Advanced Database Systems

Spring 2024

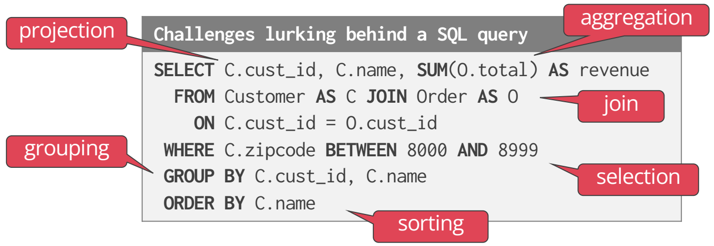
Lecture #16:

Access Methods

R&G: Chapter 14

1

Query Evaluation



DBMS query processors do not execute a query as a large monolithic block…

… but split the query into a number of specialised routines, the **query operators**

2

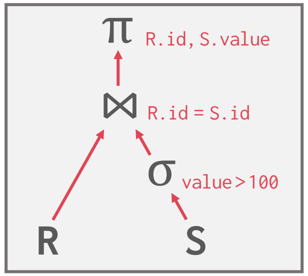
Query Plan

The operators from (extended) RA are arranged in a **tree** called **query plan**

- Edges indicate data flow (I/O of operators)
- Data flows from the leaves towards the root

The output of the root is the query result

```
SELECT R.id, S.value
  FROM R, S
 WHERE R.id = S.id
   AND S.value > 100
```



RA operators: selection (σ), projection (π), union (∪), intersection (∩), difference (–), product (×), join (⋈), renaming (ρ), assignment (R ← S), duplicate elimination (δ), aggregation (γ), sorting (τ), division (R / S)

3

Query Operators

For RA operator ⊛, a typical DBMS query engine may provide

- **different implementations** ⊛′, ⊛″, … all semantically equivalent to ⊛ with different performance characteristics

Variants (⊛′, ⊛″, …) are called **physical** operators

- implement the **logical** operator ⊛ of the relational algebra

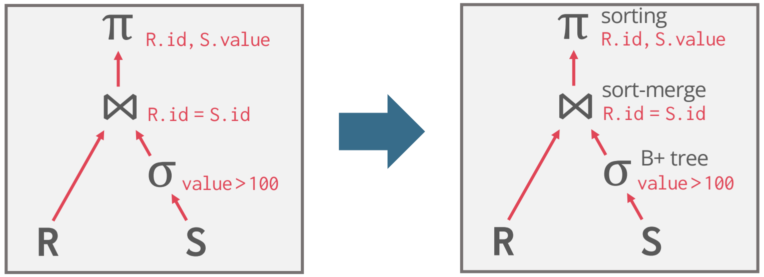
Physical operators exploit properties such as:

- presence or absence of indexes on the input file(s),
- sortedness and size of the input file(s),
- space in the buffer pool, buffer replacement policy, etc.

4

Query Plans

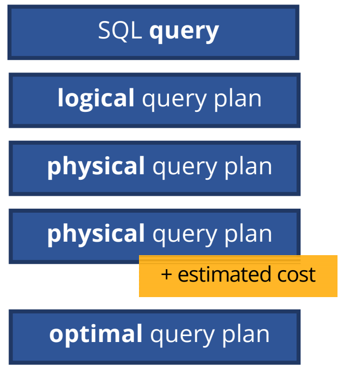
Logical Plan / **Physical Plan**



Query optimisation = choose "best" physical plan
(among many alternatives)

Query Evaluation Workflow

1. **Parse** given query
2. **Translate** query to RA
3. **Enumerate** plans by selecting physical operators and order of operators
4. **Estimate** cost of physical query plans
5. **Select** the "optimal" query plan
   - Space of possible plans far too large
   - Some type of approximation is used
   - No guarantee to find optimal query plan



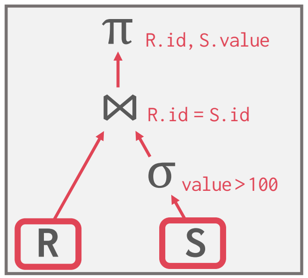
Access Methods

An **access method** (path) is a way the DBMS can access the data stored in a table
- Not defined in relational algebra
- Includes selection **predicates**

Three basic approaches:
- Sequential scan
- Index scan
- Multi-Index / "Bitmap" scan

Choice depends on #pages needed to read

```
SELECT R.id, S.value
  FROM R, S
 WHERE R.id = S.id
   AND S.value > 100
```



