Advanced Tunning

CE384: Database Design
Maryam Ramezani
Sharif University of Technology
maryam.ramezani@sharif.edu

Query Plan

```sql
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```

Catalog

clustered (ssn), *unclustered* (ename), *unclustered* (did)

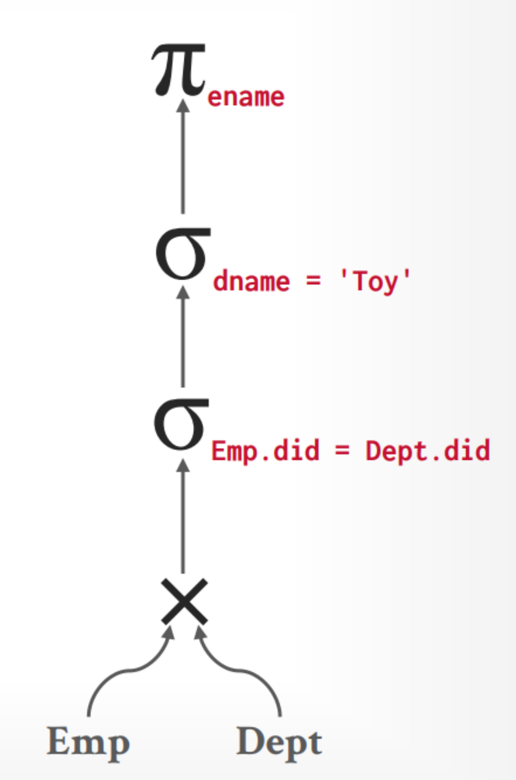
Emp(<u>ssn</u>,ename,addr,sal,did)

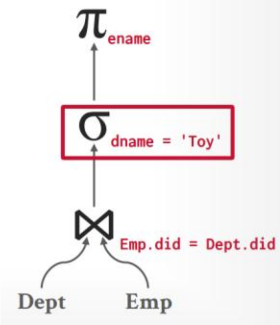
10,000 records
1,000 pages

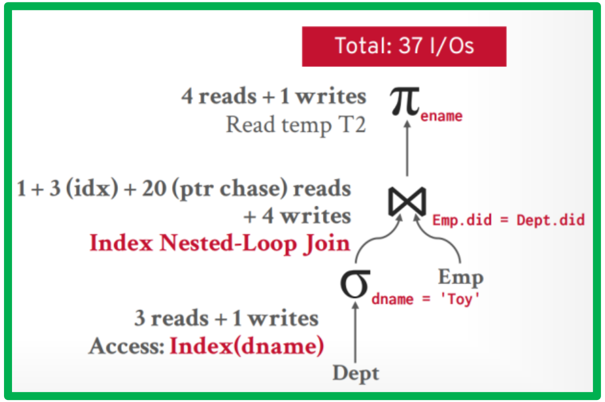
clustered (did), *unclustered* (dname)

Dept(<u>did</u>,dname,floor,mgr)

500 records
50 pages







Query Optimization

Heuristics / Rules

→ Rewrite the query to remove (guessed) inefficiencies.

→ Examples: always do selections first or push down projections as early as possible.

→ These techniques may need to examine catalog, but they do not need to examine data.

Cost-based Search

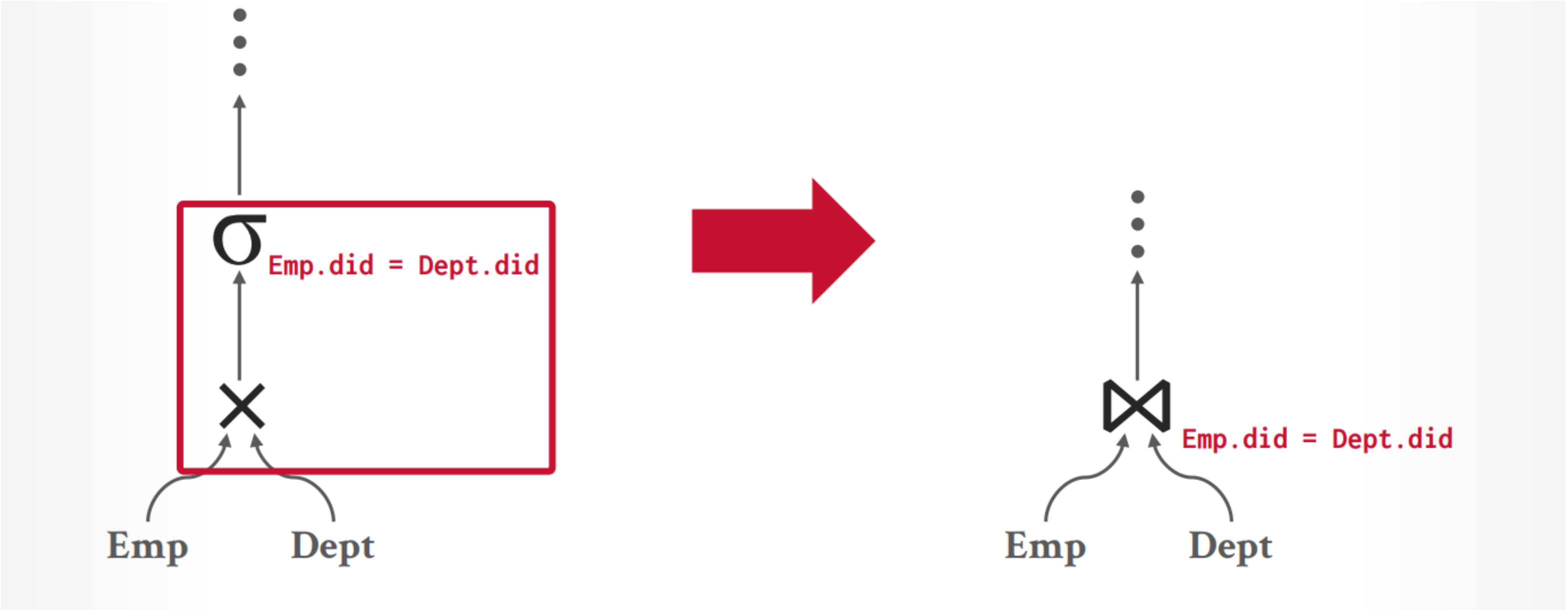
→ Use a model to estimate the cost of executing a plan.

→ Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.

PREDICATE PUSHDOWN

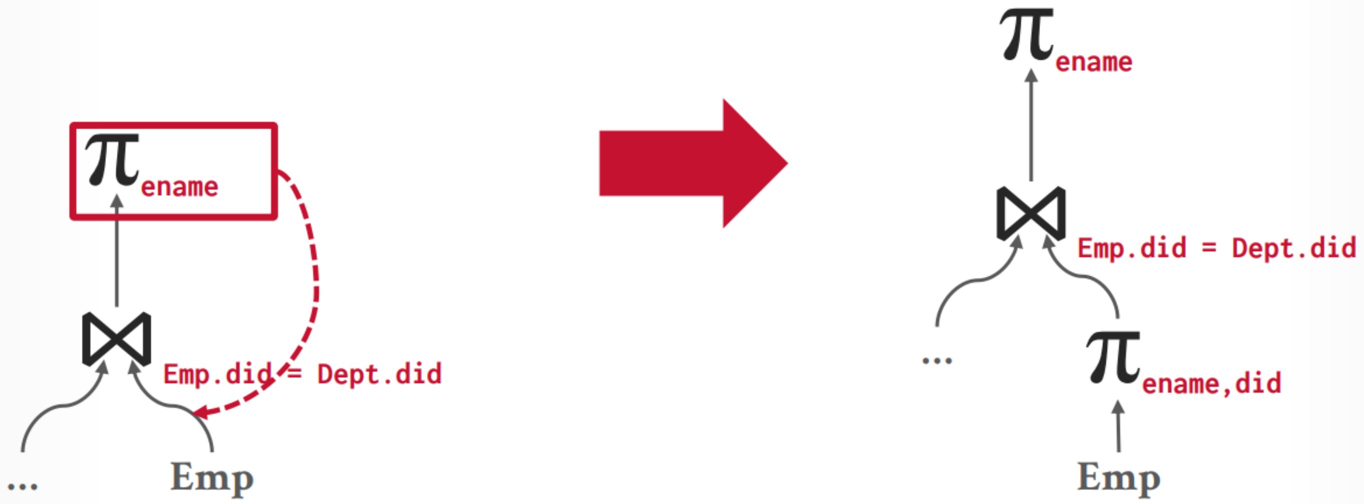
$\pi_{\text{ename}}\left(\sigma_{\text{dname} = \text{'Toy'}}(\text{Dept} \bowtie \text{Emp})\right)$ **Rewrite** $\pi_{\text{ename}}\left(\text{Emp} \bowtie \sigma_{\text{dname} = \text{'Toy'}}(\text{Dept})\right)$

REPLACE CARTESIAN PRODUCT



$\ldots(\sigma_{\text{Dept.did} = \text{Emp.did}}(\text{Dept} \times \text{Emp}))$ → $\ldots(\text{Emp} \bowtie_{\text{Emp.did} = \text{Dept.did}} \text{Dept})$

PROJECTION PUSHDOWN



$\pi_{Emp.ename}(\ldots \bowtie_{did} Emp)$ **Rewrite** $\pi_{Emp.ename}(\ldots \bowtie_{did} (\pi_{ename,\, did}\, Emp))$

Query Optimization

Heuristics / Rules

→ Rewrite the query to remove (guessed) inefficiencies.

→ Examples: always do selections first or push down projections as early as possible.

→ These techniques may need to examine catalog, but they do <u>not</u> need to examine data.

Cost-based Search

