Introduction to Database Systems IDBS - Spring 2024

Lecture 8 - Indexing & Query Processing

B+ Trees
Performance Tuning
Query Processing

Readings: PDBM 12.3.8, 13.1

Omar Shahbaz Khan

General Info

HOMEWORK 3 - OUT NOW!

- DEADLINE: 16. April 2024 23:59

Last Time in IDBS…

Indexing

- Indexes are data structures that facilitate access to data from disk
  - … if conditions are a prefix of indexed attributes
  - Clustered indexes store tuples that match a range condition together
  - Some queries can be answered looking only at the index (a covering index for query)
  - Indexes slow down updates and insertions

- The choice of whether to use an index is made by the DBMS for every instance of a query
  - May depend on query parameters
  - Don’t have to name indexes when writing queries

Wake Up Task!

- Normalize the following relation $R(A,B,C,D,E,F)$ into 3NF/BCNF
  - Functional Dependencies:
    - $AB \rightarrow EF$
    - $A \rightarrow C$
    - $E \rightarrow D$
  - Candidate key(s):
  - Prime Attributes:
  - Non-Prime Attributes:
  - Current Normal Form:
  - Decomposition:

Wake Up Task!

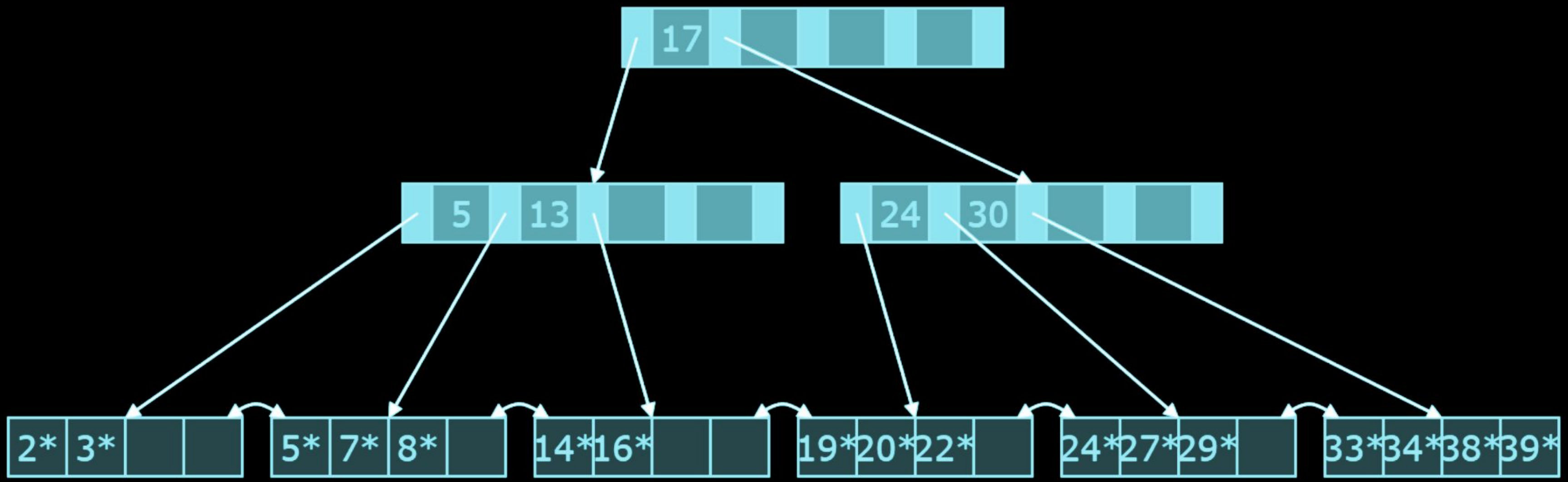
- Normalize the following relation $R(A,B,C,D,E)$ into 3NF/BCNF
  - Functional Dependencies:
    - $AB \rightarrow EF$
    - $A \rightarrow C$
    - $E \rightarrow D$

  - Candidate key(s): AB
  - Prime Attributes: A, B
  - Non-Prime Attributes: C, D, E, F
  - Current Normal Form: $1NF$
  - Decomposition: $R1(A,C)$ $R2(D,E)$ $R3(A,B,E,F)$

