# DB Algorithms: Accessing Data

# Basic Architecture

- ~~Application Layer - what most users see, talks SQL~~
- Parsing/Planning Layers - the intelligence
- Runtime or execution Layer - the brawn - NEXT
- ~~Storage Layer - where data resides, may include simple access layer~~

Applications

Parsing

Planning

Processing

Data Access

Data in SSD/HDD

# What do we have?

- Tables stored mostly as B+ Tree with a primary index
  - May be stored as Heap File with no primary index
  - May be stored using an Extendible Hashing structure
  - May be stored in columnar or hybrid format
  - May have multiple copies stored in different storage methods
- Secondary Indexes (Unique and Non-unique) - look like tables internally
  - This is an additional “sub-table” or structure
  - B+ Tree or Hash structure
- (Aggregate) Join Indexes, Materialized Views - may look like tables internally

# Simplest Query

```
db1=# select * from students;
   sid    | last_name  | first_name | status
---------+------------+------------+--------
 2016001 | Shatdal    | Ambuj      |      5
 2015001 | Lincoln    | Abraham    |      1
 2014101 | Obama      | Barack     |      2
 2012144 | Bush       | George     |      4
 2012101 | Washington | George     |      3
(5 rows)
```

- Simply scan the table students to get all rows all columns

```
db1=# explain select * from students;
                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on students  (cost=0.00..1.05 rows=5 width=122)
(1 row)
```

# WHERE clause

```
db1=# select * from students where last_name = 'Obama';
   sid    | last_name | first_name | status
----------+-----------+------------+--------
 2014101 | Obama     | Barack     |      2
(1 row)
```

- Scan the table students to get rows
- Apply condition and keep the rows for which condition is true

```
db1=# explain select * from students where last_name = 'Obama';
                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on students  (cost=0.00..1.06 rows=1 width=122)
   Filter: ((last_name)::text = 'Obama'::text)
(2 rows)
```

# WHERE clause

- No matter how complex the condition (excluding subqueries) on a single table, it can always be evaluated by looking at all rows
- This is the default method
- Subqueries are a different beast altogether

# Equality Conditions

- WHERE c1 = <value1> [… and c2 = <value 2> …]
- 4 possibilities for the column(s)
  - primary index column(s) are a subset of the column(s)
  - There is a unique SI on a subset of the column(s)
  - There is a non unique SI on a subset of the column(s)
  - None of the above

# Equality Condition - PI Access Path

- If there is a PI available, one could go through the primary index to find the rows
  - Look up the value in the B+ Tree index
  - Find the leaf node
  - Find the row(s) containing the value
  - apply any “residual conditions”
  - Return the qualifying row(s)

# Residual Conditions

- Conditions other than the ones that apply to the index being used

# PI Access Path Example

```
db1=# select * from courses where dept = 'CS' and cid = 564;
 dept | cid | name
------+-----+------
 CS   | 564 | DBMS
(1 row)
```

- If there WHERE has equality on the primary index (primary key) then we could also use the index to locate the row(s)

```
db1=# explain select * from courses where dept='CS' and cid = 564;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using courses_pkey on courses  (cost=0.15..8.17 rows=1 width=98)
   Index Cond: (((dept)::text = 'CS'::text) AND (cid = 564))
(2 rows)
```

# Equality Condition - USI Access Path

- If there is a USI available, one could go through the secondary index to find the rows
  - Look up the value in the index
  - Find the leaf/data node
  - Find the *rid* of the row containing the value
  - Look up the row using the *rid*
  - Apply any residual conditions
  - Return the qualifying row, if any

# Equality Condition - NUSI Access Path

- If there is a NUSI available, one could go through the secondary index to find the rows
    - Look up the value in the index
    - Find the leaf/data node
    - Find the *rids* of the rows containing the value
        - this could be a bitmap, for example
    - Sort the *rids*
    - Look up the rows using the *rids*
    - Apply any residual conditions
    - Return the qualifying row, if any

# NUSI Access Path Example

```
hw2=# select * from sales where dept = 1;
 store | dept |  weekdate   | weeklysales
-------+------+-------------+-------------
     1 |    1 | 2010-02-05 |     24924.5
     1 |    1 | 2010-02-12 |     46039.5
     etc…
```

- If there WHERE has equality on the primary index (primary key) then we could also use the index to locate the row(s)

```
hw2=# explain select * from sales where dept = 1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on sales  (cost=129.24..2493.43 rows=6815 width=16)
   Recheck Cond: (dept = 1)
   ->  Bitmap Index Scan on salesdept  (cost=0.00..127.53 rows=6815 width=0)
         Index Cond: (dept = 1)
(4 rows)
```

# No Index - Scan and Evaluate

```
db1=# select * from students where last_name = 'Obama';
   sid    | last_name | first_name | status
----------+-----------+------------+--------
 2014101 | Obama     | Barack     |      2
(1 row)
```

- Scan the table students to get rows
- Apply condition and keep the rows for which condition is true

```
db1=# explain select * from students where last_name = 'Obama';
                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on students  (cost=0.00..1.06 rows=1 width=122)
   Filter: ((last_name)::text = 'Obama'::text)
(2 rows)
```

# Multiple NUSI’s

- WHERE C1 = ‘A’ and C2 = 10
- Both are non-unique and both have a NUSI
- Lookup both the indexes
- AND the *rid*-lists (bitmap or otherwise)
- Sort the *rids*
- Look up the rows and apply residual conditions

# SQL Operations

- SELECT query blocks
- Optional UNION, INTERSECT, EXCEPT operations on them

# SQL Operations - SELECT block

- Select rows from one or more tables
- If more than 1 table, must join (or cross join) them
  - Can have fairly involved WHERE conditions
- Optionally aggregate the rows - GROUP BY
  - Evaluate Having - can get complex
- Optionally evaluate ordered-analytic (window) functions
- Optionally do a DISTINCT