→ Use a model to estimate the cost of executing a plan.

→ Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.

SINGLE-RELATION QUERY PLANNING

Pick the best access method.

→ Sequential Scan

→ Binary Search (clustered indexes)

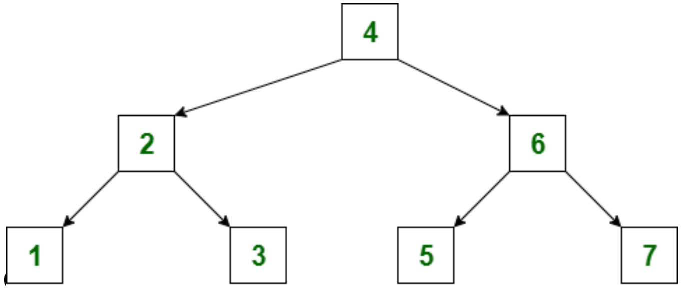
→ Index Scan

Predicate evaluation ordering.

Simple heuristics are often good enough for this.

B-Tree

- B-Tree is a type of multilevel index
- from another standpoint: it's a type of balanced tree
- Invented in 1972 by Boeing engineers R. Bayer and E. McCreight
- A B-tree can be thought of as a generalized binary search tree
  - multiple branches rather than just L or R
- Trees are always perfectly balanced
- Some wasted space in the nodes is tolerated
- The big idea: When a node is full, it splits.
- middle value is propagated upward
- If we're lucky, there's room for it in the level above
- two new nodes are at same level as original node
- Height of tree increases only when the root splits
- A very nice property
- This is what keeps the tree perfectly balanced
- Recommended: split only "on the way down"
- On deletion: two adjacent nodes recombine if both are < h
- B-Tree Insert and Delete?
  - https://www.cs.usfca.edu/~galles/visualization/BTree.html



B-Tree

B-Tree Concepts

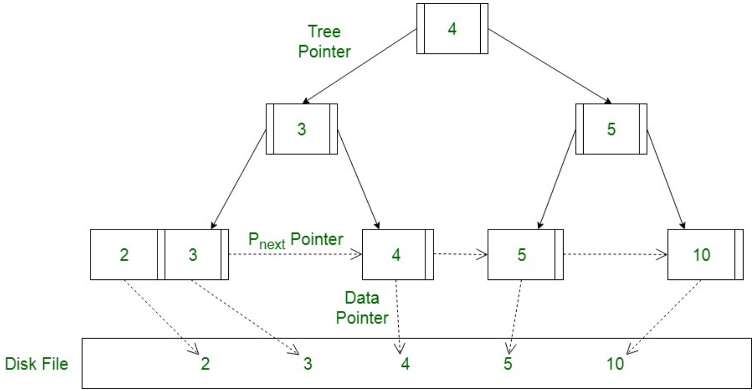
- Each node contains
  - tree (index node) pointers, and
  - key values (with record or page pointers)
- Given a key K and the two node pointers L and R around it
  - All key values pointed to by L are $< K$
  - All key values pointed to by R are $> K$

B+ Tree

- Two big differences:
  - Original B-trees had record pointers in all of the index nodes; B+ trees only in leaf nodes
    - Given a key K and the two node pointers L and R around it
      - All key values pointed to by L are < K
      - All key values pointed to by R are **>=** K
  - B+ tree data pages are linked together to form a sequential file

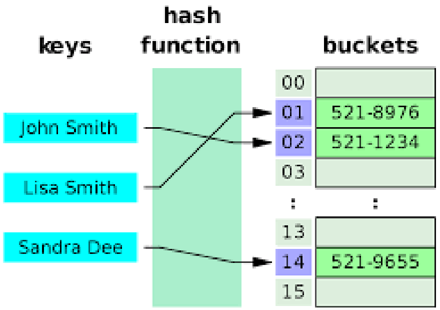