This Time…

-- TODO

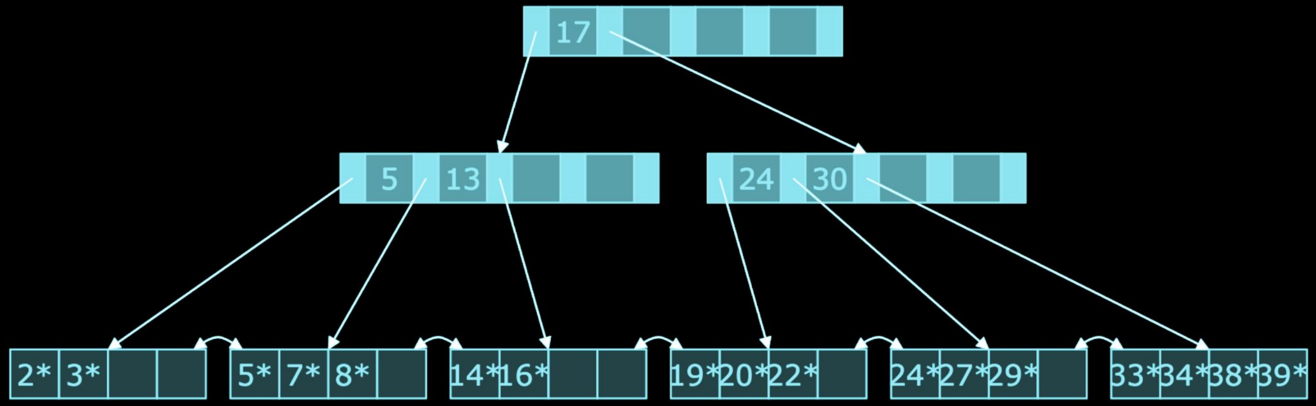
- B+ Tree
- Processing Selections
- Query Evaluation
  - Intro to Join Evaluation
  - Intro to Grouping/Aggregation (brief)


17
5
13
24
30
2*
3*
5*
7*
8*
14*
16*
19*
20*
22*
24*
27*
29*
33*
34*
38*
39*

B+ Trees

- The most common index type
  - … in relational systems
- Supports equality and range queries
- Dynamic structure
  - Adapts to insertions and deletions
  - Maintains a balanced tree

A Sample B+ Tree



- X^* represents (search key, pointer list) pairs
  (X, [address of tuple X_1, ...])
  - Unique index: Only one entry in list

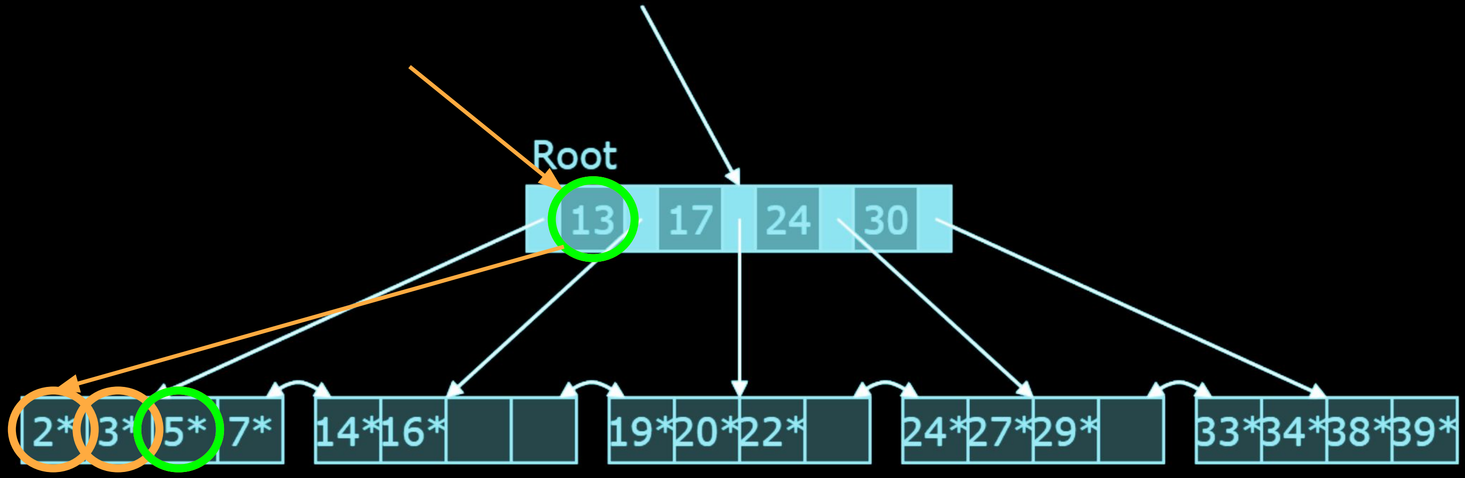
- Key values are sorted: $K_1 < \ldots < K_d$
  (d is maximum capacity or order of node)
  - For any two adjacent key values (K_i, K_{i+1}),
    the pointer P_i points to a node covering all values in the interval $[K_i, K_{i+1}]$

Each node / leaf is one disk page!