Sequential Scan

For each page in the table
- Retrieve it from the buffer pool
- Iterate over each tuple and check if it matches (arbitrary) predicate *p*

```
for page in table.pages:
  for t in page.tuples:
    if evalPred(p,t):
      // do something!
```

The DBMS keeps an internal **cursor** that tracks the last examined page

I/O cost = read N pages

Number of output pages = $sel(p) \cdot N$ pages
- *sel(p)* – **selectivity** of predicate *p* is the fraction of tuples satisfying predicate *p*
- The selection operator often processes tuples "on-the-fly" (no writing to disk)

Index Scan

The DBMS picks an index to find the tuples that the query needs

Which index to use depends on:

- What attributes the index contains
- What attributes the query references
- The attributes' value domains
- Predicate composition
- Whether the index has unique or non-unique keys
- Whether the index is clustered or unclustered
- …

Index Scan

Suppose that a single table has two indexes

- Tree index 1 on **age**
- Index 2 on **dept**

```
SELECT * FROM Students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'UK'
```

Scenario #1

There are 99 people under the age of 30 but only 2 people in the CS department

Scenario #2

There are 99 people in the CS department but only 2 people under the age of 30

Recap: Indexes and Selection

Basic selection: <key> <*op*> <constant>

- Equality selections (*op* is =)
- Range selections (*op* is one of <, >, <=, >=, BETWEEN)
- B+ trees provide both
- Hash indexes provide only equality

Recap: Indexes and Ordering

Can index on any ordered subset of columns. Order matters!

- Determines the selection predicates supported

In an ordered index (e.g., B+ tree), the keys are ordered **lexicographically** by the search key columns:

- Ordered by the 1st column
- 2 entries match on 1st column? Ordered by 2nd
- Match on 1st and 2nd column? Ordered by 3rd
- …

ID	Name	Age	Salary
123	Jones	31	300
443	Smith	32	400
244	Gold	55	140
134	Alvaro	55	400
221	McDonald	79	300

Ordered lexicographically by the search key (Age, Salary)

Search Key and Ordering

A tree index with a **composite search key** on columns (A_1, A_2, …, A_n) "matches" a selection predicate if:

- The predicate is a conjunction of $m \geq 0$ equality clauses of the form:
  - $A_1 = c_1$ AND $A_2 = c_2$ … AND $A_m = c_m$
- and at most 1 additional range clause of the form:
  - AND A_{m+1} *op* c_{m+1}, where *op* is one of <, >, <=, >=, BETWEEN

Why does this "match"? Lookup and scan in lexicographic order

- Can do a lookup on equality conjuncts to find start-of-range
- Can do a scan of contiguous data entries at leaves
  - Scan while A_{m+1} *op* c_{m+1} holds. If no range clause, scan all matches to the first *m* conjuncts

Search Key and Ordering

A tree index on (Age, Salary) matches which range predicates?

Legend

- Green for rows we visit that are in the range
- Red for rows we visit that are not in the range

ID	Name	Age	Salary
123	Jones	31	300
443	Smith	32	400
244	Gold	55	140
134	Alvaro	55	400
221	McDonald	79	300

Search Key and Ordering

A tree index on (Age, Salary) matches which range predicates?

- ✓ Age = 31 and Salary = 400

ID	Name	Age	Salary
123	Jones	31	300
443	Smith	32	400
244	Gold	55	140
134	Alvaro	55	400
221	McDonald	79	300

Search Key and Ordering

A tree index on (Age, Salary) matches which range predicates?

- ✓ Age = 31 and Salary = 400
- ✓ Age = 55 and Salary > 200

ID	Name	Age	Salary
123	Jones	31	300
443	Smith	32	400
244	Gold	55	140
134	Alvaro	55	400
221	McDonald	79	300

Search Key and Ordering

A tree index on (Age, Salary) matches which range predicates?

- ✓ Age = 31 and Salary = 400
- ✓ Age = 55 and Salary > 200
- ✗ Age > 31 and Salary = 400

ID	Name	Age	Salary
123	Jones	31	300
443	Smith	32	400
244	Gold	55	140
134	Alvaro	55	400
221	McDonald	79	300

✗ Not a lexicographic range.
Either visits useless rows or “bounce through” the index.

Search Key and Ordering

A tree index on (Age, Salary) matches which range predicates?