B+ Tree Index Files

- Main disadvantage of the index-sequential file organization is that performance degrades as the file grows both for index lookups and sequential scans.
- B+ tree index structure is most widely used of several index structures that maintain their efficiency despite insertion and deletion of data.



Hashing

- Can we avoid the IO operations that the result from accessing the index file?
- Hashing offers a way.
- It also provides a way of constructing indices (which need nor be sequential).



Summary

Feature	Hash	B-tree	B+-tree
Structure	Hash table buckets	Balanced tree, data in internal & leaves	Balanced tree, data only in leaves, leaves linked
Suitable for	Exact match queries	Exact & range queries	Exact & range queries
Ordering	No	Yes	Yes
Range queries	No	Yes	More efficient
Data storage	Bucket chains	Internal nodes + leaves	Leaves only
Traversal	No ordering	In-order traversal	Fast sequential via linked leaves

Index Creation in Postgres

CREATE [UNIQUE] INDEX [CONCURRENTLY] [[IF NOT EXISTS] ***name***]

ON [ONLY] ***table_name***

[USING ***method***]

btree, hash, gist, spgist, gin, and brin

({ ***column_name*** | (***expression***) }

[COLLATE ***collation***] [***opclass*** [(***opclass_parameter*** = ***value*** [, ...])]] [ASC | DESC] [NULLS { FIRST | LAST }] [, ...]

)

[INCLUDE (***column_name*** [, ...])]

[WITH (***storage_parameter*** [= ***value***] [, ...])] [TABLESPACE ***tablespace_name***]

[WHERE ***predicate***]

Logging in Postgres

- SHOW config_file;
  - logging_collector = on
  - log_directory = 'log'
  - log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
  - log_statement = 'all'
  - log_duration = on
  - log_connections = on
  - log_disconnections = on
- pg_ctl restart
  - Path in docker, ubuntu, ...: cd /var/lib/postgresql/data/log
  - Path in windows: C:\Program Files\PostgreSQL\16\data\log

Data Storage in Postgres

- Find the path of table:

```
SELECT relname, relfilenode, pg_relation_filepath(oid)
FROM pg_class
WHERE relname LIKE 'students';
```

  - Let's jump into the data and see what we can uncover! :D

Examples

SELECT *

FROM STT

WHERE STNC='0010010017';

Unique on STNC

SELECT *

FROM STCOT

WHERE COID='40384' AND GRADE > 20;

$$\sigma_{< COID=40384 \ AND \ GRADE>20 >} SCR$$

Index on COID

$$\sigma_{<GRADE>20>}(\sigma_{<COID=40384>} SCR)$$

Examples

```
SELECT  *
FROM  STCOT
WHERE  COID='40384'  OR  GRADE > 20;
```

Index on COID. What happened?

```
SELECT  *
FROM  STT
WHERE  STMJR = 'phys'
ORDER BY  BIRTH_DATE;
```

```
SELECT  count(*)
FROM  STT
WHERE  BIRTH_DATE > '1370-01-01'  AND BIRTH_DATE < '1377-01-01';
```

B Tree Index on BIRTH_DARE?

Examples

```
CREATE  TABLE  STPhones (
   STID  char(10),
   Phone  char(11),
   Primary Key (STID, Phone)
);
```

```
CREATE  TABLE  STPhones (
   STID  char(10),
   Phone  char(11),
   Primary Key (Phone, STID)
);
```

SELECT Phone from STT WHERE STID='444';

Examples

```
CREATE  TABLE  STCOT (
...,
PRIMARY KEY (STID, COID, TR, YR),
... );

SELECT   COID, TR, YR, GRADE
FROM   STCOT
WHERE   STID = 444;
```

Examples

```sql
CREATE  TABLE  STCOT (
...,
PRIMARY KEY (STID, COID, TR, YR),
... );

SELECT   COID, TR, YR, GRADE
FROM   STCOT
WHERE   STID = 444;
```

```sql
CREATE  TABLE  STCOT (
...,
PRIMARY KEY (COID, STID, TR, YR),
... );

SELECT   COID, TR, YR, GRADE
FROM   STCOT
WHERE   STID = 444;
```

```sql
CREATE  TABLE  STCOT (
...,
PRIMARY KEY (STID, YR, TR, COID),
... );

SELECT   COID, TR, YR, GRADE
FROM   STCOT
WHERE   STID = 444   AND  TR=1  AND YR = 1400;
```