Nodes have a minimum and maximum capacity

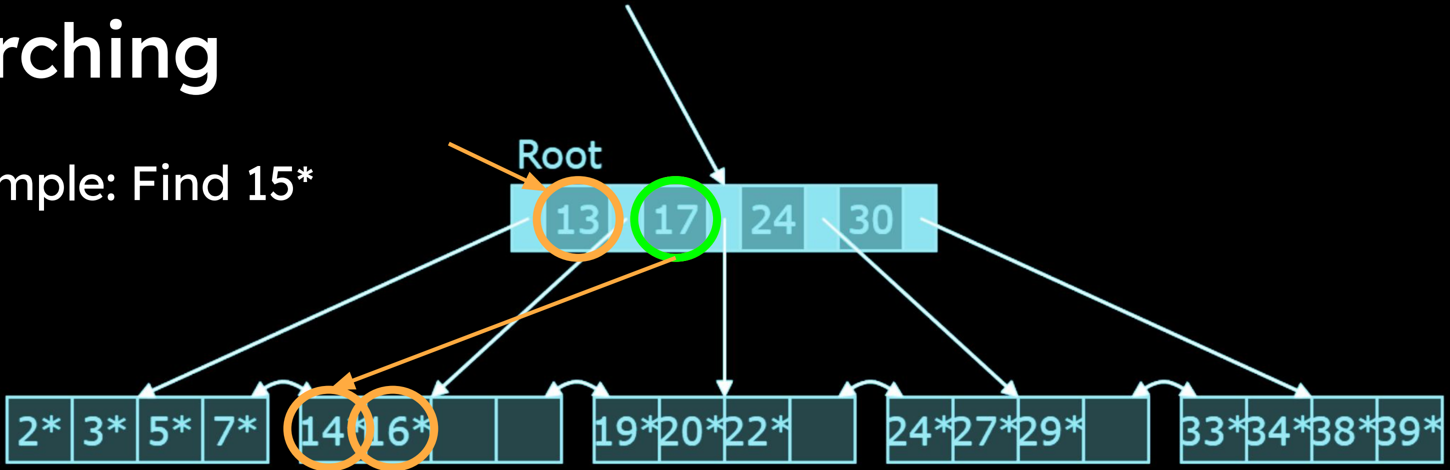
Searching

- Begin at the root
- Comparisons guide the search to the appropriate leaf
- Example: Find 5*

