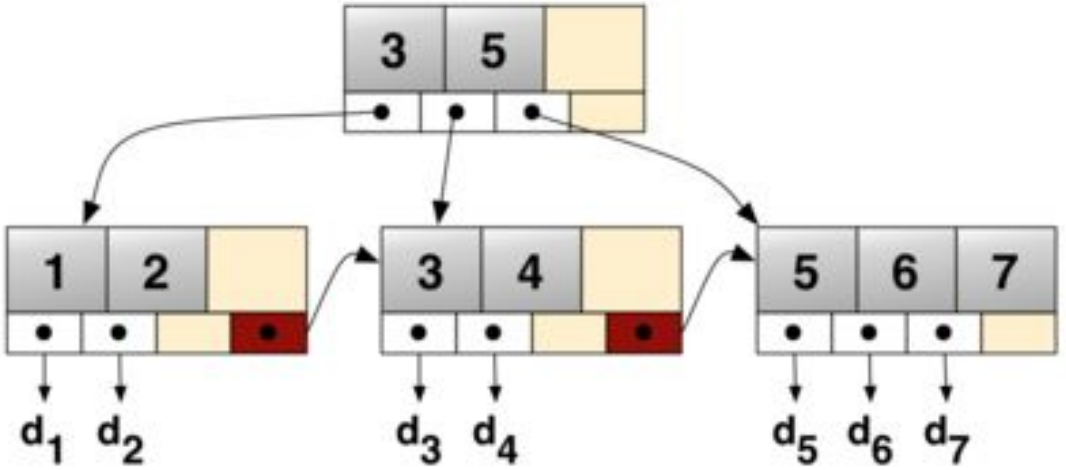
- B-Tree
  - Self-balancing
  - Maintains sort order
  - O(n) space, O(log n) search/insert/delete operations
  - > 2 children per node
  - Ideal for block-oriented storage (filesystems, databases)
- B+ Tree
  - Extension of B-Tree data structure
  - Leaves linked together in a linked list (no substantial increase in space usage or maintenance cost)

Binary Search Tree vs. B+ Tree

CAL POLY



Binary Search Tree



B+ Tree, keys 1-7 point to data items (rows) d_1 - d_7

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

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.

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

To view index details in MySQL:

```
SHOW INDEX FROM <table name>;
```

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 / memory space
- Each index must be maintained as data changes (`INSERT`, `UPDATE`, `DELETE`)

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

Improving Query Performance

Three general options:

1. Revise your SQL query
2. Create or change table indexes
3. Change structure of underlying data
    a. Normalization/denormalization
    b. Different database system or different storage engine

Relational Division

Find bakery customers who have purchased *every* flavor of Danish.

Possible approaches in SQL:

1. Joins (Lab 4)
2. Group by / Having (Lab 5)
3. Subqueries (Lab 6)