- ✓ Age = 31 and Salary = 400
- ✓ Age = 55 and Salary > 200
- ✗ Age > 31 and Salary = 400
- ✓ Age = 31

ID	Name	Age	Salary
123	Jones	31	300
443	Smith	32	400
244	Gold	55	140
134	Alvaro	55	400
221	McDonald	79	300

✗ Not a lexicographic range.
Either visits useless rows or “bounce through” the index.

Search Key and Ordering

A tree index on (Age, Salary) matches which range predicates?

- ✓ Age = 31 and Salary = 400
- ✓ Age = 55 and Salary > 200
- ✗ Age > 31 and Salary = 400
- ✓ Age = 31
- ✓ Age > 31

ID	Name	Age	Salary
123	Jones	31	300
443	Smith	32	400
244	Gold	55	140
134	Alvaro	55	400
221	McDonald	79	300

✗ Not a lexicographic range.
Either visits useless rows or “bounce through” the index.

Search Key and Ordering

A tree index on (Age, Salary) matches which range predicates?

- ✓ Age = 31 and Salary = 400
- ✓ Age = 55 and Salary > 200
- ✗ Age > 31 and Salary = 400
- ✓ Age = 31
- ✓ Age > 31
- ✗ Salary = 300

ID	Name	Age	Salary
123	Jones	31	300
443	Smith	32	400
244	Gold	55	140
134	Alvaro	55	400
221	McDonald	79	300

✗ Not a lexicographic range.
Either visits useless rows or “bounce through” the index.

Index-Only Scan

Index-only plans

Queries might be answered without retrieving any tuples from one or more of the table if a suitable index is available

Index-only scans

Retrieve only matching search keys from index pages, without reading data pages

- Often much faster than heap scans due to small index sizes

```
SELECT E.dno, COUNT(*)
  FROM Employee E
GROUP BY E.dno
```

Index on E.dno

```
SELECT E.dno, MIN(E.salary)
  FROM Employee E
GROUP BY E.dno
```

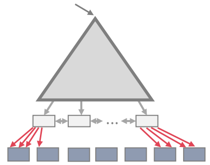
Tree index on (E.dno, E.salary)

```
SELECT AVG(E.salary)
  FROM Employee E
 WHERE E.age = 25
   AND E.salary > 300
```

Tree index on (E.age, E.salary)

Clustered B+ Tree Scan

A **clustered B+ tree** index whose search key matches the selection predicate ***p*** is clearly the superior method



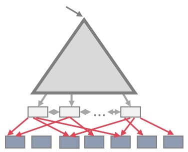
I/O cost = *2-4* + (to reach a leaf page)

sel(p) · (# of leaf pages) (to scan leaf pages)

If variant **B** or **C**, we may also need to access data records

- Requires reading *sel(p) · (# of data pages)* pages
- But if the query uses only search key attributes, then **no need to access data records**!

Unclustered B+ Tree Scan



Accessing an unclustered B+ tree index can be expensive

- I/O cost ≈ # of matching **leaf index entries**
- But index-only scans as fast as with clustered B+ trees!

If ***sel(p)*** indicates a large number of qualifying records, it pays off to

- read the matching index entries <*k, rid*>
- sort those entries on their *rid* field
- access the pages in sorted *rid* order

Lack of clustering is a minor issue if ***sel(p)*** is close to 0

Hash Index Scan

A hash index matches a selection predicate ***p*** only if:

1) ***p*** contains a term of the form ***A = c***, and
2) the hash index has been built over column ***A***

Composite search keys must be bounded entirely

- A hash index on *(age, dept)* matches ***age = 27 AND dept = 'CS'***
- But does <u>not</u> match ***age = 27***

Use index to jump to the bucket of qualifying tuples

- Scan pages in that bucket looking for matches
- If search key values are unique, terminate after finding a match
- Otherwise, scan all pages in that bucket

Multi-Index Scan

If there are multiple indexes that the DBMS can use for a query:

- Compute sets of record IDs using each matching index
- Combine these sets based on the query's predicates (union vs. intersect)
- Retrieve the records and apply any remaining terms

Set intersection can be done with bitmaps, hash tables, or Bloom filters

Postgres calls this Bitmap Scan

Multi-Index Scan

Suppose that a single table has two indexes

- Tree Index 1 on **age**
- Index 2 on **dept**

```
SELECT * FROM Students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'UK'
```

DBMS may decide to use both indexes

- Retrieve the record ids satisfying **age < 30** using Tree Index 1
- Retrieve the record ids satisfying **dept = 'CS'** using Index 2
- Take their intersection
- Retrieve records and check **country = 'UK'**