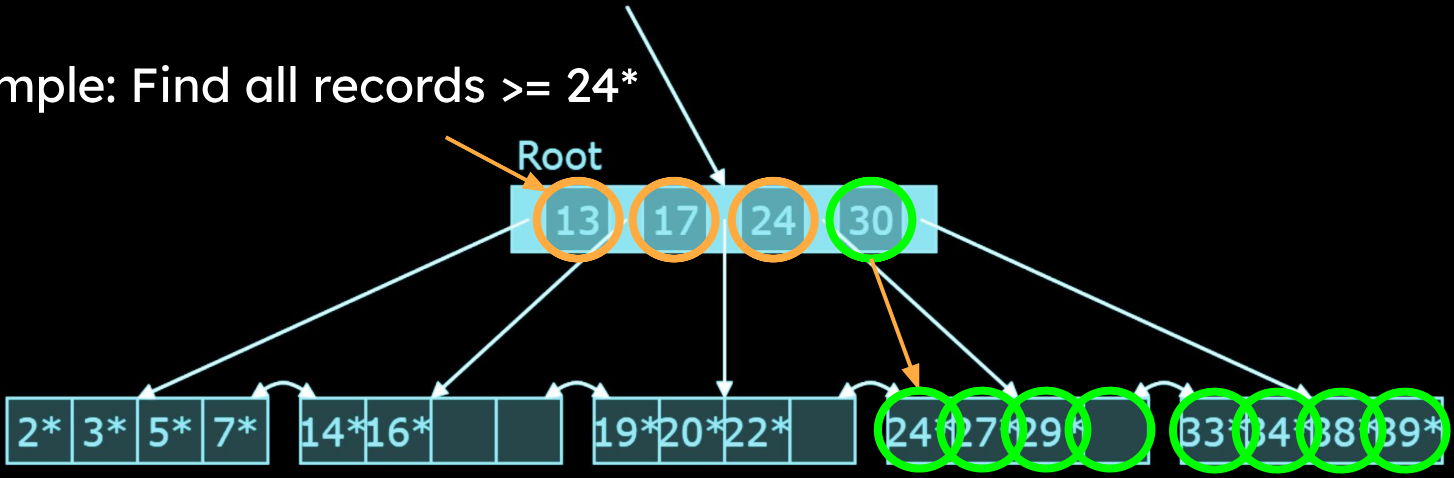


Searching

- Example: Find 15*



- Example: Find all records >= 24*

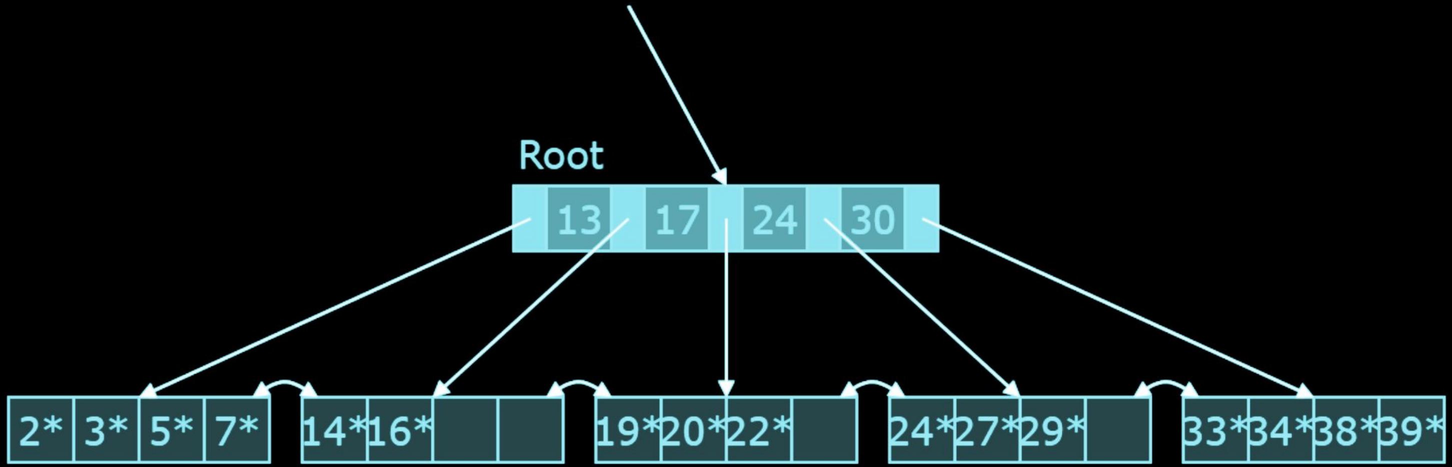


Intra-Node Searching

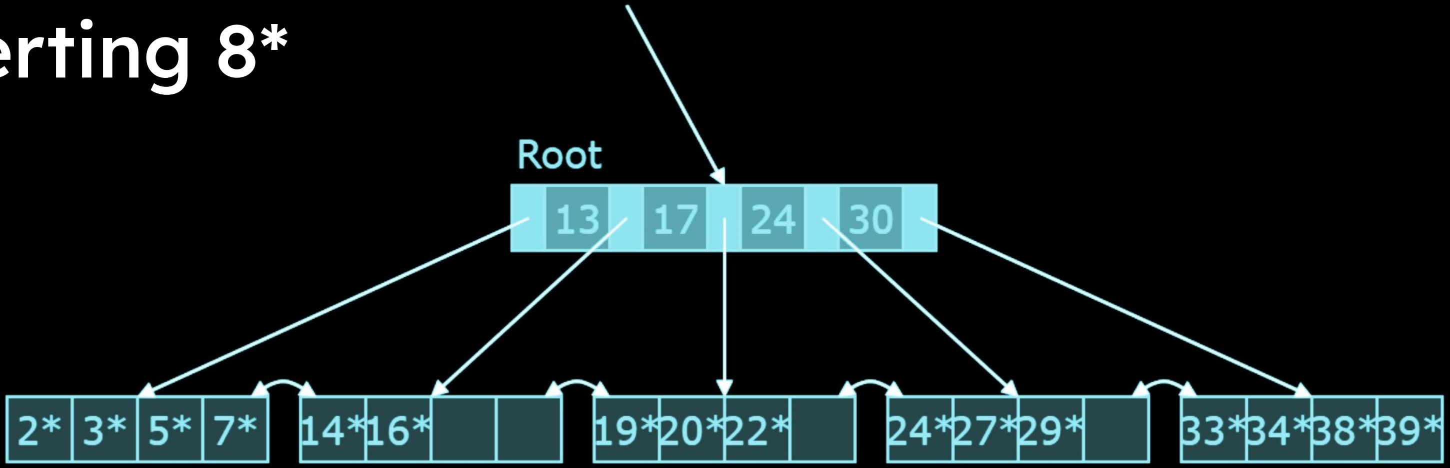
- I have used scans
- B+ tree nodes have hundreds of key values
- Use binary search!

Inserting 8*

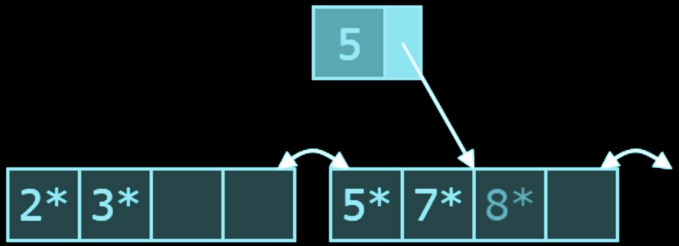
- Leaf is full, must therefore split
- Root is full, must therefore split



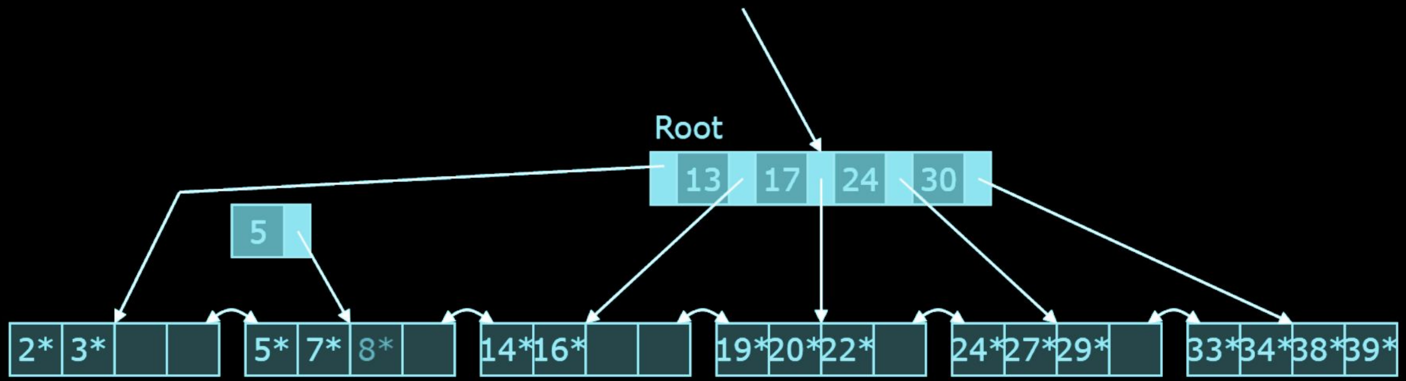
Inserting 8*



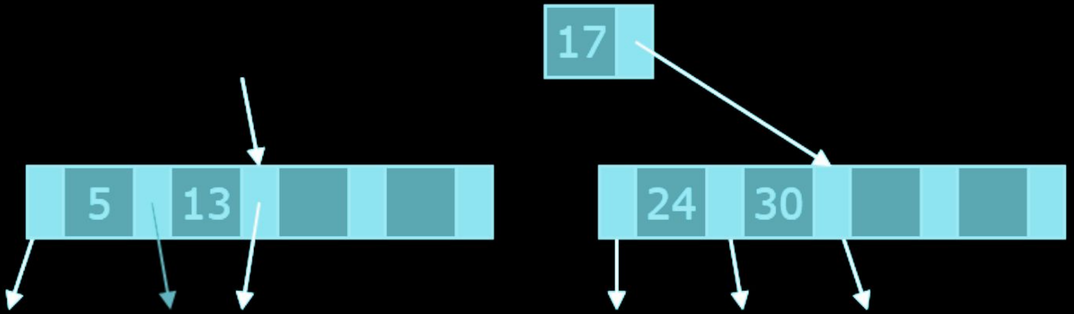
- First split the leaf
- Copy middle search key to parent



Inserting 8*

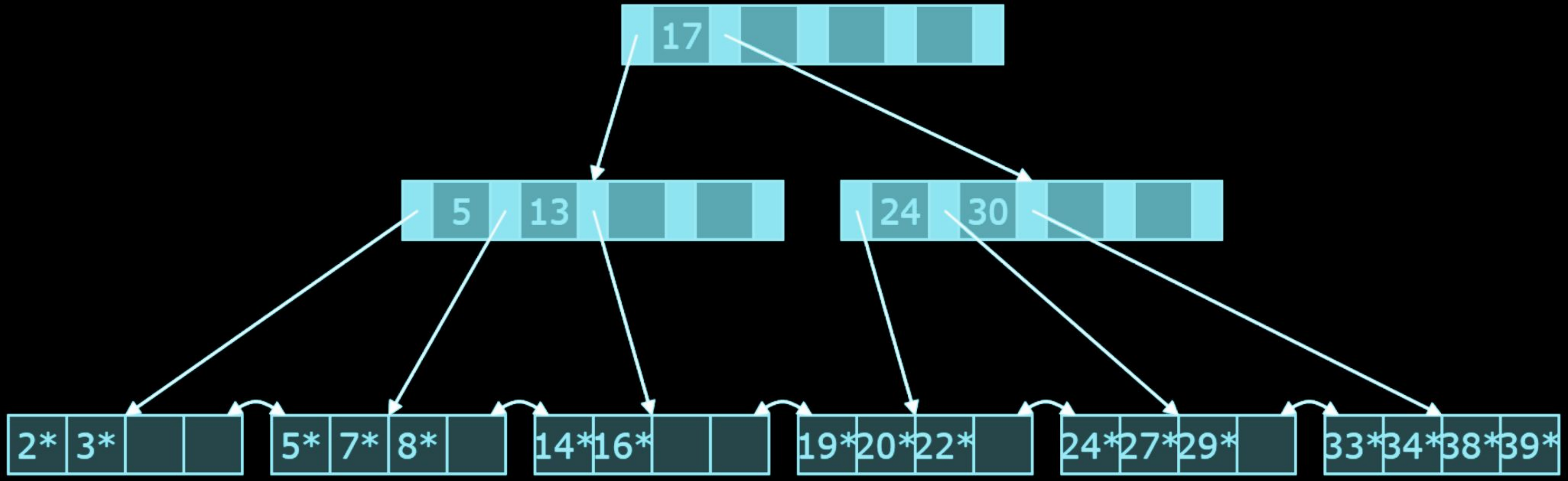


- Then split the root
- <u>Move </u>the middle search key
  Into the new root



After Inserting 8*

- Trees grow wider, then higher



Storage Capacity

- A typical tree:
  - Order: 1000 (~=16 KB per page / 16 bytes per entry)
  - Utilization: 67% (usual numbers in real life)
  - Fanout: 670

- Capacity
  - Root: 670 records
  - Two levels: 670^2 = 448,900 records
  - Three levels: 670^3 = 300,763,000 records
  - Four levels: 670^4 = 201,511,210,000 records

- Top levels may fit in memory
  - Level 1 = 1 page = 16 KB
  - Level 2 = 670 pages = 11 MB
  - Level 3 = 448,900 pages = 7 GB

Clustered Indexes in Major DBMS

- PostgreSQL
  - Cannot specify a clustered index!
  - Manual `CLUSTER` command!
- SQL Server
  - Table stored in clustered index
  - Primary keys can be unclustered
  - Indexes maintained dynamically
- DB2
  - Table stored in clustered index
  - Explicit command for index reorganization
- Oracle
  - No clustered index until 10g
  - Index organized table (unique/clustered)
  - Indexes maintained dynamically
- MySQL
  - Primary key is clustered
  - Table stored in clustered index
  - Indexes maintained dynamically

-- TODO

✓ B+ Tree

- Processing Selections

- Query Evaluation
  - Intro to Join Evaluation
  - Intro to Grouping/Aggregation (brief)

Selections

Processing Simple Selections

- Point and range queries on the attribute(s) of a clustered index are almost always best performed using an index scan
- Unclustered indexes should only be used with high selectivity queries
- Exception: Covering index is good for any selectivity
- If no index exists, a full table scan is required!
- If no “good” index exists, a full table scan is preferred!

Processing Complex Selections

- We consider the conjunction (”and”) of equality and range conditions
- No relevant index: Full table scan
- One index relevant:
  - Highly selective: Use that index
  - If not: Full table scan
- Multiple relevant indexes:
  - One is highly selective: Use that index
  - No single condition matching an index that is highly selective: Can “intersect” the returned sets

```sql
SELECT *
  FROM A
 WHERE A.year = 2024
   AND A.lecture = 8
```

Using a Highly Selective Index

- Basic idea:
  - Retrieve all matching tuples (few)
  - Filter according to remaining conditions

- If index is clustered or *covering*:
  Retrieving tuples is particularly efficient, and the index does not need to be highly selective.

Using several Less Selective Indexes

- For several conditions $C_1, C_2, \ldots$ matched by indexes:
  - Retrieve the addresses R_i of tuples matching C_i
    - The addresses are in the index leaves!
  - Compute the intersection $R = R_1 \cap R_2 \cap \ldots$
  - Retrieve the tuples in R from disk (in sorted order)

- Remaining problem:
  - How can we estimate the selectivity of a condition or of a combination of conditions?
  - Use some stats and probabilistic assumptions…

Example

Query:

```sql
SELECT title
  FROM Movie
 WHERE year = 1990
   AND studioName = 'Disney';
```

Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine two indexes to identify rows fulfilling both conditions.
5. Use one composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 and their titles in a composite covering index.

Example - Variant 1

Query:

```
SELECT title
  FROM Movie
 WHERE year = 1990
   AND studioName = 'Disney';
```

Available Indexes

```
CREATE INDEX yearIdx ON Movie(year);
CREATE INDEX studIdx ON Movie(studioName);
```

Which strategies are possible and which indexes would be used?

Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine two indexes to identify rows fulfilling both conditions.
5. Use one composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 and their titles in a composite covering index.

Example - Variant 2

Query:

```
SELECT title
  FROM Movie
 WHERE year = 1990
   AND studioName = 'Disney';
```

Available Indexes

```
CREATE INDEX yearIdx ON Movie(year);
CREATE INDEX yearStudIdx ON Movie(year,
studioName);
CREATE INDEX coveringIdx ON Movie(year,
studioName, title); --INCLUDE (title);
```

Which strategies are possible and what indexes would be used?

Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine two indexes to identify rows fulfilling both conditions.
5. Use one composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 and their titles in a composite covering index.

Example - Variant 3

Query:

```sql
SELECT title
  FROM Movie
 WHERE year = 1990
   AND studioName = 'Disney';
```

Available Indexes

```sql
CREATE INDEX idIdx ON Movie(id);
CREATE INDEX titleIdx ON Movie(title);
```

Which strategies are possible and what indexes would be used?

Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine two indexes to identify rows fulfilling both conditions.
5. Use one composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 and their titles in a composite covering index.

Processing Complex Selections Revisited

- We have considered the conjunction (”and”) of a number of equality and range conditions.

- What about disjunctive (”or”) selections?
  - One full table scan
    OR
  - Multiple “and” queries

-- TODO

✓ B+ Tree

✓ Processing Selections

- Query Evaluation
  - Intro to Join Evaluation
  - Intro to Grouping/Aggregation (brief)


JOINS
I am inevitable.
imgflip.com

Query Evaluation in a Nutshell

- SQL rewritten to (extended) relational algebra
- The building blocks in DBMS query evaluation are algorithms that implement relational algebra operations.
  - Join is the most important one!
- May be based on:
  - Reading everything / Sorting / Hashing
  - Using indexes can sometimes help!
- The DBMS optimizer knows the characteristics of each approach, and attempts to use the best one in a given setting

Join Evaluation in a Nutshell

- Join is the most important operation!

- May be based on:
  - Reading everything / Sorting / Hashing
  - Using indexes can sometimes help!

- We consider a simple join:

```
R JOIN S ON S.ID = R.ID
```

  - Extends to more complex joins in a straightforward way

Nested Loop Joins

- The following basic algorithm can be used for any join

```
for each tuple in R
    for each tuple in S
        if r.ID = s.ID
        then output (r, s)
```

- If the join condition is complex/broad, sometimes this is the only/best choice

```
R JOIN S ON S.ID <> R.ID
```

- See animation example on LearnIT

Role of Index in Nested Loop Joins

- If there is an index that matches the join condition, the following algorithm can be considered:
  - For each tuple in R use the index to locate matching tuples in S

- See animation example on LearnIT

- Good if |R| is small compared to |S|

- If many tuples match each tuple, a clustered or covering index is preferable

Example

Query:

```
SELECT *
  FROM Movie M,
  JOIN Producer P
    ON M.producer = P.id
 WHERE M.year = 2015
   AND P.birthdate < '1940-01-01';
```

Some Possible strategies:

1. Use index to find 2015 tuples, use index to find matching tuples in Producer.

2. Use index to find producers born before 1940, use index to find matching movies.

3. NL join Movie and Producer, then filter.

Problem Session

What would be good indexes for this query?

```sql
SELECT A.street, A.streetno
  FROM person P
  JOIN address A ON A.person_id = P.id
 WHERE P.lastname = 'Bohr'
   AND P.firstname LIKE 'Niels%';
```

Merge Join

- `R JOIN S ON R.ID = S.ID`

- Merge Join:
  - Step 0: Sort R and S on ID
  - Step 1: Merge the sorted R and S

- See animation example on LearnIT

- Cost:
  - If already sorted: $O(|R| + |S|)$
  - If not sorted: $O(|R|\log|R| + |S|\log|S| + |R| + |S|)$

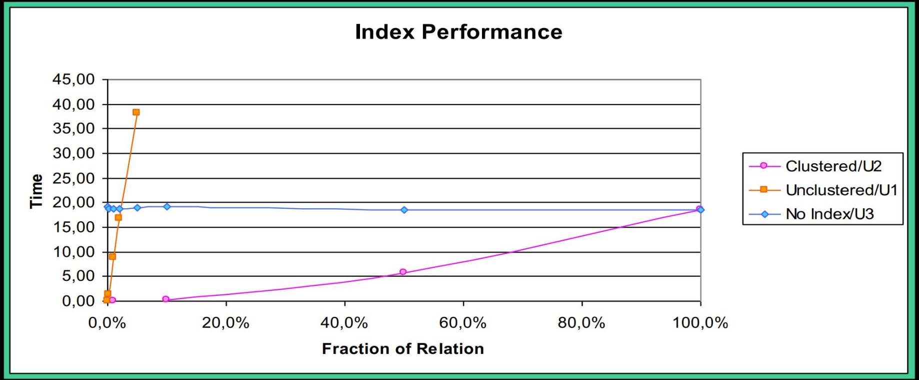
  - Can we do better?

Role of Index in Merge Joins

- Indexes can be used to read data in sorted order

- When is this a win?
  - Index is clustered
  - Index is covering

- When is this a loss?
  - Index is unclustered



Hash Join

- `R JOIN S ON R.ID = S.ID`
  - Best if S fits in RAM

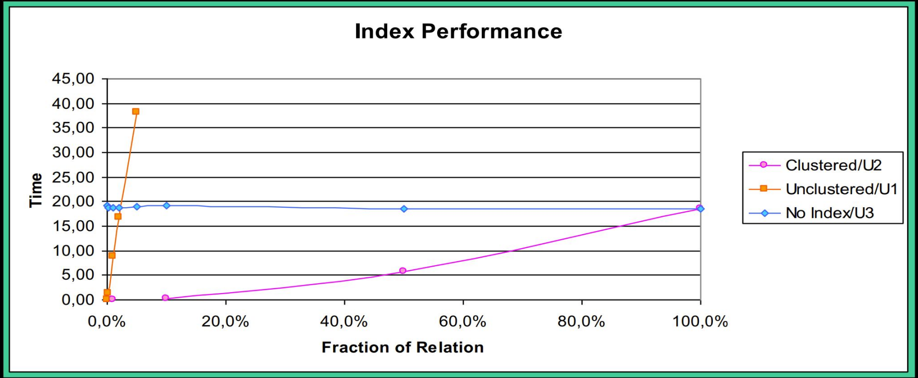
- Hash Join:
  - Step 0: Create a good hash function for ID
  - Step 1: Create a hash table for S in memory
  - Step 2: Scan R and look for matching tuples in the hash table

- See animation example on LearnIT

- Cost: O(|R| + |S|)
  - What if S does not fit in RAM?

Role of Index in Hash Joins

- Hash joins read all the relations by default
- How can indexes be useful?
  - Apply to non-join conditions - before join
  - Index is covering



Comparison of Join Algorithms

- Nested loops join:
  - Very costly $O(|R| \cdot |S|)$
  - Works for any condition ➔ sometimes only option
- Merge join:
  - Works well if data is clustered
  - Works well if relations are large and similar in size
- Hash join:
  - Works well if one relation is small
  - Is that often the case?

Grouping Operations

- Many operations are based on grouping records (of one or more relations) according to the values of some attribute(s):
  - Join (group by join attributes)
  - Group by and aggregation (obvious)
  - Set operations (group by all attributes)
  - Duplicate elimination (group by all attributes)

- Most database systems implement such grouping efficiently using sorting or hashing

-- TODO

- ✓ B+ Tree
- ✓ Processing Selections
- ✓ Query Evaluation
  - ✓ Intro to Join Evaluation
  - ✓ Intro to Grouping/Aggregation (brief)

Takeaways

My query is so slow

Have you tried creating an index?

The performance difference between well-tuned and poorly-tuned applications can be massive!

The DBMS does its best to optimize queries, but sometimes it needs help!

- Query tuning – rewrite as joins or non-correlated subqueries
- Indexes – solve 90+% of all other performance problems

If that is not sufficient…

- Materialized views / Partitioning / Denormalization
- PostgreSQL Query Planner
- Beyond the scope of this course!

Next Time in IDBS…

Introduction to Database Systems IDBS - Spring 2024

Lecture 9 - Relational Databases

Storage models
Architecture of a DBMS
Main memory DBMSs

Readings: PDBM 2

Eleni Tzirita Zacharatou